Protection des architectures hétérogènes multiprocesseurs dans les systèmes embarqués : Une approche décentralisée basée sur des pare-feux matériels

Pascal Cotret

THESE / UNIVERSITÉ DE BRETAGNE-SUD
sous le sceau de l'Université européenne de Bretagne

pour obtenir le titre de
DOCTEUR DE L'UNIVERSITÉ DE BRETAGNE-SUD
Mention : STIC
Ecole doctorale SICMA

présentée par
Pascal Cotret

Préparée à l'Unité Mixte de Recherche 6285
Lab-STICC, Laboratoire des Sciences et Techniques de l'Information, de la Communication et de la Connaissance (Lorient)

Protection des architectures hétérogènes multiprocesseurs dans les systèmes embarqués : Une approche décentralisée basée sur des pare-feux matériels

Thèse soutenue le 11 décembre 2012
devant le jury composé de :

Viktor Fischer
Professeur des universités, Laboratoire Hubert-Curien
Université Jean-Monnet, Saint-Etienne / *président*

Daniel Chillet
Maître de conférences HDR, Laboratoire IRISA (équipe CAIRN)
Université de Rennes 1, Lannion / *rapporteur*

Bruno Rouzeyre
Professeur des universités, Laboratoire LIRMM
Université de Montpellier 2 / *rapporteur*

Jean-Philippe Diguet
Directeur de recherche CNRS, Laboratoire Lab-STICC
Université de Bretagne-Sud, Lorient / *examinateur*

Guy Gogniat
Professeur des universités, Laboratoire Lab-STICC
Université de Bretagne-Sud, Lorient / *examinateur*

Protection des architectures hétérogènes multiprocesseurs dans les systèmes embarqués : une approche décentralisée basée sur des pare-feux matériels

Thèse
présentée le 11 Décembre 2012
À l'Université de Bretagne-Sud
Laboratoire Lab-STICC
École doctorale SICMA
pour l'obtention du grade de Docteur - Mention STIC
par Pascal Cotret

Université
de Bretagne-Sud

Jury :
Viktor FISCHER, *président du jury*
Professeur des universités, Université Jean-Monnet
Daniel CHILLET, *rapporteur*
Maître de conférences HDR, Université de Rennes 1
Bruno ROUZEYRE, *rapporteur*
Professeur des universités, Université de Montpellier 2
Jean-Philippe DIGUET, *examinateur*
Directeur de recherche, Université de Bretagne-Sud
Guy GOGNIAT, *directeur de thèse*
Professeur des universités, Université de Bretagne-Sud

Lorient, Université de Bretagne-Sud, 2012

Nunquam retrorsum.

Abstract

Embedded systems are used in several domains and are parts of our daily life : we use them when we use our smartphones or when we drive our modern cars embedding GPS, light/rain sensors and other electronic assistance mechanisms. These systems process sensitive data (such as credit card numbers, critical information about the host system and so on) which must be protected against external attacks as these data are transmitted through a communication link where the attacker can connect to extract sensitive information or inject malicious code within the system. Unfortunately, embedded systems contain more and more components which make more and more security breaches that can be exploited to provoke attacks.

One of the goals of this thesis is to propose a method to protect communications and memories in a multiprocessor architecture implemented in a FPGA reconfigurable chip. The method is based on the implementation of hardware mechanisms offering monitoring and cryptographic features in order to give a secured execution environment according to a given threat model. The main goal of the solution proposed in this work is to minimize perturbations in the data traffic ; it is considered that it can be accomplished by focusing on the latency impact of our security mechanisms. Our solution is also sensible to attack events : as soon as an attack is detected, an update process of security policies can be enabled.

Following an analysis of implementation results, two extensions of the basic solution are described : a fully-secured flow for startup/maintenance of FPGA-based multiprocessor systems and a method to improve attacks detection in order to take into account software parameters in multitasks applications.

Résumé

Les systèmes embarqués sont présents dans de nombreux domaines et font même partie de notre quotidien à travers les smartphones ou l'électronique embarquée dans les voitures par exemple. Ces systèmes manipulent des données sensibles (codes de carte bleue, informations techniques sur le système hôte...) qui doivent être protégées contre les attaques extérieures d'autant plus que ces données sont transmises sur un canal de communication sur lequel l'attaquant peut se greffer pour extraire des données ou injecter du code malveillant. Le fait que ces systèmes contiennent de plus en plus de composants dans une seule et même puce augmente le nombre de failles qui peuvent être exploitées pour provoquer des attaques.

Les travaux menés dans ce manuscrit s'attachent à proposer une méthode de sécurisation des communications et des mémoires dans une architecture multiprocesseur embarquée dans un composant reconfigurable FPGA par l'implantation de mécanismes matériels qui proposent des fonctions de surveillance et de cryptographie afin de protéger le système contre un modèle de menaces prédéfini tout en minimisant l'impact en latence pour éviter de perturber le trafic des données dans le système. Afin de répondre au mieux aux tentatives d'attaques, le protocole de mise à jour est également défini.

Après une analyse des résultats obtenus par différentes implémentations, deux extensions sont proposées : un flot de sécurité complet dédié à la mise en route et la maintenance d'un système multiprocesseur sur FPGA ainsi qu'une amélioration des techniques de détection afin de prendre en compte des paramètres logiciels dans les applications multi-tâches.

Remerciements

En premier lieu, je tiens à remercier Guy Gogniat, directeur du Lab-STICC et mon directeur de thèse, de m'avoir encadré pendant ces trois années. Il a toujours été de bon conseil dans les décisions à prendre et a su être un soutien important dans les phases de rédaction d'articles et de ce manuscrit. Merci encore Guy pour ces trois ans.

Je remercie également le professeur Viktor Fischer du Laboratoire Hubert-Curien à Saint-Etienne d'avoir accepté d'être le président du jury.

Je remercie Mr Daniel Chillet (Laboratoire IRISA-CAIRN à Lannion) et Mr Bruno Rouzeyre du Laboratoire LIRMM de Montpellier d'être les rapporteurs de cette thèse.

Je remercie enfin Jean-Philippe Diguet (Lab-STICC) d'avoir endossé le rôle d'examinateur.

Je remercie également les personnes du laboratoire Lab-STICC qui ont fait de ces 3 années une thèse pas totalement banale. Je pense notamment à Cédric ×2, Dominique, Florence, Ghizlane, Jean-Christophe, Romain, Sébastien, Vianney, Virginie, Willy, Youenn, Yvan. J'en oublie sûrement, désolé pour eux...

Un p'tit merci au passage (on arrête avec les anaphores) à l'équipe CAIRN du laboratoire IRISA qui m'accueille depuis septembre à Lannion en qualité d'ATER pour m'avoir permis de terminer cette thèse dans de bonnes conditions.

Et enfin, un grand grand merci à Patrick et Martine (plus connus sous les pseudonymes de Papa et Maman) de me « supporter » dans tous les sens du terme (!) depuis quelques années maintenant, même à distance.

Lorient, Décembre 2012 P. C.

Liste des publications

[Cotret *et al.* 2010] Pascal Cotret, Jérémie Crenne et Guy Gogniat. *Sécurisation des communications dans une architecture multiprocesseur.* In MajecSTIC (MAnifestation des JEunes Chercheurs en Sciences et Technologies de l'Information et de la Communication), pages 163-170, Bordeaux, France, Octobre 2010.
[Gaspar *et al.* 2010] Lubos Gaspar, Viktor Fischer, Florent Bernard, Lilian Bossuet et Pascal Cotret. *HCrypt : A Novel Concept of Crypto-processor with Secured Key Management.* In ReConFig (International Conference on ReConFigurable Computing and FPGAs), pages 280-285, Cancùn, Mexique, Décembre 2010. IEEE.
[Cotret *et al.* 2011] Pascal Cotret, Jérémie Crenne, Guy Gogniat, Jean-Philippe Diguet, Lubos Gaspar et Guillaume Duc. *Distributed security for communications and memories in a multiprocessor architecture.* In RAW (18th Reconfigurable Architectures Workshop), pages 326-329, Anchorage, AK, Etats-Unis, Mai 2011. IEEE.
[Crenne *et al.* 2011a] Jérémie Crenne, Pascal Cotret, Guy Gogniat, Russell Tessier et Jean-Philippe Diguet. *Efficient key-dependent message authentication in reconfigurable hardware.* In FPT (International Conference on Field-Programmable Technology), pages 1-6, New Delhi, Décembre 2011. IEEE.
[Cotret *et al.* 2012a] Pascal Cotret, Jérémie Crenne, Guy Gogniat et Jean-Philippe Diguet. *Bus-based MPSoC security through communication protection : A latency-efficient alternative.* In FCCM (20th Annual IEEE International Symposium on Field-Programmable Custom Computing Machines), pages 200-207, Toronto, ON, Canada, Avril 2012. IEEE.
[Cotret *et al.* 2012b] Pascal Cotret, Florian Devic, Guy Gogniat, Benoît Badrignans et Lionel Torres. *Security enhancements for FPGA-based MPSoCs : a boot-to-runtime protection flow for an embedded Linux-based system.* In ReCoSoC (7th International Workshop on Reconfigurable Communication-centric Systems-on-Chip), York, Angleterre, Juillet 2012. IEEE.
[Cotret *et al.* 2012c] Pascal Cotret, Guy Gogniat, Jean-Philippe Diguet et Jérémie Crenne. *Lightweight reconfiguration security services for AXI-based MPSoCs.* In FPL (International Conference on Field Programmable Logic and Applications), Oslo, Norvège, Août 2012. IEEE.

Table des matières

Table des figures

Liste des tableaux

Introduction

Les systèmes embarqués font désormais partie de la vie quotidienne : les smartphones, l'électronique dans les automobiles, l'électronique dédiée à la navigation marine, etc. Les systèmes embarqués sont également utilisés dans des domaines spécifiques tels que l'aérospatial ou le domaine militaire. En ce qui concerne les applications militaires, on peut citer par exemple le robot BigDog développé par Boston Dynamics [1].

Ce robot a pour tâche principale d'effectuer le transport de lourdes charges sur des opérations militaires. Ce type de robot possède différentes fonctions :

- Un GPS lui permet de se repérer.
- Il est également capable d'effectuer des communications radio pour communiquer des informations techniques et stratégiques.
- Une batterie de capteurs et de caméras permettent au robot de pouvoir analyser son environnement.

Ce type de systèmes est capable de transmettre et de recevoir des données de différentes natures. Le robot peut récupérer les valeurs de capteurs (gestion du mouvement, images des caméras...) mais manipule également des informations plus critiques telles que le code associé au système embarqué qui gère le fonctionnement du robot ou des données plus critiques qui doivent rester confidentielles (messages stratégiques à chiffrer, communications entre les différentes unités militaires...). Ces données sont éventuellement transmises par liaison et doivent être stockées dans des espaces sécurisés du système embarqué dans le robot afin que ces données ne soient pas accessibles à de potentiels attaquants.

1. http://www.bostondynamics.com/robot_bigdog.html

Ce robot est basé sur un système embarqué avec des processeurs et un système d'exploitation basé sur Unix nommé QNX [2]. Le micro-ordinateur doit être capable de récupérer les informations des différents capteurs (position GPS, caméras, pression, force) et également de commander les composants du robot (par exemple, actionner le système hydraulique qui permet au robot d'avancer ou transmettre des données par liaison radio).

Il faut donc que ces systèmes embarqués possèdent des mécanismes de sécurité qui leur permettent de pouvoir s'exécuter dans un environnement sécurisé : c'est-à-dire que, dans le cas idéal, le système sécurisé doit être capable de filtrer les attaques provenant de l'extérieur mais également de détecter une attaque qui surviendrait à l'intérieur même de l'architecture (au cas où un attaquant ait pu trouver une faille dans le système).

Les mécanismes de sécurité appliqués à ces architectures hétérogènes doivent respecter, au-delà du cahier des charges en termes de sécurité (algorithme cryptographique, modèle de menace...), des contraintes physiques liées à l'implantation sur des systèmes embarqués : contraintes de consommation, de temps de traitement, etc.

Pour le prototypage des systèmes embarqués, l'utilisation d'une technologie reconfigurable telle que les FPGA est une solution intéressante (en comparaison avec les circuits utilisant la technologie ASIC) : elle permet d'implémenter un système embarqué dans un temps de développement court tout en ayant la possibilité d'être reconfiguré afin de tester une version mise à jour du système embarqué (contrairement aux circuits ASIC dont la structure est définitive une fois qu'ils ont été produits).

Dans ce manuscrit, on s'intéresse plus particulièrement à la protection des architectures hétérogènes multiprocesseurs au niveau des communications et des mémoires : on considère qu'il s'agit de points névralgiques à prendre en compte dans le développement de systèmes embarqués sécurisés, en particulier sur des technologies FPGA. En effet, les espaces de stockage de données et les canaux de communication sont des supports pour déployer des attaques compromettant le fonctionnement du système.

2. http://www.qnx.com/

Afin d'adresser cette problématique, je propose un manuscrit organisé en 7 chapitres. Le chapitre 1 de ce manuscrit donne des éléments de contexte sur ce qu'est la sécurité et sur les différentes fonctions cryptographiques que l'on peut utiliser.

Le chapitre 2 est un état de l'art sur le sujet qui se focalise essentiellement sur la protection des communications. La protection des mémoires est également évoquée dans une seconde partie.

Les chapitres 3 et 4 décrivent en détail l'implémentation d'une solution à base de pare-feux matériels sur un composant FPGA pour protéger les communications et les mémoires selon un modèle de menaces prédéfini. Les mécanismes qui permettent de mettre à jour les contextes de sécurité en cas d'attaque sont décrits dans le chapitre 4.

Le chapitre 5 est une analyse complète des résultats obtenus par des implémentations des mécanismes de sécurité proposés dans ce manuscrit.

Le chapitre 6 propose des extensions à la solution proposée dans les chapitres 3 et 4. La première concerne l'intégration des pare-feux matériels proposés dans ce manuscrit dans un flot de protection du boot d'un système d'exploitation embarqué préexistant ; la deuxième concerne l'amélioration de la finesse de détection des pare-feux au niveau logiciel (plus précisément au niveau tâche).

Enfin, le chapitre 7 présente les conclusions ainsi que les perspectives sur l'approche proposée dans ce manuscrit.

1 Contexte

Dans ce chapitre, quelques notions sur les systèmes embarqués sont décrites. Dans une seconde partie, on explique les motivations en termes de sécurité ainsi qu'un inventaire non exhaustif des attaques existantes. Enfin, quelques éléments sur la cryptographie sont donnés dans la dernière partie du chapitre.

Sommaire

1.1 Systèmes embarqués : généralités

De nos jours, les systèmes embarqués sont utilisés dans la vie de tous les jours : électronique grand public, automobile, télécommunications - réseaux, etc. Que ce soit dans les voitures ou les smartphones, ces systèmes sont capables de réaliser des fonctions complexes et sont fabriqués dans des technologies de pointe pour respecter des cahiers des charges de plus en plus contraignants : la miniaturisation des systèmes embarqués ainsi que les contraintes en termes de performance (vitesse d'exécution, puissance consommée...) sont des paramètres qui doivent être pris en compte dans le développement de ce type de circuits.

De plus, les systèmes embarqués tentent d'implémenter de plus en plus de composants matériels dans une seule et même carte, le tout étant généralement géré par une couche logicielle composée d'un système d'exploitation ainsi que de plusieurs applications. Dans le cycle de développement des systèmes embarqués, plusieurs paramètres sont à étudier : les performances à atteindre par rapport à l'application visée, la durée de vie que l'on souhaite, etc. Depuis quelques temps, les problématiques de sécurité deviennent une étape essentielle dans le développement des systèmes embarqués ([Kocher *et al.* 2004]) : le nombre de failles augmente avec le nombre de composants et ceux-ci sont en mesure de manipuler et de transmettre des données de différentes natures qui doivent être traitées dans un environnement d'exécution sécurisé afin d'éviter les fuites de données.

Dans le domaine de la sécurité des systèmes embarqués, on peut définir différents axes d'étude pour un seul et même système. D'après [Ravi *et al.* 2004], les critères de sécurité généraux à observer sont les suivants : la sécurisation des communications, la sécurisation des espaces de stockage, la sécurisation des entrées-sorties ainsi que l'authentification des utilisateurs. Pour répondre à ces différents points, les développeurs de systèmes embarqués peuvent proposer des mécanismes ayant pour but de réaliser ces différents critères de sécurité.

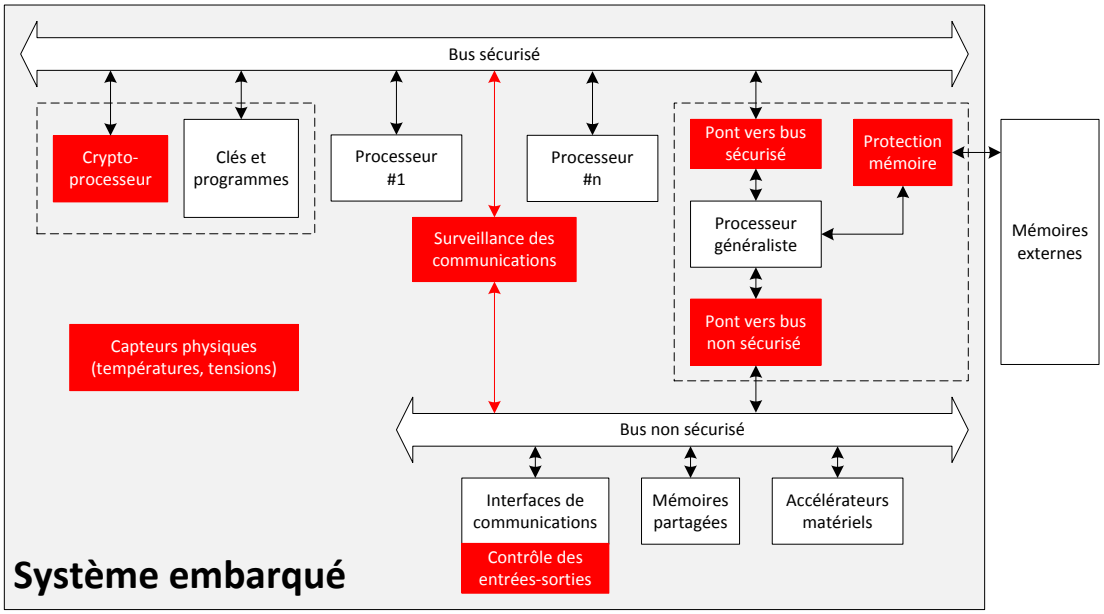


FIGURE 1.1 – Exemple de système embarqué sécurisé

Le système proposé dans la figure 1.1 est une représentation d'un système avec des mécanismes de protection qui permettent de définir des zones sécurisées et non sécurisées afin de distribuer correctement le flux de données entre les différents composants. Par exemple dans la figure 1.1, un cryptoprocesseur permet de chiffrer les communications sur le bus sécurisé et les communications avec le milieu extérieur sont contrôlées par les interfaces de communications tandis que les données stockées dans les mémoires externes sont également protégées au niveau du contrôleur mémoire. Un processeur généraliste est également présent afin de faire la transition entre l'espace sécurisé et l'espace non sécurisé.

Les travaux proposés dans cette thèse s'intéressent plus particulièrement à la protection des communications : on considère qu'il s'agit d'un aspect primordial dans le développement des systèmes embarqués car les canaux de communications voient défiler des données de différentes natures (applications, données confidentielles, éléments cryptographiques) qui n'ont pas toujours les mêmes besoins en termes de sécurité (certaines données ne doivent en aucun cas être révélées à un attaquant potentiel tandis que d'autres, comme des signaux de contrôles, ne nécessitent pas autant d'attention). Par extension, le stockage des données est également un point qui sera abordé dans ce manuscrit : les mémoires constituent un autre point névralgique de la sécurité des systèmes embarqués.

1.2 Taxonomie non exhaustive des attaques sur les systèmes embarqués

Lorsqu'un système embarqué est attaqué, on distingue généralement deux catégories : les attaques matérielles et les attaques logicielles. En ce qui concerne les attaques matérielles, l'objectif de l'attaquant est d'utiliser des propriétés physiques du circuit pour neutraliser le système ou en extraire des informations (données confidentielles sur l'utilisateur, clés cryptographiques, codes de processeur). Une des possibilités consiste à utiliser les attaques par canaux cachés [Guilley & Pacalet 2004] : par une bonne connaissance des caractéristiques physiques du système en mesurant la puissance du système par une attaque de type DPA (*Differential Power Analysis*) ou en analysant les émanations électromagnétiques par une EMA (*ElectroMagnetic Analysis*), on peut extraire des informations telle que la clé secrète d'un algorithme de chiffrement [Peeters *et al.* 2007]. L'autre catégorie concerne les attaques dites logicielles. Dans le monde informatique, cela revient aux virus, chevaux de Troie et autres vers ([Dagon *et al.* 2004] propose des exemples de ces attaques appliquées aux téléphones portables). Pour les systèmes embarqués, il va s'agir de modifier le contenu de mémoires ou d'injecter du code contenant des instructions malveillantes qui vont modifier le comportement du système.

Toutes ces attaques ont pour objectif d'altérer, d'une manière ou d'une autre, le comportement d'un système embarqué donné. Un des scenarii possible est de provoquer des dénis de services : le but est d'envoyer un grand nombre de requêtes au système cible afin de faire en sorte qu'il ne réponde plus ou de diminuer sa durée de vie ([Nash *et al.* 2005]). Il est également possible d'attaquer le système en provoquant un *buffer overflow* (scenario déroulé par exemple dans [Coburn *et al.* 2005]) : dans ce cas, le système va aller lire des données modifiées par l'attaquant afin de modifier le code associé à un processeur par exemple. Enfin, on peut citer plusieurs attaques s'intéressant aux liens entre les mémoires et le coeur du système embarqué :

- *Replay* : sauvegarder l'état de la mémoire à un instant t et réutiliser cette sauvegarde à un instant $t + \Delta t$. Dans ce cas, l'organisation de la mémoire aura toujours la même structure.
- *Spoofing* : injecter des données personnelles au niveau du lien de communication entre la mémoire externe et le système embarqué.
- *Relocation* : modifier l'organisation des différentes pages mémoires. Dans ce cas, le système qui veut aller lire les données de la page n°3 va, par exemple, récupérer les données de la page n°4 sans le savoir.

Globalement, toutes les attaques décrites dans cette partie sont des exemples de techniques qui montrent que la sécurité doit être prise en compte dans le développement de systèmes embarqués afin qu'ils ne soient pas détournés de leur fonction première.

1.3 Sécurité : généralités

Comme cela a été dit dans le paragraphe précédent, il existe différents vecteurs d'attaques et donc différentes techniques de protection pour chacun. Si on se réfère à l'architecture présentée dans la figure 1.1, on peut dire que les mécanismes de protection peuvent être représentés sous la forme de capteurs de données physiques (températures, tensions ou courant pour contrer les attaques physiques), des capteurs de données systèmes (surveillance des communications pour limiter les dénis de service ou les attaques de type *buffer overflow*) ou encore des fonctions cryptographiques (pour la protection des données dans les blocs mémoires ou le chiffrement des données par un cryptoprocesseur).

Les travaux présentés dans cette thèse s'intéressent aux deux dernières métriques. L'essentiel des travaux sur la protection des communications est décrit dans la suite du manuscrit pour un cahier des charges donné. La suite de ce chapitre s'attache à donner quelques notions sur les paramètres cryptographiques que sont la confidentialité, l'intégrité et l'authentification.

1.3.1 Confidentialité

La confidentialité consiste à remplacer un texte par un autre grâce à un mécanisme de chiffrement pour que la donnée soit illisible par un attaquant.

Dans la figure 1.2, Alice veut transmettre un message (*"Bonjour"*) à Bob. Le problème est que ce message ne doit pas être compréhensible pour Eve (qui symbolise un attaquant). Par conséquent, un mécanisme de chiffrement par *clé secrète* est implémenté afin que le message transmis sur le canal de communication soit différent. Dans la figure 1.2, Alice va chiffrer le message avec la clé avant de le transmettre à Bob qui pourra le déchiffrer avec la même clé (ceci suppose que la clé a été transmise entre les deux protagonistes par l'intermédiaire d'un lien sécurisé). Alors, l'attaquant n'est capable de récupérer qu'une version modifiée du message original. Il existe deux types de chiffrement : le chiffrement dit *symétrique* et le chiffrement *asymétrique*.

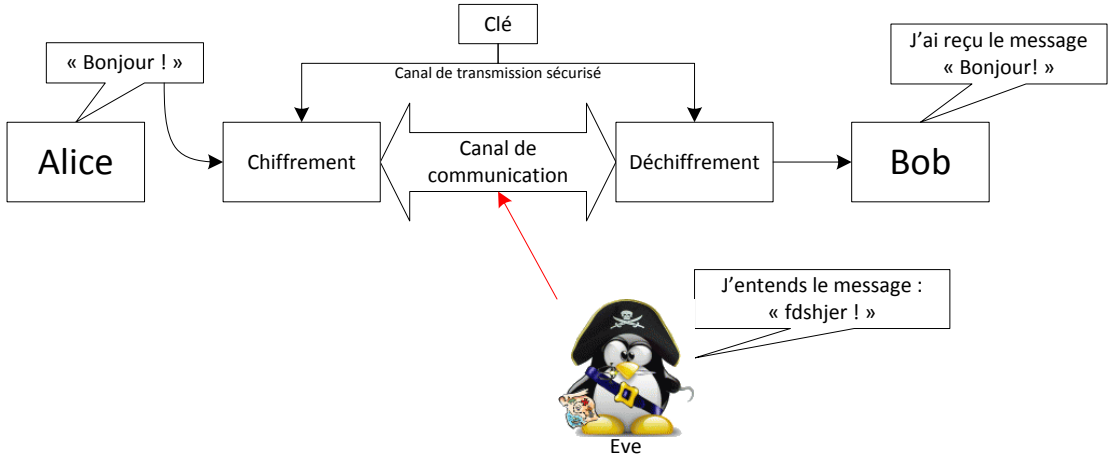


FIGURE 1.2 – Confidentialité - Mécanisme de chiffrement par clé privée

Le chiffrement symétrique correspond à celui représenté dans la figure 1.2 : il n'y a qu'une seule clé (dite *secrète*) partagée entre l'expéditeur et le destinataire du message. Un exemple d'algorithme de chiffrement symétrique est l'AES (*Advanced Encryption Standard*) : cet algorithme a été formalisé en 2000 suite à un concours lancé par le NIST [1] et a été approuvé par la NSA [2]. Cet algorithme existe sous différents modes de fonctionnements qui ont chacun leurs avantages et inconvénients en termes de performances (latence, quantité de données nécessaires en amont du chiffrement) et en termes de sécurité ; [Crenne 2011] propose une description détaillée de ces différents modes. L'autre type de chiffrement est dit *asymétrique*. La principale différence réside dans le fait que celui-ci repose sur un fonctionnement avec deux clés (une clé *publique* et une clé *privée*) : sur la base du mécanisme proposé dans la figure 1.2, Alice va chiffrer le message avec la clé publique et Bob le déchiffre avec sa clé privée. Même si la clé publique est diffusée (contrairement à la clé secrète), l'attaquant ne peut pas déchiffrer le message. Un exemple d'algorithme de chiffrement asymétrique est l'algorithme RSA (Rivest, Shamir, Adleman du nom des trois auteurs de ce standard) décrit dans les années 1970 [3].

1. National Institute of Standards and Technologies
2. National Security Agency
3. http://worldwide.espacenet.com/publicationDetails/biblio?CC=US&NR=4405829&KC=&FT=E&locale=en_EP

1.3.2 Intégrité

L'intégrité consiste à vérifier que le message n'a pas été altéré pendant la transmission.

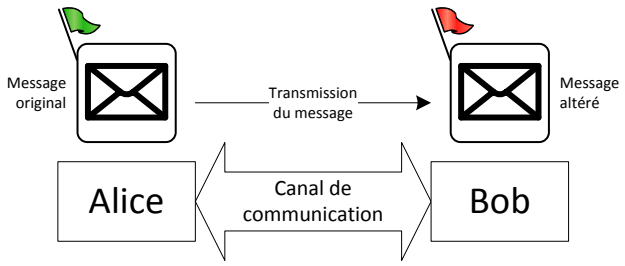


FIGURE 1.3 – Mécanisme d'intégrité

Dans la figure 1.3, le message qu'envoie Alice à Bob a été modifié soit par une perturbation du canal de communication, soit par une injection d'une donnée contrefaite. Pour vérifier l'intégrité, on peut associer à la donnée une valeur témoin qui permet de détecter si la donnée a été modifiée (dès que la donnée change, la valeur témoin change).

L'intégrité peut être réalisée par l'implémentation d'une fonction de hachage : ces fonctions produisent des données de taille fixe à partir de données arbitraires de telle sorte qu'une modification de la donnée en entrée de la fonction de hachage change la valeur de sortie (qu'on appelle *hash*). Parmi les fonctions de hachage les plus connues, on peut citer MD5 (*Message Digest 5*, inventé par Rivest en 1991) ou la famille des SHA (*Secure Hash Algorithm*, dont la troisième version devrait sortir d'ici la fin de l'année 2012 par un concours organisé par la NIST [4]. D'autres structures plus complexes, telles que les arbres de Merkle [Merkle 1987], existent : la donnée est divisée en sous-blocs de taille égale et chacun à sa propre valeur de hachage, ces valeurs sont ensuite compressées jusqu'à obtenir un hash global de la donnée. On peut l'utiliser pour vérifier l'intégrité d'un message sans en connaître la totalité (il suffit de connaître le hash d'une partie de la donnée que l'on souhaite vérifier).

4. http://csrc.nist.gov/groups/ST/hash/sha-3/index.html

1.3.3 Authentification

L'authentification consiste à vérifier que l'initiateur de la requête est bien celui attendu.

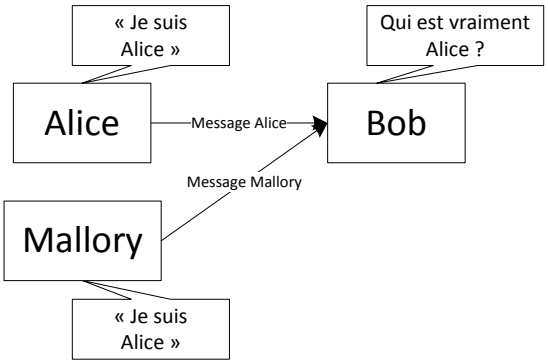


FIGURE 1.4 – Mécanisme d'authentification

Dans la figure 1.4, la situation est telle que le destinataire de la communication (Bob dans ce cas) reçoit deux informations provenant de sources différentes (Alice et Mallory) mais se proclamant tous les deux comme étant le seul et unique initiateur de la transaction (tous les deux disent être Alice). Par conséquent, il faut un mécanisme qui permet d'authentifier l'initiateur de la transaction (un certificat qui permet de dire quels sont les initiateurs autorisés). On trouve ce type de mécanisme dans le protocole SSL (*Secure Socket Layer*, dont une implémentation libre existe [5]).

5. http://www.openssl.org/

1.3.4 Combinaisons alternatives

Plutôt que d'utiliser deux fonctions séparément pour réaliser deux des propriétés énoncées ci-dessus (confidentialité, intégrité et authentification), il existe des fonctions qui combinent plusieurs propriétés en un seul module.

– MAC (*Message Authentication Code*[6]), permet de vérifier l'intégrité et l'authentification d'un message.
– CBC-MAC est un dérivé du chiffrement DES qui permet aussi de vérifier l'authentification d'un message[7].
– AES-GCM [McGrew & Viega 2004, Crenne *et al.* 2011b] est une fonction qui permet de réaliser la confidentialité et l'authentification (éventuellement l'intégrité) en utilisant le chiffrement AES (des détails sur cet algorithme sont donnés ultérieurement dans le manuscrit étant donné qu'il s'agit de l'algorithme choisi dans ces travaux).

1.4 Conclusion

Ce chapitre présente succinctement les propriétés qui seront utilisées dans la suite de ce manuscrit. Concrètement, lorsqu'on évoque la sécurité des systèmes embarqués, on s'intéresse donc à la protection des communications dans les systèmes multiprocesseurs en utilisant des mécanismes qui font office de capteurs du trafic de données ainsi qu'à la protection des mémoires par l'utilisation de ces capteurs combinée à l'utilisation de propriétés cryptographiques que sont la confidentialité et l'intégrité. Dans la suite de ce manuscrit, on étudiera les performances et les contraintes d'une implémentation de ces propriétés sur un système multiprocesseur embarqué sur un composant reconfigurable de type FPGA.

6. http://en.wikipedia.org/wiki/Message_authentication_code
7. http://www.itl.nist.gov/fipspubs/fip113.htm

2 État de l'art

Ce chapitre a pour but de faire une présentation des travaux existants dans le domaine ainsi que de positionner les travaux présentés dans cette thèse et d'en décrire les principales caractéristiques.

Sommaire

En amont à l'état de l'art, il faut bien prendre conscience que les systèmes embarqués sont de plus en plus complexes. En effet, les systèmes embarqués (qu'ils soient basés sur des technologies reconfigurables telles que les FPGAs ou pas) contiennent de plus en plus de composants au sein d'une seule et même puce.

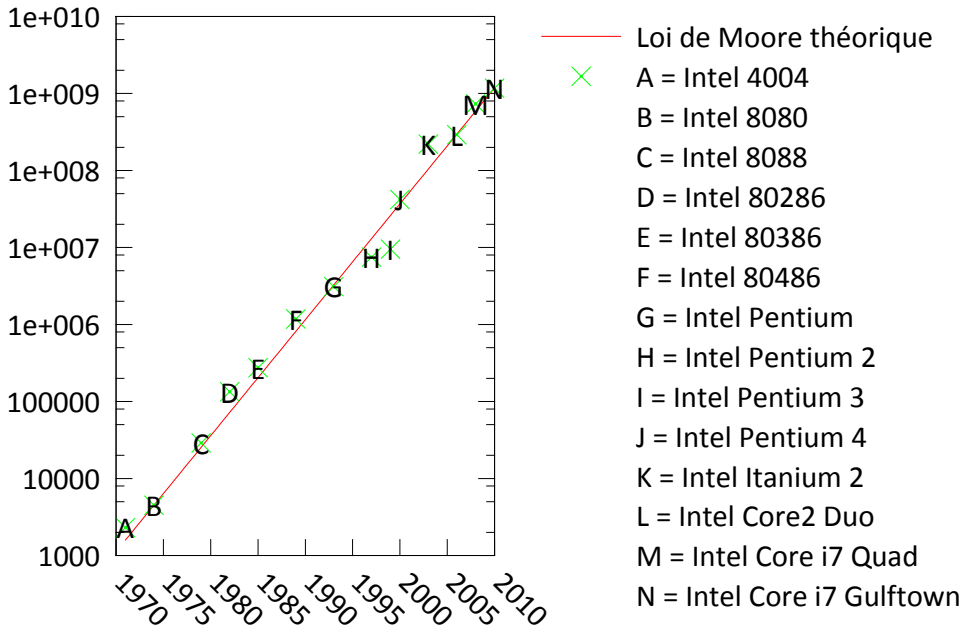


FIGURE 2.1 – Loi de Moore - Nombre de transistors pour divers modèles de processeur

D'après la loi de Moore [1] (voir figure 2.1), le nombre de transistors dans les processeurs des ordinateurs double tous les 18 mois : elle confirme le fait qu'avec l'avancement des hautes technologies, les processeurs (et par extension les systèmes embarqués) ont tendance à contenir de plus en plus de composants dans une seule et même puce. Malgré tout, cette hypothèse n'est pas valable à long terme : la finesse de gravure ne pourra pas diminuer indéfiniment et les constructeurs s'intéressent déjà aux puces multi-coeurs et aux transistors en trois dimensions (par exemple, dans les processeurs Ivy Bridge d'Intel [2]).

1. Les données sont issues de la page http://fr.wikipedia.org/wiki/Microprocesseur
2. http://www.bbc.co.uk/news/technology-17785464

Dans le fonctionnement des systèmes embarqués, il y a différentes phases : des temps de calcul (par exemple, une compression d'images ou un traitement du signal par un processeur), des transmissions de données entre les composants ou encore des lectures-écritures dans des mémoires. De notre point de vue, l'aspect le plus important est au niveau des communications au sein du système embarqué : les données qui circulent sur le moyen de communication du système sont de différentes natures (données, codes processeur ou informations confidentielles sur l'utilisateur) et n'ont pas non plus les mêmes besoins en termes de sécurité. Par extension, on s'intéressera également à la protection des mémoires : il s'agit d'une brèche qui peut être exploitée pour perturber les communications.

La suite de ce chapitre présente tout d'abord les architectures de communications qui sont susceptibles de nous intéresser. Ensuite, un état de l'art sur les mécanismes de protection existants permet de montrer le positionnement choisi pour les travaux présentés dans cette thèse par rapport aux travaux existants dans le domaine.

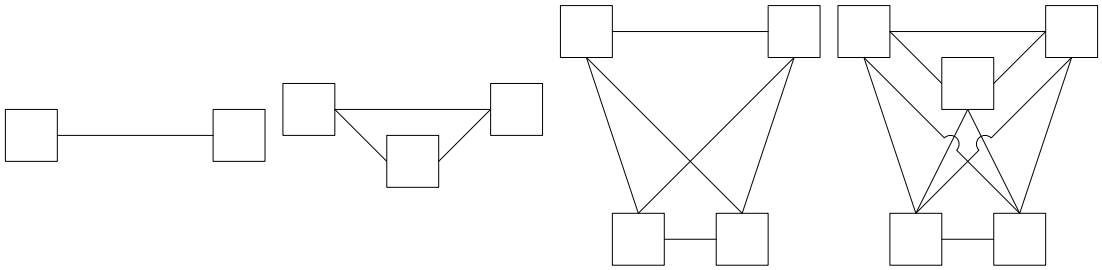
2.1 Communications dans les architectures multiprocesseurs

2.1.1 Bus : topologies et protocoles

Dans les systèmes embarqués, il faut que les composants de l'architecture puissent communiquer entre eux. Il existe plusieurs manières d'implémenter un réseau de communication entre les éléments du système multiprocesseur. La solution la plus simple consiste à implémenter des connexions point-à-point entre les composants.

Cette approche facilite le protocole de communication : étant donné que les liens sont individuels, il n'y a pas besoin de mécanisme d'arbitrage et la latence de communication est minimale. Par contre, son inconvénient majeur se situe au niveau architectural : plus il y a de composants dans le système à implémenter, plus il faut de liens de communication pour les relier entre eux. On propose de déterminer la « taille » de l'architecture (autrement dit, le nombre de liens nécessaire en fonction du nombre de composants).

Soit N le nombre de composants et L le nombre de liens nécessaires. On suppose également que les liens de communication sont bidirectionnels (*half-duplex* ou *full-duplex*, la communication peut avoir lieu dans les deux sens mais pas forcément simultanément). Enfin, on suppose que les mécanismes de sécurité ne sont pas mutualisés (1 mécanisme par lien). Les figures 2.2a à 2.2d illustrent les architectures pour les 4 premiers cas (jusqu'à $N = 5$).



(a) N=2, L=1 (b) N=3, L=3 (c) N=4, L=6 (d) N=5, L=10

FIGURE 2.2 – Nombre de liens dans une architecture avec liaisons point-à-point

On remarque également que :

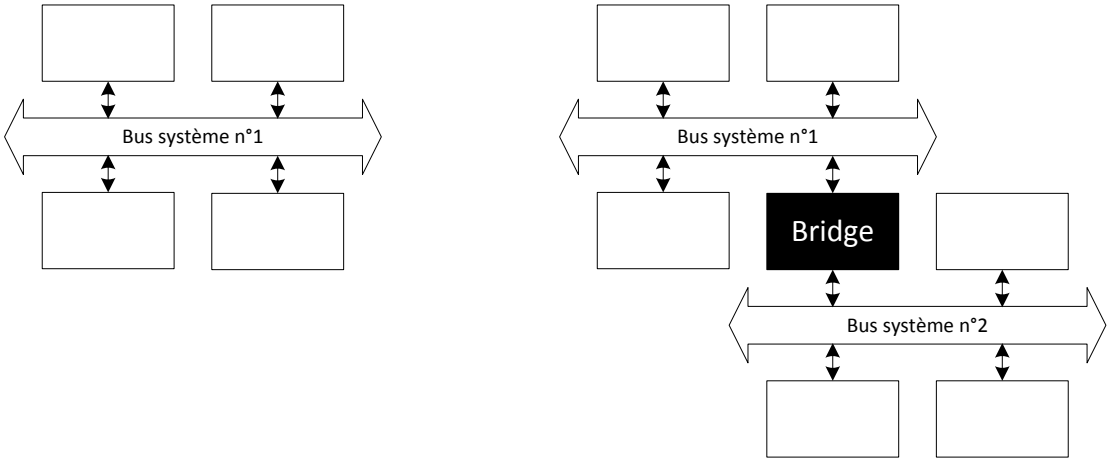
$$\begin{aligned} &\text{Pour N=2,} \quad L = N \times \tfrac{1}{2} \\ &\text{Pour N=3,} \quad L = N \\ &\text{Pour N=4,} \quad L = N \times \tfrac{3}{2} \\ &\text{Pour N=5,} \quad L = N \times 2 \end{aligned} \tag{2.1}$$

On peut en déduire que l'évolution du nombre de liens L en fonction du nombre d'éléments N suit l'équation :

$$L = \binom{N}{2} = \frac{N!}{2! \times (N-2)!} = N \times \frac{N-1}{2} \tag{2.2}$$

Cette fonction est représentée dans la figure 2.4 : l'évolution du nombre de liens suit une fonction en parabole, le nombre de liens augmente rapidement avec le nombre d'éléments dans l'architecture. D'un point de vue sécurité, la tendance est la même tant que les mécanismes de sécurité restent individuels (s'ils étaient mutualisés, on perdrait de l'intérêt de la liaison point-à-point).

Dans un second temps, il existe des approches où le lien de communication est partagé entre les différents composants du système embarqué (voir figure 2.3a). Dans ce cas, la gestion des communications est réalisée par un arbitre [Poletti *et al.* 2003] qui est capable de choisir dans quel ordre les transactions doivent se dérouler en fonction d'une politique définie par le protocole choisi ou par l'utilisateur (par exemple : round-robin, priorité fixe...). Dans ces bus, il n'y a qu'une seule transaction à la fois. D'un point de vue architectural, les bus ont des limites quant au nombre de composants admis par chaque médium de communication (c'est-à-dire, le maximum qu'un arbitre est capable de gérer en même temps) qui dépendent du protocole choisi (PLB, AXI ...), on considère généralement que ces limites sont de l'ordre de 32 éléments par bus (d'après [Xilinx 2011a], on peut connecter jusqu'à 16 maîtres à 16 esclaves sur un bus AXI).



(a) Bus simple (b) Bus hiérarchique

FIGURE 2.3 – Topologies de bus

Comme précédemment, on propose de calculer le nombre de liens L nécessaires pour connecter N éléments grâce à des bus AXI. On suppose qu'on travaille sur un bus simple : d'après [Xilinx 2011a], le nombre d'éléments maximum sur un lien est de 32. On a donc :

$$L = 1, \forall N \leq 32 \tag{2.3}$$

La figure 2.4 présente l'évolution du nombre de liens en fonction du nombre d'éléments pour les liaisons point-à-point et bus AXI simple.

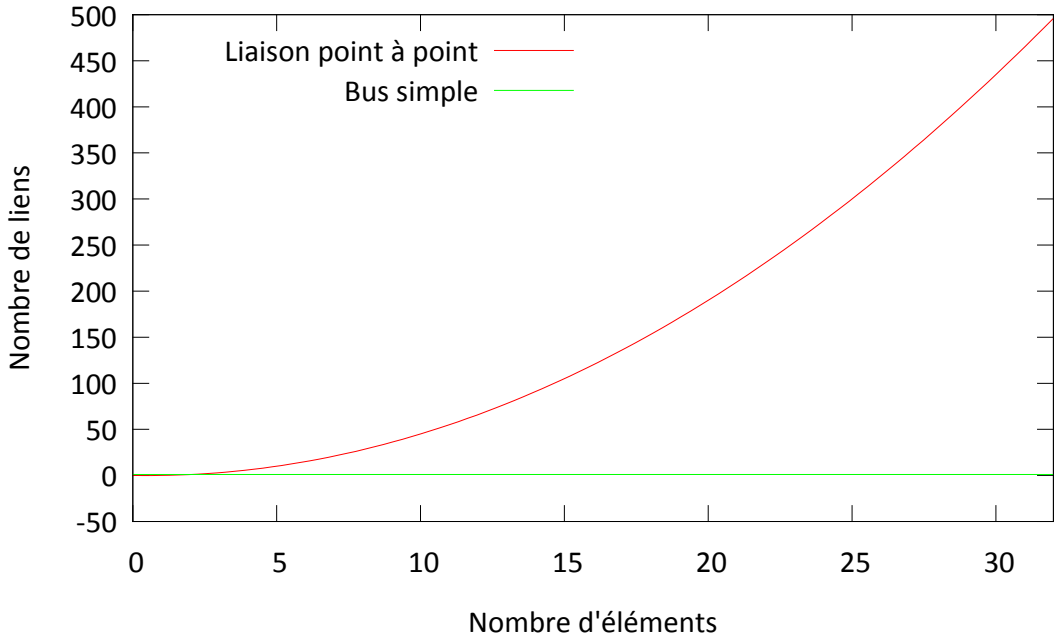
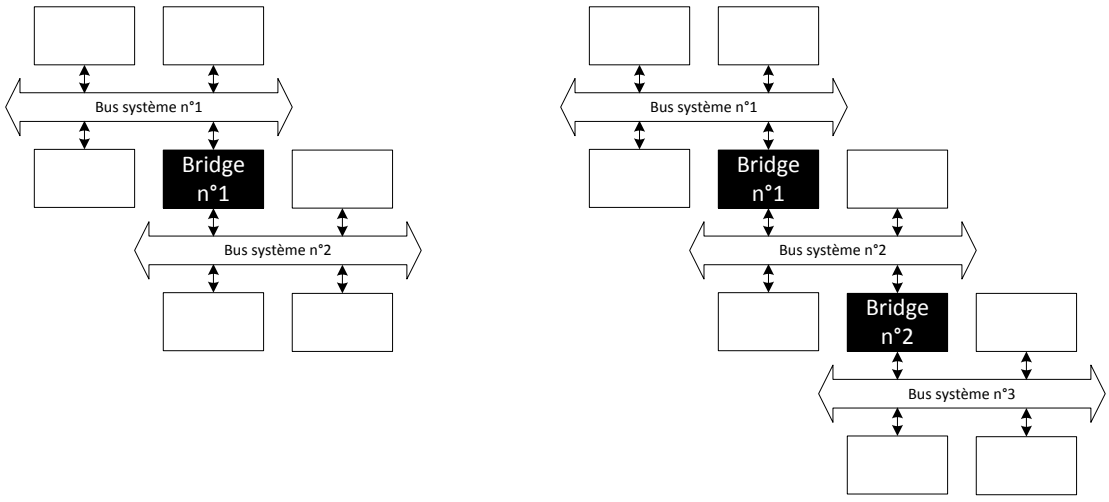


FIGURE 2.4 – Évolution du nombre de liens en fonction du nombre d'éléments

En termes de nombre de liens (par extension, en termes de surface), la solution par bus simple est beaucoup plus avantageuse. Pour des systèmes où le bus simple ne suffit plus, on peut utiliser des bus hiérarchiques qui sont une succession de bus simples interconnectés par des bridges (voir figure 2.5).

Le raisonnement pour connaître le nombre d'éléments maximum N sur un bus hiérarchique à L niveaux (c'est-à-dire à L bus simples) est le suivant. D'après la figure 2.5a, le bridge compte comme 1 élément sur chaque bus simple. Pour la structure à 3 niveaux (figure 2.5b), le bus n°2 possède 2 bridges. Par conséquent :

$$\begin{aligned} N = 31 + 31 + 1 = 63 \quad &\text{éléments dont 1 bridge pour} \quad L = 2 \\ N = 31 + 30 + 31 + 2 = 94 \quad &\text{éléments dont 2 bridges pour} \quad L = 3 \end{aligned} \tag{2.4}$$



(a) 2 niveaux (b) 3 niveaux

FIGURE 2.5 – Bus hiérarchiques

Dans tous les cas, le coût en surface est moindre par rapport à une solution point-à-point (voir figure 2.4) étant donné que les ressources pour transmettre les données sont mutualisées. Par contre, on augmente la pénalité en termes de latence car les transactions sur ce type de système doivent généralement être supervisées par un arbitre de bus qui décide la priorité à donner à telle ou telle transaction. Pour les technologies reconfigurables telles que les FPGAs, certains fabricants ont leur standard (Avalon pour Altera...), d'autres utilisent un protocole open-source (Wishbone[3]), d'autres exemples de protocoles sont donnés par [Mitic & Stojcev 2006] qui propose une description et un comparatif des caractéristiques de chaque standard.

Chacun a ses avantages et ses inconvénients mais de part l'utilisation d'une plateforme Xilinx pour les implémentations ultérieures, le protocole de bus utilisé sera celui proposé par le fondeur Xilinx. En réalité, il en existe deux : PLB, un standard en phase de devenir obsolète implémenté dans les outils de développement du fabricant de FPGA Xilinx (et fourni par IBM).

3. Projet géré par Opencores : http://opencores.org/opencores,wishbone

Ce protocole dispose de deux bus :
– PLB (Processor Local Bus [4]), pour les transactions à haut-débit (processeurs, mémoires).
– OPB (On-chip Peripheral Bus), pour les transactions plus lentes (entrées, sorties).

AXI, la version actuelle sur la base d'un standard de ARM (c'est celui qu'on choisira dans la suite de ce manuscrit) se base également sur deux bus :
– AXI ([Xilinx 2011a, ARM 2012]), analogue à PLB.
– AXI-Lite, analogue à OPB.

2.1.2 NoC

Pour des architectures manycores (contenant de nombreux éléments), on peut faire appel à des techniques s'inspirant des réseaux informatiques [5].

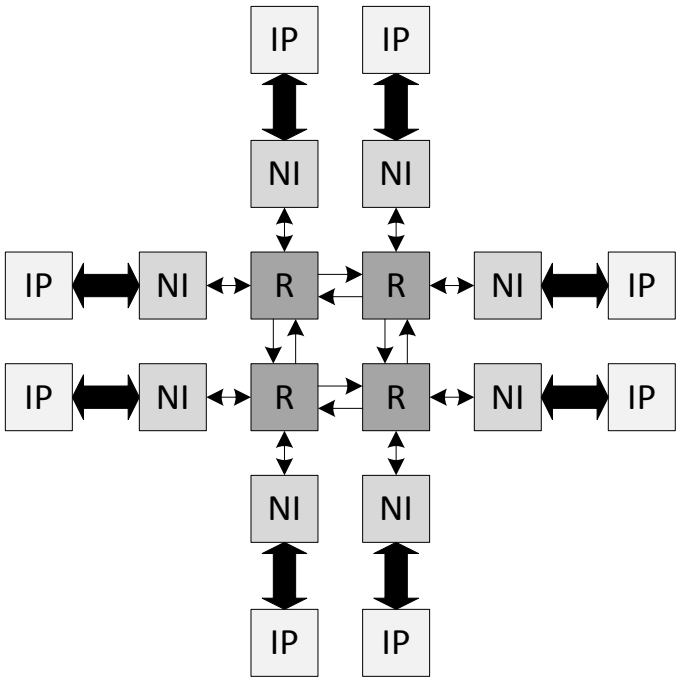


FIGURE 2.6 – Exemple d'architecture d'un réseau sur puce

4. http://www.xilinx.com/support/documentation/ip_documentation/plb_v46.pdf
5. http://en.wikipedia.org/wiki/Network_On_Chip

Les réseaux sur puce (NoC, *Network-on-Chip*) sont constitués de *routeurs* (R) et d'*interfaces réseaux* (NI) à la manière d'un réseau informatique avec ses routeurs et ses ordinateurs (qui représentent donc les différentes IP du système implanté sur le FPGA). Cette technologie est facilement extensible si un élément est rajouté dans l'architecture. Contrairement aux technologies de type bus, il n'y a pas de standard très répandu pour les systèmes embarqués dont l'architecture de communication est basée sur des réseaux sur puce. Généralement, ces réseaux sont souvent personnalisés pour satisfaire un besoin particulier : sécurité des transactions, adaptation du réseau aux modifications de l'environnement extérieur, architectures adaptées au haut-débit. Parmi celles-ci, on peut citer :

– SPIN [Adriahantenaina *et al.* 2003] : les précurseurs du NoC en 2003 au laboratoire LIP6.
– μSpider [Evain 2006] : un exemple du laboratoire Lab-STICC, pour des problématiques de contraintes de temps réel (un flot de conception est proposé dans cette optique).
– xPipes [Bertozzi & Benini 2006] : développé conjointement par les universités de Stanford et de Bologne, spécialisée dans les architectures gigacores (c'est-à-dire contenant de très nombreux éléments).

2.2 Travaux existants

Dans la littérature, plusieurs études se sont intéressées à la sécurité pour les systèmes embarqués [Ravi *et al.* 2004, Kocher *et al.* 2004]. Il en ressort que les mécanismes de sécurité peuvent se matérialiser sous deux formes : des blocs matériels ou des fonctions logicielles. Les solutions logicielles ne nécessitent généralement pas de modules matériels supplémentaires mais offrent des performances moindres en termes de latence par rapport à une solution purement matérielle. Par exemple, les auteurs de [Mencer *et al.* 1998] se sont intéressés à l'implémentation d'un algorithme de chiffrement pour les communications mobiles sur différentes technologies (DSP, FPGA, ASIC…) : il en ressort qu'une implémentation matérielle sur FPGA du chiffrement est environ 10 fois plus rapide qu'une implémentation sur DSP. La réactivité des mécanismes (c'est-à-dire le fait de bloquer l'attaque avant qu'elle ait le temps de se répandre dans le système) est essentielle pour garantir au système embarqué une protection efficace.

Les travaux existants dans le domaine de la protection des communications sont nombreux. Tout d'abord, on s'intéresse aux travaux sur les NoCs puis les bus. Dans un second temps, on exposera quelques solutions hybrides mélangeant des éléments matériels et logiciels. Enfin, comme la protection des mémoires fait aussi partie des points à observer, une vue d'ensemble de quelques solutions est donnée dans la dernière partie de cette section.

2.2.1 NoC

Pour les architectures à grande échelle (c'est-à-dire contenant de nombreux éléments) avec une architecture basée sur un réseau sur puce, on considère l'étude de trois solutions existantes. Premièrement, Diguet et al. [Diguet *et al.* 2007]proposent une solution où les contrôles de sécurité sont effectués dans chaque interface de manière distribuée (figure 2.7).

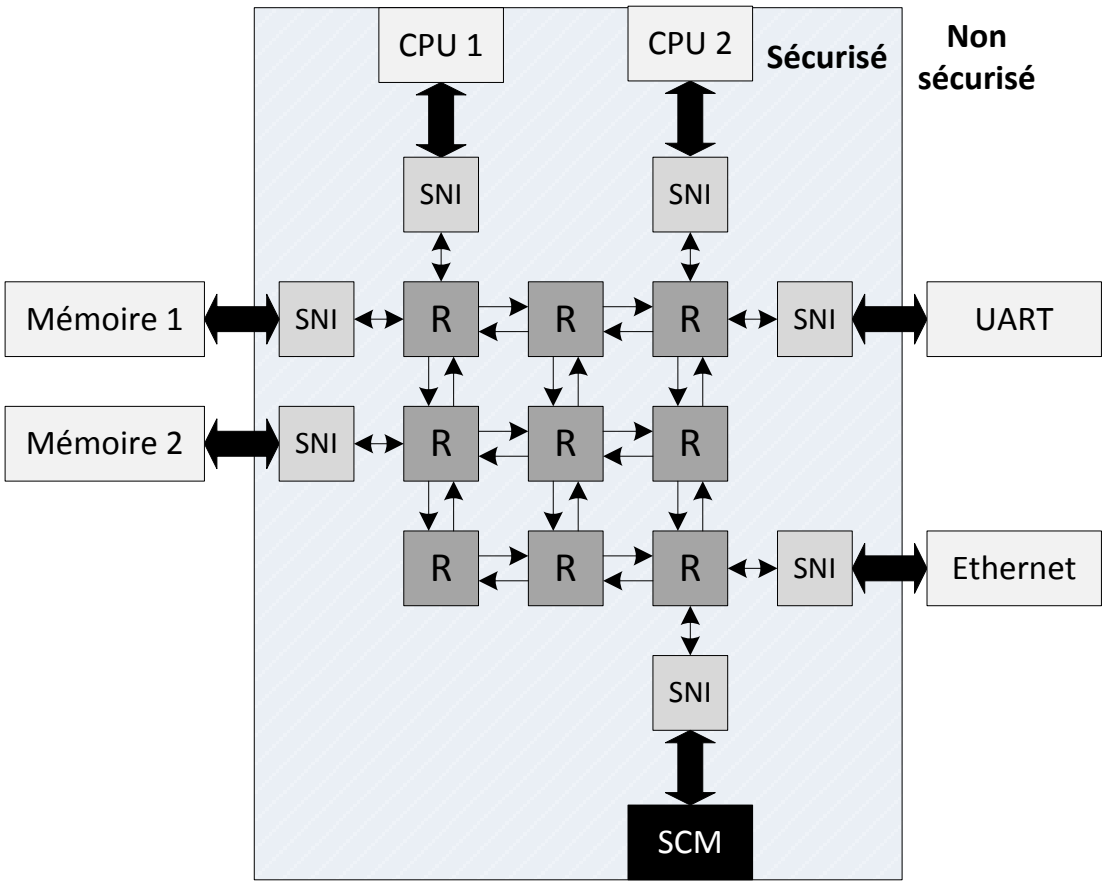


FIGURE 2.7 – Solution de [Diguet *et al.* 2007]

Les interfaces réseau SNI (*Secured Network Interface*) renvoient ensuite des informations à une unité de gestion SCM (*Security and Configuration Manager*) qui s'occupe des différentes configurations et des éventuelles contre-attaques ; la mise à jour des mécanismes de sécurité est réalisée par des reconfigurations dynamiques partielles des mécanismes de sécurité. Les travaux proposés par [Diguet *et al.* 2007] s'intéressent à des scenarii tels que le détournement de processeur ou de méthodes de déni de service. Globalement, le modèle de menace comprend tout ce qui génère ces scenarii mais ne propose pas de chiffrement (par contre, il y a de l'authentification et de l'intégrité) et le modèle ne prend pas en compte les attaques physiques (injections de fautes, attaques par canaux cachés...). De plus, il est possible de mettre à jour les mécanismes de sécurité par une reconfiguration dynamique partielle des éléments concernés. Un des inconvénients de cette solution est qu'elle ne gère pas le chiffrement des mémoires : toutes les données stockées sont lisibles par un attaquant. A priori, les transmissions d'informations entre les interfaces SNI et le gestionnaire SCM se font par le NoC où transite également les données : il s'agit d'informations de natures différentes, il faudrait mieux qu'elles utilisent des canaux dédiés.

Fiorin et al. [Fiorin *et al.* 2007b, Fiorin *et al.* 2008b, Fiorin *et al.* 2008a] proposent une alternative à cette contribution en ajoutant des « capteurs » à l'intérieur de l'interface sécurisée pour améliorer la finesse des contrôles. Chaque interface sécurisée, considérée comme étant de confiance, est constituée d'un ensemble de sondes, d'unités de protection et d'un noyau dédié à la gestion du réseau de communication.

Ces sondes peuvent bloquer les transactions entrantes selon des paramètres stockés dans une mémoire addressable par contenu (CAM, *Content-Adressable Memory*) considérée de confiance. Un gestionnaire de la sécurité du système (*Network Security Manager* dans la figure 2.8) collecte les informations provenant des interfaces pour détecter les collisions et les erreurs dans le trafic de données. Contrairement à la solution proposée dans [Diguet *et al.* 2007], les mécanismes de la solution de Fiorin et al. peuvent être mis à jour sans reconfiguration partielle (ils sont mis à jour par une modification des règles contenues dans les mémoires CAM) mais ne contiennent pas de services cryptographiques qui permettent de chiffrer d'une manière ou d'une autre les données qui transitent par les interfaces sécurisées. Tout comme la solution précédente, les travaux de Fiorin et al. ont pour objectif de détecter les attaques qui provoquent des dénis de services et qui visent à extraire des informations confidentielles du système. Pour ce faire, des mécanismes sont implantés aux interfaces et permettent de filtrer les accès selon des règles prédéfinies.

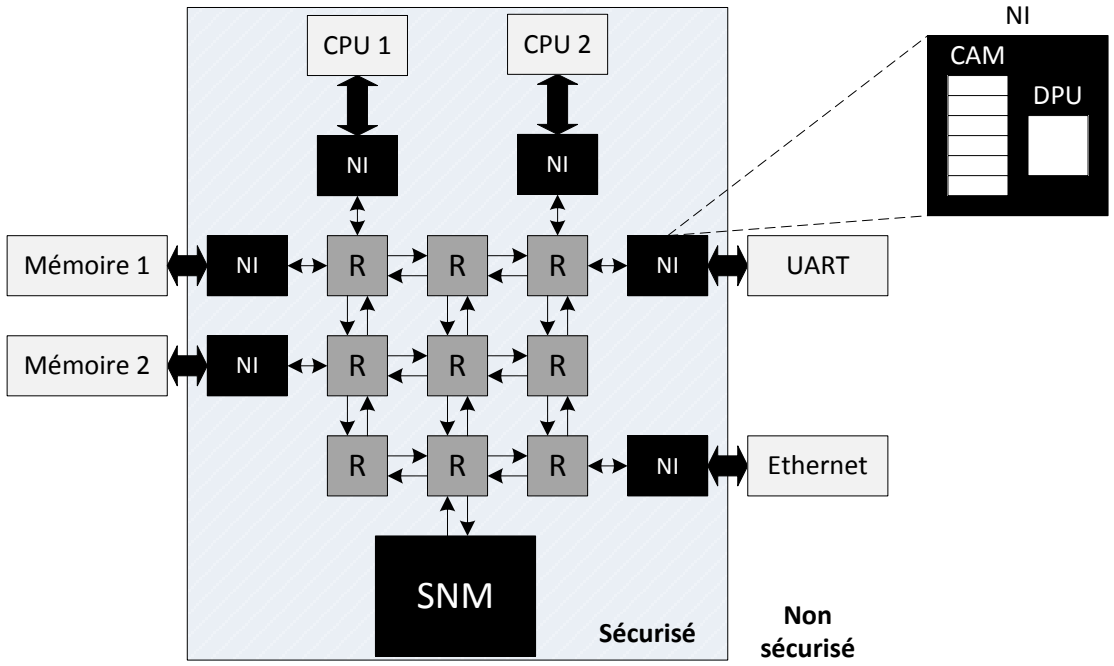


FIGURE 2.8 – Solution de [Fiorin *et al.* 2007a]

La solution proposée par Fiorin et al. est basée sur une interface sécurisée avec un mécanisme nommé DPU (pour *Data Protection Unit*, voir figure 2.8). Ce mécanisme autorise ou non une transaction en fonction des paramètres stockés dans les mémoires de confiance. Dans cette implémentation, le module DPU ne suffit pas à protéger le système dans le cas où l'IP source de la transaction a été modifiée par un attaquant par l'une des deux méthodes suivantes :

– Modification du comportement de l'IP initiatrice de la transaction par modification de son code source ou une reconfiguration dynamique partielle. Par exemple, il peut s'agir d'un cas où un processeur veut effectuer des transactions qui ne respectent pas les règles imposées par la DPU. Le modèle de Fiorin et al. ne couvre pas la protection des mémoires contenant les codes et applications liées aux processeurs ainsi que la protection du canal de reconfiguration dynamique partielle par un port dédié (sous réserve que l'attaquant soit capable de faire un bitstream valide).
– Modification du contenu des mémoires CAM contenant les règles de sécurité. Si un attaquant est capable de modifier les valeurs des paramètres de sécurité liés aux interfaces DPU (écriture sur un autre port ou attaque matérielle provoquant des événements de type bitflip qui modifient la valeur d'un ou plusieurs bits de la mémoire).

Compte tenu du cahier des charges proposé dans leurs travaux, Fiorin et al. sont capables de détecter des attaques de type déni de service mais ne peuvent pas anticiper la modification d'une IP en amont par un attaquant.

Enfin, concernant les travaux relatifs à la sécurité dans les NoC, on peut également citer les travaux de Sepulveda et al. ([Sepulveda *et al.* 2012b, Sepulveda *et al.* 2012a]).

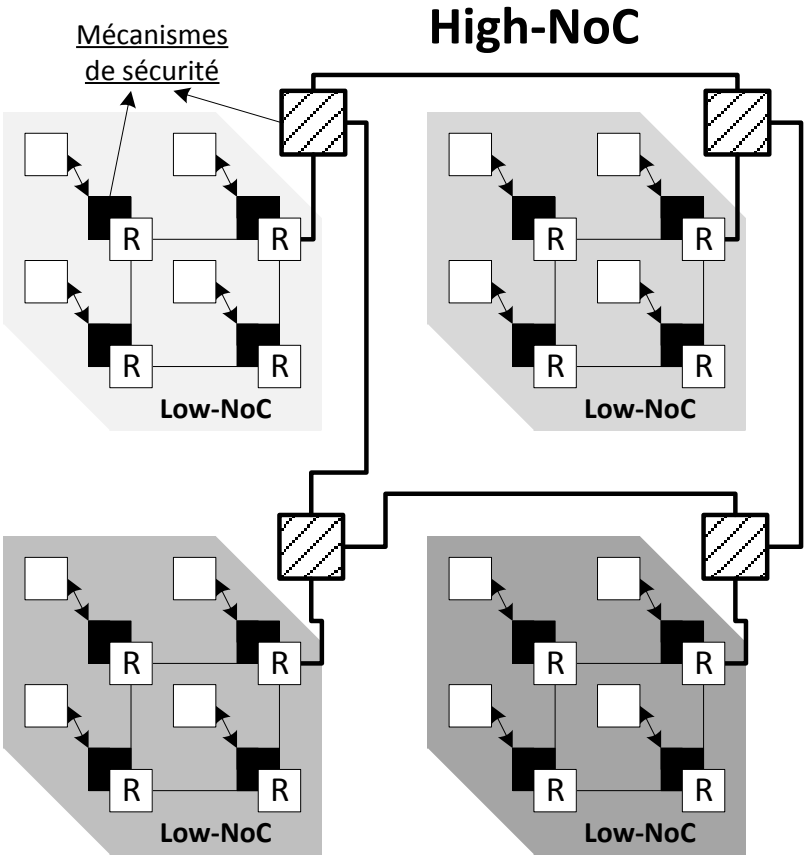


FIGURE 2.9 – Solution de [Sepulveda *et al.* 2012b]

Là où les travaux cités précédemment s'attachaient à la protection d'un seul NoC, la structure proposée par Sepulveda et al. permet de définir une certaine hiérarchie de la sécurité appliquée au système. Concrètement, sur la figure 2.9, on distingue deux catégories de NoC : les *low-NoC* et les *high-NoC*, chacun *low-NoC* a sa propre politique de sécurité. Par conséquent, le NoC hiérarchique proposé ici permet de définir un ensemble de NoCs plus simples, chacun pouvant être régi par des paramètres de sécurité différents : le NoC principal (ou *high-NoC*) permet alors de gérer le trafic qui vient de ses différents clusters.

Le panel d'attaques qui peut être couvert par la solution de [Sepulveda *et al.* 2012b] est assez large : les mécanismes de sécurité proposés permettent de vérifier les droits d'accès et également d'avoir un contrôle de l'authentification des données ; on pourrait donc détecter les scenarii de type *hijacking* où les processeurs sont détournés de leur fonction première ou tout ce qui relève de l'extraction d'informations confidentielles. Comme dans les solutions précédentes, comme le NoC hiérarchique et ses mécanismes de protection sont considérés comme étant de confiance, les attaques proviennent de l'extérieur (entrées-sorties ou mémoires). Par contre, cette solution ne possède pas non plus de fonction de chiffrement des données. Enfin, les attaques par déni de service (dans le sens de production de données à des fréquences surélevées) ne sont a priori pas détectées : des séries de paquets sont insérées sur le réseau mais avec des droits d'accès corrects (par exemple, en modifiant le contenu d'une mémoire contenant du code processeur).

2.2.2 Bus

Pour les systèmes multiprocesseurs ayant une architecture de communication à base de bus, la contribution majeure a été publiée par Coburn[Coburn *et al.* 2005] (voir figure 2.10). Cette approche est similaire aux travaux de Fiorin[Fiorin *et al.* 2007a, Fiorin *et al.* 2007b, Fiorin *et al.* 2008b, Fiorin *et al.* 2008a], elle est basée sur des interfaces réseaux avec des mécanismes de sécurité additionnels nommés SEI (*Security Enforcement Module*) implémentés entre chaque IP et le bus système. Chaque SEI va extraire des informations de l'IP et du bus qui permettent de définir ou non si la transaction courante est valide (c'est-à-dire si elle respecte les contraintes de sécurité du système). La principale contrainte de cette solution est que ces informations sont renvoyées à un gestionnaire centralisé qui est lui seul responsable de faire les opérations qui permettront finalement de bloquer ou non une transaction. Le surcoût en latence va donc forcément être augmenté, et donc provoquer des perturbations dans les communications au sein du système multiprocesseur (voir la comparaison sur les approches distribuée et centralisée dans le chapitre 5).

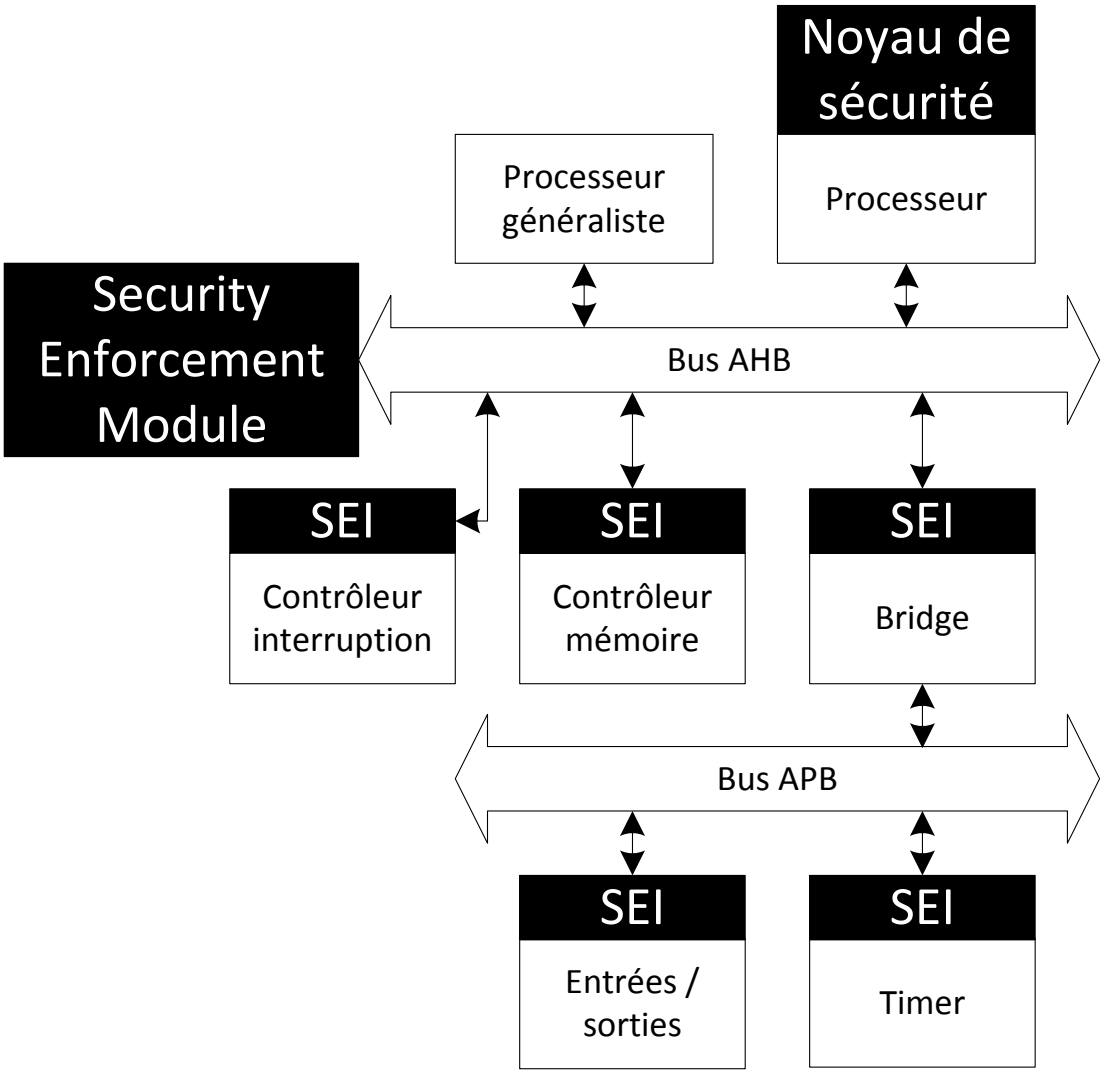


FIGURE 2.10 – Solution de Coburn et al. sur une architecture multiprocesseur générique

Cette solution ne propose pas la mise à jour des règles de sécurité ni de fonction cryptographique. Par contre, elle paraît la plus complète sur le modèle de menace pris en compte (même s'il n'est pas clairement exposé dans l'article) : les accès ainsi que la valeur de données spécifiées par le concepteur peuvent être contrôlés, enfin le module de contrôle de séquence SPU (*Sequence Protection Unit*) permet quant à lui de détecter des séquences d'attaques non désirées : le terme est assez large, il permet d'englober les attaques de type déni de services ou détournement du fonctionnement de processeurs.

L'approche proposée par Coburn et al. suggère de centraliser les contrôles au sein d'un gestionnaire de la sécurité du système (SEM, *Security Enforcement Manager*). Autrement dit, dès lors que ce gestionnaire est corrompu, il est possible de mettre hors d'usage le système : le SEM est un bloc matériel qui, sous certaines conditions peut être remplacé par une version corrompue grâce à une reconfiguration dynamique partielle (sous réserve que l'attaquant soit capable de produire un bitstream valide). Tout comme la solution proposée par Fiorin et al. [Fiorin *et al.* 2008b], les paramètres qui permettent de définir telle ou telle règle de sécurité sont stockés dans des mémoires qui peuvent être attaquées soit par une attaque matérielle (par exemple, une attaque électromagnétique) soit par un port non utilisé de la mémoire contenant les paramètres de sécurité.

2.2.3 Codesign logiciel-matériel

Pour protéger le système contre un modèle d'attaque prédéfini, d'autres solutions proposent une séparation physique des composants afin de définir des zones sécurisées et non sécurisées. Par exemple, les auteurs de [Huffmire *et al.* 2008] utilisent une séparation physique de certains blocs (nommés *moats*) durant le placement-routage où le routage est interdit, ce qui permet d'isoler certaines zones du FPGA ; d'autres modules (nommés *drawbridges*) permettent alors de réaliser des ponts sécurisés entre les différentes zones isolées du FPGA.

D'autres contributions proposent de combiner des éléments matériels à des éléments logiciels pour fournir des procédures de sécurité au système cible. Les travaux de Porquet [Porquet 2010] et de Plouviez [Plouviez 2011] se sont concentrés sur une solution de ce type : les auteurs proposent un mécanisme pour gérer plusieurs logiciels de manière sécurisée sur une architecture hétérogène où l'espace mémoire est partagé et protégé.

Le module logiciel *Global Trusted Agent* est en charge de définir les droits d'accès de chaque logiciel à l'espace mémoire alors que la partie matérielle est distribuée autour du réseau d'interconnexion afin d'analyser les différentes transactions et de vérifier qu'elles soient bien en accord avec les droits d'accès logiciels.

ARM a proposé son propre protocole, Trustzone [Xu *et al.* 2008], qui permet deux zones dont l'une est plus « privilégiée » que l'autre : cela peut être par exemple, une zone dédiée au noyau du système d'exploitation et une autre pour les applications sensibles telles que les problématiques d'authentification ou d'allocation mémoire. Un module matériel est en charge de surveiller que les données ne passent pas d'une zone à l'autre (le module *Monitor* dans la figure 2.11) et que, si nécessaire, les communications soient protégées (différenciation des deux modes par un bit de décision, le bit NS pour *Non-Secure bit* stocké dans un registre d'un coprocesseur).

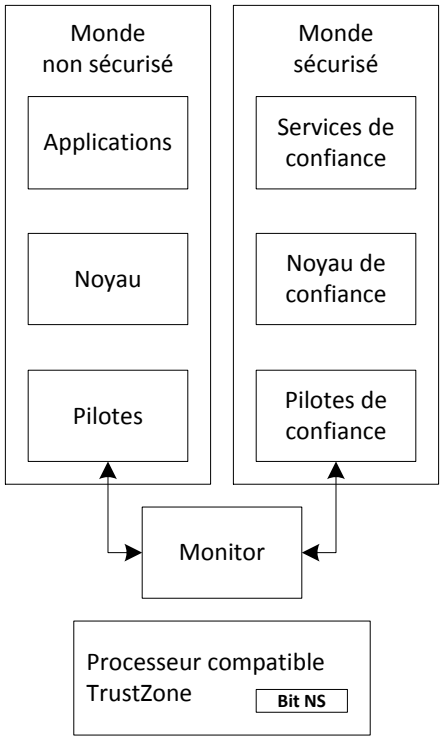


FIGURE 2.11 – Architecture de la solution Trustzone

2.2.4 Protection de la mémoire externe

D'après les travaux précédemment décrits, la protection des mémoires est un autre point sensible du développement de système à microprocesseurs. Ces systèmes sont soumis à de nombreuses attaques [Elbaz *et al.* 2006b]. La protection des mémoires en termes de confidentialité n'est pas suffisante : les attaques de type rejeu, relocation et spoofing (voir chapitre 3) peuvent malgré tout permettre d'interagir avec le système lorsqu'un processeur effectue des requêtes vers la mémoire. Par conséquent, il faut utiliser d'autres propriétés telles que l'intégrité et l'authentification afin de protéger les mémoires contre un panel d'attaques plus large.

XOM ([Lie *et al.* 2003]) est une solution qui permet de combiner la confidentialité et l'intégrité dans un système multiprocesseur où la mémoire externe n'est pas sécurisée : par conséquent, si le système d'exploitation est stocké en externe, il n'est pas de confiance. L'implémentation d'un tel système suppose que la structure du processeur soit modifiée en plus d'y ajouter des fonctions matérielles : l'impact en performances est assez important puisque les auteurs de [Lie *et al.* 2003] énoncent une perte de 50%.

L'architecture AEGIS ([Suh 2005]) est basée sur un processeur sécurisé qui contient également des fonctions de confidentialité et d'intégrité : tout ce qui sort ou rentre dans le processeur est sécurisé. En dehors du processeur, tout est considéré comme n'étant pas de confiance (et donc soumis à des attaques physiques et logicielles) : les mémoires, les entrées-sorties... Selon les configurations (taille des caches du processeur), la perte en performances va de 3,8% (taux de cache miss pour les données de 6,25%) à 130% (implémentation du processeur sans caches).

En 2006, [Elbaz *et al.* 2006b] propose une solution nommée PE-ICE qui permet de vérifier l'intégrité en parallèle du calcul de la confidentialité (réalisée par un bloc AES). Le modèle de menace est similaire : le système en lui-même est considéré comme sûr ; par contre, le bus processeur-mémoire ne l'est pas : la solution s'intéresse particulièrement aux attaques dites de "l'homme du milieu" (spoofing, rejeu, relocation). Dans le pire cas, les pertes en performances par rapport à une version sans protection sont de l'ordre de 20%. Si le système de base est protégé en confidentialité seule (par un chiffrement AES), la perte due à l'introduction de la solution PE-ICE n'est plus que de 4%.

[Vaslin *et al.* 2008] proposent un bloc confidentialité-intégrité basé sur une fonction AES (pour la confidentialité) et un contrôle de redondance cyclique (pour la vérification de l'intégrité). Les implémentations proposées avaient pour but de minimiser l'impact sur la mémoire et la consommation en énergie, les pertes en performances sont au minimum de 13-14%. Les auteurs de [Crenne *et al.* 2011b] proposent une extension à ces travaux : pour avoir des propriétés de confidentialité et d'intégrité en un seul module, ils utilisent l'algorithme AES-GCM (celui qui est décrit et utilisé ultérieurement dans ce manuscrit) : la partie intégrité de cet algorithme est réalisée par une fonction à faible latence ([Crenne *et al.* 2011a]).

2.3 Positionnement

2.3.1 Cahier des charges

Dans le cadre du projet SecReSoC [6], les applications visées sont destinées à être implémentées sous la forme d'une architecture multiprocesseur avec peu de composants, une communication par bus partagé est donc suffisante. En ce qui concerne les implémentations, elles sont réalisées sur une plateforme FPGA ML605 du fondeur Xilinx [Xilinx 2011b] : les technologies reconfigurables telles que les FPGAs permettent d'avoir des prototypes de systèmes complets en un temps de développement réduit. Par conséquent, les communications sont basées sur le protocole AXI proposé dans les outils de développement Xilinx. Étant donnés les travaux existants, notre contribution s'attache aux objectifs suivants :

– Des mécanismes de sécurité avec une faible latence : on veut éviter de créer des perturbations dans le réseau de communication existant. C'est d'ailleurs aussi la raison pour laquelle notre contribution est essentiellement basée sur des mécanismes matériels.
– La sécurité du système doit pouvoir être mise à jour en temps réel : lorsqu'une attaque est détectée, les mécanismes sont mis à jour avec de nouveaux paramètres. Par conséquent, on doit aussi être capable de s'adapter si une nouvelle application est implantée dans le système embarqué.
– Notre contribution doit couvrir un modèle de menaces prédéfini (détaillé dans la prochaine section).

6. http://labh-curien.univ-st-etienne.fr/secresoc/doku_wiki/doku.php

2.3.2 Modèle de menaces

Cette section présente les concepts généraux du modèle de menaces détaillé dans le chapitre 3. Tout d'abord, on peut envisager que l'attaquant d'un système embarqué peut avoir plusieurs objectifs :

– Détournement de processeur : écriture d'un code source contrefait pour provoquer des défaillances dans le système embarqué (accès illicites, écriture d'informations erronées).
– Extraction d'informations secrètes : révélation de clés cryptographiques ou de hash, données confidentielles relatives aux utilisateurs du système.
– Déni de service : c'est-à-dire le contournement des services de sécurité pour bloquer le système, désactivation des communications, injection de données factices pour provoquer des pics dans le trafic de données.

Une fois que les objectifs de l'attaquant sont connus, des scenarii comportementaux sont proposés pour illustrer les attaques définies dans la suite de ce document. Le premier scenario considère que le système s'exécute normalement lorsqu'un processeur commence à envoyer des blocs de données à une fréquence surélevée qui met le système dans un état où l'application principale ne répond plus. Une telle attaque peut être critique dans la mesure où les blocs de données et de contrôle sont majoritairement transmis à travers le bus de communication principal du système. Le deuxième scenario est basé sur le fait qu'un processeur veuille à un instant donné lire une mémoire contenant des clés cryptographiques : si l'attaquant est capable de les récupérer, le système n'est plus dans un environnement sécurisé. Les scenarii décrits ci-dessus sont réalisables si l'attaquant a accès aux mémoires où les données et les codes source sont stockés. Par conséquent, deux caractéristiques doivent être nécessairement remplies par le système multiprocesseur pour être sûr que le protocole de communication se déroule correctement :

– Si une attaque est détectée, le système doit réagir aussi vite que possible. Les données liées à cette attaque peuvent être ignorées car elles contiennent des commandes malveillantes.
– Les attaques doivent avoir le minimum d'influence sur le comportement attendu du système : l'application principale doit toujours respecter le cahier des charges.

2.3.3 Comparatif

Les solutions proposées dans ces travaux visent donc à créer un compromis des principales solutions existantes ([Fiorin *et al.* 2007a, Coburn *et al.* 2005]) pour une application sur des architectures multiprocesseurs où le lien de communication est constitué d'un bus.

	[Coburn *et al.* 2005]	**[Fiorin *et al.* 2008a]**	**Notre solution**
Paramètres quantitatifs			
Surface	6,6%	Environ 21%	
Latence	<1%	N/A	Objectif principal
Occupation mémoire	N/A	N/A	N/A
Paramètres qualitatifs			
Protocole de communication	Bus AHB/APB	Réseau sur puce NoC	Bus AXI-4
Granularité	Niveau IP	Niveau IP	Niveau IP extensible au niveau tâche
Modèle de menace couvert	Attaques sur le bus	Mémoire et bus externe	Mémoire et bus externe
Mise à jour de la communication	Non	Oui	Oui
Fonctions cryptographiques	Non	Non	Oui

TABLE 2.1 – Résultats individuels

Le tableau 2.1 et une synthèse de différents paramètres qui sont utilisés pour construire une solution qui soit un compromis des solutions existantes. La partie supérieure du tableau présente un comparatif des ressources physiques des différentes solutions : on le verra plus en détail dans le chapitre 3, mais l'objectif principal de la solution proposée dans ce manuscrit est d'avoir une sécurité à faible latence. Dans la partie inférieure du tableau, les différentes solutions sont comparées d'un point de vue qualitatif : notre solution partage le même modèle de menace que [Fiorin *et al.* 2008a]dans le sens où la mémoire et le bus externe reliant la puce à la mémoire ne sont pas considérés comme étant des composants sécurisés, alors que la solution de [Coburn *et al.* 2005] ne considère que le bus comme une unité à protéger.

Aucune des deux références proposées ne comporte de fonction cryptographique (confidentialité, intégrité) ce qui est le cas dans notre solution comme on le verra dans la suite de ce manuscrit. Enfin, la granularité de la protection définit à quel point les mécanismes de sécurité peuvent différencier les différents échanges. Dans les trois cas (y compris notre solution), un filtrage se fait au niveau des adresses : autrement dit, les mécanismes peuvent s'appliquer à une partie de l'espace mémoire et fournir à chaque IP (qui a son propre espace mémoire) une configuration des paramètres de sécurité différente ; une extension proposée dans le chapitre 6 montrera qu'il est possible d'améliorer la granularité de protection de notre solution au niveau logiciel pour que chaque tâche puisse avoir son propre environnement sécurisé.

2.4 Conclusion

Finalement, des mécanismes de protection existent pour tous les types d'architectures : que les communications soient basées sur des bus ou des réseaux sur puce, les travaux existants permettent généralement de filtrer certaines données afin qu'elles n'infectent pas profondément le système. Néanmoins, la plupart du temps, les problèmes de mémoires restent non résolus : les travaux de références dans la protection des communications ([Coburn *et al.* 2005, Fiorin *et al.* 2008a]) ne proposent pas de bloc de protection des mémoires en termes cryptographiques. Typiquement, les travaux de [Crenne 2011] sont un exemple de ce qui est réalisable pour protéger les mémoires en confidentialité et en intégrité. Les travaux proposés dans cette thèse sont en quelque sorte le point de liaison entre ces deux domaines dans la mesure où ils aspirent à réaliser un compromis des travaux existants sur la protection des communications tout en intégrant des propriétés cryptographiques qui permettent d'étendre le modèle de menace tout en proposant des performances qui permettent d'intégrer facilement les mécanismes de sécurité dans une architecture multiprocesseur.

3 Protection statique des communications

Ce chapitre a pour but de poser les bases des mécanismes de sécurité développés dans cette thèse : quel type d'attaque est pris en compte ? Quelle est la structure des mécanismes proposés ? Est-ce que le modèle de menaces est bien couvert ?

Sommaire

3.1 Cadre de la solution

3.1.1 Contraintes sur la plateforme

Les contributions apportées par cette thèse visent à proposer des mécanismes de sécurité pour des architectures multiprocesseurs. Ces architectures peuvent être implémentées sur des circuits intégrés pour application spécifique ASIC (*Application Specific Integrated Circuit*) ou des technologies reconfigurables de type FPGA (*Field-Programmable Gate Array*). Dans les chapitres suivants, les implémentations sont réalisées sur une technologie reconfigurable de type FPGA pour plusieurs raisons :

– Cette technologie permet d'implémenter rapidement des architectures fonctionnelles. De plus, ces composants FPGA sont reconfigurables des dizaines de fois (contrairement aux composants de type ASIC qui ne peuvent pas être modifiés une fois sortis de l'usine du fondeur).
– Chaque puce FPGA contient d'autres éléments que la logique du FPGA en lui-même : des blocs DSP (*Digital Signal Processor*, composants optimisés pour le traitement du signal) mais également des mémoires internes (Block RAM) et des interfaces vers des mémoires externes (par exemple, DDR) dans lesquelles on peut stocker des données. Dans les FPGAs du fondeur Xilinx [Xilinx 2012c], il peut y avoir jusqu'à 4 Mo pour des FPGAs de la famille Xilinx Virtex-6 avec un temps d'accès plus rapide que celui d'une mémoire externe classique de type DDR [Xilinx 2011d, Xilinx 2011c] : par exemple, pour une lecture, le temps de latence d'une Block RAM est de 1 cycle alors qu'elle est au mieux de 32 cycles pour une mémoire de type DDR3.
– Ces travaux, réalisés dans le cadre du projet ANR SecReSoC [1], visent des applications du type chiffreurs-signeurs avec des propriétés de faible coût matériel et de reconfigurabilité des blocs de chiffrement. Les applications visées par ce projet ne nécessitent pas un grand nombre d'éléments ; par conséquent, une architecture de communication simple de type bus est suffisante (si des extensions à l'architecture de base sont prévues, la communication par bus hiérarchique sera choisie : dans ce cas, plusieurs bus traditionnels sont connectés par l'intermédiaire de bridges).
– D'après l'état de l'art, il existe plusieurs protocoles de bus (voir section 2.2.2). Notre choix s'est porté sur le protocole AXI du fabricant ARM : ce protocole est standardisé dans les dernières versions des outils Xilinx (14.2, à l'heure de la rédaction de ce manuscrit) et on peut espérer une compatibilité avec d'autres plateformes utilisant le protocole AXI (par exemple, les SoC Zynq de Xilinx qui sont des systèmes hétérogènes embarquant des FPGAs associés à des processeurs ARM de type Cortex A9 [Xilinx 2012d]).

1. http://labh-curien.univ-st-etienne.fr/secresoc/doku_wiki/doku.php

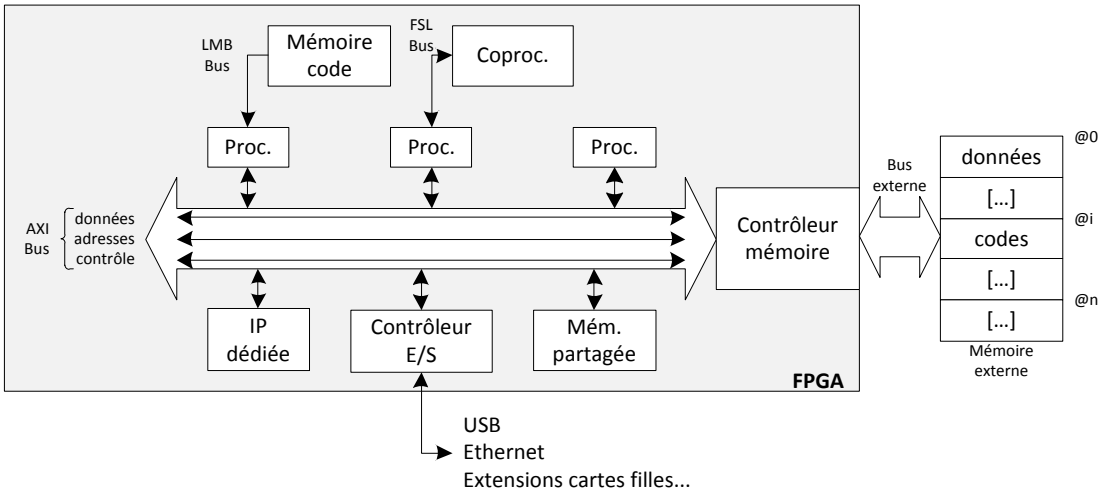


FIGURE 3.1 – MPSoC générique sur puce FPGA

Dans la suite de ce chapitre, on considère un système semblable à celui qu'on peut trouver dans la figure 3.1. Les systèmes multiprocesseurs qui sont pris en compte contiennent des processeurs hétérogènes (certains associés à un co-processeur, d'autres avec une mémoire de code embarquée, d'autres avec des mémoires caches...), des mémoires internes et externes, des entrées-sorties (liaison série, Ethernet...). Tous ces composants vont permettre d'illustrer différentes communications (inter-processeurs, écriture-lecture de mémoires...) et de montrer qu'ils peuvent être protégés selon un modèle de menaces donné. La suite de ce chapitre propose de présenter le modèle de menaces choisi ainsi que les mécanismes de protection mis en place sur une architecture semblable à celle proposée dans la figure 3.1.

3.1.2 Modèle de menaces

3.1.2.1 Transactions internes

Tout d'abord, on prend en compte le cas d'une communication entre deux processeurs embarqués (softcores tel que le Microblaze ou hardcore comme le PowerPC). On suppose qu'un des deux processeurs a été compromis par une intervention extérieure (injection d'une reconfiguration dynamique partielle contenant un processeur contrefait ou modification du code source associé à ce même processeur).

Dans ce cas, le processeur modifié peut envoyer des données incorrectes au processeur cible (soit par des valeurs aléatoires, soit en modifiant le format des données transmises) : les calculs produits deviennent erronés et inutilisables (dans le sens où on ne peut pas savoir si les données sont correctes) par l'utilisateur final du système. Les seules options envisageables sont d'isoler le processeur attaqué du reste du système ou de faire une remise à zéro complète du système : on recharge alors le bitstream initial et les mémoires contenant les logiciels sont réinitialisées.

3.1.2.2 Transactions externes

Ensuite, on étudie le cas où un processeur doit aller récupérer du code en mémoire externe : vu que, dans un premier temps, l'implémentation simple propose des processeurs sans caches, tous les codes processeur sont stockés dans la mémoire externe en clair (la suite du chapitre propose les mécanismes pour protéger ces codes). Si aucun mécanisme de sécurité n'est implémenté dans le système embarqué, le code est interprété normalement par le processeur qui, par conséquent, peut effectuer des transactions illégales (accéder à un composant non autorisé selon les cahiers des charges du système, renvoyer les données d'une mémoire critique contenant des données confidentielles vers une interface utilisateur) ou même provoquer une augmentation brutale du débit de données de façon à surcharger l'arbitre responsable de l'organisation temporelle des transactions sur le bus de communication : on observe alors des phénomènes de *deadlock* (surcharges des canaux de communication) ou de *livelock* (données qui circulent indéfiniment dans le circuit de communication : pertes de latence, bande passante et puissance). Finalement, il s'agit d'une autre faille dans les communications que l'attaquant peut exploiter sur le système multiprocesseur envisagé.

3.1.2.3 Vecteurs d'attaques

Dans les systèmes multiprocesseurs définis dans la section 3.1.1, on suppose que les attaques ne peuvent survenir que dans deux zones : la mémoire externe et le bus externe (c'est-à-dire le bus contenant les signaux physiques de la DDR qui relie le circuit FPGA à la mémoire DDR elle-même, voir figure 3.2).

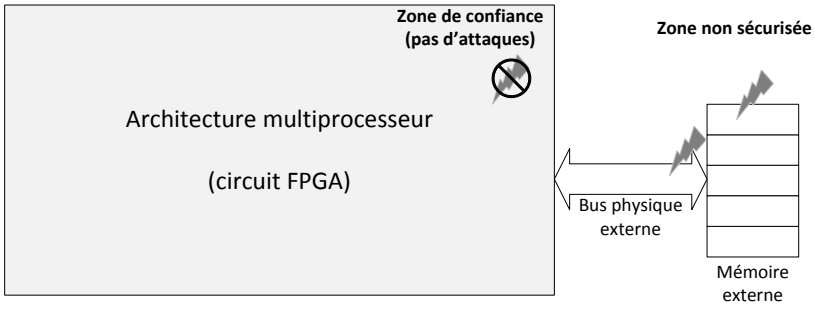


FIGURE 3.2 – Zones de confiance sur une architecture multiprocesseur

Concrètement, on considère donc toutes les possibilités qui visent à altérer le contenu ou extraire des informations du bus externe et de la mémoire externe. Tout ce qui est contenu dans la puce FPGA (toute l'architecture matérielle dont les processeurs, les mémoires internes...) n'est pas soumis à des attaques : on parle alors de zone de confiance (on suppose qu'aucune attaque ne vise la puce FPGA en elle-même).

3.1.2.4 Synthèse

Finalement, les mécanismes de sécurité proposés dans ce travail doivent permettre de protéger les deux types de transactions présentées ci-dessus. Autrement dit, les mécanismes doivent respecter les propriétés suivantes :
- Bloquer les accès illégaux en termes d'adresse (filtrage des adresses).
- Pour éviter des attaques de type *buffer overflow*, le format des données transmises à chaque opération doit être vérifié.
- Les données stockées en mémoire externe doivent être protégées, des primitives cryptographiques doivent être implémentées dans ce but.
- Les mécanismes de sécurité doivent aussi respecter un cahier des charges (défini conjointement par l'utilisateur et le concepteur du système) : certaines zones mémoires peuvent ne pas être protégées si elles ne sont pas considérées comme étant critiques.

3.1.3 Vue haut-niveau de la solution proposée

Pour prémunir un système basé sur une architecture multiprocesseur du modèle de menaces défini dans la section 3.1.2, des mécanismes de sécurité (nommés « pare-feux » ou « firewalls » dans la suite du manuscrit) sont implémentés à l'intérieur du système cible.

Ce système est composé de processeurs, de mémoires (internes et externes) et d'IP dédiées à l'application finale (ça peut être, par exemple, un traitement d'image ou des calculs statistiques).

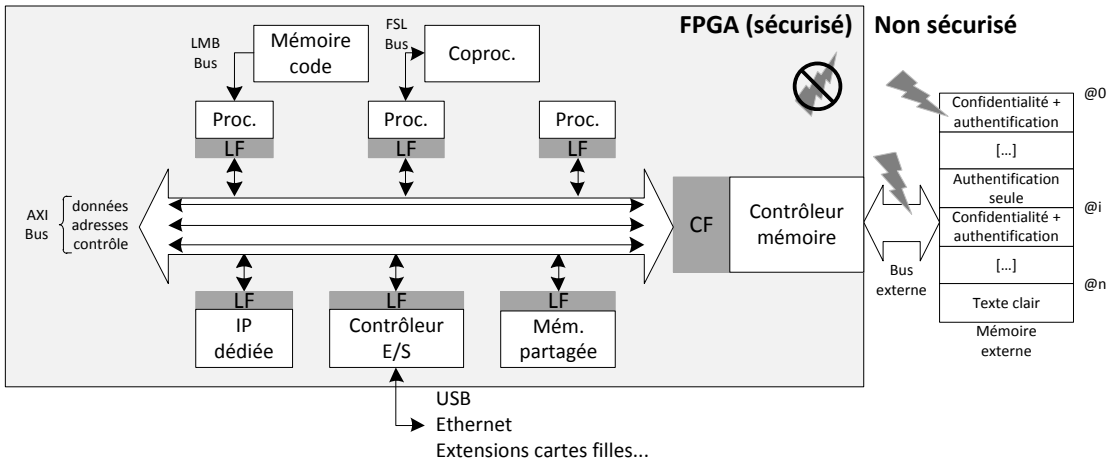


FIGURE 3.3 – Solution proposée implantée sur un MPSoC générique

Toutes les ressources (hormis la mémoire externe) sont implantées dans une puce FPGA et sont connectées à un bus de communication utilisant le protocole AXI [ARM 2012]. La contribution de cette thèse propose d'ajouter des mécanismes de sécurité à chaque interface entre un composant (IP ou processeur) et le bus de communication afin de fournir une passerelle sécurisée aux transactions entre les différents composants. La figure 3.3 présente un tel système avec des éléments internes et une mémoire externe. Chaque ressource est connectée par une interface nommée *Local Firewall* (LF) qui permet de filtrer les accès selon des règles prédéfinies conjointement par le concepteur et l'utilisateur. La mémoire externe se connecte par une interface particulière nommée *Cryptographic Firewall* (CF) qui, en plus de fournir des services comparables au *Local Firewall*, ajoute une couche de fonctions cryptographiques (confidentialité, authentification...). Les règles (ou politiques de sécurité) des deux types de pare-feux sont stockées dans des mémoires internes (de type Block RAM) intégrées dans les pare-feux et considérées de confiance, c'est-à-dire auxquelles l'attaquant ne peut pas accéder.

Pour un système complet, il faut ajouter un système d'exploitation. On peut citer, par exemple, des solutions de type Linux embarqué : ARMadeus [2] ou Petalinux [3] en font partie. Il existe également d'autre systèmes moins complexes qui permettent tout de même de faire des opérations multi-tâches sur des bases d'UNIX (on peut citer μCOS [4], compatible avec le standard POSIX). Les travaux présentés dans cette thèse considèrent, sauf mention contraire, que les systèmes multiprocesseurs sont gouvernés par le système d'exploitation *standalone* [5] de Xilinx. *Standalone* est le logiciel minimal qui permet d'utiliser les fonctions de base, telles que les interactions avec les processeurs (interruption, mailbox, mutex...) ou d'accéder aux différentes entrées-sorties du système : on peut ainsi créer un Exemple Concret Minimal (ECM) qui illustre les différents événements qui peuvent survenir dans une architecture multiprocesseur plus complexe.

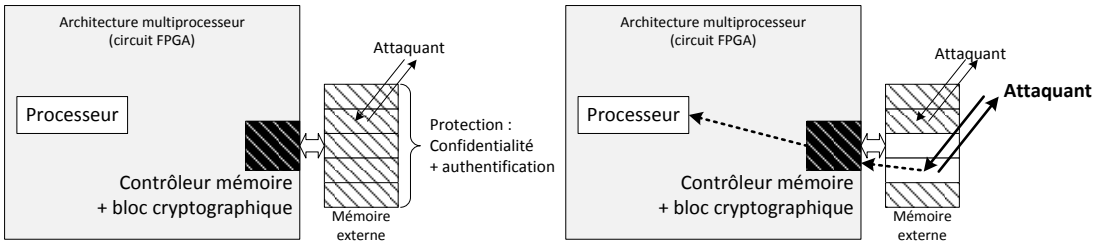
Concernant l'ECM proposé dans ce chapitre, on considère également que les processeurs peuvent avoir des mémoires caches et que les filtrages d'adresses se font sur les adresses physiques. Dans une implémentation à base de processeurs Microblaze, l'unité de gestion mémoire (MMU, *Memory Management Unit*) est optionnelle ([Xilinx 2012a]) : en *mode réel*, la traduction adresse virtuelle vers adresse physique est directe ; dans le *mode virtuel*, le Microblaze est alors équipé d'une MMU dont il faut connaître la structure pour formuler correctement les adresses à filtrer (on suppose que le concepteur est capable de connaître cette structure). Dans la suite de ce chapitre, on considère donc des processeurs qui exécutent le système d'exploitation *standalone* et la transformation entre adresses virtuelles et physiques est directe.

3.1.4 Politiques de sécurité

Les pare-feux présentés dans cette thèse utilisent des politiques de sécurité stockées dans des mémoires considérées comme étant de confiance (mémoires internes de type Block RAM, à l'intérieur de la puce FPGA). Ces politiques contiennent un ensemble de paramètres qui permettent de détecter les comportements anormaux du système. D'après le modèle de menace décrit dans la section 3.1.2, les mécanismes de protection proposés dans ce travail doivent permettre de détecter les attaques provenant du bus externe et des mémoires externes (protéger contre les modifications réalisées par un attaquant).

2. http://www.armadeus.com/francais/
3. http://www.petalogix.com/products/petalinux
4. http://micrium.com/page/products/rtos/os-ii
5. http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_1/oslib_rm.pdf

La protection la plus radicale consisterait à protéger la totalité des mémoires avec des services cryptographiques (confidentialité, authentification). Cette situation est représentée dans la figure 3.4a : dans ce cas, l'attaquant ne peut ni déchiffrer la mémoire externe ni modifier son contenu. Malheureusement, cette option a un surcoût important en termes de latence (voir section 3.3). Une alternative consiste à protéger partiellement l'espace mémoire avec des fonctions cryptographiques uniquement pour les sections les plus critiques.



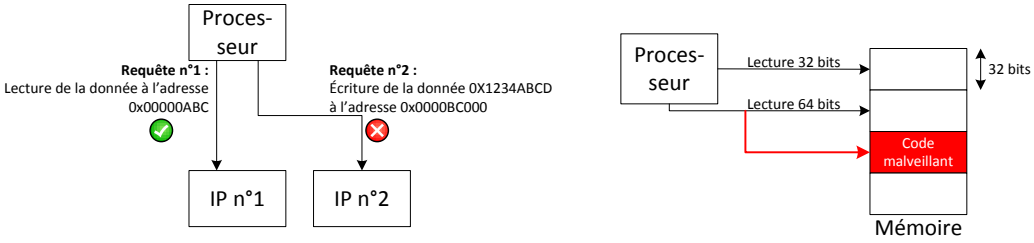
(a) Protection totale (b) Protection partielle

FIGURE 3.4 – Protection d'une mémoire externe en confidentialité et authentification

La deuxième situation est représentée dans la figure 3.4b : ici, la partie hachurée est protégée comme dans la figure 3.4a et les parties non hachurées sont en clair. L'attaquant peut alors lire et écrire sans problème ces données en clair. Dès lors qu'une section mémoire est en clair (que ce soit une donnée ou du code processeur), l'attaquant peut utiliser cette faille pour mettre en péril le fonctionnement du système : dans la figure 3.4b, le processeur vient lire du code en clair qui a été modifié par l'attaquant (flèches en gras). A partir de ce moment, il peut tout à fait faire des tentatives de buffer overflows ou des accès à des blocs matériels non autorisés (par exemple des blocs de chiffrement contenant des clés cryptographiques). Par conséquent, il est nécessaire de trouver d'autres méthodes de détection des attaques au sein du système embarqué pour les zones mémoire où le texte est en clair.

Fondamentalement, les politiques de sécurité prennent en compte les droits d'accès en lecture-écriture (lecture-écriture, lecture seule, écriture seule, aucun accès autorisé), les formats de données autorisés (des données peuvent être transmises sur 8, 16 ou 32 bits) et des informations cryptographiques (modes de confidentialité-authentification, clés et codes d'authentification de message).

Ainsi, en accord avec les exigences en termes de sécurité définies par l'utilisateur, les zones mémoires en clair sont protégées contre les accès illégaux (par le contrôle des droits en lecture-écriture, voir figure 3.5a) et les tentatives de buffer overflow (lecture d'une donnée plus longue pour accéder à du code malveillant, voir figure 3.5b) par le contrôle du format de donnée de la transaction courante.

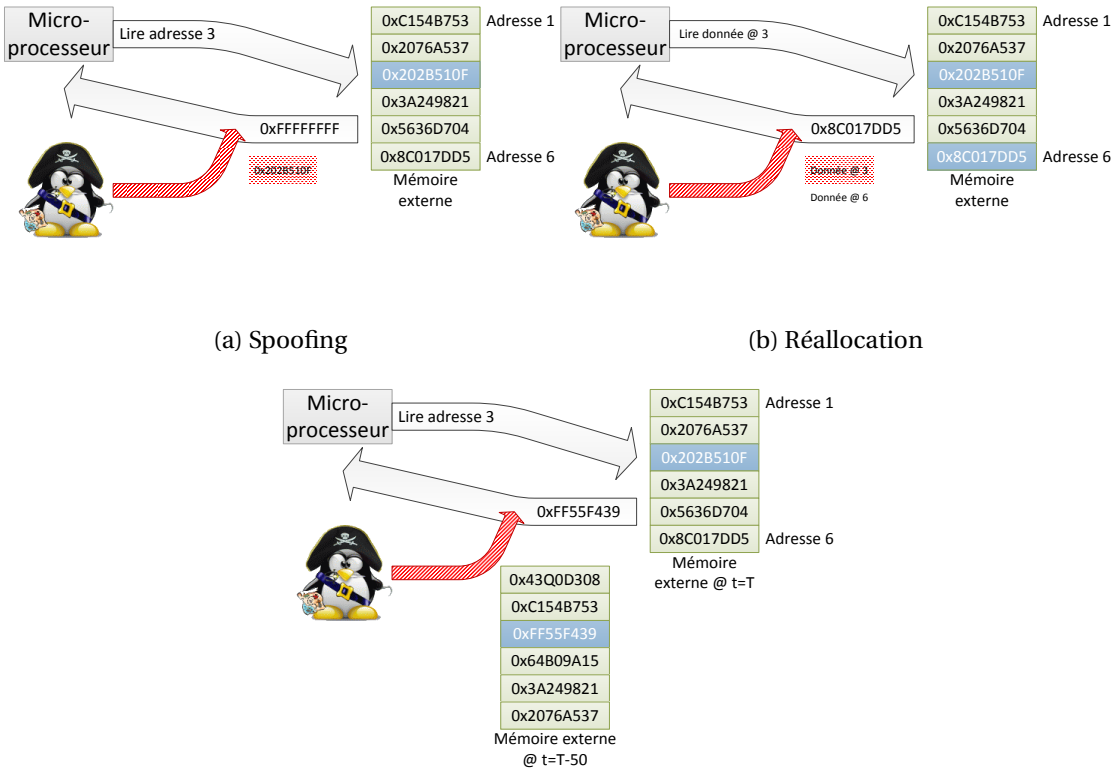


(a) Accès illégaux

(b) Protection partielle

FIGURE 3.5 – Protection des droits en lecture-écriture et du format

Des marqueurs de temps (horodateurs, ou *timestamps*) sont utilisés pour surveiller les temps d'accès à la mémoire externe (détection des attaques par rejeu). Les attaques par spoofing et réallocation sont aussi visées par les mécanismes de sécurité (filtrage de l'espace d'adresses mémoire et utilisation de primitives d'authentification).



(a) Spoofing

(b) Réallocation

(c) Rejeu

FIGURE 3.6 – Spoofing, rejeu et réallocation

La figure 3.6a illustre le principe du spoofing dans lequel l'attaquant est capable de renvoyer sur le bus externe une donnée complètement différente de celle attendue par le processeur sans qu'il le sache (ici, la donnée $0x202B510F$ que le processeur est sensé lire est remplacée par $0xFFFFFFFF$). La figure 3.6b illustre le principe de la réallocation, l'organisation des pages mémoire est permutée par l'attaquant : dans ce cas, les données situées aux adresses 3 et 6 sont inversées. Enfin, le principe de l'attaque par rejeu est présenté dans la figure 3.6c : l'attaquant a ici réalisé une sauvegarde de la mémoire pour la rejouer ultérieurement (dans la figure 3.6c, la donnée renvoyée au processeur à l'instant T est bien celle située à l'adresse 3 mais avec la valeur qu'elle avait au temps $T-50$.

3.2 Local Firewall

Chaque pare-feu est composé de différents blocs (tels que le *Security Builder* ou le *Firewall Interface*) connectés comme indiqué dans la figure 3.7. Tandis que le *Security Builder* est le module qui est responsable de la gestion des politiques de sécurité et de leur application, le module *Firewall Interface* est le point de passage obligatoire vers le monde extérieur. Ce dernier module agit comme une sorte de passerelle entre le bus de communication AXI-4 et l'IP associée (qui peut être une IP dédiée à l'application, un contrôleur d'entrée-sortie, un contrôleur mémoire...).

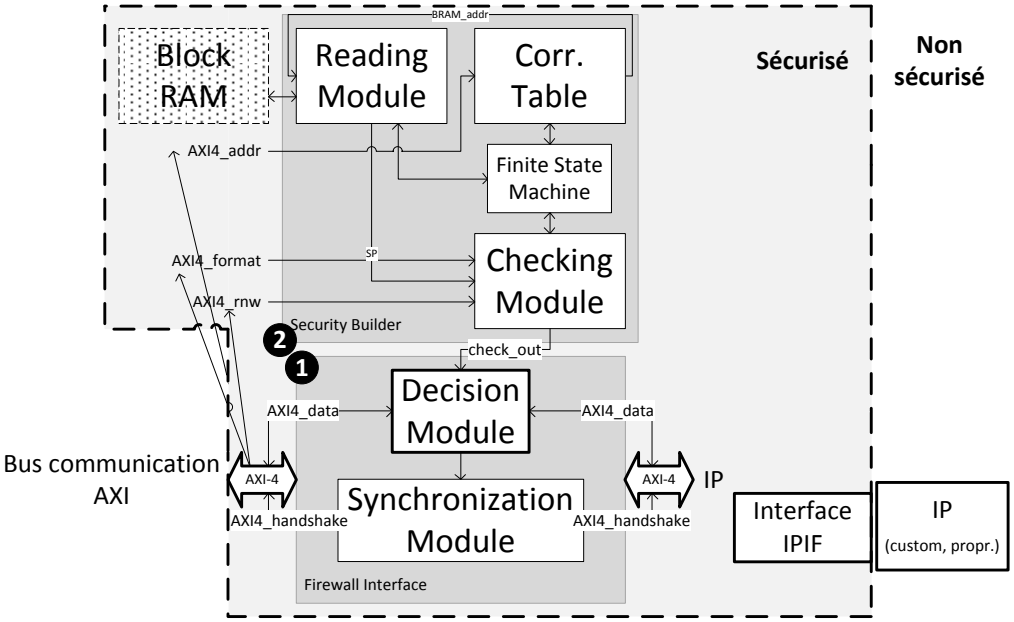


FIGURE 3.7 – Structure d'un Local Firewall

Toute la structure du pare-feu est considérée comme étant de confiance (car implanté en matériel dans la puce FPGA), la seule partie considérée comme non sécurisée étant constituée du bus externe et de la mémoire externe (pour le cas d'un *Cryptographic Firewall* uniquement, voir figure 3.3).

3.2.1 Firewall Interface

Le module *Firewall Interface* (portant le numéro 1 dans la figure 3.7) réalise essentiellement 2 tâches :

– Une fois qu'un bloc de données a été analysé et validé (ses paramètres concordent avec la politique de sécurité associée), le module *Firewall Interface* transmet la donnée à l'élément cible (bus de communication système ou IP) ; cette opération est effectuée par le *Decision Module*.
– Ce module *Firewall Interface* synchronise les signaux du protocole de communication tels que les signaux de poignée de main (*AXI_WVALID*, *AXI_ARREADY*...) ou d'autres signaux de contrôle (*AXI_WSTRB*, *AXI_WLAST*...) dans le but de respecter les transactions entre deux composants ; le bloc de données est synchronisé avec ses signaux de contrôle correspondants pour éviter des pertes ou des duplications de données. Le *Synchronization Module* est composé d'une série de bascules où le signal d'horloge (tout du moins son front montant) est le signal d'acquittement *check_out* envoyé par le *Security Builder* quand toutes les opérations liées à l'analyse du bloc de données courant sont terminées.

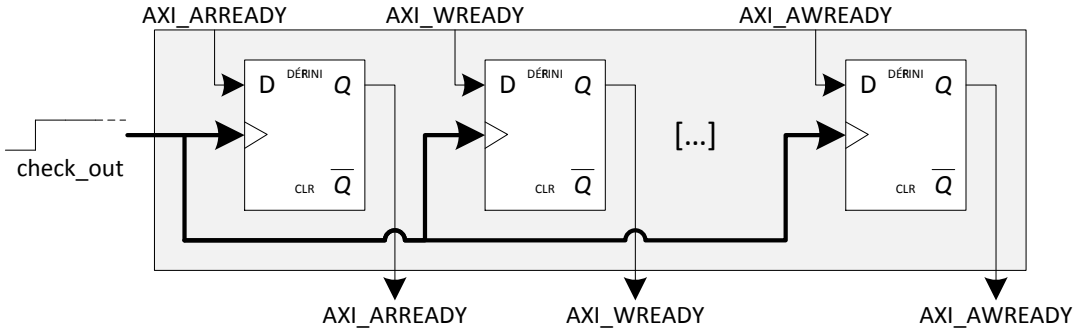


FIGURE 3.8 – Structure d'un Synchronization Module

Étant donné que toutes les bascules sont connectées en parallèle, la latence du module *Firewall Interface* pour traiter un bloc de données de 32 bits est de 2 cycles (les interfaces du bus de communication et des IP sont sur 32 bits) : 1 cycle pour le *Decision Module* et 1 cycle pour la synchronisation. Pour N blocs de 32 bits (sans gestion de propriétés de burst ou de pipeline), la latence suit l'équation suivante :

$$latency(N) = N * 2 \tag{3.1}$$

Dans les implémentations futures, on peut imaginer une version pipelinée qui devrait diminuer la latence globale pour un certain nombre de données traitées, l'équation pour une architecture pipelinée à m étages deviendrait :

$$latency(N) = m + N - 1 \tag{3.2}$$

Ceci poserait sûrement quelques problèmes dans le protocole de communication. En tout cas, le *Firewall Interface* nécessiterait 2 cycles d'horloge pour le premier bloc de données et 1 cycle supplémentaire pour chaque bloc de données suivant : pendant que la donnée(n) est traitée par le *Synchronization Module*, la donnée(n+1) est traitée par le *Decision Module*. Ce point n'a pas été approfondi, il devrait être étudié dans des travaux futurs ayant pour but d'optimiser les performances de la solution.

3.2.2 Security Builder

Le *Security Builder* est le composant principal des pare-feux (noté 2 dans la figure 3.3). Il est constitué de 4 sous-modules :

– **Correspondence Table (CorrTable)**. Les politiques de sécurité sont stockées dans une Block RAM considérée de confiance (coin supérieur gauche de la figure 3.3) et peuvent être identifiées par une adresse. Chaque politique est définie pour un espace d'adresses physiques donné (délimité par la limite basse et la limite haute), la transcription vers les adresses des politiques se fait de la manière suivante :

Algorithme 1 Algorithme de la Correspondence Table

$bus\_addr \leftarrow Bus\_address$
$bram\_addr \leftarrow 0x00000000$
for $sousmodule = 1$ to N **do**
  if $bus\_addr > reg_{low}$ and $bus\_addr < reg_{high}$ **then**
    $bram\_addr \leftarrow reg_{out}$
  end if
end for
if $bram\_addr = 0$ **then**
  $notFoundFlag \leftarrow 1$
end if
$BRAM\_address \leftarrow bram\_addr$

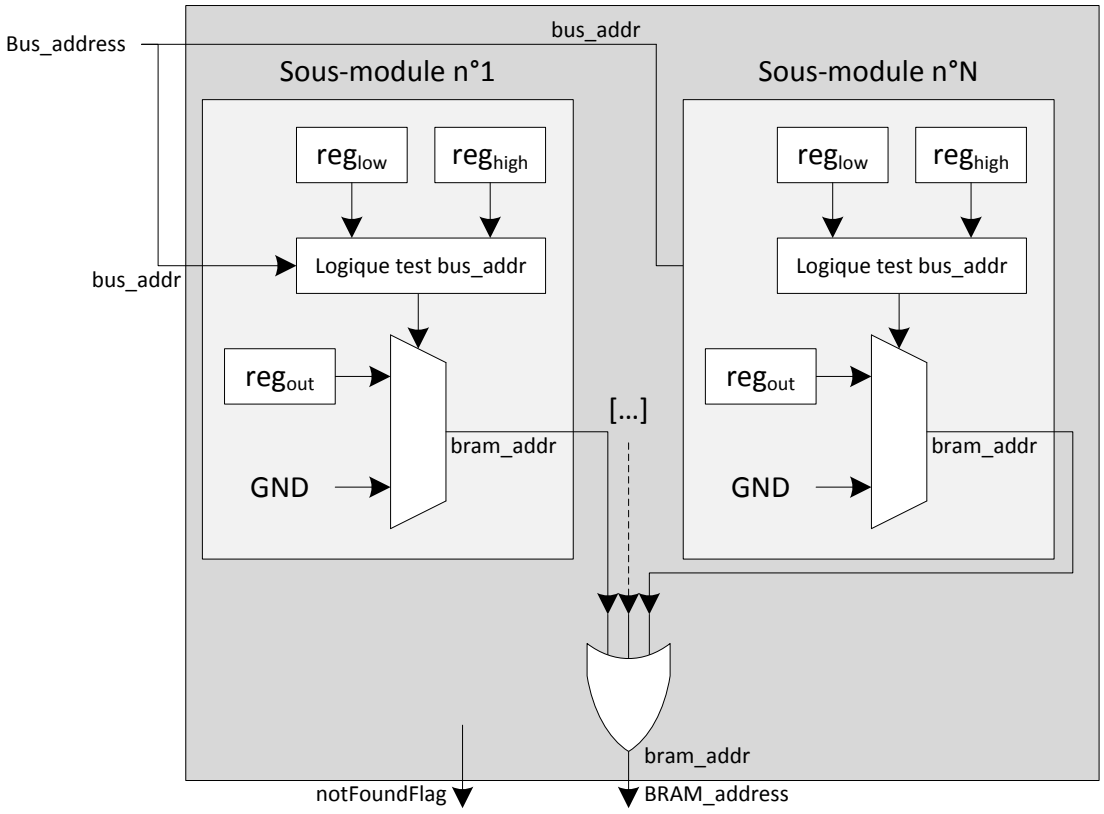


FIGURE 3.9 – Implémentation de la Correspondence Table

CorrTable définit l'adresse où aller lire la politique de sécurité ($BRAM\_address$) pour un espace d'adresses donné (reg_{low} et reg_{high}), cette opération est faite en 1 cycle d'horloge : d'après la figure 3.10, l'appartenance de l'adresse bus à un espace d'adresse est effectuée par tous les sous-modules en parallèle (chaque sous-module vérifiant un espace distinct). Par exemple, l'adresse physique du bus 0x1234ABCD (contenue dans l'espace d'adresses $[0x12340000; 0x1234FFFF]$) est à analyser avec la politique de sécurité localisée dans la Block RAM à l'adresse 0xFF00FF00. Les relations entre ces adresses sont effectuées par la *Correspondence Table*. Les registres contenant les limites des espaces d'adresses (reg_{low} et reg_{high}) ainsi que les adresses Block RAM où sont stockées les politiques (reg_{out}) sont définies par le concepteur (les aspects mise à jour sont traités dans le chapitre 4). D'un point de vue matériel, il y a autant de sous-modules que de politiques à vérifier (la limite du nombre de modules qui peuvent être embarqués est démontrée dans le chapitre 5).

– **Reading Module (ReadMod)**. *ReadMod* est en charge de la lecture de la politique de sécurité dans la Block RAM dédiée et de l'extraction des paramètres de sécurité à transmettre au *Checking Module*. Pour un seul mot de 32 bits, un buffer de lecture est rempli en 1 cycle d'horloge puis les paramètres sont transmis au *Checking Module* en 1 cycle supplémentaire.
– **Checking Module (CheckMod)**. Pour une implémentation simple, le *Checking Module* va vérifier deux paramètres : l'accès en lecture-écriture et le format grâce à un ensemble de comparateurs et une fonction de test (module *Test =! 0* dans la figure 3.10).

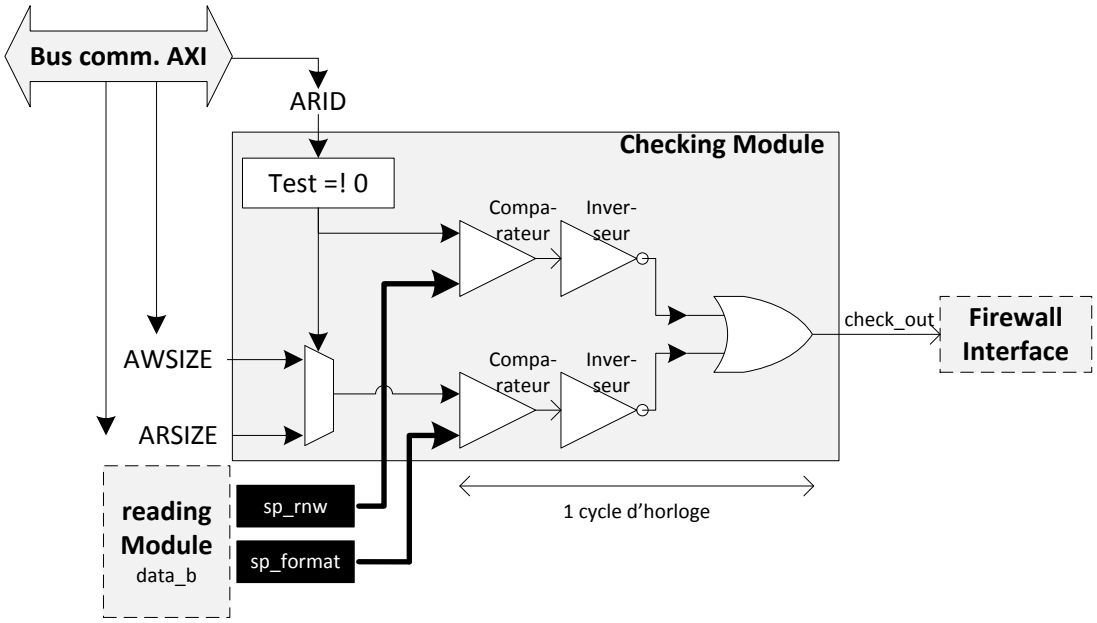


FIGURE 3.10 – Implémentation du Checking Module

Premièrement, il y a un test préliminaire sur la valeur du signal *ARID* extrait du bus : si celui-ci est non-nul, la donnée qui est en train d'être vérifiée est associée à une lecture, sinon ($ARID = 0$), il s'agit d'une écriture. La sortie de cette fonction de test sert ensuite à contrôler les deux paramètres :

– Il est utilisé en entrée $select$ d'un multiplexeur pour définir le signal contenant la valeur du format (*ARSIZE* et *AWSIZE* contiennent la valeur du format pour une lecture et une écriture, voir [ARM 2012]).
– Ensuite les deux paramètres (accès en lecture-écriture et format) sont comparés avec les valeurs *sp_rnw* et *sp_format* extraites de la politique de sécurité.

Comme les comparateurs sont instanciés en parallèle (voir figure 3.10), tous les paramètres sont analysés en même temps. Sur la figure 3.10, les inverseurs et la porte OU produisent un signal global représentatif de l'état des contrôles effectués par le *Checking Module* : si les entrées d'au moins un comparateur sont différentes, le

signal de sortie *check_out* est nul ; dans le cas où les entrées de chaque comparateur sont égales, la transaction est validée par la politique de sécurité associée, le signal *check_out* est à sa valeur haute.

– **Finite State Machine (FSM)** : la machine à états finis qui gère le fonctionnement de chaque pare-feu.

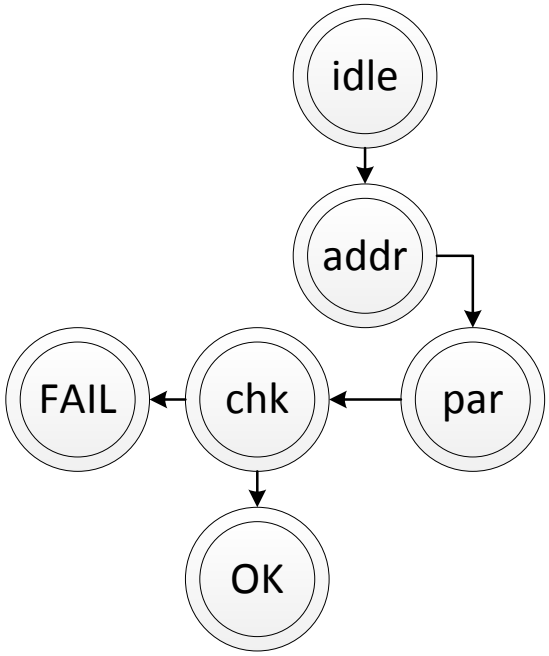


FIGURE 3.11 – Machine à états associée au module Security Builder

La machine à état de la figure 3.11 représente le comportement simplifié du module *Security Builder* intégré au sein d'un *Local Firewall* (voir figure 3.7). Elle contient les états principaux de ce module (sans détail du procédé en cas d'erreur ni les aspects cryptographiques) :

– État *idle*. Le module *Security Builder* est en attente d'une nouvelle donnée à analyser en provenance du module *Firewall Interface*.
– État *addr*. Permet de récupérer l'adresse Block RAM où est localisée la donnée.
– État *par*. Il s'agit de l'étape de lecture de la politique de sécurité.
– État *chk*. Il s'agit de l'étape de vérification des paramètres effectuée en 2 cycles : 1 cycle pour le pré-test et 1 cycle pour la partie combinatoire pure (comparateurs et portes). Si les paramètres concordent avec la politique, on passe en état *OK*, sinon en état *FAIL*.

Pour avoir un fonctionnement du pare-feu complet, il faut rajouter l'étape finale qui est la transmission du signal *check_out* au sous-bloc *Firewall Interface* : si les contrôles effectués concordent avec la politique de sécurité, les signaux sont transmis en synchronisation avec les données ; dans le cas contraire, les données ne sont pas transmises (on verra ultérieurement comment empêcher une transmission de

données sans pour autant bloquer le fonctionnement du système). La vérification d'un mot de 32 bits prend 4 cycles d'horloge. Par conséquent, le contrôle de N mots de 32 bits, dans une architecture non pipelinée, est définie par l'équation suivante :

$$latency(N) = N * 4 \tag{3.3}$$

Dans le cas où le *Security Builder* est implémenté sous forme d'une architecture pipelinée à m étages, la latence suit l'équation :

$$latency(N) = 3N - m(m-2) \tag{3.4}$$

Comme précédemment, il s'agit d'un raisonnement préliminaire à une étude approfondie où l'objectif serait d'améliorer les performances.

3.3 Cryptographic Firewall

Le *Cryptographic Firewall* est un pare-feu un peu plus particulier : il se branche au niveau du contrôleur de la mémoire externe (entre le bus de communication et la mémoire externe) et propose, en plus des fonctions de sécurité telles que celles implémentées dans le *Local Firewall* (section 3.2), des fonctions cryptographiques qui permettent de protéger des données en termes de confidentialité et d'authentification.

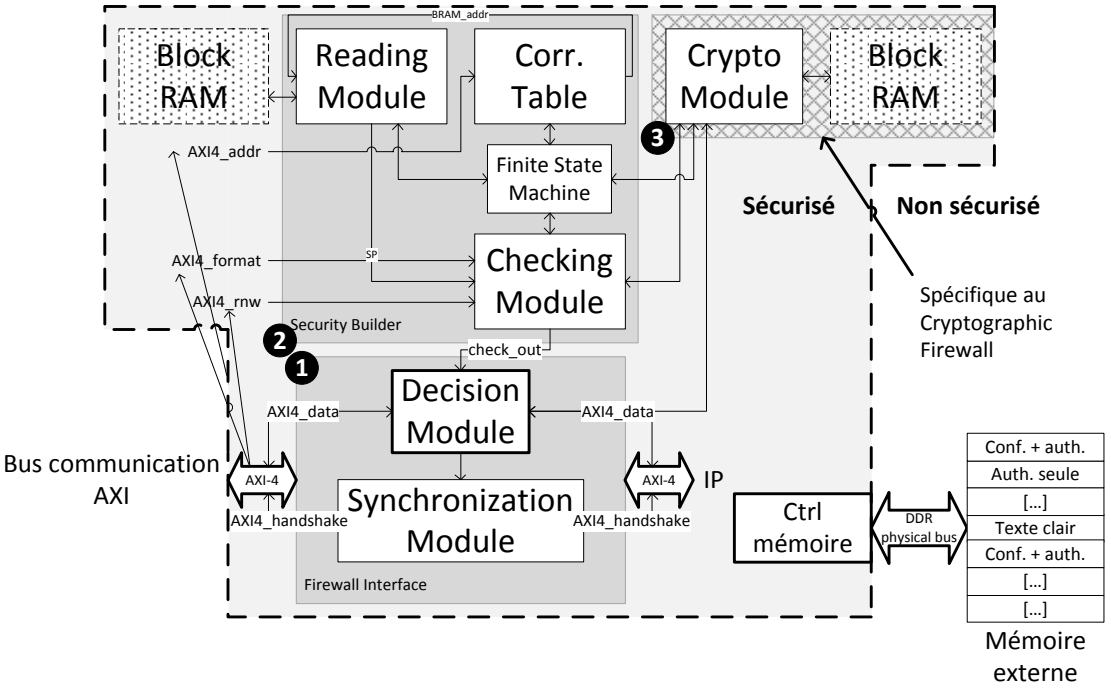


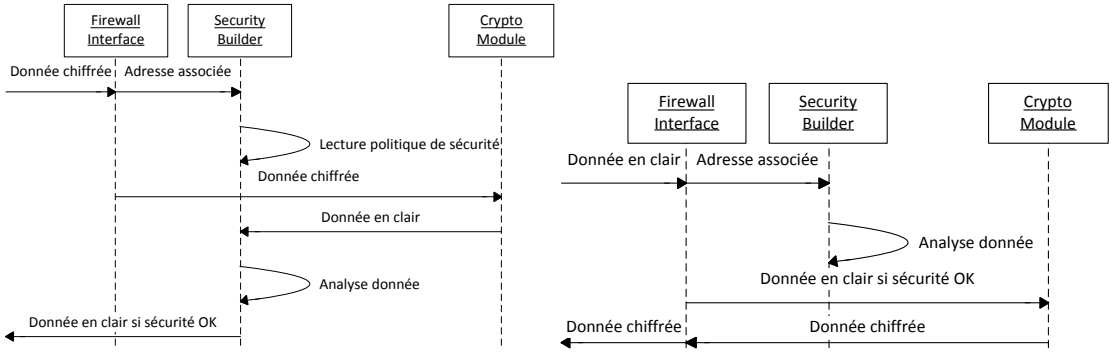
FIGURE 3.12 – Structure d'un Cryptographic Firewall

3.3.1 Fonctionnalités communes

A partir de ce point, la seule différence entre le *Local* et le *Cryptographic Firewall* est le *Security Builder*. Étant donné que les deux types de pare-feux ne gèrent pas les mêmes paramètres (voir la section 3.1.4), les *Reading Module* et *Checking Module* ont une structure différente. De plus, le chemin de données est modifié pour pouvoir y intégrer le module dédié aux fonctions de confidentialité et d'intégrité implémenté dans le *Cryptographic Firewall* :

– S'il s'agit d'une lecture : la première étape est de récupérer la politique de sécurité associée à la donnée analysée (issue de la mémoire externe), elle sera traitée par le bloc cryptographique (déchiffrement et vérification de l'authenticité si nécessaire) avant d'être analysée par le *Security Builder* (vérification des droits d'accès, format...). On pourrait inverser les étapes d'analyse et de déchiffrement pour gagner en latence : on laisse le processus dans cet ordre au cas où, dans des implémentations ultérieures, la valeur de la donnée devrait être analysée (dans ce cas, la donnée doit forcément être déchiffrée avant que la politique associée soit vérifiée).
– Dans le cas d'une écriture, le chemin de données emprunte le *Security Builder* avant de traverse le bloc de traitement cryptographique.

D'après le schéma 3.12, le *Cryptographic Firewall* a une structure proche d'un *Local Firewall*. Le bloc de chiffrement est adapté en conséquence pour fournir des aspects cryptographiques en plus des contrôles proposés par un *Local Firewall*. L'implémentation du bloc de chiffrement sur les chemins de données est de la forme suivante :



(a) Lecture (b) Écriture

FIGURE 3.13 – Chemins de données dans un Cryptographic Firewall

La seule différence avec le chemin de données d'un *Local Firewall* est le passage par le *Crypto Module* pour les opérations de chiffrement et de déchiffrement.

3.3.2 Choix des modes de confidentialité et d'intégrité

Dans le cadre de cette contribution, les mécanismes proposés doivent être, dans la mesure du possible, efficaces en termes de latence ajoutée : les implémentations doivent minimiser la latence de ces fonctions. Par conséquent, la protection totale des mémoires externes avec des moyens cryptographiques n'est pas conseillée. Il faut donc étudier les différentes possibilités qui permettent d'avoir des propriétés de confidentialité et d'authentification "à la carte" dans un seul bloc matériel avec la latence la plus faible qui soit et en gardant une certaine robustesse en termes de sécurité proposée. D'après les différents algorithmes présentés dans la section 1.3, on considère les quatre solutions suivantes :

– AES + MD5. La solution la plus simple est de prendre le standard AES pour la confidentialité et d'y ajouter une fonction de hachage de type MD5 pour l'intégrité. MD5 est un algorithme assez ancien (développé en 1992) mais toujours utilisé dans certaines applications.
– AES + SHA-2. Inventé à la suite de découvertes de failles dans MD5. D'après plusieurs rapports [6], d'autres failles ont été découvertes dans ces algorithmes, ce qui les rend très vulnérables et a amené à chercher d'autres implémentations plus robustes.
– AES + SHA-3. Le NIST [7] a lancé en 2007 un concours pour définir SHA-3, un successeur à SHA-2. Le vainqueur de ce concours a été annoncé en octobre 2012 : il s'agit de Keccak [8] développé par STMicroelectronics et NXP. Pour les résultats, on utilise des éléments de la base de données de l'outil ATHENA développé à l'université George Mason [9] en prenant des résultats pour la famille de FPGA Xilinx Virtex-6.
– AES-GCM. Une utilisation détournée de la fonction AES de base qui est un mode de chiffrement authentifié. La confidentialité est assurée par le chiffrement du texte clair en mode CTR et l'authentification est effectuée par le calcul d'un MAC fondé sur le hachage universel. [Crenne 2011] propose une comparaison des différents modes d'AES et une étude approfondie du mode AES-GCM.

6. http://eprint.iacr.org/2004/207.pdf
7. National Institute of Standards and Technology
8. http://keccak.noekeon.org/
9. http://cryptography.gmu.edu/athenadb/fpga_hash/table_view

Pour choisir la fonction la plus appropriée à notre problème, deux métriques sont envisagées : la latence sur un bloc de données de 32-bits ainsi que le débit du bloc cryptographique. Les résultats associés aux différentes options sont présentées dans le tableau 3.1.

	Latence (Nombre de cycles)	**Débit**
AES + MD5 [Järvinen *et al.* 2005]	90	jusqu'à 725 Mbits/s
AES + SHA-2 [Chaves *et al.* 2006]	74	jusqu'à 1,8 Gbits/s
AES + SHA-3 (Keccak)	25	environ 30 Gbits/s
AES-GCM [McGrew & Viega 2004]	22	30 Gbits/s

TABLE 3.1 – Comparatif des modes disponibles pour le bloc cryptographique

D'après le tableau 3.1, la meilleure option est l'algorithme AES-GCM (environ quatre fois plus rapide que les solutions avec fonction de hachage). La structure de ce mode est décrite dans la figure 3.14. D'après les premiers résultats, Keccak (le standard pour SHA-3) a l'air de donner des résultats très intéressants (et semblables à la solution AES-GCM) mais trop récent pour pouvoir en faire une étude approfondie.

La contribution apportée dans cette thèse propose des services cryptographiques flexibles basés sur un algorithme AES-GCM [McGrew & Viega 2004]. D'après le schéma de la figure 3.14, un unique module a été implémenté pour produire différents modes (« confidentialité et authentification » , « authentification seule » ou « texte clair » : les ports des multiplexeurs et démultiplexeurs sont implémentés de manière à ce que les paramètres des politiques de sécurité qui définissent l'activation de la confidentialité et de l'authentification (*Cmode* et *Imode*) permettent d'emprunter les différents chemins de données qui produisent le texte chiffré et le tag (voir un by-pass total si aucune option cryptographique n'est activée) ; les clés utilisées pour le chiffrement-déchiffrement sont extraites des politiques de sécurité et envoyées par le *Reading Module* (tout comme les informations ayant attrait aux MACs).

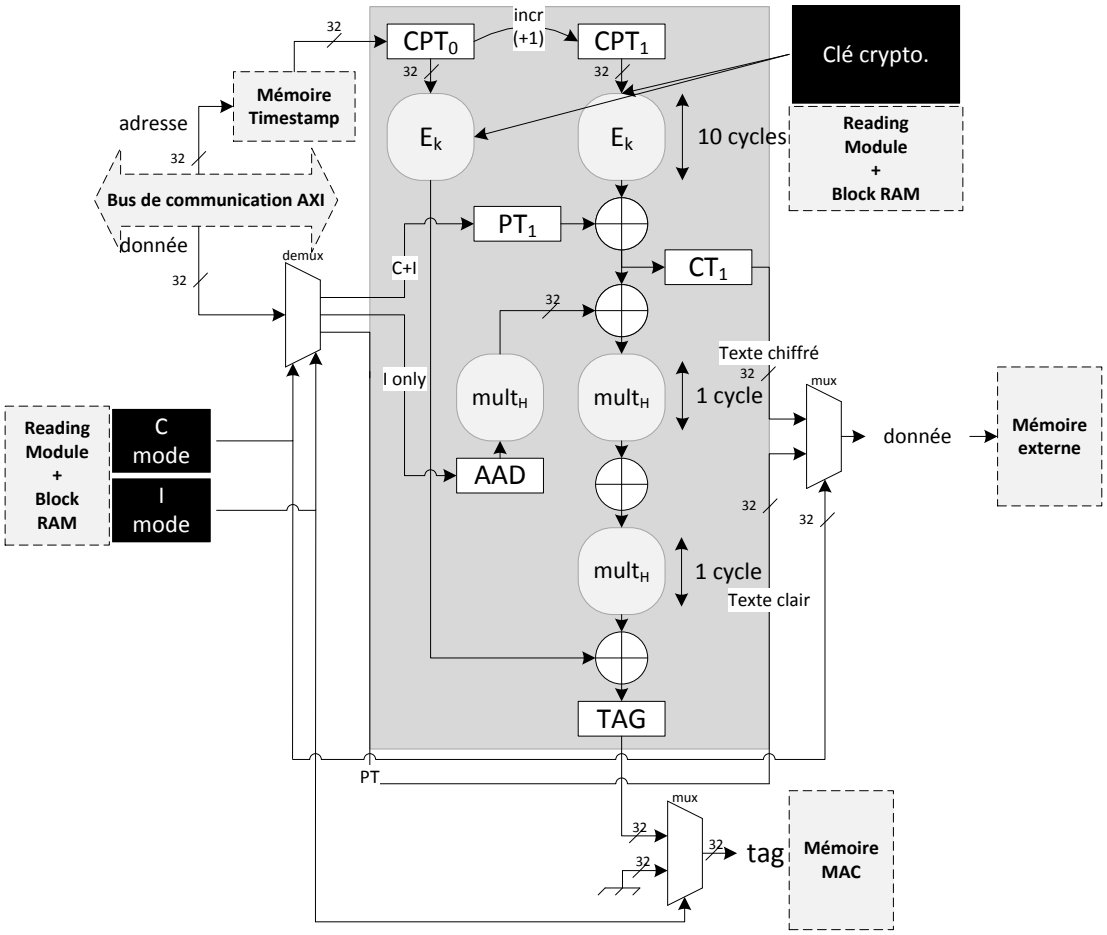


FIGURE 3.14 – Cryptographic Module : AES-GCM

Dans l'algorithme AES-GCM, la confidentialité est exécutée par le chiffrement du texte clair. On utilise la fonction AES en mode CTR dans les blocs E_k pour générer les keystreams (grâce à la clé extraite de la politique de sécurité stockée dans une mémoire interne de confiance) ; une valeur d'horodatage (ou *timestamp*) est utilisée en entrée du circuit AES-GCM pour éviter les attaques par rejeu. Enfin, une opération ou exclusif est réalisée entre le keystream et le texte clair pour produire le texte chiffré qui peut alors être transmis au contrôleur de la mémoire externe. L'authentification est exécutée par le calcul d'un MAC fondé sur le hachage universel. La donnée AAD peut également être authentifiée sans être chiffrée (utilisée dans le mode où la politique de sécurité est en authentification seule). Le choix entre les différentes options (confidentialité et authentification, authentification seule, texte en clair) est réalisé par extraction des paramètres depuis les politiques de sécurité : des multiplexeurs et démultiplexeurs sont implémentés pour que le chemin de données produise le texte chiffré et-ou le tag en fonctions des paramètres *C mode* et *I mode*.

Dans la figure 3.14, le chiffrement d'une donnée de 32 bits (E_k utilise une clé de 128 bits et un vecteur de 128 bits contenant la donnée sur 32 bits suivie de zéros), est effectué en 10 cycles d'horloge et l'authentification est accomplie en 2 cycles (passage à travers 2 multiplieurs de Galois $MULT_H$ [Crenne *et al.* 2011a, Crenne *et al.* 2011b]). La latence globale pour un ensemble de N données de 32 bits protégées en confidentialité et en authentification suit l'équation suivante :

$$latency(N) = 10 + (10 + 2) * N \tag{3.5}$$

En utilisant l'algorithme AES-GCM, les pare-feux fournissent des services cryptographiques à faible latence tout en gardant la flexibilité du choix des modes de confidentialité et d'authentification. En particulier pour l'authentification, AES-GCM prend 2 cycles d'horloge là où l'implémentation de fonctions de hachage telles que MD5 (respectivement SHA-2) prend 64 (respectivement 80) cycles d'horloge pour la même opération. Le tag produit par le bloc AES-GCM n'est pas chiffré, il est stocké dans une mémoire considérée comme étant de confiance (type Block RAM). Étant donné que les Block RAM sont des mémoires à deux ports, un port est laissé vacant pour des opérations de mise à jour des paramètres de sécurité en cas d'attaque. Le principal problème de cet algorithme est le stockage sécurisé des données cryptographiques (clés, horodateurs et tags) : une étude de faisabilité sur le stockage de ces paramètres est donnée dans le chapitre 5.

3.4 Politiques de sécurité : formulation et choix matériels

Chaque pare-feu (*Local* et *Cryptographic*) possède sa propre mémoire Block RAM (quelques dizaines d'octets suffisent à stocker les politiques associées à un pare-feu) qui contient les politiques de sécurité ; cette mémoire est directement connectée au *Reading Module*. Les politiques de sécurité sont indexées par la *Correspondence Table* définie dans la section 3.2.2.

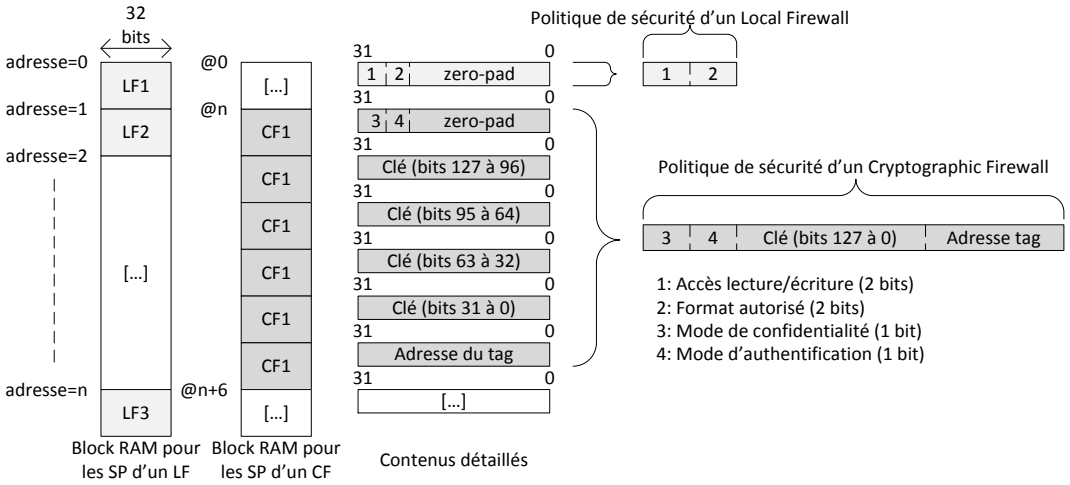


FIGURE 3.15 – Organisation des politiques de sécurité dans les mémoires Block RAM

La politique de sécurité associée à un Local Firewall est formulée sur un mot de 32 bits tandis que la politique associée à un Cryptographic Firewall tient quant à elle sur 6 mots de 32 bits (LF et CF dans la figure 3.15). La politique associée au LF contient les droits d'accès en lecture-écriture (bloc numéroté 1 dans la figure 3.15) et les droits sur le format. La politique associée au LF contient les modes cryptographiques (confidentialité, authentification...), la clé cryptographique ainsi que l'adresse du tag.

Chaque bloc est indexé par une adresse (par exemple, l'adresse 0) qui permet au *Reading Module* de savoir où lire le premier mot de la politique de sécurité à extraire pour une analyse des paramètres. Dans le cas d'un *Cryptographic Firewall*, un peu de logique est ajoutée pour permettre de combiner les différents mots de 32 bits tout en connaissant uniquement l'adresse du premier mot. La figure 3.15 présente l'agencement des politiques de sécurité dans les mémoires internes Block RAM pour les deux types de pare-feux.

Les Block RAMs ont des ports de données de 32 bits : par conséquent, une politique d'un *Local Firewall* tient sur 1 mot et une politique d'un *Cryptographic Firewall* tient sur 6 mots.

		Valeurs			
Champ	**Règle**	00	01	10	11
#1	Lecture autorisée	Non	Non	Oui	Oui
	Écriture autorisée	Non	Oui	Non	Oui
#2	Format	4 bits	8 bits	16 bits	32 bits

TABLE 3.2 – Description des paramètres en Block RAM

En termes de latence, lire une politique de LF prend 1 cycle et une politique de CF 6 cycles. Chaque pare-feu a sa propre mémoire Block RAM, séparées des mémoires contenant les timestamps et les MACs, et un port de la Block RAM est laissé vacant pour les mécanismes de mise à jour (écriture de nouveaux paramètres quand une attaque est détectée).

		Valeurs	
Champ	**Règle**	0	1
#3	Confidentialité	Non	Oui
#4	Authentification	Non	Oui

TABLE 3.3 – Description des paramètres cryptographiques

La structure des champs notés #1, #2, #3 et #4 est décrite dans les tableaux 3.2 et 3.3. Pour la confidentialité et l'authentification, les deux bits dédiés permettent de connecter correctement le bloc cryptographique à base d'AES-GCM pour effectuer les différents modes (confidentialité et authentification, authentification seule, texte clair). Ces paramètres sont stockés dans des mémoires de type Block RAM intégrées dans la puce FPGA où est également implémenté le système multiprocesseur : d'après le modèle de menaces décrit dans la section 3.1.2, les données contenues dans ces mémoires peuvent rester en clair étant donné que l'attaquant n'est pas considéré comme étant capable de s'attaquer directement à la puce FPGA.

3.5 Conclusion

Ce chapitre présente des mécanismes de sécurité qui peuvent s'intégrer dans une architecture multiprocesseur où le réseau de communication est basé sur un bus utilisant le protocole AXI. Ces mécanismes sont basés uniquement sur des fonctions matérielles utilisant les ressources du FPGA, il n'y pas d'altération de la couche logicielle du système embarqué final (que ce soit au niveau du système d'exploitation ou au niveau de l'application maître).

Les mécanismes embarquent des fonctions de surveillance mais aussi des fonctions cryptographiques qui donnent à l'utilisateur final la possibilité de définir des zones plus ou moins protégées (cryptographie et surveillance ou surveillance seule) selon les adresses physiques de l'espace mémoire global. Dans le cadre d'une implémentation plus complète avec un système d'exploitation de type Linux, les pare-feux doivent avoir connaissance d'une manière ou d'une autre de la méthode de traduction des adresses logiques en adresses physiques (c'est-à-dire de la structure de l'unité de gestion mémoire embarquée dans le processeur [Xilinx 2012a]).

4 Mises à jour en temps réel

Ce chapitre propose une solution qui permet de mettre à jour en temps réel les mécanismes de sécurité proposés dans le chapitre 3.

Sommaire

4.1 Problèmes liés à la solution proposée

4.1.1 Mise à jour du système

En utilisant les mécanismes de sécurité statiques proposés dans le chapitre 3, l'architecture multiprocesseur sur laquelle sont implantés les pare-feux est protégée contre des menaces s'attaquant à la mémoire externe ou au bus externe (entre le FPGA et la mémoire externe) sans pour autant chiffrer la totalité de la mémoire, ce qui aurait un impact important en termes de latence [Cotret *et al.* 2012a]. Malheureusement, il s'agit d'une solution statique où les politiques de sécurité ne peuvent pas être modifiées et où la sécurité ne peut donc pas s'adapter à un nouvel environnement (adaptation de la sécurité selon les attaques, modification du cahier des charges des transactions autorisées).

La mise à jour de la sécurité du système peut être réalisée par une reconfiguration dynamique partielle des pare-feux ou une reconfiguration complète en téléchargeant de nouveaux bitstreams dans le circuit FPGA. Toutefois, ces solutions ne conviennent pas dans la mesure où elles impliquent des modifications matérielles qui peuvent perturber le fonctionnement normal du système (disponibilité des composants pendant l'exécution de l'application...). Ce chapitre se concentre sur l'implémentation d'une solution basée sur des modifications « en temps réel » des Block RAMs sans avoir à télécharger de bitstream (complet ou partiel) pour mettre à jour la sécurité du système et sans interrompre son exécution.

4.1.2 Cahier des charges pour la flexibilité

En proposant une solution flexible, on veut pouvoir mettre à jour les politiques de sécurité afin de réagir en cas d'attaque sur l'une des IPs du système. Les mécanismes proposés doivent induire le minimum de perturbations possibles sur le système multiprocesseur où sont implantés les pare-feux. C'est-à-dire qu'ils doivent respecter les propriétés suivantes :

– Pas de blocage du système. Dans la mesure du possible, le fonctionnement du système ne doit pas être arrêté, cela pourrait nuire à l'utilisation qui en est faite. De plus, il faut aussi éviter que des données malveillantes fuitent au cours du processus de mise à jour des politiques de sécurité.
– Mise à jour à faible latence. Plus la sécurité du système est modifiée rapidement, mieux c'est.

– Modes plus ou moins sécurisés. Les pare-feux doivent proposer une hiérarchie de niveaux de protection pour pouvoir faire évoluer le système dans un état de sécurité plus ou moins permissif selon le contexte dans lequel il est placé. D'après les paramètres proposés dans le chapitre 3, on considère que les différents niveaux de sécurité seront définis par les droits d'accès en lecture-écriture.

4.2 Évolution des modes de sécurité

4.2.1 Description

Deux classes de composants sont définies en fonction de leur capacité à manipuler des informations confidentielles. Les IPs dites « critiques » (par exemple, les implémentations d'algorithmes cryptographiques) ne doivent pas révéler d'informations lorsqu'une attaque est détectée. Si un attaquant est capable d'extraire des clés et des signatures, cela serait une faille majeure pour le système embarqué. Dans ce cas, dès qu'une attaque est détectée, les IPs critiques doivent être isolées du reste du système dans un mode de type « erreur-quarantaine ».

Pour les IPs non-critiques, un niveau de sécurité intermédiaire est défini pour autoriser uniquement les lectures et non les écritures. Cette caractéristique permet de faire des sauvegardes de certains paramètres et données avant que l'IP infectée ne soit totalement isolée. Par conséquent, on se retrouve avec deux flots d'évolution des modes de sécurité en fonction des IPs :
– IP non critiques : « Situation initiale » ⇒ « Lecture seule » ⇒ « Erreur-quarantaine ».
– IP critique : « Situation initiale » ⇒ « Erreur-quarantaine ».

4.2.1.1 Situation initiale

C'est la situation la plus générique où une IP est protégée avec plusieurs niveaux de sécurité : par exemple, une partie de l'espace mémoire associé à l'IP en question peut être accessible en lecture et en écriture tandis qu'une autre partie est accessible en lecture seule.

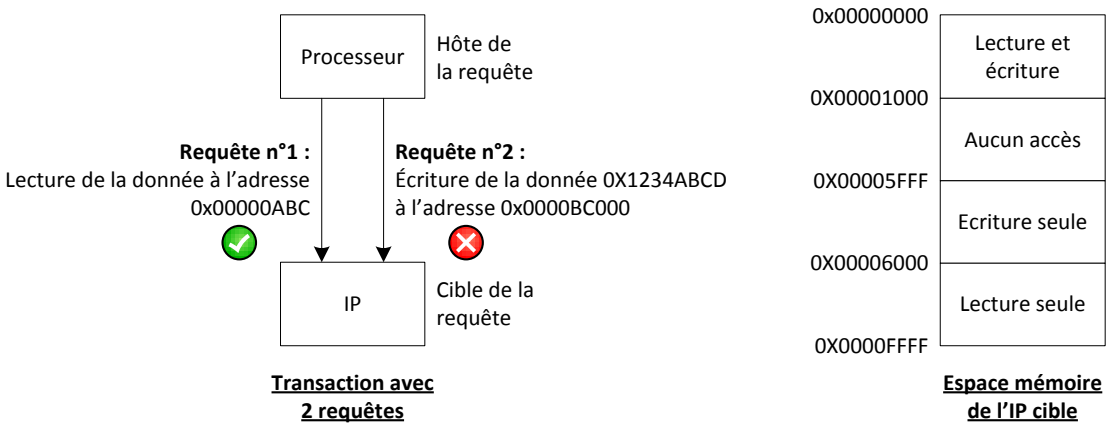


FIGURE 4.1 – Modes de sécurité en situation initiale

La figure 4.1 présente une de ces situations dans lesquelles on a plusieurs politiques et des requêtes qui sont autorisées tandis que d'autres sont refusées. On suppose qu'un processeur va émettre deux requêtes :
– **Requête n°1 :** Le processeur veut lire la donnée qui se situe à l'adresse $0x00000ABC$.
– **Requête n°2 :** Le processeur veut écrire la donnée $0x1234ABCD$ à l'adresse $0x0000BC00$.
Les différentes politiques associées à l'espace mémoire de l'IP sont présentées sur la droite de la figure (par exemple, l'espace d'adresses $[0x00000000; 0x00001000]$ est protégé en lecture-écriture). Dans cette figure, la requête n°1 est autorisée (l'adresse $0x00000ABC$ est gouvernée par une politique de sécurité qui autorise les lectures et les écritures ; la requête n°2 est bloquée car la politique associée n'autorise que les lectures.

4.2.1.2 Lecture seule

Ce cas est plus restrictif, les seules requêtes autorisées sont les lectures : cela est utile pour des IPs non critiques dont le contenu peut être lu pour une éventuelle sauvegarde (non détaillée ici).

La figure 4.2 présente une situation analogue à celle de la figure 4.1 à la différence que la totalité de l'espace mémoire de l'IP est gouverné par une politique qui autorise uniquement les lectures. Par conséquent, en utilisant les mêmes requêtes que dans le cas précédent, la requête n°1 est autorisée alors que la requête n°2 est bloquée.

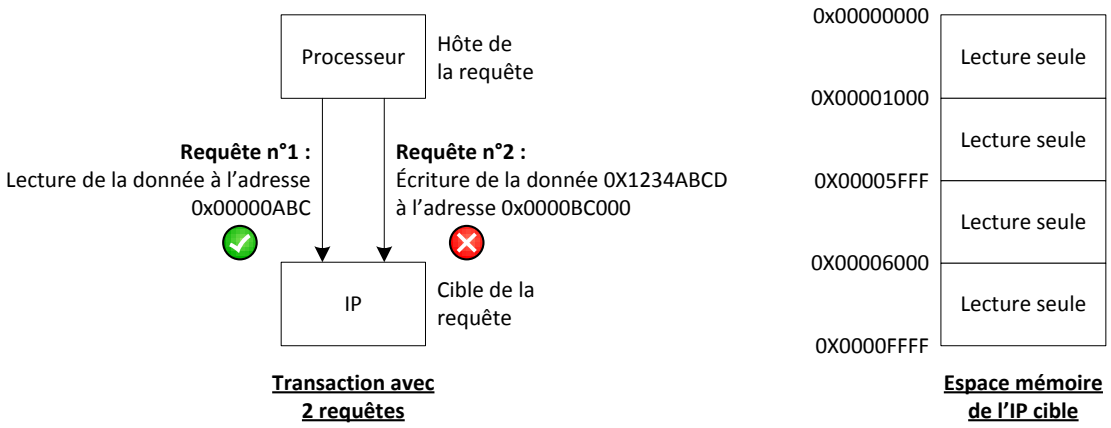


FIGURE 4.2 – Modes de sécurité en lecture seule

4.2.1.3 Erreur-quarantaine

Ce mode est assimilé à une mise en quarantaine où l'IP est verrouillée (en termes d'accès en lecture-écriture). Le but est de maintenir l'exécution du système active sur le processeur hôte de la requête même si une attaque est détectée de sorte que le système ne présente pas de dysfonctionnements (blocage de l'exécution de l'application...). L'IP est isolée des autres composants de l'architecture : si un autre élément essaie d'y accéder, il récupère un code d'erreur interprété par le processeur généraliste comme étant un code particulier.

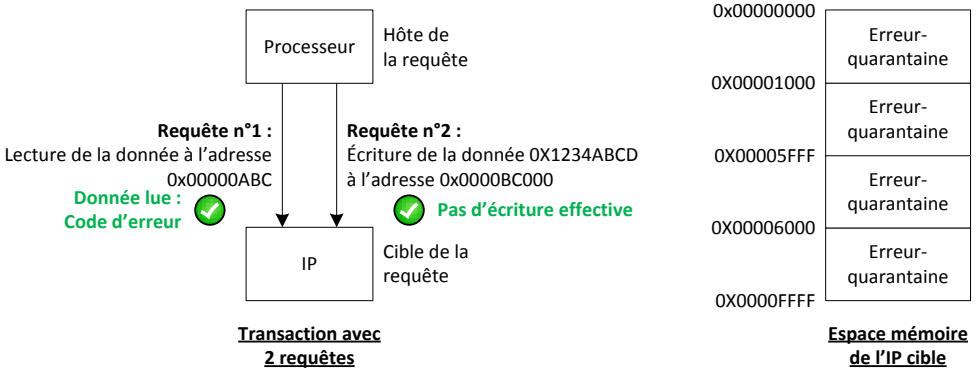


FIGURE 4.3 – Modes de sécurité en erreur-quarantaine

La figure 4.3 reprend la même situation que dans les figures 4.1 et 4.2 mais avec une politique « erreur-quarantaine » instaurée pour la totalité de l'espace mémoire de l'IP. Dans tous les cas, le processeur hôte de la requête reçoit une information selon laquelle la requête s'est bien déroulée même si ce n'est pas le cas en réalité : la donnée qui est sensée être lue est un fait un code préprogrammé et les écritures ne sont pas

effectives. Si le processeur généraliste est attaqué et que le pare-feu associé est mis dans le mode « erreur-quarantaine », un signal est émis par le *Local Firewall* afin que le système soit réinitialisé (le processus est détaillé ultérieurement dans ce chapitre).

4.2.1.4 Flux de mise à jour

En cas d'attaque (détectée par l'IP de surveillance décrite ultérieurement), le processeur de mise à jour sauvegarde le contenu actuel des politiques de sécurité associée au pare-feu attaqué dans une mémoire interne dédiée et applique un niveau de sécurité supérieur selon les deux flux définis précédemment. Par exemple, si le niveau actuel de sécurité est du type « lecture seule », le mode de sécurité suivant sera le mode « erreur-quarantaine » (voir figure 4.4).

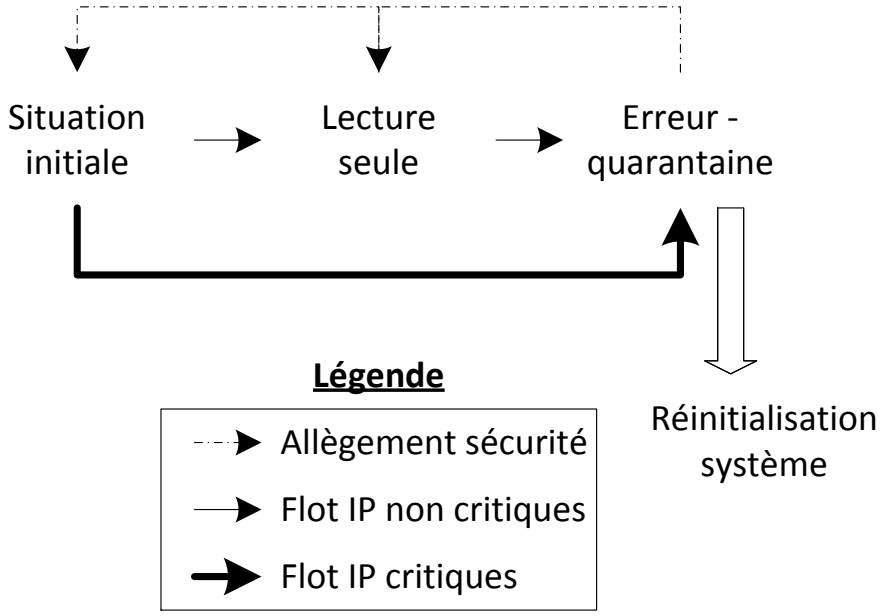


FIGURE 4.4 – Évolution des modes de sécurité

Ensuite, s'il n'y a pas d'attaque pendant une fenêtre de temps donné (définie par le concepteur des mécanismes de sécurité), le niveau de protection associé à une IP peut être fixé à un niveau plus permissif. Dans ce cas, sans événement lié aux attaques pendant un certain nombre de cycles d'horloge, le niveau de sécurité d'une IP peut passer de « erreur-quarantaine » à « situation initiale » (on considère que le processeur dédié à la mise à jour a connaissance des différents niveaux de sécurité des interfaces IP en lisant une mémoire dédiée). Ce procédé est détaillé dans les perspectives du chapitre 7.

4.2.1.5 Réinitialisation du système

Dans le cas le plus critique (une attaque est détectée alors qu'une IP est dans une configuration « erreur-quarantaine »), on considère que le système doit être complètement réinitialisé. Par conséquent, le bitstream original ainsi que la configuration des politiques de sécurité associée sont rechargées dans le composant FPGA. Le système redémarre alors depuis un état sûr.

4.2.2 Règles d'évolution

Quand un pare-feu doit être mis à jour avec une nouvelle politique de sécurité, il y a potentiellement deux zones à modifier :

– Le module *Correspondence Table* : grâce à ces différents registres, il permet de définir les différents espaces mémoire à protéger par des politiques de sécurité. Dans la suite de ce chapitre, on considère que la totalité de l'espace mémoire de chaque IP est protégé par une ou plusieurs politiques de sécurité : par conséquent, le seul composant devant être mis à jour est la mémoire contenant les politiques de sécurité.
– La mémoire Block RAM contenant les politiques de sécurité.

D'après la section 4.1.2, l'évolution des politiques de sécurité se fait uniquement sur les droits d'accès en lecture-écriture. Malgré tout, d'après la structure des Block RAMs pour les deux types de pare-feux présentée dans la figure 4.5, plusieurs informations sont stockées sur un seul mot de 32 bits.

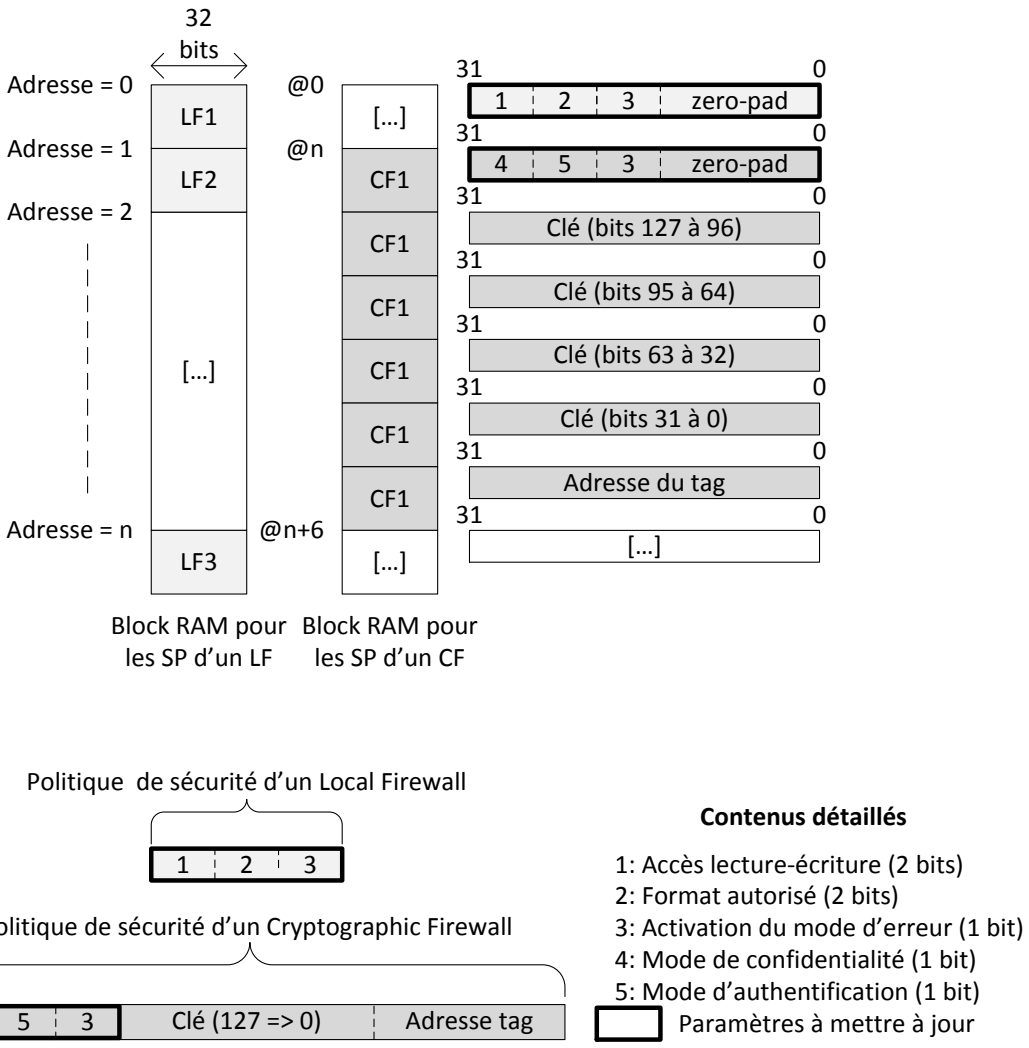


FIGURE 4.5 – Agencement des politiques de sécurité dans les Block RAM pour Local et Cryptographic Firewall

D'après la figure 4.5, les droits d'accès en lecture-écriture sont stockés dans un mot de 32 bits qui comprend également le format autorisé ainsi qu'un bit d'activation du mode erreur-quarantaine (décrit ultérieurement). Pour ce qui est des politiques associés aux *Cryptographic Firewalls*, on peut mettre à jour le premier mot de 32 bits contenant le bit d'erreur ainsi que les bits d'activation de la confidentialité et d'intégrité (champs 3, 4 et 5 dans la figure 4.5). Comme dit précédemment, les règles d'évolution s'intéressent uniquement aux droits d'accès en lecture-écriture : ce paramètre permet de mieux illustrer les évolutions des règles (contrairement aux modifications sur le format qui seraient plus compliquées à illustrer) et on considère également que ces droits d'accès se concrétisent sur tous les types d'échanges (que ce soit en interne ou en externe à destination de la mémoire DDR, contrairement aux bits d'activation des fonctions de confidentialité et d'intégrité). Étant donné que les bus de données des BRAMs sont sur 32 bits, la modification des droits d'accès en lecture-écriture passe par la réécriture de mots de 32 bits qui contiennent d'autres informations (mots entourés en gras dans la partie supérieure de la figure 4.5).

Étant donné que l'on considère uniquement la mise à jour des droits d'accès en lecture-écriture, on suppose que les attaques visent à provoquer des accès illégaux pour compromettre le fonctionnement du système multiprocesseur où sont implantés les pare-feux. Une fois que l'information d'activation de la mise à jour *recfgEn* (voir figure 4.10) a été levée, en accord avec la hiérarchie des niveaux de protection définie précédemment, le processeur dédié à la mise à jour calcule les nouvelles valeurs à écrire en Block RAM. Écrire un mot de 32 bits en Block RAM se fait en un cycle d'horloge [1]. Par conséquent, la reconfiguration de N politiques de sécurité dans un pare-feu se fait suivant l'équation suivante :

$$Durée\,de\,mise\,\text{à}\,jour = N\,cycles \tag{4.1}$$

4.2.3 Implémentation du code d'erreur

Le mode « code d'erreur » peut être interprété comme une mise en quarantaine de l'élément défectueux : l'IP est virtuellement déconnectée des autres éléments du système, les opérations de lecture et d'écriture ne sont pas autorisées.

1. http://www.xilinx.com/support/documentation/ip_documentation/axi_bram_ctrl/v1_03_a/ds777_axi_bram_ctrl.pdf

La structure interne d'un pare-feu de type *Local Firewall* est présentée dans la figure 4.6. Les principales fonctions réalisées par un *Local Firewall* sont les suivantes :

– Le module *Security Builder* est en charge de récupérer les politiques de sécurité et de les appliquer sur les données à analyser.
– Le module *Firewall Interface* gère essentiellement la communication et les tâches de synchronisation entre le bus système et l'IP rattachée au pare-feu.

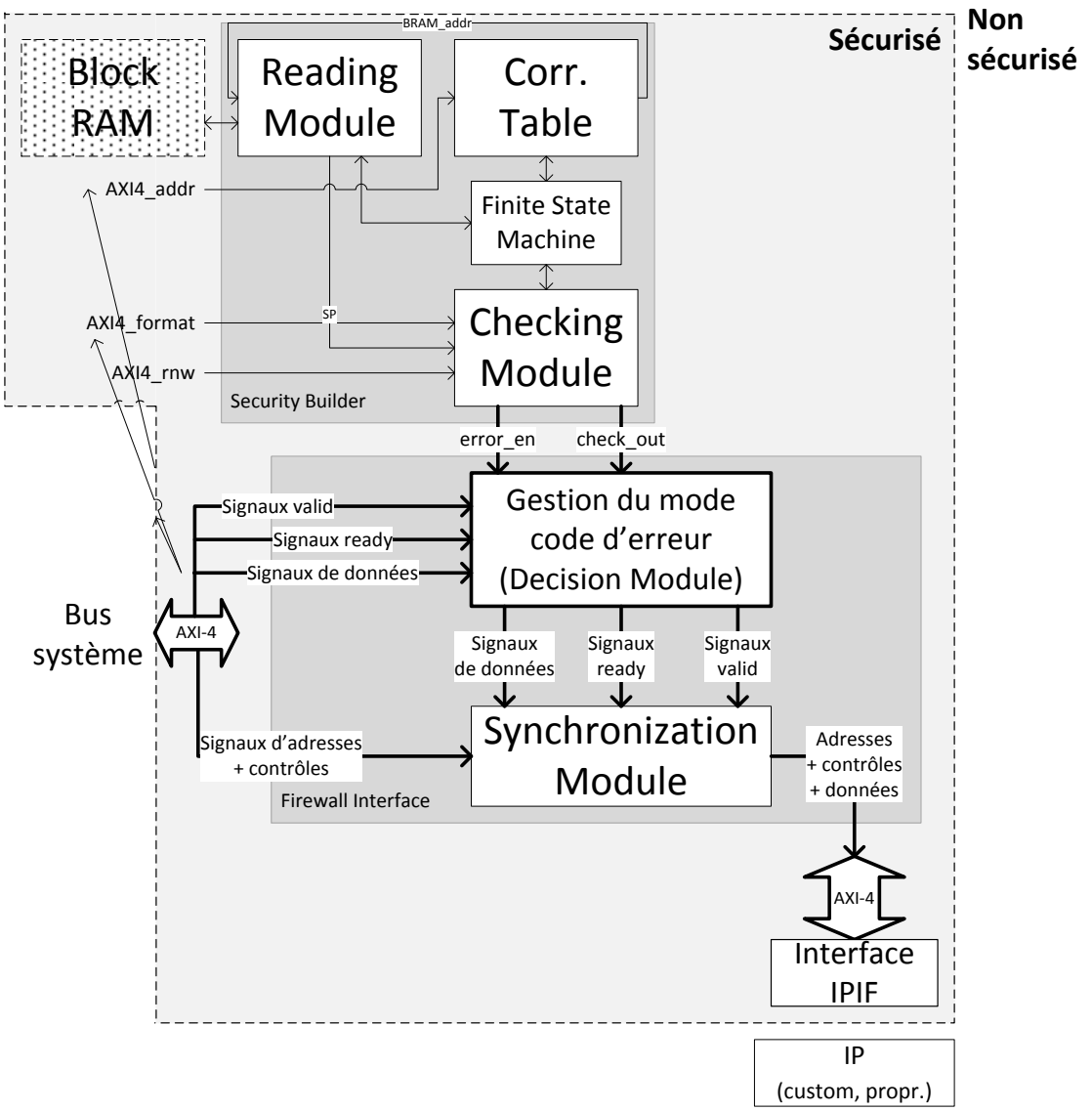


FIGURE 4.6 – Structure interne d'un Local Firewall

La figure 4.6 contient la structure de base d'un *Local Firewall*. Les modules ainsi que la Block RAM contenant les politiques de sécurité sont présentées. Dès lors que le mode d'erreur-quarantaine doit être activé, l'information concernant l'activation de ce mode est envoyée par le *Security Builder* en utilisant le signal *check_out* : ce signal est issu du *Checking Module* (voir figure 4.6) et représente l'état de sortie des contrôles (c'est-à-dire si les contrôles se sont bien déroulés ou non). Ce signal est transmis au module *Firewall Interface* qui utilise la propriété de « poignée de main » du protocole de communication AXI pour bloquer les lectures et les écritures : les différents canaux de ce protocole (adresse et donnée en lecture et en écriture, un cinquième canal contient des signaux de réponse aux écritures) possèdent chacun leur paire de signaux *valid-ready* (*valid* pour la source et *ready* pour la cible d'une transaction entre deux IP) [ARM 2012]. Ces paires de signaux doivent être à l'état haut en même temps pour que la transaction soit effective ; pour bloquer ces échanges, la méthode la plus simple consiste à maintenir les signaux de type *ready* à l'état bas. Les interfaces AXI d'entrée-sortie du module *Firewall Interface* (…$\_in$ pour les signaux d'entrée et …$\_out$ pour les signaux de sortie, voir figure 4.9) sont donc soumises à un contrôle par les pare-feux. Les chronogrammes représentant l'évolution des paires de signaux *valid-ready* pour les canaux d'écriture et de lecture sont donnés dans les figures 4.7 et 4.8.

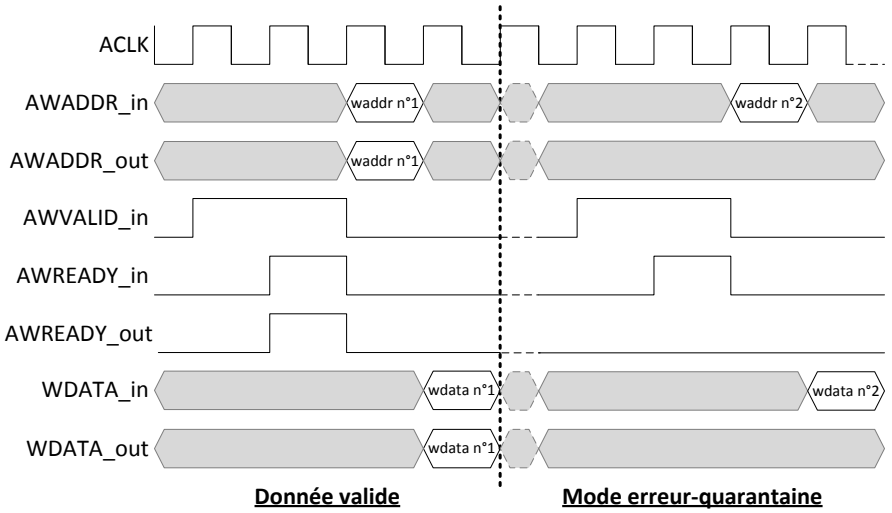


FIGURE 4.7 – Mode erreur pour les canaux d'écriture

Tout d'abord, il faut noter que les signaux *valid-ready* utilisés dans les diagrammes (*arready_in, awready_in*…) sont les signaux relatifs aux adresses des transactions : dans le protocole AXI [ARM 2012], il y a effectivement deux poignées de main : la première sur les adresses et la deuxième sur les données.

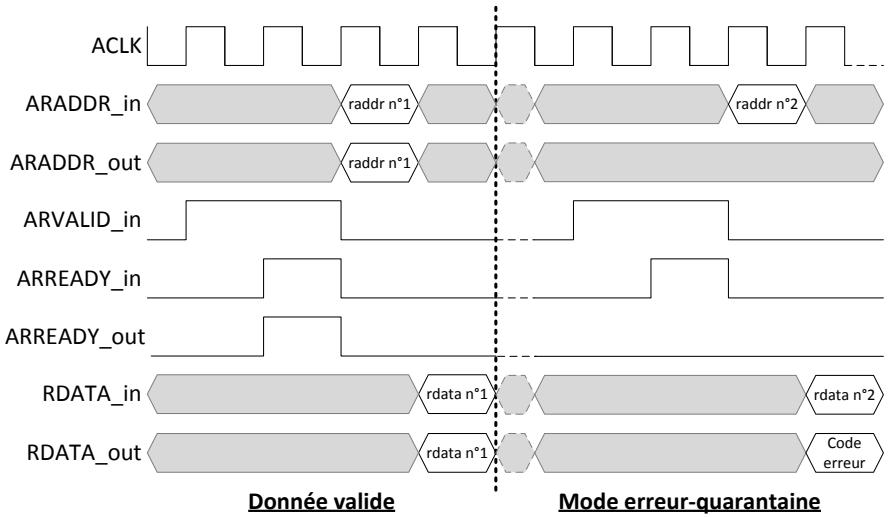


FIGURE 4.8 – Mode erreur pour les canaux de lecture

Par conséquent, étant donné qu'il y a une continuité dans le processus de ces deux poignées de main, il suffit de bloquer les adresses pour que les données ne soient pas transmises. Pour chaque chronogramme, la première moitié (en amont de la ligne en pointillés) montre le protocole poignée de main pour une transaction valide, la deuxième moitié (en aval des pointillés) montre ce même protocole avec le mode erreur-quarantaine activé. Que ce soit pour la lecture ou pour l'écriture, l'information est disponible quand les signaux *valid* et *ready* sont actifs en même temps (*awvalid_in* et *awavalid_out* pour le canal d'adresse en écriture). Lorsque le mode erreur-quarantaine est actif, les deux signaux de type *ready* en adresse (*awready_out* et *arready_out*) sont maintenus au niveau bas. Dans le cas d'une lecture, un code d'erreur défini par le concepteur du système est envoyé sur le canal de donnée en lecture (« code erreur » sur le chronogramme 4.8) pour ne pas bloquer le processeur source de la requête de lecture (en attente d'une donnée pour continuer son processus). Dans le cas d'une écriture, la donnée est tout simplement ignorée, il n'y a pas d'écriture effective dans l'élément cible de la transaction.

L'implémentation de ce mode particulier est détaillée dans la figure 4.9. Premièrement, le bit *check_out* est utilisé comme une entrée de multiplexeurs qui permettent de bloquer les signaux de poignée de main sur les canaux d'adresse : sur la figure 4.9, les deux premiers multiplexeurs vont donc renvoyer une sortie nulle (*awready_out* et *arready_out* à 0). Dès que les politiques sont mises à jour, le signal *error_en* est utilisé en entrée d'un troisième multiplexeur pour renvoyer la valeur du registre contenant le code d'erreur plutôt que le signal d'entrée sur le canal de donnée en écriture.

Quand le système fonctionne normalement, les signaux sont directement transmis entre l'entrée et la sortie de l'interface bus ($\ldots\_out \Leftarrow \ldots\_in$) ; dans le cas d'une attaque, les signaux de type *ready* sont maintenus à leur valeur basse et la donnée en lecture est remplacée par un code d'erreur (défini en dur dans un registre du module *Firewall Interface* du pare-feu).

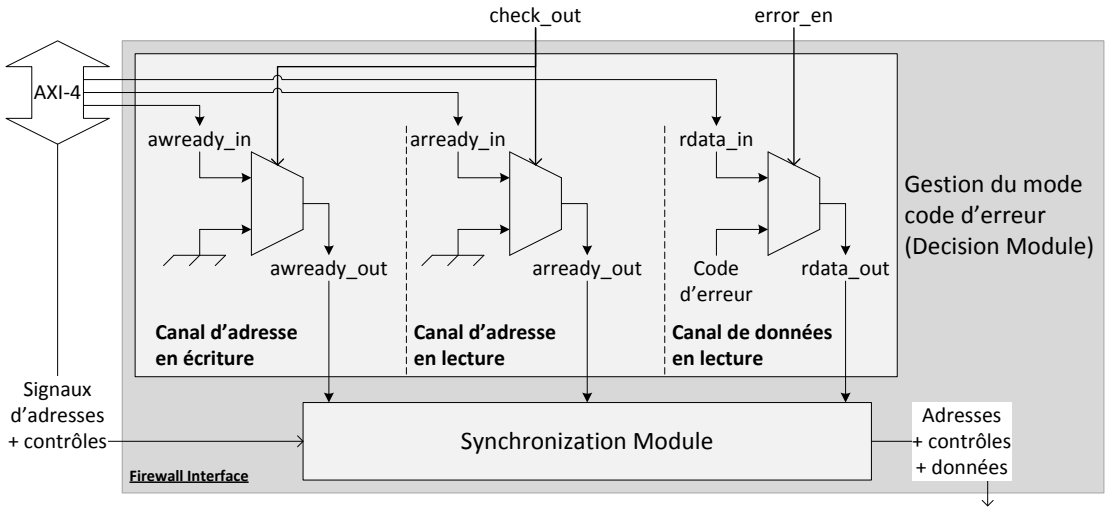


FIGURE 4.9 – Implémentation du mode erreur-quarantaine dans le module Firewall Interface

4.3 Solution flexible

Lorsqu'une attaque est détectée, les Block RAMs doivent être mises à jour avec de nouvelles politiques de sécurité pour garder un environnement d'exécution sécurisé. Tous les composants sont connectés à travers un bus de type AXI-Lite (*bus de mise à jour*), un processeur dédié (*processeur de mise à jour*) garde une trace de tous les événements liés à la sécurité du système cible.

4.3.1 Vue générale de la solution

L'architecture globale permettant d'avoir une solution flexible est proposée dans la figure 4.10. Tous les composants sont connectés par l'intermédiaire d'un bus de type AXI-Lite (bus de mise à jour) et gérés par un processeur dédié (processeur de mise à jour) qui sauvegarde les événements importants dans un fichier de log lisible par le processeur généraliste du système (horodateurs, attaques et déroulement de la mise à jour des politiques de sécurité).

Chaque pare-feu a une connexion avec l'IP de surveillance par l'intermédiaire d'un bus personnalisé. Tandis que le timer de log et l'IP de surveillance sont utilisés pour détecter et rapporter une attaque, les contrôleurs Block RAMs sont utilisés pour mettre à jour les politiques de sécurité.

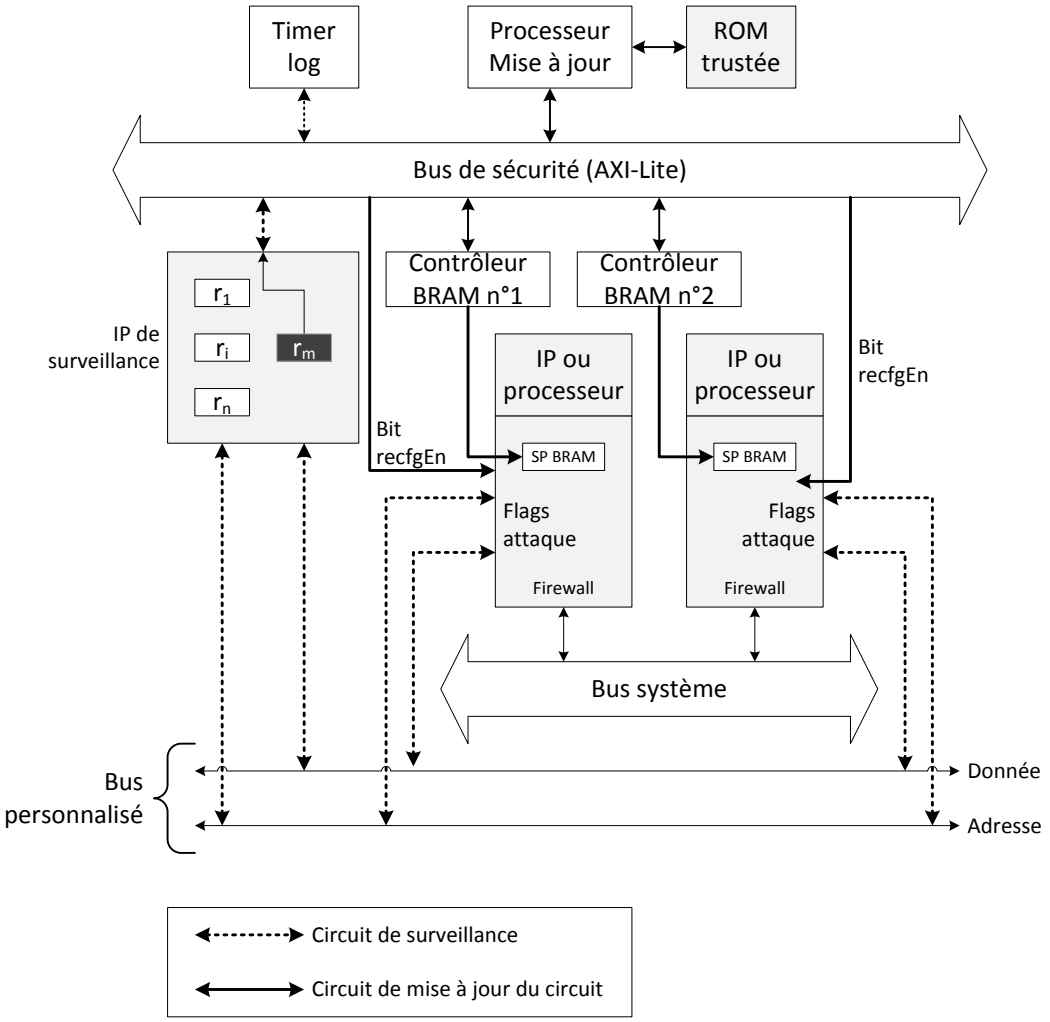


FIGURE 4.10 – Architecture globale de la zone de surveillance et de mise à jour des politiques de sécurité

4.3.2 Surveillance des attaques

La première phase réalisée par l'architecture proposée dans la figure 4.10 est la surveillance des événements d'attaque par une IP dédiée (partie gauche de la figure 4.10). Pour détecter les attaques, certains signaux (ou flags) sont extraits des pare-feux (*Local* et *Cryptographic*) :

– *checkFlag* (cF) : un des contrôles (lecture-écriture ou format) a échoué dans le *Checking Module*. Ce signal est analogue au signal *check_out* évoqué précédemment.
– *notFoundFlag* (nF) : tentative d'accès à une adresse inconnue (c'est-à-dire qui n'est pas répertoriée dans la *Correspondence Table*).
– *authenticationFlag* (aF) : la vérification de l'authentification échoue (flag spécifique au *Cryptographic Firewall* rattaché au contrôleur de la mémoire externe).

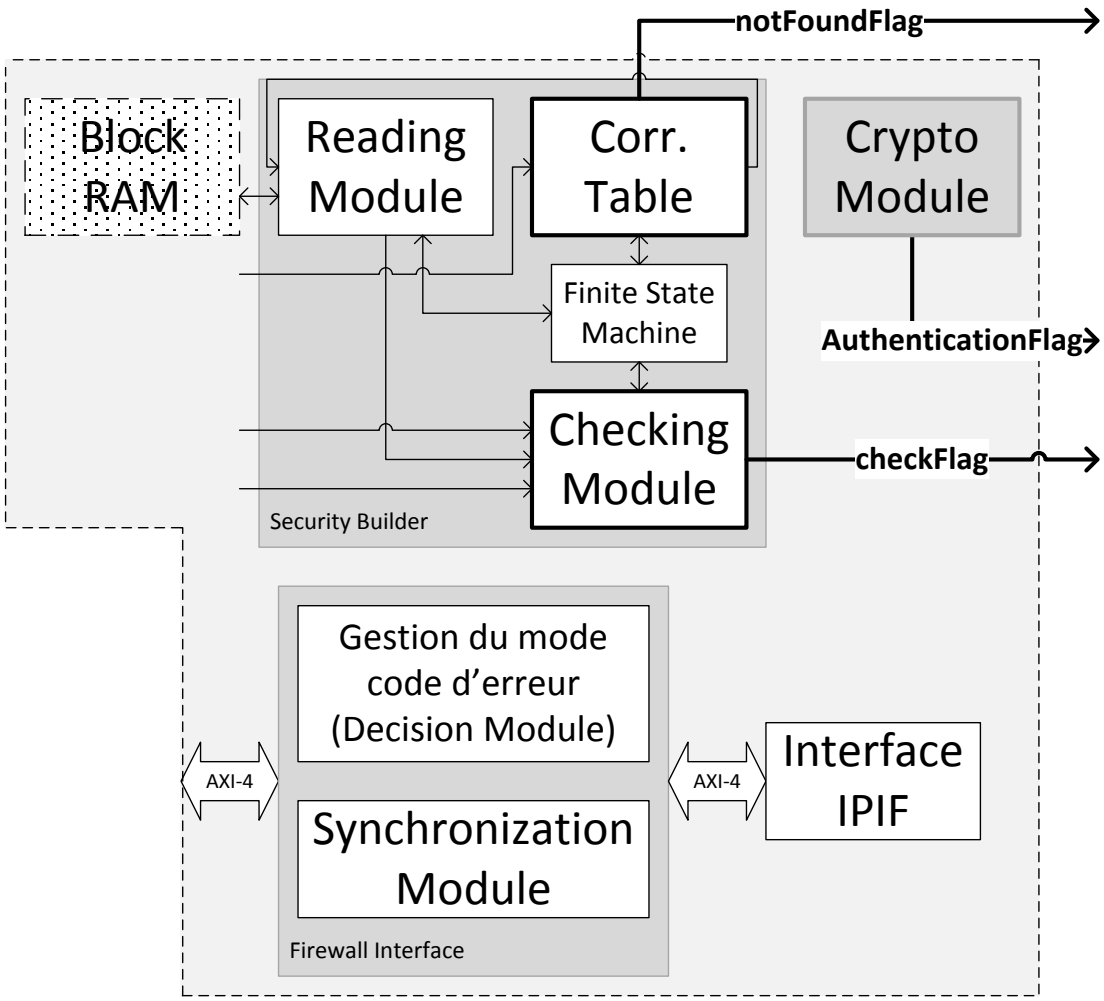


FIGURE 4.11 – Structure générique d'un pare-feu (LF-CF) avec détection des attaques par flags

La figure 4.11 représente une structure mixte de *Local Firewall* et de *Cryptographic Firewall*. Chaque flag est stocké sur un bit significatif dans les registres internes de l'IP de surveillance qui est composée d'autant de registres configurables qu'il y a de pare-feux dans l'architecture multiprocesseur protégée.

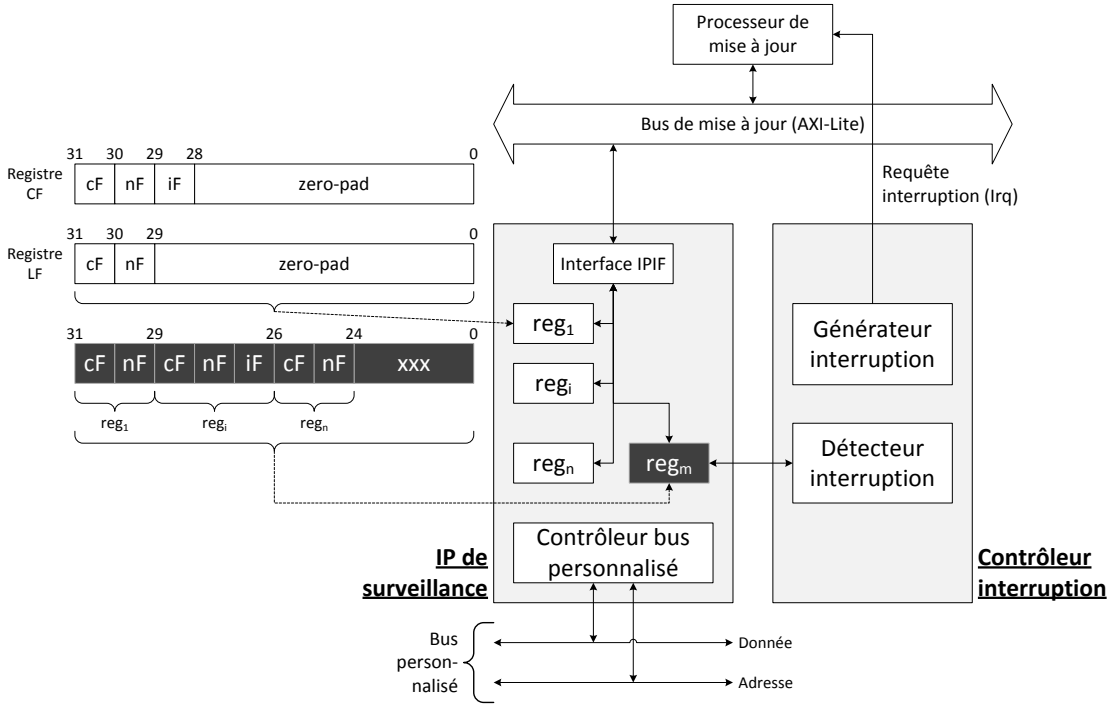


FIGURE 4.12 – Architecture de l'IP de surveillance et routine d'interruption

Dans la figure 4.12, les registres sont embarqués dans l'IP de surveillance. Chaque pare-feu transmet les informations de flags à son registre dédié dans l'IP de surveillance par l'intermédiaire d'un bus personnalisé (un bus d'adresse et un bus de donnée pour router les flags vers le bon registre). Chaque registre reg_i est sur 32 bits mais contient seulement 3 bits utiles (correspondant aux trois flags), le reste étant *zéro-paddé* (rempli de zéros). Le registre principal reg_m est une concaténation des bits utiles des différents registres individuels associés à chaque pare-feu (les reg_i). Étant donné que le registre principal reg_m est également sur 32 bits, on peut lui associer les bits utiles de 10 registres de pare-feu (comme chaque registre de pare-feu a 3 bits utiles, cela fait 30 bits nécessaires dans le registre principal) ; si c'est nécessaire, un deuxième registre principal, reg_{m2} est implémentée dans l'IP de surveillance (les 10 premiers pare-feux seront stockés dans reg_m et les autres dans reg_{m2}, voir figure 4.13).

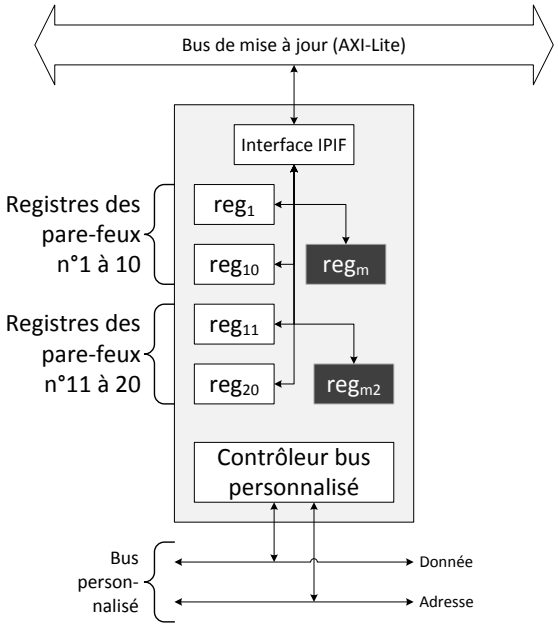


FIGURE 4.13 – Architecture de l'IP de surveillance avec 2 registres principaux

Ensuite, un contrôleur d'interruption récupère la valeur du registre principal et envoie une requête d'interruption au processeur de mise à jour dès qu'un des bits utiles du registre principal est nul (c'est-à-dire quand une erreur est détectée par l'intermédiaire d'un des flags, voir figure 4.12). Ce processeur de mise à jour (qui exécute la routine d'interruption) a plusieurs caractéristiques :

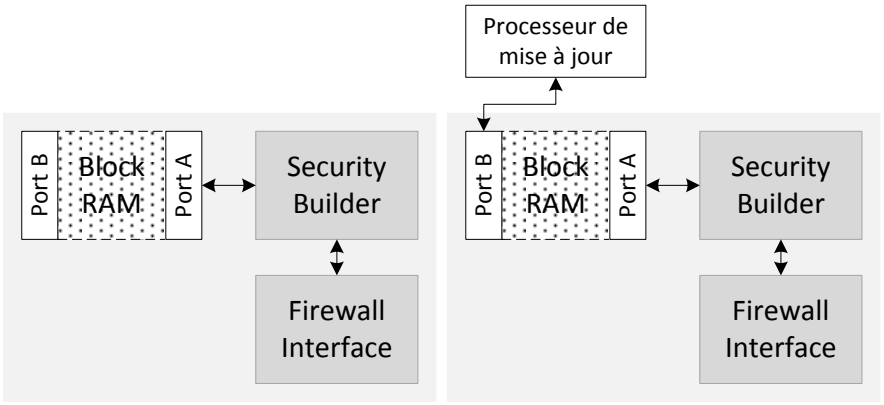
– Il connaît le contexte de sécurité du système : il sait quel mode de sécurité est associé à chaque pare-feu (voir section 4.2).
– Quand un pare-feu doit être mis à jour, les politiques sont placées dans un mode plus restrictif (voir section 4.2).

Au delà de ces aspects de surveillance, c'est également ce processeur qui est responsable de la mise à jour des politiques de sécurité instaurées dans les Block RAM associées à chaque pare-feu.

4.3.3 Mise à jour des politiques de sécurité

4.3.3.1 Modifications architecturales sur la protection statique

Par rapport à la protection statique proposée dans le chapitre 3, la structure du pare-feu est un peu modifiée.



(a) Structure avant modification (b) Structure après modification

FIGURE 4.14 – Modifications sur la protection statique

Dans la version statique (voir figure 4.14a), un seul port de la Block RAM est utilisé (connexion matérielle avec le pare-feu) ; pour la version qui peut être mise à jour (voir 4.14b), on utilise les deux ports de la Block RAM (connexion en matériel avec le pare-feu + connexion par un contrôleur BRAM avec le processeur de monitoring/mise à jour de la sécurité du système). Cette solution permet de mettre à jour les politiques de sécurité en utilisant un bus de communication dédié.

4.3.3.2 Propriétés du protocole de communication AXI

Les transactions dans le protocole AXI sont basées sur un mécanisme classique de type « poignée de main » avec des couples de signaux valid-ready.

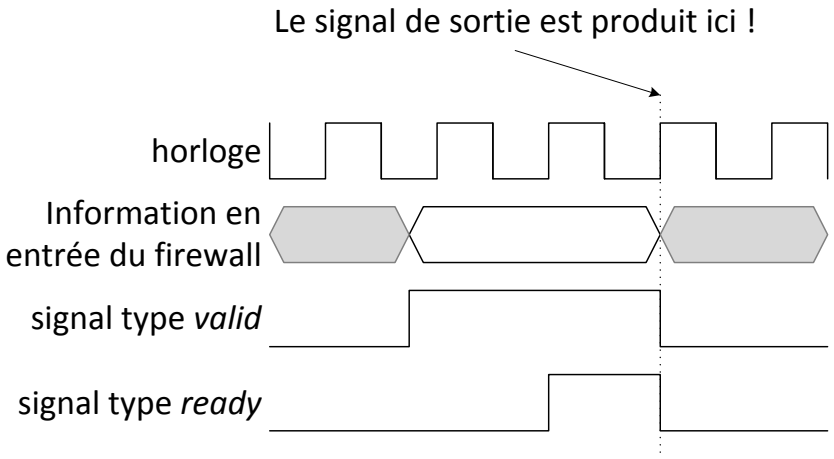


FIGURE 4.15 – Chronogramme du mécanisme poignée de main

Chaque canal du standard AXI (adresse et données pour la lecture et l'écriture ainsi qu'un canal de réponse en écriture [ARM 2012]) a sa propre paire de signaux *valid-ready*. L'information en sortie est valable uniquement quand les deux signaux sont à l'état haut (figure 4.15), il n'y a aucune transmission si un des deux signaux est à l'état bas. Toutes les transactions utilisant le protocole AXI se font en deux étapes. Il y a d'abord une poignée de main sur les signaux d'adresse puis une autre sur les signaux de données (le deuxième est dépendant du premier).

4.3.3.3 Description du protocole de mise à jour

En se basant sur le mécanisme « poignée de main » du protocole AXI, un mécanisme est implanté dans le module *Firewall Interface* (FI) de chaque pare-feu (*Local* ou *Cryptographic*) pour gérer les différentes étapes de mise à jour des politiques de sécurité. La machine à états correspondante est présentée dans la figure 4.16.

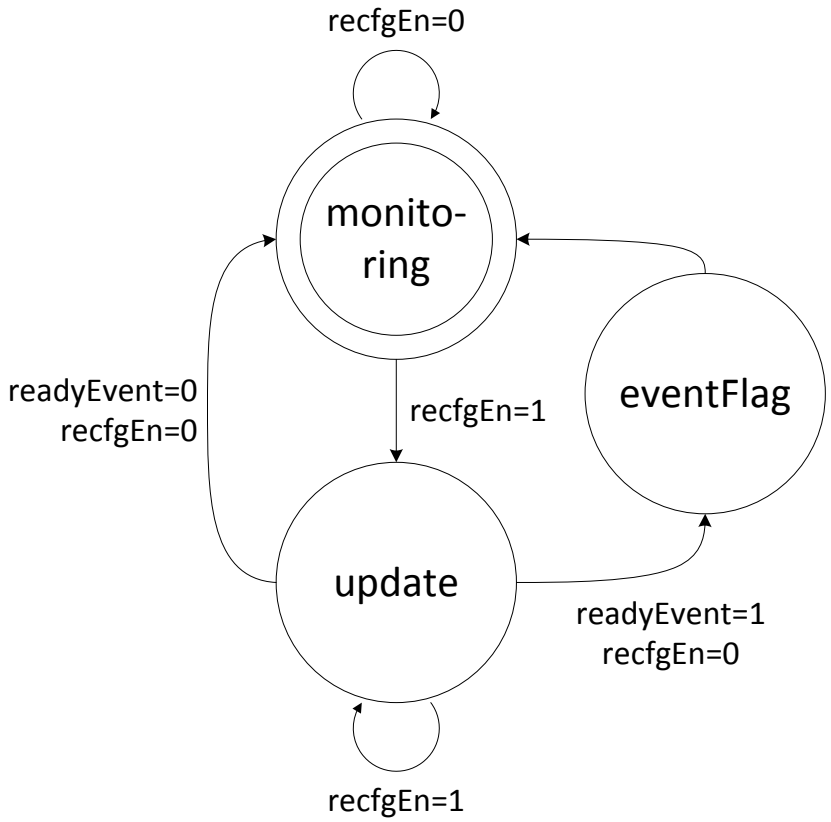


FIGURE 4.16 – Diagramme de la machine à états finis pour la mise à jour des politiques de sécurité

Quand une attaque est détectée dans un pare-feu, les sorties des signaux *ready* du module FI sont maintenues au niveau bas de sorte qu'aucune transaction ne puisse avoir lieu. Cette étape a pour but de « geler » les accès au bus pour se prémunir de tentative de corruption des données pendant la mise à jour de la sécurité du système. Ensuite, les différentes étapes de la mise à jour se déroulent en accord avec l'architecture simplifiée présentée dans la figure 4.17.

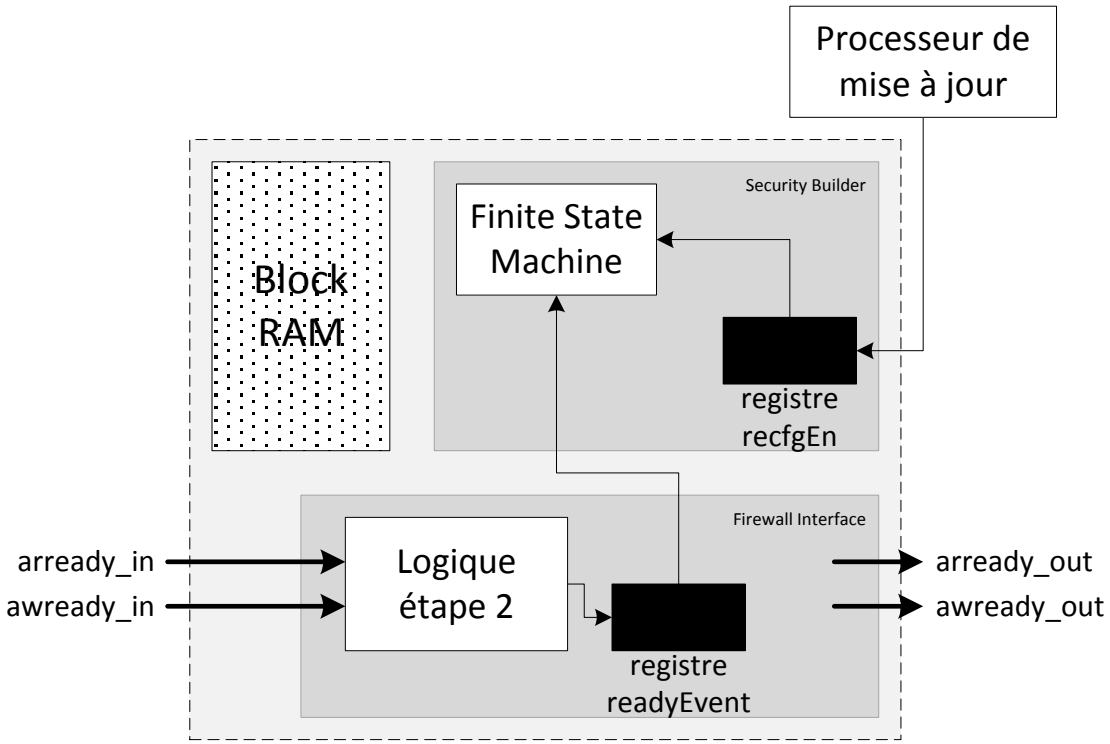


FIGURE 4.17 – Architecture partielle d'un pare-feu avec éléments de mise à jour

Tout d'abord, la mise à jour est activée par le processeur dédié à cette tâche en écrivant la valeur 1 dans le registre *recfgEn* qui est intégré dans le module *Security Builder* du pare-feu : la machine à états passe alors de l'état *monitoring* à l'état *update*. Ensuite, pendant que le processeur effectue logiciellement son processus, si un front montant parvient sur l'un des deux signaux *ready* (*arready_in* ou *awready_in*) qui sont bloqués en sortie le temps que la mise à jour soit effectuée (voir section 4.2.3, c'est-à-dire qu'on obtient *arready_out*,*awready_out* = 0 tant que *recfgEn* = 1), alors la valeur 1 est écrite dans le registre *readyEvent* embarqué dans le module *Firewall Interface*. Lorsque que la mise à jour est effectuée, le processeur dédié écrit la valeur 0 dans le registre *recfgEn* pour signifier la fin de la mise à jour et le relâchement des signaux de type *ready*.

Finalement, selon la valeur du registre *readyEvent*, la donnée qui est bloquée en entrée du module *Firewall Interface* est analysée si et seulement si $readyEvent = 1$ (c'est-à-dire s'il y a eu un front montant sur les signaux *ready* pendant la mise à jour) : ainsi, la donnée a été bloquée pendant la mise à jour mais est analysée avec les nouvelles politiques de sécurité qui ont été écrites par le processeur dédié à la mise à jour.

La latence globale du processus de mise à jour des politiques de sécurité embarquées dans les mémoires internes Block RAMs dépend du nombre de politiques de sécurité à mettre à jour. Globalement, le processus se déroule en cinq étapes :

– Copie des flags extraits des pare-feux dans l'IP de surveillance et blocage des pare-feux. Cette étape est réalisée en 1 cycle d'horloge.
– Exécution de la routine d'interruption. Une requête d'interruption est envoyée en 2 cycles.
– Calcul de la nouvelle configuration des politiques de sécurité. Il s'agit de la latence du programme exécuté par le processeur de mise à jour. L'algorithme de décision est basée sur une structure simple : connaissant l'état actuel des politiques à mettre à jour, on passe à la politique suivante telle qu'elle est décrite dans l'évolution des modes de sécurité décrit dans la section 4.2. Pour une implémentation de l'algorithme, la nouvelle configuration est calculée en 148 cycles. Néanmoins, d'autres algorithmes pourraient être considérés. Dans ce cas, le temps de réponse sera impacté par le temps nécessaire à la prise de décision.
– Mise en place de la nouvelle configuration. Cette partie dépend du nombre de pare-feux et du nombre de politiques à reconfigurer. Les détails de latence sont donnés ultérieurement.
– Réactivation de l'application principale. Désactivation du mode erreur-quarantaine en 1 cycle.

4.3.4 Interfaces utilisateur

On utilise des marqueurs de temps (ou *timestamps*) + les informations d'attaques pour avoir des fichiers de log qui permettent de garder une trace des événements liés à la sécurité : attaque détectée, mise à jour en cours, mise à jour terminée. Le fichier de log se présente sous la forme suivante :

```
log,t=400ns :
attaque détectée sur le Local Firewall n° 2.
log,t=410ns :
Processus de mise à jour enclenché...
log,t=410ns :
Remplacement de la politique actuelle par une politique de type
erreur-quarantaine.
log,t=450ns :
Mise à jour du Local Firewall n° 2 terminée !
log,t=470ns :
Code d'erreur DEADB10C détecté sur le Local Firewall n° 2.
log,t=470ns :
Erreur sur le canal de donnée en lecture du Local Firewall n° 2. Ne
pas traiter les données suivantes...
log,t=500ns :
Code d'erreur BADCAB1E détecté sur le Local Firewall n° 2.
log,t=500ns :
Erreur sur le canal de donnée en lecture du Local Firewall n° 2. Ne
pas traiter les données suivantes...
```

Comme dit précédemment, ce fichier (stocké dans une mémoire de confiance accessible par l'utilisateur du système) contient différentes informations extraites de l'architecture présentée dans la figure 4.10.

– Des marqueurs de temps, obtenus par le *log timer*.
– Les informations sur la détection des attaques sont extraites de l'IP de monitoring (lecture des registres contenant les flags des pare-feux).
– Le reste du processus dépend seulement du code source de l'application en elle-même.

4.4 Conclusion

Ce chapitre propose des extensions au chapitre 3 pour permettre d'avoir une sécurité flexible au sein d'une architecture multiprocesseur basée sur un bus de type AXI. La solution proposée permet de mettre à jour les politiques de sécurité embarquées dans les interfaces contenant les mécanismes de sécurité sans interrompre le fonctionnement du système et sans avoir de fuite de données contrefaites : dès qu'une donnée est détectée comme étant erronée, celle-ci est bloquée par le pare-feu correspondant.

Dans ce chapitre, les attaques sont enregistrées dans le registre principal de l'IP de surveillance. Quand il y a plusieurs attaques en même temps, la seule modification se situe dans le registre principal. On considère que le processeur de mise à jour est capable de gérer les routines d'interruptions de chaque scenario. Un système multiprocesseur sécurisé avec les pare-feux résiste à ces différents scenarii :

- **1 attaque sur un pare-feu.** Le scenario le plus simple, le composant rattaché au pare-feu qui est en train d'être mis à jour attend que la mise à jour des paramètres de sécurité soit terminée. Pour N politiques à mettre à jour, la latence est de N cycles. Ici, il s'agit seulement de la latence de la mise à jour du pare-feu en lui-même.
- **1 attaque sur N pare-feux.** Quand N pare-feux sont attaqués, ils sont tout d'abord bloqués selon leur ordre dans le registre principal de l'IP de surveillance (positionnement des bits liés au pare-feu dans le mot de 32 bits). Le processeur de mise à jour calcule les nouvelles politiques à implanter. Finalement, chaque pare-feu est remis dans son état *monitoring* dès qu'il a été mis à jour. Les pare-feux qui sont dans l'état *update* ne sont pas affectés car ils ne sont pas capables de transmettre des données. Cette méthode autorise toute IP mise à jour à être utilisée par l'application principale exécutée par le processeur généraliste. La latence de mise à jour dépend de la position du pare-feu dans la file d'attente des pare-feux à modifier : le premier pare-feu est modifié en N cycles, alors que le pare-feu placé en position n^o k dans la file d'attente de mise à jour est modifié en $k(N)$ cycles car on doit attendre que les $k-1$ pare-feux aient été mis à jour.

5 Implémentations et analyse de la solution

Ce chapitre se concentre sur l'analyse des implémentations des mécanismes proposés dans les chapitres 3 et 4. Certains résultats sont issus de simulations et d'autres d'implémentations réelles sur une plateforme FPGA Xilinx ML605 (avec un FPGA Virtex-6 xc6vlx240t1156-1). Dans un premier temps, un cas d'études est décrit avant de présenter les résultats ainsi qu'une analyse de ces valeurs.

Sommaire

5.1 Cas d'étude

5.1.1 Architecture

Pour les implémentations et les simulations réalisées dans ce chapitre, le raisonnement se base sur un cas d'étude qui se veut représentatif d'un système embarqué réel. L'architecture témoin proposée dans la figure 5.1 est une architecture multiprocesseur à 2 processeurs softcore Microblaze (fréquence de fonctionnement 100 MHz, pas de caches d'instruction ou de données dans un premier temps) : le processeur MB_1 sera utilisé pour communiquer avec les IPs du système et le processeur MB_2 pour aller lire et écrire des données dans la mémoire externe (bien évidemment, ces tâches pourraient être réalisées par un seul et même processeur mais on garde cette structure pour pouvoir illustrer les différents types de communications qui peuvent intervenir dans le système).

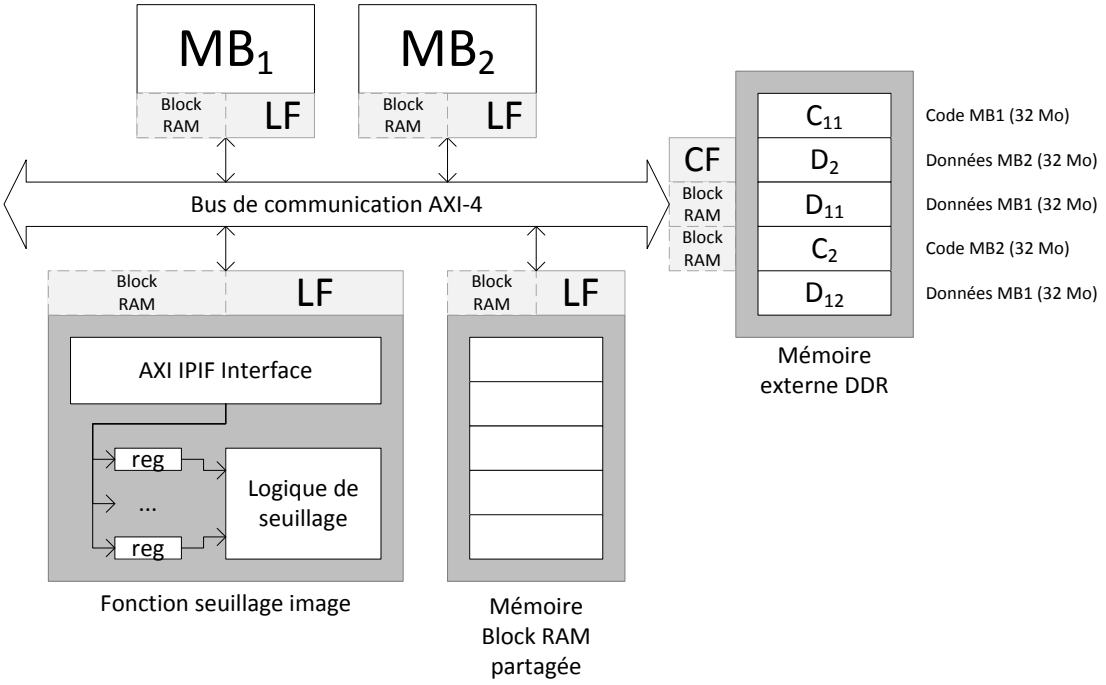


FIGURE 5.1 – Architecture du cas d'étude

Ce cas d'étude possède également 2 IPs :

– Une fonction de traitement d'image à niveaux de gris, un seuillage de l'image (pour obtenir une image en noir et blanc uniquement). Cette IP contient une dizaine de registres programmables qui vont servir à définir la valeur du seuil désiré ainsi que le type d'image en entrée de la fonction de seuillage.

– On suppose que les images temporaires (entre deux traitements) ainsi que d'autres informations telles que le code de fonctions de traitement plus poussées ou une banque de profils d'utilisateurs, qui permet de définir une chaîne de traitement dédiée à chaque utilisateur en fonction de son identifiant, sont stockées dans la mémoire partagée Block RAM.

On considère également que la mémoire externe est composée de différentes sections de pages mémoire de taille égale (chaque section est d'une longueur totale de 32 Mo) et partagées entre les deux processeurs :
– Pour le processeur MB1 : une section de code (C_{11}) et deux sections de données (D_{11} et D_{12})
– Le processeur MB2 possède quant à lui une section de code (C_{21}) et une section de données (D_{21}).
Toutes ces sections sont définies avec des politiques de sécurité et des paramètres cryptographiques décrits dans le paragraphe suivant.

5.1.2 Politiques de sécurité

En ce qui concerne les aspects cryptographiques, les sections mémoire peuvent être protégées en confidentialité-intégrité (C_{11} et D_{11}), en intégrité seule (D_{12}) ou même en clair, c'est-à-dire sans protection (comme C_{21} et D_{21}). On suppose que les politiques de contrôle du format de données sont uniformes : seules les données de 32 bits sont autorisées dans toutes les transactions qui se déroulent dans le système embarqué. Enfin, les politiques de sécurité sont construites pour avoir les droits d'accès suivants en termes de lecture-écriture :

	Mémoire partagée	**Traitement d'images**	**Mémoire externe**	
			C_{21}, D_{21}	C_{11}, D_{11}, D_{12}
MB1	Lecture seule	Lecture Écriture	Aucun accès	Lecture Écriture
MB2	Lecture Écriture	Écriture seule	Lecture Écriture	Aucun accès

TABLE 5.1 – Droits d'accès pour le cas d'étude

Ces politiques sont donc construites pour pouvoir illustrer toutes les options (en termes de politiques et d'options cryptographiques) sur une architecture multiprocesseur implantée sur un circuit FPGA.

5.1.3 Logiciel

Dans un second temps, le cahier des charges du cas d'étude comprend également des contraintes au niveau logiciel. On en distingue deux sous-ensembles : les applications et le système d'exploitation.

5.1.3.1 Applications

Dans un premier temps, des applications customisées sont mises en oeuvre dans le système embarqué afin de pouvoir illustrer toutes les situations clés qui peuvent arriver (accès à la mémoire externe, communications entre les deux processeurs, communications entre un processeur et une IP...). L'application principale, nommée *picProc*, est basée sur une image témoin au format JPEG (précédemment chiffrée et écrite en mémoire externe par un processeur de configuration). Le déroulement de l'application est illustré dans le diagramme de séquence sur la figure 5.2.

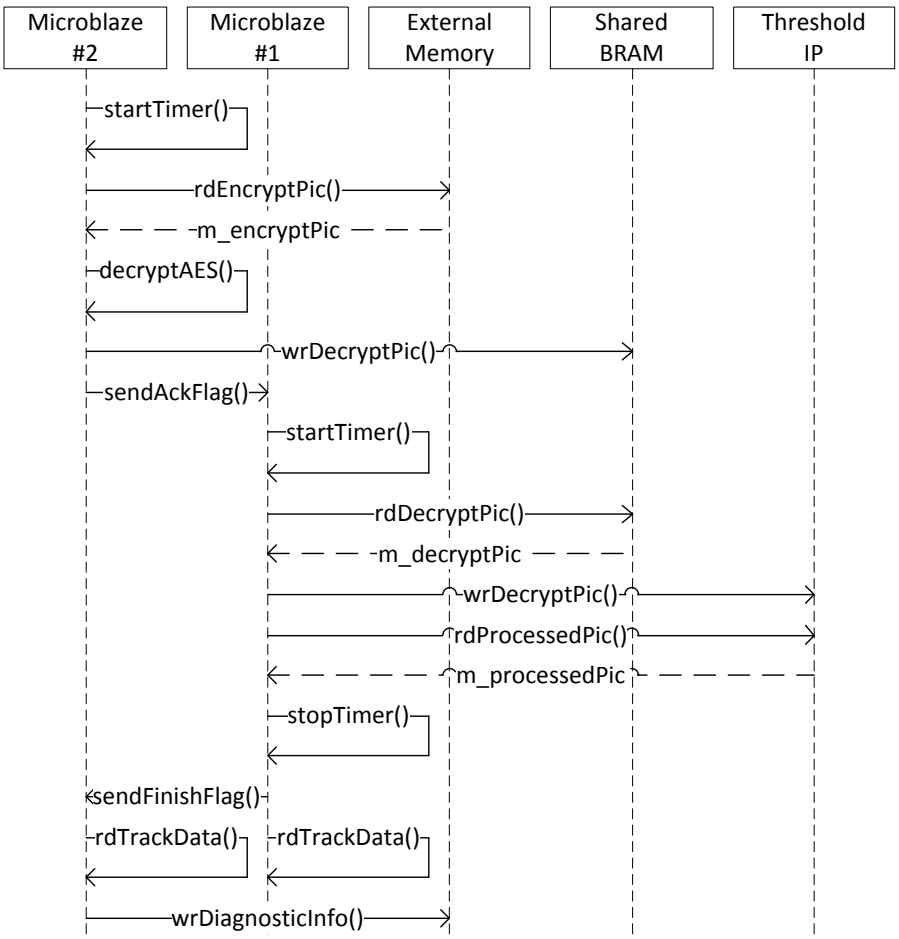


FIGURE 5.2 – Diagramme de séquence de l'application picProc

Toutes les opérations qui se déroulent dans le système sont sauvegardées dans un fichier de log. La première opération est l'activation d'un timer associé au processeur MB_2. Ensuite, MB_2 va lire l'image en mémoire externe (*rdEncryptPic()*) et procéder à un déchiffrement logiciel par une fonction *decryptAES()*. L'image, désormais en clair, est écrite dans la mémoire partagée et un message est envoyé au processeur MB_1 (*wrdecryptPic()*) pour lui dire de prendre le relais (par l'intermédiaire de la fonction *sendAckFlag()*).

Une fois que le processeur MB_1 a activé son propre timer (*startTimer()*) il va lire l'image en clair stockée dans la Block RAM partagée et la transmettre à l'IP de traitement d'image pour effectuer le seuillage (fonction *wrDecryptPic()*). Une fois le traitement effectué, le processeur MB_1 va lire l'image modifiée et la réécrire dans la mémoire externe grâce à la fonction *rdProcessedPic()*. Finalement, les deux processeurs vont arrêter leurs timers respectifs, récupérer les informations de diagnostic à écrire dans le fichier de log (*rdTrackData()*) et transmettre les informations à la mémoire externe.

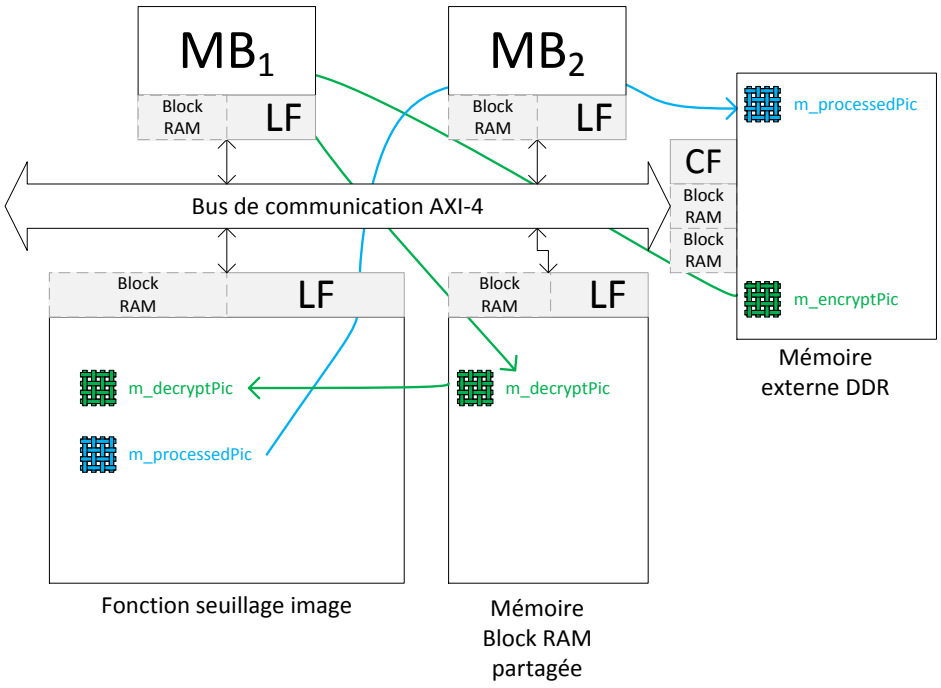


FIGURE 5.3 – Illustration du déroulement de picProc sur le cas d'étude

D'autres applications sont également utilisées pour l'évaluation du cas d'étude :
- *picDRM* est une application exécutée par un processeur qui va vérifier les DRM de l'image cible pendant que l'autre processeur effectue des opérations de lecture-écriture.
- *picDec* est une application qui fait un simple déchiffrement logiciel d'une image par un unique processeur. Elle sert de mesure de référence dans les études en latence.

Dans un deuxième temps, des applications issues de la suite de benchmark miBench [1] ainsi que d'autres applications de calcul customisées (calcul de FFT, calculs matriciels, etc...) sont utilisées pour illustrer les aspects présentés dans la section 6.1.

5.1.3.2 Système d'exploitation

En ce qui concerne les systèmes d'exploitation, deux options sont étudiées :
- Dans une version simplifiée du cas d'étude, l'architecture multiprocesseur est gérée par un système d'exploitation *standalone* (fourni par les outils de développement Xilinx [Xilinx 2012b]). Il ne gère pas les aspects multi-tâches mais permet de pouvoir communiquer entre les différents processeurs de l'architecture ainsi que, par exemple, provoquer des interruptions.
- Dans un second temps, on considère un système plus complexe : les fonctions décrites précédemment (*picProc*...) sont développées pour être exécutées sur un système compatible avec le standard POSIX [2] (dans les outils de développement, il s'agit du noyau Xilkernel de Xilinx mais on pourrait tout aussi bien implémenter sur μCOS [3] ou μCLinux [4]).

Dans les simulations du chapitre 6 sur la protection du boot d'un Linux embarqué, on utilisera le noyau Linux natif (disponible sur http://kernel.org/) dans sa version 2.631.

1. http://www.eecs.umich.edu/mibench/
2. http://standards.ieee.org/develop/wg/POSIX.html
3. http://micrium.com/page/products/rtos/os-ii
4. http://www.uclinux.org/

5.2 Résultats

5.2.1 Surface

5.2.1.1 Pare-feux seuls

Dans un premier temps, l'étude se concentre sur les pare-feux seuls (sans prise en compte des autres blocs du cas d'étude). D'après la description matérielle des pare-feux présentée dans le chapitre 3, la surface des pare-feux a une certaine dépendance avec le nombre de politiques intégrées dans chaque pare-feu. En fait, le module *Correspondence Table* (voir section 3.2.2) est le seul module dont la surface est dépendante du nombre de politiques que l'on souhaite intégrer dans le pare-feu. Le module *Correspondence Table* permet de définir les différents espaces d'adresses : c'est-à-dire qu'il permet de définir où aller chercher la politique de sécurité pour l'adresse de la transaction en cours d'analyse.

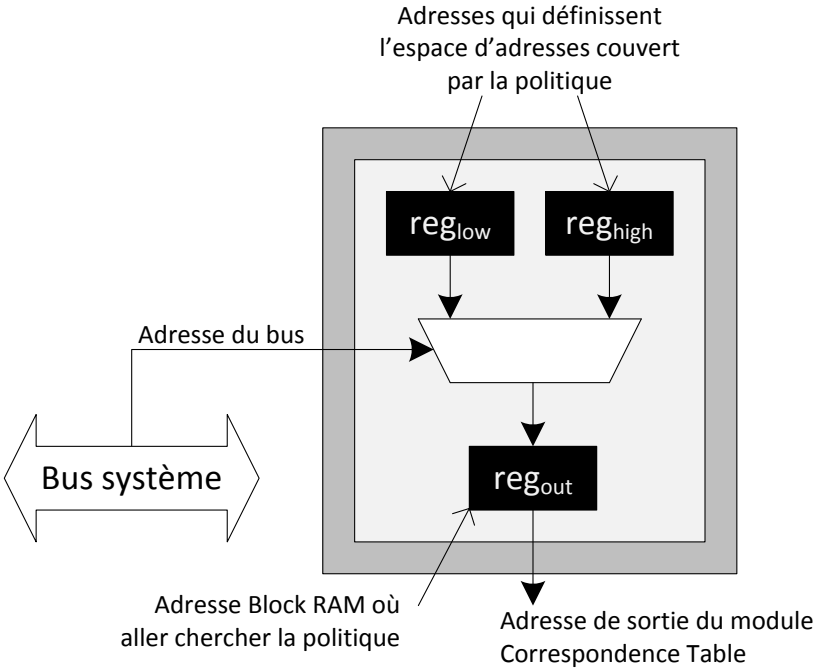


FIGURE 5.4 – Structure du module Correspondence Table pour 1 politique de sécurité

D'après l'algorithme défini dans la section 3.2.2, une politique de sécurité a besoin de trois registres de 32 bits : deux registres pour définir les frontières de l'espace d'adresse que couvre la politique (reg_{low} et reg_{high} dans la figure 5.4) et un troisième pour indiquer l'adresse en Block RAM où est stockée la politique à utiliser pour vérifier les paramètres de la transaction qui est en train d'être analysée (registre reg_{out} dans la figure 5.4).

D'après la formulation des politiques de sécurité présentée dans la section 3.4, on note les possibilités suivantes :
– Droits en lecture-écriture : 4 choix (lecture et écriture, une des deux ou aucune).
– Concernant le format, il y a 3 possibilités (8, 16 ou 32 bits).
– Les paramètres dédiés aux fonctions cryptographiques (choix des modes de confidentialité et d'intégrité) sont spécifiques au *Cryptographic Firewall* : il y a 3 choix possibles (confidentialité-intégrité, intégrité seule, clair) mais cela ne rentre pas en compte dans la suite du raisonnement.

Par conséquent, pour les droits d'accès et le format, il y a 12 combinaisons au maximum : autrement dit, on a au maximum 12 combinaisons possibles des paramètres de lecture-écriture et de format (par exemple une politique « lecture-écriture pour les données de 32 bits », une autre politique « lecture seule pour les données de 16 bits », etc. Étant donné que l'on souhaite conserver des modèles de sécurité relativement simples, on considère que l'on peut gérer au maximum 10 espaces d'adresses dans chaque pare-feu : dans le pire cas, chaque espace d'adresse a sa propre politique mais les politiques peuvent aussi être partagées entre plusieurs espaces d'adresses.
– A priori, les applications visées ne nécessitent pas autant de « découpage » de l'espace mémoire associé à chaque IP.
– En embarquant 10 politiques de sécurité, on dispose d'une couverture assez large au niveau des possibilités de contrôle. Malgré tout, s'il est nécessaire d'en modifier, les politiques peuvent toujours être mises à jour avec les mécanismes présentés dans le chapitre 4.

Les outils de développement donnent *de facto* la possibilité de créer des IP avec 32 registres programmables. Comme il est nécessaire d'avoir trois registres par politique, il en faut 30 au maximum (3x10 politiques) ce qui est donc possible.

Pour les études en surface, il est important de savoir quelle est la dépendance de la surface du module *Correspondence Table* en fonction du nombre de politiques.

Nombre de politiques	Slices	Slice regs	LUTs	Surcoût moyen
2	9	0	32	référence
4	17	0	39	× 0,55
6	29	0	64	× 1,61
8	37	0	94	× 2,52
10	48	0	117	× 3,49

TABLE 5.2 – Évolution de la surface du module Correspondence Table en fonction du nombre de politiques

Le tableau 5.2 montre les ressources nécessaires en termes de slices, LUTs et registres

en fonction du nombre de politiques à intégrer, il donne également le surcoût moyen par rapport à l'implémentation à 2 politiques de sécurité (qui constitue une référence). Les optimisations faites par l'outil de synthèse font que les registres sont synthétisés par des LUTs. Le rapport du surcoût en surface est de 3 (pour une implémentation avec 10 politiques de sécurité). Sauf mention contraire, on considère dans la suite le pire cas, c'est-à-dire des pare-feux avec un module *Correspondence Table* pouvant contenir 10 politiques de sécurité (la surface reste acceptable comparée aux autre modules comme on le verra dans le tableau suivant). Le tableau 5.3 présente les résultats d'implémentations des pare-feux.

		Slices	**Slice regs**	**LUTs**	**# Block RAMs**
Local Firewall	**FI**	76	120	68	0
	SB	23	3	55	1
	Total	99	123	293	1
Crypto Firewall	**FI**	76	120	153	0
	SB	23	3	55	1
	CM	1 166	2 038	2 396	14
		89,42%	94,31%	89,10%	93,33%
	Total	1 304	2 161	2 689	15
MicroBlaze		1 179	1 298	1 829	10

TABLE 5.3 – Résultats de synthèse des pare-feux seuls

Dans le tableau 5.3, le processeur Microblaze est dans une configuration classique (fréquence de fonctionnement à 100 MHz, pas de caches d'instructions ou de données). Un *Local Firewall* a un coût en surface peu élevé par rapport au *Cryptographic Firewall* (la surface d'un LF prend en moyenne 9% de celle d'un CF) et au Microblaze (dans ce cas, le rapport moyen est de 11%). Ceci est essentiellement dû au fait que la majeure partie de la surface vient du bloc de chiffrement AES-GCM (qui compte pour 90% de la surface d'un *Cryptographic Firewall* : il comprend plusieurs éléments dont une fonction AES, des multiplieurs de Galois...). Maintenant qu'on connaît les ressources en surface nécessaires pour l'implantation des pare-feux, on peut définir un ensemble d'équations qui permet d'estimer la surface utile dans une architecture multiprocesseur avec x pare-feux locaux et y pare-feux cryptographiques.

L'établissement de ces équations repose sur plusieurs hypothèses :
– Étant donné qu'il s'agit d'une estimation, on suppose que le fonctionnement de l'outil de synthèse est linéaire : autrement dit, on suppose que la synthèse d'une structure avec 2 modules identiques donnera une surface double d'une synthèse d'un module simple. Même si ce n'est pas réellement le cas, cela permet tout de même d'avoir un ordre de grandeur de la surface occupée par les implémentations sur des supports commerciaux proposées dans la suite de ce chapitre.
– Comme dit précédemment, la surface du module *Correspondence Table* est dépendante du nombre de politiques de sécurité ; on prend donc l'implémentation du pire cas avec 10 politiques de sécurité (voir tableau 5.2).

Les équations obtenues en termes de slices, registres et LUTs sont de la forme suivante :

$$numSlices = 138 \times x + 1,304 \times y \tag{5.1}$$
$$numRegs = 123 \times x + 2,161 \times y \tag{5.2}$$
$$numLuts = 293 \times x + 2,689 \times y \tag{5.3}$$

Les équations 5.2, 5.3 et 5.3 représentent la surface des pare-feux dans leur implémentation statique telle que celle présentée dans le chapitre 3. Pour prendre en compte les ajouts utilisés pour la mise à jour des politiques (comme dans le chapitre 4), on considère l'implémentation d'un *Local Firewall* et on observe les surcoûts en surface qui sont décrits dans le tableau 5.4.

		Slices	**Regs**	**LUTs**	**BRAMs**
Solution statique (Local Firewall)		138	123	293	1
Ajouts solution de mise à jour	Temps réel	6	0	5	0
	Mode d'erreur	17	0	18	0
	Connexions	5	13	15	0
	Surcoût total	+20,29%	+10,57%	+12,97%	+0,00%

TABLE 5.4 – Résultats de synthèse des éléments dédiés à la mise à jour de la sécurité

La première chose à noter dans le tableau 5.4 est que celui-ci comprend uniquement les modifications sur un pare-feu. Pour avoir une solution complète, il faudrait ajouter des éléments de l'architecture présentée dans le chapitre 4 : un bus, un processeur dédié à la mise à jour et au monitoring et une IP de monitoring (qui récupère les informations de contrôle des différents pare-feux). Autrement, les ajouts sur un *Local Firewall* donne des surcoûts corrects (au maximum 20%) : ce surcoût serait moindre sur un *Cryptographic Firewall*. La contribution la plus important vient de la fonction qui réalise le mode « code d'erreur-quarantaine » : cette fonction nécessite quelques registres et un peu de logique de connexions (partie « Connexions » dans le tableau 5.4). La contribution « Temps réel » correspond quant à elle aux bascules utilisées dans le blocage des données pendant la mise à jour (en neutralisant les signaux de poignée de main pour bloquer les transactions).

5.2.1.2 Surcoûts sur le cas d'étude

Pour étudier les surcoûts sur une architecture plus complète, on considère le cas d'étude décrit au début de ce chapitre. Pour les pare-feux, les résultats seront pris depuis le tableau 5.2 (c'est-à-dire avec une implémentation « pire cas » des modules dont la surface est dépendante du nombre de politiques de sécurité). Les résultats sont donnés dans le tableau 5.5.

	Slices	**Regs**	**LUTs**	**BRAMs**
Solution non protégée	5 446	7 195	8 354	32
Solution avec pare-feux et sans mise à jour	7 302 +34,08%	9 848 +36,87%	12 215 +46,22%	51 +37,25%
Solution avec pare-feux + mise à jour	7 442 +1,92%	9 913 +0,66%	12 405 +1,55%	51 +0,00%

TABLE 5.5 – Résultats en surface des différentes configurations

Les trois configurations prises en compte sont : le cas d'étude sans mécanisme de protection, une version améliorée avec les pare-feux dans leur implémentation statique (voir chapitre 3) et enfin la version complète qui peut être mise à jour en temps réel (4). La version améliorée avec les pare-feux a un surcoût en surface assez important (de l'ordre de 34% en termes de slices) : ceci est en grande partie dû au bloc cryptographique AES-GCM embarqué dans le pare-feu lié au contrôleur de la mémoire externe. La logique ajoutée pour les opérations de mise à jour en temps réel implique un surcoût inférieur à 2% par rapport à la version statique.

5.2.1.3 Résultats sur des plateformes comemrciales

L'étude des pare-feux a également été effectuée sur deux plateformes commerciales. La première est le circuit MP21x de NEC (également étudié par Coburn [Coburn *et al.* 2005]).

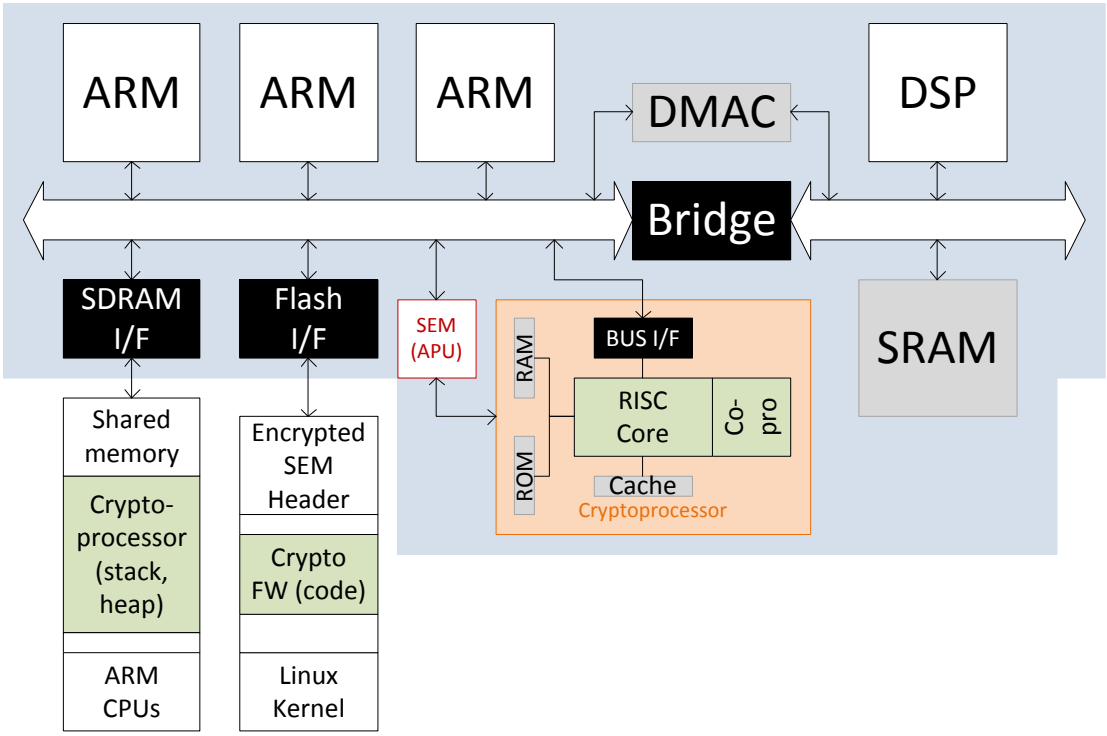


FIGURE 5.5 – Architecture simplifiée du NEC MP21x

La figure 5.5 présente une approche simplifiée de ce système. Il est composé de trois processeurs ARM 926, un cryptoprocesseur capable d'exécuter différents algorithmes (tels que RSA et AES). La plateforme MP21x possède également un mécanisme de politiques de sécurité. Pour ce cas d'étude, on considère que chaque processeur a des droits d'accès vers les autres IPs : ce processeur pourrait, par exemple, lire une des mémoires associées au cryptoprocesseur et écrire dans les autres. La deuxième plateforme est le Samsung Exynos 4210 basé sur une architecture à deux processeurs ARM Cortex A9 utilisée, par exemple, dans les smartphones tels que le Galaxy S2 (de Samsung également !). Ce système sur puce contient également des mémoires sécurisées, des périphériques vidéo et un moteur cryptographique. Selon le schéma de la figure 5.5, on considère que la SDRAM et la Flash ont besoin d'être protégées avec des fonctions cryptographiques. Par conséquent, il faut 5 *Local Firewalls* (LF) et 2 *Cryptographic Firewalls* (CF) pour la SDRAM et la Flash. Pour la plateforme Exynos 4210, il faut 12 LF pour 1 CF (interface mémoire).

On considère que chaque pare-feu contient 10 politiques de sécurité (et par conséquent 10 espaces mémoires à analyser pour chaque pare-feu) : d'après l'architecture de la figure 5.5, cela permet à chaque processeur ARM d'avoir au moins une politique sur chaque élément du SoC (voir même 2 politiques sur certains blocs plus critiques qui contiennent des données de plusieurs natures tels que la mémoire Flash qui contient le noyau du système d'exploitation mais également du code lié au cryptoprocesseur). De plus, on s'intéresse ainsi aux pare-feux dans leur version « flexible », qui peut être mise à jour (voir tableau 5.4).

	Nombre de LF + CF	**Slices**	**Slice Registres**	**LUTs**	**Nombre de Block RAMs**
MP21x	5 + 2	3 298	4 937	6 843	35
Exynos 4210	12 + 1	2 960	3 637	6 205	27

TABLE 5.6 – Implémentation des pare-feux sur des plateformes commerciales

Le tableau 5.6 est une proposition qui permet de montrer comment ces plateformes pourraient être sécurisées. Les résultats présentés sont simulés pour une carte ML605 du fabricant Xilinx embarquant un FPGA de la famille Virtex-6 (XC6VLX240T1156-1). L'implémentation des pare-feux avec mise à jour prend 8,75% du FPGA. Pour la plateforme Exynos 4210, les mécanismes prennent 7,86% du même modèle de FPGA.

5.2.1.4 Comparatif avec les travaux existants

Une comparaison quantitative et qualitative est faite avec les contributions majeures existantes. Le comparatif suivant prend en compte les travaux de Coburn [Coburn *et al.* 2005] et les travaux de Fiorin [Fiorin *et al.* 2008a]. Ces deux contributions proposent des approches ayant sensiblement les mêmes objectifs que les travaux présentés dans cette thèse (qui a pour but de propose un compromis des aspects positifs des travaux existants). Dans le tableau 5.6, il y a un comparatif quantitatif et qualitatif :

– Pour des résultats quantitatifs, on compare la surface du bloc de sécurité proposé dans les travaux choisis par rapport à un processeur qui sert de mesure de référence (la taille des caches ainsi que la fréquence de fonctionnement sont données entre parenthèses). Pour la solution des pare-feux proposée dans ces travaux, le bloc de sécurité est un *Local Firewall* (les fonctions cryptographiques sont un ajout par rapport aux solutions existantes : par conséquent, le bloc AES-GCM et les mémoires contenant les horodateurs et les tags ne sont pas pris en compte).

– Pour les aspects qualitatifs, on propose de regarder le protocole de communication utilisé, les aspects mise à jour des blocs de sécurité ainsi que les fonctions cryptographiques.

	Interface SEI [Coburn *et al.* 2005] (16KB / 150 MHz)	**Interface DPU [Fiorin *et al.* 2008a] (8KB / 100 MHz)**	**Notre solution (8KB / 100 MHz)**
bloc de sécurité ——————- processeur	6,20%	25%	11,30%
Protocole de communication	AHB/APB	NoC	AXI-4
Mise à jour de la communication	Non	Oui	Oui
Fonctions cryptographiques	Non	Non	Oui

TABLE 5.7 – Comparatif avec les travaux existants

En termes de surface, la solution proposée dans ces travaux n'est pas aussi performante que la solution SECA de Coburn [Coburn *et al.* 2005] : étant donné que l'approche des pare-feux est distribuée (les contrôles des pare-feux proposés dans ce manuscrit se font à chaque interface, contrairement à SECA), la logique de contrôle est dupliquée dans notre solution (d'autant plus qu'il faut rajouter un peu de logique pour gérer les aspects mise à jour des règles de sécurité). Par contre, les pare-feux sont tout de même plus légers que la solution proposée par Fiorin [Fiorin *et al.* 2008a] : ceci vient du fait que, pour la mise à jour des règles embarquées, la solution de Fiorin utilise un mécanisme de buffer pour garder les données entrantes (qui vont être analysées par leurs mécanismes de sécurité) le temps de rafraîchir les politiques de sécurité ; a contrario, notre solution utilise une propriété du protocole de communication AXI (mécanisme de type « poignée de main ») qui permet de suspendre les communications sans avoir de mémoire tampon pour stocker les données intermédiaires.

Pour comparer au mieux les différentes approches, on propose ensuite de comparer deux approches d'implémentation des pare-feux sur le cas d'étude proposé au début de ce chapitre :

– Implémentation distribuée : c'est celle proposée ici. Les contrôles (effectués par les pare-feux) sont mis en place à chaque interface entre une IP et le bus de communication du système. C'est aussi le type d'approche utilisée dans les travaux de [Fiorin *et al.* 2008a].
– Implémentation centralisée : dans ce cas, les contrôles sont effectués dans une seule interface et les mécanismes implémentés aux interfaces entre les IPs et le bus contiennent uniquement la logique permettant de transmettre et de recevoir les signaux utiles aux contrôles effectués (paramètres système, adresse à laquelle on veut accéder...).

	Slices	**Regs**	**LUTs**	**BRAMs**
Approche distribuée	7 302	9 848	12 215	51
Approche centralisée	6 750	9 356	11 043	47

TABLE 5.8 – Implémentations distribuée et centralisée

Pour l'approche centralisée (semblable à ce qui est fait dans SECA [Coburn *et al.* 2005]), la surface est forcément moindre dans la mesure où la logique de vérification des règles de sécurité est implémentée dans un bloc principal (ce qui impose un surcoût en latence démontré ultérieurement...). Malgré tout, étant donné que la surface des pare-feux de type *Local Firewall* est relativement faible par rapport aux pare-feu de type *Cryptographic Firewall*, le gain maximum est de l'ordre de 8%. Le paramètre surface est important dans le développement d'une telle solution dans la mesure où les ressources des systèmes embarqués ne sont pas infinies mais le cahier des charges imposé dans le chapitre 2 suggère que la latence des mécanismes de sécurité importés est encore plus importante.

5.2.2 Latence

Pour mesurer la latence des pare-feux dans des scenarii plus complexes impliquant plusieurs éléments de l'architecture, des simulations ont été faites avec les scenarii suivants :

– S0 : latence d'un *Local Firewall*.
– S1 : communication entre deux éléments de la puce (utilisation de deux *Local Firewalls*).

– S2 : communication entre un processeur et une zone de la mémoire externe protégée en confidentialité et en intégrité (intervention d'un *Local Firewall* et d'un *Cryptographic Firewall*).
– S3 : communication entre un processeur et une zone de la mémoire externe en intégrité seule.
– S4 : communication entre un processeur et une zone de la mémoire externe où le texte est en clair (seulement une vérification de la politique de sécurité associée).

Chaque scenario correspond au traitement d'une donnée de 32 bits par les pare-feux. La latence d'un *Local Firewall* (scenario S0) sert de mesure de référence, les autres scenarii impliquent deux pare-feux d'un système multiprocesseur sécurisé (S2, S3 et S4 font intervenir un *Local* et un *Cryptographic Firewall* tandis que le scenario S1 a besoin de deux *Local Firewalls*.

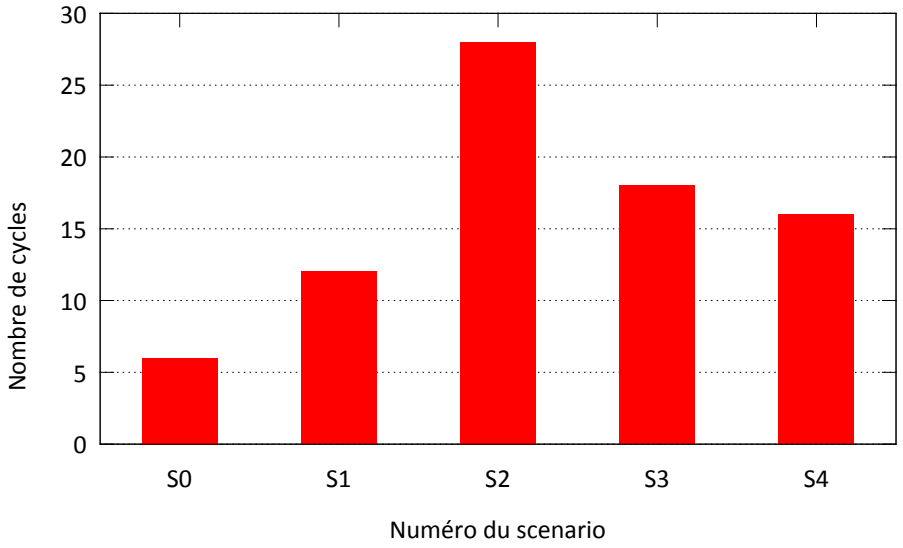


FIGURE 5.6 – Histogramme - Latence de simulations de scenarii

L'histogramme dans la figure 5.6 montre le résultat de ces simulations. Le scenario le plus critique en termes de latence est S2 : dès lors que l'on accède à une donnée stockée en mémoire externe protégée avec des fonctions cryptographiques, le surcoût imposé par le chiffrement-déchiffrement et le calcul-vérification de l'intégrité fait que le scenario S2 dure 28 cycles d'horloges :
– **6 cycles** pour les opérations de vérification des politiques de sécurité comme dans un *Local Firewall*, cela correspond au scenario S0.

– D'après la description du bloc de chiffrement dans le chapitre 3, la protection en confidentialité et intégrité de N blocs de données est effectuée en $10 + (10 + 2) \times N$ **cycles**, soit **22 cycles** pour une donnée. On obtient donc les 28 cycles qui correspondent au scenario S2 dans la figure 5.6.

S1 et S4 ont des résultats proches car, dans cette simulation, il n'y a pas de *Checking Module* (1 cycle d'horloge est ainsi préservé) dans un *Cryptographic Firewall* et la lecture de la politique de sécurité associée à un pare-feu cryptographique prend 5 cycles supplémentaires que la lecture d'une politique pour un *Local Firewall* : par conséquent, le scenario S4 prend 4 cycles d'horloge de plus que le scenario S1.

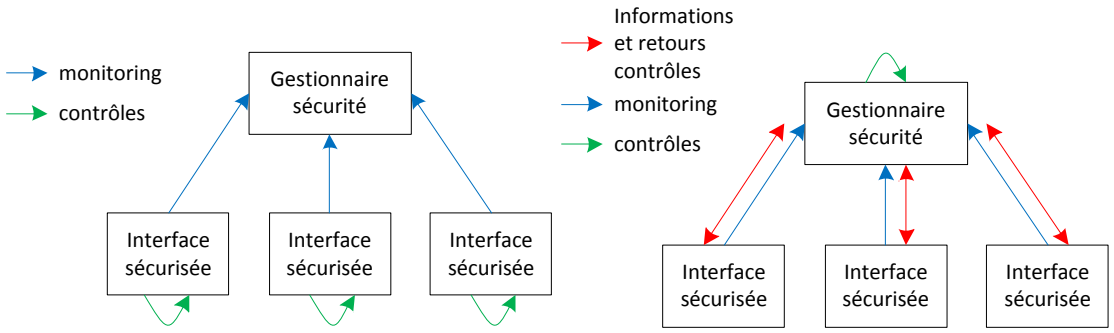
Par extension, il est possible de faire des mesures sur des scenarii réels en exécutant les applications *picProc*, *picDRM*, *picDec* définies dans le cas d'étude (voir section 5.1). Dans cette nouvelle configuration, on considère le cas d'étude présenté au début de ce chapitre (section 5.1) mais avec des caches d'instructions et de données de 8 kilo-octets. Pour chaque application, il a fallu déterminer le nombre d'accès à la mémoire sur la totalité de l'exécution des applications (pour différencier le nombre d'accès qui devraient transiter par un *Local Firewall* ou un *Cryptographic Firewall* qui n'ont pas le même impact en latence). En utilisant les registres spécifiques du Microblaze, il est possible d'en extraire des informations telles que le nombre de cache miss (et donc d'accès à la mémoire externe pour aller récupérer du code processeur) et d'en déduire le nombre global d'accès à la mémoire externe (cache miss et lectures-écritures de l'application elle-même) : les signaux de traçage (*Trace Interface Description*, dans la documentation constructeur du Microblaze [Xilinx 2012a] permettent de détecter les cache miss (ou cache hit) ; en incrémentant un compteur à chaque événement, on obtient le nombre total de cache miss-hits en fin d'exécution de l'application. Le nombre d'accès à la mémoire par l'application se fait également par un de ces signaux de traçage.

Par conséquent, on considère que sur la durée totale de l'application, le temps est divisé en deux catégories : les accès vers la mémoire externe (cache miss ou lecture-écriture de l'application, qui font donc intervenir un pare-feu cryptographique et un pare-feu local), des accès en internes (qui font intervenir uniquement des *Local Firewall*). Finalement, les mesures sont mises en corrélation avec des timers pour en extraire le temps d'exécution de l'application et donc le surcoût en latence des trois applications proposées résumés dans le tableau 5.9.

	Temps d'exécution (ms)	Nombre de cache miss	Nombre d'accès à la mémoire DDR	Surcoût en latence
picProc	3 623	12 672 395	34 063 398	17,76%
picDrm	1 084	3 017 237	9 462 055	9,43%
picDec	382	793 226	4 736 966	4,18%

TABLE 5.9 – Surcoûts de la sécurité sur des applications logicielles

L'application la plus simple (picDec) à un surcoût inférieur à 5% : elle n'utilise pas les fonctions de confidentialité-intégrité du *Cryptographic Firewall* et n'utilise qu'un seul processeur. Par contre, l'application la plus pénalisante (PicProc) a un surcoût, malgré tout acceptable, d'environ 18% : ceci est dû au fait que l'application utilise les fonctionnalités cryptographiques associées à la mémoire externe mais aussi à l'utilisation de tous les éléments de l'architecture (les deux IPs et les deux processeurs). Afin de pouvoir comparer les travaux présentés dans cette thèse aux travaux de référence en termes de latence, on décide de transposer le modèle des pare-feux dans une approche centralisée à l'image de ce qui est fait dans le modèle SECA [Coburn *et al.* 2005].



(a) Distribué (b) Centralisé

FIGURE 5.7 – Schémas de fonctionnement des mécanismes de sécurité

Dans la version distribuée (telle que celle utilisée par les pare-feux), les contrôles se font en interne et des informations de monitoring sont renvoyées au gestionnaire de sécurité (symbolisé par l'IP de monitoring et le processeur de mise à jour dans ces travaux). Dans la version centralisée (par exemple, SECA [Coburn *et al.* 2005]), chaque interface sécurisée envoie des informations utiles pour les contrôles effectués par le gestionnaire de sécurité du système.

Toutes ces connexions sont considérées comme étant résistantes aux attaques, mais la centralisation de la sécurité impose un surcoût en latence par la transmission des informations utiles aux contrôles de la part de chaque interface sécurisée implantée dans le système embarqué. Dans cette thèse, on considère que chaque transaction (un aller-retour entre une interface sécurisée et le gestionnaire principal, ce qui correspond à la transmission d'entrées utilisées pour les contrôles puis aux résultats de ces vérifications) par l'intermédiaire d'un bus de type AXI-Lite requiert 4 cycles d'horloge supplémentaires (la latence de transmission d'une donnée sur un bus utilisant le protocole AXI étant de 2 cycles, un aller-retour est effectué en 4 cycles [ARM 2012, Chang *et al.* 2009]). Par conséquent, pour l'application *picDec* défini précédemment, une implémentation centralisée des services de sécurité (telle que SECA) donne un surcoût en latence de 6,18% alors que la solution à base de *Local Firewall* et de *Cryptographic Firewall* peut se faire avec un surcoût en latence de 4,18% ce qui fait un gain de 33% par rapport à l'approche SECA.

5.2.3 Occupation mémoire

En dehors de la mémoire externe (qui peut être de plusieurs giga-octets), on considère que l'occupation des ressources en mémoires internes (Block RAM) du FPGA sur lequel est implanté le système multiprocesseur doit être étudiée. D'après la description matérielle des pare-feux dans les chapitres 3 et 4, les mémoires internes peuvent être utilisées pour stocker :

– Les politiques de sécurité : les paramètres à vérifier par les pare-feux.
– Les horodateurs : utilisés par le module cryptographique AES-GCM (contre-mesure contre les attaques par rejeu). On suppose qu'ils sont générés en interne par des compteurs.
– Les tags : lorsqu'il y a besoin de protéger le système avec une propriété d'intégrité, un tag doit être créé.

Dans un premier temps, on s'intéresse uniquement aux tags produits par le bloc AES-GCM. D'après la description qui en est faite dans le chapitre 3, à chaque bloc de données de 128 bits est produit un tag de longueur équivalente. Les implémentations présentées dans ce chapitre se basent sur un FPGA Xilinx de la famille Virtex-6 (XC6VLX240T1156-1) : ce modèle de FPGA possède 14 976 Kbits de mémoire interne Block RAM. Par conséquent, si on veut stocker tous les tags dans des mémoires internes (qui sont considérées comme sécurisées et n'ont pas besoin de protection supplémentaire), on peut protéger 1,87 Mo de données en termes d'intégrité.

On peut en déduire deux caractéristiques :
– Au premier abord, on peut considérer que cette option n'est pas très efficace (peu de données peuvent être stockées au final...). Pour être plus efficace, il faudrait soit stocker les tags dans une mémoire externe (mais en la protégeant, ce qui va entraîner un surcoût en latence) ou utiliser une technique telle que les filtres de Bloom [Bloom 1970, Crenne 2011] : sous certaines conditions, cette technique permet de diminuer l'empreinte mémoire de 96%.
– Il faut aussi également rappeler que l'un des intérêts des pare-feux matériels proposés dans ces travaux est de proposer une flexibilité dans les protections offertes : tout ce qui est écrit en mémoire externe n'est pas forcément protégé en intégrité ; même si les données sont stockées en clair, les mécanismes à base de pare-feux bloquent tout de même certaines attaques.

Par conséquent, on considère dans la suite que les ressources en mémoires internes sont suffisantes pour les applications visées : soit un mécanisme de type filtre de Bloom a été implémenté, soit le cahier des charges de l'application fait que peu de données ont besoin d'être protégées en termes d'intégrité. Dans la suite de ce paragraphe, on considère donc uniquement l'occupation mémoire due aux politiques de sécurité.

En prenant en compte la formulation des politiques de sécurité définie dans la section 3.4, une simple politique de sécurité d'un *Local Firewall* tient sur 4 octets (24 octets pour un *Cryptographic Firewall* en y ajoutant les données cryptographiques). Deux options sont proposées :
– Chaque processeur a des droits uniformes sur les autres IPs : par exemple, un processeur peut écrire dans toute la mémoire externe en clair. Ce cas est représenté par une unique politique de sécurité.
– Chaque processeur a N « options » sur chaque IP cible : par exemple, un processeur peut lire une page de la mémoire externe où les données sont en clair et écrire une autre page où les données sont chiffrées. Plusieurs politiques de sécurité sont utilisées dans ce cas.

La deuxième option est une généralisation d'une implémentation de base. Plus il y a de politiques associées à chaque pare-feu, plus il faut de mémoire pour stocker tous ces paramètres.

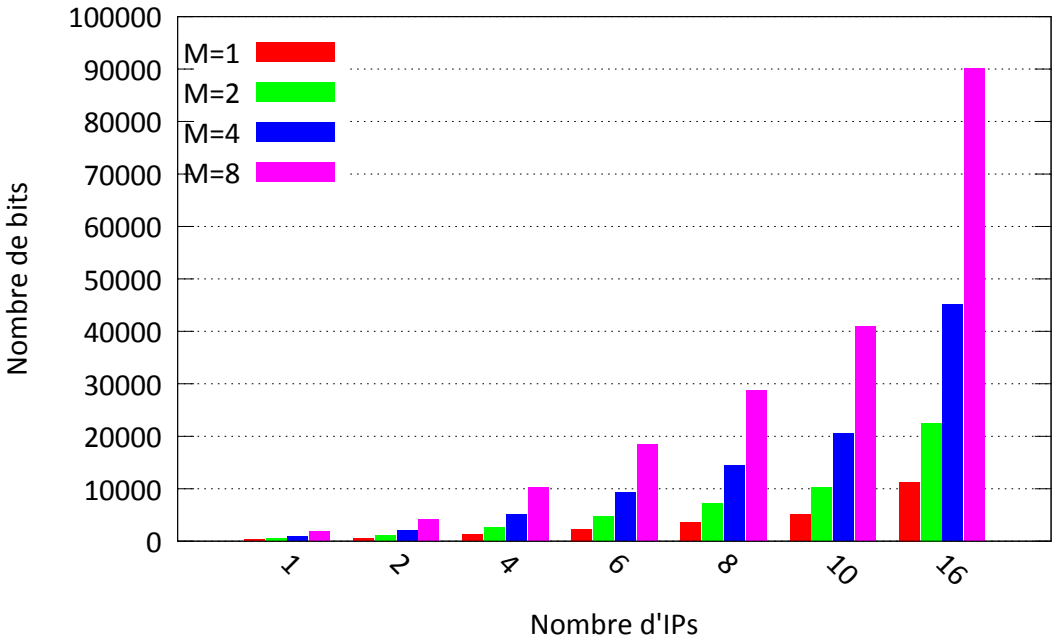


FIGURE 5.8 – Occupation mémoire en fonction du nombre de politiques de sécurité

La figure 5.8 montre le nombre d'octets nécessaires pour N IPs (axe des abcisses) pour M politiques de sécurité par IPs. Pour un nombre fixe d'IPs, l'occupation mémoire suit une fonction linéaire (décrite ultérieurement). Dans le cas d'un système à grande échelle (16 IPs et une mémoire externe, ce qui représente le seuil maximal pour un bus simple), les politiques de sécurité consomment quasiment 11 Ko de Block RAM (moins de 20% des Block RAMs embarqués dans un FPGA Xilinx Virtex-6 xc6vlx240t, le modèle embarqué dans la carte de développement ML605 [5]).

Ensuite pour estimer l'occupation mémoire sur les plateformes commerciales utilisées dans l'étude sur la surface consommée par les pare-feux (MP21x de NEC et Saumsung Exynos 4210), on considère que dans ces systèmes chaque IP n'a besoin que d'une seule politique de sécurité. Pour calculer l'équation qui permet de connaître l'occupation mémoire d'un architecture multiprocesseur qui contient x *Local Firewalls* et y *Cryptographic Firewalls*, on suppose que :

- Chaque IP sécurisée avec un *Local Firewall* a accès à toutes les autres IPs du système (y compris les mémoires externes).
- Il y a $x + y$ IPs dans le système.
- Une IP ne peut pas interagir avec elle-même.
- Chaque politique de *Local Firewall* tient sur un mot de 32 bits.

5. http://www.xilinx.com/products/boards-and-kits/EK-V6-ML605-G.htm

On considère qu'il va y avoir deux contributions que l'on va calculer séparément : une partie de la mémoire est occupée par les *Local Firewalls*, le reste est occupé par les *Cryptographic Firewalls*.

Pour ce qui est de la contribution due aux *Local Firewalls* :

– Un *Local Firewall* peut interagir avec tous les autres (qui sont donc au nombre de $x+y-1$).
– Chaque pare-feu possède 1 politique de sécurité soit 32 bits. Pour x pare-feux, cela donne $32 \times x$.

Par conséquent, l'occupation mémoire des *Local Firewalls* suit l'équation :

$$(x + y - 1) \times 32x \tag{5.4}$$

– Chaque IP sécurisée avec un *Local Firewall* peut accéder à la mémoire externe (dans ce cas, chaque IP a besoin d'une seule politique de sécurité, il y a donc x **politiques** dans 1 *Local Firewall*).
– Chaque espace d'adresses est géré par une politique de sécurité d'un *Cryptographic Firewall* sur **192 bits**.

L'occupation mémoire d'un *Cryptographic Firewall* est régie par l'équation suivante :

$$192 \times x \tag{5.5}$$

Cette équation devient $192xy$ s'il y a y *Cryptographic Firewall* dans le système cible. Dans la suite de cette partie, l'équation sera notée comme étant l'équation 5.5. L'évolution de l'occupation mémoire pour les plateformes NEC MP21x et Samsung Exynos 4210 suit l'équation de surface qui est la somme des deux contributions (en nombre de bits) :

$$\begin{aligned} & (x + y - 1) \times 32x && \text{voir l'équation 5.4} \\ & + 192 \times xy && \text{voir l'équation 5.5} \\ = \; & 32 \times x^2 + 224 \times xy - 32 \times x \end{aligned} \tag{5.6}$$

Cette équation peut être représentée par la figure en trois dimensions 5.9. D'après la figure 5.9 et l'équation 5.6, l'occupation mémoire est acceptable (864 octets pour le SoC Samsung Exynos 4210 et 360 octets pour le SoC NEC MP21x.

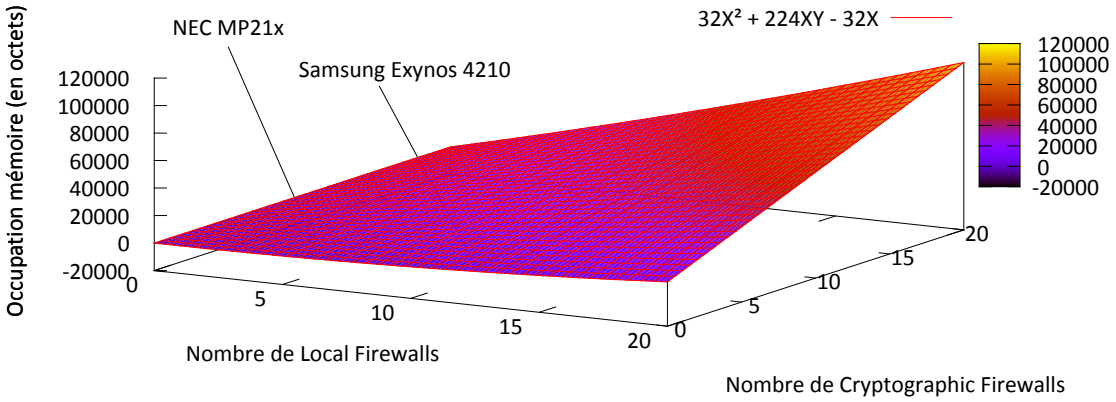


FIGURE 5.9 – Occupation mémoire en fonction du nombre de pare-feux

5.3 Gestion et stockage des clés

La solution proposée dans le chapitre 3 part du principe que chaque politique d'un *Cryptographic Firewall* possède sa propre clé : autrement dit, chaque page mémoire associée à une politique est chiffrée avec une clé différente comme le montre le schéma de la figure 5.8.

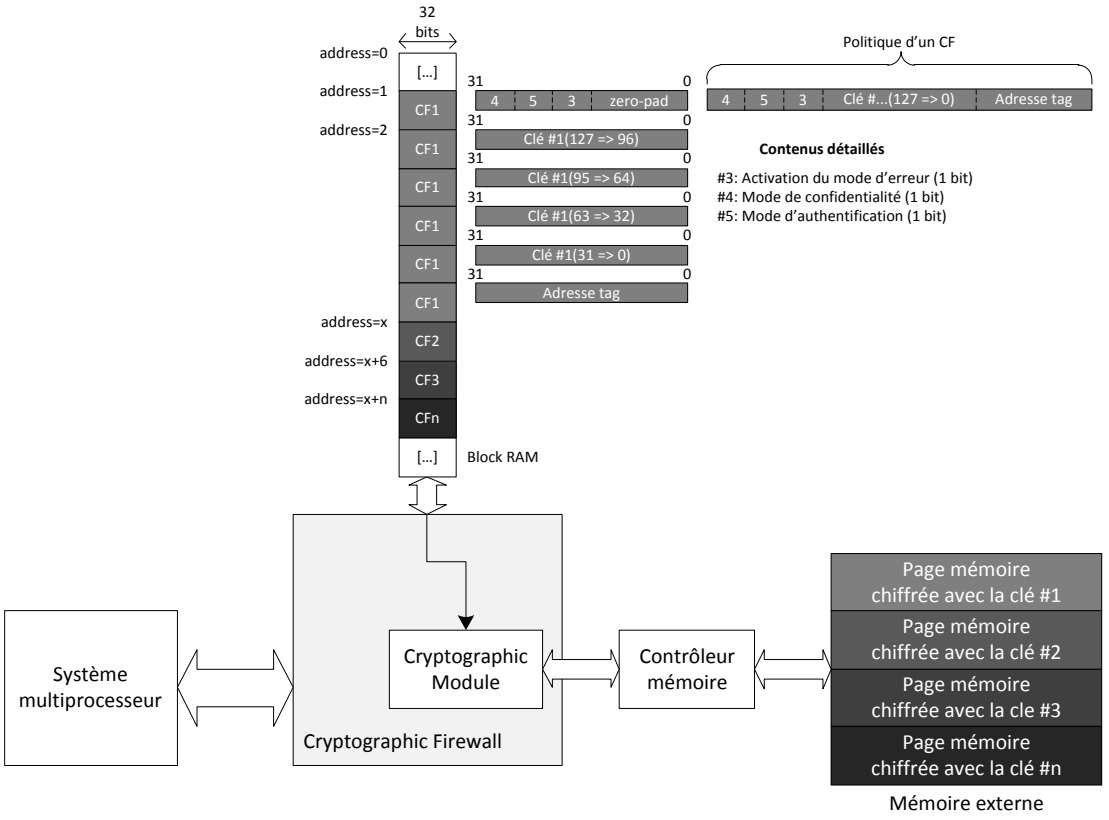


FIGURE 5.10 – Gestion d'une mémoire chiffrée avec plusieurs clés

La multiplication des clés présentée dans la figure 5.10 a l'avantage de permettre une structure plus complexe à attaquer mais également une consommation plus importante des mémoires internes pour stocker toutes ces clés. On se limite à 2 clés : une clé pour l'utilisateur (chiffrement de ses données) et une clé « super utilisateur » (ou administrateur) qui sert à chiffrer des données accessibles uniquement aux développeurs du système (par exemple, des programmes de référence à implanter au démarrage du système).

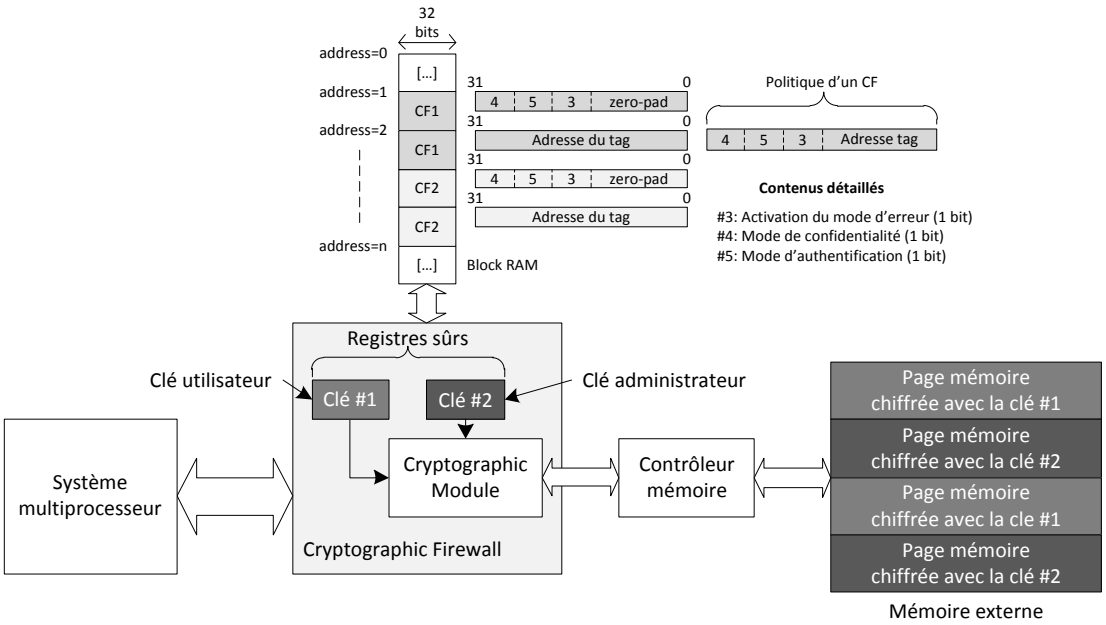


FIGURE 5.11 – Gestion d'une mémoire chiffrée avec 2 clés

Dans le cas d'une implémentation où la mémoire externe est chiffrée avec 2 clés (voir figure 5.11), les politiques de sécurité contiennent uniquement les modes de confidentialité et d'authentification tandis que le couple de clés est stocké dans des registres considérés sûrs associés au bloc de chiffrement dans le *Cryptographic Firewall*. Comme dans un pare-feu classique, la politique de sécurité est lue depuis la Block RAM en parallèle de la lecture des clés dans les registres sûrs.

L'occupation mémoire d'une telle approche diffère de celle présentée dans la section 5.2.3. Dans l'implémentation de la section 5.2.3, l'occupation mémoire d'une architecture avec x *Local Firewalls* et y *Cryptographic Firewalls* suit l'équation 5.6 qui est :

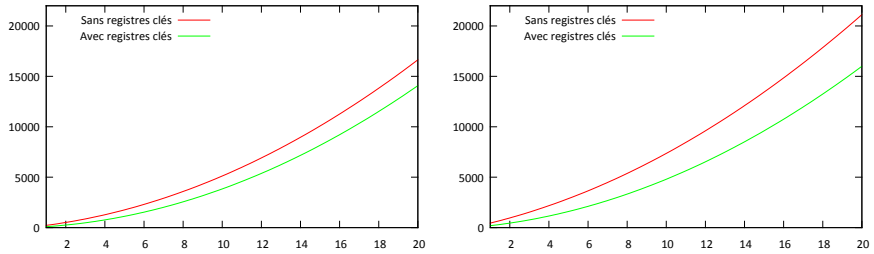
$$32 \times x^2 + 224 \times xy - 32 \times x \tag{5.7}$$

D'après l'équation 5.5, la mémoire occupée par y *Cryptographic Firewalls* est de $192xy$ (192 bits, dont 128 pour la clé). L'approche proposée dans cette section suggère de stocker les clés dans des registres du pare-feu et non plus en mémoire. Par conséquent, l'espace mémoire nécessaire pour y *Cryptographic Firewalls* est devenu $(192-128)xy = 96xy$ (on économise les 128 bits de clé cryptographique qui étaient précédemment stockés en Block RAM).

L'équation complète de l'occupation mémoire pour une gestion de la sécurité avec 2 clés est :

$$32 \times x^2 + 96 \times xy - 32 \times x \tag{5.8}$$

Les figures 5.12a (respectivement 5.12b) montre l'occupation mémoire pour 1 (respectivement 2) *Cryptographic Firewalls*. Dans chaque graphique, on trouve les courbes pour les deux implémentations : celle sans registres clés (exposée dans le chapitre 3 et celle avec les registres clés présentée ici.



(a) Pour 1 Cryptographic Firewall (b) Pour 2 Cryptographic Firewalls

FIGURE 5.12 – Occupation mémoire avec prise en compte ou non des registres clés

Les clés étant stockées dans des registres du module cryptographique, elles ne sont pas comptabilisées dans l'occupation mémoire totale (voir la démonstration de l'équation 5.8). Pour la solution à 2 clés (voir figure 5.11), la parallélisation de la lecture des politiques et des clés permet, en plus de diminuer la taille des Block RAM nécessaires, d'améliorer également la latence de lecture des politiques de sécurité. D'après la description des pare-feux dans le chapitre 3, la lecture d'une politique d'un *Cryptographic Firewall* prend 6 cycles. D'après la figure 5.11, les Block RAMs ne contiennent plus les clés : une politique ne prend plus que 8 octets en Block RAM et est donc lue en 2 cycles (les registres contenant les clés sont eux lus en 1 cycle en parallèle de la Block RAM).

5.4 Synthèse

Ce chapitre a permis de présenter des résultats d'implémentations des différentes versions des pare-feux pour observer les coûts en ressources physiques des solutions proposées et des simulations ont été faites pour des exemples sur des plateformes commerciales et déterminer certaines limites de l'approche par pare-feux comme

notamment les aspects occupation mémoire des paramètres de sécurité. Ces différents calculs ont également permis de faire un comparatif avec les solutions existantes (principalement [Coburn *et al.* 2005] et [Fiorin *et al.* 2008a]).

5.5 Conclusion

La solution à base de pare-feux proposée dans ces travaux a généralement des coûts en surface relativement importants : dans une implémentation classique contenant les deux types de pare-feux (*Local* et *Cryptographic*), l'impact en surface est essentiellement dû au bloc de chiffrement AES-GCM et aux éléments d'architecture dédiés à la mise à jour (un bus et un processeur spécifique). Étant données les évolutions des techniques de fabrication de FPGA (qui contiennent de plus en plus de ressources dans une seule puce), il n'est pas primordial d'étudier une implémentation dont l'empreinte en surface serait minimal. En termes d'occupation des mémoires internes au FPGA, l'approche proposée permet de protéger une quantité de données qui reste faible (quelques méga-octets). Pour diminuer la quantité de mémoire nécessaire, il faut soit implémenter une technique de compression des tags (comme les filtres de Bloom) ou restreindre le modèle de menace proposé au début du chapitre 3 afin d'avoir des mémoires plus conséquentes pouvant contenir des tags et sans avoir besoin de protection supplémentaire.

La caractéristique première des pare-feux, d'après le cahier des charges, est de proposer des mécanismes à faible latence pour éviter de perturber les communications qui se déroulent dans l'architecture multiprocesseur et ne pas avoir de retards excessifs dans les communications ou des problèmes de synchronisation entre les données et les signaux de contrôle. La latence de détection d'une attaque est faible (moins de 10%) mais on peut aussi faire des efforts dans les étapes de mise à jour des politiques de sécurité. Actuellement, la mise à jour fait intervenir un processeur et l'exécution d'un logiciel pour déterminer la nouvelle configuration : comme les pare-feux bloquent les données pendant la mise à jour, il n'y pas de fuites de données malveillantes mais il serait intéressant de regarder si une optimisation du protocole de mise à jour par une implémentation matérielle serait possible.

6 Extension des mécanismes de protection

Les chapitres précédents ont montré une solution qui permet de protéger une architecture multiprocesseur embarquée sur un composant de type FPGA contre certaines attaques. Ce chapitre propose deux extensions à cette solution : son intégration dans un flot sécurisé existant concernant le boot d'un système d'exploitation de type Linux ; la deuxième extension porte sur la granularité au niveau logiciel pour les mécanismes de type pare-feux proposés dans les chapitres 3 et 4.

Sommaire

6.1 Du bitstream à l'exécution de l'application

6.1.1 Problématique

Les systèmes embarqués deviennent de plus en plus complexes : les composants basés sur les technologies reconfigurables de type FPGA permettent d'embarquer de nombreux éléments dans une seule et unique puce. Lors de l'implémentation d'un système embarqué sur un FPGA, on distingue deux types d'information : les informations liées à la structure matérielle du système (fonctions des différents modules, interconnexions, machines à états...) sont contenues dans un fichier *bitstream* stocké, par exemple, sur un serveur dédié tandis que les informations liées aux aspects logiciels (système d'exploitation, applications et données) sont généralement stockées dans une mémoire non volatile. De plus, il peut exister des dépendances entre les données liées au matériel et les données liées au logiciel : dans le cas d'une architecture multiprocesseur, le système d'exploitation peut ne pas être adapté à la gestion simultanée de plusieurs processeurs ; le système d'exploitation doit aussi connaître l'organisation mémoire du système pour savoir où sont stockés les codes des différentes applications ainsi que les données associées.

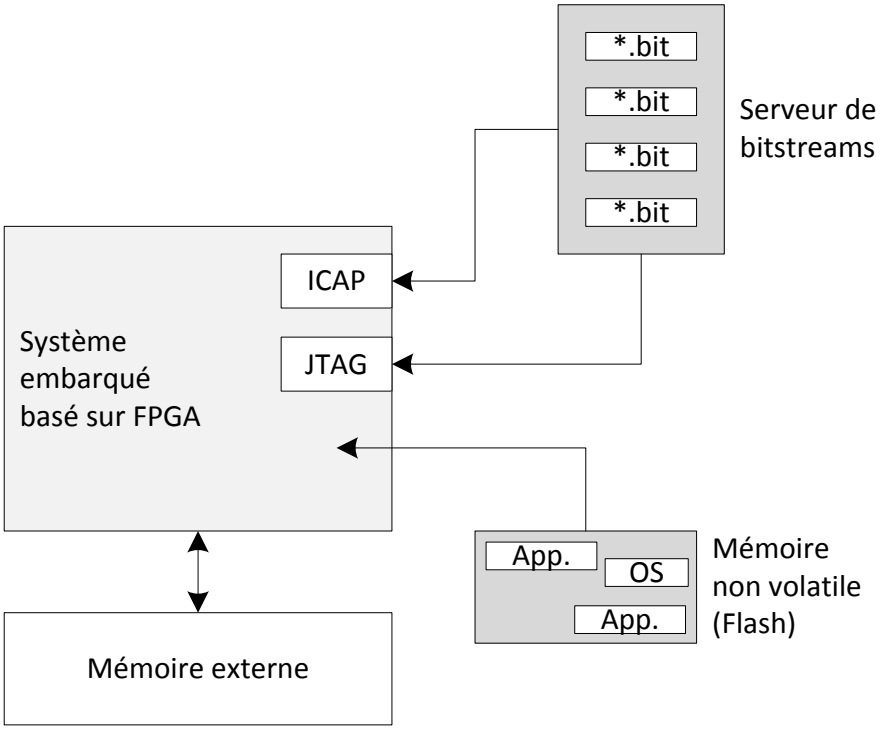


FIGURE 6.1 – Amorçage d'un système embarqué sur un FPGA

La figure 6.1 est une représentation d'un système embarqué sur une puce FPGA. L'architecture matérielle (depuis le bitstream) est chargée par l'intermédiaire d'un port dédié (JTAG, *Joint Test Action Group*). Dès lors que l'architecture matérielle est téléchargée sur la puce FPGA, un logiciel minimal (nommé *bootloader*) est exécuté sur le FPGA et copie les applicatifs liés au système (système d'exploitation et applications) depuis une mémoire non volatile vers une mémoire externe classique du type DDR.

Une fois le système mis en place, il est tout à fait possible de faire une mise à jour de l'architecture matérielle du système : soit on télécharge un bitstream complet soit on peut reconfigurer uniquement une partie de l'architecture (on utilise alors le procédé de la Reconfiguration Dynamique Partielle) en injectant un bitstream partiel par l'intermédiaire du port ICAP (*Internal Configuration Access Port*) du FPGA. Dans tous les cas, il faut que le téléchargement du bitstream se fasse de manière sécurisée afin de garantir la sécurité du système. Les fondeurs de FPGA intègrent des mécanismes qui permettent de sécuriser le téléchargement du bitstream. Le fondeur Xilinx propose une authentification par une fonction HMAC couplée à un chiffrement par un algorithme AES du bitstream [1] ; Altera, un autre fondeur utilise aussi un chiffrement AES pour la protection des bitstreams [2].

En ce qui concerne la mise à jour de la partie logicielle du système embarqué, les mécanismes de protection sont à la charge de l'utilisateur. Dans un système brut, il n'y a pas de lien sécurisé entre l'espace de stockage du logiciel (par exemple, une mémoire Flash non volatile) et les ressources internes au FPGA. L'attaquant peut alors se greffer sur ce lien de communication pour injecter des données malveillantes ou écouter ce qui transite (on parle alors d'*eavesdropping*) sur le canal de communication entre la mémoire Flash et le FPGA. Si le logiciel et le matériel d'un système embarqué ont été téléchargés correctement (c'est-à-dire sans être attaqués), il reste les problèmes liés à l'exécution en temps réel du système : dans son fonctionnement normal, le système doit également être protégé afin de continuer à s'exécuter dans un environnement sécurisé.

1. http://www.xilinx.com/products/technology/design-security/
2. http://www.altera.com/devices/fpga/stratix-fpgas/about/security/stx-design-security.html

Il y a des travaux existants concernant la protection du bitstream et le chargement sécurisé d'un système d'exploitation. Par contre, il n'y a pas de flot complet comprenant également la protection en temps réel du système cible. La suite de cette partie s'attache à montrer comment la solution des pare-feux présentée dans les chapitres 3 et 4 peut être intégrée à des travaux existants pour fournir un flot sécurisé complet depuis le chargement du bitstream jusqu'à l'exécution de l'application.

6.1.2 Travaux existants

Pour sécuriser un flot de chargement complet depuis le bitstream jusqu'à l'exécution de l'applicatif, la première étape est de protéger le chargement du bitstream. Les fondeurs de FPGA tels que Xilinx ou Microsemi (ex-Actel) proposent la technologie ISP (*In-System Programming*) pour protéger le bitstream en terme de confidentialité et d'intégrité (utilisation d'un MAC basé sur un algorithme AES) : malheureusement, cette solution est inefficace contre certaines attaques telles que le rejeu (ou *replay*) [3]. Dans un autre registre, les auteurs de [Drimer 2008] s'intéressent au chargement sécurisé de bitstream dans les FPGA basés sur la technologie SRAM (par exemple, dans la famille Virtex-II du fondeur Xilinx) : cela veut dire que ces FPGA sont basés sur une technologie volatile qui nécessite le téléchargement d'un bitstream à chaque allumage du système embarqué à base de FPGA ; la solution proposée est de sauvegarder le bitstream dans une mémoire Flash (non volatile) sécurisée. Devic et ses co-auteurs ([Devic *et al.* 2010]) proposent quant à eux une solution qui permet de protéger le chargement de bitstream contre les attaques par rejeu pour les FPGA possédant des mémoires non-volatiles.

Le noyau du système d'exploitation est généralement stocké dans une mémoire Flash externe qui doit être protégée. Généralement, l'intégrité est vérifiée à l'aide de fonctions de hachage telles que celles proposées dans les solutions Discretix [4] ou Atmel [5]. Les auteurs de [Devic *et al.* 2011] ont implémenté un mécanisme qui vérifie l'intégrité du noyau Linux en utilisant une fonction de hachage SHA-256 (*Secure Hash Algorithm*) qui permet de contrer les attaques par rejeu. Cette contribution apporte également une propriété de flexibilité en permettant à l'utilisateur de mettre à jour le noyau du système sans changer le bitstream associé.

3. http://www.docstoc.com/docs/33394717/In-System-Programming-%28ISP%29-of-Actels-Low-Power-Flash-Devices
4. http://www.discretix.com/secureboot/index.html
5. http://www.atmel.com/Images/doc6282.pdf

A ce stade, aucun mécanisme n'est proposé pour intégrer aux protections existantes la sécurisation de l'exécution du noyau Linux : dans la solution proposée par [Devic *et al.* 2011], le noyau du système d'exploitation est chargé de manière sécurisée dans une mémoire externe mais il est écrit en clair (et donc assujetti à des attaques). Dans la suite de cette partie, on explique comment les pare-feux décrits dans les chapitres précédents peuvent s'intégrer dans le flot de [Devic *et al.* 2011] afin de fournir aux concepteurs de systèmes embarqués utilisant une technologie FPGA un flot complètement sécurisé depuis le chargement du bitstream jusqu'à l'exécution du système d'exploitation et de l'application associée ; des résultats d'implémentation sont également donnés dans la fin de cette partie.

6.1.3 Boot flexible d'un Linux embarqué

Dans le protocole existant fourni par Xilinx, un des fondeurs de FPGA, le bitstream est copié de manière sécurisée depuis une mémoire Flash externe vers le FPGA puis le système démarre. Pendant ce temps, les mémoires Block RAM embarquées dans le FPGA sont initialisées avec un petit logiciel nommé « bootloader » exécuté par un processeur généraliste : le noyau Linux démarre une fois qu'il a été copié par le bootloader dans la mémoire externe DDR (une autre façon de procéder est de copier directement le noyau Linux de la Flash vers la DDR en utilisant un contrôleur DMA ce qui diminue le temps de boot, comme le montre les résultats présentés ultérieurement).

Malheureusement, dans ce type d'approche, il y a une faille de sécurité : un attaquant peut facilement mettre à jour la mémoire Flash avec un noyau contrefait car il n'y a pas de vérification d'intégrité des données contenues dans la mémoire Flash.

C'est la raison pour laquelle les auteurs de [Devic *et al.* 2011] ajoutent des modules au flot décrit précédemment (voir figure 6.1) pour avoir une vérification flexible de l'intégrité. Une fonction de hachage, le bloc SHA implémenté dans le FPGA, calcule un hash qui est comparé par le bootloader avec son propre hash : le noyau Linux démarre uniquement si les valeurs sont égales.

L'autre implémentation, basée sur une fonction de cryptographie asymétrique RSA, permet à l'utilisateur final de changer le noyau (dans la mémoire DDR) sans changer le bitstream : cette fonction n'est pas utilisée ici pour son chiffrement mais pour vérifier la signature du noyau Linux. Le flot sécurisé se déroule en deux étapes :

– La fonction de hachage génère le hash du noyau Linux.
– Ensuite, le bootloader (exécuté par le processeur généraliste) vérifie la signature du hash stocké dans la mémoire Flash externe avec le hash généré et sa clé publique.

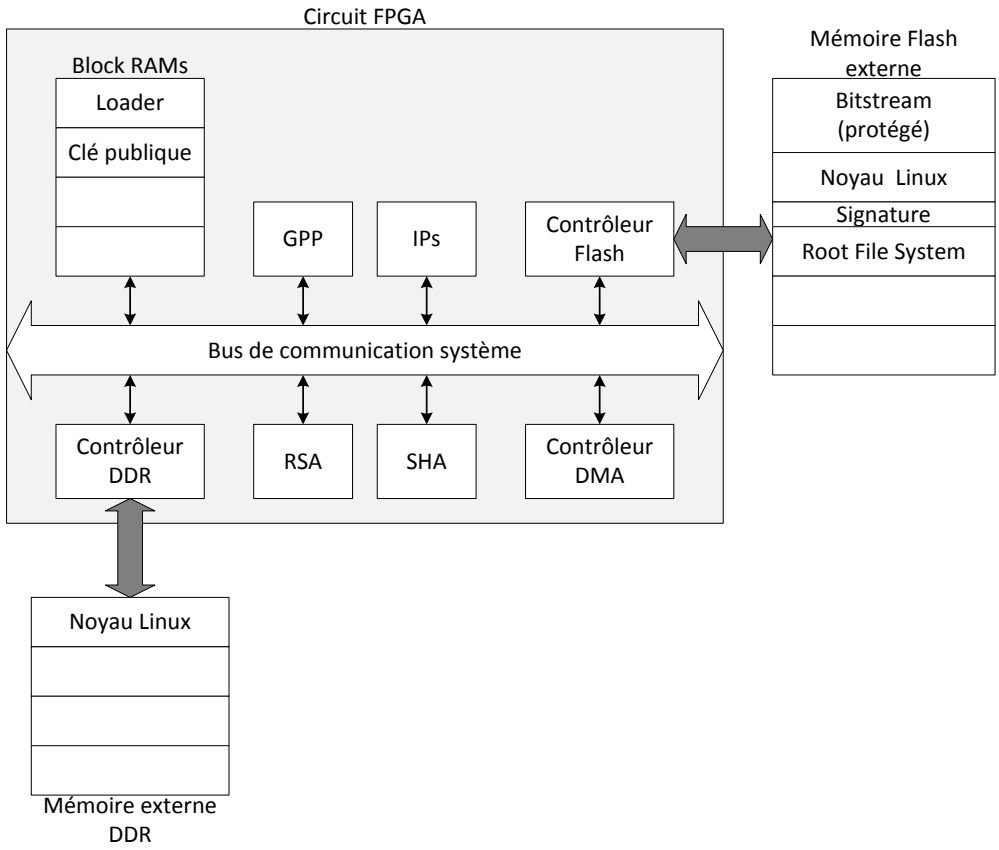


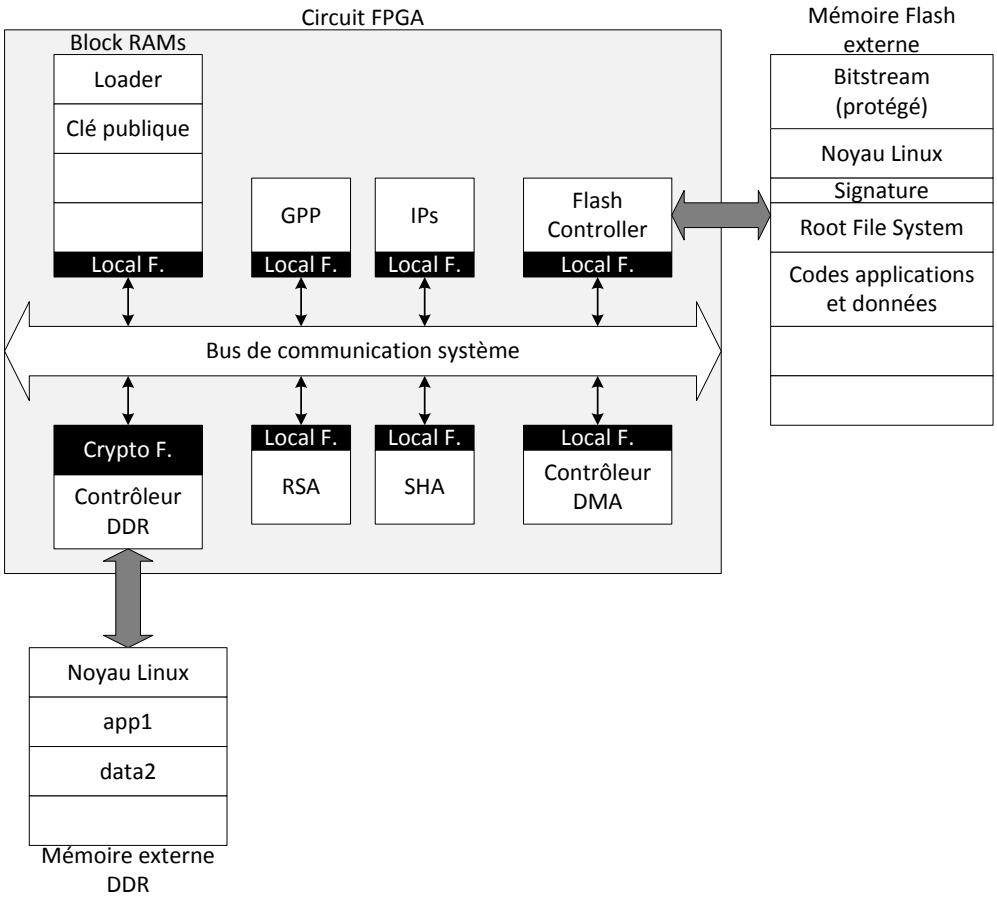
FIGURE 6.2 – Boot flexible d'un Linux embarqué

Dans la figure 6.2, la fonction RSA n'améliore pas le niveau de sécurité de la solution, la seule motivation est d'ajouter de la flexibilité aux mécanismes existants tout en minimisant les surcoûts en latence d'une telle opération (plus le noyau du système d'exploitation est chargé rapidement, mieux c'est). Á ce point, le noyau Linux est stocké en clair dans la mémoire externe DDR. Il s'agit d'une faille majeure étant donné qu'un attaquant peut facilement modifier le contenu de la DDR après le boot pour perturber le fonctionnement du système embarqué : c'est la raison pour laquelle des mécanismes doivent être implémentés au niveau de la DDR.

6.1.4 Contribution personnelle

Dans cette section, on propose d'ajouter les pare-feux décrits dans les chapitre 3 et 4 pour donner des propriétés de confidentialité et d'intégrité à la mémoire externe : ces deux fonctions sont réalisées par un pare-feu de type *Cryptographic Firewall*. Ce pare-feu particulier, basé sur un algorithme AES-GCM, permet au développeur d'un système embarqué de protéger le noyau Linux stocké dans la mémoire externe en termes de confidentialité et d'authentification (ou authentification seule), en accord avec le cahier des charges du système.

Le schéma de la figure 6.3 montre l'intégration des pare-feux dans le flot précédent (figure 6.2). Dans ce cas, le noyau Linux n'est pas la seule entité à être protégée par le *Cryptographic Firewall*. Les applications ou les données utilisées par un processeur implémenté dans le FPGA (considéré comme étant sécurisé) peuvent être protégées avec des fonctions de sécurité flexibles grâce à la logique embarquée dans le *Cryptographic Firewall*.



Etapes du boot

1. Le loader est stocké dans les Block RAMs à l'allumage depuis le bitstream

2a. Le loader copie le noyau dans la RAM (passant à travers le Crypto Firewall)

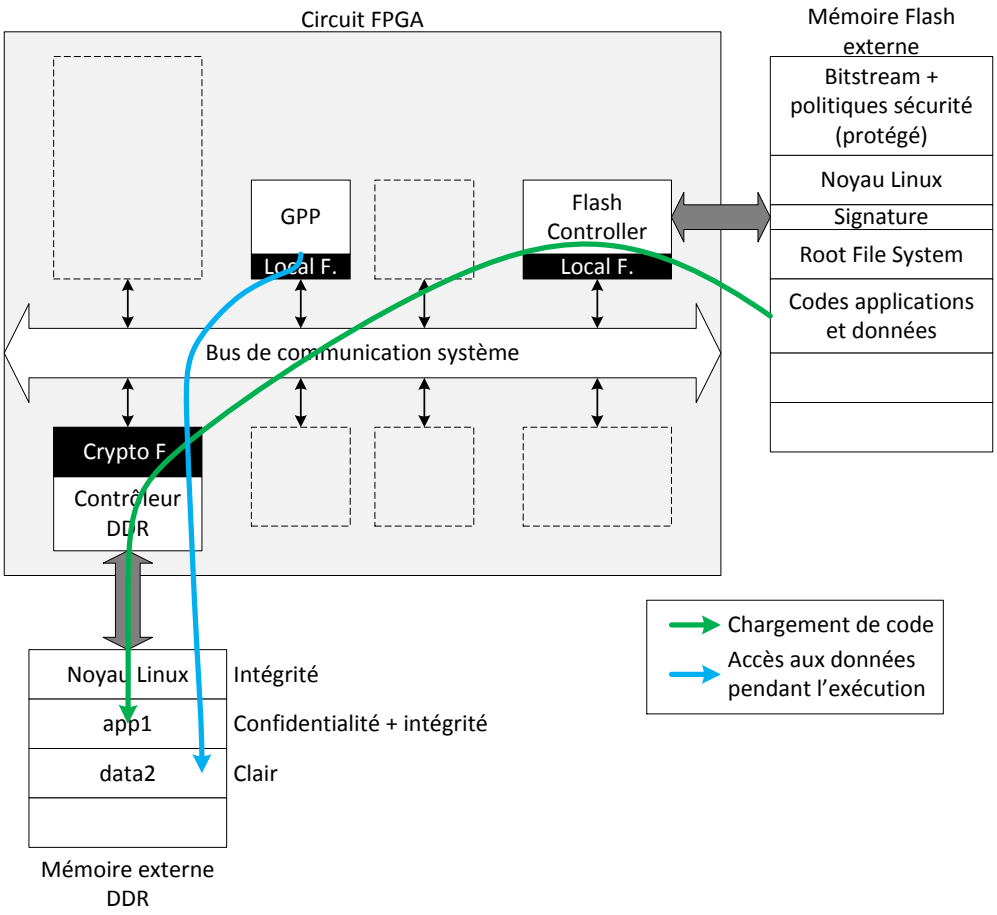
2b. Le hash du noyau est calculé grâce à la fonction SHA

3. Le loader vérifie l'intégrité du noyau en vérifiant sa signature (avec la fonction RSA)

4. Le loader se connecte au noyau et le Linux démarre

FIGURE 6.3 – Boot flexible avec protection de la mémoire externe

La figure 6.4 présente une version simplifiée de la figure 6.3. La configuration initiale des politiques de sécurité associées aux différents pare-feux (Block RAMs individuelles) est chargée en même temps que le bitstream qui est protégé. Le noyau Linux est protégé en intégrité tandis que la page mémoire *app1* est protégée en confidentialité et authentification et la page *data2* est en clair.



Etapes du boot

1. Le loader est stocké dans les Block RAMs à l'allumage depuis le bitstream

2a. Le loader copie le noyau dans la RAM (passant à travers le Crypto Firewall)

2b. Le hash du noyau est calculé grâce à la fonction SHA

3. Le loader vérifie l'intégrité du noyau en vérifiant sa signature (avec la fonction RSA)

4. Le loader se connecte au noyau et le Linux démarre

FIGURE 6.4 – Chemin de données et stockage des politiques dans une implémentation avec le boot flexible

Dans ce cas, les codes proviennent également de la mémoire externe (flot de données en vert sur la figure 6.4) : ils suivent donc le chemin de données qui passe par le contrôleur Flash directement vers la mémoire externe (en passant par le *Cryptographic Firewall* qui permet de vérifier les règles de confidentialité-intégrité) si le contrôleur DMA est activé. Pendant l'exécution, la protection de l'architecture se déroule de manière classique (comme elle a été décrite dans le chapitre 3, flot de données en bleu sur la figure 6.4) : le processeur généraliste (GPP) écrit une donnée dans la page mémoire *data2* : le chemin de données suit donc le *Local Firewall* associé au GPP ainsi que le *Cryptographic Firewall* au niveau du contrôleur de la mémoire externe. La décision de protéger telle ou telle section mémoire avec telle ou telle fonction cryptographique dépend du cahier des charges du système embarqué.

6.1.5 Résultats et analyse

La solution proposée a été implémentée sur une plateforme FPGA Xilinx ML605 embarquant un FPGA de la famille Virtex-6 (xc6vlx240t1156-1).

6.1.5.1 Surface

Tout d'abord, l'objectif est de mesurer l'impact en surface des différents blocs sur une architecture de base (un processeur, des IPs, des Block RAMs et des mémoires (Flash et DDR) pour arriver finalement au système complet sécurisé de la figure 6.3.

Dans cette architecture, le processeur généraliste GPP est un processeur softcore Microblaze qui tourne à une fréquence de 100 MHz et qui ne possède pas de caches, la mémoire externe est de type DDR3 et il y a également des contrôleurs Flash (pour le chargement du système d'exploitation) ainsi qu'un contrôleur DMA (pour accélérer le temps de boot [Devic *et al.* 2011]). Pour ce faire, on implémente l'architecture du système de base et on observe les surcoûts cumulés de chaque contribution visant à protéger le boot du Linux embarqué (contributions de [Devic *et al.* 2011]) et son exécution en temps réel (contributions présentées dans cette thèse, notamment dans les chapitres 3 et 4). Pour obtenir une architecture telle que celle de la figure 6.3, il faut implémenter 7 *Local Firewalls* et 1 *Cryptographic Firewall* (pour la protection de la mémoire externe DDR3).

	Slice regs	Slice LUTs	# Block RAMs
Architecture de base	8 950	9 179	46
+ Hard SHA-256	2 044 (22,83%)	2 452 (26,71%)	1 (2,17%)
+ DMA	534 (28,80%)	939 (36,94%)	4 (10,87%)
+ RSA-1024	684 (36,44%)	989 (47,72%)	4 (19,57%)
+ AES-GCM (Crypto Firewall)	2 161 (60,59%)	2 689 (77,30%)	15 (52,17%)
+ Local Firewall	123 (70,21%)	293 (99,36%)	1 (67,39%)

TABLE 6.1 – Surcoûts en surface

Dans le tableau 6.1, la surface de chaque bloc est donnée individuellement et le surcoût en surface cumulé est entre parenthèses. En ce qui concerne la protection du boot seul (SHA, DMA et RSA), on remarque que l'IP SHA-256 a une surface importante (elle provoque à elle seule un surcoût de plus de 20%). Dès lors, on peut se demander si une implémentation logicielle de cette fonction de hachage ne serait pas plus bénéfique pour la protection du boot du système d'exploitation : étant donné que l'objectif principal reste d'avoir des mécanismes à faible latence, la partie suivante montre les différences dans les domaines temporel et fréquentiel entre les implémentations logicielle et matérielle de la vérification de l'intégrité du noyau Linux.

Le module DMA sert également à accélérer la procédure de boot : si la surface devient un goulot d'étranglement dans le développement de ce type de système, il est possible de se passer du DMA pour charger le noyau du système d'exploitation depuis la mémoire non volatile vers la mémoire externe. En ce qui concerne la protection en temps réel ajoutée par les pare-feux, la majeure partie de la surface est dûe au *Cryptographic Firewall* : celui-ci embarque notamment un bloc de chiffrement AES-GCM qui, étant assez complexe, occupe une surface conséquente (voir chapitre 5). L'empreinte en surface des *Local Firewalls* reste tout à fait acceptable : le surcoût entre une version de l'architecture sans et avec pare-feux est au maximum de 13% (en termes de LUTs).

6.1.5.2 Latence

L'objectif principal des solutions proposées ici est d'avoir des mécanismes à faible latence. Si le boot du système d'exploitation est trop lent, ce n'est pas pratique pour l'utilisateur et cela peut même devenir critique en temps réel où des communications peuvent disparaître ou être dupliquées (voir les chapitres précédents). Les métriques pour évaluer la performance des différents modules dans le tableau 6.2 sont : le débit (cela permettra de s'adapter aux évolutions du noyau si la taille est modifiée) et le surcoût en latence du boot (le temps de boot du noyau sans mécanismes de protection sert de mesure de référence puis on observe les différences en ajoutant au fur et à mesure les différentes contributions proposées).

	Nombre de cycles	**Surcoût latence sur le temps de boot**	**Débit**	**Gain**
Soft SHA-256	295 860 775	2,959 s	0,95 MB/s	N/A, référence
Hard SHA-256	13 650 588	0,137 s	7,29 MB/s	$x7,7$
+ DMA	4 534 179	0,044 s	66,67 MB/s	$x70$

TABLE 6.2 – Surcoût des blocs dédiés au boot

Dans le tableau 6.2, on mesure ici les différentes options pour le boot avec vérification de l'intégrité du noyau Linux : autrement dit, on considère l'étude de la fonction de hachage ainsi que le transfert DMA (*Direct Memory Access*). La mesure de référence est une implémentation logicielle du hachage SHA-256. Comme on pouvait s'y attendre, l'implémentation matérielle est plus rapide (un rapport $x7$) et on améliore grandement ce rapport (on passe à un rapport de $x70$) si on ajoute le transfert DMA : dans ce cas le noyau Linux est copié directement de la mémoire non volatile Flash vers la mémoire externe DDR (on évite le traitement par le bootloader, mais on garde malgré tout la protection en confidentialité et en intégrité apportée par le *Cryptographic Firewall*).

Ensuite, pour la protection du stockage du noyau Linux (intervention du *Cryptographic Firewall* à l'interface de la mémoire externe), les performances de la fonction AES-GCM sont étudiées. Pour rappel, le boot d'un noyau Linux s'effectue en 280ms lorsqu'il n'y pas de mécanismes de protection [Devic *et al.* 2011]. Le tableau 6.3 présente les deux options possibles pour la protection de la mémoire externe qui sont définies en fonction du cahier des charges de l'utilisateur et de l'application visée : confidentialité-intégrité ou intégrité seule.

	Surcoût latence sur le temps de boot	Pourcentage	Throughput
Boot sans protection	280ms	référence	N/A
AES-GCM (C+I)	8ms	+ 2,86%	2,68 MB/s
AES-GCM (I only)	0,9ms	0,32%	23,49 MB/s

TABLE 6.3 – Surcoûts des options de chiffrement

Dans les deux cas, on obtient des surcoûts en latence très faibles (inférieurs à 3%), c'est une des raisons pour lesquelles l'algorithme AES-GCM a été choisi. Par contre, si on met en lien ces résultats avec le paragraphe précédent, on s'aperçoit que le débit du module AES-GCM est plus faible que le mécanisme de boot avec le transfert DMA activé : dans ce cas, AES-GCM devient un goulot d'étranglement et il faudrait trouver une solution ayant un débit plus important mais toujours avec une robustesse de l'algorithme cryptographique.

6.1.5.3 Résultats de benchmarks

Pour mesurer l'impact de la solution proposée sur la protection du système en temps réel (c'est-à-dire une fois que le système Linux est booté), on observe l'implantation d'applications de référence dans la mémoire externe : cette étude se base sur les applications benchmarks définies dans la section 5.1.3 ; même si elles n'ont pas de spécificités Linux ou POSIX, on considère qu'elles nous permettent d'illustrer ces aspects chiffrement d'applications témoins. D'un point de vue matériel, on considère le système suivant : une architecture monoprocesseur (avec un softcore Microblaze) qui exécute un Linux avec les mécanismes proposés dans [Devic *et al.* 2011], on y ajoute seulement un timer pour pouvoir faire les mesures en latence.

Les applications benchmarks sont stockées dans une mémoire non volatile (ce qui constitue une banque d'applications fournies par les développeurs où le système d'exploitation va « piocher » l'application qui l'intéresse) et vont être traitées par le *Cryptographic Firewall*, le transfert DMA est également activé. Pour chaque application, le temps de boot (i.e. d'écriture dans la mémoire externe) sert de mesure de référence. On observe ensuite les différents surcoûts en latence en fonction des différentes options cryptographiques possibles : confidentialité-intégrité (C+I), intégrité seule (I seule) ou texte clair (PT).

Application	Taille de code (octets)	Temps de boot (ms)	Surcoûts (ms)			Surcoût max (%)
			TC	C+I	I	
miBench						
basicmath	40 940	0,781	0,131	0,245	0,144	31,37%
bitcnt1	10 876	0,207	0,035	0,065	0,038	31,40%
bitcnt2	11 004	0,21	0,035	0,066	0,038	31,43%
bitcnt3	11 668	0,223	0,037	0,069	0,041	30,94%
bitcnt4	11 292	0,215	0,036	0,067	0,040	31,16%
bitstrng	11 068	0,211	0,035	0,066	0,039	31,28%
dijkstra	23 036	0,439	0,074	0,138	0,081	31,44%
stringsearch	15 476	0,295	0,049	0,092	0,054	31,19%
Custom						
choleski	56 564	1,079	0,181	0,338	0,199	31,33%
crc	26 492	0,505	0,085	0,159	0,093	31,49%
dft	41 924	0,8	0,134	0,251	0,147	31,38%
fft	33 284	0,635	0,106	0,199	0,117	31,34%
fir	21 908	0,418	0,07	0,131	0,077	31,34%
lu	40 948	0,781	0,131	0,245	0,144	31,37%
matrix	29 620	0,565	0,095	0,177	0,104	31,33%
nbody	29 396	0,561	0,094	0,176	0,103	31,37%
radix	18 916	0,361	0,06	0,113	0,066	31,30%
wht	17 316	0,33	0,055	0,103	0,060	31,21%

TABLE 6.4 – Surcoûts des applications benchmarks

Dans l'ensemble, les résultats du tableau 6.4 montrent que le boot sécurisé d'applications est réalisé avec des surcoûts en latence raisonnables (entre 16% et 32% selon l'option cryptographique choisie), par exemple :

– *dijkstra* (une implémentation de l'algorithme de Dijkstra) donne un surcoût de 31,34% si elle est protégée en confidentialité et en intégrité.
– Pour une application plus volumineuse (telle que *choleski*, une implémentation de la factorisation matricielle de Choleski), on obtient une augmentation de 31,33% de la latence.

Finalement, avec un module AES-GCM qui permet de faire de la cryptographie flexible (dans le sens où on peut facilement choisir le mode avec les politiques de sécurité), on a un mécanisme complet qui permet d'avoir un flot de chargement sécurisé à faible latence depuis le chargement du bitstream jusqu'à l'exécution de l'application.

6.2 Amélioration de la granularité de protection

6.2.1 Problématique

Dans l'implémentation actuelle des pare-feux (décrite dans les chapitre 3 et 4), les politiques de sécurité sont définies par rapport à un espace d'adresses donné : par exemple dans la figure 6.5, la politique P_1 couvre l'espace d'adresse $[0x00002000; 0x00002FFF]$.

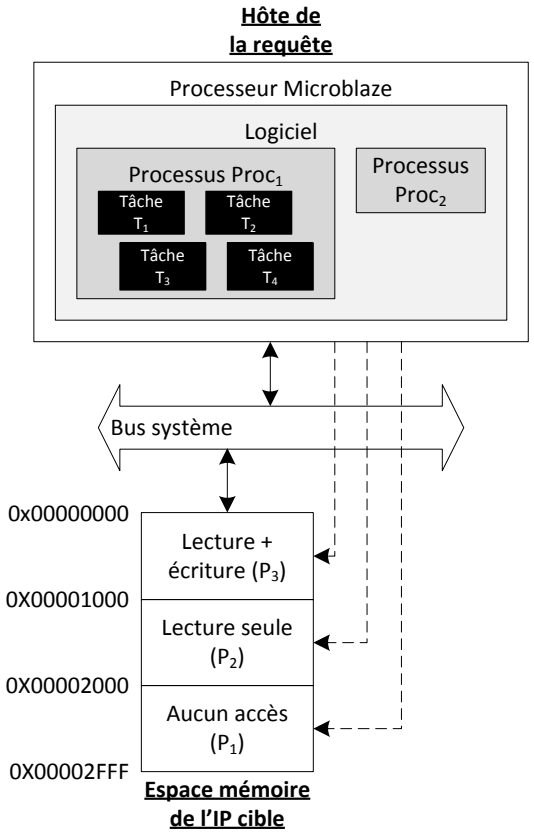


FIGURE 6.5 – Filtrage en fonction des adresses

La figure 6.5 représente un sous-ensemble d'un système multiprocesseur complet qui est représentatif de l'amélioration du niveau de granularité que l'on veut présenter : le processeur fait une requête vers une adresse appartenant à l'espace mémoire illustré dans la partie basse de la figure.

On considère que cette transaction est soumise à trois politiques de sécurité (en terme de droits de lecture-écriture uniquement) :
– P_3 : lecture et écriture sont autorisées entre les adresses $0x00000000$ et $0x00000FFF$.
– P_2 : seule la lecture est autorisée entre les adresses $0x00001000$ et $0x00001FFF$.
– P_1 : aucun accès n'est autorisé entre les adresses $0x00002000$ et $0x00002FFF$.
Le problème avec ce type d'approche est qu'il n'y a pas de spécification des politiques en fonction des processus : par exemples, les processus $Proc_1$ et $Proc_2$ peuvent tous les deux accéder aux mêmes politiques de sécurité. Il serait utile d'appliquer différentes politiques de sécurité en fonction du processus initiateur de la transaction. Cette fonctionnalité peut servir dans le cas où, dans un système multi-processus, un seul des processus est malveillant : tout le code étant stocké en clair (ou si l'attaquant a réussi à contourner les protections cryptographiques implantées), si l'attaquant modifie le code lié à un processus, on doit être capable de détecter les opérations illégales de ce processus sans pour autant bloquer les autres.

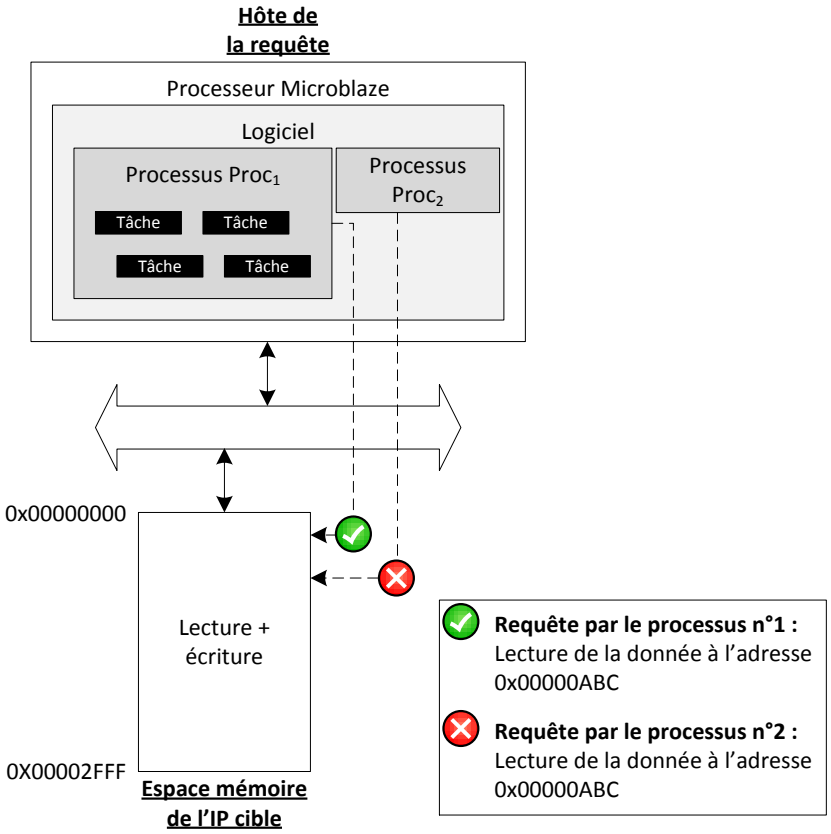


FIGURE 6.6 – Filtrage en fonction des tâches

Dans la figure 6.6, les deux processus $Proc_1$ et $Proc_2$ veulent lire une donnée à la même adresse (pas forcément simultanément) : d'après les politiques mises en place, le processus $Proc_2$ n'est pas autorisé à lire la donnée. Par conséquent, pour procéder à ce filtrage approfondi, il faut récupérer d'une manière ou d'une autre l'identifiant du processus.

6.2.2 Contexte

D'un point de vue logiciel, les systèmes multiprocesseurs que l'on considère sont gérés par un système d'exploitation : soit Xilkernel (le noyau minimal fourni dans les outils de développement Xilinx [Xilinx 2012b]), soit une distribution plus complète de type μCLinux[6]. Un des points communs de ces 2 solutions est le fait qu'elles soient compatibles avec le standard POSIX[7] qui fournit une batterie d'APIs pour la gestion des processus, des problématiques de temps réel et des processus légers (ou *threads*). Les processus sont en haut de la hiérarchie logicielle : un processus peut créer plusieurs threads (qui partagent le même espace mémoire) et les processus sont indépendants les uns des autres (contrairement aux threads). Dans tous les cas, les processus et les threads possèdent leur propre identifiant qui peut être appelé grâce à des routines dédiées dans le code de l'application (*get_pid*, *get_tid*...).

6.2.3 Solutions et analyse

Deux approches peuvent être envisagées pour l'identification des processus par les mécanismes de sécurité proposés dans ces travaux : récupération de l'identifiant de manière logicielle ou de manière matérielle (c'est-à-dire sans modification du système d'exploitation ni des applications). Dans la suite de cette section, les deux possibilités sont décrites et analysées mais il faut toujours garder à l'esprit que les mécanismes de protection implantés dans une architecture multiprocesseur doivent perturber le moins possibles les communications (autrement dit, la latence la plus faible possible) et un minimum de modifications sur l'architecture en elle-même (que ce soit du côte du logiciel ou du matériel).

6. http://www.uclinux.org/
7. http://en.wikipedia.org/wiki/POSIX

6.2.3.1 Solution logicielle

Tout d'abord, pour la solution logicielle, le but est de récupérer l'identifiant au début de la transaction entre le processeur et l'IP cible. Lorsque le système d'exploitation est compatible avec le standard POSIX, on peut utiliser une fonction dédiée $get\_pid()$ (pour un processus) ou $get\_tid()$ (pour un thread). Cette fonction renvoie l'identifiant du processus (ou du thread) courant qui peut alors être utilisé pour le traitement réalisé par les pare-feux. La figure 6.7 présente le flot d'identification des processus au sein d'un système sécurisé par les mécanismes de pare-feux.

La première étape consiste à récupérer l'identifiant du processus par appel de la routine dédiée (code de l'application dans l'encadré en haut à droite). Ensuite, cet identifiant est copié dans un registre dédié de 32 bits (noté *Registre ID* sur la figure 6.7). Enfin, le code du pare-feu (plus particulièrement le module *Security Builder*) est modifié afin de prendre en compte l'identifiant lors des contrôles : lors d'un accès à une adresse donnée (qui aura donc ses droits en lecture-écriture ainsi que son format accepté), le *Registre ID* est utilisé comme une entrée du *Checking Module*.

Par conséquent, cela induit également quelques modifications dans les politiques de sécurité : l'information sur les processus compatibles avec telle ou telle politique doit apparaître. Étant donnée la structure des politiques de sécurité dans les Block RAM (voir figure 6.8a), il y a suffisamment de bits de réserve pour pouvoir coder les processus associés à chaque politique de sécurité.

Par exemple, dans la figure 6.8b, les politiques de sécurité P_1 (respectivement P_2) adaptées à des *Local Firewall* sont compatibles avec les processus $proc_1$, $proc_2$ (respectivement $proc_1$, $proc_2$ et $proc_3$). Chaque identifiant de processus est codé sur 3 bits (ce qui autorise un maximum de 8 processus dans l'application). Comme les données dans les Block RAM sont contenues dans des mots de 32 bits, il n'y a pas de surcoût en occupation mémoire par l'amélioration de la granularité au niveau logiciel. Les seules modifications sont :
– L'exécution de la routine $get\_pid()$ dans le code de l'application.
– L'ajout d'un registre de 32 bits dans la description matérielle du pare-feu.

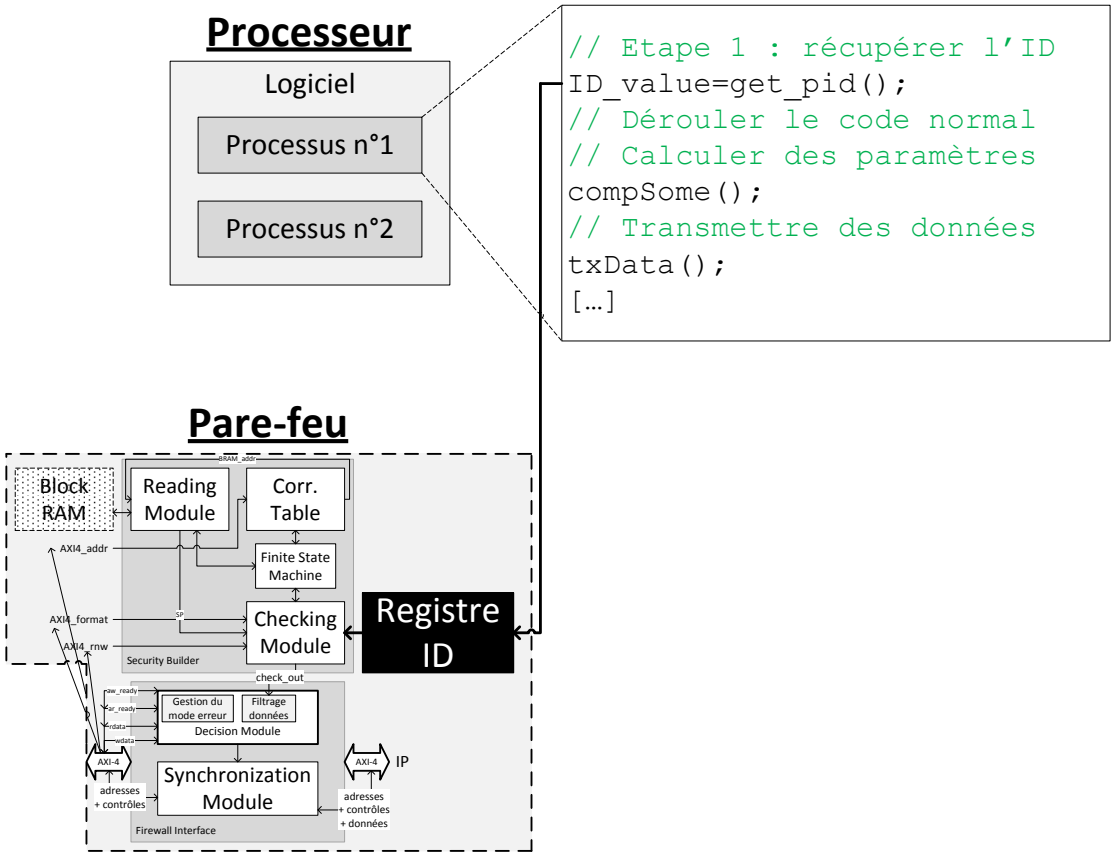
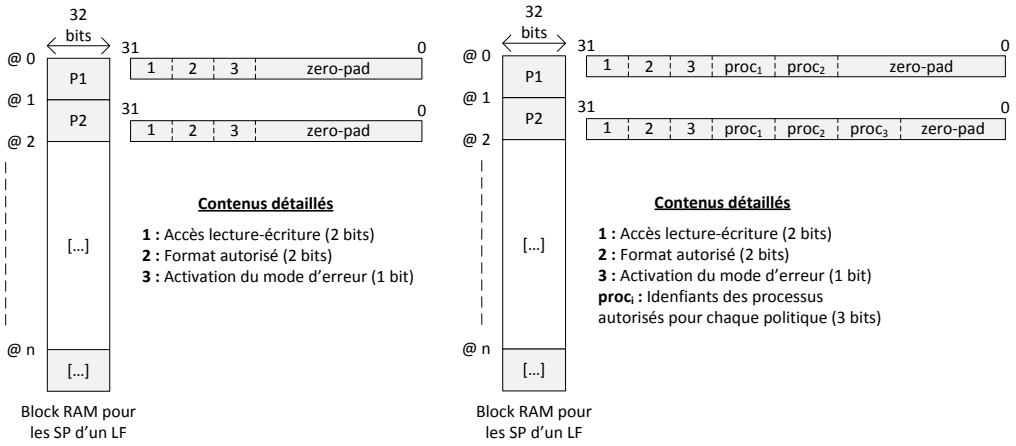


FIGURE 6.7 – Identification par solution logicielle



(a) Sans contrôle des processus

(b) Avec contrôle des processus

FIGURE 6.8 – Structure des politiques dans les Block RAM

6.2.3.2 Solution matérielle

Étant donné que les mécanismes de sécurité proposés dans ces travaux doivent minimiser les perturbations sur le système multiprocesseur, une solution à base de blocs matériels uniquement permettrait non seulement de minimiser le surcoût en latence mais également d'améliorer la portabilité : si l'application où le système d'exploitation sont mis à jour, les mécanismes de sécurité doivent être toujours compatibles avec le nouvel environnement. Dans l'implémentation du système de la figure 6.6, il existe deux points avec lesquels le développeur peut essayer d'interagir pour en extraire des informations qui permettraient d'identifier les différents processus qui sont exécutés par le processeur.
– Au niveau du bus (basé sur le protocole AXI).
– Au niveau du processeur (softcore Microblaze dans ce cas d'étude).
D'après les spécifications techniques du protocole AXI [ARM 2012], il doit être possible d'extraire un identifiant du processus ou de la tâche en cours d'exécution (notion de « multiple exclusive threads » évoquée dans la documentation) ; néanmoins cette possibilité parait relativement complexe et ne sera donc pas étudiée en détail.

Par contre, l'identification par l'intermédiaire du processeur est plus facile à implémenter. D'après la documentation constructeur [Xilinx 2012a], il existe un registre nommé le « Process Identifier Register (PIR) » (faisant partie des *Special Purpose Registers*) qui contient l'identifiant du processus qui est en train d'effectuer une transaction. Ce registre est consulté par la MMU (*Memory Management Unit*, ou unité de gestion mémoire) du processeur Microblaze lorsqu'elle doit transformer les adresses logiques en adresses physiques. Par conséquent, on considère qu'il suffit de lire l'information contenue dans le PIR et de la renvoyer vers le pare-feu pour analyser les droits associés au processus actuel.

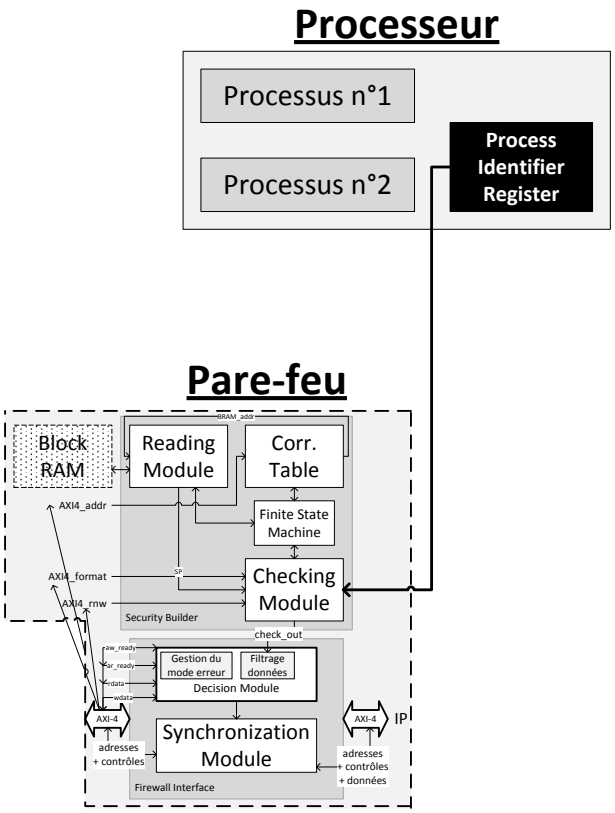


FIGURE 6.9 – Identification par solution matérielle

Le schéma de la figure 6.9 présente une implémentation simple qui permet d'extraire l'information d'identification des processus. Conformément a ce qui a été dit précédemment, la valeur contenue dans le registre PIR est renvoyée par une liaison point-à-point vers le bloc *Security Builder* pour aider à l'analyse des données. Autrement dit, dès qu'un processus veut effectuer une transaction, les données et adresses sont analysées normalement et l'identifiant du processus initiateur de la transaction est alors directement associé aux différents contrôles par l'intermédiaire de la liaison point-à-point avec le registre PIR du Microblaze. Dans ce cas, les modifications apportées par rapport à une protection par des pare-feux classiques sont :

– La modification des Block RAM (comme dans la solution logicielle) mais qui n'engendre pas de surcoût additionnel en termes de latence ou de surface.
– L'ajout d'une connexion point-à-point entre un registre existant du processeur et le pare-feu.

Cette solution a l'avantage de ne pas modifier la partie logicielle du système que l'on veut protéger.

6.2.3.3 Analyse des deux solutions

Pour comparer les deux solutions, on propose de regarder à la fois les surcoûts matériels et des aspects qualitatifs tels que l'impact sur une architecture existante de l'implantation de l'amélioration de la granularité au niveau processus. Le comparatif est résumé dans le tableau 6.5.

	Solution logicielle	**Solution matérielle**
Modifications sur l'architecture	Block RAM registre dans pare-feu	Block RAM liaison point-à-point
Modifications sur l'applicatif	Code de l'application -	-
Mise à jour d'application	Modification de la nouvelle application	-
Surcoût en surface	1 registre de 32 bits	Quasi nul
Surcoût en latence	Exécution d'une routine	Nul

TABLE 6.5 – Comparatif entre les solutions logicielle et matérielle

Il est possible de différencier le traitement réalisé par les pare-feux au niveau du processus que ce soit par une solution logicielle ou une solution matérielle. La solution matérielle est celle qui est la plus performante pour plusieurs raisons :

- Dans un objectif faible latence, la solution matérielle convient très bien : le seul ajout dans le chemin de données vient d'une liaison point-à-point qu'on suppose parfaite (c'est-à-dire n'ajoutant pas de délai dans le traitement réalisé par les pare-feux).
- Il n'y pas de modification de l'application ou du système d'exploitation : si des mises à jour logicielles ont lieu, la solution matérielle est toujours valable car elle extrait une information directement depuis un registre particulier du processeur.
- La solution logicielle paraît plus complexe à mettre en place : il faut analyser le code de l'application mais aussi modifier la structure du pare-feu pour y ajouter un registre qui contiendra temporairement l'information qui identifie le processus courant.
- Un des points négatifs de la solution matérielle est qu'elle n'est pas portable : étant donné qu'elle utilise des registres spécifiques au Microblaze, elle ne s'applique qu'à ce processeur ou qu'à ceux qui sont compatibles. par contre, la solution logicielle est totalement portable du moment que le système d'exploitation est compatible avec le standard POSIX.

6.3 Conclusion

Ce chapitre a démontré que l'utilisation des pare-feux n'était pas limitée à ce qui a été décrit dans les chapitres 3 et 4 : ils peuvent être utilisés pour donner une couche supplémentaire de protection à un flot existant de protection du boot d'un système d'exploitation de type Linux embarqué. De plus, il a été démontré que le niveau de granularité des contrôles effectués (au niveau de l'adresse des blocs IP) pouvait être étendu au niveau logiciel afin de définir différentes stratégies de communication en fonction du processus qui émet la requête. Malgré l'intégration de ces améliorations, les pare-feux gardent des temps de traitement satisfaisants et la mise à jour de la technologie (pour s'adapter à de nouvelles fonctions) n'est pas très compliquée.

Finalement, on a une structure flexible (les politiques sont stockées dans des Block RAMs) et il est possible de rajouter des connexions et des registres sans trop de modifications de la logique de contrôle et tout en maintenant des performances convenables.

7 Conclusions

La première partie de ce chapitre est consacrée à la présentation d'une perspective qui pourrait être intégrée dans les solutions proposées dans ce manuscrit. La deuxième partie propose une synthèse des différentes contributions développées pendant cette thèse.

Sommaire

7.1 Perspective : tolérance aux fautes

Dans le comportement classique d'un système, quelques données incorrectes dues à des erreurs de transmissions peuvent être transmises à travers l'architecture de communication. Dans l'implémentation actuelle de la version des pare-feux avec mise à jour, toutes les données incorrectes ont été détectées comme des attaques (suivies automatiquement par une reconfiguration des politiques à un niveau de sécurité supérieur). Cette fonctionnalité peut être améliorée pour distinguer les données incorrectes des attaques proprement dites. La métrique utilisée pour cette différenciation est le taux de données invalides sur un ensemble :

– Si le taux est supérieur à une valeur seuil, les données incorrectes sont considérées comme étant des attaques.
– Si le taux est inférieur à cette valeur seuil, les données erronées sont considérées comme des erreurs de transmissions.

Bien évidemment, chaque donnée qui provoque une erreur nécessite le blocage puis la mise à jour du pare-feu associé. S'il n'y a pas d'erreur dans les X données suivantes (X étant la valeur seuil), les politiques de sécurité peuvent être réglées à un niveau de sécurité plus permissif (par exemple, de « lecture seule » à « lecture-écriture »). La figure 7.1 montre le principe de l'implémentation matérielle de cette fonctionnalité de l'IP de surveillance (ou monitoring IP).

Cette fonctionnalité est basée sur un ensemble de compteurs cadencés par l'horloge du bus et activés par le signal *checkFlag* issu de chaque pare-feu. Quand une donnée est détectée comme étant erronée, le signal *checkFlag* passe à sa valeur haute. Une mise à jour est exécutée et le compteur est démarré. Lorsque que le compteur atteint la valeur seuil (*checkFlag* est à sa valeur basse pendant, par exemple, les 5 cycles suivants), le compteur est remis à zéro et une requête de mise à jour vers un niveau de sécurité plus permissif est envoyée au pare-feu. Cette information se symbolise par l'écriture d'un bit *updateLight (uL)* dans les bits de réserve des registres embarqués dans l'IP de surveillance (voir figure 7.1). Cette fonctionnalité est actuellement dans une version préliminaire et doit être analysé plus profondément, cependant un mécanisme de ce type est intéressant pour fournir à l'utilisateur final un système sécurisé avec une réduction des faux positifs.

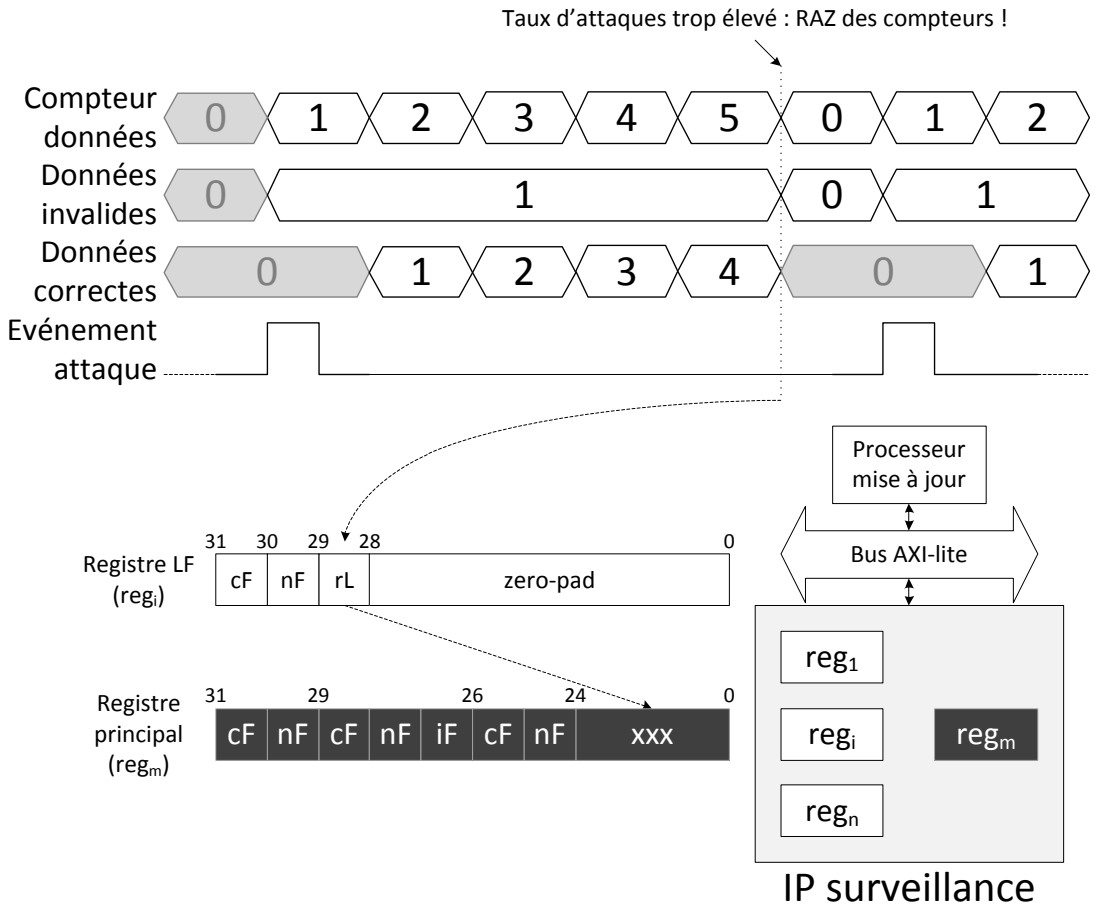


FIGURE 7.1 – Implémentation de la tolérance aux fautes dans l'IP de surveillance

7.2 Synthèse des contributions

Cette seconde partie du chapitre s'intéresse à la synthèse des contributions apportées par cette thèse. D'après les travaux existants décrits dans l'état de l'art, l'intérêt pour la protection des communications et des mémoires s'accroît depuis quelques années. Néanmoins, aucune approche n'a tenté de créer une combinaison efficace des fonctions cryptographiques qui servent à protéger les blocs mémoires externes avec les fonctions de surveillance et de contrôle des communications. L'approche par pare-feux décentralisés permet de fournir aux concepteurs de systèmes embarqués une sécurité qui protège à la fois les communications et les mémoires dans une architecture multiprocesseur.

L'implémentation des pare-feux proposée dans les chapitres 3 et 4 montre qu'il est possible de sécuriser le système en utilisant uniquement des blocs matériels. La partie logicielle ajoutée correspond uniquement à l'application implantée dans le processeur dédié aux fonctions de mise à jour des politiques de sécurité embarquées dans les pare-feux (détection des attaques, calcul des nouvelles configurations...). Globalement, pour implanter les mécanismes à base de pare-feux sur une architecture multiprocesseur existante, il y a quelques contraintes à respecter :

– Le protocole de communication doit être basé sur un bus AXI. Ceci limite donc notre solution à des architectures contenant peu de composants dans un premier temps (32 selon les spécifications du protocole).
– Au-delà de l'intégration des pare-feux, une architecture dédiée à la mise à jour doit être intégrée. Cette approche n'est pas valable si le système existant consomme une très grande partie des ressources matérielles disponibles sur le FPGA.
– Le développeur doit connaître la répartition de l'espace mémoire au niveau des adresses physiques. Dans le chapitre 3, on suppose que les implémentations se font sur un système d'exploitation simplifié où les adresses logiques sont identiques aux adresses physiques. Dans un cas plus général, le développeur doit avoir une maîtrise complète du système d'exploitation.

Ceci limite quelque peu le champ d'applications de l'approche par pare-feux matériels mais elle a au moins l'avantage de proposer une solution plus complète et plus « malléable » (dans le sens où la mise à jour des règles de sécurité est facilitée) que les solutions existantes.

L'un des atouts principaux de la solution par pare-feux est son efficacité en termes de latence. En effet, dans le chapitre 5, on remarque que sur un scenario complet impliquant une architecture multiprocesseur et un système d'exploitation, les surcoûts

en latence varient entre 4 et 17% (selon les options choisies en ce qui concerne les aspects cryptographiques). Par extension, l'implémentation proposée permet de ne pas pénaliser de manière trop importante le flot de chargement sécurisé proposé dans le chapitre 6. A priori, il y a deux goulots d'étranglement qui pénalisent un tant soit peu la solution proposée dans cette thèse.

Premièrement, d'un point de vue architectural, les implémentations étudiées ici se limitent à un bus simple qui pose un seuil quant à la « taille » de l'architecture (autrement dit, le nombre de composants qui peuvent se brancher sur ce médium de communication). Si on souhaite implanter les pare-feux sur une architecture à plus grande échelle, il existe deux solutions : des bus hiérarchiques ou un réseau sur puce (NoC). Le NoC paraît plus adapté car le chemin critique entre deux extrémités d'une architecture de communication avec des bus hiérarchiques peut vite devenir pénalisant. Étant donné que le protocole de mise à jour utilise des propriétés du protocole AXI, il faudrait donc intégrer dans chaque pare-feu un module de conversion AXI-NoC. Ceci implique forcément une latence supplémentaire qui n'est pas souhaitable. A priori, on peut imaginer un compromis en utilisant une solution avec des NoC hiérarchiques, similaire à ce qui est fait dans [Sepulveda *et al.* 2012b, Sepulveda *et al.* 2012a] : dans cette structure, chaque sous-NoC serait soumis à une politique de sécurité et la connexion avec le NoC principal se ferait à travers un bridge contenant un pare-feu vérifiant les règles de sécurité. Les communications internes à un sous-NoC seraient considérées de confiance alors que les accès aux mémoires externes se feraient uniquement par l'intermédiaire d'un pare-feu avec des fonctions cryptographiques connecté au NoC principal du système.

Deuxièmement, il y a également une contrainte par rapport au stockage du matériel cryptographique. Le modèle de menace décrit dans le chapitre 3 considère que les seuls blocs mémoires de confiance sont les mémoires internes du FPGA (Block RAM). L'impact en termes de consommation de mémoire provient en partie des clés utilisées dans la fonction de chiffrement : une approche avec uniquement deux clés est proposée dans le chapitre 5 afin de minimiser la quantité de mémoire interne nécessaire. Dans un second temps, toujours dans un objectif de faible latence, on a supposé que les tags (qui permettent de vérifier l'intégrité) étaient également stockés dans le FPGA. Étant donné notre modèle de menaces, si ces informations doivent être stockées en mémoire externe (pour augmenter la quantité de données qui peuvent être analysées), elles sont alors chiffrées ce qui implique un surcoût supplémentaire en termes de latence.

Il faut donc réaliser un compromis entre la latence ajoutée et la quantité de données maximale pouvant être protégée en intégrité : [Crenne 2011] propose une approche récente à base de filtres de Bloom qui permet d'avoir des taux de compressions intéressants. Une autre solution serait d'utiliser des structure du type arbre de Merkle mais nous n'avons pas eu le temps d'étudier en profondeur ces possibilités.

Concernant les extensions possibles, on a vu que les pare-feux peuvent être utilisés pour compléter un flot de boot existant [Devic *et al.* 2011] et obtenir alors un flot sécurisé complet depuis le chargement du bitstream. De plus, une deuxième extension permet (sous certaines conditions) de définir des politiques de sécurité en fonction de la tâche active : les pare-feux ne filtrent plus uniquement au niveau des adresses mais aussi en fonction des identifiants des tâches, ce qui permet d'améliorer la granularité des mécanismes de protection.

Finalement, même si la sécurité matérielle reste un domaine assez récent, il est possible de réaliser des fonctions efficaces en espérant que les améliorations qui vont arriver avec l'avancée de la technologie de fabrication des puces FPGA (multiplication des couches ou composants hybrides FPGA-processeur) vont permettre d'aller plus loin dans la protection des architectures multiprocesseurs.

Bibliographie

[Adriahantenaina *et al.* 2003] Adrijean Adriahantenaina, Hervé Charlery, Greiner Alain, Laurent Mortiez et Cesar Albenes Zeferino. *SPIN : a Scalable, Packet Switched, On-chip Micro-network*. In DATE (Design, Automation and Test in Europe Conference and Exhibition), pages 70–73, Munich, Allemagne, Mars 2003. IEEE.

[ARM 2012] ARM. *AMBA AXI and ACE Protocol Specification AXI3, AXI4, and AXI4-Lite, ACE and ACE-Lite*. Rapport technique, ARM, 2012.

[Badrignans *et al.* 2011] Benoît Badrignans, Jean-Luc Danger, Viktor Fischer, Guy Gogniat et Lionel Torres. Security Trends for FPGAS. Springer, 2011.

[Bertozzi & Benini 2006] Davide Bertozzi et Luca Benini. *Xpipes : A Network-on-Chip Architecture for Gigascale Systems-on-Chip*. IEEE Circuits and Systems Magazine, vol. 4, no. 2, pages 18–31, Septembre 2006.

[Bloom 1970] Burton H. Bloom. *Space/time trade-offs in hash coding with allowable errors*. ACM TECS (Transactions on Embedded Computing Systems), vol. 13, no. 7, pages 422–426, Juillet 1970.

[Chang *et al.* 2009] N.Y.-C. Chang, Y.-Z. Liao et T.-S. Chang. *Analysis of shared-link AXI*. IET Computers & Digital Techniques, vol. 3, no. 4, pages 373–383, Novembre 2009.

[Chaves *et al.* 2006] Ricardo Chaves, Georgi Kuzmanov, Leonel Sousa et Stamatis Vassiliadis. *Improving SHA-2 hardware implementations*. In CHES (Workshop on Cryptographic Hardware and Embedded Systems), pages 1–15, Yokohama, Japon, Octobre 2006. IACR.

[Coburn *et al.* 2005] Joel Coburn, Srivaths Ravi, Anand Raghunathan et Srimat Chakradhar. *SECA : Security-Enhanced Communication Architecture*. In CASES (International Conference on Compilers Architecture and Synthesis for Embedded Systems), pages 78–89, San Francisco, CA, Etats-Unis, Septembre 2005. IEEE.

[Cotret *et al.* 2010] Pascal Cotret, Jérémie Crenne et Guy Gogniat. *Sécurisation des communications dans une architecture multiprocesseur*. In MajecSTIC (MAni-

festation des JEunes Chercheurs en Sciences et Technologies de l'Information et de la Communication), pages 163–170, Bordeaux, France, Octobre 2010.

[Cotret *et al.* 2011] Pascal Cotret, Jérémie Crenne, Guy Gogniat, Jean-Philippe Diguet, Lubos Gaspar et Guillaume Duc. *Distributed security for communications and memories in a multiprocessor architecture.* In RAW (18th Reconfigurable Architectures Workshop), pages 326–329, Anchorage, AK, Etats-Unis, Mai 2011. IEEE.

[Cotret *et al.* 2012a] Pascal Cotret, Jérémie Crenne, Guy Gogniat et Jean-Philippe Diguet. *Bus-based MPSoC security through communication protection : A latency-efficient alternative.* In FCCM (20th Annual IEEE International Symposium on Field-Programmable Custom Computing Machines), Toronto, ON, Canada, Mai 2012. IEEE.

[Cotret *et al.* 2012b] Pascal Cotret, Florian Devic, Guy Gogniat, Benoît Badrignans et Lionel Torres. *Security enhancements for FPGA-based MPSoCs : a boot-to-runtime protection flow for an embedded Linux-based system.* In ReCoSoC (7th International Workshop on Reconfigurable Communication-centric Systems-on-Chip), York, Angleterre, Juillet 2012. IEEE.

[Cotret *et al.* 2012c] Pascal Cotret, Guy Gogniat, Jean-Philippe Diguet et Jérémie Crenne. *Lightweight reconfiguration security services for AXI-based MPSoCs.* In FPL (International Conference on Field Programmable Logic and Applications), Oslo, Norvège, Août 2012.

[Crenne *et al.* 2011a] Jérémie Crenne, Pascal Cotret, Guy Gogniat, Russell Tessier et Jean-Philippe Diguet. *Efficient key-dependent message authentication in reconfigurable hardware.* In FPT (International Conference on Field-Programmable Technology), pages 1–6, New Delhi, Inde, Décembre 2011. IEEE.

[Crenne *et al.* 2011b] Jérémie Crenne, Romain Vaslin, Guy Gogniat et Jean-Philippe Diguet. *Configurable Memory Security In Embedded Systems.* ACM TECS (Transactions on Embedded Computing Systems), vol. V, no. 0, pages 1–26, Septembre 2011.

[Crenne 2011] Jérémie Crenne. *Sécurité haut-débit pour les systèmes embarqués à base de FPGAs.* Manuscrit de thèse, Université de Bretagne-Sud, 2011.

[Dagon *et al.* 2004] David Dagon, Tom Martin et Thad Starner. *Mobile Phones as Computing Devices : The Viruses are Coming!* IEEE Pervasive Computing, pages 11–15, Décembre 2004.

[Devic *et al.* 2010] Florian Devic, Lionel Torres et Benoît Badrignans. *Secure protocol implementation for remote bitstream update preventing replay attacks on FPGA.* In FPL (International Conference on Field Programmable Logic and Applications), pages 179–182, Milan, Italie, Août 2010. IEEE.

[Devic *et al.* 2011] Florian Devic, Lionel Torres et Benoît Badrignans. *Securing Boot of an Embedded Linux on FPGA*. In RAW (18th Reconfigurable Architectures Workshop), pages 189–195, Anchorage, AK, Etats-Unis, Mai 2011. IEEE.

[Diguet *et al.* 2007] Jean-Philippe Diguet, Samuel Evain, Romain Vaslin, Guy Gogniat et Emmanuel Juin. *NOC-centric Security of Reconfigurable SoC*. In NOCS (International Symposium on Networks-on-Chip), pages 223–232, Princeton, NJ, Etats-Unis, Mai 2007. IEEE.

[Drimer 2008] Saar Drimer. *Volatile FPGA design security - A survey*. Rapport technique, University of Cambridge, Cambridge, Royaume-Uni, Avril 2008.

[Elbaz *et al.* 2005] Reouven Elbaz, Lionel Torres, Gilles Sassatelli, Pierre Guillemin, Claude Anguille, Christian Buatois et Jean-Baptiste Rigaud. *Hardware Engines for Bus Encryption : a Survey of Existing Techniques*. In DATE (Design, Automation and Test in Europe Conference and Exhibition), pages 40–45, Münich, Allemagne, Mars 2005.

[Elbaz *et al.* 2006a] Reouven Elbaz, Lionel Torres, Gilles Sassatelli, Pierre Guillemin, Michel Bardouillet et Albert Martinez. *A Comparison of Two Approaches Providing Data Encryption and Authentication on a Processor Memory Bus*. In PATMOS (Power And Timing Modeling, Optimization and Simulation), pages 267–279, Montpellier, France, Septembre 2006. Springer.

[Elbaz *et al.* 2006b] Reouven Elbaz, Lionel Torres, Gilles Sassatelli, Pierre Guillemin, Michel Bardouillet et Albert Martinez. *A Parallelized Way to Provide Data Encryption and Integrity Checking on a Processor-Memory Bus*. In DAC (Design Automation Conference), pages 506–509, San Francisco, CA, Etats-Unis, Juillet 2006. IEEE.

[Evain 2006] Samuel Evain. *µSpider. Environnement de Conception de Réseau sur Puce*. Manuscrit de thèse, Institut National des Sciences Appliquées de Rennes, 2006.

[Fiorin *et al.* 2007a] Leandro Fiorin, Gianluca Palermo, Slobodan Lukovic et Cristina Silvano. *A Data Protection Unit for NoC-based Architectures*. In CASES (International Conference on Compilers Architecture and Synthesis for Embedded Systems), pages 167–172, Salzbourg, Autriche, Octobre 2007. ACM.

[Fiorin *et al.* 2007b] Leandro Fiorin, Cristina Silvano et Mariagiovanna Sami. *Security Aspects in Networks-on-Chips : Overview and Proposals for Secure Implementations*. In DSD (Euromicro Conference on Digital System Design Architectures, Methods and Tools), numéro 7945 de 0, pages 7–10, Lübeck, Allemagne, Août 2007. IEEE.

[Fiorin *et al.* 2008a] Leandro Fiorin, Slobodan Lukovic et Gianluca Palermo. *Implementation of a Reconfigurable Data Protection Module for NoC-based MPSoCs*.

In IPDPS (International Parallel and Distributed Processing Symposium), pages 1–8, Miami, FL, Etats-Unis, Avril 2008. IEEE.

[Fiorin *et al.* 2008b] Leandro Fiorin, Gianluca Palermo et Cristina Silvano. *A Security Monitoring service for NoCs*. In CODES+ISSS (International Conference on Hardware/Software Codesign and system synthesis), pages 197–202, Atlanta, GA, Etats-Unis, Octobre 2008. ACM Press.

[Gaj & Chodowice 2001] Kris Gaj et Pawel Chodowice. *Fast implementation and fair comparison of the final candidates for Advanced Encryption Standard using Field Programmable Gate Arrays*. In RSA Security Conference, numéro October 2000 de 0, pages 1–16, San Francisco, CA, Etats-Unis, Avril 2001.

[Gaspar *et al.* 2010] Lubos Gaspar, Viktor Fischer, Florent Bernard, Lilian Bossuet et Pascal Cotret. *HCrypt : A Novel Concept of Crypto-processor with Secured Key Management*. In ReConFig (International Conference on ReConFigurable Computing and FPGAs), pages 280–285, Cancùn, Mexique, Décembre 2010. IEEE.

[Gassend *et al.* 2003] Blaise Gassend, Dwaine Clarke, Marten Van Djik, Srinivas Devadas et Ed Suh. *Caches and merkle trees for efficient memory authentication*. In HPCA (High Performance Computer Architecture Symposium), pages 1–14, Anaheim, CA, Etats-Unis, Février 2003.

[Gogniat *et al.* 2008] Guy Gogniat, Tilman Wolf, Wayne Burleson, Jean-Philippe Diguet, Lilian Bossuet et Romain Vaslin. *Reconfigurable Hardware for High-Security/High-Performance Embedded Systems : The SAFES perspective*. IEEE TVLSI (Transactioons on Very Large Scale Integration), vol. 16, no. 2, pages 144–155, Février 2008.

[Guilley & Pacalet 2004] Sylvain Guilley et Renaud Pacalet. *SoC security : a war against side-channels*. Annale des Télécommunications), vol. 59, no. 7–8, pages 998–1009, Juillet 2004.

[Huffmire *et al.* 2008] Ted Huffmire, Brett Brotherton, Nick Callegari, Jonathan Valamehr, Jeff White, Ryan Kastner et Tim Sherwood. *Designing Secure Systems on Reconfigurable Hardware*. ACM TODAES (Transactions on Design Automation of Electronic Systems), vol. 13, no. 3, pages 1–24, Juillet 2008.

[Järvinen *et al.* 2005] Kimmo Järvinen, Matti Tommiska et Jorma Skyttä. *Hardware Implementation Analysis of the MD5 Hash Algorithm*. In IEEE, editeur, HICSS (Hawaii International Conference on System Sciences), volume 0, pages 1–10, Hawaii, Etats-Unis, Janvier 2005.

[Kirsch & Mitzenmacher 2006] Adam Kirsch et Michael Mitzenmacher. *Less Hashing , Same Performance : Building a Better Bloom Filter*. Random Structures & Algorithms, vol. 33, no. 2, pages 456–467, 2006.

[Kocher *et al.* 2004] Paul Kocher, Ruby Lee, Gary McGraw, Anand Raghunathan et Srivaths Ravi. *Security as a New Dimension in Embedded System Design.* In DAC (Design Automation Conference), pages 753–760, San Diego, CA, Etats-Unid, Juin 2004. ACM Press.

[Lie *et al.* 2003] David Lie, Chandramohan Thekkath et Mark Horowitz. *Implementing an untrusted operating system on trusted hardware.* In SOSP (Symposium on Operating Systems Principles), pages 178–192, Bolton Landing, NY, États-Unis, Décembre 2003. ACM.

[McGrew & Viega 2004] David A McGrew et John Viega. *The Security and Performance of the Galois / Counter Mode (GCM) of Operation.* Rapport technique, Cisco Systems, 2004.

[Mencer *et al.* 1998] Oscar Mencer, Martin Morf et Michael J. Flynn. *Hardware software tri-design of encryption for mobile communication units.* In ICASSP (International Conference on Acoustics, Speech and Signal Processing), pages 3045–3048, Seattle, WA, États-Unis, Mai 1998. IEEE.

[Merkle 1987] Ralph Merkle. *A Digital Signature Based on a Conventional Encryption Function.* In CRYPTO (A Conference on the Theory and Applications of Cryptographic Techniques on Advances in Cryptology), pages 369–378, Santa Barbar, CA, Etats-Unis, Août 1987. Springer-Verlag.

[Mitic & Stojcev 2006] Milica Mitic et Mile Stojcev. *An Overview of On-Chip Buses.* Facta Universatis (NIS) - Series : Electronics and Energetics, vol. 19, no. 3, pages 405–428, Décembre 2006.

[Nash *et al.* 2005] Daniel Nash, Thomas Martin, Dong Ha et Michael Hsiao. *Towards an intrusion detection system for battery exhaustion attacks on mobile computing devices* . In PERCOMW (Pervasive Computing and Communications Workshops), pages 141–145, Chengdu, Chine, Mars 2005. IEEE.

[Peeters *et al.* 2007] Eric Peeters, François-Xavier Standaert et Jean-Jacques Quisquater. *Power and electromagnetic analysis : improved model, consequences and comparisons.* Elsevier Integration, the VLSI Journal - Special issue : Embedded cryptographic hardware, vol. 40, no. 1, pages 52–60, Janvier 2007.

[Plouviez 2011] Geoffrey Plouviez. *Etude, spécification, vérification formelle de mécanismes de virtualisation sécurisés pour architecture many-cores.* Manuscrit de thèse, Université Pierre et Marie Curie, 2011.

[Poletti *et al.* 2003] Francesco Poletti, Davide Bertozzi, Luca Benini et Alessandro Bogliolo. *Reconfigurable Hardware for High-Security/High-Performance Embedded Systems : The SAFES perspective.* DAES (Design Automation for Embedded Systems), vol. 8, no. 2, pages 189–210, Juin 2003.

[Porquet 2010] Joël Porquet. *Architecture de sécurité dynamique pour systèmes multi-processeurs intégrés sur puce*. Manuscrit de thèse, Université Pierre et Marie Curie, 2010.

[Publication 2001] Federal Information Processing Standards Publication. *Advanced Encryption Standard (AES)*. Rapport technique, FIPS, 2001.

[Ravi *et al.* 2004] Srivaths Ravi, Anand Raghunathan, Paul Kocher et Sunil Hattangady. *Security in embedded systems : Design challenges*. ACM TECS (Transactions on Embedded Computing Systems), vol. 3, no. 3, pages 461–491, Août 2004.

[Sepulveda *et al.* 2012a] Johanna Sepulveda, Guy Gogniat, Cesar Pedraza, Ricardo Pires, Wang Jiang Chau et Marius Strum. *Hierarchical NoC-based security for MP-SoC dynamic protection*. In LASCAS (Latin American Symposium on Circuits And Systems), pages 1–4, Playa del Carmen, Mexique, Mars 2012. IEEE.

[Sepulveda *et al.* 2012b] Johanna Sepulveda, Ricardo Pires, Guy Gogniat, Wang Jiang Chau et Marius Strum. *QoSS hierarchical NoC-based architecture for MPSoC dynamic protection*. International Journal of Reconfigurable Computing, vol. 3, no. 2012, pages 1–10, Janvier 2012.

[Suh 2005] Gookwon Edward Suh. *AEGIS : A Single-Chip Secure Processor*. Information Security Technical Report, vol. 10, pages 63–73, Janvier 2005.

[Vaslin *et al.* 2008] Romain Vaslin, Guy Gogniat, Jean-Philippe Diguet, Russell Tessier, Deepak Unnikrishnan et Kris Gaj. *Memory Security Management for Reconfigurable Embedded Systems*. In FPT (International Conference on Field-Programmable Technology), pages 153–160, Taipei, Taiwan, Décembre 2008. IEEE.

[Xilinx 2011a] Xilinx. *AXI Reference Guide*. Rapport technique, Xilinx Corporation, 2011.

[Xilinx 2011b] Xilinx. *Virtex-6 FPGA Configuration User Guide (UG360)*. Rapport technique, Xilinx corporation, 2011.

[Xilinx 2011c] Xilinx. *Virtex-6 FPGA Memory Interface Solutions - User Guide*, Mars 2011. Disponible à l'adresse : http://www.xilinx.com/support/documentation/ip_documentation/ug406.pdf.

[Xilinx 2011d] Xilinx. *Virtex-6 FPGA Memory Resources*, Avril 2011. Disponible à l'adresse : http://www.xilinx.com/support/documentation/user_guides/ug363.pdf.

[Xilinx 2012a] Xilinx. *MicroBlaze Processor Reference Guide - Embedded Development Kit EDK 14.2*, Juillet 2012. Disponible à l'adresse : www.xilinx.com/support/documentation/sw_manuals/xilinx14_2/mb_ref_guide.pdf.

[Xilinx 2012b] Xilinx. *OS and Libraries Document Collection*, Juillet 2012. Disponible à l'adresse : http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_2/oslib_rm.pdf.

[Xilinx 2012c] Xilinx. *Virtex-6 Family Overview*, Janvier 2012. Disponible à l'adresse : http://www.xilinx.com/support/documentation/data_sheets/ds150.pdf.

[Xilinx 2012d] Xilinx. *Zynq-700 EPP overview*, Juin 2012. Disponible à l'adresse : http://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf.

[Xu *et al.* 2008] Yan-Ling Xu, Wei Pan et Xin-Guo Zhang. *Design and Implementation of Secure Embedded Systems Based on TrustZone*. In ICESS (International Conference on Embedded Software and Systems), pages 136–141, Chengdu, Chine, Juillet 2008. IEEE.