Méthodes et modèles pour la vérification formelle de l'attestation à distance sur microprocesseur

Jonathan Certes

THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par : *l'Université Toulouse 3 Paul Sabatier (UT3 Paul Sabatier)*

Présentée et soutenue le *22/06/2023* **par :**

Jonathan CERTES

Méthodes et modèles pour la vérification formelle de l'attestation à distance sur microprocesseur

---

JURY

Aurélien FRANCILLON	Professeur des universités	Rapporteur
Marie-Laure POTET	Professeur des universités	Rapporteur
Eric ALATA	Maître de conférences	Examinateur
Sébastien BARDIN	Senior Researcher	Examinateur
Abdelmalek BENZEKRI	Professeur des universités	Examinateur
Guy GOGNIAT	Professeur des universités	Examinateur
Philippe QUEINNEC	Professeur des universités	Examinateur
Benoît MORGAN	Maître de conférences	Directeur de thèse

École doctorale et spécialité :
MITT : Domaine STIC : Sureté de logiciel et calcul de haute performance

Unité de Recherche :
Institut de Recherche en Informatique de Toulouse (IRIT)

Directeur(s) de Thèse :
Benoît MORGAN

Rapporteurs :
Aurélien FRANCILLON et *Marie-Laure POTET*

Remerciements

Les travaux présentés dans ce manuscrit ont été effectués à l'Institut de Recherche en Informatique de Toulouse (IRIT), Unité Mixte de Recherche multi-tutelle : Centre National de la Recherche Scientifique (CNRS) et Universités Toulousaines, au sein du site INPT. Je remercie André-Luc Beylot qui a assuré la responsabilité du site INPT de l'IRIT depuis mon arrivée.

Je remercie également Philippe Queinnec puis Jan-Georg Smaus, responsables de l'équipe ACADIE (Assistance à la Certification d'Applications DIstribuées et Embarquées) de m'avoir permis de réaliser ces travaux au sein du groupe.

Je tiens à exprimer ma profonde reconnaissance et un grand respect à Yamine Aït-Ameur, Professeur à l'INPT et responsable du département Numérique et Mathématiques de l'ANR (Agence Nationale de la Recherche), sans qui cette thèse n'aurait jamais pu voir le jour. Malgré sa charge de travail importante, Yamine s'est rendu disponible pour répondre à mes interrogations et inquiétudes, a su m'orienter dans les choix stratégiques le long de ma thèse et s'est impliqué dans mon travail sans souhaiter être crédité pour ses contributions.

Je suis également profondément reconnaissant envers Vincent Nicomette, Professeur des universités à l'INSA de Toulouse, dont la contribution est plus qu'apparente dans ce manuscrit, mais qui ne souhaite pas non plus être crédité. Malgré une charge de travail importante, Vincent a su se rendre disponible aux instants critiques de la rédaction des publications et du manuscrit ; ses retours ont toujours été constructifs et pertinents.

Bien évidemment, je souhaite exprimer ma reconnaissance infinie envers Benoît Morgan, chercheur en sécurité informatique à INTEL et enseignant-chercheur associé à l'INPT, pour m'avoir encadré pendant ma thèse. Benoît a su diriger ma thèse avec brio : il a su encadrer les travaux et m'orienter dans une direction cohérente, tout en me laissant libre des choix de mes recherches et de la manière d'aborder les défis techniques. Je tiens à souligner son fort investissement dans les travaux décrits dans ce manuscrit, qui représentent le fruit d'une véritable collaboration ; ses qualités techniques et scientifiques remarquables ont été un atout pour leur réalisation. Je souligne également ses qualités humaines et son enthousiasme, générateurs d'un environnement empli de sincérité et de bienveillance. Ceci est d'autant plus vrai que mon intérêt pour la sécurité informatique et mes connaissances puisent leurs sources dans les enseignements de Benoît.

J'adresse également mes sincères remerciements aux membres du jury qui ont accepté de juger mon travail :

— Aurélien Francillon, Professeur des universités à EURECOM Sofia-Antipolis ;
— Marie-Laure Potet, Professeur des universités à l'INP Grenoble ;
— Eric Alata, Maître de conférences à l'INSA Toulouse ;
— Sébastien Bardin, *Senior Researcher* au CEA Palaiseau ;
— Abdelmalek Benzekri, Professeur des universités à l'Université Toulouse III ;
— Guy Gogniat, Professeur des universités à l'Université Bretagne Sud ;
— Philippe Queinnec, Professeur des universités à l'INP Toulouse ;
— Benoît Morgan, Maître de conférences à l'INP Toulouse

Je remercie particulièrement Aurélien Francillon et Marie-Laure Potet qui ont accepté d'être

rapporteurs de ma thèse. Une attention particulière à Aurélien Francillon et Sébastien Bardin qui sont auteurs des travaux qui ont nourri mon intérêt pour la recherche en sécurité informatique et la vérification formelle.

Je remercie tous les membres de l'IRIT avec qui j'ai partagé de très bons moments, enrichissants scientifiquement, mais aussi humainement. Merci à tous les membres de l'équipe ACADIE que j'ai, avec grand plaisir, côtoyés au quotidien ou au travers de visio-conférences. Je remercie particulièrement Yamine Aït-Ameur, Neeraj Singh, Marc Pantel, Philippe Queinnec et Aurélie Hurault pour leur support lorsque j'en avais le plus besoin. Merci à mes chers collègues doctorants : Adam Shimi, Ismail Mendil et Peter Riviere. Un grand merci à Guillaume Dupont, aujourd'hui Maître de conférences à l'INPT, pour nos discussions passionnantes, ses enseignements et ses contributions dans les travaux présentés dans ce manuscrit.

Je tiens à exprimer ma gratitude à l'ensemble du personnel du Département Informatique de l'INP-ENSEEIHT, qui m'a donné l'opportunité d'enseigner durant ces années de thèse. Enseigner au sein de l'INP-ENSEEIHT a été une formidable expérience et je tiens, à ce propos, à remercier les enseignants de m'avoir fait confiance et d'effectuer un travail pédagogique excellent. En particulier, merci à Benoît Morgan, Ronan Guivarch, Philippe Mauran, André-Luc Beylot et Geraldine Morin. Un grand merci à Agnès Requis, Annabelle Sansus, Sylvie Armengaud et Vanessa Adjeroud qui m'ont toujours, et avec bonne humeur, permis de surmonter mon incapacité administrative symptomatique.

Je remercie ma famille qui m'a toujours soutenu, quelles que soient les circonstances et durant les moments difficiles. Je remercie mes parents de m'avoir transmis la curiosité, l'envie d'apprendre, et d'avoir mis en œuvre tous les moyens pour que je puisse les assouvir. Je reste intimement convaincu de mon privilège vis-à-vis de mon éducation. J'espère les avoir rendus fiers. Je remercie ma compagne Amanda pour son soutien inconditionnel ; elle qui traversait comme moi une épreuve difficile sur le plan professionnel, je suis convaincu que nous n'aurions pu aboutir l'un sans l'autre.

Merci à Choüch et à mon idole, qui me prouvent régulièrement que la véritable amitié résiste au temps et à la distance, peu importe les chemins que nous empruntons dans la vie.

Merci à mes amis proches pour tout ce qu'ils sont, ne changez rien : Babou, Benlord, Juju, Aurélien, Otávio, Yvan. C'est quand le DOTR ?

Merci à Juju pour ces longues soirées de relecture : leur étendue n'est qu'inversement proportionnelle à mes capacités sémantiques en langage naturel.

Table des matières

Introduction

Les systèmes informatiques ont aujourd'hui atteint un niveau de complexité spectaculaire et le risque lié aux malveillances n'a jamais été aussi élevé. En effet, la puissance de calcul des architectures matérielles n'a cessé d'augmenter en même temps que les performances atteintes par les réseaux. Aussi, ce gain de puissance a permis l'introduction de nouveaux paradigmes tels que la virtualisation assistée par le matériel, la virtualisation légère, les architectures multicœurs, l'intégration de circuits reprogrammables, etc. Les concepteurs de systèmes d'information ont par opportunité naturellement usé de ces paradigmes pour mutualiser et densifier de plus en plus l'utilisation de ressources matérielles. Cette densification mène à héberger nombre d'applications hétérogènes complexes, parmi lesquelles certaines sont potentiellement malveillantes ou corrompues. Ces précédents aspects atteignent leur paroxysme notamment avec les architectures de type cloud, les téléphones portables, etc. Par conséquent, afin de concevoir des systèmes raisonnablement sécurisés, il est devenu quasi indispensable de considérer des modèles d'adversaires très puissants et intrusifs.

Vis-à-vis de ces modèles d'adversaires, garantir la sécurité de l'exécution de logiciel sur un système complexe est un problème difficile. Tout d'abord, son algorithme doit être vérifié de la conception à l'implémentation, afin d'en éliminer les potentielles vulnérabilités. Puis, le système sur lequel il s'exécute doit lui aussi être vérifié pour garantir l'intégrité de son environnement d'exécution, afin de ne pas mettre à mal les propriétés de sécurité précédemment vérifiées. Malheureusement, les systèmes d'information modernes et leur contexte industriel sont aujourd'hui d'une complexité telle qu'il apparait très difficile, voire impossible, d'appliquer efficacement un schéma de vérification formelle complet. Il existe donc deux grandes approches nécessaires et complémentaires pour faire face à ce constat : l'approche tolérance aux intrusions et l'approche élimination des vulnérabilités.

La tolérance aux intrusions propose des mécanismes permettant de garantir un ensemble d'attributs de sécurité même dans le cas d'une intrusion réussie. Ces mécanismes peuvent consister, par exemple, en une reconfiguration dynamique du logiciel et / ou du matériel pour s'affranchir de la vulnérabilité utilisée pour l'intrusion.

L'élimination des vulnérabilités consiste en la vérification exhaustive du système pour s'assurer de l'absence de vulnérabilités préalablement identifiées. Ces mécanismes sont déployés sur les systèmes avant leur mise en production de sorte que les intrusions ne peuvent pas réussir.

Dans le cadre de cette thèse, nous nous focalisons sur l'élimination des vulnérabilités dans une architecture co-conçue, logicielle et matérielle. Nous proposons de traiter la problématique de l'attestation à distance et de l'ensemble minimaliste d'éléments (logiciels et matériels) qui lui sont considérés de confiance. L'attestation à distance est un problème de sécurité dans lequel un *vérifieur* veut vérifier l'intégrité d'un algorithme en place sur une machine distante, *prouveur*, qui peut être corrompue. Nous exprimons formellement les propriétés de sécurité attendues du système. Par exemple, les schémas d'attestation à distance utilisent en général un secret partagé qui permet à une fonction d'attestation d'authentifier un tag d'intégrité. La sécurité de l'architecture repose, entre autres, sur la confidentialité de ce secret. La confidentialité est donc une propriété à vérifier. D'autres propriétés sont aussi essentielles telles que l'atomicité de l'exécution de la fonction d'attestation et l'immuabilité de son code. Une fois nos propriétés de sécurité définies, nous les confrontons à une implémentation sur un système moderne, réaliste

et fonctionnel, que nous modélisons. En effet, il est facile de démontrer la sécurité d'un système non communicant ou qui ne s'exécute pas : nous devons nous assurer que notre modèle est suffisamment proche de la réalité. Les propriétés de sécurité sont vérifiées sur un modèle d'une sous partie représentative. Ce modèle est obtenu par traduction automatique de l'architecture matérielle, décrite dans un langage de description matérielle, vers un formalisme outillé sur lequel nous pouvons efficacement vérifier nos propriétés de sécurité. Enfin, nous modélisons les interactions auxquelles notre système sera confronté, c'est-à-dire notre modèle de menaces. Il est nécessaire de formellement définir le comportement d'un adversaire : les propriétés de sécurité ne seront vérifiées que dans ce cadre. Par exemple, en ce qui concerne l'attestation à distance, nous considérons un adversaire capable d'exécuter arbitrairement du code sur la machine à vérifier, le prouveur, sans être capable d'y accéder physiquement. La preuve de sécurité est obtenue si nous arrivons à démontrer que les propriétés de sécurités définies sont respectées pour notre modèle de système vis-à-vis de notre modèle de menaces. Aussi, afin de prouver notre concept, nous concevons, à l'aide des méthodes précédentes, des systèmes en co-conception logicielle et matérielle. Ces systèmes sont implémentés dans un environnement fonctionnel, sur lequel il est possible de jouer de réelles attaques ou encore d'évaluer l'impact sur les performances causé par l'ajout des mesures de sécurité.

Ce manuscrit s'articule ainsi ; dans le premier chapitre, nous présentons le contexte de nos travaux en détaillant les paradigmes des systèmes d'information modernes. Nous définissons les concepts de vulnérabilité, d'attaque et d'intrusion, utiles à la compréhension de cette thèse. Nous illustrons ces concepts par des exemples sur les architectures modernes, pour en déduire notre modèle de menaces. Afin d'argumenter efficacement sur la sécurité de telles architectures, nous présentons les mécanismes de tolérance aux intrusions et d'éliminations des vulnérabilités tels qu'adoptés dans l'état de l'art. Nous détaillons le fonctionnement de l'attestation à distance et ses prérequis en matière de sécurité vis-à-vis du modèle de menaces adopté. Cette description est accompagnée d'une présentation de l'état de l'art en matière de racine de confiance statique, de solutions d'architectures co-conçues et de vérification formelle pour la sécurité. Nous terminons le chapitre en détaillant une solution d'attestation à distance pour microcontrôleur formellement vérifiée, qui forme une base pour nos travaux, et nous présentons les objectifs de la thèse.

Les second et troisième chapitres fournissent des rappels techniques, respectivement sur le fonctionnement des architectures modernes qui constituent notre environnement de développement et la stratégie de vérification formelle que nous envisageons. Dans le second chapitre, nous décrivons la procédure envisagée pour proposer une extension co-conçue à un système moderne existant, pris sur étagère. Le fonctionnement des périphériques, utiles à la communication avec cette extension, est également décrit. Dans le troisième chapitre, nous décrivons les méthodes formelles, de modélisation et de vérification, que nous utilisons. Celles-ci s'accompagnent d'une introduction des outils de traduction et des formalismes de modélisation. La problématique du passage à l'échelle de la vérification formelle est elle aussi introduite.

Le quatrième chapitre présente notre première contribution : une méthode de vérification formelle automatique et industrialisable des systèmes matériels. La description de cette méthode est accompagnée de la publication d'un ensemble d'outils entièrement basé sur des logiciels libres. Cette solution permet une vérification de nos propriétés de sécurité sur notre extension matérielle, tout en se prémunissant de l'explosion combinatoire, inhérente au passage à l'échelle.

Le cinquième chapitre est dédié à la présentation de la contribution majeure de cette thèse :

une méthode de modélisation et une architecture de sécurité permettant d'étendre un microprocesseur moderne avec une racine de confiance statique dédiée à l'attestation à distance. Cette racine de confiance est co-conçue à l'aide de logiciel et de matériel ; une implémentation est proposée. Nous proposons une stratégie de vérification formelle de cette architecture et une première implémentation, dont nous vérifions la sécurité vis-à-vis d'un modèle de menaces simplifié. Le cœur de la contribution réside dans le fait que l'architecture proposée est adaptée aux microprocesseurs pris sur étagère, dont aucune modification matérielle n'est possible. L'extension matérielle, en revanche, est adaptable à une potentielle évolution du modèle de menaces.

Le sixième chapitre décrit une nouvelle contribution : la proposition d'une solution d'attestation de configuration du microprocesseur dans un environnement corrompu. Ici, nous intégrons au modèle de menaces des capacités de l'adversaire que nous avions précédemment exclues. En d'autres termes, nous étendons notre racine de confiance statique pour faire face à un adversaire plus fort, raffinant notre modèle de menaces. Notre solution d'attestation requiert le développement d'un algorithme de confiance, dont l'architecture est hautement dépendante du microprocesseur ciblé et de l'extension matérielle choisie. Ici aussi, une implémentation est proposée.

Enfin, le dernier chapitre de ce manuscrit propose des extensions supplémentaires, afin d'étendre encore les capacités de l'adversaire et raffiner davantage notre modèle de menaces, sans toutefois apporter de garantie soutenue par une vérification formelle. L'objectif de ce chapitre est de raisonner efficacement à propos du réalisme de la solution que nous proposons pour l'attestation à distance sur microprocesseur.

Ce manuscrit est accompagné de ressources complémentaires, modèles et outils, publiés sous licence libre et rendus accessibles publiquement [1]. Ces ressources permettent au lecteur une implémentation du système, une étude des modèles que nous proposons et un rejeu des vérifications formelles.
Dans les différents chapitres, nous proposons des expériences illustrant nos travaux. Ces expériences sont proposées dans des encadrés comme celui-ci. Des documents annexes fournissent des détails techniques au sujet de ces expériences.
L'ensemble des outils logiciels requis pour rejouer les expériences est en accès gratuit. Nous tentons au maximum d'avoir recours à des logiciels libres. Le nom de chacun de ces outils est mentionné dans ce manuscrit, lors de son introduction. Afin d'implémenter l'extension matérielle que nous proposons sur un système physique, une carte de développement Digilent Cora Z7-07S est nécessaire.

Chapitre 1

Contexte scientifique et État de l'art

Sommaire

Dans ce manuscrit, nous proposons des méthodes et modèles pour la vérification formelle de l'attestation à distance sur microprocesseur. Afin de contextualiser nos travaux, nous introduisons dans ce chapitre les paradigmes des systèmes d'informations modernes et nous rappelons les définitions des concepts de vulnérabilités, d'attaques et d'intrusions. Nous rappelons brièvement les travaux concernant la sécurisation d'un algorithme sur un système moderne et complexe, puis nous en déduisons un modèle de menace en cohérence avec les différentes problématiques rencontrées. Ensuite, nous proposons un état de l'art dans les domaines de l'attestation à distance et de la vérification formelle pour la sécurité. Nous détaillons plus particulièrement de précédents travaux dans le domaine de l'attestation à distance vérifiée formellement, qui servent de base à nos travaux sur des architectures plus complexes. Finalement, nous décrivons les objectifs de la thèse, auxquels nous tentons de répondre dans ce manuscrit.

1.1 Paradigmes des systèmes d'informations modernes

La complexité des systèmes informatiques modernes n'a fait qu'augmenter ces dernières décennies. La puissance de calcul des architectures matérielles n'a cessé d'augmenter et la surface physique des circuits électroniques n'a fait que réduire, entraînant une réduction des coûts de fabrication.

Cette augmentation de puissance et cette réduction des coûts a poussé les constructeurs de matériels à la course à la performance, dans laquelle l'objectif primordial est mettre rapidement un système sur le marché. La conséquence d'une telle course est que la conception est finalement réalisée avec une prise en compte faible voire inexistante de la sécurité, menant à l'apparition de vulnérabilités pouvant être exploitées dans le cadre d'attaques exploitant des faiblesses dans le matériel. Les exemples les plus parlants sont les classes d'attaques telles que Spectre [2], Meltdown [3] ou encore Rowhammer [4], qui démontrent que la vulnérabilité des ordinateurs grand public est réelle et demande un changement de paradigme rapide dans la prise en compte de la sécurité.

Les systèmes matériels sont complexes à sécuriser dans la mesure où proposer une correction de faille de sécurité est souvent très difficile, voire impossible si l'architecture n'a pas été conçue pour être modulaire. En effet, les attaques ciblant les couches basses du matériel exploitent directement les défauts de fabrication des composants. Dans certains cas, il est possible de proposer une correction uniquement en désactivant purement et simplement les accélérations matérielles, au détriment des performances.

Meltdown, par exemple, exploite le fait que les processeurs optimisent l'utilisantion du *pipeline* en exécutant les instructions dans le désordre (exécution *out-of-order*). Lors de l'exécution out-of-order, le processeur va exécuter des instructions potentiellement illégitimes, en amont de la détection d'une faute (*transient instructions*). Un adversaire a donc l'opportunité d'exécuter effets de bord biens choisis pour encoder de l'information (dans le cache). Cette information est par la suite observée à l'aide d'une canal caché (*covert channel*).

La contre-mesure adoptée par la communauté Linux, KAISER [5], a revu la cartographie de l'espace mémoire privilégié du système d'exploitation (appelé noyau) lorsqu'une application s'exécute, afin qu'elle ne puisse plus accéder à des espaces mémoire confidentiels. Pour des raisons de performances, cette solution est loin d'être suffisante, mais reste cependant la seule à l'heure actuelle pour les processeurs déja sur le marché. Les versions ultérieures des processeurs on vu leur conception changée de façon à complexifier l'accès direct à l'espace noyau par les *transient instructions*. Les attaques citées précédemment démontrent par conséquent qu'il est extrêmement complexe de sécuriser les architectures matérielles modernes, car de nouvelles classes d'attaques seront découvertes en même temps que ces systèmes évolueront. Il est en outre indispensable de protéger les prochaines architectures matérielles, en prenant en compte l'aspect sécurité dès la conception du système.

Le gain de puissance des ressources matérielles a aussi permis l'introduction de nouveaux paradigmes pour les systèmes d'information, tels que les architectures multicœurs, l'intégration de circuits reprogrammables, mais aussi la virtualisation assistée par le matériel, etc. L'utilisation de ces ressources a naturellement été exploitée par les concepteurs de systèmes pour les mutualiser et les densifier afin d'exécuter de plus en plus d'applications à moindre

coût. Cette densification mène à héberger nombre d'applications hétérogènes complexes (systèmes d'exploitations, hyperviseurs, virtualisation légère, etc.) sur une même infrastructure matérielle.

Toujours dans un souci de réduction des coûts, les fournisseurs de service se sont appuyés sur la virtualisation pour limiter le nombre de machines physiques et proposer des services d'hébergement à un plus grand nombre d'utilisateurs. Dans ce cas, plusieurs systèmes d'exploitations, contrôlés par des utilisateurs différents, fonctionnent dans des environnements virtualisés sur une même machine physique. De nombreuses attaques peuvent être réalisées sur ces systèmes complexes [6] et le partage des ressources physiques avec un autre utilisateur, potentiellement malveillant, peut introduire de nouvelles vulnérabilités. Parmi les attaques possibles, nous pouvons citer la détection et l'identification d'environnement virtualisé [7], l'évasion de machine virtuelle [8, 9, 10] et toutes les stratégies d'élévation des privilèges.

Par conséquent, dans un souci de sécurisation des systèmes, même distants, il est devenu quasiment indispensable de considérer des modèles d'adversaires très puissants et intrusifs, avec des privilèges élevés.

1.2 Vulnérabilités, attaques et intrusions

Dans cette section, nous fournissons des rappels sur la terminologie de la sécurité informatique et présentons ses enjeux dans le cadre de l'exécution d'un algorithme sur un système complexe. Aussi, nous définissons les concepts que nous traitons dans ce manuscrit.

1.2.1 Terminologie de la sécurité informatique

La sûreté de fonctionnement est un concept générique, qui englobe des attributs de fiabilité, sécurité-innocuité et maintenabilité [11, 12], avec ceux de la sécurité-immunité (disponibilité, confidentialité et intégrité). Dans ce manuscrit, nous traitons de la sécurité-immunité. Cette section définit le vocabulaire nécessaire pour la lecture de celui-ci, en commençant par celui de la sûreté de fonctionnement et en l'appliquant ensuite à la sécurité-immunité.

1.2.1.1 Sûreté de fonctionnement

La sûreté de fonctionnement d'un système informatique est définie comme « la propriété qui permet aux utilisateurs du système de placer une confiance justifiée dans le service qu'il leur délivre »[11]. Le service délivré correspond au comportement du système tel que perçu par les utilisateurs. La sûreté de fonctionnement comporte trois axes principaux : les attributs (qui décrivent les points de vue adoptés pour l'évaluer), les entraves (qui sont les évènements pouvant l'affecter) et les moyens permettant de l'atteindre ou de l'améliorer. L'arbre représenté sur la figure 1.1 décrit ces différents axes.

Attributs

Nous pouvons définir les attributs de la sûreté de fonctionnement comme les propriétés complémentaires suivantes :

— **disponibilité** : la capacité d'un système à être prêt à l'utilisation lors d'une demande d'un utilisateur, en supposant que les moyens nécessaires soient assurés.
— **fiabilité** : l'aptitude à assurer le service pendant une durée donnée.

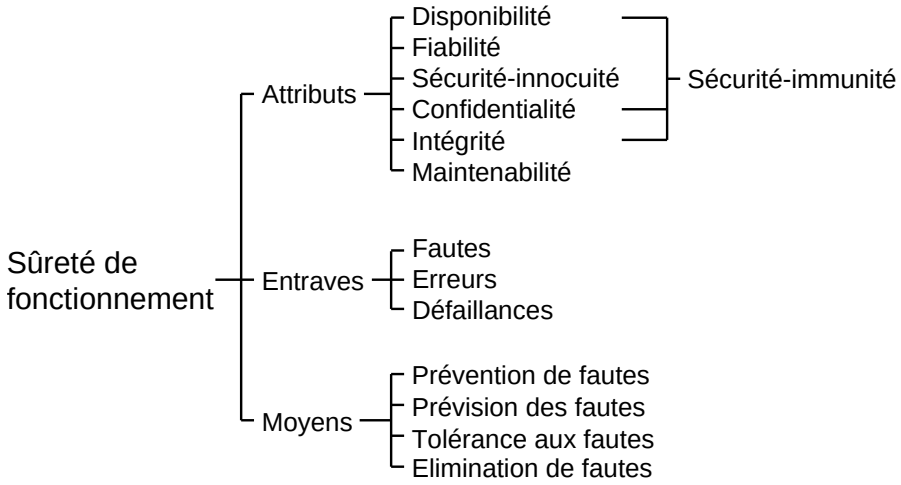

FIGURE 1.1 – Arbre de la sûreté de fonctionnement

— **sécurité-innocuité** : la capacité de ne pas causer de conséquences critiques ou catastrophiques.
— **confidentialité** : la capacité à ne pas divulguer des informations aux utilisateurs non autorisées.
— **intégrité** : la non altération inappropriée de l'information qui compose le système.
— **maintenabilité** : l'aptitude d'un système à revenir dans un état de fonctionnement correct après être réparé ou avoir subi des évolutions.

Entraves

Nous parlons de non-sûreté de fonctionnement lorsqu'au moins une partie de ces attributs ne peut plus être assurée pour cause d'une entrave, c'est-à-dire d'une faute, d'une erreur ou d'une défaillance. Nous pouvons définir ces entraves comme suit :

— **défaillance** : le comportement qui survient lorsque le service délivré dévie de l'accomplissement de la fonction du système.
— **erreur** : la partie de l'état du système qui est susceptible d'entraîner une défaillance.
— **faute** : la cause adjugée ou supposée d'une erreur.

Lorsqu'une faute du système produit une erreur, cette faute est dite *active*. Lorsqu'une erreur produit de nouvelles erreurs, celle-ci *se propage*. Par propagation, si une erreur affecte le service délivré par le système, alors une défaillance survient. Dans le cas où le système est une sous-partie d'un second système plus large, ou dans le cas où un troisième système dépend du premier, une défaillance est une faute pour le second ou le troisième système. La succession de ces entraves permet de compléter la « chaîne fondamentale » suivante :

$$... \rightarrow faute \rightarrow erreur \rightarrow défaillance \rightarrow faute \rightarrow ...$$

Moyens

Afin de mettre en place une sûreté de fonctionnement dans un système, le concepteur dispose de plusieurs moyens. Ces moyens sont des solutions éprouvées pour réduire les conséquences des entraves sur les attributs et casser les enchaînements de la chaîne fondamentale,

donc améliorer la fiabilité du système. Nous pouvons définir ces moyens comme suit :

- **prévention de fautes** : procédé qui consiste à éviter l'introduction de fautes durant le développement du système. Ce procédé s'appuie sur des méthodologies de développement et des bonnes pratiques d'implémentation.
- **prévision des fautes** : procédé qui consiste à estimer la présence de fautes (de manière quantitative et/ou probabiliste) et leurs conséquences sur le système.
- **tolérance aux fautes** : procédé qui consiste à mettre en place des mécanismes pour que le service fourni par le système soit maintenu, même en présence de fautes. Dans ce cas, le fonctionnement dégradé du système est accepté.
- **élimination des fautes** : procédé qui consiste à réduire le nombre de fautes. Ce procédé s'appuie sur des techniques de vérification avancées pour détecter les fautes identifiées et les enlever avant l'implémentation du système. Après implémentation, tenir à jour les défaillances rencontrées permet d'identifier les fautes qui les ont provoquées et les retirer lors des phases de maintenance.

1.2.1.2 Sécurité-immunité

Nos travaux, décrits dans ce manuscrit, s'inscrivent dans le cadre de la sécurité-immunité, qui est définie comme la combinaison des attributs de disponibilité, de confidentialité et d'intégrité. Elle a pour objectif la protection d'un système face à l'introduction d'erreurs et aux fautes intentionnelles, que nous appelons **malveillances**. Les attributs de la sûreté de fonctionnement, définis dans la section précédente, sont génériques. Dans cette section, nous les appliquons au contexte de la sécurité-immunité. Dans la suite de ce manuscrit, le terme *sécurité* fait référence à la sécurité-immunité tandis que le terme *sûreté* fait référence à la sécurité-innocuité (sauf exception explicitement précisée).

Attributs de la sécurité

Dans le cadre de la sécurité, nous considérons seulement trois des attributs de la sûreté de fonctionnement : la disponibilité, la confidentialité et l'intégrité. Les spécifications de nos systèmes sont déterminées en fonction de ces propriétés.

Entraves à la sécurité

Le projet MAFTIA [13] précise le concept de faute intentionnelle en définissant la malveillance, concept sur lequel les travaux décrits dans ce manuscrit se concentrent. Il existe deux classes de malveillances : les logiques malignes et les intrusions. Les logiques malignes sont des entraves intentionnelles qui sont conçues pour provoquer des dégâts ou pour faciliter les futures intrusions. Elles entravent en général la disponibilité et l'intégrité. Les intrusions sont elles-aussi des entraves intentionnelles, mais qui ont pour objectif d'accéder à des données confidentielles. Elles entravent en général l'intégrité et la confidentialité. Les entraves appliquées à la sécurité sont définies comme suit :

- **intrusion** : défaillance malveillante, initiée intentionnellement depuis l'extérieur du système pendant son fonctionnement.
- **attaque** : erreur d'interaction externe au système, initiée intentionnellement, dont le but est de violer un ou plusieurs attributs de sécurité. L'attaque peut aussi être définie comme une tentative d'intrusion, réussie ou non.

— **vulnérabilité** : faute qui peut être accidentelle, intentionnelle malveillante ou intentionnelle non malveillante, placée dans les exigences, la spécification, à la conception ou à la configuration du système. La vulnérabilité peut être exploitée avec une attaque pour créer une intrusion.

Lorsqu'une vulnérabilité est exploitée, cette exploitation est une attaque : une tentative de logique maligne ou d'intrusion, dont l'objectif peut être d'introduire de nouvelles vulnérabilités. Dans le cas où le système est une sous-partie d'un second système plus large, ou dans le cas où un troisième système dépend du premier, une intrusion introduit une vulnérabilité pour le second ou le troisième système. La succession de ces entraves permet de compléter la « chaîne fondamentale » suivante :

$$... \rightarrow \textit{vulnérabilité} \rightarrow \textit{attaque} \rightarrow \textit{intrusion} \rightarrow \textit{vulnérabilité} \rightarrow ...$$

Le projet MAFTIA [13] décrit deux autres types de fautes dues à l'homme, cette fois-ci non intentionnelles : les fautes de conception et les fautes d'interactions. Ces fautes peuvent introduire des vulnérabilités ou permettre des intrusions dans le système :

— **fautes d'interaction** : fautes externes, dues à la mauvaise utilisation du système. Par exemple, un utilisateur d'une paire de clés RSA pour le protocole SSH, peut accidentellement envoyer sa clé privée à un administrateur.

— **fautes de conception** : fautes ajoutées de manière non intentionnelle à la conception du système. Par exemple, un algorithme peut omettre de configurer certains registres critiques pour la sécurité [14], l'implémentation d'un protocole peut être vulnérable [15], ou certains matériels peuvent encore embarquer des fonctions de débug privilégiées (contrôleur JTAG encore présent).

Les fautes d'interaction, non intentionnelles, ajoutent des vulnérabilités au système. Elles permettent potentiellement des intrusions qui ajouteront à leur tour de nouvelles vulnérabilités. Bien que très sensibles pour la sécurité des systèmes, les fautes d'interaction ne font pas partie de l'objet des travaux décrits dans ce manuscrit. Nous considérons que les moyens employés pour lutter contre celles-ci sont la prévention et la prévision, qui font plutôt partie du domaine de la formation.

Néanmoins, nous considérons que les fautes de conception, même non intentionnelles, seront potentiellement exploitées de manière malveillante par des attaques. Nous prenons donc celles-ci en considération dans ce manuscrit et déterminons par conséquent les moyens de luttes contre les malveillances.

Moyens de lutte contre les malveillances

Nous pouvons, encore une fois, spécialiser les moyens adaptés à la sûreté de fonctionnement pour les appliquer à la sécurité. Ici, les moyens sont des solutions éprouvées pour réduire le nombre d'intrusions réussies sur le système. Nous pouvons les définir comme suit :

— **prévention de vulnérabilité** : procédé qui consiste à éviter l'introduction de vulnérabilités durant le développement du système. Il s'appuie sur des méthodologies de développement et des bonnes pratiques d'implémentation.

— **prévision des attaques** : procédé qui consiste à estimer la présence de vulnérabilités (de manière quantitative et/ou probabiliste) et la présence d'attaques dont le systèmes sera la cible. Ce procédé s'appuie sur la définition d'un profil d'adversaire, ce qui inclut son domaine, ses motivations, ses niveaux de ressources et ses degrés de ténacité.

— **tolérance aux intrusions** : procédé qui consiste à mettre en place des mécanismes pour détecter les intrusions et se réparer, se reconfigurer, tout en continuant de garantir une disponibilité de service et/ou l'intégrité des données pendant une attaque [16]. Dans nos travaux, nous détectons les intrusions à l'aide de tests d'intégrité de l'environnement logiciel d'une machine et de l'authentification via un secret partagé. Nous appelons la phase de réparation le *recouvrement*. Le recouvrement doit permettre de reconstituer un état sûr du système (isolation des données sensibles, redémarrage, réinstallation, etc.).
— **élimination des vulnérabilités** : procédé qui consiste à réduire le nombre de vulnérabilités. Ce procédé s'appuie sur des techniques de vérification avancées pour détecter les vulnérabilités identifiées et les enlever avant l'implémentation du système.

Toujours dans le cadre du projet MAFTIA, les auteurs avancent que seule l'élimination de vulnérabilités est réellement pertinente [13]. En effet, il est impossible d'interdire à un utilisateur d'attaquer un système dès lors que ce système communique avec l'extérieur. Si une vulnérabilité existe, alors une intrusion peut potentiellement réussir.

Le développement d'un système est souvent complexe, et l'élimination des vulnérabilités est une tâche difficile. Les concepteurs peuvent se baser sur les méthodes formelles pour vérifier l'absence de vulnérabilités identifiées. En amont de cette vérification, une phase de spécification formelle (modélisation du comportement attendu vis-à-vis de la vulnérabilité) et de modélisation du système est nécessaire.

1.2.2 Problèmes liés à la sécurisation des systèmes informatiques modernes

Une conséquence de la densification des systèmes informatiques, aujourd'hui, est que les systèmes matériels hébergent désormais nombre d'applications hétérogènes et complexes. L'augmentation des performances mène les utilisateurs à héberger des applications sur des systèmes distants et à y accéder via le réseau public (Internet). Parmi ces applications, toutes ne sont pas contrôlées par un seul et même utilisateur et certaines sont potentiellement malveillantes ou corrompues. Dans cette section, nous listons les différents problèmes auxquels doit faire face la sécurisation des systèmes modernes.

Partage des ressources

La première difficulté est bien évidemment le partage des ressources. Les systèmes modernes sont multi-utilisateurs, et considérer tous les utilisateurs de confiance semble impossible. En effet, certains utilisateurs peuvent réaliser des fautes d'interaction, intentionnelles ou non, et ajouter des vulnérabilités au système complet. Certains autres utilisateurs peuvent être malveillants et réaliser des attaques sur le système. Dans le cas d'un hébergement, par exemple chez un fournisseur commercial d'architecture de type cloud, l'administrateur possède des privilèges élevés sur le système et peut potentiellement être lui-même malveillant.

Diversité des systèmes

Également, dans le cas d'un hébergement chez un fournisseur commercial, plusieurs systèmes virtualisés peuvent cohabiter sur la même machine physique. Dans ce cas, un système d'exploitation hôte procède à une ségrégation des systèmes virtualisés et est chargé d'autoriser ou d'interdire les accès. Les administrateurs et utilisateurs de chaque système virtua-

lisé peuvent être malveillants et réaliser des attaques sur le systèmes physique. Ces attaques peuvent être de type logiciel [8, 9, 10] ou matériel [2, 3, 4] et peuvent cibler les périphériques matériels ou les zones mémoires des autres systèmes virtualisés. Même en choisissant une architecture logicielle dépourvue de vulnérabilité (hyperviseur, système d'exploitation, applications), les systèmes des autres utilisateurs et le système d'exploitation hôte ne sont pas maîtrisés.

Paradigme client/serveur

Le paradigme client/serveur permet de rendre un système accessible par un accès distant, souvent via un réseau qui n'est pas maîtrisé par l'utilisateur (par exemple : Internet). L'avantage est qu'il rend le système disponible à tout instant. Les contrôles d'accès sont généralement réalisés par une authentification des utilisateurs (mot de passe, signature numérique, etc.). L'inconvénient est qu'il peut être accédé par tout utilisateur connecté au réseau. Et malheureusement, la présence de vulnérabilités dans le système peut rendre les attaques possibles à tout instant et ce, par un grand nombre d'utilisateurs malveillants.

Maintien des performances

Une autre problématique lors de la sécurisation d'un système est le maintien des performances et de la durée de vie. Lorsqu'une vulnérabilité est découverte sur un système déjà accessible au grand public, nombreux sont les correctifs de sécurité applicables qui dégradent les performances et/ou la durée de vie du système.

Par exemple, comme décrit précédemment, la contre-mesure adoptée par la communauté Linux contre l'attaque Meltdown [3, 5] propose une isolation stricte de l'espace mémoire alloué au noyau par rapport à celui alloué à l'espace utilisateur. Cette modification a un impact lourd sur les performances globales de la machine, en particulier lors des changements de contexte.

Dans le cas de l'attaque Rowhammer [4], la vulnérabilité est une fuite électromagnétique dans les mémoires vives DRAM causée par une réduction de surface. La contremesure proposée par les auteurs est d'augmenter la fréquence de rafraîchissement de la mémoire. Malheureusement, celle-ci provoque une augmentation de la consommation moyenne et une réduction de la durée de vie de telles mémoires vives ; les performances sont également impactées [4].

Coûts de maintenance

Impacter une contremesure vis-à-vis d'une vulnérabilité a un coût de migration. Dans le cas d'une contremesure logicielle, le coût de migration est relativement faible, même si celle-ci implique une mise à jour de l'ensemble des systèmes vulnérables (qui peut être conséquente, selon le nombre de systèmes impactés). Dans le cas d'une contremesure matérielle, le coût de migration peut être plus élevé car celle-ci implique potentiellement un changement de matériel et une migration du logiciel.

Dans le cas de l'attaque Spectre [2], par exemple, la vulnérabilité réside dans le comportement des microprocesseur qui effectuent des prédictions de branchement. Dans ce cas, il n'est possible de proposer une correction qu'en désactivant les accélérations matérielles, au détriment des performances. L'autre alternative est de migrer le système vers un microprocesseur non-vulnérable, par exemple vers une version plus récente, après un correctif matériel proposé par le constructeur.

1.3 Sécurisation d'un algorithme sur un système complexe

Dans ce manuscrit, nous proposons un mécanisme de sécurisation d'un algorithme sur un système complexe, dont l'architecture matérielle repose sur un microprocesseur moderne. Notre objectif est d'être capable de **détecter une intrusion en vérifiant l'intégrité d'un algorithme et de son environnement**. Nous tirons profit des mécanismes de tolérance aux intrusions ainsi que de prévention d'intrusion, telle que l'élimination des vulnérabilités. Dans cette section, nous introduisons ces moyens de sécurisation ainsi que les travaux à l'état de l'art sur lesquels nous pouvons nous appuyer.

1.3.1 Tolérance aux intrusions

Le principe de tolérance aux intrusions réside dans le fait qu'une intrusion est le résultat d'une faute intentionnelle, et que la présence d'une faute peut être tolérée. Pour cela, lorsqu'une intrusion est réussie, le système tolérant doit être capable de détecter cette intrusion, et d'assurer un recouvrement afin de rétablir un état sûr. L'avantage d'une telle approche est que la vérification du système peut être partielle et que des vulnérabilités peuvent persister tant que le recouvrement empêche les intrusions d'en ajouter de nouvelles.

Dans cette catégorie, nous pouvons citer Wahab et al. [17], qui tirent profit de l'interface de débug matérielle des microprocesseurs ARM, nommée *CoreSight*, en la couplant à un circuit de logique programmable, de type *Field-Programmable Gate Array* (FPGA). Les auteurs proposent une extension matérielle dédiée au *Dynamic Information Flow Tracking* haute-performances. Également, nous pouvons citer Lee et al. [18], qui utilisent aussi *CoreSight* et proposent une extension matérielle permettant la détection d'attaques *Return-Oriented Programming*. La détection se base sur l'analyse des informations de branchement et ne nécessite aucune modification du cœur du microprocesseur.

Dans la catégorie recouvrement, nous pouvons citer SCIT/HES [19], une solution de tolérance aux intrusions auto-correctrice. La stratégie employée est de proposer une diversification logicielle et/ou matérielle du même système dans une ferme de serveurs et de périodiquement rétablir un état sûr pour ceux-ci. Une extension matérielle spécifique, inaccessible par l'adversaire car isolée, permet de prédire la période de rétablissement même en présence d'attaques.

La tolérance aux intrusions propose donc des mécanismes qui permettent de supprimer, par reconfiguration du système, les vulnérabilités exploitées pour une intrusion détectée. L'utilisation de méthodes formelles permet, une fois la vulnérabilité identifiée, de prouver son absence par une vérification du système. Dans ce cas, nous parlons d'élimination des vulnérabilités.

1.3.2 Élimination des vulnérabilités

Dans le cas de l'élimination des vulnérabilités, les concepteurs considèrent une partie finie d'un système et de ses interactions avec un adversaire, puis garantissent que l'union des deux ne possède aucune vulnérabilité pouvant mener à un certain nombre d'intrusions qui ont été identifiées.

Dans cette catégorie, nous pouvons citer les solutions de vérification formelle de logiciel ainsi que la certification de transformations de leur description. Ahman et al. proposent F$\star$, un langage de programmation fonctionnel, accompagné d'un assistant de preuve, permettant la vérification formelle de logiciel [20]. Protzenko et al. proposent une transformation de Low$\star$,

sous-ensemble de F⋆, en langage C, assurée par le compilateur certifié KreMLin [21]. Également, Leroy propose CompCert, un compilateur C certifié pour produire des exécutables exempts de tout défaut de compilation [22]. Il peut être utilisé pour produire un exécutable préservant les propriétés de sécurité vérifiées en Low⋆ [23]. Les architectures cibles supportées par CompCert sont PowerPC, ARM, RISC-V et x86 [24].

Nous pouvons également citer les solutions de vérification formelle de matériel. Clarke et al. ont défini les solutions de vérification formelle de circuits numériques [25, 26]. La spécification est décrite dans un langage de logique temporelle [25], comme introduit par Pnueli [27]. Berezin et al. propose également de raisonner sur des compositions pour la vérification de tels circuits [28]. Ceci permet d'appliquer efficacement les méthodes de *model-checking* [29] à des modèles décrits en langage de description matérielle (HDL).

1.4 Modèle de menaces

Afin d'illustrer nos travaux, nous définissons dans cette section notre modèle de menaces. Celui-ci est en cohérence avec les paradigmes des systèmes d'information modernes.

Nous considérons donc un **modèle de menaces fort** pour un algorithme dont l'intégrité est à vérifier. L'adversaire a, au préalable, réalisé des attaques sur le système sur lequel notre algorithme s'exécute : il a procédé à une élévation de ses privilèges au rang le plus élevé. Il lui est donc possible de lire et écrire dans toutes les mémoires auxquelles le microprocesseur a accès, de reconfigurer les périphériques, d'empoisonner les mémoires cache, d'effectuer des attaques par canaux auxiliaires, etc. Modéliser les capacités de l'adversaire revient donc à modéliser les fonctionnalités du microprocesseur. Également, l'adversaire possède un clone du système : il peut donc reproduire son environnement sur un système où il a physiquement accès. Cela rend par exemple possible la connexion d'un analyseur logique et l'observation des signaux sur le matériel lors d'une exécution de logiciel malveillant.

Certaines garanties sont nécessaires afin de pouvoir dériver des propriétés de sécurité. Nous considérons que notre adversaire n'a **pas d'accès physique au système**. Il ne peut donc prendre son contrôle qu'au travers d'un accès distant, via le réseau public. Cela exclut donc de ses capacités les attaques intrusives, où une modification du matériel ou une observation de son empreinte électromagnétique est requise. Cela nous autorise donc à nous appuyer sur du matériel pour garantir l'intégrité de notre algorithme et de son environnement.

Considérer un tel modèle de menaces peut se résumer à considérer un paradigme de sécurisation, nommé *attestation à distance*.

1.5 Attestation à distance

L'attestation à distance est une méthode qui permet de vérifier dynamiquement l'intégrité d'un algorithme s'exécutant sur une machine distante. Cette méthode s'appuie sur deux composants principaux : un **protocole cryptographique** d'attestation à distance ; ainsi qu'une architecture de vérification de l'intégrité d'environnement d'exécution. Cette architecture est en général composée d'un ensemble minimaliste d'éléments logiciels et matériels qui sont considérés de confiance. Ces éléments de confiance sont aussi appelés **racine de confiance statique**.

1.5.1 Protocole cryptographique

L'attestation à distance permet d'établir une racine de confiance dynamique, c'est-à-dire à l'exécution, même en présence d'un adversaire capable de corrompre l'intégralité d'une machine distante, nommée *prouveur*, à l'exception de sa racine de confiance statique. À l'issue de l'exécution du protocole, un utilisateur, nommé *vérifieur*, pourra décider de la confiance à accorder à l'algorithme distant, avant de lui transmettre par exemple des données sensibles. Notons bien que l'attestation à distance telle que définie dans ce manuscrit est par construction vulnérable aux attaques de type *Time Of Check Time Of Use* (TOCTOU), d'où la notion temporelle des propriétés vérifiées. L'objectif est donc de vérifier si cet algorithme possède, à un instant donné, les propriétés de sécurité acceptables [30]. Ces propriétés de sécurité sont généralement vérifiées à l'aide d'un protocole question-réponse comme illustré sur la figure 1.2.

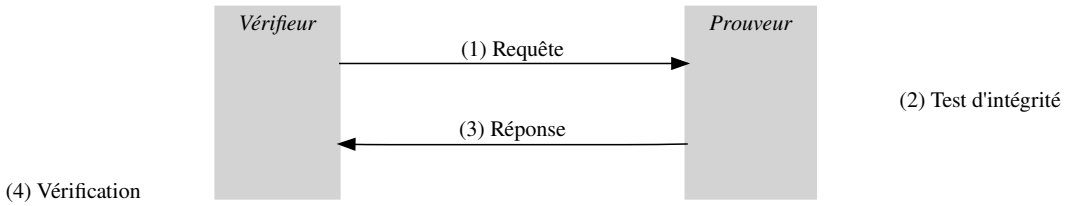

FIGURE 1.2 – Protocole d'attestation à distance

Le protocole d'attestation à distance se décompose en quatre étapes :

1. Dans un premier temps, le vérifieur envoie une requête d'attestation au prouveur. Cette requête est généralement accompagnée d'un challenge, par exemple un *nonce* pour se protéger contre les attaques par rejeu.
2. Le prouveur réalise un test d'intégrité : il calcule par exemple un tag d'intégrité sur sa mémoire et son environnement, le cas échéant sur le challenge.
3. Le prouveur retourne au vérifieur le résultat du calcul.
4. À partir du résultat, le vérifieur décide de l'état de validité du prouveur.

La requête d'attestation permet de vérifier un certain nombre de mesures de l'état de la machine distante, par exemple de sa mémoire et de ses registres de configuration privilégiés. Peuvent être inclus dans le test d'intégrité : un calcul sur l'algorithme lui-même, les tables de traduction de l'unité de gestion mémoire (MMU, pour *Memory Management Unit*), la configuration de l'interface de débug, etc. Le choix des zones mémoires inclues dans le calcul dépend des propriétés de sécurité à vérifier lors de l'exécution du protocole. Une valeur représentative de l'état courant est transférée au vérifieur à l'aide de la réponse.

Le protocole d'attestation à distance s'inscrit dans le cadre de la tolérance aux intrusions : c'est un mécanisme de détection d'intrusions. En effet, l'analyse de la réponse par le vérifieur permet une détection si le résultat du calcul n'est pas égal à une valeur attendue.

Bien évidemment, ce mécanisme ne fonctionne que dans le cas où l'intégrité du résultat transmis par le prouveur est garantie. Se pose alors la question de la capacité d'un adversaire de forger une réponse ou de tricher sur le calcul du tag d'intégrité. Le vérifieur doit donc

posséder un moyen d'assurer l'intégrité de la réponse. Nous appelons ce moyen la racine de confiance statique : un ensemble minimaliste d'éléments logiciels et matériels du prouveur considéré de confiance.

1.5.2 Racine de confiance statique

L'attestation à distance nécessite donc l'exécution d'un challenge dans un environnement potentiellement corrompu. La garantie de l'exécution correcte de ce challenge est essentielle (nécessaire) à la sécurité du protocole. Deux stratégies principales se distinguent pour l'authentification de la réponse :

1. soit le challenge est conçu de telle sorte qu'une modification de l'environnement d'exécution par l'adversaire modifie nécessairement les attributs du calcul de la réponse (principalement capturés par le temps) ;
2. soit on s'efforce de garder inconnue de l'adversaire une transformation dépendante de l'état d'intégrité de la machine prouveur (utilisation de primitives cryptographiques telle qu'un tag d'intégrité).

Nous pouvons citer Morgan et al. [31], Checkmate [32], et Pioneer [33] comme travaux représentatifs de la première catégorie. Ces premiers travaux d'attestation à distance utilisent seulement du logiciel et basent l'authenticité de la réponse sur un temps d'exécution attendu. Ce modèle de sécurité est vulnérable à l'augmentation de la puissance de calcul de l'adversaire. Il est par exemple envisageable d'augmenter la fréquence d'exécution de la machine (*overclock*) pour réduire la durée du calcul [33]. Également, Kovah et al. évaluent un *Trusted Platform Module* (TPM) matériel et proposent un modèle d'attestation basé sur la mesure des cycles d'horloge durant l'exécution du code [34]. Ils montrent que des TPM d'un même modèle et d'un même fabriquant n'utilisent pas le même nombre de cycles d'horloge pour exécuter le même code [34]. Une calibration du matériel est donc nécessaire pour paramétrer l'exécution du protocole d'attestation.

Les travaux SMART [35] et VRASED [36] s'inscrivent quant à eux clairement dans la seconde catégorie. Cette famille de travaux utilise la cryptographie pour mesurer l'authenticité de la réponse. Cela implique de protéger la confidentialité des secrets cryptographiques impliqués dans la transformation : le calcul de tag d'intégrité. Du support matériel est chargé de réaliser les contrôles d'accès et une preuve de sécurité démontre l'incapacité de l'adversaire à pouvoir forger une réponse. Une **élimination des vulnérabilités est donc réalisée sur un sous-ensemble du système, de taille modeste ; ceci permet d'établir la racine de confiance statique**. Cette solution permet de démontrer la sécurité globale d'une méthode d'attestation proposée à défaut de la sécurité d'un système complet. Notons que l'échange du secret se fait généralement à la mise en route de la machine prouveur, lors d'un accès physique par le vérifieur, et n'est pas intégré dans le protocole. Selon le choix de la transformation, l'impossibilité de calcul d'un tag d'intégrité sans connaître le secret est admise [37].

Dans ce manuscrit, nous choisissons de traiter la seconde stratégie. Celle-ci implique donc de définir un test d'intégrité authentifié, et que l'architecture choisie garantisse la confidentialité des secrets. Notons que les constructeurs de matériel ont déjà proposé des solutions d'environnement d'exécution de confiance, ou *Trusted Execution Environments* (TEE), qui peuvent aider à établir ces racines de confiance statique (ARM *Trustzone*, *Intel Trusted eXecution Technology*). Certaines de ces solutions propriétaires proposent déjà des solutions d'attestation d'enclaves utilisateur (Intel *Software Guard eXtensions*), voire de machines virtuelles

(Intel *Trust Domain eXtensions*). Ces solutions propriétaires ne sont malheureusement pas distribuées avec leurs modèles et leurs spécifications formelles. Il est donc difficile d'argumenter sérieusement en faveur de l'absence de vulnérabilités et, par conséquent, impossible aujourd'hui d'apporter des garanties formelles concernant ces propositions.

1.5.2.1 Test d'intégrité authentifié

Le test d'intégrité choisi pour l'attestation à distance dépend des propriétés de sécurités que le vérifieur souhaite obtenir lors de l'exécution du protocole. L'algorithme utilisé pour le calcul du tag d'intégrité et les zones mémoires inclues dans le calcul sont donc à définir en amont de la mise en production.

Zinzindohoué et al. proposent HACL⋆, une bibliothèque cryptographique écrite en C et formellement vérifiée [23]. Cette bibliothèque inclut différentes primitives cryptographiques modernes, telles que des algorithmes de chiffrement, de signature, des fonctions de hachage et des calculs de tags d'intégrité (Poly1305, HMAC). Le code source pour chaque primitive cryptographique est vérifié formellement sur un modèle décrit en langage F⋆, compilé ensuite vers du langage C. La transformation du modèle en C depuis F⋆ préserve les propriétés de sécurité vérifiées [23]. Ces vérifications formelles incluent la conformité, la sûreté mémoire, l'indépendance au secret et des contre-mesures vis-à-vis des attaques par canaux auxiliaires. Les primitives cryptographiques proposées par HACL⋆ sont utilisées par VRASED pour le calcul du tag d'intégrité [36]. Un prérequis pour que les propriétés de sécurité soient vérifiées sur l'implémentation finale **est que l'adversaire ne connaisse pas le secret et que l'exécution soit atomique et ininterrompue**.

1.5.2.2 Architecture co-conçue

Les travaux SMART [35] et VRASED [36] ont ouvert la voie à un domaine de recherche sur la sécurité des protocoles d'attestation à distance. Ils ont notamment proposé les premières versions d'architectures co-conçues formellement vérifiées. En outre, ils se sont limités au cas d'étude de machines relativement simples, qu'ils s'autorisent à lourdement modifier : microcontrôleurs pour circuits programmables, aussi appelés *softcores*.

Dans ces travaux, **la confidentialité du secret et la sécurité de l'exécution de la fonction d'attestation sont contrôlées par un moniteur matériel qui observe le comportement de la machine prouveur**. Ces contrôles n'autorisent la machine à s'exécuter que dans le cas où celle-ci se trouve dans un état sûr. Si un état non sûr est atteint, la machine est reconfigurée par le moniteur vers un état sûr (redémarrage). La preuve de sécurité est obtenue par modélisation de la machine prouveur, du moniteur et de l'adversaire comme un système à états transitions sur lequel on vérifie l'absence de transition interdite.

1.6 Vérification formelle pour la sécurité

La vérification formelle apporte une réponse aux exigences de sécurité de l'attestation à distance. Elle permet l'établissement d'une preuve, basée sur des axiomes ou des propriétés démontrées, de la sécurité du système et de son implémentation.

Nous appelons *spécifications* l'ensemble des propriétés que doit satisfaire le système et qui garantissent l'absence de vulnérabilité. La vérification formelle s'effectue généralement en trois

étapes. D'abord, le système (logiciel ou matériel) est modélisé, par exemple sous forme d'un automate. Ensuite, les spécifications que doit satisfaire le système sont formellement énoncées, entre autres la confidentialité du secret dans le cas de l'attestation à distance. Finalement, du ***model-checking*** ou une **preuve** démontre que le système vérifie ses spécifications.

Le *model-checking* est une méthode de vérification exhaustive et automatique. Cependant, cette méthode souffre d'explosion combinatoire lorsque le nombre d'états nécessaires pour modéliser le système dépasse la quantité de mémoire de la machine utilisée pour la vérification. Une solution pour faire face à l'explosion combinatoire est l'utilisation de modèles abstraits lors de la vérification de propriétés. Nous appelons *modèle concret* la modélisation du système proche de son implémentation finale et nous appelons *abstraction* le procédé de transformation du modèle concret en un modèle abstrait. L'établissement d'une relation de simulation entre le modèle concret et son modèle abstrait peut garantir l'héritage des preuves réalisées sur le modèle abstrait vers le modèle concret.

D'autres outils de vérification formelle, comme les assistants de preuves, sont capables d'assister un ingénieur à qui revient la tâche de construire une preuve. Ils permettent l'écriture et la vérification de preuves mathématiques. En logique du premier ordre ou en logique temporelle, à partir d'un ensemble de propriétés et des règles de sémantiques, un assistant de preuves permet de démontrer l'implication de nouvelles propriétés.

Dans le cas de compositions de systèmes fonctionnant en parallèle, comme les systèmes matériels implémentés dans un FPGA, une combinaison de *model-checking* et de preuve permet de démontrer des spécifications complexes :

— Le système est décomposé en modèles abstraits.
— Les spécifications sont décomposées en propriétés plus petites, adaptées aux modèles abstraits. Nous appelons *propriétés locales* ces propriétés plus petites.
— Les propriétés locales sont vérifiées sur les modèles abstraits par *model-checking*.
— Une relation de simulation, établie entre les modèles abstraits et le modèle concret, garantit un héritage que les propriétés locales sont vérifiées.
— Les propriétés locales forment alors des hypothèses pour prouver les spécifications.
— Finalement, un assistant de preuves permet de construire une démonstration mathématique que les spécifications sont vérifiées.

Une stratégie de preuve sur un système co-conçu et des rappels techniques sur les automates et propriétés temporelles sont décrits dans le chapitre 3.

1.6.1 Model-checking compositionnel

Le procédé d'abstraction d'un système est un problème non trivial. Tout d'abord, le modèle abstrait doit être suffisamment réduit pour permettre à la machine qui réalise la vérification de ne pas souffrir d'explosion combinatoire. Également, abstraire un système de manière excessive, c'est-à-dire sur-approximer son comportement, conduit à l'apparition de faux négatifs [38]. C'est le cas où les outils de vérification fournissent un contre-exemple qui viole la propriété sur le modèle abstrait mais où le modèle concret s'en protège via un mécanisme qui a été retiré lors de l'abstraction. On parle alors de faux (*spurious*) contre-exemple. Le ***model-checking*** **compositionnel** est un ensemble de stratégies d'abstraction et de recommandations pour garantir l'absence de faux contre-exemples lors de la vérification.

Nous appelons *variable locale* un élément individuel du modèle dont la valeur peut varier au cours du temps, et *prédicat* une référence à ce même élément individuel dans une propriété

à vérifier. Nous appelons également *raffinement* le procédé de transformation d'un modèle abstrait en un modèle concret.

Kurshan, Clarke et Veieth ont formalisé les stratégies d'abstraction les plus communes pour garantir l'absence de faux négatifs [26]. Parmi ces stratégies, nous pouvons citer deux grandes familles : les stratégies itératives de raffinement guidées par les contre-exemples (*counterexample-guided abstraction refinement*) et les stratégies de réduction par localisation (*localization reduction*). La première famille propose d'abstraire des variables locales et prédicats par un ensemble fini de valeurs constantes et d'itérer sur l'abstraction du modèle jusqu'à ce qu'un contre-exemple donné soit alors prouvé ou réfuté. En effet, l'abstraction de prédicats peut conduire à l'apparition de faux contre-exemples qui sont alors vérifiés a-posteriori et de manière itérative : le modèle abstrait est raffinée au fur et à mesure des itérations. La seconde famille propose de réduire le modèle en masquant seulement les variables locales qui n'ont pas d'impact sur les prédicats référencés dans la propriété à vérifier. Cette solution est moins performante en termes de réduction mais garantit l'absence de faux contre-exemples dès la première vérification et ne nécessite pas d'itération. C'est la solution que nous avons choisie dans cette thèse car nous concevons entièrement notre système et pouvons nous assurer d'une interdépendance minimale entre les variables du systèmes : l'architecture du système est donc choisie afin de faciliter la **réduction par localisation**.

Berezin, Campos et Clarke détaillent l'application des précédentes stratégies d'abstraction à des systèmes construits à partir de compositions fonctionnant en parallèle [28]. Cette application s'adapte parfaitement aux systèmes, synchrones et asynchrones, tels qu'implémentés dans les FPGA où l'accent est mis sur le parallélisme. Parmi ces détails, l'application de la stratégie qui nous intéresse est celle de la réduction par localisation. Les auteurs soulignent la nécessité de décomposer ou partitionner la relation de transition d'un système (qui décrit ses changements d'états) sous une forme conjonctive ou disjonctive pour chacune des variables locales du système. Ces procédés sont appelés la **composition parallèle paresseuse** (*lazy parallel composition*) et la création de **relation de transition partitionnée** (*partitioned transition relation*). Ensuite, l'élimination des variables locales qui n'impactent pas sur le système lors de la vérification d'une propriété est réalisée et l'algorithme de réduction est fourni [28]. Suivre les directives décrites par les auteurs lors de la construction d'un modèle abstrait garantit l'absence de faux négatifs lors de la vérification d'une propriété locale. Nous qualifions alors le modèle abstrait de correct (*sound*) vis-à-vis de cette vérification sur le modèle concret.

1.6.2 Relations de simulation

Nous admettons ici être en possession de deux modèles distincts, pour un même système, représentés sous la forme d'automates. Nous qualifions le premier modèle de modèle concret, qui représente le comportement du système proche de son implémentation finale. Nous qualifions le second modèle de modèle abstrait, qui est le résultat d'une transformation réalisée dans le but de vérifier une propriété locale. Nous souhaitons nous assurer que le modèle abstrait est correct vis-à-vis de la vérification de cette propriété sur le modèle concret. Pour cela, il est possible d'établir une **relation de simulation** entre les deux modèles telle que la transformation préserve les propriétés vérifiées.

Milner introduit le concept de simulation entre des systèmes comme une association de ces systèmes telle qu'ils se comportent de la même manière, au sens où un système simule l'autre [39]. Dans sa définition la plus générale, étant donné un système de transition étiqueté ($\mathcal{S}$, Λ,

$\rightarrow$), où :

— $\mathcal{S}$ représente l'ensemble des états du système,
— Λ représente l'ensemble des étiquettes,
— $\rightarrow \subseteq \mathcal{S} \times \Lambda \times \mathcal{S}$ représente la relation de transition du système,

une relation de simulation $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{S}$ est une relation telle que, pour chaque paire d'états $p, q \in \mathcal{S}$, si le couple $(p, q) \in \mathcal{R}$, alors pour toute étiquette $\lambda \in \Lambda$ et pour tout état $p' \in \mathcal{S}$:

$$p \xrightarrow{\lambda} p'$$

implique qu'il existe un état $q' \in \mathcal{S}$ tel que :

$$q \xrightarrow{\lambda} q' \text{ et } (p', q') \in \mathcal{R}$$

Lorsque nous souhaitons établir une relation de simulation entre deux systèmes, nous devons montrer qu'il existe une relation de simulation $\mathcal{R}$ telle que, pour chaque paire d'états $p, q \in \mathcal{S}$, le couple $(p, q) \in \mathcal{R}$. Dans ce cas, les états p et q sont dits similaires et nous notons $p \preceq q$, où $\preceq$ est appelée la relation de similarité.

Toutes les relations de simulations ne permettent pas de garantir la préservation des propriétés vérifiées, seulement certaines relations de simulations particulières. Lors de la création du modèle abstrait à partir du modèle concret, une transformation basée sur une simulation particulière doit donc être astucieusement choisie. Wang, Hatchel et Somenzi classifient les méthodes d'abstraction en deux catégories : les transformations préservant les propriétés (*property-preserving*) et les transformations conservatrices (*conservative*) [38]. Les transformations conservatrices sont les transformations qui apportent des simplifications drastiques lors de l'abstraction mais dont l'établissement d'une relation de simulation n'est pas suffisante pour garantir la préservation des propriétés. Les transformations préservant les propriétés sont moins efficaces en termes de réduction mais garantissent que toutes les propriétés temporelles vérifiées sur l'abstraction le sont également sur le modèle concret : ce sont les transformations qui nous intéressent dans cette thèse. Parmi les transformations préservant les propriétés, nous pouvons citer les simplifications basées sur une simulation par inclusion de langage [40]. Ce type de transformation préserve la logique temporelle linéaire (LTL) [38].

Une famille de relations de simulation qui nous intéresse particulièrement est celle des **relations de simulation paramétrée par une connexion de Galois**. Bensalem, Bouajjani, Loiseaux et Sifakis démontrent la préservation des propriétés de logique temporelle lorsque deux systèmes sont liés par une relation de simulation paramétrée par une connexion de Galois [41]. Dans ce cas, le modèle abstrait est une simplification du modèle concret basée sur une simulation par inclusion de langage. Si le modèle abstrait a été construit à partir d'une réduction par localisation, où une relation de transition partitionnée a été créée, établir une relation de simulation paramétrée par une connexion de Galois est assez direct. En effet, chaque état du modèle abstrait est le résultat du choix d'une valeur donnée pour chacune des variables locales qui ont été conservées ; chaque état du modèle concret est le résultat du choix d'une valeur donnée pour chacune des variables locales de tout le système. Chaque état du modèle abstrait est donc similaire à tous les états du modèle concret où les variables locales conservées ont la même valeur et où la relation de transition partitionnée mène à un nouvel état similaire. Le langage du modèle abstrait est donc inclus dans le langage du modèle concret et les deux modèles partagent les preuves qu'une propriété LTL est vérifiée [41].

1.6.3 Modélisation de systèmes matériels

De nombreux travaux ont été réalisés pour simplifier, voire automatiser, la construction de modèles abstraits pour la vérification formelle de systèmes matériels décrits en langage *Verilog*. Parmi les solutions proposées, nous pouvons citer, par exemple, le guidage intelligent des choix des prédicats dans les stratégies itératives de raffinement guidées par les contre-exemples. Également, nous pouvons citer les travaux améliorant la réduction par localisation, généralement par le découpage de vecteurs pour l'extraction de prédicats.

Les travaux de Jain, Kroening, Sharygina et Clarke se situent dans la première catégorie : les auteurs proposent d'utiliser des algorithmes de calcul de plus petite précondition (*weakest precondition*) pour déterminer les itérations guidées par les contre-exemples [42]. Les conditions de transitions des variables locales, décrites dans le *Verilog*, sont développées et un calcul de plus petite précondition détermine les valeurs constantes à appliquer aux variables locales et prédicats à abstraire. Cette solution permet de réduire le nombre d'itérations afin de prouver ou réfuter un contre-exemple, même si elle ne s'applique que dans le cas des faux contre-exemples dus à une valeur de prédicat impossible et pas au cas d'une transition impossible [42].

Toujours dans la catégorie du raffinement guidé par les contre-exemples, les travaux de Ho, Chauhan, Roy, Mishchenko et Brayton proposent d'utiliser l'abstraction de fonctions non-interprétées pour déterminer les itérations [43]. Les auteurs extraient, depuis le langage *Verilog*, les fonctions arithmétiques appliquées sur les variables locales et utilisent ces mêmes fonctions pour contraindre les prédicats à abstraire lors des itérations. Les mêmes fonctions arithmétiques sont laissées non-interprétées pour augmenter l'abstraction. Lors des itérations, selon la cause de l'apparition du faux contre-exemple, de nouvelles contraintes sont ajoutées aux prédicats et les fonctions non-interprétées sont raffinées [43].

Dans la seconde catégorie, pour améliorer la réduction par localisation, Andraus et Sakallah proposent une extraction des prédicats par un découpage intelligent des vecteurs [44]. Les accès aux champs des vecteurs sont identifiés depuis le langage *Verilog* et de nouveaux prédicats sont définis en conséquence. Des fonctions d'extraction et de concaténation sont également ajoutées au modèle abstrait et laissées non-interprétées. Cette solution permet un découpage des variables locales et la mise en parallèle des traitements : accompagnée d'un choix approprié des prédicats dans les propriétés à vérifier, les modèles abstraits construits lors d'une réduction par localisation sont donc d'autant plus réduits.

Avec une vision plus globale du système, Irfan, Cimatti, Griggio, Roveri et Sebastiani proposent une solution afin de permettre la vérification formelle de tout un système matériel, sans abstraire son comportement. Les auteurs fournissent *Verilog2SMV* [45], un outil de conversion de langage *Verilog* vers le langage SMV, utilisé pour la description d'automates. *Verilog2SMV* supporte un typage haut niveau pour les variables locales, similaire au langage *Verilog* pour les *wires* et *registers* [45, 46]. En particulier, un des objectifs de *Verilog2SMV* est la gestion efficace des systèmes comportant des mémoires [45]. *Verilog2SMV* est construit sous forme d'une extension pour *yosys* [47], la suite libre de synthèse *Verilog*. La conversion en langage SMV est donc réalisée après une mise à plat et une synthèse du système.

1.7 VRASED : stratégie de vérification formelle

Plusieurs travaux se sont intéressés à la vérification formelle de l'attestation à distance. De Oliveira Nunes, Eldefrawy, Rattanavipanon, Steiner et Tsudik proposent VRASED (*Verifiable Remote Attestation for Simple Embedded Devices*) [36], une méthode d'attestation à distance co-conçue et prouvée. Dans ce manuscrit, nous proposons de réutiliser la stratégie de vérification formelle proposée par les auteurs et d'étendre leurs travaux à l'attestation à distance sur des microprocesseurs modernes. Dans cette section, nous détaillons cette stratégie de vérification.

1.7.1 Test d'intégrité authentifié

L'approche proposée est une modification du prouveur afin d'y établir une racine de confiance statique. Cette modification ajoute une fonction d'attestation permettant le calcul d'un tag d'intégrité authentifié.

La stratégie est de réutiliser les algorithmes proposés par HACL⋆ pour garantir la conformité, la sûreté mémoire et l'indépendance au secret. En particulier, la fonction d'attestation calcule un HMAC sur une zone mémoire prédéfinie, contenant un algorithme à attester et un challenge, avec le secret partagé (le partage est réalisé en amont de la mise en production). Tout registre matériel modifié par la fonction d'attestation est rétabli à sa valeur initiale lorsque celle-ci se termine. L'impossibilité du calcul du tag d'intégrité sans connaître le secret est admise [37]. Également, **sans accès (direct ou indirect) au secret, la probabilité qu'un adversaire obtienne des informations sur le secret depuis la zone mémoire attestée et le résultat du calcul est négligeable** [23].

La compilation de la fonction d'attestation est réalisée avec un outil non-certifié mais la préservation sémantique est admise [36] : l'architecture du MSP430 ne fait pas partie des cibles supportées par CompCert. Il est donc considéré axiomatique que, si le pointeur d'instruction du microcontrôleur se trouve à la première instruction de l'algorithme et qu'aucune interruption n'intervient avant que celui-ci n'atteigne la dernière, alors un HMAC correct est obtenu. Cet axiome est formellement exprimé dans un langage de logique temporelle.

1.7.2 Architecture co-conçue

Les travaux proposés par VRASED s'inscrivent dans le cadre d'un adversaire distant, s'exécutant sur le prouveur et possèdant le plus haut niveau de privilèges. L'implémentation est réalisée sur un microcontrôleur MSP430. Une modification lourde du cœur du processeur fournit des observables, tels que le pointeur d'instruction ou les signaux d'interruption, à une extension matérielle. Les travaux apportent la preuve qu'**aucun accès (direct ou indirect) au secret n'est possible par l'adversaire et que l'exécution de la fonction d'attestation n'est jamais interrompue**. Cette preuve établit donc une racine de confiance statique.

L'extension matérielle garantit donc les propriétés de sécurité complémentaires à celles fournies par HACL⋆. Pour vérifier ces propriétés de sécurité, le comportement du microcontrôleur est modélisé à partir des signaux internes, observables suite à une modification du cœur, comme suit :

— le pointeur d'instruction contient l'adresse de l'instruction qui sera exécutée au prochain cycle d'horloge ;

— lorsqu'une lecture ou une écriture est réalisée par le microcontrôleur, le vecteur de donnée contient l'adresse où l'accès est réalisé. Lors d'une lecture, le signal *read-enable* est levé. Lors d'une écriture, le signal *write-enable* est levé.
— lors d'un redémarrage du microcontrôleur, le signal *reset* est levé et tous les registres (dont le pointeur d'instruction) sont réinitialisés. La séquence de démarrage est définie par le matériel et ne peut pas être modifiée sans accès physique. L'état dans lequel le microcontrôleur se trouve après un redémarrage est sûr.
— lorsqu'une exception est levée, le signal *irq* est levé.

Des automates de sécurité assurent les contrôles d'accès au secret, ainsi que l'immuabilité et l'atomicité de l'exécution de la fonction d'attestation. Les signaux précédemment listés constituent les points d'entrée des automates de sécurité. Une ségrégation spatiale du secret, du code de la fonction d'attestation et d'une pile exclusive à son exécution permet de rendre observables les signaux d'accès aux zones mémoire protégées (de type RAM). En fonction de l'évolution de ces signaux, les automates transmettent un *reset* au microcontrôleur en cas de future compromission.

Sur le microcontrôleur MSP430, les seuls emplacements mémoires où la fonction d'attestation peut écrire sont les registres matériels et les différentes RAM. Un axiome garantit que les registres matériels sont réinitialisés après l'exécution de la fonction d'attestation. La ségrégation spatiale proposée par VRASED et le contrôle des accès permettent d'interdire au microcontrôleur d'accéder aux différentes RAM utilisées pour le calcul du tag d'intégrité (lorsque son pointeur d'instruction ne se trouve pas dans la fonction d'attestation).

1.7.3 Soundness et sécurité

Le modèle d'adversaire est le suivant : l'adversaire peut modifier entièrement le code de l'environnement du prouveur. Les étapes suivantes décrivent l'obtention des preuves de *soundness* et de sécurité par VRASED :

1. La *soundness* et la sécurité sont formellement exprimées sous forme de propriétés temporelles. Ces propriétés forment les spécifications. Parmi celles-ci :
   — Tout accès direct au secret ne peut être effectué par du code autre que la fonction d'attestation.
   — Tout espace mémoire écrit par la fonction d'attestation ne peut être lu par l'adversaire.
   — Toute extension matérielle ajoutée au microcontrôleur n'entrave pas l'exécution de la fonction d'attestation.
2. Des axiomes et des propriétés locales sont exprimés dans le même formalisme que les spécifications. Parmi celles-ci :
   — Tant que les espaces mémoire stockant des données utilisées par la fonction d'attestation ne varient pas durant son exécution, le calcul du *HMAC* est correct (conformité fonctionnelle du code garantie par HACL⋆).
   — Les accès directs au secret ne sont autorisés que durant l'exécution de la fonction d'attestation (contrôles d'accès au secret).
   — La fonction d'attestation ne peut être modifiée par un adversaire (immuabilité du code).
   — L'exécution de la fonction d'attestation ne peut être interrompue et se fait de la première instruction à la dernière (atomicité du code).

3. Les preuves de *soundness* et sécurité sont obtenues par réécriture : lorsque l'implication des spécifications par la conjonction des axiomes et propriétés locales est une tautologie.
4. Du *model-checking* valide que le système vérifie les propriétés locales. L'outil de traduction Verilog2SMV [45] transforme la description HDL du système en un formalisme outillé sur lequel les propriétés sont vérifiées.

1.7.3.1 Soundness

La *soundness* est vérifiée lorsque l'ajout des automates de sécurité n'entrave pas l'exécution de la fonction d'attestation [36]. Une spécification formalise que, si le pointeur d'instruction du microcontrôleur se trouve à la première instruction de la fonction d'attestation, et qu'aucune interruption n'intervient avant que celui-ci n'atteigne la dernière, alors un HMAC correct est obtenu. Une conjonction d'axiomes (comportement de la fonction d'attestation, décrit dans la section 1.7.1) et de propriétés locales implique cette spécification. Une réécriture prouve que cette implication est une tautologie.

Cette réécriture est réalisée à l'aide d'un algorithme de réduction, qui implémente des règles d'équivalences entre différentes expressions [48]. Le fonctionnement de cet algorithme est décrit dans le chapitre 3, dans la section 3.3.2. La réécriture est donc automatisée mais dépend des capacités de la machine sur laquelle elle est réalisée, ainsi que de la difficulté du problème à résoudre. Ici, la spécification, les axiomes et les propriétés locales sont choisis suffisamment simples pour permettre une preuve en un temps acceptable.

1.7.3.2 Sécurité

La sécurité de l'attestation à distance est spécifiée sous forme d'un jeu : le système est considéré sécurisé si et seulement si la probabilité qu'un adversaire gagne le jeu en un temps probabiliste polynomial est négligeable. Notons que cette spécification de la sécurité est respectée par l'implémentation de la primitive cryptographique fournie par HACL⋆, dans l'hypothèse où aucun accès (direct ou indirect) au secret n'est possible par l'adversaire.

VRASED vérifie donc la spécification suivante : un adversaire ne peut obtenir d'information sur le secret, soit en accédant directement à son emplacement en mémoire, soit en accédant aux données écrites par la fonction d'attestation. L'adversaire ne peut donc gagner le jeu qu'en un temps probabiliste polynomial identique à celui d'un adversaire d'HACL⋆.

La spécification est formalisée sous forme de propriétés temporelles. Ici aussi, une conjonction d'axiomes (comportement de la fonction d'attestation) et de propriétés locales implique cette spécification. Une réécriture, automatisée par le même algorithme de réduction que précédemment, prouve que cette implication est une tautologie.

1.8 Objectifs de la thèse

Dans cette section, nous présentons les objectifs de la thèse auxquels nous tentons de répondre dans ce manuscrit. Les méthodes de conception de système d'attestation à distance vérifiée formellement sont adaptées, dans l'état de l'art, aux microcontrôleurs simples. Notre objectif principal est d'**étendre ce domaine en réutilisant au mieux les preuves de *soundness* et sécurité et de les appliquer aux microprocesseurs. La solution pro-**

posée devra être adaptée aux paradigmes des systèmes d'information modernes et devra garantir des performances élevées.

Nos travaux s'inscrivent dans la continuité des travaux initiés par De Oliveira Nunes et al. [36], sur l'attestation à distance vérifiée formellement. Nous proposons ici d'établir plusieurs problématiques et les solutions que nous avons envisagées. Finalement, nous proposons un cas d'étude et listons les verrous auxquels nous faisons face et que nous tentons de lever dans ce manuscrit.

1.8.1 Problématiques

Outre le fait qu'une extension aux microprocesseurs apporte une augmentation de la surface d'attaque et que la définition de la sécurité doit être adaptée, nous faisons face à des problématiques concernant le choix du cas d'étude.

Différents constructeurs de microprocesseurs proposent différentes architectures. Ces architectures sont souvent accompagnées de performances élevées mais sont des systèmes fermés [49], utilisables en boite noire, que nous ne pouvons pas étudier et/ou modifier. Néanmoins, des *Systems on Chip* (*SoC*) modernes proposent une architecture où le microprocesseur est étroitement lié avec un circuit FPGA [50]. Du support matériel peut y être implémenté. Également, nous assistons à l'émergence des premières architectures ouvertes et leurs implémentations dans des projets libres [51, 52]. Une modification lourde du cœur du système est donc envisageable sur ces architectures.

Afin de procéder au choix de notre cas d'étude, nous considérons les problématiques suivantes : le maintien des performances, les coûts de conception, les coûts de maintenance, et l'accès à un modèle. Nous définissons ici ces problématiques afin d'argumenter efficacement en matière de solutions possibles pour l'attestation à distance sur microprocesseur.

Maintien des performances

L'ajout d'un mécanisme de sécurité est souvent synonyme de calculs supplémentaires et donc de réduction des performances. Dans le cas où ce mécanisme repose sur du support matériel, la fréquence de fonctionnement devient une problématique. Si nous proposons un système matériel, nous devons nous assurer que le système ne soit pas ralenti en permanence. Nous pouvons accepter des ralentissements durant le calcul du tag d'intégrité mais ceci ne doit pas impacter le fonctionnement "normal", sans quoi cibler un processeur moderne perd tout son sens.

Coûts de conception

Proposer une solution de système vérifié formellement consiste à démontrer l'absence de vulnérabilité vis-à-vis d'un modèle de menaces identifié. L'écriture de preuves formelles est un travail fastidieux et profiter d'une preuve établie sur un système réduit grandement les coûts de conception. Aussi, tout comme VRASED profite de la preuve établie par HACL⋆ pour le code de la fonction d'attestation, nous devons profiter au maximum des preuves établies VRASED pour nos travaux. Cette problématique implique donc de réduire au maximum l'implémentation de fonctionnalités existantes. Cela sous-entend aussi de réutiliser les primitives cryptographiques vérifiées formellement par HACL⋆ et donc de se baser sur du logiciel pour la

fonction d'attestation. En effet, à notre connaissance, il n'existe pas de bibliothèque matérielle vérifiée formellement qui propose des calculs de tags d'intégrité.

Coûts de maintenance

La troisième problématique que nous considérons est celle du coût de maintenance. En effet, proposer du support matériel pour garantir des propriétés de sécurité implique deux possibilités : soit utiliser un composant programmable (souvent avec une fréquence de fonctionnement réduite), soit utiliser un composant immuable. Dans le cas de l'utilisation d'un composant immuable, proposer une maintenance après la découverte d'une vulnérabilité signifie retirer le composant et le remplacer par une nouvelle version, où cette vulnérabilité a été éliminée. Si tout le système est un composant immuable, alors le coût de maintenance devient non-négligeable, d'autant plus lorsque seule une partie du système est à mettre à jour.

Accès aux sources

La dernière problématique que nous considérons est celle de l'accès aux sources du système. En ce qui concerne le support matériel à proposer, la question ne se pose pas : nous concevons nous même le système et nous pouvons le modéliser. Par contre, en ce qui concerne le microprocesseur, utiliser un système ouvert et/ou accompagné de spécifications formelles permet d'avoir un fort degré de confiance en notre modèle, pour son comportement. Ainsi, la vérification formelle se trouve simplifiée. Dans le cas où nous utilisons un microprocesseur fourni par un constructeur de matériel, alors nous devons émettre des hypothèses sur son comportement, que nous traduisons depuis sa documentation. Cela implique avoir une confiance tacite avec la documentation. Même si ceci semble discutable [53], cela reste néanmoins, avec un travail d'ingénierie inverse, la seule solution.

1.8.2 Solutions envisagées

Compte tenu des problématiques que nous venons de citer, nous considérons plusieurs solutions envisageables pour notre cas d'étude de l'attestation à distance sur microprocesseur. Dans cette section, nous décrivons trois possibilités que nous considérons viables : la proposition d'une modification d'un ASIC (*Application-Specific Integrated Circuit*) dont l'architecture est ouverte, l'implémentation d'un système ouvert sur FPGA (*softcore*), et l'utilisation de *SoC* propriétaires qui incluent un FPGA.

Modification d'un système complexe ouvert

Proposer une modification d'un microprocesseur dont l'architecture est ouverte semble être la solution qui offre les meilleurs performances. En effet, si l'objectif est de rester cohérent vis-à-vis des paradigmes des systèmes d'information modernes, alors une modification d'un ASIC est la meilleure solution. La fréquence de fonctionnement pourra être élevée, ce durant le fonctionnement "normal" et durant l'exécution d'une fonction d'attestation. Également, l'aspect ouvert de l'architecture permet d'accéder à un modèle du système et d'obtenir aisément les hypothèses nécessaires à la vérification formelle.

En contrepartie, cette solution se heurte à la fois à des coûts de conception et des coûts de maintenance élevés. D'un point de vue des coûts de conception, la difficulté réside dans

le placement/routage du cœur du système après que celui-ci ait subi une modification lourde (connexions au pointeur d'instruction, aux signaux de lecture/écriture, etc.). Les contraintes de temps de propagation des portes logiques ayant été respectées sur l'ASIC non-modifié, elles peuvent être violées après modification. Respecter ces contraintes implique alors une nouvelle conception ou une réduction de la fréquence de fonctionnement, au détriment des performances.

D'un point de vue des coûts de maintenance, si le système est immuable, alors nous faisons face à la problématique précédemment énoncée : une découverte de vulnérabilité implique un remplacement du système. Lier cet ASIC modifié à un FPGA où du support matériel y est implémenté est une solution plus élégante en termes de maintenance et est également une solution envisageable.

Softcore d'un système ouvert

Une autre solution, dans le cas où nous souhaitons proposer une modification d'un microprocesseur dont l'architecture est ouverte, est d'implémenter celui-ci dans un FPGA, à la manière des *softcores*. Dans ce cas, les coûts de conception et maintenance sont tous deux réduits. En effet, une simple carte de développement comportant un FPGA est nécessaire et le placement/routage sera aisé (compte tenu d'une fréquence d'horloge réduite). La vérification formelle de la solution est simplifiée par l'aspect ouvert de l'architecture, qui pourra être aisément modélisée. De plus, une maintenance du système implique simplement une reprogrammation et ne nécessite aucun remplacement de composant immuable. Ici aussi, l'aspect ouvert de l'architecture facilite la vérification formelle.

Malheureusement, cette solution ne fonctionne qu'à performances réduites. En effet, si les circuits programmables de type FPGA proposent un parallélisme efficace, leurs performances en terme de fréquence de fonctionnement n'atteint pas celle des ASIC. De ce fait, nous nous éloignons de notre objectif qui est de proposer une solution adaptée à un système moderne et cohérent vis-à-vis des paradigmes des systèmes d'information.

SoC propriétaire : périphérique de confiance

La troisième solution que nous envisageons est de tirer profit des *SoC* modernes, qui forment un environnement de travail adapté : les performances sont assurées par un microprocesseur pris sur étagère et du support matériel peut être implémenté dans la partie FPGA. Dans ce cas, aucune modification du microprocesseur ne peut être réalisée : le support matériel doit simplement être connecté à celui-ci, il ne constitue donc pas exactement une extension. Dans ce manuscrit, nous employons le terme *périphérique de confiance* pour désigner ce support matériel. Le périphérique de confiance vient en appui au microprocesseur et peut permettre d'établir une racine de confiance statique. Cette solution présente les meilleurs compromis en termes de performances, coûts de conception et coûts de maintenance. Le microprocesseur est déjà implémenté, donc les contraintes de placement/routage sont respectées. De plus, une simple reprogrammation du FPGA permet la maintenance du système en cas de nouvelle vulnérabilité avérée.

La contrainte majeure est l'aspect propriétaire du microprocesseur : le cœur du système ne peut pas être modifié et il n'est pas possible d'accéder aux observables extraits par VRASED (pointeur d'instruction, signaux de lecture/écriture, etc.). De plus, modéliser son comporte-

ment requiert une traduction manuelle de la documentation : nous ne pouvons pas traduire son HDL en un formalisme outillé sur lequel les propriétés peuvent être vérifiées.

Le tableau 1.1 récapitule les points forts et faiblesses des différentes solutions que nous envisageons, en fonction des problématiques considérées.

#	Solution	Performances	Conception	Maintenance	Modélisation
1	Modification	✔	✘	✘	✔
2	*Softcore*	✘	✔	✔	✔
3	*SoC*	✔	✔	✔	✘

TABLE 1.1 – Solutions envisagées : problématiques considérées

Notons que pour chacune des solutions, nous disposons d'un microprocesseur capable d'exécuter du logiciel et de support matériel dont nous maîtrisons la conception. Il est possible d'envisager, pour le calcul du tag d'intégrité, d'exécuter une primitive cryptographique logicielle, fournie par HACL⋆ et par conséquent, vérifiée formellement ; ceci requiert une maîtrise de l'exécution réalisée par le microprocesseur. Nous pouvons également envisager d'implémenter une primitive cryptographique matérielle et ainsi garantir une exécution correcte quelque soit l'état du microprocesseur ; ceci requiert une vérification formelle de l'implémentation.

1.8.3 Cas d'étude

Nous considérons toutes les solutions listées précédemment comme valides. Chacune peut être considérée comme un travail de recherche. À notre connaissance, il n'existe pas d'implémentation pour ces solutions. Dans cette thèse, nous avons choisi de nous consacrer à l'attestation à distance sur les *SoC*. Cette décision est justifiée par les raisons de maintien de performances et de coûts, mais certains des travaux que nous décrivons dans ce manuscrit peuvent être appliqués aux autres solutions.

Nous traitons donc le cas des microprocesseurs ARMv7, sur lesquels nous pouvons apporter du support avec un périphérique de confiance, mais dont nous ne pouvons pas modifier le cœur. Le *SoC* choisi est un Xilinx Zynq-7000. Cet environnement de travail nous permet de développer une application adaptée à un système moderne, que nous pouvons aisément tester sur une carte de développement. Des rappels techniques à propos de l'architecture du Zynq-7000, utiles à la compréhension de ce manuscrit, sont fournis dans le chapitre 2.

La solution que nous proposons doit être vérifiée formellement, cela implique la modélisation du système (du microprocesseur et de son périphérique de confiance) **et une formalisation des spécifications.** Nous avons choisi une approche de vérification de bas vers le haut : c'est-à-dire que nous développons d'abord du matériel dans un langage HDL, dont nous réalisons une abstraction pour obtenir un modèle haut niveau et pouvoir jouer des preuves. La stratégie que nous envisageons, à propos de la modélisation, de la formalisation de spécifications et de la vérification, est décrite dans le chapitre 3.

Nous éludons la problématique de la définition du test d'intégrité. En effet, bien que définir les zones mémoires à attester soit important pour garantir les propriétés de sécurité attendues par le vérifieur, nous considérons que cette étape est dépendante de l'application. De plus,

notre test d'intégrité consiste à exécuter du logiciel sur le prouveur : toutes les zones mémoires accessibles par celui-ci peuvent donc être incluses dans le calcul du tag d'intégrité. Nous considérons donc que cette définition pourra être réalisée une fois la racine de confiance établie.

1.8.4 Contributions

Dans les travaux de cette thèse, nous avons fait le choix de travailler avec un microprocesseur pris sur étagère, que nous ne pouvons pas modifier. Afin de réaliser une vérification formelle de notre solution, nous exprimons ses spécifications en fonction des données disponibles dans la documentation. Une telle spécification implique que **nous ne pouvons pas nous limiter aux systèmes simples, dont la vérification automatique peut être réalisée en un temps acceptable**. En effet, concevoir un système qui vérifie ces spécifications et réaliser une vérification sur son modèle se heurte à une explosion combinatoire. Afin de passer la vérification à l'échelle, **nous proposons donc une méthode pour la vérification formelle automatisée de systèmes matériels**. Cette contribution est décrite dans le chapitre 4.

Une fois le passage à l'échelle de la vérification formelle possible, nous souhaitons permettre l'établissement d'une racine de confiance statique pour l'attestation à distance. Comme le microprocesseur est un système fermé, nous ne pouvons pas modifier son cœur et n'avons donc pas accès aux observables nécessaires à la sécurité de l'exécution de la fonction d'attestation (identifiés par VRASED). Afin de pallier ce manque, **nous proposons donc une architecture permettant d'obtenir les mêmes propriétés de sécurité à partir de nouveaux observables, sans modification du cœur**. Également, **nous modélisons le comportement du microprocesseur et de ses périphériques, puis accompagnons la solution que nous proposons d'une vérification formelle**. Cette contribution est décrite dans le chapitre 5.

Nous choisissons, tout comme VRASED, de profiter de la preuve de sécurité apportée par HACL⋆ en exécutant une primitive cryptographique logicielle pour la fonction d'attestation. Cette exécution se fait dans un environnement potentiellement corrompu (sur le prouveur) et le microprocesseur fournit une surface d'attaque plus large qu'un microcontrôleur simple. Nous faisons confiance au microprocesseur pour fonctionner, pour une configuration donnée, de manière identique à celle décrite dans sa documentation. Cependant, nous ne faisons pas confiance à cette configuration lors de l'exécution de la fonction d'attestation. Nous **proposons donc une solution d'attestation de configuration du microprocesseur et de ses périphériques, afin que l'environnement d'exécution ne mette pas à mal les propriétés garanties par la fonction d'attestation**. Cette contribution est décrite dans le chapitre 6.

Chapitre 2

Architecture du *Systems On Chip* Zynq-7000

Sommaire

Pour notre cas d'étude, nous avons choisi, comme environnement de travail, les microprocesseurs ARMv7. Plus particulièrement, ceux étroitement liés à un circuit programmable de type FPGA dans les *SoC* Xilinx Zynq-7000. Pour réaliser nos tests, nous avons choisi comme carte de développement une Cora-Z7, conçue et distribuée par Digilent.

Dans ce chapitre, nous proposons des rappels techniques sur cette architecture et décrivons les méthodes de programmation que nous utilisons. La section 2.1 présente l'architecture du Zynq-7000, en particulier les capacités qu'offre le FPGA. La section 2.2 présente les fonctionnalités du microprocesseur ARMv7 Cortex A9, que nous utilisons dans cette thèse. Finalement, les sections 2.3 et 2.4 détaillent respectivement le protocole de communication matériel AXI-Lite et l'interface de débug *CoreSight*, dont nous tirons profit dans cette thèse.

Dans une optique de rejeu des expériences illustrant nos travaux, les environnements de développement que nous préconisons pour l'implémentation finale et pour étudier le comportement du microprocesseur sont décrits en annexe A, dans la section A.1.

2.1 Circuit logique programmable

Le *SoC* Zynq-7000 est un produit conçu par Xilinx ; il existe en plusieurs versions où les modèles du microprocesseur et du FPGA varient [54]. Toutes les versions incluent un microprocesseur ARM Cortex-A9 ; le nombre de cœurs et la fréquence de fonctionnement nominale peuvent varier. Le FPGA peut être un Artix-7 ou un Kintex-7 ; le nombre de cellules logiques et blocs RAM peut varier.

Nous travaillons avec une carte de développement Digilent Cora-Z7 contenant le modèle du Zynq-7000 XC7007S. Le microprocesseur ne contient qu'un seul cœur et sa fréquence de

fonctionnement nominale est de $667MHz$. Les cœurs cortex A9 ne supportent pas le *simultaneous multithreading* (SMT). Le FPGA est un Artix-7 contenant 23K cellules logiques et $1.8Mb$ de blocs RAM. C'est le modèle le moins coûteux, mais il est suffisant pour nos travaux. Notons que l'absence de multi-cœurs et de SMT sont même des prérequis, comme justifié dans le chapitre 6.

Dans cette section, nous décrivons la méthode de programmation du FPGA Artix-7, les fonctionnalités que nous utilisons et les composants fournis par le constructeur.

2.1.1 FPGA Artix-7

Un FPGA est un circuit dont le comportement est programmable. Ce type de circuit s'appuie sur des cellules logiques contenant elles-mêmes des mémoires et des portes logiques qui peuvent être interconnectées. La programmation de la logique du circuit est réalisée à partir d'un *bitstream* qui décrit exhaustivement la configuration des interconnexions et l'état initial des mémoires. La somme des composants matériels du FPGA et sa configuration simule un modèle de circuit de plus haut niveau généralement décrit dans un langage *Register Transfer Level* (RTL).

Plus précisément, la sémantique d'un *bitstream* est directement dépendante des composants du FPGA. Aussi, la configuration du circuit est stockée de façon éphémère dans une mémoire de configuration. Cette mémoire de configuration est constituée par interprétation des commandes du *bitsteam* par un chargeur embarqué. Enfin, il est par conséquent possible de reprogrammer à volonté un FPGA afin de simuler d'autres circuits.

Les paragraphes suivants décrivent un flot de conception représentatif des FPGA Xilinx. La synthèse, le *place-and-route* et la génération du *bitstream* sont réalisés par le logiciel, propriétaire mais distribué gratuitement, Xilinx Vivado.

La synthèse consiste à traduire une description du circuit au niveau RTL en une connexion de composants, disponibles dans le FPGA et modélisés dans une bibliothèque. Cette bibliothèque est fournie par le constructeur. Le *place-and-route* consiste à sélectionner les composants qui sont utilisés ainsi que leurs connexions sur le système physique. Cette étape est réalisée en fonction d'une liste de contraintes fournies par l'utilisateur et est hautement dépendante du FPGA cible. La génération du *bitstream* formate les données conformément au fonctionnement du composant de programmation présent dans le FPGA. Ce fonctionnement est laissé opaque par le constructeur et l'algorithme de génération du *bitstream* n'est pas documentée [55].

La synthèse d'un circuit décrit en *Verilog* peut être réalisée par *yosys* si la bibliothèque fournie par Xilinx lui est importée [47]. Xilinx Vivado permet la synthèse de circuit décrits en *Verilog*, *VHDL*, et depuis une description graphique du circuit, sous forme de schéma modifiable depuis l'interface graphique. Une combinaison de ces trois types de description peut être fournie à l'outil. Le processus de synthèse, de *place-and-route* et de génération du *bitstream* peut être réalisé depuis l'interface graphique de Xilinx Vivado ou depuis son interface de programmation [56]. Dans cette thèse, nous utilisons une combinaison de schémas et de *Verilog* afin de décrire nos circuits. Nous réalisons les schémas à l'aide de l'interface de programmation.

Ressources : le dossier `2_1_1_vivado/` contient un script `vivadoMakefile.tcl` qui permet d'automatiser la synthèse, le *place-and-route* et la génération du *bitstream* avec Xilinx Vivado.
Également, un exemple de description de circuit, écrit en *Verilog*, est fourni ainsi qu'un fichier de contraintes. Il est possible de générer un *bitstream* à partir de ces éléments. Le `Makefile` permet d'automatiser l'exécution du script `vivadoMakefile.tcl`.

La fréquence de fonctionnement du FPGA est déterminée par un signal d'horloge. Ce signal peut être fourni par un quartz externe, présent sur la carte de développement Cora-Z7, ou par le microprocesseur ARM à l'aide d'une interface disponible dans le *SoC* [50]. Dans le cas de l'utilisation d'une horloge externe, le fonctionnement du FPGA est indépendant du microprocesseur. Dans le cas d'une horloge fournie par le microprocesseur, une synchronisation avec celui-ci devient possible.

2.1.2 IP Cores Xilinx

La connexion entre le FPGA et le microprocesseur, à l'aide de son interface dédiée, est réalisée lors du *place-and-route*. Pour que cette connexion soit rendue possible, le microprocesseur doit faire partie du schéma décrivant le circuit. Une configuration de ses périphériques est alors nécessaire pour permettre le bon fonctionnement des interactions (fréquences d'horloges, initialisation des périphériques, etc.). Lorsque le microprocesseur fait partie du circuit, il est alors connecté, lors de la synthèse, comme un composant non-programmable (d'un point de vue matériel). Xilinx fait référence à ce composant sous l'appellation *Processing System PS7*.

Il est également possible de connecter d'autres composants, dont la description au niveau RTL n'est pas fournie, mais qui sont néanmoins programmés sur le FPGA. Le constructeur fournit une bibliothèque de composants, programmables ou non, qui peuvent être utilisés dans le schéma [57]. Dans ce cas nous parlons d'*IP cores*[1]. Parmi ces *IP cores*, nous trouvons des interfaces de communication, des contrôleurs de mémoires, etc.

Afin de permettre au microprocesseur de communiquer avec les *IP cores*, lorsque celui-ci fait partie du circuit, les outils de la suite Xilinx Vivado automatisent les tâches de configuration des périphériques en fonction des connexions réalisées sur le schéma. Cette automatisation se présente sous la forme d'un programme, décrit en langage C. Un fichier d'initialisation, sous-ensemble de ce programme, est généré lors du *place-and-route* et permet entre autres l'initialisation des périphériques. Le contenu de ce fichier d'initialisation varie donc automatiquement en fonction des connexions présentes sur le schéma.

Ressources : le dossier `2_1_2_ipcore/` contient un exemple de script TCL permettant d'instancier une IPcore Xilinx : le fichier `hw/schematic.tcl`. L'IPcore choisie est le *Processing System PS7*, soit le cœur ARM-v7 contenu dans le *SoC*. Elle est connectée à des modules écrits en *Verilog* et instanciés dans le FPGA. Le fichier `hw/design_0.pdf` fournit le schéma réalisé par le script TCL.
Il est possible de générer un *bitstream* et les fichiers de configuration du *Processing System PS7* à partir de ces éléments. Le `Makefile` permet d'automatiser ces tâches.

1. IP = *Intellectual Property*

2.2 ARMv7 Cortex-A9

Lorsque le *Processing System PS7* fait partie du circuit à implémenter, nous pouvons utiliser les fonctionnalités du microprocesseur ARMv7 Cortex-A9 et de ces périphériques. Pour cela, des programmes de configuration sont nécessaires. Dans cette section, nous décrivons l'environnement logiciel qui s'exécute sur le microprocesseur. Cette exécution se fait après la programmation du FPGA depuis un *bitstream*.

2.2.1 *First Stage Boot Loader*

Un programme simple d'initialisation résident est exécuté sur le microprocesseur lors de la mise sous tension (*boot rom*). Une des responsabilités principales de ce programme est de configurer et exécuter un chargeur de démarrage plus complexe. Xilinx fait référence à ce programme sous l'appellation *First Stage Boot Loader* (FSBL). Les sources d'un FSBL adapté au microprocesseur présent dans le Zynq-7000 sont fournies par le constructeur [50]. Accompagné d'un fichier d'initialisation généré lors de la synthèse, ce FSBL permet une initialisation des périphériques pour les rendre prêts à communiquer avec les *IP cores* programmés dans le FPGA.

Un environnement de démarrage sécurisé est également fourni par le constructeur. Il permet de vérifier l'intégrité d'un FSBL et d'un *bitstream* chiffrés. Dans le mode de fonctionnement principal, la *boot rom* vérifie dans un premier temps l'intégrité de l'image binaire du FSBL qui va lui même vérifier l'intégrité du *bitstream* avant de programmer le FPGA. Dans ce cas, la donnée est présente dans une mémoire flash physiquement connectée au *SoC*. Ce protocole garantit que la séquence de démarrage ne peut pas être corrompue sans accès physique et que l'état dans lequel le microprocesseur se trouve après un redémarrage est sûr. Dans nos travaux, nous n'utilisons pas cette fonctionnalité mais considérons qu'elle sera utilisée lors de la mise en production. De ce fait, nous considérons axiomatique que provoquer un *reset* du microprocesseur permet le recouvrement.

Le constructeur fournit un environnement de développement dédié à la modification et à la compilation du FSBL [58]. Cet environnement de développement est propriétaire mais réalise une instrumentation du compilateur libre `gcc`. Afin de maîtriser la chaîne de compilation et améliorer les performances, nous avons choisi de nous affranchir de cette solution et d'utiliser seulement les outils de la suite `gcc`. Nous redistribuons, avec les sources de nos travaux, une copie des sources d'un FSBL développé par Xilinx, dont la licence permet une telle redistribution. Nous accompagnons ces sources d'une recette de compilation (de type `Makefile`), permettant la compilation et l'édition des liens du FSBL avec les outils de la suite `gcc`.

Ressources : le dossier `2_2_1_fsbl/` contient les sources pour compiler un FSBL compatible avec le Zynq-7000 XC7Z007S, disponible sur la carte de développement Digilent Cora-Z7. Les fichiers d'initialisation `ps7_init.c` et `ps7_init.h` sont absents des sources, ils doivent être générés avec Xilinx vivado lors d'une synthèse d'un circuit contenant le *Processing System PS7*.
Un exemple de fichiers d'initialisation est fourni dans le même dossier. Un FSBL peut être généré lors d'une compilation avec *gcc*. Le `Makefile` permet d'automatiser la compilation.

2.2.2 Application

Après la mise sous tension du *SoC*, le FSBL s'exécute et se termine, soit par une mise en attente du microprocesseur (cas d'une image ne comportant que le FSBL), soit par un branchement vers un second programme (cas d'une image comportant plusieurs programmes). Xilinx fait référence à ce second programme sous l'appellation *application*. L'application peut être un programme auto suffisant ou un système d'exploitation complet. Il est possible, depuis l'application, de modifier de nouveau l'état du microprocesseur et de ses périphériques, pourvu que son exécution soit réalisée avec des privilèges suffisamment élevés. Dans notre cas d'étude, l'environnement logiciel du prouveur se limite à l'application. C'est donc depuis cette application que l'adversaire s'exécute avec le plus haut niveau de privilèges. C'est aussi depuis cette application que la partie logicielle de notre racine de confiance s'exécute.

L'application peut donc interagir avec les périphériques du microprocesseur et, dans le cas où le *bitstream* programmé permet une connexion avec des *IP cores*, des composants implémentés dans le FPGA. Parmi les périphériques, nous pouvons citer le module d'interconnexion *Advanced eXtensible Interface* (AXI), dont le fonctionnement est décrit dans la section 2.3 et l'interface de débug *CoreSight*, dont le fonctionnement est décrit dans la section 2.4.

Ressources : le dossier `2_2_2_app/` contient les sources pour compiler une application compatible avec le Xilinx Zynq-7000. Ce programme transmet un message via la liaison série et effectue des lectures et écritures sur les ports à usage général (GPIO) du FPGA. Le `Makefile` permet d'automatiser la compilation.

Pour communiquer avec les périphériques et, par extension, avec les composants implémentés dans le FPGA, l'exécution d'instructions spécifiques est nécessaire. Le microprocesseur supporte l'exécution d'instructions de type ARM, *Thumb* et *Jazelle* [49]. Dans cette thèse, nous n'utilisons que des instructions de type ARM, mais l'adversaire considéré peut exécuter des instructions de tous les types pour modifier le comportement des périphériques. Parmi les instructions que nous utilisons, nous pouvons citer celles provoquant un branchement indirect [49]. Le terme **branchement** désigne un saut dans l'exécution. Un branchement est dit **indirect** lorsque l'adresse de destination n'est pas une valeur immédiate. Si *CoreSight* est configurée pour fournir des traces d'exécution, l'adresse de destination d'un branchement indirect est donnée dans la trace [59].

2.3 Protocole AXI-Lite

AXI est un protocole de communication synchrone, avec une horloge unique, qui suit le paradigme maître-esclaves. Le protocole AXI fait partie de la spécification des bus de communication *Advanced Microcontroller Bus Architecture*, le standard ouvert de ARM [60]. Une communication entre un maître AXI et un esclave AXI permet la lecture et l'écriture de données dans l'esclave. L'échange de donnée lors de cette communication est appelé *transaction*. AXI-Lite est une variante du protocole AXI, où chaque adresse de lecture/écriture doit être explicitement spécifiée pour que la transaction ait lieu. La communication entre un maître AXI et un esclave AXI-Lite est possible [60].

Le microprocesseur ARM Cortex-A9 possède un périphérique d'interconnexion AXI maître. Les *IP cores* fournis par Xilinx permettent l'implémentation et la connexion de périphériques

d'interconnexion maître et d'esclaves dans le FPGA. Les travaux décrits dans ce manuscrit s'appuient sur l'utilisation d'esclaves AXI-Lite et une observation des signaux de transaction pour établir une racine de confiance statique. La figure 2.1 montre les connexions établies au niveau du bus entre une interface d'interconnexion maître AXI et un esclave AXI-Lite. Les signaux sont nommés en fonction de la spécification du protocole [60].

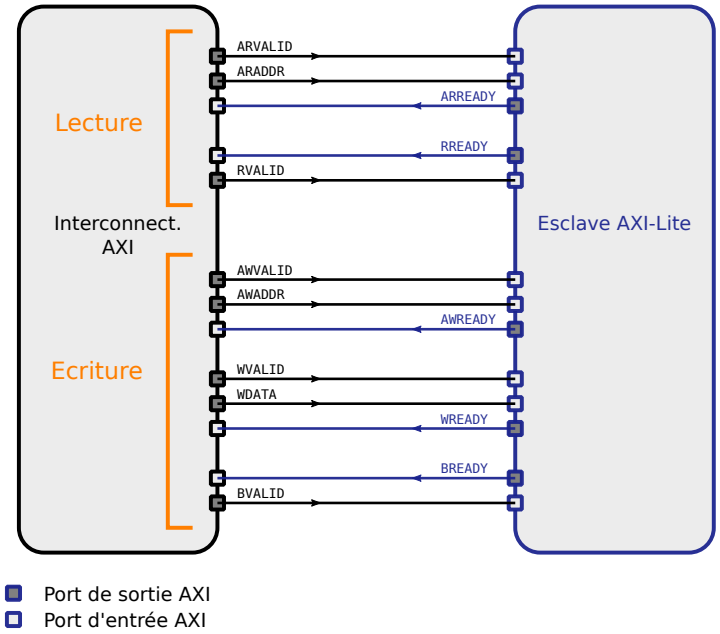

FIGURE 2.1 – Bus de communication entre un maître AXI et un esclave AXI-Lite

Lors d'un accès mémoire sur l'un des esclaves, une requête est envoyée par le maître et l'esclave répond lorsqu'il est prêt. Les étapes suivantes décrivent une transaction de lecture depuis un esclave AXI-Lite :

1. le maître lève le signal *ARVALID* pour annoncer qu'il souhaite initier une transaction. Le maître transmet également l'adresse à laquelle il souhaite lire sur le vecteur *ARADDR*.
2. l'esclave répond à la requête en levant le signal *ARREADY*.
3. la transaction a lieu lorsque *ARVALID* et *ARREADY* sont levés sur le même front d'horloge.
4. le maître et l'esclave accusent réception/envoi avec respectivement les signaux *RVALID* et *RREADY*.

Les étapes suivantes décrivent une transaction d'écriture dans un esclave AXI-Lite :

1. le maître lève les signaux *AWVALID* et *WVALID* pour annoncer qu'il souhaite initier une transaction. Le maître transmet, en même temps, l'adresse relative à laquelle il souhaite écrire sur le vecteur *AWADDR*. Également, il transmet la valeur qu'il souhaite écrire sur le vecteur *WDATA*.
2. l'esclave répond à la requête en levant les signaux *AWREADY* et *WREADY*.
3. la transaction a lieu lorsque *AWVALID*, *WVALID*, *AWREADY* et *WREADY* sont levés sur le même front d'horloge.

4. le maître et l'esclave accusent envoi/réception avec respectivement les signaux *BREADY* et *BVALID*.

La figure 2.2 montre un exemple de plusieurs transactions sur le bus AXI-Lite. Les signaux représentés en noir sont émis par le maître tandis que les signaux représentés en bleu sont émis par l'esclave. La transaction 1 représente un accès en lecture tandis que les transactions 2, 3 et 4 représentent des accès en écriture. Les signaux d'accusation d'envoi/réception *RREADY*, *RVALID*, *BREADY* et *BVALID* sont omis pour simplifier la lecture.

La différence entre les transactions 2, 3 et 4 est l'ordre de présence des signaux *AWVALID* et *WVALID*. Ces signaux sont levés l'un après l'autre ou tous les deux en même temps. Dans les trois cas, la transaction est valide. Néanmoins, celle-ci n'a lieu que lorsque les trois signaux *AWVALID*, *WVALID*, *AWREADY* et *WREADY* sont levés.

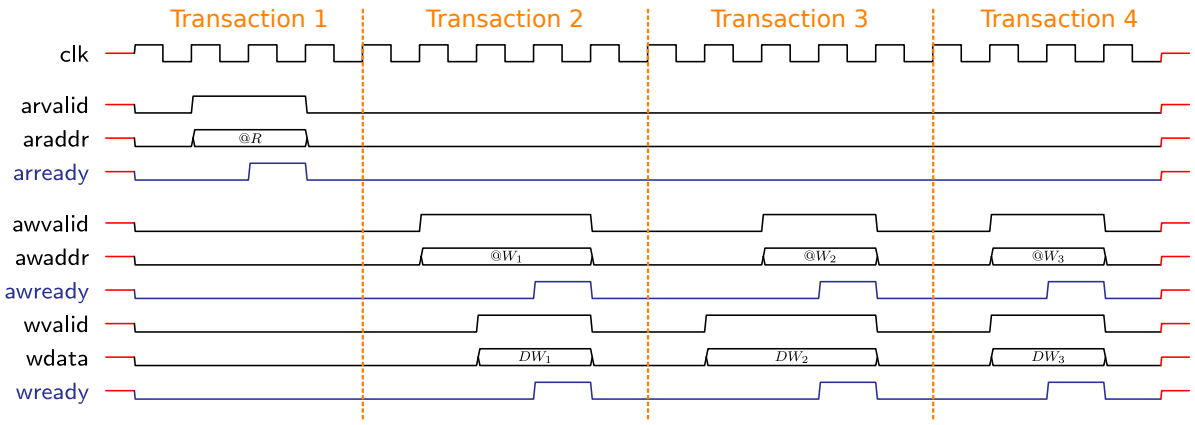

FIGURE 2.2 – Exemple d'une lecture et trois écritures sur le bus AXI-Lite

Le protocole AXI-Lite prévoit que le signal d'horloge, permettant la synchronisation entre le maître et les esclaves, soit fourni par le maître. Dans le cas de l'utilisation du périphérique d'interconnexion AXI maître du microprocesseur, une entrée est prévue pour fournir une horloge externe. Une configuration du périphérique permet de choisir l'horloge utilisée pour la synchronisation : soit une horloge interne au microprocesseur, dont la fréquence peut être paramétrée, soit l'horloge externe connectée sur l'entrée prévue à cet effet [50].

2.4 Interface de débug *CoreSight*

CoreSight est une architecture matérielle ARM permettant le débogage de microprocesseurs. Cette architecture permet, entre autres, de réaliser des traces d'exécution dans un format décrit par le standard ARM *Program Flow Trace* (PFT) [59]. Une trace est une liste ordonnées d'états du microprocesseur qui est représentative d'une exécution. Dans le cas de *PFT*, les traces représentent les branchements exécutés par le processeur. Dans le cœur Cortex-A9 que nous utilisons, PFT est implémenté par le module Program Trace Macrocell (PTM) [61]. PTM propose un protocole de collecte non-invasive de ces traces sans ralentir le microprocesseur. Les traces de *CoreSight* peuvent être communiquées à un périphérique matériel extérieur au microprocesseur, par exemple présent dans le FPGA.

La connexion entre le PTM et le FPGA se fait par l'intermédiaire du port TPIU [62]. Un port d'entrée permet de fournir l'horloge utilisée pour la synchronisation. Cette horloge peut être externe (fournie par le FPGA) ou interne au microprocesseur. La communication est unidirectionnelle : le PTM émet des traces d'exécution mais ne peut pas recevoir de donnée.

Le TPIU permet d'accéder aux traces d'exécution émises, au travers de trois signaux : *TRACE_CLK*, *TRACE_CTL* et le vecteur *TRACE_DATA*. Le signal *TRACE_CLK* est une image de l'horloge utilisée pour la synchronisation, dont la fréquence est divisée par deux. Le signal *TRACE_CTL* indique si de la donnée est présente ou non. Dans le cas où de la donnée est présente, celle-ci est placée sur le vecteur *TRACE_DATA*. La largeur du vecteur peut-être choisie entre 8, 16 et 32 bits, selon la configuration du PTM. L'émission d'une trace est une succession de vecteurs de données, transmis en série sur le vecteur *TRACE_DATA*. Dans la suite de cette thèse, nous utilisons la dénomination *CoreSight* pour faire référence au protocole de trace PFT mis en œuvre pour le module PTM.

CoreSight peut être configuré pour émettre des traces d'exécutions si et seulement si le cœur exécute un programme présent dans certaines plages d'adresses mémoire. Nous parlons alors de *plages d'adresses à tracer*. Les traces d'exécutions sont une succession de *paquets*, dont la taille est variable et alignée sur 8 bits. Le constructeur ARM fournit une description des différents paquets dans la documentation [59]. Pour notre cas d'étude, certains paquets sont utiles à l'établissement de notre racine de confiance statique, mais pas tous. Nous nous intéressons aux paquets de type *isync* et *branch*, qui contiennent l'adresse de destination du pointeur d'instruction et les informations sur les signaux d'exception.

CoreSight émet un paquet de type *isync* lorsque le pointeur d'instruction fait sa première entrée, de manière linéaire (sans branchement, *fall-through*) dans une plage d'adresses à tracer. *CoreSight* émet un paquet de type *branch* lorsque :

— le flot d'exécution entre dans une plage d'adresses à tracer à l'aide d'un branchement (direct ou indirect) ;
— le pointeur d'instruction se situe dans une plage d'adresses à tracer et un branchement indirect est exécuté ou une exception est levée.

Un paquet est composé de un ou plusieurs octets. Le premier octet, appelé *entête*, indique le type du paquet. Le format du paquet dépend de la configuration de *CoreSight*, de la valeur du pointeur d'instruction et du jeu d'instructions à cette adresse. Les paquets sont également *compressés* de telle sorte que les éléments de l'état du cœur inchangées entre deux émissions ne soient pas répétés [59]. Par exemple, dans le cas de branchement indirect ou d'exception, seuls les bits modifiés du pointeur d'instruction sont transmis dans les paquets. Aussi, lors de l'exécution de branchements conditionnels, dont l'adresse est en valeur immédiate, la trace n'exprime que la vérité du prédicat de branchement. En effet, le pointeur d'instruction peut être déduit par analyse statique du programme. La figure 2.3 fournit un exemple, extrait de la documentation, de paquet de type *branch* dans le cas où le jeu d'instructions à l'adresse de destination est de type ARM [59].

Dans cet exemple, le bit de poids faible de l'entête permet d'identifier le type de paquet *branch*. Les bits marqués `C` sont des indicateurs de compression : si ce bit est à 1, alors l'octet suivant est transmis ; sinon, il est omis. La taille d'un paquet *branch* peut donc varier entre 1 et 7 octets. La documentation [59] donne l'emplacement des bits de compression pour tous les types de paquets que *CoreSight* peut émettre.

Même si certains types de paquets ne sont pas utiles pour établir notre racine de confiance

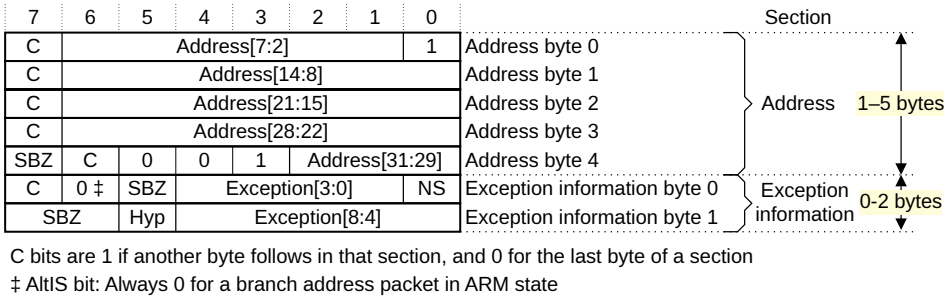

FIGURE 2.3 – Détails du paquet de type *branch*, jeu d'instructions ARM

statique, *CoreSight* peut tout de même les émettre. De plus, l'émission des paquets se fait en série sur le vecteur *TRACE_DATA*. Afin d'identifier les entêtes des paquets *isync* et *branch*, il est donc nécessaire de connaitre la taille de tous les paquets émis précédemment, et donc de décompresser tous les paquets que *CoreSight* peut émettre. Une description de la décompression que nous réalisons est fournie dans la section 4.3.1.

2.5 Conclusion

Dans ce chapitre, nous avons fourni des rappels techniques à propos de l'architecture du Zynq-7000. Nous avons décrit le processus de synthèse et d'utilisation des IP Cores (fournies par Xilinx) sur les FPGA Artix-7, présents dans la partie circuit logique programmable du *SoC*. Dans les prochains chapitres, nous décrirons la vérification formelle de notre solution, qui sera réalisée sur les systèmes matériels implémentés dans ce FPGA et composée de tels IP Cores.

Également, nous avons décrit l'environnement logiciel qui s'exécute sur le microprocesseur ARMv7 Coretx-A9 (FSBL et application). Lorsque nous traiterons le cas de l'attestation à distance sur un système moderne, le prouveur sera composé d'un ensemble logiciel s'exécutant sur un tel microprocesseur. La configuration de son environnement et de celui de ses périphériques, ainsi que les attaques réalisée par l'adversaire, seront réalisées dans ce même environnement.

Finalement, nous avons décrit les protocoles AXI-Lite et le fonctionnement de l'interface de débug *CoreSight*. Ceci représente le fonctionnement matériel des périphériques du microprocesseur ARMv7 Coretx-A9. Nous nous appuyerons sur ces fonctionnalités pour vérifier nos propriétés de sécurité ; les spécifications que nous exprimerons pour notre solution s'appuieront sur le comportement de ces périphériques.

Dans le chapitre 3, nous décrivons la stratégie de vérification formelle que nous envisageons pour les systèmes matériels implémentés dans le FPGA Artix-7.

Chapitre 3

Vérification formelle : stratégie proposée

Sommaire

Dans ce chapitre, nous décrivons la stratégie de vérification formelle que nous proposons d'employer pour l'élimination des vulnérabilités. Cette description est accompagnée de rappels techniques sur la modélisation et la vérification. La section 3.1 décrit comment assurer la préservation de la preuve de sécurité jusqu'à l'implémentation finale du système. La section 3.2 décrit notre stratégie de modélisation des systèmes et de leurs spécifications. La section 3.3 décrit notre stratégie de vérification, c'est-à-dire comment est-ce que nous garantissons l'absence de vulnérabilité et établissons notre racine de confiance statique.

Dans une optique de rejeu des expériences illustrant nos travaux, des rappels techniques et exemples de modélisation et de vérification sont fournis en annexe A, dans la section A.2.

3.1 De la preuve à l'implémentation

Dans cette thèse, notre objectif est de 1) vérifier formellement l'absence de vulnérabilité sur un périphérique de confiance et 2) l'implémenter. Ce périphérique de confiance apporte du support à un système existant. La vérification formelle est réalisée à partir d'un modèle abstrait des éléments qui le composent : par exemple, une description du logiciel avant sa compilation ou une description du matériel avant sa synthèse et son *place-and-route*. Pour chaque sous-système composant le périphérique de confiance, une preuve est établie que des spécifications sont vérifiées. Afin que cette preuve soit valable sur l'implémentation finale, les outils qui composent la chaine de transformation du modèle (compilation, synthèse, etc.) doivent être de confiance pour préserver les propriétés vérifiées. Dans cette section, nous proposons une solution de préservation des propriétés vérifiées pour les différents outils de transformation.

3.1.1 Implémentation du logiciel

Dans le cadre de l'attestation à distance, la sécurité d'une implémentation est basée sur la conformité de l'exécution d'une fonction d'attestation. Cette fonction comporte entre autres des primitives cryptographiques dont les exigences en matière de sécurité sont fortes. Pour cette raison, tout comme VRASED [36], nous proposons de réutiliser une bibliothèque logicielle vérifiée formellement, qui fournit une implémentation d'algorithmes cryptographiques modernes : HACL⋆ [23]. En particulier, nous utilisons la fonction `EverCrypt_HMAC_compute_sha2_256()` de la bibliothèque, qui calcule un HMAC-SHA256 sur une zone mémoire donnée et écrit le résultat à une adresse choisie. Le code C vérifié de cette fonction est lui aussi distribué librement [1] par la communauté du projet Everest [63]. Nous réutilisons ce code pour nos travaux. En conséquence, nous profitons de sa preuve de sécurité et considérons les propriétés vérifiées comme axiomatiques avant la compilation.

Depuis le code C réutilisé, nous proposons d'utiliser le compilateur CompCert [22] pour obtenir un programme, en assembleur ARM, exécutable sur le microprocesseur du Zynq-7000. Pour rappel, CompCert est un compilateur vérifié formellement. Cette vérification certifie que le programme exécutable produit préserve la sémantique du code source utilisé [22]. Nous considérons donc les propriétés vérifiées par HACL⋆ comme axiomatiques lors de l'exécution de la fonction `EverCrypt_HMAC_compute_sha2_256()` sur le microprocesseur du Zynq-7000.

CompCert fournit également un outil d'édition des liens, même si celui-ci n'est pas vérifié formellement [24]. À des fins de prototypage, nous pouvons utiliser les outils standard d'édition des liens pour lier des objets générés par CompCert. Cette solution est adaptée aux projets de co-conception réalisés avec la suite Xilinx Vivado qui utilise gcc et non CompCert [64]. Informellement, nous émettons l'hypothèse de transformation **T1** que l'utilisation d'outils d'édition des liens de la suite gcc n'altère pas la sémantique de la fonction `EverCrypt_HMAC_compute_sha2_256()` compilée avec CompCert. Ceci est une faiblesse, car la préservation sémantique n'est formellement garantie que dans le cas d'un projet compilé, assemblé et lié avec CompCert [22]. L'environnement de compilation devra donc être adapté avant la mise en production.

Le schéma représenté sur la figure 3.1 résume l'utilisation de l'infrastructure logicielle qui permet la préservation des propriétés vérifiées sur le logiciel jusqu'à la création d'un exécutable.

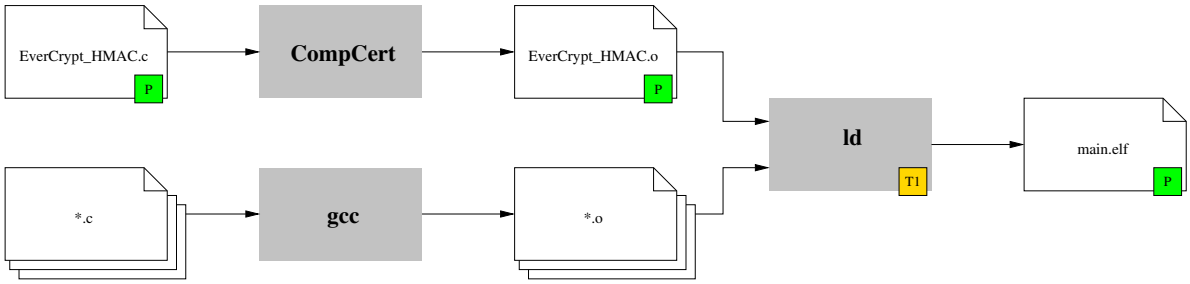

FIGURE 3.1 – Compilation d'un projet en préservant les propriétés de sécurité

1. `https://github.com/project-everest/hacl-star`

La fonction `EverCrypt_HMAC_compute_sha2_256()` garantit des propriétés de sécurité **P**, elle est décrite dans le fichier `EverCrypt_HMAC.c`. Ce fichier est compilé en assembleur ARM par CompCert. Le reste du projet est compilé avec gcc. L'hypothèse de transformation **T1** suggère que les propriétés **P** sont préservées lors de l'édition des liens et donc jusqu'à la création de l'exécutable.

Ressources : le dossier `3_1_1_hmac/` contient les sources d'un FSBL et d'une application pour exécuter la fonction `EverCrypt_HMAC_compute_sha2_256()` de HACL⋆ sur le Zynq-7000. La compilation de cette fonction est réalisée avec CompCert et celle du reste du projet avec gcc. Un script gdb permet d'instrumenter openOCD pour la programmation. Un Makefile permet d'automatiser toutes ces tâches.

3.1.2 Implémentation du matériel

Sur le système qui exécute la fonction d'attestation, tout comme VRASED [36], nous devons garantir que l'adversaire n'accède pas (directement ou indirectement) au secret et que l'exécution de la fonction d'attestation n'est jamais interrompue. À ces fins, nous proposons donc l'ajout d'un périphérique de confiance au microprocesseur. Son matériel est décrit en langage *Verilog* au niveau RTL. Il est synthétisé puis implémenté dans la partie FPGA du *SoC*, et vérifie des propriétés de sécurité que nous formalisons.

Le modèle sur lequel nous vérifions les propriétés de sécurité est un automate, décrit en langage SMV et généré par *Verilog2SMV*. La conversion est réalisée à partir de l'arbre syntaxique représentant le circuit synthétisé par *yosys* [47, 65]. Nous émettons l'hypothèse de transformation **T2** que *Verilog2SMV* préserve la sémantique du circuit lors de la génération de l'automate. Ainsi, si nous vérifions formellement des propriétés de sécurité sur l'automate décrit par le modèle SMV, alors ces propriétés sont également vérifiées par le circuit résultant de la synthèse.

Notons qu'à notre connaissance, il n'existe pas de preuve, aujourd'hui, que *Verilog2SMV* préserve la sémantique du circuit. Cependant, l'objectif de l'outil est de permettre une vérification formelle de circuits électroniques numériques [65]. C'est la raison pour laquelle nous considérons notre hypothèse de transformation **T2** réaliste.

Nous proposons donc de réaliser la synthèse du circuit avec *yosys*, d'extraire une description du circuit synthétisé dans un format interchangeable et de l'importer dans Xilinx Vivado pour le *place-and-route* et la génération du *bitstream*. Nous émettons l'hypothèse de transformation **T3** que Vivado ne modifie pas la sémantique du circuit lors de ces étapes et que les propriétés de sécurité vérifiées sont préservées. Ceci est une faiblesse, car la préservation sémantique n'est pas formellement garantie par Xilinx. À notre connaissance, il n'existe pas, aujourd'hui, d'outil de *place-and-route* formellement vérifié qui cible les FPGA Artix 7. Cependant, des outils *open-source* existent [66] et un projet de support des FPGA Artix 7 est en cours [67], ouvrant la voie à une future possible vérification formelle. Comme nous choisissons de vérifier notre circuit au niveau du résultat de la synthèse, si des travaux futurs (ou des travaux existants dont nous n'avons pas connaissance) garantissent la préservation sémantique, notre périphérique de confiance peut d'ores et déjà être réutilisé.

Le schéma représenté sur la figure 3.2 résume l'utilisation de l'infrastructure logicielle qui permet de préserver les propriétés formellement vérifiées sur le matériel jusqu'à la création d'un *bitstream* à implémenter sur le FPGA.

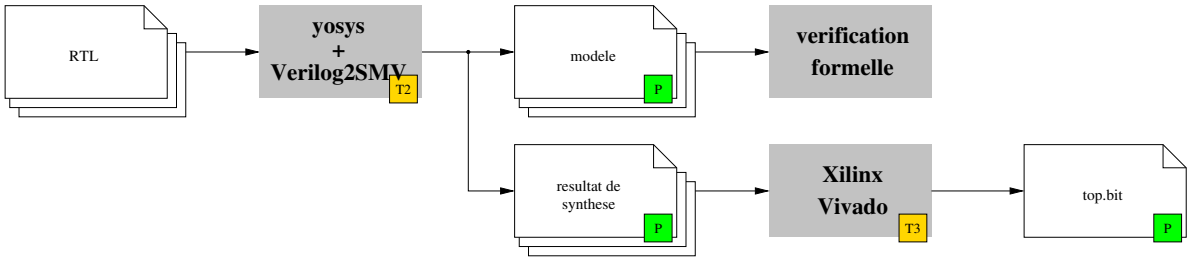

FIGURE 3.2 – Synthèse d'un projet en préservant les propriétés de sécurité

Une étape de vérification formelle de propriétés **P** est réalisée sur le modèle généré par *Verilog2SMV*. L'hypothèse de transformation **T2** garantit que ce modèle préserve la sémantique du circuit synthétisé par *yosys*. De ce fait, les propriétés vérifiées par le modèle sont également vérifiées par le résultat de la synthèse. L'hypothèse de transformation **T3** garantit alors que le *place-and-route* et la génération du *bitstream* par Vivado préservent la sémantique circuit. Ainsi, le circuit décrit par le *bitstream* servant à programmer le FPGA vérifie lui aussi les propriétés **P**.

Ressources : le dossier `3_1_2_yosys/` contient les sources d'un circuit décrit en *Verilog* au niveau RTL et des contraintes de placement et routage. Il est possible de synthétiser ce circuit avec yosys, réaliser le place-and-route et la génération du *bitstream* avec Vivado et programmer le FPGA du Zynq-7000. Un script gdb permet d'instrumenter openOCD pour la programmation. Un Makefile permet d'automatiser toutes ces tâches.

3.2 Modélisation

Afin de vérifier nos propriétés de sécurité sur notre implémentation, nous procédons à une modélisation du système. Les modèles de circuits synthétisés par *yosys* et construits par *Verilog2SMV* décrivent des **automates de Büchi déterministes**. Nous modélisons le comportement du microprocesseur par des propriétés exprimées en **logique temporelle PSL** (*Property Specification Language*), que nous traduisons depuis la documentation et considérons axiomatiques. Les propriétés vérifiées sur le modèle du périphérique de confiance sont elles aussi exprimés en logique temporelle PSL, tout comme la *soundness* et la sécurité permettant d'établir notre racine de confiance statique.

Dans cette section, nous fournissons des rappels techniques relatifs à la modélisation d'automates de Büchi déterministes et de propriétés de logique temporelle PSL. Nous donnons un aperçu des opérateurs de logique temporelle que nous utilisons.

3.2.1 Automates de Büchi déterministes

Afin de modéliser les circuits électroniques numériques, *Verilog2SMV* s'appuie sur la théorie des automates et plus particulièrement des automates de Büchi déterministes [68]. Un

automate de Büchi déterministe est une construction mathématique abstraite, susceptible d'être dans un nombre fini d'états, mais étant à chaque instant dans un seul état à la fois. Le changement d'état est contraint par une fonction de transition. La notion d'instant est donc présente : l'automate peut changer d'état au cours d'une suite d'instants comme décrit par la fonction de transition. L'état dans lequel se trouve l'automate à un instant donné est appelé *état courant*. Un changement successif d'état au cours de différents instants est appelé une *exécution* de l'automate. La représentation d'une exécution de l'automate sous forme d'une suite d'états parcourus est appelée la *trace d'exécution*.

La section A.2 de l'annexe A propose des rappels techniques sur la théorie des automates. Plus particulièrement, la section A.2.1 fournit une définition des automates finis déterministes et une description des alphabets, des différents types d'états et des fonctions de transition.

Un automate de Büchi est un automate fini, opérant sur des mots infinis, avec un ensemble d'états finaux. Une trace d'exécution est acceptée par un automate si et seulement si elle passe infiniment souvent par au moins un état final. Un automate de Büchi déterministe se définit sous la forme d'un n-uplet $A = \langle \Sigma, \mathcal{Q}, \Delta, I, \mathcal{F} \rangle$, où :

— Σ est un alphabet.
— $\mathcal{Q}$ est un ensemble fini d'états.
— $I \subseteq \mathcal{Q}$ est un ensemble d'états initiaux.
— $\Delta : \mathcal{Q} \times \Sigma \to \mathcal{Q}$ est une fonction de transition.
— $\mathcal{F} \subseteq \mathcal{Q}$ est un ensemble d'états finaux.

Cette définition est utilisée par les outils de *model-checking* lors de la vérification.

3.2.2 Logiques temporelles LTL, PSL

La logique temporelle permet de définir des formules, ou propriétés, sur l'avenir de prédicats dans un système à transitions. La logique temporelle est utilisée en *model-checking* pour décrire les propriétés que doit vérifier un automate [69]. La définition d'une propriété temporelle est en quelque sorte une formalisation de la spécification, affranchie des ambigüités inhérentes au langage naturel. Il existe plusieurs formalisations de logique temporelle, prenant en compte diverses modalités de base. Les plus connues sont les logiques CTL (*Computation Tree Logic*) [25] et LTL [27].

Une autre logique temporelle, que nous avons choisie pour formaliser nos spécifications, est le langage PSL, standardisé par IEEE en tant que standard 1850-2010 [70]. PSL est basé sur deux styles complémentaires : LTL, en quoi il peut aussi être converti [71], et SERE (pour *Sequential Extended Regular Expressions*), une variante des expressions rationnelles. L'avantage majeur de PSL par rapport à LTL est qu'il fournit des opérateurs temporels étendus, simplifiant l'écriture de propriétés temporelles complexes [72]. Il a également été conçu pour être simple à lire : les mots clefs décrivant les opérateurs temporels ont été empruntés au langage naturel (en langue anglaise).

En plus des opérateurs de la logique du premier ordre, tels que la conjonction ($\wedge$), disjonction ($\vee$), négation ($\neg$) et implication ($\rightarrow$), PSL fournit des opérateurs temporels et quantificateurs. Les opérateurs temporels suivants trouvent leur équivalence en LTL, ce sont les opérateurs temporels de base. Ils sont utilisés pour décrire spécifications et propriétés locales que doit respecter notre système :

— **always**(ϕ) : est vérifiée si la propriété ϕ est vérifiée pour tous les états futurs.
— **never**(ϕ) : est équivalent à **always**($\neg\phi$).

— **next**(ϕ) : est vérifiée si la propriété ϕ est vérifiée à l'état suivant du système.
— **eventually**(ϕ) : est vérifiée s'il existe un état futur où ϕ est vérifiée.
— (ψ) **until** (ϕ) : est vérifiée s'il existe un état futur où ϕ est vérifiée et que ψ est vérifiée à tous les états avant celui-ci.

Les opérateurs temporels de base sont accompagnés d'opérateurs temporels étendus. Ceux-ci peuvent être vus comme du sucre syntaxique à la LTL. Ces opérateurs sont utiles à la définition de nos spécifications et propriétés locales :

— **next_event_a**(ψ)[*range*](ϕ) : est vérifiée si ϕ est vérifiée à tous les états suivants où ψ est vérifiée, dans l'intervalle défini par *range*, où un intervalle est un ensemble de nombre entiers positifs consécutifs.
— **next_event**(ψ)[i](ϕ), où i est un nombre entier positif : est un raccourci pour **next_event_a**(ψ)[$i : i$](ϕ).

Le dernier opérateur d'intérêt pour décrire nos propriétés est le quantificateur **forall** :

— **forall** i **in** $\{range\}$: $\phi(i)$: est vérifiée si la propriété paramétrée $\phi(i)$ est vérifiée pour toutes les valeurs de i prises dans l'intervalle défini par *range*. Cette expression est équivalente à une conjonction des propriétés paramétrées. Par exemple :
forall i **in** $\{0 : 2\} : \phi(i) \equiv \phi(0) \wedge \phi(1) \wedge \phi(2)$

En annexe A, la section A.2.2 décrit les associations entre propriétés de logique temporelle et automates finis déterministes. Ces associations sont utilisées par les outils de *model-checking* pour la vérification de propriétés temporelles sur des automates.

3.3 Vérification

Après la modélisation du système et des propriétés à vérifier, nous procédons à une vérification formelle. Nous utilisons des outils de ***model-checking*** pour obtenir la garantie qu'une propriété de logique temporelle est vérifiée par un automate. À partir d'axiomes ou de propriétés locales, nous établissons une **preuve** d'un théorème qui exprime la sécurité de l'exécution de la fonction d'attestation. Nous établissons ainsi notre racine de confiance statique, et garantissons par extension la sécurité de l'attestation à distance. Le *model-checking* et la preuve peuvent être séparés en plusieurs étapes mais la méthodologie reste identique pour chaque étape.

Dans cette section, nous introduisons les outils de *model-checking* que nous utilisons et présentons les fonctionnalités utiles à la vérification de notre implémentation. Nous introduisons également les méthodes de preuve que nous utilisons.

3.3.1 Model-checking

Le *model-checking* est une méthode de vérification formelle de propriétés temporelles. Elle est exhaustive et automatique ; elle se base sur les principes de la théorie des automates [73]. Dans le cas où une propriété n'est pas vérifiée par un automate, un contre-exemple est fourni, illustrant la violation. En annexe A, la section A.2.3.1 détaille les concepts de vérification par *model-checking* et de dépliage des automates.

Dans cette thèse, nous utilisons le model-checker *NuSMV* [74, 75] pour la vérification de nos propriétés locales. *NuSMV* est un model-checker libre qui implémente le *model-checking symbolique* par *diagrammes de décision binaires* (BDD : de l'anglais *Binary Decision Diagrams*). Les automates de Büchi sont décrits en langage SMV et *NuSMV* supporte la vérification de

propriétés de logique temporelle décrites en PSL [75]. Également, *NuSMV* supporte la vérification d'invariants qui ne nécessite pas de dépliage et améliore grandement les performances [75]. Un invariant est une propriété sans opérateur temporel à l'exception de l'opérateur **next**. Vérifier un invariant revient à vérifier une propriété qui est toujours vraie, soit vérifier une propriété contenue dans un opérateur PSL **always**.

Pour la manipulation de propriétés temporelles, tout comme VRASED [36], nous utilisons *Spot* [48], une bibliothèque C++ dont les fonctionnalités peuvent être appelées depuis un interpréteur Python. *Spot* a pour objectif principal la manipulation des propriétés temporelles et des automates : elle contient des algorithmes pour réaliser indépendamment les étapes du *model-checking*. Cela inclut la conversion de propriétés temporelles en automates, mais aussi la déterminisation d'automates et le produit [48]. Des règles de sémantique et d'équivalence permettent une réécriture des propriétés temporelles et, par extension un filtrage des automates. En annexe A, la section A.2.3.1 détaille les concepts de déterminisation d'automates et de produit. La section A.2.3.2 détaille la vérification de propriétés locales sur un automate par *model-checking* avec *Spot*.

Ressources : le modèle SMV `a_2_3_model_checking/nusmv/main.smv` contient la définition de l'automate représenté sur la figure A.5 et des propriétés ϕ et ψ de l'expression A.2. Le `Makefile` permet d'automatiser l'exécution de *NuSMV* et d'observer un contre-exemple qui viole la propriété ψ.

Ressources : le script Python `a_2_3_model_checking/print/mc.py` utilise Spot, il permet d'obtenir les automates produits représentés sur la figure A.7 et de les exporter au format *dot*. Le `Makefile` permet d'automatiser son exécution et une conversion au format *pdf*.

3.3.2 Preuve

En théorie la seule limitation du *model-checking* est que l'automate à vérifier doit être fini, ce qui est toujours le cas dans cette thèse, car nous manipulons des modèles de circuit électroniques numériques. En pratique, cependant, la limitation majeure du *model-checking* est le *problème d'explosion combinatoire du nombre d'états du système*, où la mémoire nécessaire pour modéliser l'automate de manière exhaustive excède la mémoire physique de la machine utilisée pour le *model-checking*.

Dans ce cas, une stratégie de preuve peut être envisagée. Nous travaillons alors sur les propriétés temporelles et non sur l'automate décrivant le système. Un prérequis est seulement de vérifier par *model-checking* des propriétés locales. Ces mêmes propriétés locales représentent alors les hypothèses selon lesquelles la spécification peut être prouvée.

Vérifier indépendamment différentes propriétés locales sur le même automate revient à vérifier la conjonction de celles-ci [76]. Les propriétés locales doivent donc être astucieusement choisies pour que leur conjonction soit suffisante pour prouver la spécification. La preuve de la spécification par la conjonction des propriétés locale est appelée établissement de *théorème*, ou de *lemme*. Un lemme est un résultat sur lequel s'appuie la démonstration d'un théorème plus important.

Dans cette thèse, nous envisageons deux possibilités pour établir un théorème :

1. soit en utilisant une méthode automatique de filtrage de propriété temporelle ;
2. soit en effectuant manuellement une réécriture de l'implication.

La première solution requiert l'utilisation des fonctions de filtrage du *model-checker* pour réduire l'implication. Le filtrage est une minimisation d'un automate construit depuis la propriété temporelle. Cette méthode a l'avantage d'être entièrement automatique : elle est basée sur l'utilisation d'un solveur SAT. L'inconvénient est qu'elle requiert un dépliage de l'automate résultant et se heurte également à l'explosion combinatoire.

La seconde solution requiert l'utilisation d'un assistant de preuves et, potentiellement, d'une bibliothèque définissant la sémantique de la logique temporelle. Cette méthode à l'avantage de ne pas être dépendante de la mémoire de la machine sur laquelle le théorème est démontré. Elle est également très rapide à vérifier : la puissance de calcul requise pour jouer la preuve est minime. En contrepartie, l'inconvénient de cette méthode est qu'elle requiert une connaissance en logique et une maîtrise de l'assistant de preuves : l'écriture de preuves formelles est un travail fastidieux.

Pour obtenir une preuve automatique, comme VRASED [36], nous instrumentons les fonctions de filtrage de Spot [48]. Spot implémente deux procédures différentes de minimisation basées sur l'utilisation de solveur SAT : une procédure qui crée un automate de Büchi déterministe et une procédure qui crée un ω-automate fini déterministe (dont la condition d'acceptation est différente mais que nous ne décrivons pas ici). Les deux procédures sont basées sont sur le solveur SAT *PicoSAT 965* [77] mais il est également possible d'utiliser un solveur SAT externe. Les détails des différentes procédures sont fournis dans une publication dédiée [78].

Dans notre cas, nous souhaitons prouver une spécification depuis des propriétés locales. L'automate filtré doit donc résulter d'une tautologie et ne posséder qu'un seul état : le type d'automate créé n'importe donc pas sur le résultat et nous pouvons utiliser les deux procédures de minimisation sans restriction. Spot fournit également un algorithme de filtrage automatique qui fait appel à la procédure de filtrage la plus appropriée. Cette solution est décrite dans une seconde publication [79] et est parfaitement adaptée à notre application. La documentation de Spot fournit des détails techniques sur la minimisation d'automates [2] ; le terme *formule* est employé pour désigner une propriété temporelle.

Pour automatiser notre preuve, nous pouvons créer un script Python dans lequel nous écrivons une formule : une implication de la spécification par la conjonction des propriétés locales. Ensuite, nous soumettons cette formule à l'algorithme de minimisation de Spot. Plusieurs cas de figure sont possibles :

1. si l'algorithme permet de réduire l'implication à une tautologie, alors ceci prouve que notre spécification est également vérifiée.
2. si l'algorithme s'arrête après une réduction de l'implication à une expression formée de prédicats et d'opérateurs temporels, alors deux cas de figure sont possibles :
   — soit les propriétés locales ne suffisent pas à garantir la spécification ;
   — soit l'algorithme de minimisation d'automates de Spot ne parvient pas à trouver une solution qui réduit l'implication à une tautologie.

   Dans les deux cas de figure, nous ne pouvons pas conclure que les propriétés locales suffisent à garantir la spécification.

2. En particulier la section *SAT-based Minimization of Deterministic ω-Automata*, accessible à l'adresse `https://spot.lrde.epita.fr/satmin.html`.

3. si l'algorithme utilise toute la mémoire de la machine et ne trouve pas de solution avant cela, alors nous faisons face à une explosion combinatoire : nous ne pouvons pas conclure au sujet de l'implication.

En annexe A, la section A.2.4.1 détaille un exemple de preuve automatique de spécifications à partir de propriétés locales. La preuve de théorème automatique est réalisée à l'aide des fonctionnalités de filtrage de *Spot*.

Ressources : le script Python `a_2_4_preuve/spot/minimize.py` utilise Spot, il permet de reproduire l'expérience et obtenir les résultats décrits par l'expression A.4.
Dans ce même script, un exemple de propriété provoquant une explosion combinatoire est proposé : il illustre le cas où Spot ne trouve pas de solution et où nous ne pouvons pas conclure. Le `Makefile` permet d'automatiser son exécution.

Si nous souhaitons éviter de nous heurter à l'explosion combinatoire lors de la preuve d'un théorème, nous pouvons avoir recours à une preuve manuelle à l'aide d'un assistant de preuves. Dans cette thèse, nous avons choisi l'assistant de preuves Coq pour démontrer nos théorèmes.

Coq permet l'expression de spécifications et le développement de programmes cohérents avec celles-ci [80]. Nous n'utilisons pas cet assistant pour développer des programmes mais seulement pour prouver notre spécification. Des propriétés temporelles, que nous vérifions au préalable par *model-checking*, forment nos hypothèses. À partir de ces hypothèses et d'une implémentation des règles sémantiques de la logique temporelle, nous prouvons un théorème qui garantit la vérification de notre spécification.

La bibliothèque `LTL_Coq`, développée par Cenobio Moisés Vázquez Reyes, propose une implémentation des règles sémantiques de la logique temporelle LTL. Cette bibliothèque est distribuée librement et est accessible sur le site web GitHub [3]. Elle contient entre autres les définitions d'un état, d'une exécution, d'une formule temporelle, ainsi que les preuves de différents théorèmes. Parmi les théorèmes fournis par cette bibliothèque, nous pouvons trouver l'application de la négation sur une formule quelconque et les règles de décomposition des opérateurs temporels. La distributivité des opérateurs **next**, **always**, **eventually** au travers de la conjonction et de la disjonction est également démontrée.

Les théorèmes fournis avec cette bibliothèque sont utiles à nos démonstrations mais ils ne sont pas toujours suffisants pour prouver nos théorèmes. Aussi, nous démontrons parfois des lemmes à partir des définitions et de l'implémentation des règles sémantiques de la LTL. L'avantage d'une telle approche est que ces lemmes peuvent être prouvés à partir de formules LTL quelconques et pas seulement à partir de prédicats. Ainsi, nous pouvons réutiliser nos lemmes pour prouver plusieurs théorèmes, mêmes si ceux-ci s'appliquent à des spécifications et des propriétés locales différentes.

En annexe A, la section A.2.4.2 détaille un exemple de preuve manuelle de spécifications à partir de propriétés locales. La preuve de théorème est réalisée à l'aide de Coq et les règles sémantiques de la LTL sont définies par la bibliothèque `LTL_Coq`.

3. `https://github.com/spidermoy/LTL_Coq`

Ressources : le programme `a_2_4_preuve/coq/proof.v` utilise Coq, il permet de reproduire l'expérience et obtenir la preuve de ϕ depuis les propriétés ϕ_1 et ϕ_2 de l'expression A.3 fournie en annexe A. Lors de cette preuve, les termes a et b font référence à des formules temporelles quelconques.
Dans ce même programme, deux lemmes sont également prouvés, nécessaires à l'obtention de la preuve de ϕ de l'expression A.3, dont l'involution de la négation sur une propriété temporelle quelconque. Le `Makefile` permet d'automatiser sa compilation et son exécution.

3.4 Conclusion et perspectives

Dans ce chapitre, nous avons décrit la stratégie de vérification formelle que nous envisageons. Nous avons décrit les infrastructures logicielles nécessaire à la préservation des propriétés de sécurités vérifiées jusqu'à l'implémentation finale. Pour la modélisation, nous avons choisi une approche ascendante : c'est-à-dire que nous développons du matériel dans un langage HDL, dont nous réalisons une abstraction pour obtenir un modèle haut niveau. Pour la vérification, nous avons envisagé du *model-checking* et une stratégie de preuve : ceci permettra de réduire la taille des propriétés locales.

Néanmoins, lors de la vérification de systèmes complexes, une explosion combinatoire peut être provoquée par la taille du modèle, c'est-à-dire de l'automate sur lequel nous vérifions ces propriétés locales. Dans ce cas, nous parlons du problème de passage à l'échelle de la méthode de vérification. Nous faisons face à ce problème lors de la vérification des spécifications que nous formalisons pour notre périphérique de confiance. Les méthodes que nous décrivons dans ce chapitre ne permettent pas de réaliser du *model-checking* sur celui-ci : nos modèles sont en effet trop conséquents.

Pour réduire cet impact, il est possible d'utiliser des modèles abstraits lors de la vérification. Des solutions de décomposition existent dans l'état de l'art et permettent de réduire la taille de l'automate à vérifier pour chaque propriété locale. Dans cette thèse, nous proposons une nouvelle solution d'automatisation de la vérification formelle des systèmes matériels. Cette méthode est adaptée aux systèmes complexes, décrits en langage *Verilog* et dont l'architecture est modifiable, tels que ceux que l'on rencontre dans l'industrie. Notre solution est une approche adaptable à n'importe quelle suite de logiciels de conception assistée par ordinateur (ou EDA, pour *Electronic Design Automation*, en anglais) et de vérification formelle. Nous fournissons une implémentation qui fonctionne entièrement avec des logiciels libres. Cette contribution est décrite dans le chapitre 4.

Chapitre 4

Méthode de vérification formelle automatisée de systèmes matériels

Sommaire

Dans ce chapitre, nous décrivons la première contribution de notre travail de recherche, dont l'objectif est d'étendre l'attestation à distance vérifiée formellement aux microprocesseurs modernes pris sur étagère. Cette contribution consiste à proposer une méthode, automatique et industrialisable, de vérification formelle de systèmes matériels complexes. Celle-ci inclut la construction de modèles abstraits corrects et est accompagnée d'un ensemble d'outils associés basé sur des logiciels libres.

En effet, assurer la sécurité de l'attestation à distance nécessite l'établissement d'une racine de confiance statique. Pour cela, nous proposons de concevoir un périphérique de confiance et vérifier formellement la sécurité que celui-ci garantit. La description de ce périphérique fera l'objet du chapitre 5. Or, nous ciblons les microprocesseurs pris sur étagère, que nous ne pouvons pas modifier : ceci implique que nous ne pouvons pas nous limiter aux systèmes simples. En effet, lors de la vérification de notre périphérique de confiance, le *model-checking* se heurte à une explosion combinatoire. Pour passer à l'échelle, nous devons donc réaliser un partitionnement du système et envisager une stratégie de preuve. Également, nous devons

réaliser des abstractions pour chacune des partitions. Cette étape permet de construire un modèle abstrait, dédiée à la vérification d'une propriété locale : c'est donc un système plus petit qui possède localement le même comportement. Construire un modèle abstrait approprié est un problème non-trivial :

— il est nécessaire de réduire suffisamment le système pour que le *model-checking* puisse être réalisé en un temps acceptable ;
— et il est nécessaire que l'abstraction soit correcte pour que la propriété temporelle qu'elle vérifie soit également vérifiée par le système concret.

Ce chapitre aborde ces problèmes et propose des solutions et outils pour y faire face. Ces travaux ont fait l'objet d'une publication [81] et d'une présentation à la conférence *Embedded Real Time Systems* (ERTS) 2022.

La section 4.1 décrit le contexte industriel vis-à-vis de la vérification formelle de systèmes matériels. La section 4.2 présente notre approche avec ses dépendances logicielles associées. La section 4.3 présente un cas d'étude sur lequel nous avons mis en œuvre notre méthode et la section 4.4 détaille cette mise en œuvre. Finalement, la section 4.5 présente les limitations de cette approche et propose différentes perspectives et solutions que nous appliquons dans le reste de ce manuscrit.

Dans une optique de rejeu des expériences illustrant nos travaux, les détails techniques au sujet des outils associés à la méthode sont fournis en annexe B.

4.1 Contexte industriel

Le contexte industriel actuel est tel que la vérification formelle de systèmes matériels a été adoptée dans les domaines de conception des systèmes critiques et/ou sécurisés [82, 36], mais pas encore à grande échelle. La vérification est généralement réalisée au travers de simulations de circuits et de tests. Des vecteurs d'entrée spécifiques sont choisis pour être fournis au modèle (dans le cas des simulations) ou au système lui-même (dans le cas des tests) et les vecteurs de sortie sont observés afin de déterminer si le comportement est correct. En ce sens, des outils de vérification proposent des solutions pour observer les vecteurs d'entrée et de sortie sous forme de chronogrammes, par exemple à l'aide du format *Value Change Dump* (VCD) défini dans la norme du langage *Verilog* [46]. Également, nous trouvons des outils de synthèse et d'implémentation, permettant de programmer un circuit sur FPGA et procéder à une phase de test. Nous pouvons citer, dans ce domaine, les outils de la suite *Vivado* [57]. Cependant, cette approche traditionnelle est limitée, car son objectif est seulement d'identifier l'origine de défauts potentiels et non pas de rigoureusement démontrer leur absence.

Néanmoins, des outils de vérification formelle existent et font partie des suites logicielles d'EDA majoritairement utilisées dans l'industrie. Nous pouvons par exemple citer la suite *Jasper Design Automation*, développée par *Cadence*, la suite *Questa* ou *OneSpin*, développée par *Siemens* ou encore la suite *VC Formal*, développée par *Synopsys* [83]. Toutes ces solutions proposent la possibilité de vérification formelle de systèmes matériels vis-à-vis de propriétés exprimées en langage PSL ou sous forme d'assertions *SystemVerilog*. Également, ces outils possèdent généralement une interface de programmation, qui permet de les instrumenter.

La raison majeure pour laquelle la vérification formelle n'est pas encore adoptée à grande échelle est le problème d'explosion combinatoire. En effet, les systèmes industriels modernes, bien que partitionnés lors de la phase d'étude architecture [84], sont composés de partitions

présentant de nombreuses mémoires et dont le nombre d'états nécessaires pour les modéliser est trop important. Les travaux à l'état de l'art proposent des solutions pour abstraire de tels systèmes : si l'objectif est seulement de construire un modèle abstrait, fournir la description RTL est suffisant pour la plupart des méthodes [42, 43, 44].

Malheureusement, une autre raison pour laquelle la vérification formelle n'est pas encore adoptée à grande échelle est la complexité des spécifications. Par "complexité des spécifications", nous sous-entendons deux aspects :

— Tout d'abord, **le langage de formalisation utilisé n'est pas toujours suffisamment expressif pour que l'expression formelle soit rendue accessible**.
— Ensuite, **l'automate construit à partir de la spécification, à lui seul, suffit à heurter la vérification au problème d'explosion combinatoire**.

Dans ce cas, il est nécessaire d'envisager une stratégie de preuve et de ne vérifier que des propriétés locales sur un modèle abstrait.

La problématique lors de l'utilisation des solutions proposées par les travaux à l'état de l'art est que l'écriture des propriétés locales, cohérentes à la fois vis-à-vis de la preuve de spécification et de la construction du modèle abstrait, est laissée à l'expertise du concepteur. En particulier, dans le cas d'une extraction des prédicats par un découpage intelligent des vecteurs [44], de nouveaux prédicats sont ajoutés aux modèles abstraits et la preuve de la spécification doit être réalisée à partir de ceux-ci. Une approche que nous pensons plus appropriée, et que nous utilisons considérablement dans cette thèse, est d'**écrire les propriétés locales seulement de manière cohérente vis-à-vis de la preuve, puis construire le modèle abstrait en fonction du résultat de cette écriture**. Également, nous pensons que, **pour qu'une stratégie de preuve soit envisagée à grande échelle dans l'industrie, celle-ci doit être automatique**, par exemple en se basant sur les algorithmes de filtrage des outils de *model-checking*.

Afin d'apporter une réponse à cette problématique, nous introduisons dans ce chapitre une approche de vérification, couplée à des outils d'automatisation basés sur des logiciels libres. Notre solution présente deux contributions majeures :

1. une traduction automatique des propriétés temporelles, écrites en langage PSL, vers un langage dédié à une stratégie de preuve automatique. Notre outil supporte également l'abstraction de fonctions non interprétées, si besoin, pour soulager les procédures de minimisation basées sur l'utilisation de solveurs SAT ;

2. une suite logicielle qui construit, de manière automatique, des modèles abstraits d'un système matériel adaptés à la preuve d'un théorème. La correction de l'abstraction est certifiée et l'outil de *model-checking* est automatiquement sollicité. Les résultats de la vérification sont centralisés et les contre-exemples sont présentés sous forme de chronogrammes.

L'ensemble des outils que nous proposons est distribué sous licence libre et est rendu accessible publiquement [1]. Le défi pour cette solution est de garantir une approche qui réduit suffisamment le système et les propriétés locales, pour que le *model-checking* et la preuve de théorème soient réalisables en un temps acceptable. Nous considérons comme cas d'étude la vérification d'un décompresseur de traces *CoreSight*, dont la spécification est imposée par le format des traces tel que décrit dans la documentation [59].

4.2 Méthode de vérification

Dans l'approche que nous proposons, les spécifications sont formellement exprimées en langage PSL. Les propriétés locales, vérifiées lors de *model-checking*, sont astucieusement choisies pour que leur conjonction soit suffisante pour prouver la spécification. Toutes les propriétés PSL sont converties en langage LTL par le model-checker *NuSMV*, que nous considérons correct. Pour cette tâche, il est également possible de se baser sur des travaux qui proposent une traduction prouvée pour préserver la sémantique du PSL [71]. Ensuite, nous tirons profit des algorithmes de minimisation de *Spot* pour prouver la spécification à partir de la conjonction des propriétés locales.

Le circuit est décrit en langage *Verilog*, il est synthétisé par *yosys* et traduit par *Verilog2SMV* en automate de Büchi, dont le modèle est décrit en langage SMV. Pour pallier le problème de l'explosion combinatoire, nous proposons une méthode pour automatiquement construire des modèles abstraits, pour chaque propriété locale, basée sur une réduction par localisation. Une relation de simulation par inclusion de langage est établie entre chaque modèle abstrait et le modèle concret : ceci apporte, pour chaque modèle, un certificat que l'abstraction est correcte et que les propriétés locales sont préservées. Finalement, les propriétés locales sont vérifiées par le model-checker *NuSMV*.

Notre approche diffère des précédents travaux dans le sens où nous proposons une méthode de réduction par localisation, dont l'abstraction est déduite depuis les propriétés locales. Nous pensons que décrire des propriétés locales avec un objectif de preuve de théorème est la clé pour prouver des spécifications complexes, même si cela implique une réduction non optimale lors de l'abstraction. De plus, le pilier de notre solution est un algorithme de synthèse de circuits numériques : les modèles abstraits générés sont donc le résultat d'une réduction locale proche de l'implémentation finale du système.

Différentes suites de logiciels EDA sont disponibles dans l'industrie, dont la plupart proposent des solutions de synthèse et de vérification formelle. L'approche que nous proposons est agnostique en matière de choix de suite EDA : elle peut être implémentée à partir de n'importe quelle suite à partir du moment où nous pouvons instrumenter les outils avec un haut degré de granularité. En ce sens, l'implémentation que nous proposons ici est basée uniquement sur des logiciels libres.

Le schéma représenté sur la figure 4.1 représente notre approche de vérification formelle automatisée, qui se décompose en trois étapes majeures :

— Étape 1 : contrairement aux précédents travaux [42, 43, 44], nous commençons par un traitement de la spécification. Nous construisons des propriétés locales en prenant les précautions nécessaires pour que leur conjonction suffise à prouver la spécification.
— Étape 2 : une fois que nous sommes en possession de propriétés locales, nous prouvons la spécification depuis leur conjonction. Pour cela, nous utilisons les algorithmes de manipulation de formules LTL et de minimisation d'automates fournis par *Spot*.
— Étape 3 : finalement, nous pouvons automatiquement abstraire le système à partir des propriétés locales. L'objectif est que nous ayons choisi nos propriétés locales telles que la réduction par localisation construise un modèle suffisamment abstrait pour que la vérification soit conduite en un temps raisonnable. Le model checker *NuSMV* est utilisé pour la vérification des propriétés locales.

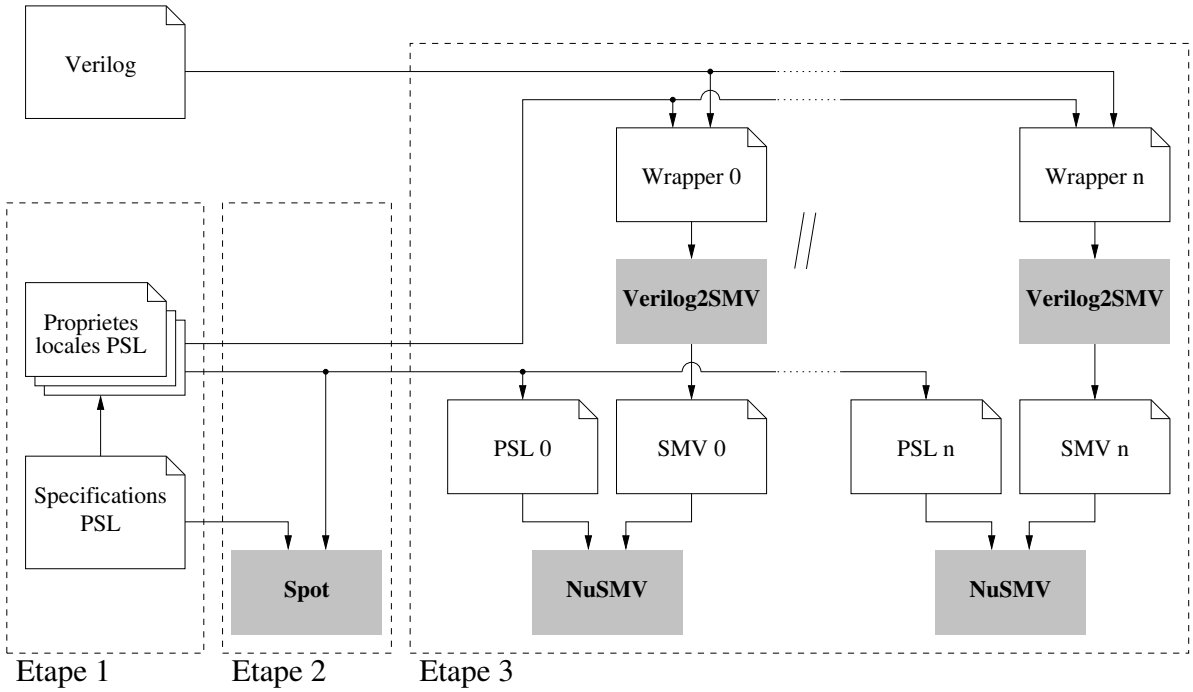

FIGURE 4.1 – Approche de vérification formelle automatisée

4.2.1 Étape 1 : expression des propriétés locales

Notre méthode nous permet de tirer profit des opérateurs temporels étendus fournis par le langage PSL. Ce sous-ensemble du langage PSL facilite grandement l'écriture des spécifications et la décomposition en propriétés locales. Ce même sous-ensemble est supporté par *NuSMV* pour l'écriture des propriétés et le *model-checking* [75]. Une description des opérateurs, que nous considérons utiles à l'écriture des spécifications et des propriétés locales, est fournie dans le chapitre 3, dans la section 3.2.2.

Lors de l'expression des propriétés locales, nous rappelons que l'objectif est de s'assurer que leur conjonction est suffisante pour prouver la spécification. Même si l'utilisation du langage PSL simplifie cette tâche par rapport à un langage moins expressif (comme le langage LTL), une expertise en logique est néanmoins requise. Nous procédons donc à une décomposition manuelle des spécifications et ne nous attardons pas sur de potentielles solutions automatiques. Ceci est la seule étape manuelle de notre approche : les autres étapes sont automatisées, y compris celle de la preuve de la spécification. Un exemple de décomposition manuelle des spécifications est donné dans la section 4.4.1.

4.2.2 Étape 2 : stratégie de preuve

La preuve des spécifications est automatiquement établie à l'aide des fonctionnalités de filtrage de *Spot*. Pour permettre cette automatisation, nous avons instrumenté les fonctionnalités d'analyse grammaticale (*parsing*) et de traduction de *NuSMV* dans le but de convertir les propriétés PSL (langage qui n'est pas entièrement supporté par *Spot*) en LTL. Cette instrumentation a été réalisée à l'aide de *pyNUSMV* [85]. *pyNUSMV* est une bibliothèque *Python*

dédiée au prototypage et à l'expérimentation à partir des algorithmes de *model-checking* de *NuSMV*. Notre instrumentation permet d'exprimer les opérateurs temporels étendus seulement à partir d'opérateurs temporels de base, dans un langage dont la grammaire est reconnaissable par *Spot*. Pour illustrer cette traduction, nous pouvons par exemple citer l'opérateur **next_event**, qui peut être exprimé seulement en utilisant les opérateurs **next**, **always** et **until**, comme montré ci-après :

$$\textbf{next\_event}(\psi)[1](\phi) \equiv (\neg\psi \ \ \textbf{until} \ \ (\psi \wedge \phi)) \vee \textbf{always}\,(\neg\psi)$$

Cette fonctionnalité permet de traduire toute expression contenant des opérateurs temporels étendus supportée par *NuSMV*. Il n'est donc pas nécessaire de saisir les subtilités de la sémantique du langage PSL pour obtenir une preuve de théorème automatique. Pour mieux cerner le comportement d'une telle traduction, la section B.1 de l'annexe B décrit les équivalences entre PSL et LTL. Plus particulièrement, la section B.1.1 propose des exemples de définition pour les fonctions de traduction des opérateurs temporels étendus.

Une fois que nous sommes en possession de propriétés temporelles reconnaissables par *Spot*, nous construisons un script *Python* qui permet un filtrage. Le filtrage est réalisé sur une propriété construite depuis la conjonction des propriétés locales et d'une implication des spécifications. *Spot* est instrumenté pour produire un automate de Büchi déterministe qui reconnait le langage décrit par cette propriété et pour appliquer ses algorithmes de minimisation. Comme décrit dans la section 3.3.2, l'objectif est de réduire l'implication et montrer que c'est une tautologie. Le script *Python* que nous construisons contient déjà toute cette instrumentation et peut être exécuté sans modification.

Cette méthode est entièrement automatique, mais elle requiert un dépliage de l'automate résultant et se heurte à l'explosion combinatoire. Afin de réduire l'impact du dépliage, nous fournissons des abstractions de fonctions non interprétées lors de la construction de la propriété à filtrer. Ces abstractions sont axiomatiques et implicites, elles sont donc à considérer par le concepteur lors de la première étape (l'expression des propriétés locales). Parmi ces abstractions, nous pouvons citer :

— l'abstraction d'opérations logiques entre vecteurs par des prédicats. Par exemple, une égalité entre deux vecteurs est seulement considérée comme étant vraie ou fausse à un instant donné. Si une telle égalité est présente des deux côtés de l'implication, c'est-à-dire dans les propriétés locales et dans la spécification, nous pouvons donc réduire l'implication sans considérer la taille des deux vecteurs.
— l'abstraction des quantificateurs **forall**. Nous ne les remplaçons pas par une conjonction de propriétés paramétrées, mais considérons seulement la propriété quantifiée (c'est-à-dire avec l'indice du quantificateur). Cette abstraction réduit énormément la complexité de la minimisation, mais requiert pour le concepteur de s'assurer manuellement que toutes les valeurs de l'intervalle du quantificateur sont vérifiées par *model-checking*.

L'équivalence entre les expressions 4.1 et 4.2 donne un exemple d'axiome implicite. Dans cet exemple, la propriété représentée par l'expression 4.1 est vérifiée par *model-checking* ; la propriété représentée par l'expression 4.2 est celle qui est considérée par *Spot* lors du filtrage durant l'exécution du script *Python* que nous construisons.

$$\textbf{forall } i \textbf{ in } \{0:255\} : \textbf{always}[(in_1 = i) \rightarrow \textbf{eventually}(out = i)] \tag{4.1}$$

$$\iff$$

$$\textbf{always}[in_1\_equals\_i \rightarrow \textbf{eventually}(out\_equals\_i)] \tag{4.2}$$

Dans cet exemple, la conjonction de 256 propriétés est abstraite en une seule propriété où le prédicat $in_1\_equals\_i$ implique toujours que le prédicat out_equals_i sera éventuellement vérifié. Comme les tailles des vecteurs in_1 et *out* n'apparaissent pas dans les expressions, il est de la responsabilité du concepteur de s'assurer que l'intervale $\{0 : 255\}$ pour l'indice i couvre toutes les valeurs possibles pour ces vecteurs. La section B.1.2 de l'annexe B fournit des recommandations à suivre pour tirer profit au maximum des abstractions implicites. Dans le cas où ces abstractions ne sont pas suffisantes pour permetttre une preuve en un temps acceptable, une réécriture des propriétés locales lors de l'étape 1 est requise.

Notre outil de traduction automatique est implémenté sous forme d'un programme nommé *psl2spot*. Il est développé en *Python*, est distribué sous licence libre et est rendu accessible publiquement [1].

4.2.3 Étape 3 : réduction par localisation

Le système que nous souhaitons abstraire est un système matériel, un circuit, décrit en langage *Verilog* au niveau RTL. Afin de générer automatiquement un modèle abstrait, dédié à la vérification d'une propriété locale, nous commençons par énoncer une hypothèse que nous nommons **H0**. Nous formulons notre hypothèse comme suit :

H0 : le modèle concret est composé de multiples systèmes finis fonctionnant en parallèle. Lorsqu'une sortie du circuit n'est pas utilisée, l'étape d'optimisation de la synthèse sépare ces multiples systèmes, supprime les registres non utilisés et n'altère pas le reste du circuit.

Nous tirons profit de l'étape d'optimisation de la synthèse pour construire une réduction par localisation du modèle : nous déconnectons des sorties du circuit. Les variables locales du modèle qui n'ont pas d'impact sur les prédicats référencés dans la propriété à vérifier sont donc masquées. Au travers de l'hypothèse **H0**, nous considérons que les multiples systèmes, qui composent le modèle et sont utiles, ne sont pas altérés. De ce fait, nous considérons qu'il existe une relation de simulation paramétrée par une connexion de Galois entre le modèle concret et les modèles abstraits que nous construisons.

Comme **H0** est une hypothèse, nous réalisons une vérification *a posteriori* et fournissons un certificat, après chaque abstraction, que le modèle abstrait est simulé par le modèle concret. Les propriétés locales vérifiées sur le modèle abstrait sont alors préservées. Cette certification est intégrée à notre méthode et est automatiquement appliquée à tous les modèles abstraits que nous construisons.

L'avantage industriel d'une vérification *a posteriori* est qu'il devient possible d'**utiliser n'importe quel algorithme d'optimisation et n'importe quelle configuration lors de la synthèse** tant que notre certificat garantit la correction du modèle abstrait. Un second avantage est que ce même certificat garantit la **correction vis-à-vis du résultat de la synthèse**, proche de l'implémentation finale. L'unique restriction est d'utiliser le même algorithme et la même configuration pour synthétiser à la fois le modèle concret et le modèle

abstrait.

Concernant notre implémentation basée sur des logiciels libres, pour chaque propriété locale, nous procédons comme suit :

1. Nous construisons un *wrapper Verilog* pour le circuit où toutes les sorties dont aucun prédicat ne fait référence dans la propriété à vérifier sont déconnectées. Nous utilisons l'interface de programmation VPI de *Verilog* [46] pour identifier les entrées/sorties du circuit.
2. *Verilog2SMV* synthétise et traduit en automate de Büchi le modèle *Verilog*, encapsulé dans le *wrapper*, dans le langage SMV. L'étape d'optimisation embarquée dans l'algorithme de synthèse de *yosys* procède à une réduction par localisation du modèle, dédiée à la vérification de la propriété locale.
3. La vérification de propriétés locales contenant des quantificateurs est séparée en plusieurs sous-propriétés non paramétrées : une pour chaque valeur des propriétés paramétrées par le quantificateur. Cette approche réduit le coût du *model-checking* car la taille du BDD augmente exponentiellement avec la complexité de la propriété locale.
4. *NuSMV* est utilisé pour vérifier que la propriété locale est satisfaite par le modèle abstrait. Si la propriété n'est pas vérifiée, alors *NuSMV* génère un contre-exemple que nous convertissons au format VCD.

Ces opérations sont répétées pour chaque propriété locale comme décrites par le schéma encadré et nommé *Etape 3* dans la figure 4.1. Une description du fonctionnement des outils implémentant la méthode est donnée en annexe B.

4.2.4 Certificat de correction

Nous établissons une relation de simulation entre le modèle concret et chaque modèle abstrait telle que les propriétés temporelles sont préservées. Cette tâche est réalisée en comparant les fonctions de transition des automates de Büchi. Pour que la certification de correction soit compréhensible, nous devons d'abord introduire certains concepts au sujet des modèles décrits en langage SMV.

Les systèmes matériels que nous concevons sont modélisés en langage SMV par des automates de Büchi déterministes. Un automate décrit en langage SMV est défini par des variables d'entrée, des variables d'état, des valeurs initiales et des fonctions de transition [74, 45] :

— les variables d'entrée du modèle SMV représentent les ports d'entrée du système matériel (entrées du module *Verilog*) ; l'ensemble des valeurs possibles des n-uplets pour les variables d'entrée représente l'alphabet (Σ) de l'automate de Büchi.
— les variables d'état du modèle SMV représentent les mémoires du système (registres *Verilog*) sous forme de bits, vecteurs ou tableaux multidimensionnels ; l'ensemble des valeurs possibles des n-uplets pour les variables d'état représente l'ensemble fini d'états ($\mathcal{Q}$) de l'automate de Büchi.
— leurs valeurs initiales possibles sont le résultat d'une traduction de la directive *Verilog initial* [46] ; l'ensemble d'états initiaux (I) de l'automate de Büchi est le résultat du produit de toutes les valeurs initiales pour toutes les variables d'état du modèle SMV.
— pour finir, les fonctions de transition du modèle SMV définissent, pour chaque variable d'état du modèle SMV, l'état que cette variable doit prendre en fonction de son état

actuel, des valeurs des variables d'entrée et des valeurs des autres variables d'état. La fonction de transition globale (Δ) de l'automate de Büchi est aussi le résultat du produit de toutes les fonctions de transition du modèle SMV.

L'ensemble d'états finaux ($\mathcal{F}$) est toujours vide dans un modèle SMV, car ceux-ci sont une conséquence de la vérification d'une propriété sur l'automate. En effet, les états finaux sont, en *model-checking*, les états qui satisfont la négation d'une propriété sur le produit du modèle et de l'automate généré depuis la propriété à vérifier.

Les connexions des *wires Verilog* sont définies par *Verilog2SMV* comme des macros [46, 45]. Chaque macro est exprimée en logique du premier ordre (sans opérateur temporel) en fonction des variables d'état et variables d'entrée du modèle SMV. Elles n'interviennent donc pas dans la définition de l'automate de Büchi. Cependant, nous pouvons néanmoins les utiliser dans la définition des propriétés à vérifier.

Dans les modèles SMV, les systèmes sont donc représentés avec une relation de transition partitionnée sous forme conjonctive [28] : chaque partition (δ) de la fonction de transition (Δ) pour le modèle est donc une fonction de transition SMV, pour une variable d'état SMV. Pour établir notre relation de simulation, nous vérifions que toutes les partitions de la fonction de transition et les valeurs initiales des variables d'état du modèle abstrait sont identiques à celles présentes dans le modèle concret. De ce fait, nous établissons une simulation paramétrée par une connexion de Galois, où chaque état du modèle abstrait est similaire à tous les états du modèle concret dont les variables d'état ont la même valeur. Le modèle abstrait est alors une simplification du modèle concret basée sur une simulation par inclusion de langage et les propriétés de logique temporelle sont préservées [41].

Algorithme 1 Comparaison des fonctions de transition

Entrée:
C : modèle concret,
A : modèle abstrait
Sortie:
valeur booléenne : correction de l'abstraction

```
1: [Δ_C, I_C] = parse(C)
2: [Δ_A, I_A] = parse(A)
3: normalise(Δ_C) ; normalise(I_C)
4: normalise(Δ_A) ; normalise(I_A)
5: pour tout δ_A dans partition(Δ_A) faire
6:    si non (∃δ_C ∈ partition(Δ_C)|δ_C ≡ δ_A) alors
7:       retourne Faux
8:    fin si
9: fin pour
10: pour tout q_A dans partition(I_A) faire
11:    si non (∃q_C ∈ partition(I_C)|q_C ≡ q_A) alors
12:       retourne Faux
13:    fin si
14: fin pour
15: retourne Vrai
```

L'algorithme 1 représente comment nous réalisons la comparaison des fonctions de transition SMV. Pour s'exécuter, il nécessite deux modèles construits depuis la description RTL

du circuit : le modèle concret (C) et le modèle abstrait (A). Les deux modèles sont construits par *Verilog2SMV* avec le même algorithme d'optimisation lors de la synthèse et les mêmes configurations. L'algorithme retourne une valeur booléenne qui atteste de la correction de l'abstraction. Cette valeur est $Vraie$ si l'expression de toutes les fonctions de transition SMV et valeurs initiales des variables d'état existent avec la même expression dans le modèle concret.

— Tout d'abord, il *parse* le modèle concret et le modèle abstrait pour en extraire leur fonction de transition globale (Δ_C, Δ_A) et l'ensemble des états initiaux (I_C, I_A). Comme expliqué plus tôt, la fonction de transition globale de chaque automate est le résultat d'un produit entre les différentes fonctions de transition (δ_C, δ_A) des variables d'état. Respectivement, l'ensemble global d'états initiaux de chaque automate est le résultat d'un produit entre les différents états initiaux (q_C, q_A) de chaque variable d'état.

— Ensuite, il réalise une normalisation telle que les descriptions des états initiaux et des fonctions de transition SMV soient uniquement exprimées en fonction des lettres de l'alphabet (Σ) et des autres variables d'état ($\mathcal{Q}$). Cette étape est importante, car des circuits différents ont été synthétisés et l'algorithme d'optimisation peut fournir un résultat comportant des connexions différentes. *Verilog2SMV* préserve la hiérarchie des circuits et exprime ses connexions sous forme de macros [45] ; les états initiaux et fonctions de transition sont naturellement exprimés en fonction de ces macros. L'étape de normalisation permet donc d'abstraire les connexions du circuit, qui ne sont pas des mémoires et n'altèrent pas le BDD lors du *model-checking*.

Une implémentation de l'algorithme 1 est distribuée sous licence libre et est rendue accessible publiquement [1]. L'équivalence entre les fonctions de transition partitionnées est vérifiée au niveau syntaxique. Ceci est justifié par le fait que de nombreux modèles abstraits doivent avoir leur correction vérifiée (un par propriété locale décomposée) pour un seul circuit et que les performances sont améliorées. Cette implémentation utilise *pyNUSMV*. En conséquence, les fonctions de *parsing* et de normalisation résultent d'une instrumentation de *NuSMV* : le même *model checker* est utilisé pour vérifier la correction de l'abstraction et la propriété locale. Également, cela rend cette implémentation adaptable facilement pour répondre à un besoin particulier, par exemple en vérifiant une comparaison sémantique plutôt que de vérifier une équivalence.

Ressources : le script Python `4_2_4_certificat/equivalence.py` vérifie les équivalences entre les fonctions de transition de deux modèles. Il est accompagné de trois modèles SMV : un modèle concret, un modèle abstrait correct et un modèle abstrait incorrect. Le script Python permet la vérification de la correction des modèles abstraits vis-à-vis du modèle concret. Un `Makefile` permet d'automatiser la vérification pour les deux modèles abstraits.

4.3 Cas d'étude

Nous avons mis en œuvre notre méthode pour vérifier formellement les différents circuits qui composent le périphérique de confiance que nous avons conçu dans cette thèse. Une description détaillée de la stratégie que nous avons employée pour obtenir les propriétés de sécurité nécessaire à l'attestation à distance est donnée dans le chapitre 5. Prouver un théorème global sur la sécurité est un processus itératif impliquant du *model-checking* et de la preuve pour chaque module *Verilog*. La preuve peut être manuelle (à l'aide d'un assistant de preuves) ou

automatique (en utilisant les fonctionnalités de filtrage de *Spot*). Nous établissons des lemmes pour chacun de ces modules *Verilog* en suivant l'approche que nous avons décrite dans la section 4.2.

Afin d'illustrer cette approche, nous proposons dans cette section un cas d'étude centré sur une itération où nous réalisons une preuve de théorème automatique. L'itération que nous avons choisie consiste à vérifier des propriétés temporelles complexes et prouver un lemme sur le décompresseur de traces *CoreSight*. Une description de la compression des traces *CoreSight* est donnée dans la section 2.4. Ce choix est judicieux, car le décompresseur de traces est le module *Verilog* du périphérique qui contient le plus de mémoires : c'est donc notre goulot d'étranglement pour le *model-checking* en ce qui concerne l'explosion combinatoire. De plus, la spécification que nous proposons de vérifier ici est celle qui nécessite la construction de l'automate le plus complexe. Nous décrivons l'architecture du décompresseur dans la section 4.3.1 et la spécification que nous proposons de vérifier dans la section 4.3.2.

4.3.1 Décompresseur de traces

CoreSight transmet les traces d'exécution du microprocesseur par paquets. Ces paquets sont compressés de manière à ce que les données inchangées depuis les transmissions précédentes ne soient pas répétées. Un entête de paquet détermine le type de paquet qui est transmis et des bits de compression informent sur la présence d'octets suivants dans la donnée [59]. Notre décompresseur se charge de reconstruire les paquets non compressés à partir de ces informations. La figure 4.2 donne une représentation des entrées/sorties du décompresseur.

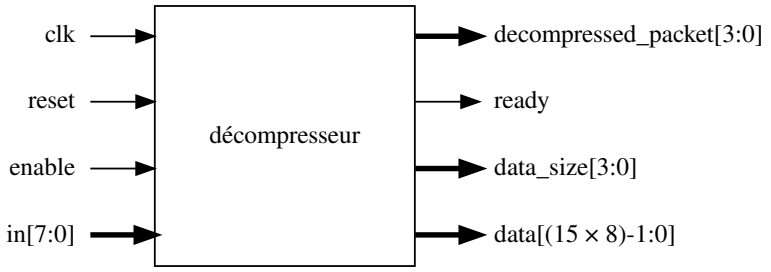

FIGURE 4.2 – Entrées/sorties du décompresseur

Le décompresseur obtient la donnée transmise par *CoreSight* depuis un vecteur d'entrée de 8 bits (*in*), il possède une entrée de synchronisation via une horloge (*clk*) et un bit d'activation (*enable*). La donnée est lue par le décompresseur sur l'entrée *in* lorsque *CoreSight* active le bit *enable*. *CoreSight* peut désactiver ce bit au milieu d'une réception pour une durée indéterminée. Lorsque la donnée de tout un paquet est reçue, le décompresseur fournit le contenu sur un long vecteur qui contient le paquet en entier (*data*). La taille minimum pour définir la mémoire du décompresseur est fixée par la taille du paquet le plus large que *CoreSight* peut transmettre, soit 15 octets de mémoire pour stocker le contenu d'un paquet *isync* [59].

Le type de paquet décompressé est identifié depuis son entête. La taille attendue pour un paquet transmis sur le vecteur de sortie dépend de cette identification et du contenu du paquet ; celle-ci peut varier entre 1 et 15 octets. Le signal de sortie *decompressed_packet* donne un identifiant unique pour le paquet en cours de décompression ; nous ignorons cette fonctionnalité

pour ce chapitre. Le signal de sortie *ready* est actif à la réception du dernier octet, le vecteur de sortie *data_size* donne le nombre d'octets présents dans le paquet décompressé et le vecteur de sortie *data* donne le contenu de la donnée.

4.3.2 Spécifications

Pour garantir la sécurité de la fonction d'attestation, notre périphérique doit décompresser de manière correcte certains types de paquets. En particulier, les paquets *branch* fournis par *CoreSight* contiennent les adresses de destination des branchements indirects et les informations sur les signaux d'exceptions. Ce sont les paquets les plus longs dont la décompression doit être formellement vérifiée [59]. Dans cette section, nous décrivons, en langage naturel, les spécifications pour une décompression correcte de paquet *branch*.

— La donnée est mémorisée depuis le vecteur d'entrée *in*, sur un front montant de *clk*, lorsque le bit *enable* est activé.
— Le type de paquet est déduit selon le contenu de l'entête, c'est-à-dire le premier octet reçu.
— Si l'entrée *reset* est activée lors d'un front montant de *clk*, alors la mémorisation et la déduction du type de paquet est abandonnée. Nous appelons cela réinitialiser le décompresseur.
— La sortie *ready* est activée lorsque le décompresseur reçoit le dernier octet d'un paquet *branch* et reste active pendant une seule période d'horloge. Nous appelons cela rendre le décompresseur prêt.
— La sortie *data_size* donne le nombre d'octets présents dans le paquet décompressé lorsque la sortie *ready* est activée.
— La sortie *data* donne le contenu de la donnée présente dans le paquet décompressé lorsque la sortie *ready* est activée.

La figure 4.3 donne un exemple de réception d'un paquet long de 4 octets. Les symboles w, x, y et z représentent les octets reçus (l'octet w contient l'entête du paquet) et le signal *memory* fait référence à une mémoire interne du décompresseur. Notons que le dernier octet est aussi transmis sur la sortie *data* même si celui-ci n'est pas encore présent en mémoire.

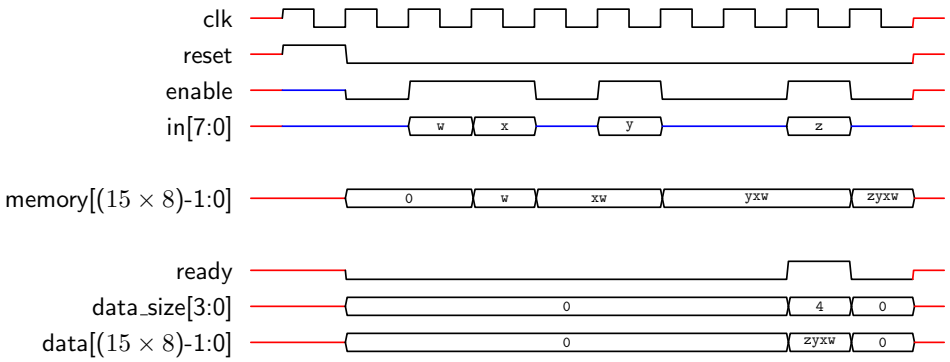

FIGURE 4.3 – Réception d'un paquet long de 4 octets

4.4 Mise en œuvre de la méthode

Dans cette section, nous montrons comment nous avons mis en œuvre notre méthode pour vérifier formellement la décompression des paquets *branch* fournis par *CoreSight*. Pour rappel, nous réalisons ici une preuve automatique (durant l'étape 2) même s'il est possible d'utiliser un assistant de preuves.

Le modèle concret du décompresseur requiert un total de 132 bits pour encoder les variables d'état SMV et 11 bits pour les variables d'entrée SMV. Sa description en langage *Verilog* contient environ 500 lignes de code. *Verilog2SMV* est utilisé pour construire le modèle concret en langage SMV, qui contient environ 5000 lignes de code. La description en langage *Verilog* et le modèle concret sont rendus accessibles publiquement [1].

Ressources : le fichier `4_4_0_model/packets_decompresser.v` représente la description du décompresseur en langage *Verilog*. Il est possible de faire appel à *Verilog2SMV* pour construire le modèle concret en langage SMV. Un `Makefile` permet d'automatiser la construction et de donner le nombre de lignes de code.

Lors du *model-checking*, durant l'étape 3, nous parallélisons l'exécution de *NuSMV* sur une ferme de calcul de 256 nœuds. Chaque nœud possède $4Go$ de RAM et un CPU mono-cœur fonctionnant à $3GHz$. Les temps de calcul et l'occupation mémoire peuvent varier selon la machine utilisée et la suite logicielle EDA choisie pour l'implémentation.

4.4.1 Étape 1 : expression des propriétés locales

La première étape consiste à exprimer formellement les spécifications et propriétés locales en langage PSL. L'expression formelle est un procédé difficile qui peut mener à des résultats différents selon les choix d'écriture du concepteur : plusieurs expressions PSL peuvent décrire les mêmes spécifications en langage naturel.

Nous fournissons ici un exemple d'expression formelle depuis les spécifications décrites en langage naturel dans la section 4.3.2. Pour faciliter la compréhension, nous omettons ici intentionnellement certains aspects de la spécification, comme par exemple la détection du type de paquet via l'entête. Nous omettons également la duplication des opérateurs **forall** lors de l'utilisation de plusieurs quantificateurs avec le même intervalle. Bien que syntaxiquement incorrect car le quantificateur PSL **forall** n'accepte qu'un seul identifiant, cela réduit grandement la représentation des expressions. Les mêmes expressions décrites en langage SMV, syntaxiquement correctes et tenant compte des entêtes, sont rendues publiquement accessibles [1].

Pour garantir la sécurité de notre périphérique, la décompression doit être correcte pour les paquets *branch* longs de 7 octets. En effet, ces octets contiennent l'adresse de destination et les informations sur les signaux d'exceptions. La correction de cette décompression est formellement exprimée par l'expression 4.3.

Cette expression PSL décrit les traces d'exécution qui commencent lorsque le décompresseur est prêt ou réinitialisé. À partir de l'instant suivant, une restriction pour les traces d'exécution est que l'entrée *reset* ne peut pas être activée sur un front montant d'horloge jusqu'à ce que la sortie *ready* soit active. Également, une autre restriction pour les traces d'exécution est que le nombre d'octets dans le paquet décompressé doit être supérieur ou égal à 7.

Le côté gauche de l'implication exprime que la donnée est mémorisée depuis l'entrée *in*, sur front montant de *clk*, lorsque le bit *enable* est activé. Ceci est vrai pour toutes les valeurs possibles entre 0 et 255 pour chacun des 7 octets. Le côté droit de l'implication exprime que la sortie *data* prend la valeur du paquet décompressé lorsque la sortie *ready* est active.

$$
\begin{aligned}
&\textbf{forall}\ i_0,i_1,i_2,i_3,i_4,i_5,i_6\ \textbf{in}\ \{0:255\} : \textbf{always}(\\
&\quad ((clk \wedge reset) \vee (clk \wedge ready)) \wedge \textbf{next}(\\
&\qquad \neg(clk \wedge reset)\ \textbf{until}\ (clk \wedge ready) \\
&\qquad \wedge \textbf{next\_event}(clk \wedge ready)(data\_size \geq 7) \\
&\qquad \wedge \textbf{next\_event\_a}(clk \wedge enable)[1:1](in = i_0) \\
&\qquad \wedge \textbf{next\_event\_a}(clk \wedge enable)[2:2](in = i_1) \\
&\qquad \wedge \textbf{next\_event\_a}(clk \wedge enable)[3:3](in = i_2) \\
&\qquad \wedge \textbf{next\_event\_a}(clk \wedge enable)[4:4](in = i_3) \\
&\qquad \wedge \textbf{next\_event\_a}(clk \wedge enable)[5:5](in = i_4) \\
&\qquad \wedge \textbf{next\_event\_a}(clk \wedge enable)[6:6](in = i_5) \\
&\qquad \wedge \textbf{next\_event\_a}(clk \wedge enable)[7:7](in = i_6) \\
&\quad) \\
&\rightarrow \\
&\quad \textbf{next}(\\
&\qquad \textbf{next\_event}(clk \wedge ready)[1](data[7:0] = i_0) \\
&\qquad \wedge \textbf{next\_event}(clk \wedge ready)[1](data[15:8] = i_1) \\
&\qquad \wedge \textbf{next\_event}(clk \wedge ready)[1](data[23:16] = i_2) \\
&\qquad \wedge \textbf{next\_event}(clk \wedge ready)[1](data[31:24] = i_3) \\
&\qquad \wedge \textbf{next\_event}(clk \wedge ready)[1](data[39:32] = i_4) \\
&\qquad \wedge \textbf{next\_event}(clk \wedge ready)[1](data[47:40] = i_5) \\
&\qquad \wedge \textbf{next\_event}(clk \wedge ready)[1](data[55:48] = i_6) \\
&\quad) \\
&)
\end{aligned}
\tag{4.3}
$$

L'expression des propriétés locales est un procédé manuel. La conjonction de ces propriétés locales doit être suffisante pour prouver la spécification 4.3. Le concepteur peut découper la spécification avec un objectif de preuve de théorème automatique ; abstraire une sortie du module dans une propriété locale réduit à la fois la complexité de l'automate construit depuis cette propriété et celle du modèle abstrait utilisé pour la vérifier.

Une solution envisageable pour exprimer les propriétés locales est de découper le vecteur de sortie *data* de sorte à faire apparaitre les 7 octets. Dans ce cas, chaque propriété locale possède un seul quantificateur sur un intervalle de 256 valeurs. Une autre solution envisageable est de découper le vecteur de sortie *data* pour faire apparaitre 7×8 bits. Dans ce cas, chaque propriété locale possède un quantificateur sur un intervalle de 2 valeurs. La première solution est intuitive, car exprimer une propriété locale revient simplement à retirer des opérateurs **next_event_a** et **next_event** de la spécification 4.3. La seconde solution requiert une réécriture plus poussée, mais réduit davantage la complexité de la propriété et du modèle abstrait, ce qui rend envisageable le *model-checking* pour des spécifications plus complexes.

Pour notre cas d'étude, la première solution est suffisante pour permettre la vérification formelle en un temps acceptable. Un exemple de propriété locale est représenté par l'expression 4.4. Un total de 7 propriétés locales, en suivant la même approche de découpage, est requis pour permettre la preuve de la spécification.

$$
\begin{array}{l}
\textbf{forall}\ i_0\ \textbf{in}\ \{0:255\}: \\
\textbf{always}(\\
\quad ((clk \wedge reset) \vee (clk \wedge ready)) \wedge \textbf{next}(\\
\quad\quad \neg(clk \wedge reset)\ \textbf{until}\ (clk \wedge ready) \\
\quad\quad \wedge\ \textbf{next\_event}(clk \wedge ready)(data\_size \geq 1) \\
\quad\quad \wedge\ \textbf{next\_event\_a}(clk \wedge enable)[1:1](in = i_0) \\
\quad) \\
\rightarrow \\
\quad\quad \textbf{next}(\\
\quad\quad\quad \textbf{next\_event}(clk \wedge ready)[1](data[7:0] = i_0) \\
\quad\quad) \\
\quad)
\end{array}
\tag{4.4}
$$

Il est possible d'avoir des propriétés locales plus expressives que nécessaire à la preuve de la spécification. Cela est utile dans le cas où la même propriété locale est utilisée pour prouver plusieurs théorèmes. Pour cette raison, la valeur de la sortie *data_size*, dans la propriété locale 4.4 est seulement contrainte à être supérieure ou égale à 1 (nous vérifions ici la seulement la décompression du premier octet). Néanmoins, en cas de modification, le concepteur doit rester prudent pour que la spécification reste prouvable.

4.4.2 Étape 2 : stratégie de preuve

Lorsque nous sommes en possession des 7 propriétés locales, elles constituent nos hypothèses pour prouver la spécification. Cette preuve peut être manuelle, automatique, ou la combinaison des deux ; nous choisissons ici de réaliser une preuve entièrement automatique. Comme nous avons construit nos propriétés locales avec pour objectif une de preuve de théorème automatique, la stratégie de preuve est intuitive :

- les expressions PSL qui apparaissent à la fois dans la spécification et dans les propriétés locales peuvent être abstraites et laissées non interprétées ; par exemple, nous pouvons remplacer les expressions PSL suivantes par de simples prédicats :
  - **next_event_a**$(clk \wedge enable)[1:1](in = i_0)$
  - **next_event**$(clk \wedge ready)[1](data[7:0] = i_0)$
- des axiomes sont ajoutés pour garder une cohérence lorsque les propriétés locales sont plus expressives que nécessaire. Par exemple, l'axiome suivant peut être défini : si la valeur de *data_size* est supérieure ou égale à 7, alors elle est supérieure ou égale à 1.
- nous pouvons utiliser nos outils pour automatiquement traduire du langage PSL vers LTL, fournir des abstractions de fonctions non interprétées (afin de réduire l'impact du dépliage) et construire un script *Python* qui permet le filtrage automatique.

La figure 4.4 schématise la stratégie que nous pouvons employer pour prouver, de manière automatique, la spécification 4.3. Cette stratégie peut être décomposée en trois étapes (notées

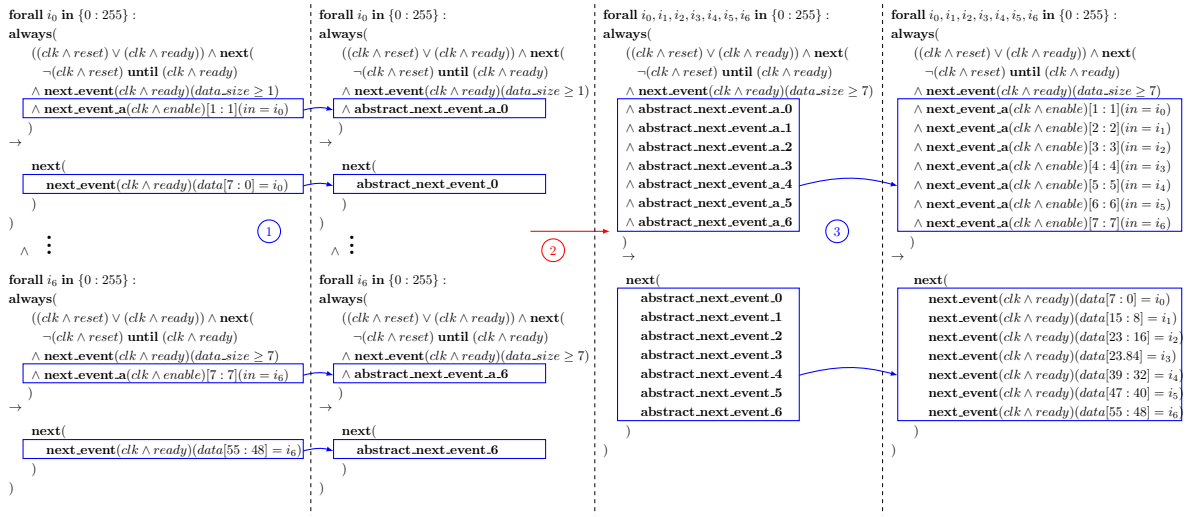

FIGURE 4.4 – Décompression des paquets *branch* : stratégie de preuve

respectivement 1, 2 et 3 sur la figure) :

1. Les expressions PSL à abstraire et à laisser non interprétées sont remplacées par de simples prédicats. Une équivalence entre ces prédicats, nouvellement introduits, et lesdites expressions est axiomatique.
2. Une preuve de théorème est réalisée à l'aide des algorithmes de filtrage de *Spot*. Cette preuve consiste à réduire l'implication de la spécification par la conjonction des propriétés, où nous utilisons les prédicats introduits lors de l'abstraction. Cette étape où nous utilisons *Spot* est représentée en rouge sur la figure 4.4.
3. Les prédicats introduits lors de l'abstraction sont raffinés, d'après les axiomes décrivant une équivalence, pour faire apparaitre la spécification sous sa forme originale.

Comme démontrer un théorème de manière automatique peut se heurter à une explosion combinatoire, il est également possible de séparer la preuve (c'est-à-dire l'étape 2 de la figure 4.4) en plusieurs morceaux pour optimiser le calcul. Dans ce cas, nous démontrons un lemme exprimant qu'une propriété intermédiaire est impliquée. Cette propriété intermédiaire est plus large que chacune des propriétés locales, mais plus petite que la spécification. Depuis plusieurs propriétés intermédiaires, chacune démontrée par un lemme, nous pouvons prouver la spécification en employant exactement la même méthode.

Par exemple, une propriété intermédiaire peut décrire la décompression des 4 premiers octets et une seconde propriété intermédiaire décrit la décompression des 3 octets suivants. Dans ce cas, nous réalisons trois preuves, une pour chacun des deux lemmes et une pour la spécification, mais nous réduisons l'effort de preuve réalisé par *Spot* lors du filtrage. Les étapes $2'$ et $2''$ de la figure 4.5 représentent, dans ce cas, comment nous optimisons le calcul. Le tableau 4.1 récapitule l'occupation mémoire et les temps de calcul, sur un nœud de la ferme de calcul, pour une preuve avec et sans optimisation.

Notons que séparer la preuve automatique en plusieurs morceaux est également une solution élégante dans le cas où nous optons pour la seconde solution d'expression des propriétés

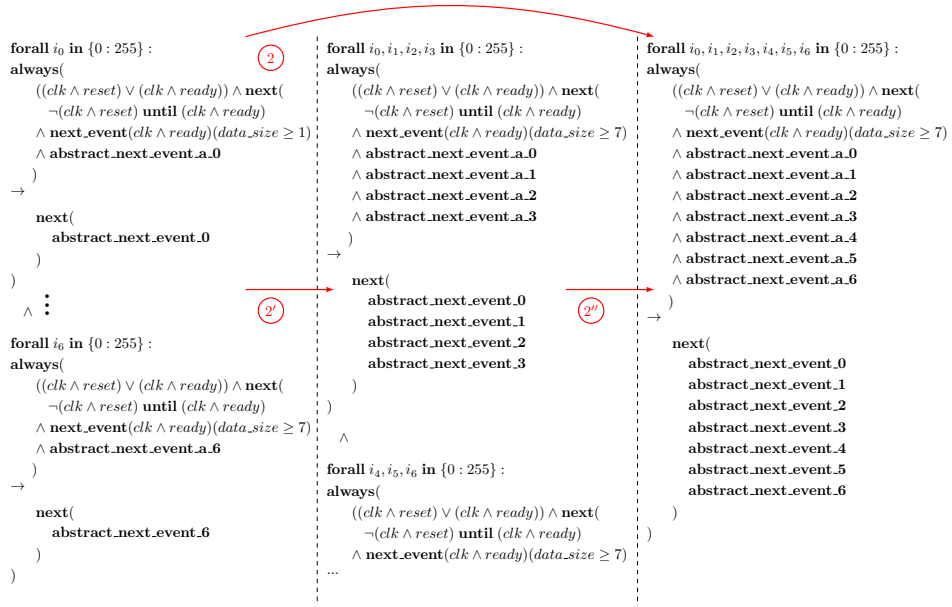

FIGURE 4.5 – Preuve automatique : réduction de l'effort de preuve

Optimisation	%MEM	Temps de calcul
Non (étape 2)	9.6	2 minutes
Oui (étapes 2′ et 2″)	< 1	< 2 secondes

TABLE 4.1 – Preuve automatique : occupation mémoire et temps de calcul

locales, c'est-à-dire découper le vecteur de sortie *data* pour faire apparaitre 7×8 bits (comme décrit dans la section 4.4.1). Dans ce cas, les 7 propriétés similaires à celle décrite par l'expression 4.4 représentent nos propriétés intermédiaires. Elles sont prouvées depuis une implication par la conjonction des propriétés locales et le reste de la preuve reste identique.

Ressources : le dossier `4_4_2_preuve_cas_etude/spot_proof/` contient les définitions des 7 propriétés locales, où l'abstraction des expressions PSL non interprétées a déjà été réalisée, ainsi que des sous-dossiers pour jouer les différentes preuves. Il est possible d'utiliser l'outil *psl2spot* pour créer des scripts *Python* et réduire automatiquement l'implication des spécifications. Un `Makefile` permet d'automatiser la preuve avec et sans les optimisations. La cible `optimized` sépare la preuve en deux morceaux tandis que la cible `one_step` la joue en un seul morceau.

Au sujet de l'abstraction des expressions PSL non interprétées, nous considérons axiomatiques les équivalences entre les propriétés contenant les expressions PSL et les mêmes propriétés utilisant des prédicats (utilisés pour l'abstraction). Une preuve écrite en coq démontre que nous pouvons bien utiliser des prédicats pour abstraire les propriétés locales et raffiner la spécification. Le programme qui démontre ceci est distribué sous licence libre et est rendu accessible publiquement [1]. La preuve utilise la bibliothèque `LTL_Coq` pour définir les règles sémantiques de la LTL. Nous considérons que la sémantique de la LTL est identique

pour cette bibliothèque et pour *Spot*. En conséquence, pour cette preuve en particulier, il n'est pas nécessaire de vérifier la correction de l'abstraction avec *Spot*.

Ressources : le dossier `4_4_2_preuve_cas_etude/coq_abstract/` contient le programme qui démontre que nous pouvons utiliser des prédicats pour abstraire les propriétés locales et raffiner la spécification. Cette preuve se base sur une définition de vecteurs que nous ne détaillons pas ici. Un `Makefile` permet d'automatiser sa compilation et son exécution.

Cependant, dans le but de prôner en faveur de l'abstraction des expressions PSL non interprétées, nous avons utilisé *Spot* pour vérifier les équivalences des propriétés locales, avec et sans abstraction, pour certains prédicats. Une vérification consiste à prouver une équivalence entre une propriété locale et sa forme abstraite, c'est-à-dire prouver la correction de l'étape 1 représentée sur la figure 4.4. Dans leur forme abstraite, pour toute valeur de l'entier $j \in \{0:6\}$, les expressions PSL suivantes sont remplacées par de simples prédicats :

— **next_event_a**$(clk \wedge enable)[j+1:j+1](in = i_j)$
— **next_event**$(clk \wedge ready)(data[8 \times (j+1) - 1 : 8 \times j)] = i_j)$

Une telle vérification est gourmande en ressources, car la complexité des expressions PSL augmente de manière exponentielle avec la valeur de l'entier j. Le tableau 4.2 récapitule l'occupation mémoire et les temps de calcul, sur un nœud de la ferme de calcul, pour une preuve d'équivalence des propriétés locales en fonction de la valeur de l'entier j.

j	%MEM	Temps de calcul
0	< 1	10 secondes
1	< 1	35 secondes
2	1.8	4 minutes
3	100	(limites de la machine atteintes)

TABLE 4.2 – Vérification facultative de la correction des abstractions

Cette expérience montre les limites de la preuve de théorème automatique lorsque nous travaillons avec des propriétés complexes. La conversion des opérateurs temporels étendus vers des opérateurs temporels de base, qui n'est pas requise si les mêmes expressions apparaissent des deux côtés de l'implication, cause une explosion combinatoire et empêche la preuve d'aboutir. La cause de cette explosion combinatoire est que le nombre d'opérateurs temporels de base grandit de manière exponentielle avec la valeur de l'entier j.

Ressources : le dossier `4_4_2_preuve_cas_etude/spot_abstract/` contient quatre sous-dossiers permettant de reproduire l'expérience. Chacun des dossiers permet d'observer l'occupation mémoire et les temps de calcul pour les différentes valeurs de l'entier j, comme listées dans le tableau 4.2. Chaque sous-dossier contient respectivement 3 fichiers comportant des propriétés PSL et un `Makefile` permettant d'automatiser le calcul. Le fichier `axioms.smv` contient les équivalences entre les expressions PSL à abstraire (à laisser non-interprétées) et les prédicats qui les remplacent. Le fichier `property.smv` contient la propriété locale sous sa forme concrète (sans abstraction). Le fichier `intermediate.smv` contient la propriété locale sous sa forme abstraite, utilisant les prédicats.

4.4.3 Étape 3 : réduction par localisation

La conjonction de nos propriétés locales est suffisante pour prouver la spécification. Désormais, du *model-checking* doit garantir que le décompresseur vérifie ces propriétés locales.

Comme nous avons opté pour des propriétés locales où le vecteur de sortie *data* est découpé par octet, notre méthode réduit automatiquement le modèle de 14 octets sur les 15 octets de mémoire. Cette réduction est valable pour chaque abstraction, quelque-soit l'octet de mémoire qui est conservé. De plus, comme nos propriétés locales possèdent un quantificateur avec un intervalle de 256 valeurs, notre méthode sépare la vérification en 256 étapes où chaque propriété locale est non paramétrée. En conséquence, la vérification d'une propriété locale sur notre modèle concret revient à 256 vérifications de sous-propriétés, réduites d'un facteur 2^8, sur un modèle abstrait réduit d'un facteur $2^{(14\times 8)}$. Le nombre de vérifications augmente donc linéairement, tandis que le BDD construit lors de la vérification est réduit de manière exponentielle.

Notons que, même si notre spécification n'était pas découpée en propriétés locales, notre méthode aurait tout de même réduit le modèle de 8 octets sur les 15 octets de mémoire. Ceci est une conséquence du fait que les 8 octets de poids fort du vecteur *data* n'apparaissent pas dans l'expression de la spécification 4.3. La procédure de vérification aurait également été décomposée en 7×256 étapes, car la spécification possède 7 quantificateurs avec un intervalle de 256 valeurs.

Pour vérifier la correction des abstractions, notre méthode permet de construitre automatiquement un modèle concret SMV en plus des différents modèles abstraits. Pour chaque modèle abstrait, une relation de simulation est établie entre celui-ci et le modèle concret précédemment construit :

- en cas de succès, *NuSMV* et appelé pour vérifier la propriété locale sur l'abstraction correspondante ;
- en cas d'échec, un journal liste les différences entre les fonctions de transition et les états initiaux des variables locales qui diffèrent. Une configuration de l'algorithme de synthèse peut être réalisée en conséquence.

NuSMV réalise du *model-checking* symbolique de chaque propriété sur l'abstraction correspondante. En fin de calcul, un fichier journal contient l'expression de la propriété vérifiée et les résultats :

- si la propriété est vérifiée, alors l'expression de la propriété locale est suivie de la mention "*is true*" ("est vraie") ;
- si la propriété est falsifiée, alors *NuSMV* fournit un contre-exemple que nos outils convertissent en VCD pour une visualisation par un outil dédié.

L'occupation mémoire et les temps de calcul dépendent à la fois de la taille du modèle abstrait et de la complexité de la propriété locale. Dans notre cas d'étude, la taille du modèle abstrait est identique pour chacune des 7 propriétés locales. La complexité de la propriété est liée à la valeur du nombre entier présent dans l'intervalle de l'opérateur **next_event** ou **next_event_a**. Le tableau 4.3 et le graphique 4.4.3 résument l'occupation mémoire et les temps de calcul de *NuSMV* sur un nœud de la ferme de calcul (en moyenne), c'est-à-dire pour la vérification d'une seule sous-propriété. L'index des lignes du tableau représente le nombre entier présent dans l'intervalle de l'opérateur **next_event_a**. Les lignes 1 à 7 correspondent aux cas de la vérification de nos propriétés locales.

Notons que, pour le bien de l'expérience, nous avons vérifié deux propriétés locales qui

#	%MEM	Temps de calcul
1	21.9	7 secondes
2	28.4	8 secondes
3	29.9	10 secondes
4	39.5	15 secondes
5	60.2	35 secondes
6	28.7	7 minutes
7	41.6	25 minutes
8	47.9	39 minutes
9	95.0	6 heures 55 minutes

TABLE 4.3 – Model-checking : occupation mémoire et temps de calcul

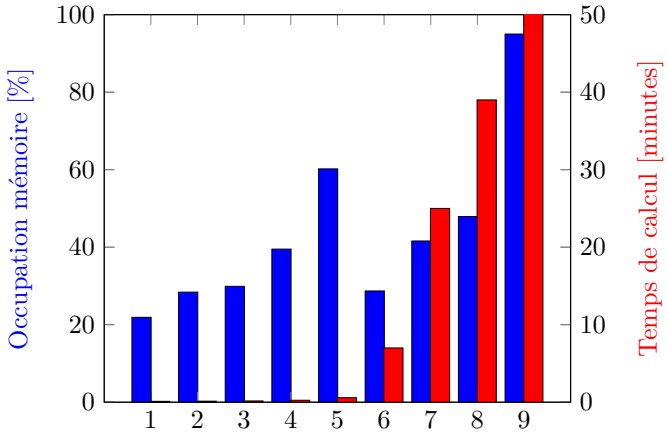

FIGURE 4.6 – Model-checking : occupation mémoire et temps de calcul

ne sont pas nécessaires à la preuve de sécurité. Pour cela, nous avons augmenté les valeurs du nombre entier présent dans l'intervalle de l'opérateur **next_event_a** à 8 et à 9. Cela permet d'évaluer notre stratégie de décomposition en propriétés locales pour des spécifications plus complexes. Cela montre que nous pouvons conserver la même approche pour vérifier la décompression de paquets contenant 8 octets de donnée, mais que nous atteignons presque les limites de la machine au neuvième octet. Les sources pour reproduire l'expérience sont rendues accessibles publiquement [1]. Nous recommandons cependant de réaliser le *model-checking* sur une ferme de calcul de 256 nœuds, sans quoi les temps de calcul pour une propriété locale seront approximativement 256 fois plus élevés.

Ressources : le dossier `4_4_3_model_checking_cas_etude/` contient les sept propriétés locales et la définition en langage *Verilog* du décompresseur de traces. Notre méthode permet d'automatiser la réduction par localisation et le *model-checking* pour vérifier les propriétés locales. Un `Makefile` permet d'automatiser l'appel à nos outils et de vérifier la première propriété locale. Pour vérifier les sept propriétés locales, il est possible d'utiliser `make` avec la cible `full`.
Une propriété non vérifiée par le décompresseur de trace est présente dans un fichier nommé `wrong_property.smv`. Il est possible d'observer un contre-exemple généré par *NuSMV* et converti en VCD en utilisant nos outils. Pour observer ce contre-exemple, il est possible d'utiliser `make` avec la cible `wrong_property`.

4.4.4 Résumé

Notre méthode permet de construire automatiquement des modèles abstraits corrects, proches de l'implémentation finale, pour notre décompresseur de traces *CoreSight*. Ainsi, nous pouvons vérifier des propriétés locales en un temps acceptable et réaliser du *model-checking* compositionnel guidé par la preuve de théorème. Les modèles abstraits sont déduits des propriétés locales utilisées pour la preuve. Ils sont construits par un algorithme d'optimisation lors d'une synthèse du circuit *Verilog*.

Nous avions plusieurs solutions pour exprimer nos propriétés locales, nous avons opté pour un découpage des vecteurs de sorties en octets, car la preuve était rendue plus accessible et le *model-checking* était toujours possible. L'avantage est que nous pouvions exprimer nos propriétés locales avec un objectif de preuve de théorème, donc trouver une stratégie de preuve était intuitif. Nous avons abstrait les expressions qui apparaissaient à la fois dans la spécification et dans les propriétés locales.

Face à une spécification plus complexe, nous aurions pu opter pour un découpage des vecteurs de sorties en bits. L'avantage est que cela aurait grandement réduit les temps de calcul et l'occupation mémoire du *model-checking* (réduction de la propriété locale et du modèle abstrait). En contrepartie, la preuve aurait dû être séparée en plusieurs étapes pour créer des propriétés intermédiaires.

Dans cette thèse, nous utilisons nos outils (et la même stratégie que montrée ici) pour la vérification formelle de tous nos circuits décrits en *Verilog*. Cela permet de vérifier des propriétés locales sur notre périphérique de confiance, de sécuriser l'exécution de la fonction d'attestation et, par extension, d'établir une racine de confiance statique. La stratégie de preuve de sécurité est décrite dans le chapitre 5. Notons simplement que nous préférons démontrer nos théorèmes manuellement, avec l'aide d'un assistant de preuves, afin de ne pas faire face à l'explosion combinatoire lors de l'étape 2 (preuve de la spécification).

4.5 Limitations et perspectives

Notre approche peut également être généralisée à la vérification formelle de tout circuit électronique numérique. Dans cette section, nous listons les potentielles limitations auxquelles le concepteur peut faire face lors d'une telle généralisation et les solutions qui permettent de s'en affranchir. De plus, nous proposons une extension pour faciliter la preuve de théorème via un assistant de preuves.

4.5.1 Sur-approximation

Une conséquence de notre critère de réduction par localisation est que l'abstraction dépend directement de la présence de prédicats, dans la propriété locale, qui font référence aux sorties du système. Aussi, la sur-approximation est une limitation possible dans le cas où certains registres doivent être conservés, afin que la relation de simulation puisse être établie.

Pour illustrer cette limitation, imaginons le cas où notre système possède une machine à états finis dont le nombre total d'états peut être encodé sur n bits. Si certaines sorties du système n'apparaissent pas dans la propriété locale, nos outils créent un *wrapper* pour les déconnecter. Dans ce cas, selon l'algorithme d'optimisation et la configuration utilisés, il est possible que le résultat de la synthèse produise une machine à états finis dont le nombre total d'états peut être encodé sur un nombre inférieur de bits : par exemple $n - 1$. Les variables d'état des modèles SMV diffèrent en taille. En conséquence, nous ne pouvons plus établir une relation de simulation paramétrée par une connexion de Galois entre le modèle concret et le modèle abstrait généré.

Cette limitation s'explique par le fait que l'algorithme d'optimisation peut être trop agressif et réduise la taille des registres lorsque certains bits ne sont plus nécessaires. Le contournement de cette limitation requiert simplement d'**indiquer à l'algorithme d'optimisation de ne pas réencoder certains vecteurs**, en particulier ceux décrivant les registres encodant l'état de la machine à états. Dans le cas de *yosys*, lors d'une synthèse décrite par une suite de commandes personnalisées, il suffit simplement de ne pas utiliser la fonctionnalité de *FSM Recoding*, soit de ne pas utiliser la commande `fsm_recode` [47].

4.5.2 psl2coq : hypothèses pour l'assistant de preuves

Malgré plusieurs abstractions, prouver la spécification 4.3 depuis la conjonction des propriétés locales peut nécessiter un temps de calcul ou une quantité de mémoire qui est trop élevée. Une solution simple, comme montré sur la figure 4.5, est de démontrer des lemmes intermédiaires. Une autre solution est de recourir à une preuve manuelle à l'aide d'un assistant de preuves, afin de ne pas dépendre de la mémoire de la machine sur laquelle le théorème est établi.

Pour rendre accessibles les propriétés vérifiées par *NuSMV* dans l'assistant de preuve Coq, nous fournissons un programme permettant la conversion des expressions PSL. Ce programme est nommé *psl2coq*. Tout comme *psl2spot*, il est développé en Python et utilise la bibliothèque *pyNUSMV*.

Contrairement, à *psl2spot*, les propriétés PSL ne sont pas converties en LTL. L'outil parcourt l'arbre syntaxique et effectue une réécriture des opérateurs temporels (même des opérateurs temporels étendus) avec la syntaxe Gallina (langage dans lequel les termes Coq sont écrits). **Une bibliothèque Coq accompagne l'outil de manière à ce que les opérateurs réécrits soient définis**. Cette bibliothèque comporte la définition d'opérateurs temporels étendus et est également accompagnée d'un ensemble de théorèmes génériques que nous avons établis et utilisons dans les preuves fournies avec ce manuscrit.

Chaque propriété, décrite en langage SMV et vérifiée sur un modèle du circuit, forme alors une hypothèse pour l'assistant de preuve Coq. Dans les chapitres 5 et 6, nous utilisons l'outil *psl2coq* pour générer automatiquement les hypothèses dont nous avons besoin pour établir nos preuves de sécurité. L'outil *psl2coq* est distribué sous licence libre et est rendu accessible publiquement [1].

Ressources : le dossier `4_5_2_psl2coq/` contient l'outil de conversion *psl2coq* et les sources de la bibliothèque Coq qui l'accompagne.
Des propriétés PSL décrites en langage SMV accompagnent l'outil. Elles peuvent être converties en sources d'un programme Coq. Un `Makefile` permet d'automatiser la conversion et la compilation du programme.

L'annexe C.5.1 détaille la preuve de la spécification 4.3 par la conjonction des propriétés locales. Dans la suite de ce manuscrit, nous utilisons cette méthode de preuve manuelle pour toutes les preuves présentées afin de s'affranchir des temps de calcul d'une preuve automatique.

4.5.3 Model-checking et réduction du temps de calcul

Une autre limitation peut apparaître lorsque le système que nous souhaitons vérifier possède un grand nombre d'entrées qui influent sur les états de l'automate, ou lorsque la mémoire d'un automate atomique excède la taille tolérée. Dans ce cas, le temps de calcul nécessaire pour vérifier, par *model-checking*, une propriété locale n'est plus raisonnable. Une solution simple pour limiter le temps de calcul consiste à altérer l'architecture du système pour le rendre partitionné. Cette stratégie doit néanmoins être anticipée, car geler l'architecture d'un système fait partie des premières étapes du flot de conception pour les circuits intégrés.

Une autre solution est de recourir à une **vérification d'invariants et non de propriétés temporelles**. Vérifier un invariant consiste à raisonner sur chaque état du BDD indépendamment. Une telle vérification ne nécessite pas de dépliage de l'automate et améliore grandement les performances. En contrepartie, la preuve des lemmes (propriétés temporelles telles que celle représentée par l'expression 4.4) requiert une plus grande expertise en logique et maîtrise de l'assistant de preuves. Nous recommandons donc de rendre le système partitionné lorsque ceci est possible afin d'éviter une démonstration trop difficile.

Afin d'illustrer la procédure, l'annexe C.5.2 propose un exemple de vérification d'invariants dans le but de prouver une propriété temporelle sur le décompresseur. Dans la suite de ce manuscrit, nous utilisons cette méthode de vérification d'invariants combinée avec une preuve afin de s'affranchir des temps de calcul trop élevés causés par le *model-checking*.

4.6 Conclusion

Dans cette thèse, nous construisons un périphérique de confiance pour microprocesseur, dont la vérification permet l'établissement d'une racine de confiance statique. Cette vérification dépend d'une modélisation du microprocesseur et de ses périphériques. En particulier, nous modélisons le comportement de l'interface de débug *CoreSight* depuis les données disponibles dans la documentation. Ce modèle nous oblige à développer un décompresseur de traces dont la mémoire et les spécifications empêchent la vérification formelle d'aboutir.

Nous nous tournons donc vers une stratégie de preuve, où des propriétés locales sont vérifiées par *model-checking* et un théorème démontre la vérification des spécifications. Malheureusement, compte tenu de la complexité de ces spécifications, exprimer des propriétés locales de manière cohérente vis-à-vis de la preuve et des modèles abstraits (dédiés à la vérification) devient une tâche difficile. Également, pour rendre la stratégie de preuve envisageable à grande échelle, la construction des modèles abstraits et la démonstration du théorème doit être automatique.

Dans ce chapitre, nous avons donc décrit une méthode pour la vérification formelle automatisée de systèmes matériels, qui permet à la vérification de passer à l'échelle. Les propriétés locales sont exprimées de manière cohérente seulement vis-à-vis de la preuve. Les modèles abstraits sont déduits de celles-ci et construits de manière automatique en tirant profit des outils de synthèse. Une vérification *a posteriori* fournit un certificat que le modèle abstrait est simulé par le modèle concret. De ce fait, la vérification formelle est réalisée sur des modèles abstraits proches de l'implémentation finale.

Notre solution permet la vérification de notre périphérique de confiance ; elle peut aussi permettre la vérification de tout système matériel numérique dont la chaîne de conception est maîtrisée. Également, celle-ci est une approche adaptable à n'importe quelle suite de logiciels d'EDA. Elle peut donc être adoptée simplement dans l'industrie. Nous avons accompagné la description de notre méthode d'un ensemble d'outils où nous avons choisi des logiciels libres pour l'implémenter. Ces outils sont également distribués sous licence libre.

Dans les prochains chapitres, nous nous appuierons sur cette méthode pour permettre l'établissement d'une racine de confiance statique sur microprocesseur. En ce sens, nous proposerons une architecture co-conçue, logicielle et matérielle, pour notre périphérique de confiance. Sa vérification sera un processus itératif impliquant du *model-checking* et de la preuve, où chaque itération suivra une procédure similaire à celle décrite dans ce chapitre. Cette contribution est décrite dans le chapitre 5.

Chapitre 5

Racine de confiance statique sur microprocesseur

Sommaire

Dans le chapitre 1, nous avons montré que l'état-de-l'art propose des architectures vérifiées formellement, permettant d'établir une racine de confiance statique sur microcontrôleur. Ces architectures résultent d'une modification matérielle au niveau du cœur du système pour extraire des observables. Également, elles sont proposées pour des systèmes simples et ouverts tels que le *openMSP430*.

Dans ce chapitre, nous proposons une solution co-conçue pour établir une racine de confiance statique sur microprocesseur pris sur étagère, c'est-à-dire un système dont le cœur ne

peut pas être modifié. Cette solution se présente sous la forme d'un périphérique de confiance. Nous proposons une stratégie de vérification formelle pour ce périphérique et une première implémentation sur le *SoC Xilinx Zynq-7000*. Notons que cette première implémentation comporte des faiblesses que nous admettons dans ce chapitre, mais auxquelles nous proposons des contre-mesures dans les chapitres suivants. Ces travaux ont fait l'objet de deux publications [86, 87] et de présentations aux conférences *Rendez-vous de la Recherche et de l'Enseignement de la Sécurité des Systèmes d'Information* (RESSI) 2020 et *International Workshop on Cyber-Security and Functional Safety in Cyber-Physical Systems* (IWCFS) 2021.

La section 5.1 décrit le contexte des travaux. La section 5.2 présente la stratégie de vérification formelle que nous envisageons. La section 5.3 décrit l'architecture que nous proposons pour notre périphérique de confiance. La section 5.4 décrit la vérification de la sécurité pour la première implémentation ; les détails sont fournis dans la section 5.5. La section 5.6 décrit la vérification de la *soundness* pour cette même implémentation. Finalement, les résultats sont donnés dans la section 5.7.

Dans une optique de rejeu des expériences illustrant nos travaux, les détails de la preuve de sécurité pour l'établissement de la racine de confiance statique sont fournis en annexe C.

5.1 Contexte

Nous proposons un prolongement des travaux de De Oliveira Nunes et al. : VRASED [36]. Pour des raisons de coûts et de performances, nous ciblons les microprocesseurs pris sur étagère. Pour des raisons de faisabilité, nous ciblions les *SoC* où le microprocesseur est accompagné d'un FPGA. Dans cette section, nous listons les problématiques liées au choix de ce système cible et nous en déduisons une définition de la sécurité de la fonction d'attestation.

5.1.1 Problématiques

La première problématique qui vient à l'esprit, lors d'une extension de l'attestation à distance aux microprocesseurs, est bien évidemment l'augmentation de la surface d'attaque. En effet, compte tenu de notre modèle de menaces, c'est-à-dire un modèle fort où toute lecture et écriture privilégiée est possible, alors **l'utilisation des périphériques du microprocesseur, ou leur reconfiguration, devient possible**. Cela inclut la reconfiguration de la MMU ou de *CoreSight*, que nous utilisons pour obtenir nos observables (voir la section 5.3.1), ainsi que l'utilisation des mémoires cache et les attaques par canaux auxiliaires. Également, dans le cas d'une implémentation matérielle dans la partie FPGA du *SoC*, il devient potentiellement possible pour un adversaire d'accéder au contenu de celui-ci.

La seconde problématique majeure concerne l'accès aux observables. Contrairement aux systèmes dont l'architecture est ouverte, il n'est pas possible de proposer une modification du cœur du microprocesseur Cortex-A9. De ce fait, **il n'est pas possible d'extraire les observables nécessaires à l'établissement de la racine de confiance de statique**, comme fait VRASED [36]. Il convient donc d'obtenir des observables différents et de s'assurer que ces nouvelles observations sont toujours suffisantes pour garantir la sécurité.

5.1.2 Définition de la sécurité

Nous prolongeons les travaux initiés par VRASED, notre définition de la sécurité considère donc un cas d'étude similaire, que nous pouvons résumer ainsi : lors de l'exécution de l'attestation à distance, le *vérifieur* va détecter toute intrusion par l'adversaire qui modifie l'environnement du *prouveur*. L'objectif de l'adversaire est donc de masquer cette modification de manière à ne pas être détectée lors de l'attestation à distance. Pour cela, l'adversaire doit forger un résultat au test d'intégrité authentifié. Nous envisageons deux méthodes pour cela :

— altérer l'exécution de la fonction d'attestation, par exemple en lui faisant attester une copie non-corrompue de l'environnement ;
— obtenir le secret et effectuer le calcul à la place de la fonction d'attestation.

Conformément à ces considérations, notre définition de la sécurité est une conjonction de deux propriétés :

— assurer l'exécution sécurisée de la fonction d'attestation ;
— maintenir la confidentialité du secret.

VRASED propose une définition formelle de la sécurité dans le cas où l'adversaire exécute du code malveillant sur un microcontrôleur simple. Cette définition se résume en quatre propriétés clés [36] :

P1. les accès au secret sont interdits lorsque le système n'est pas en train d'exécuter la fonction d'attestation (confidentialité vis-à-vis des accès non-concurrents) ;

P2. la fonction d'attestation s'exécute de manière atomique et ininterrompue, commence toujours par la première instruction et se termine toujours par la dernière (sûreté d'exécution de la fonction d'attestation) ;

P3. la fonction d'attestation n'écrit que dans des zones mémoires protégées (à l'exception du résultat au test d'intégrité authentifié) dont les accès sont interdits lorsque le système n'est pas en train d'exécuter cette même fonction d'attestation (protection contre les fuites) ;

P4. les accès directs à la mémoire (DMA, pour *Direct Memory Access*) depuis un périphérique, au secret et aux zones mémoires protégées, sont interdits (confidentialité vis-à-vis des accès concurrents).

Dans nos travaux, nous réutilisons la définition de la sécurité fournie par VRASED, que nous étendons pour tenir compte de l'augmentation de la surface d'attaque et des capacités d'un adversaire s'exécutant sur microprocesseur. Nous ajoutons donc quatre propriétés de sécurité supplémentaires. Ces propriétés sont choisies pour refléter quatre nouvelles capacités de l'adversaire que nous pouvons traiter de manière décorrélée. Selon nous, elles permettent d'assurer l'exécution sécurisée de la fonction d'attestation et maintenir la confidentialité du secret sur microprocesseur.

P5. un seul cœur du microprocesseur fonctionne durant l'exécution de la fonction d'attestation. Cette propriété est une extension à la confidentialité vis-à-vis des accès concurrents (**P4**). Dans le cas des systèmes multi-cœurs et durant l'exécution de la fonction d'attestation par un cœur, l'accès au secret doit être interdit par tous les cœurs restants.

P6. une configuration spécifique des périphériques du microprocesseur est un prérequis à l'exécution de la fonction d'attestation. Cette propriété est une extension à la sûreté

d'exécution (**P2**). En effet, dans le cas des microprocesseurs modernes, une configuration des périphériques tels que la MMU ou *CoreSight* est requise pour obtenir des observables (voir la section 5.5.1).

P7. l'utilisation des mémoires cache est désactivée durant l'exécution de la fonction d'attestation. Cette propriété est une extension à la sûreté d'exécution (**P2**) et à la protection contre les fuites (**P3**). En effet, les attaques par empoisonnement de cache ne doivent pas affecter l'exécution de la fonction d'attestation. Également, des attaques par canaux auxiliaires après son exécution ne doivent pas permettre l'extraction d'informations sur le secret.

P8. les accès en lecture et en écriture sur le contenu de la programmation du FPGA sont interdits. Cette propriété est une extension à la confidentialité vis-à-vis des accès concurrents (**P4**). En effet, la lecture des mémoires sensibles du périphérique de confiance, ou sa modification ne doivent pas être permis.

5.2 Stratégie de vérification envisagée

Notre objectif est de réutiliser au mieux les preuves de *soundness* et sécurité disponibles dans l'état de l'art. Les auteurs de VRASED ont démontré la sécurité de l'attestation à distance sur microcontrôleur dans le cas de l'exécution de primitives cryptographiques fournies par HACL⋆ [36, 23]. Dans cette section, nous décrivons la stratégie que nous envisageons : réutiliser les preuves apportées par ces travaux, adapter la vérification formelle et considérer une approche descendante pour l'extension aux microprocesseurs.

5.2.1 Positionnement

Les auteurs de VRASED proposent une stratégie de vérification que nous avons décrite dans la section 1.7. Ils admettent l'impossibilité du calcul du tag d'intégrité sans connaître le secret [37]. De plus, ils admettent que, sans accès au secret, la probabilité qu'un adversaire obtienne des informations sur celui-ci depuis la zone mémoire attestée et le résultat du calcul est négligeable [23]. Ils proposent une extension matérielle qui garantit qu'aucun accès (direct ou indirect) au secret n'est possible par l'adversaire et que l'exécution de la fonction d'attestation est sécurisée. Une preuve de sécurité leur permet d'établir une racine de confiance statique [36]. Comme nous utilisons la même bibliothèque cryptographique, nous pouvons admettre les mêmes hypothèses. **Nous pouvons donc réutiliser cette preuve sous réserve que nous parvenons aussi à garantir que, dans notre environnement, aucun accès au secret n'est possible par l'adversaire et que l'exécution de la fonction d'attestation est sécurisée**.

Pour apporter cette garantie, nous devons vérifier les propriétés de sécurité **P1** à **P8** définies dans la section 5.1.2. En plus de la preuve, les auteurs de VRASED fournissent également un moniteur matériel qui est formellement conforme aux propriétés **P1** à **P4** [36]. Plutôt que d'implémenter un nouveau moniteur matériel qui garantit les mêmes propriétés, nous choisissons de réutiliser ce même moniteur. Ainsi, la vérification formelle doit seulement être adaptée aux nouvelles contraintes.

Parmi ces contraintes, nous ne pouvons pas modifier le cœur du système et nous n'avons pas accès aux observables, qui constituent les variables d'entrée du moniteur que nous réutilisons.

Nous proposons donc de **déduire les valeurs de ces variables d'entrée à des étapes critiques de l'exécution de la fonction d'attestation, en observant les signaux fournis par l'interface de débug *CoreSight* et les signaux d'accès à des esclaves AXI-Lite**. Ensuite, nous adaptons la vérification formelle : nous établissons une preuve que cette déduction à certains instants est suffisante pour préserver les mêmes propriétés de sécurité (**P1** à **P4**).

5.2.2 Approche descendante

Pour construire notre périphérique de confiance, nous adoptons une approche ascendante : nous développons à un niveau proche du matériel (RTL), dont nous réalisons une abstraction pour obtenir un modèle haut niveau (SMV). Pour s'adapter à l'augmentation de la surface d'attaque et vérifier toutes les propriétés de sécurité nécessaires (**P1** à **P8**), la stratégie de vérification que nous choisissons est de considérer une approche descendante. C'est à dire que, pour modéliser le microprocesseur, nous concevons un modèle haut niveau que nous raffinons par la suite pour se rapprocher du comportement réel.

Dans un premier temps, nous considérons les microprocesseurs où l'on fait abstraction de certaines capacités de l'adversaire, comme par exemple utiliser les mémoires caches ou reconfigurer la MMU. Nous appelons ces abstractions des *hypothèses simplifiantes* pour le système. Nos premières hypothèses simplifiantes sont choisies telles que la surface d'attaque soit identique à celle considérée par VRASED. En d'autres termes, nous considérons dans un premier temps que notre microprocesseur ne possède pas plus de capacités que le microcontrôleur *openMSP430*. Néanmoins, le cœur de celui-ci ne peut pas être modifié.

Nous appelons ce premier système abstrait le **modèle zéro** du microprocesseur : prouver la sécurité sur ce système revient à prouver la même spécification que celle fournie par VRASED (propriétés **P1** à **P4**). Notre objectif est donc simplement de vérifier les mêmes propriétés locales puisque la preuve est déjà fournie par les auteurs [36]. Notre choix d'architecture doit permettre d'**obtenir ces mêmes propriétés locales sans modification du cœur du système**.

Ensuite, nous ajoutons une contrainte supplémentaire liée à l'augmentation de la surface d'attaque ; par exemple : "l'adversaire peut modifier la configuration des périphériques du microprocesseur avant l'exécution de la fonction d'attestation". Une propriété supplémentaire est alors ajoutée à la définition de la sécurité ; par exemple : "une configuration spécifique des périphériques du microprocesseur est obtenue avant l'exécution de la fonction d'attestation" (**P5**). Celle-ci est formellement exprimée et ajoutée à la spécification. Le système doit alors vérifier de nouvelles propriétés locales pour que la preuve reste valide. Nous nous assurons également que les propriétés précédentes sont toujours vérifiées.

Ces opérations sont **répétées pour toutes les contraintes apportées par une extension aux microprocesseurs et/ou une augmentation de la surface d'attaque**. À chaque itération, la définition de la sécurité est raffinée, le périphérique de confiance est étendu et une preuve est établie. Nous traitons ces itérations dans le chapitre 6 et l'annexe E de ce manuscrit.

5.3 Architecture proposée

Dans cette section, nous proposons une solution d'établissement de notre racine de confiance statique sans modification du cœur. Notre solution est basée sur la conception d'un périphérique de confiance, implémenté dans la partie FPGA du *SoC*, étroitement liée au microprocesseur. Son architecture est choisie telle que nous pouvons garantir les mêmes propriétés de sécurité que VRASED. Le *SoC* envisagé est un *Xilinx Zynq-7000*. Il est équipé d'un microprocesseur ARM Cortex-A9 mono-cœur et d'un FPGA Artix-7.

5.3.1 Périphérique de confiance

La région mémoire à attester contient un algorithme devant posséder les propriétés de sécurité suffisantes pour le *vérifieur*. Cet algorithme fonctionne de manière *stand-alone* : il est indépendant du reste du système. Il est chargé dans la DDR, dans une plage d'adresses physiques choisie et immuable. Si un système d'exploitation est présent, ce logiciel doit donc être indépendant de toute bibliothèque partagée et la distribution aléatoire de l'espace d'adressage (ASLR : pour *Address Space Layout Randomization*) ne doit pas être activée pour son exécution. Le challenge et le résultat du test d'intégrité sont également placés dans la DDR, dans une plage d'adresses physiques choisie et immuable. Le schéma représenté sur la figure 5.1 montre une vue d'ensemble du système.

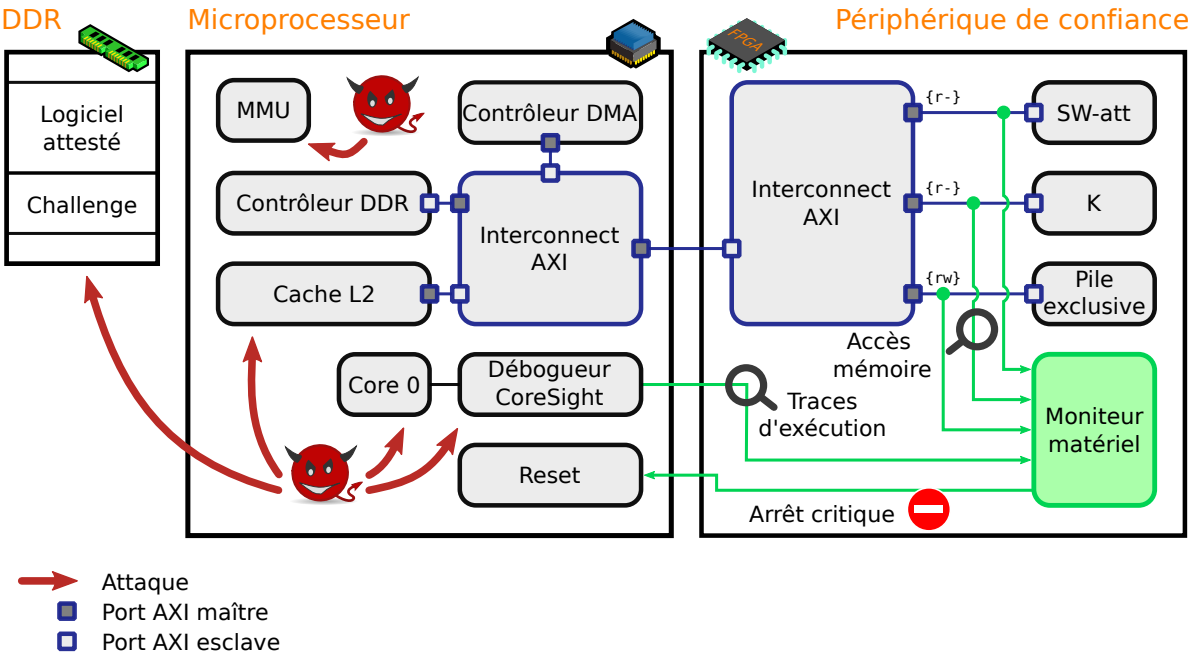

FIGURE 5.1 – Vue d'ensemble du système

Nous implémentons notre périphérique de confiance dans la partie FPGA du *SoC*. Nous procédons d'abord à une ségrégation spatiale du contenu sensible. C'est-à-dire que nous créons trois zones mémoires protégées :

— deux mémoires de type ROM, accessibles en lecture seule, contenant respectivement le code de la fonction d'attestation (`SW-att`) et le secret (`K`). L'aspect lecture seule de la

ROM sous-entend que le contenu de ces mémoires est écrit lors de la mise en production et est immuable.

— une mémoire de type RAM, accessible en lecture et écriture, qui est utilisée comme pile d'exécution exclusive à la fonction d'attestation. Ainsi, lorsque la fonction d'attestation s'exécute, les informations utilisées pour les calculs intermédiaires sont stockés dans une RAM dans le FPGA.

Ces trois zones mémoires protégées sont des esclaves AXI-Lite. Un esclave AXI-Lite est une variante d'un esclave AXI où le *burst* est interdit, ce qui signifie que tous les accès mémoire sont explicitement demandés adresse par adresse sur le bus AXI [60]. Ces mémoires sont accessibles par le microprocesseur au travers d'un module d'interconnexion AXI. Elles sont donc indépendantes du contrôleur DDR principal.

Ressources : le dossier `5_3_1_peripheral/` contient les sources pour implémenter du support matériel proposant une ségrégation spatiale. Il contient également les sources d'un FSBL et d'une application pour lire et écrire dans ces esclaves AXI-Lite depuis le microprocesseur. Un script gdb permet d'instrumenter openOCD pour la programmation. Un Makefile permet d'automatiser toutes ces tâches.

Ensuite, les zones mémoires protégées sont étendues avec un moniteur matériel. Ce moniteur matériel est un module qui observe les transactions sur le bus AXI. Une transaction a lieu lors d'un accès, par le microprocesseur, vers les zones mémoires protégées. Le moniteur stoppe l'exécution du microprocesseur, en lui transmettant un signal d'arrêt critique (*reset*), en cas de future compromission. Par exemple, le moniteur matériel transmet un *reset* au microprocesseur si celui-ci demande un accès au secret ou à la pile exclusive, et ce avant que l'esclave AXI-Lite ait répondu.

Le moniteur matériel est également connecté à l'interface de débug *CoreSight* pour garantir une exécution correcte de la fonction d'attestation. Des instructions spécifiques, des branchements indirects, sont ajoutés dans la fonction d'attestation pour obtenir la valeur du pointeur d'instruction à des instants critiques de l'attestation à distance. Également, lors d'une exception durant l'exécution de la fonction d'attestation, *CoreSight* informe le moniteur matériel d'une modification du flot d'exécution. Lorsque la fonction d'attestation s'exécute, le moniteur matériel interdit les exceptions et autorise les accès au secret / à la pile exclusive (ne transmet pas de *reset*). Un branchement indirect en début de fonction d'attestation provoque l'autorisation des accès. Un branchement indirect en fin de fonction d'attestation provoque de nouveau l'interdiction des accès.

Ressources : l'application dont les sources sont présentes dans le dossier `5_3_1_peripheral/` configure *CoreSight* et exécute des branchements indirects, puis affiche la trace d'exécution. Un script gdb permet d'instrumenter openOCD pour la programmation.

La fonction d'attestation est *bare-metal* : elle est indépendante du reste du système. Lors de son exécution, elle effectue une lecture de la région mémoire à attester et du challenge, dans la DDR, ainsi qu'une lecture du secret. La fonction d'attestation calcule alors un HMAC sur ces régions mémoire. Une fois le calcul terminé, le résultat est placé à l'adresse du challenge, dans la DDR. La fonction d'attestation procède à une remise à zéro des effets de bords du

microprocesseur avant et après son exécution : le contenu des registres est sauvegardé puis restauré après utilisation [23].

Une configuration de la MMU, réalisée avant l'exécution de la fonction d'attestation, restreint les accès en écriture. Les écritures ne sont possibles que dans la pile exclusive et à l'adresse du challenge.

Ressources : l'application dont les sources sont présentes dans le dossier `5_3_1_peripheral/` configure la MMU pour interdire les accès et exécute une écriture.
Un script gdb permet d'instrumenter openOCD pour la programmation.
Un Makefile permet d'automatiser la programmation et l'exécution.

5.3.2 Moniteur matériel

Le moniteur matériel est un sous-ensemble du périphérique de confiance, il garantit les propriétés locales nécessaires à l'établissement de la preuve de sécurité. Le choix de son architecture est adapté à la stratégie de vérification formelle que nous avons choisie et que nous détaillons dans la section 5.4.2. La figure 5.2 montre, en vert, l'architecture que nous avons choisie.

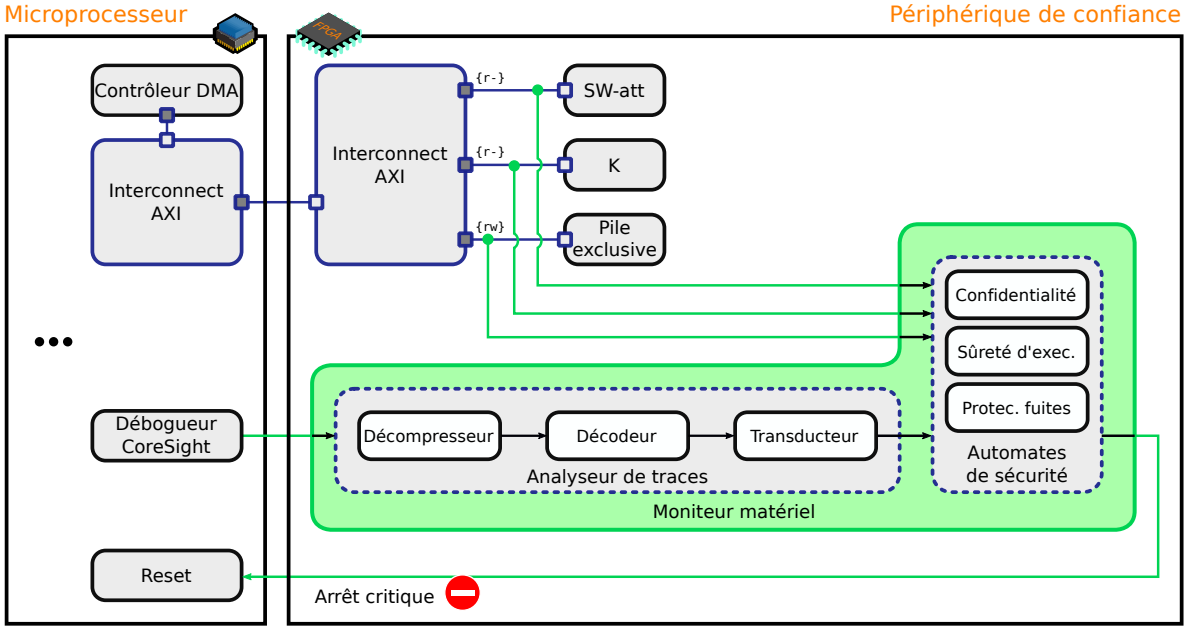

FIGURE 5.2 – Architecture du moniteur matériel

Nous pouvons séparer cette architecture en deux sous-systèmes :

— le premier est un ensemble d'automates de sécurité (à droite sur la figure 5.2). Leur rôle est de garantir des propriétés de sécurité. Ces automates sont réutilisés depuis le moniteur matériel de VRASED [36]. Certaines modifications sont néanmoins nécessaires afin d'adapter leurs variables d'entrée aux observables dont nous disposons. Ces adaptations sont discutées dans la section 5.4.2.

— le second est un analyseur de traces *CoreSight*. Son rôle est de fournir aux automates de sécurité un ensemble d'observables qui traduisent l'évolution du pointeur d'instruction

en fonction des informations de débug.

Le premier sous-système est composé de trois automates fonctionnant en parallèle, chacun dédié à la garantie de propriétés de sécurité telles que la confidentialité (**P1**), la sûreté d'exécution (**P2**) et la protection contre les fuites (**P3**). Nous ne réutilisons pas l'automate de VRASED qui garantit la confidentialité vis-à-vis des accès concurrents (**P4**). En effet, celui-ci contrôle les accès DMA aux zones mémoires protégées et nous choisissons de ne pas utiliser la fonctionnalité DMA au niveau des esclaves AXI-Lite. Nous maîtrisons les connexions au niveau du module d'interconnexion AXI esclave car celui-ci est implémenté dans le FPGA : nous n'implémentons simplement pas de module matériel pour permettre les accès DMA depuis un autre périphérique du microprocesseur.

Le second sous-système est composé de trois modules fonctionnant en série : un décompresseur, un décodeur et un transducteur. Le décompresseur et le décodeur extraient les informations des traces d'exécution transmises par *CoreSight*. Ces informations dépendent des branchements indirects exécutés et des levées d'exception. Elles constituent des observables nécessaires à la preuve de sécurité. Le transducteur traduit ces informations dans un alphabet adapté aux automates de sécurité. Il déduit, depuis les informations précédentes, la présence ou l'absence du pointeur d'instruction dans la fonction d'attestation et la levée d'exceptions.

5.4 Vérification formelle sur le modèle zéro

Comme décrit précédemment, nous adoptons une approche descendante pour la vérification. La première itération consiste à valider notre choix d'architecture : c'est-à-dire à construire un périphérique de confiance et établir une racine de confiance statique sans modification du cœur du microprocesseur. Nous validons donc ce choix en établissant une preuve de l'exécution sécurisée de la fonction d'attestation et du maintien de la confidentialité du secret sur notre modèle zéro. Dans cette section, nous listons les notations utilisées pour la modélisation, nous décrivons la procédure de vérification et nous définissons les hypothèses simplifiantes relatives au modèle zéro, afin d'exprimer formellement nos axiomes.

5.4.1 Notations

Nous réutilisons les notations employées par VRASED [36] afin de permettre un rejeu de la preuve sans modification. Les symboles suivants représentent des signaux internes au microprocesseur, ce sont des prédicats qui ne sont pas observables :

- PC (pour *Program Counter*) : valeur du pointeur d'instruction. C'est un vecteur de 32 bits.
- R_{en} (pour *Read Enable*) : signal de 1 bit indiquant qu'un accès mémoire en lecture est en cours.
- W_{en} (pour *Write Enable*) : signal de 1 bit indiquant qu'un accès mémoire en écriture est en cours.
- D_{addr} (pour *Data address*) : adresse à laquelle un accès mémoire est en cours si R_{en} ou W_{en} est levé.
- DMA_{en} (pour *DMA Enable*) : signal de 1 bit indiquant qu'un accès DMA est en cours.
- DMA_{addr} (pour *DMA address*) : adresse à laquelle un accès DMA est en cours si DMA_{en} est levé.

— *irq* (pour *Interuption ReQuest*) : signal de 1 bit indiquant qu'une exception matérielle est en cours.
— *reset* : signal de 1 bit qui provoque instantanément l'arrêt critique du microprocesseur. Ce signal est observable, il est accessible en écriture depuis la partie FPGA du *SoC*.

Pour tous ces prédicats, nous ajoutons un indice d pour signifier que la valeur du prédicat est déduite par notre moniteur matériel. Par exemple, PC_d représente la valeur déduite du pointeur d'instruction PC. Une valeur déduite n'est pas forcément égale à celle du prédicat, c'est à nous qu'il advient de prouver une équivalence à certains instants.

Les régions mémoire sont référencées par les acronymes suivants :

— KR (pour *Key Region*) : région mémoire dans laquelle le secret est stocké. Cette région est contenue dans un esclave AXI-Lite dédié, accessible en lecture seule.
— CR (pour *Critical Region*), la région critique : région mémoire dans laquelle le code de la fonction d'attestation est stocké. Cette région est contenue dans un esclave AXI-Lite dédié, accessible en lecture seule. L'adresse à laquelle est positionnée la première instruction de la fonction d'attestation (dans CR) est notée CR_{min}. L'adresse à laquelle est positionnée la dernière instruction est notée CR_{max}. Lorsque la configuration de *CoreSight* est telle qu'attendue par nos prérequis (**P6**), cette région CR est contenue dans la zone mémoire dans laquelle les traces sont actives.
— XS (pour *eXclusive Stack*) : région mémoire dans laquelle la pile exclusive est stockée. Cette région est contenue dans un esclave AXI-Lite dédié, accessible en lecture et en écriture.
— MR (pour *Memory Region*) : région mémoire dans laquelle la fonction d'attestation lit la valeur du challenge. C'est la même région mémoire où celle-ci écrit le résultat du test d'intégrité une fois son calcul terminé. Cette région est accessible dans la DDR.
— AR (pour *Attested Region*) : région mémoire à attester, sur laquelle la fonction d'attestation calcule le test d'intégrité. Cette région est accessible dans la DDR et/ou contient les registres des périphériques.

Pour simplifier la définition des prédicats, lors d'une comparaison de vecteurs, nous utilisons la notation d'égalité ou d'appartenance. Par exemple, l'expression $(PC \in CR)$, qui signifie que la valeur du pointeur d'instruction est incluse dans la région critique, est en réalité un simple prédicat pour l'assistant de preuve. Cela réduit des comparaisons de vecteurs de 32 bits à de simples booléens. De même, les expressions $(PC = CR_{min})$ et $(PC = CR_{max})$ sont aussi de simples prédicats.

Notons que dans ce chapitre, nous introduisons d'autres prédicats, utiles pour jouer les preuves. Leur signification est donnée lors de leur introduction, dans les sections suivantes.

5.4.2 Procédure de vérification formelle

Pour notre modèle zéro, nous devons prouver que notre choix d'architecture permet de vérifier les mêmes spécifications que VRASED. Le nombre de spécifications s'élève à 9 [36]. Elles constituent les hypothèses selon laquelle la sécurité est prouvée (**P1** à **P4**). Ces 9 spécifications sont exprimées à partir de prédicats qui ne sont pas observables. Nous les nommons avec la même nomenclature que VRASED [36] : LTL_2 à LTL_{10}. L'expression 5.1 représente ces spécifications.

$$
\begin{aligned}
LTL_2 &: \textbf{always}(\neg(PC \in CR) \wedge R_{en} \wedge (D_{addr} \in KR) \rightarrow reset) \\
LTL_3 &: \textbf{always}(\neg reset \wedge (PC \in CR) \wedge \textbf{next}(\neg(PC \in CR)) \rightarrow (PC = CR_{max} \vee \textbf{next}(reset))) \\
LTL_4 &: \textbf{always}(\neg reset \wedge \neg(PC \in CR) \wedge \textbf{next}((PC \in CR)) \rightarrow \textbf{next}(PC = CR_{min}) \vee \textbf{next}(reset)) \\
LTL_5 &: \textbf{always}(irq \wedge (PC \in CR) \rightarrow reset) \\
LTL_6 &: \textbf{always}(\neg(PC \in CR) \wedge (R_{en} \vee W_{en}) \wedge (D_{addr} \in XS) \rightarrow reset) \\
LTL_7 &: \textbf{always}((PC \in CR) \wedge W_{en} \wedge \neg(D_{addr} \in XS) \wedge \neg(D_{addr} \in MR) \rightarrow reset) \\
LTL_8 &: \textbf{always}(DMA_{en} \wedge (DMA_{addr} \in KR) \rightarrow reset) \\
LTL_9 &: \textbf{always}(DMA_{en} \wedge (DMA_{addr} \in XS) \rightarrow reset) \\
LTL_{10} &: \textbf{always}((PC \in CR) \wedge DMA_{en} \rightarrow reset)
\end{aligned} \tag{5.1}
$$

Afin de les vérifier, nous réutilisons les automates de sécurité fournis par VRASED. Seulement, dans notre réutilisation, les variables d'entrée de ces automates ne peuvent pas être connectées aux signaux internes du microprocesseur. Nous les renommons donc pour désigner des valeurs hypothétiquement déduites de ces signaux. La problématique est donc que ces automates de sécurité réutilisés vérifient des propriétés exprimées en fonction de ces valeurs déduites. Notre objectif est donc de réaliser une déduction correcte de ces valeurs, grâce à notre choix d'architecture, afin de vérifier les spécifications 5.1.

Pour illustrer cette problématique, nous prenons l'exemple de la spécification LTL_2. L'automate de sécurité qui vérifiait LTL_2, dans le cas de VRASED, vérifie pour nous la propriété suivante, que nous appelons P_0 :

$$
P_0 : \textbf{always}(\neg(PC_d \in CR) \wedge R_{en_d} \wedge (D_{addr_d} \in KR) \rightarrow reset) \tag{5.2}
$$

P_0 est identique à LTL_2 sauf que nous remplaçons les prédicats non-observables par leur valeur déduite. En langage naturel, P_0 décrit que, si nous déduisons que PC ne se trouve pas dans CR, et si nous déduisons qu'un accès en lecture est réalisé dans KR, alors l'automate provoque un *reset*.

Malheureusement, cette propriété à elle seule n'est pas suffisante pour garantir la spécification LTL_2. Certains axiomes sont nécessaires pour expliciter que nous déduisons correctement les prédicats non-observables. Nous définissons ces axiomes et, à partir de ceux-ci et de la propriété P_0, nous prouvons la spécification LTL_2. Cette preuve est réalisée avec l'assistant de preuves. L'expression 5.3 donne un exemple d'axiomes suffisants pour prouver la spécification LTL_2. Nous nommons la conjonction de ces axiomes A_0.

$$
\begin{aligned}
A_0 : \textbf{always}((PC_d \in CR) \leftrightarrow (PC \in CR)) \\
\wedge\ \textbf{always}(R_{en_d} \wedge (D_{addr_d} \in KR) \leftrightarrow R_{en} \wedge (D_{addr} \in KR))
\end{aligned} \tag{5.3}
$$

Ces axiomes A_0 expriment formellement que nous réalisons une déduction correcte lorsque le pointeur d'instruction se situe ou non dans la région critique, et lorsqu'un accès en lecture au secret est réalisé. Ils représentent une **obligation de preuve** pour le reste de notre moniteur matériel (autres automates de sécurité, transducteur, décodeur et décompresseur de traces).

Ressources : le dossier `5_4_2_ltl2/` contient le programme qui démontre que nous pouvons utiliser l'axiome A_0 et la propriété P_0 pour prouver le théorème LTL2. Cette preuve utilise notre bibliothèque Coq et la bibliothèque `LTL_Coq`.

Nous pouvons prouver une obligation de preuve à partir de :

— propriétés considérées axiomatiques ; ceci est le cas lorsqu'elles sont formalisées depuis la documentation du *SoC* et représentent le modèle de notre système.

— propriétés locales vérifiées par *model-checking* sur les autres sous-systèmes de notre moniteur matériel.

Dans le cas de l'obligation de preuve A_0, une modélisation des transactions sur le bus AXI, issue d'une traduction de la documentation, est suffisante pour démontrer que nous déduisons correctement un accès en lecture au secret. Des détails sur cette démonstration sont fournis dans la section 5.5.2.

Également, les propriétés locales vérifiées sur le transducteur sont nécessaires pour prouver que nous déduisons correctement que le pointeur d'instruction se situe ou non dans la région critique. Ces propriétés locales sont nécessaires mais ne sont pas forcément suffisantes. Certains axiomes peuvent rester nécessaires pour prouver notre obligation de preuve. Dans ce cas, nous pouvons appeler ces axiomes A_1 : ils forment de nouveau une obligation de preuve pour le reste du moniteur matériel (décodeur et décompresseur de traces). Nous suivons les mêmes étapes pour démontrer l'obligation de preuve A_1. Le schéma représenté sur la figure 5.3 résume la procédure de démonstration d'une propriété de sécurité.

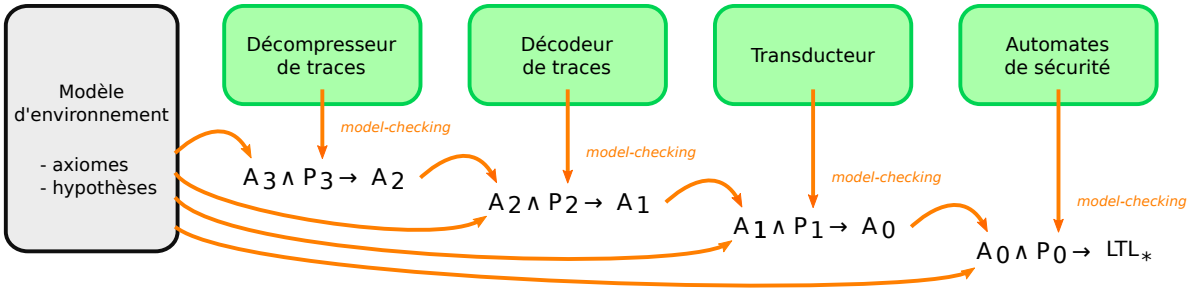

FIGURE 5.3 – Procédure de démonstration

La vérification se fait de droite à gauche :

1. les spécifications que nous souhaitons prouver sont identiques aux propriétés de sécurité garanties par les automates de VRASED. Nous utilisons les mêmes automates qui nous assurent des propriétés exprimées en fonction de prédicats différents (valeurs déduites).
2. Des propriétés P_0 sont vérifiées par *model-checking* sur l'automate. Des axiomes A_0 sont nécessaires pour que la propriété de sécurité soit toujours garantie.
3. Vérifier l'implication des axiomes A_n par la conjonction de propriétés P_{n+1} et d'axiomes A_{n+1} constitue une obligation de preuve.
4. Les propriétés P_{n+1} sont vérifiées par *model-checking* sur les modèles des sous-systèmes précédents. Des axiomes A_{n+1} sont nécessaires. L'obligation de preuve A_n est prouvée.
5. Les étapes 3 et 4 sont répétées en incrémentant n jusqu'à ce que l'obligation de preuve soit une tautologie $(A_n \rightarrow A_n)$. Dans ce cas, la preuve est terminée car les axiomes A_n sont tous issus d'une traduction de la documentation du *SoC*.

Cette procédure est appliquée à la preuve de toutes les spécifications représentées par l'expression 5.1. Des détails sur l'obtention de la spécification LTL_2 sur notre système sont donnés dans la section 5.5.

5.4.3 Hypothèses simplifiantes

Afin de prouver la sécurité sur notre modèle zéro, nous émettons des hypothèses simplifiantes. Ces hypothèses influent sur la définition des axiomes que nous traduisons depuis la documentation du *SoC*. Dans cette section, nous listons les hypothèses simplifiantes, justifions leur considération et exprimons leur impact sur l'expression formelle de nos axiomes.

Nos hypothèses simplifiantes ont pour objectif de relâcher les exigences de la preuve sur le système afin de la rendre réalisable immédiatement sur le modèle zéro. Nous émettons une hypothèse simplifiante pour chacune des propriétés **P5** à **P8** décrites dans la section 5.1.2. Nous nommons donc nos hypothèses simplifiantes **H5** à **H8** pour faire écho à ces propriétés. Une hypothèse **H9** est également définie mais n'a pas de lien avec les propriétés **P5** à **P8** : son objectif est de simplifier la preuve en s'affranchissant des temps de transmission des paquets de *CoreSight*. La liste des hypothèses simplifiantes que nous considérons sur notre modèle zéro est la suivante :

- **H5** : un seul cœur du microprocesseur fonctionne durant l'exécution de la fonction d'attestation.
- **H6** : la configuration des périphériques est dans un état défini et non modifiable par l'adversaire avant l'exécution de la fonction d'attestation.
- **H7** : les mémoires cache sont désactivées avant l'exécution des premières instructions de la fonction d'attestation.
- **H8** : l'adversaire n'accède pas au périphérique de programmation du FPGA.
- **H9** : les paquets *isync* et *branch* émis par *CoreSight* sont disponibles à l'état précédant l'exécution de l'instruction de destination.

Notons que nous supposons pouvoir décharger notre périphérique de confiance de ces hypothèses simplifiantes lors des itérations suivantes du modèle. Le chapitre 6 et l'annexe E traitent de la validité de ces suppositions.

5.4.3.1 H5 : un seul cœur du microprocesseur fonctionne durant l'exécution de la fonction d'attestation

Le moniteur matériel garantit les contrôles d'accès au secret (**P1**). Les accès aux esclaves AXI-Lite sont autorisés lorsque le pointeur d'instruction se situe dans la région critique ($PC \in CR$). Dans la vérification du modèle zéro, nous émettons l'hypothèse qu'un seul cœur du microprocesseur fonctionne durant l'exécution de la fonction d'attestation.

Cette hypothèse est trivialement vérifiée sur notre *SoC*, dont le microprocesseur ne possède qu'un seul cœur. Dans le cas général (pour tout microprocesseur), nous considérons qu'il est possible de vérifier cette hypothèse en désactivant le fonctionnement des autres cœurs avant l'exécution de la fonction d'attestation (**H6**).

Cette hypothèse **H5** se traduit par l'unicité du prédicat modélisant la valeur du pointeur d'instruction PC. En effet, dans le cas où plusieurs cœurs seraient utilisés en même temps, il serait nécessaire de modéliser plusieurs pointeurs d'instruction (PC_1, PC_2, etc.). Une conséquence directe de cette hypothèse est que nous pouvons bien considérer l'expression

($PC \in CR$) comme un prédicat : le microprocesseur ne peut pas avoir un pointeur d'instructions à plusieurs adresses différentes (dans CR et hors de CR). De la même manière, nous pouvons bien considérer que le pointeur d'instruction ne peut pas se trouver à la fois à l'adresse CR_{min} et à l'adresse CR_{max} (sous réserve que les adresses CR_{min} et CR_{max} soient distinctes). Par exemple, nous pouvons considérer axiomatique la propriété suivante :

$$\textbf{always}((PC = CR_{min}) \leftrightarrow \neg(PC = CR_{max}))$$

5.4.3.2 H6 : la configuration des périphériques est dans un état défini et non modifiable par l'adversaire avant l'exécution de la fonction d'attestation

Les propriétés de sécurité sont garanties par le moniteur matériel à partir des traces d'exécution fournies par *CoreSight*. Également, une configuration de la MMU spécifique à l'exécution de la fonction d'attestation restreint les accès en écriture. Dans la vérification du modèle zéro, nous émettons l'hypothèse que *CoreSight* fournit les traces d'exécution et les adresses de destination attendues lors de branchements indirects. Également, nous émettons l'hypothèse que la configuration de la MMU restreint les accès en écriture de la fonction d'attestation aux régions XS et MR. Nous considérons aussi que tous les accès DMA sont désactivés lors de l'exécution de la fonction d'attestation.

Nous supposons qu'il est possible d'exécuter un algorithme, avant la fonction d'attestation, qui configure les périphériques pour satisfaire ces hypothèses et que l'adversaire ne peut pas interrompre.

Cette hypothèse **H6** nous permet d'exprimer des axiomes comme, par exemple, le fait que *CoreSight* transmette un paquet *branch* lors d'un branchement indirect ou d'une exception dans la région critique. Des détails sur la modélisation formelle de la transmission de paquets par *CoreSight* sont donnés dans la section 5.5.1. Pour la configuration de la MMU qui restreint les accès, nous considérons par exemple axiomatique la propriété suivante :

$$\textbf{always}((PC \in CR) \wedge W_{en} \rightarrow (D_{addr} \in XS) \vee (D_{addr} \in MR))$$

De manière analogue, pour les accès DMA, nous considérons axiomatique la propriété suivante :

$$\textbf{always}((PC \in CR) \rightarrow \neg DMA_{en})$$

5.4.3.3 H7 : les mémoires cache sont désactivées avant l'exécution des premières instructions de la fonction d'attestation

Le moniteur matériel garantit une protection contre les fuites (**P3**). Les attaques par canaux auxiliaires depuis les mémoires cache permettent une extraction d'information pour un adversaire non privilégié [88]. Dans la vérification de notre modèle zéro, nous émettons l'hypothèse que les mémoires cache sont désactivées lors de l'exécution de la fonction d'attestation, de sorte que son exécution n'écrive aucune donnée en cache.

Tout comme l'hypothèse simplifiante **H6**, nous supposons qu'il est possible d'exécuter un algorithme, avant la fonction d'attestation et que l'adversaire ne peut pas interrompre. Nous considérons que nous pouvons désactiver les mémoires cache lors de l'exécution de cet algorithme.

Cette hypothèse **H7** permet de considérer que toute lecture dans les zones mémoires protégées se traduit par un accès sur le bus AXI. Nous pouvons donc considérer qu'un accès en lecture est visible par le moniteur matériel, même après une exécution de la fonction d'attestation. De ce fait, si nous considérons que les prédicats qui modélisent la valeur des signaux $ARVALID$ des esclaves AXI-Lite, contenant la pile exclusive et le secret, se nomment respectivement $XS\_axi\_arvalid$ et $KR\_axi\_arvalid$, nous pouvons traduire la documentation du module d'interconnexion AXI vers les propriétés suivantes :

$$\begin{aligned} &\textbf{always}(R_{en} \wedge (D_{addr} \in XS) \leftrightarrow XS\_axi\_arvalid) \\ &\textbf{always}(R_{en} \wedge (D_{addr} \in KR) \leftrightarrow KR\_axi\_arvalid) \end{aligned} \tag{5.4}$$

5.4.3.4 H8 : l'adversaire n'accède pas au périphérique de programmation du FPGA

Notre moniteur matériel assure des propriétés de sécurité que nous vérifions par *model-checking*. Toutes ces propriétés sont vérifiées dès lors que celui-ci est bien implémenté dans le FPGA. Également, la ségrégation spatiale des mémoires sensibles permet d'observer les accès en lecture réalisés par le microprocesseur lorsque celui-ci y accède par le bus AXI. Dans la vérification du modèle zéro, nous émettons l'hypothèse que l'adversaire ne reprogramme pas le FPGA et que les propriétés que nous vérifions par *model-checking* sont des hypothèses valides. De plus, nous émettons l'hypothèse que l'adversaire n'accède pas au contenu des mémoires sensibles depuis un accès en lecture à la programmation du FPGA.

Nous supposons qu'il est possible de détecter un accès en lecture ou en écriture au contenu du FPGA depuis celui-ci. Ainsi, nous considérons que nous pouvons réaliser un périphérique de confiance qui transmet un signal d'arrêt critique (*reset*) au microprocesseur en cas de futur accès.

Cette hypothèse **H8** nous permet de considérer comme des hypothèses valides toutes les propriétés locales que nous vérifions par *model-checking* sur notre moniteur matériel.

5.4.3.5 H9 : les paquets *isync* et *branch* émis par *CoreSight* sont disponibles à l'état précédant l'exécution de l'instruction de destination

Lors d'une entrée du pointeur d'instruction dans une région mémoire où *CoreSight* est configurée pour tracer le flot d'exécution, un paquet *isync* est émis par *CoreSight*. De la même manière, lors de l'exécution d'un branchement indirect par le microprocesseur, si celui-ci se trouve dans une région mémoire où *CoreSight* est configurée pour tracer le flot d'exécution, un paquet *branch* est émis. Ces deux paquets contiennent l'adresse de destination du pointeur d'instruction [59]. La transmission de ces paquets sur le TPIU se fait en série et prend plusieurs cycles d'horloge [62]. Dans la vérification du modèle zéro, nous émettons l'hypothèse que l'émission des paquets est immédiate et nous faisons abstraction du temps de transmission. En d'autres termes, nous considérons que le pointeur d'instruction atteint l'adresse de destination présente dans le paquet après la fin de la transmission.

Une telle hypothèse n'est pas réaliste si nous considérons la documentation des périphériques [59, 62]. Néanmoins, nous supposons qu'une extension supplémentaire du périphérique de confiance peut permettre de verrouiller l'exécution de la fonction d'attestation (empêcher le microprocesseur d'en sortir) puis de la déverrouiller une fois un paquet entièrement transmis au moniteur matériel.

Cette hypothèse **H9** permet de considérer que, lorsqu'un paquet est émis par *CoreSight*, son dernier octet est reçu par le décompresseur. De ce fait, si nous considérons que les prédicats $(available\_packet = branch)$ et $(available\_packet = isync)$ indiquent respectivement que *CoreSight* émet un paquet *branch* ou un paquet *isync*, que le signal *ready* est transmis par le décompresseur à la réception du dernier octet d'un paquet, alors nous pouvons considérer axiomatique les propriétés 5.5 et 5.6 suivantes :

$$\mathbf{always}((available\_packet = branch) \rightarrow ready) \tag{5.5}$$

$$\mathbf{always}((available\_packet = isync) \rightarrow ready) \tag{5.6}$$

5.5 Théorème : détails de la vérification

Prouver la sécurité sur notre modèle zéro consiste à prouver les 9 spécifications représentées par l'expression 5.1. Nous avons donc établi 9 théorèmes, un par spécification. Dans cette section, nous détaillons l'établissement d'un théorème, celui qui démontre la protection contre les accès directs au secret (spécification LTL_2). Établir un théorème pour une autre spécification est un travail analogue, où nous pouvons réutiliser les lemmes démontrés dans cette section. Les preuves de l'établissement de tous les théorèmes sont disponibles publiquement et distribuées sous licence libre [1].

Cette section est divisée en cinq parties. Tout d'abord, nous définissons nos axiomes, exprimés depuis la documentation et l'architecture logicielle que nous avons choisie pour le contenu de l'esclave AXI-Lite (où la fonction d'attestation est stockée). Ensuite, pour chacun des quatre systèmes matériels (automate de sécurité réutilisé, transducteur, décodeur et décompresseur de traces), nous listons les propriétés obtenues par *model-checking*, définissons les obligations de preuves résultantes et détaillons la stratégie de preuve pour les obligations de preuve précédentes. L'annexe C fournit des détails techniques sur l'architecture matérielle et les propriétés locales vérifiées par chaque système.

5.5.1 Axiomes : modèle d'environnement

Sans accès aux sources du microprocesseur et de ses périphériques, nous émettons l'hypothèse que ceux-ci fonctionnent comme décrits dans la documentation. Nous traduisons la documentation en propriétés temporelles, que nous considérons axiomatiques. Les preuves de nos théorèmes reposent sur ces axiomes. De plus, le choix de notre architecture, de la configuration spécifique des périphériques et nos hypothèses simplifiantes nous permet de définir de nouveaux axiomes. Dans cette section, nous définissons ces axiomes et justifions leur existence par des choix de conception.

5.5.1.1 Etat initial

Lorsque le *SoC* est mis sous tension, ou à la réception d'un signal d'arrêt critique, nous considérons que la séquence d'initialisation commence par un FSBL qui ne peut être modifié par l'adversaire. Ce choix se justifie par la possibilité d'avoir recours à la cryptographie pour réaliser une initialisation d'images (FSBL et *bitstream*) chiffrées et authentifiées [50] et que notre adversaire ne possède pas d'accès physique au *SoC*.

En conséquence, nous considérons axiomatique que notre système ne démarrera pas en plaçant PC dans CR. De plus, nous considérons axiomatique qu'à l'état initial, nous avons le même comportement qu'après un arrêt critique. Les expressions 5.7 et 5.8 représentent de tels axiomes.

$$\mathbf{always}(reset \rightarrow \neg(PC \in CR)) \tag{5.7}$$

$$reset \tag{5.8}$$

5.5.1.2 Fonctionnement du matériel

Lorsque le *SoC* est mis sous tension, notre FSBL programme le FPGA avec le *bitstream* qui contient notre périphérique de confiance tel que défini dans la section 5.3.1. Par conséquent, les propriétés locales que nous vérifions sur notre moniteur matériel sont considérées comme des hypothèses valides.

De plus, nous utilisons pour le FPGA, une horloge fournie par un quartz externe. Cette horloge est active dès la mise sous tension et ne peut pas être arrêtée. Nous appelons *clk* le prédicat qui représente qu'une horloge est active. Nous considérons donc axiomatique que ce prédicat soit toujours actif. L'expression 5.9 représente cet axiome.

$$\mathbf{always}(clk) \tag{5.9}$$

5.5.1.3 Modélisation de *CoreSight* et des branchements indirects

La configuration de *CoreSight* permet de définir la plage d'adresses à tracer. Nous choisissons cette plage d'adresses telle que la région critique est incluse dans celle-ci. Les seules instructions présentes dans cette plage d'adresses et absentes de la région critique sont celles qui permettent un branchement indirect vers CR_{min} et celle située à CR_{max}. Cette plage d'adresses est localisée dans l'esclave AXI-Lite où la fonction d'attestation est stockée.

Dans cette plage d'adresse, il y a deux branchements indirects. Le premier mène à CR_{min}, c'est-à-dire à la première instruction de la région critique. Le second est localisé à la dernière instruction de la région critique. Il mène à CR_{max}, c'est-à-dire qu'il permet d'en sortir. Les instructions situées dans la région critique ne contiennent aucun branchement indirect, à l'exception de la dernière que nous avons ajoutée. La figure 5.4 représente l'architecture logicielle dans cet esclave AXI-Lite. La plage d'adresses tracée est représentée en vert tandis que la région critique est représentée en rouge. Les adresses CR_{min} et CR_{max} sont pointées en bleu sur la figure.

Ressources : le dossier `5_6_2_soundness/` contient les sources pour implémenter un esclave AXI-Lite contenant un programme. Il contient également les sources d'un FSBL et d'une application pour configurer *CoreSight* et exécuter ce programme. Des branchements indirects sont présents, la trace d'exécution peut être visualisée. Un script gdb permet d'instrumenter openOCD pour la programmation.

Notre configuration de *CoreSight* et notre ajout de branchements indirects assurent que toute entrée du pointeur d'instruction dans la région critique provoque une émission d'un paquet de type *isync* ou *branch*. Dans ces paquets, l'adresse de destination est localisée dans la région critique ($packet\_address \in CR$). Ce comportement se traduit par l'axiome A_{enter} (expression 5.10).

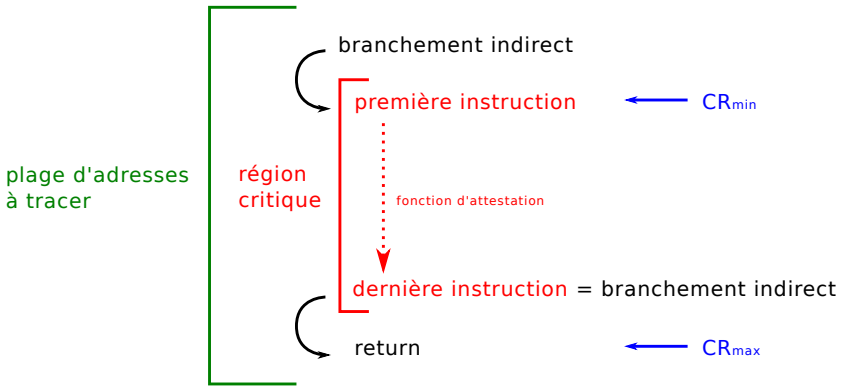

FIGURE 5.4 – Architecture logicielle dans l'esclave AXI-Lite

$$A_{enter} : \textbf{always}(\quad (5.10)$$
$$(\neg(PC \in CR) \wedge \textbf{next}(PC \in CR))$$
$$\rightarrow$$
$$(((available\_packet = isync) \vee (available\_packet = branch)) \wedge (packet\_address \in CR))$$
$$)$$

Également, comme les instructions de la région critique ne contiennent aucun branchement indirect et que la plage d'adresse est continue, *CoreSight* n'émet aucun paquet de type *isync* ou *branch* tant que le pointeur d'instruction est localisé dans la région critique (sauf dans le cas où une exception est levée). Ce comportement est exprimé par l'axiome A_{in} (expression 5.11).

$$A_{in} : \textbf{always}(\quad (5.11)$$
$$((PC \in CR) \wedge \textbf{next}(PC \in CR) \wedge \neg irq)$$
$$\rightarrow$$
$$\neg((available\_packet = isync) \vee (available\_packet = branch))$$
$$)$$

L'ajout de branchements indirects assure également que toute sortie du pointeur d'instruction de la région critique provoque une émission d'un paquet de type *branch*, que ce soit une sortie par la dernière instruction ou par la levée d'une exception. Ce comportement est exprimé par l'axiome A_{exit} (expression 5.12).

$$A_{exit} : \textbf{always}(\quad (5.12)$$
$$((PC \in CR) \wedge \textbf{next}(\neg(PC \in CR)))$$
$$\rightarrow$$
$$((available\_packet = branch) \wedge \neg(packet\_address \in CR))$$
$$)$$

Pour finir, la plage d'adresses à tracer contient la région critique et deux instructions

supplémentaires. Étant donné que la région critique est *bare metal*, aucune trace n'est émise si le pointeur d'instruction se situe hors de la région critique et n'y entre pas. Ce comportement est exprimé par l'axiome A_{out} (expression 5.13).

$$\begin{aligned} A_{out} : \ & \textbf{always}(\\ & \quad (\neg(PC \in CR) \wedge \textbf{next}(\neg(PC \in CR))) \\ & \rightarrow \\ & \quad \neg((available\_packet = isync) \vee (available\_packet = branch)) \\ &) \end{aligned} \tag{5.13}$$

5.5.2 Propriétés P_0 et obligations de preuve A_0 : automates de sécurité

L'automate de sécurité que nous réutilisons vérifie, sans modification, la propriété locale P_0 représentée par l'expression 5.2. Afin de vérifier la spécification LTL_2, nous devons prouver l'obligation de preuve A_0 représentée par l'expression 5.3. Cette obligation de preuve est une conjonction de deux expressions, que nous traitons indépendamment. Les expressions 5.14 et 5.15 représentent les deux lemmes formant notre obligation de preuve.

$$L_{0_A} : \textbf{always}((PC_d \in CR) \leftrightarrow (PC \in CR)) \tag{5.14}$$

$$L_{0_B} : \textbf{always}(R_{en_d} \wedge (D_{addr_d} \in KR) \leftrightarrow R_{en} \wedge (D_{addr} \in KR)) \tag{5.15}$$

Il est possible que certains lemmes puissent être prouvés facilement, si nous modifions légèrement les automates fournis par VRASED. C'est le cas du lemme L_{0_B}. En effet, nous pouvons étendre l'automate de sécurité pour déduire les prédicats R_{en_d} et D_{addr_d} depuis une autre source. Nous émettons l'hypothèse que nous pouvons déduire un accès en lecture du secret lorsque le signal $ARVALID$ de l'esclave AXI-Lite contenant le secret est actif. À la place, nous observons donc le prédicat $KR\_axi\_arvalid$. Cette hypothèse se traduit par une équivalence comme montré par l'expression 5.16.

$$\textbf{always}(R_{en_d} \wedge (D_{addr_d} \in KR) \leftrightarrow KR\_axi\_arvalid) \tag{5.16}$$

Or, lorsque le signal $ARVALID$ de l'esclave AXI-Lite contenant le secret est actif, nous avons bien une équivalence avec un accès en lecture au secret. Ceci est défini par l'axiome représenté par l'expression 5.4, conséquence de notre hypothèse simplifiante **H7**. À partir de cet axiome, nous pouvons prouver le lemme L_{0_B} simplement.

Une autre manière de raisonner est de modifier l'automate de sécurité pour observer directement le prédicat $KR\_axi\_arvalid$. Dans ce cas, nous vérifions par *model-checking* que l'automate modifié vérifie la propriété locale P_0' suivante :

$$P_0' : \textbf{always}(\neg(PC_d \in CR) \wedge KR\_axi\_arvalid \rightarrow reset)$$

Notre obligation de preuve A_0 est alors modifiée et le lemme L_{0_B} nécessaire pour la prouver devient identique à l'axiome représenté par l'expression 5.4. La preuve de L_{0_B} est immédiate.

Ressources : le dossier `5_5_2_automate_securite/` contient la description de l'automate de sécurité modifié, en langage *Verilog*, ainsi que la propriété locale P'_0 décrite en langage SMV. Notre méthode de vérification formelle automatisée permet de vérifier que la propriété P'_0 est vérifiée par l'automate de sécurité modifié.
Un `Makefile` permet d'automatiser la vérification formelle.

Dans d'autres cas, prouver un lemme nécessaire à notre obligation de preuve A_0 peut être un travail difficile. Ceci est le cas du lemme L_{0_A} : nous devons prouver que nous déduisons correctement la présence du pointeur d'instruction dans la région critique. Les sections suivantes décrivent la procédure pour prouver le lemme L_{0_A} à partir de propriétés locales, vérifiées par *model-checking*. Notons, qu'une fois prouvé, L_{0_A} peut être réutilisé pour la preuve des autres spécifications (5.1) qui utilisent le prédicat $(PC \in CR)$.

5.5.3 Propriétés P_1 et obligations de preuve A_1 : transducteur

Le transducteur vérifie des propriétés locales P_1, qui ne sont pas suffisantes pour prouver le lemme L_{0_A}. La preuve de L_{0_A} nécessite donc d'établir des obligations de preuves A_1 pour le décodeur et le décompresseur. Dans cette section, nous listons les propriétés P_1 qui sont vérifiées par *model-checking* sur le transducteur, nous en déduisons des lemmes intermédiaires qui sont suffisants pour prouver L_{0_A} et des obligations de preuves A_1 pour le reste du moniteur. La section C.1 de l'annexe C revient sur la démonstration et fournit des détails techniques concernant le transducteur et sa vérification.

5.5.3.1 Propriétés locales P_1

Afin de prouver notre lemme L_{0_A}, nous utilisons les propriétés locales vérifiées par *model-checking* sur le transducteur. La conjonction de ces propriétés locales forme P_1 (figure 5.3). Afin de décrire ces propriétés locales et expliciter leur impact sur notre stratégie de preuve, nous devons fournir plus de précisions sur le fonctionnement du transducteur.

Le transducteur est un automate qui mémorise la valeur du prédicat $(PC_d \in CR)$:

- si le décompresseur n'indique pas que la donnée d'un paquet *isync* ou *branch* est prête (*ready*), alors le transducteur ne modifie pas la valeur de ce prédicat.
- dans le cas où la donnée est prête, si le décodeur fournit une adresse située dans CR, alors le décompresseur vérifie ce prédicat.
- également, si le décodeur fournit une adresse située hors de CR, alors le décompresseur falsifie ce prédicat.

Conformément à ce comportement, nous exprimons formellement 5 propriétés locales que nous vérifions sur le transducteur. Nous nommons ces 5 propriétés locales P_{enter}, P_{in}, P_{exit}, P_{out} et P_{start}. Leur expression formelle est décrite en annexe C. Ces propriétés locales expriment que :

P_{enter} : $(PC_d \in CR)$ est vérifié au prochain état du système lorsque nous recevons un paquet contenant une adresse située dans CR (cas où PC_d entre dans CR).

P_{in} : si $(PC_d \in CR)$ est vérifié et que nous ne recevons pas un paquet contenant une adresse située hors de CR, alors il reste vérifié (cas où PC_d est situé dans CR).

P_{exit} : $(PC_d \in CR)$ est falsifié au prochain état du système lorsque nous recevons un paquet contenant une adresse située hors de CR (cas où PC_d sort de CR).

P_{out} : si $(PC_d \in CR)$ est falsifié et que nous ne recevons pas un paquet contenant une adresse située dans CR, alors il reste falsifié (cas où PC_d est situé hors de CR).

P_{start} : à l'état initial du système, $(PC_d \in CR)$ est falsifié.

Ressources : le fichier `5_5_3_mc_transducteur/transducer.spec.smv` contient la définition des 5 propriétés locales en PSL. Notre méthode de vérification formelle automatisée de vérifier, par *model-checking*, que le transducteur vérifie ces propriétés.
Un `Makefile` permet d'automatiser cette vérification.

5.5.3.2 Lemmes intermédiaires

Prouver le lemme L_{0_A} depuis les propriétés locales P_1 est un travail difficile. Afin de simplifier la preuve, nous proposons de prouver quatre lemmes intermédiaires, adaptés aux propriétés locales P_1. Nous nommons respectivement ces quatre lemmes intermédiaires L_{enter}, L_{in}, L_{exit} et L_{out}. Ces quatre lemmes intermédiaires forment des hypothèses nécessaires et suffisantes pour prouver L_{0_A}. Leur expression formelle est décrite par les expression 5.17, 5.18, 5.19 et 5.20 ci-après.

$$L_{enter} : \mathbf{always}(\neg(PC \in CR) \wedge \mathbf{next}(PC \in CR) \rightarrow \mathbf{next}(PC_d \in CR)) \quad (5.17)$$

$$L_{in} : \mathbf{always}((PC_d \in CR) \wedge \mathbf{next}(PC \in CR) \rightarrow \mathbf{next}(PC_d \in CR)) \quad (5.18)$$

$$L_{exit} : \mathbf{always}((PC \in CR) \wedge \mathbf{next}(\neg(PC \in CR)) \rightarrow \mathbf{next}(\neg(PC_d \in CR))) \quad (5.19)$$

$$L_{out} : \mathbf{always}(\neg(PC_d \in CR) \wedge \mathbf{next}(\neg(PC \in CR)) \rightarrow \mathbf{next}(\neg(PC_d \in CR))) \quad (5.20)$$

Nous prouvons ces quatre lemmes à partir des quatre premières propriétés locales vérifiées sur le transducteur : nous prouvons L_{enter} à partir de la propriété P_{enter}, nous prouvons L_{in} à partir de la propriété P_{in}, etc. Également, des axiomes et des obligations de preuve sont nécessaires à la démonstration, ils forment A_1 (figure 5.3).

Les axiomes nécessaires résultent de notre modélisation du fonctionnement du matériel, par exemple : *clk* est toujours actif (expression 5.9). Également, ils modélisent le comportement de *CoreSight* et des branchements indirects : l'axiome A_{enter} permet de prouver L_{enter} (expression 5.10), l'axiome A_{in} permet de prouver L_{in} (expression 5.11), etc. Dans le cas où la preuve se base sur une émission de paquet par *CoreSight* (lemmes L_{enter} et L_{exit}), nous utilisons notre hypothèse simplifiante **H9** pour abstraire le temps de transmission (expressions 5.5 et 5.6).

Les obligations de preuve nécessaires nous permettent de déduire les propriétés locales à vérifier sur le reste du moniteur matériel (décodeur et décompresseur de traces). Dans la suite de cette section, nous nous focalisons sur la preuve du lemme intermédiaire L_{enter} et des obligations de preuve A_1 résultantes. La même stratégie est appliquée aux autres lemmes intermédiaires. L'annexe C détaille la preuve pour les quatre lemmes intermédiaires.

5.5.3.3 Obligations de preuve A_1

L'axiome 5.10 permet de prouver L_{enter} (5.17). Il est exprimé en fonction des prédicats *available_packet* et *packet_address*. Ces prédicats représentent respectivement le type de paquet **émis par *CoreSight*** et la valeur de l'adresse **contenue dans celui-ci**. La première propriété locale que nous vérifions sur le transducteur permet aussi de prouver L_{enter}. Elle est exprimée en fonction de prédicats qui représentent le type de paquet **identifié par le décompresseur** et l'adresse **décodée par le décodeur**.

Les obligations de preuve nécessaires, A_1, doivent faire le lien entre les paquets émis par *CoreSight* et la donnée fournie par le décompresseur et le décodeur de traces. Elles expriment donc un décodage correct des paquets décompressés et une décompression correcte des paquets émis par *CoreSight*.

Nous pouvons définir quatre obligations de preuves, suffisantes pour prouver le lemme intermédiaire L_{enter}. Deux sont à démontrer depuis les propriétés locales vérifiées sur le décodeur (P_2) et les deux autres depuis les propriétés locales vérifiées sur le décompresseur (P_3). Par exemple, pour le décodeur :

— si un paquet est prêt et que l'**adresse décompressée** dans celui-ci est dans CR, alors le décodeur fournit une **adresse décodée** dans CR ;
— si un paquet est prêt et le décodeur fournit une adresse décodée dans CR, alors le paquet décompressé contient aussi une adresse dans CR.

Réciproquement, pour le décompresseur, nous avons deux obligations de preuve qui indiquent que l'**adresse décompressée** correspond à l'**adresse contenue dans le paquet** transmis par *CoreSight* (si un paquet est prêt). La section C.1 de l'annexe C donne une expression formelle des obligations de preuves A_1.

Pour résumer, le lemme L_{enter} est prouvé à partir de l'axiome 5.10, des obligations de preuves que nous venons d'énoncer et de la première propriété locale de P_1 (vérifiée sur le transducteur). Un raisonnement similaire permet de prouver les quatre lemmes intermédiaires.

Ressources : le dossier `5_5_3_lemmes_intermediaires/` contient le programme qui démontre les quatre lemmes intermédiaires tels que décrits dans la section 5.5.3.2. Cette preuve utilise notre bibliothèque Coq et la bibliothèque `LTL_Coq`.
Dans ce dossier, le fichier `axioms.v` contient les définitions des axiomes telles que décrites dans les sections 5.4.3 et 5.5.1. Le fichier `proof_obligations.v` contient les définitions des obligations de preuve A_1. Les propriétés locales P_1, vérifiées sur le transducteur, sont converties à l'aide de notre programme *psl2coq*.
Un `Makefile` permet d'automatiser la conversion des propriétés locales et la preuve des lemmes intermédiaires.

5.5.3.4 Preuve du lemme L_{0_A}

La dernière étape de vérification, au niveau du transducteur, est de prouver le lemme L_{0_A} (5.14) à partir des lemmes intermédiaires (5.17, 5.18, 5.19 et 5.20). L_{0_A} spécifie une équivalence à tout instant entre les prédicats ($PC \in CR$) et ($PC_d \in CR$). Nous séparons sa preuve en celle de deux implications. Les expressions 5.21 et 5.22 représentent ces deux implications.

$$\textbf{always}((PC \in CR) \rightarrow (PC_d \in CR)) \tag{5.21}$$

$$\textbf{always}((PC_d \in CR) \rightarrow (PC \in CR)) \tag{5.22}$$

Nous avons donc deux lemmes à établir. Nous réalisons une preuve par induction sur l'état dans lequel nous nous trouvons. Ceci est possible car toutes les expressions que nous manipulons sont des invariants (propriétés temporelles dans un opérateur **always**). Nous prouvons donc que les deux implications sont vérifiées à l'état initial, puis que pour tout état, si celles-ci sont vérifiées, alors elles le sont également à l'état suivant.

À l'état initial, pour l'implication 5.21, nous pouvons prouver $\neg(PC \in CR)$ simplement à partir de notre modèle d'environnement (axiomes représentés par les expressions 5.7 et 5.8).

Pour l'implication 5.22, nous utilisons P_{start}, la cinquième propriété locale de P_1 : celle qui vérifie que ($PC_d \in CR$) est falsifiée.

Pour la preuve par induction, nous manipulons les lemmes intermédiaires et démontrons les deux implications. La première implication peut être prouvée à partir des lemmes L_{enter} et L_{in} (5.17, 5.18), tandis que la seconde implication peut être prouvée à partir des lemmes L_{exit} et L_{out} (5.19, 5.20). La démonstration est rendue disponible publiquement et distribuée sous licence libre [1].

Ressources : le dossier `5_5_3_lemme_l0/` contient le programme qui démontre, à partir des quatre lemmes intermédiaires, les deux implications représentées par les expressions 5.21 et 5.22, c'est-à-dire notre lemme L_{0_A}.
Un `Makefile` permet d'automatiser la preuve du lemme L_{0_A}.

5.5.4 Propriétés P_2 et obligations de preuve A_2 : décodeur

Nous avons identifié des obligations de preuve A_1 pour le transducteur. Parmi celles-ci, nous devons prouver que, lorsqu'un paquet est prêt, alors l'adresse décodée se trouve (ou non) dans CR lorsque l'adresse contenue dans le paquet émis par *CoreSight* se trouve (ou non) dans CR. En d'autre termes, nous devons prouver que le décodeur décode correctement l'adresse contenue dans un paquet.

Lors de la vérification formelle du décodeur, nous émettons l'hypothèse A_2 que le décompresseur fournit un paquet décompressé identique à celui transmis par *CoreSight*. Pour le décodage, nous vérifions une propriété plus forte que celle nécessaire à la preuve de A_1 : nous vérifions que, lorsqu'un paquet est prêt, l'adresse décodée est identique à l'adresse contenue dans le paquet. De ce fait, si l'une se trouve (ou non) dans CR, alors l'autre s'y trouve également. Après cette vérification, selon l'hypothèse A_2, nous pouvons alors considérer les obligations de preuve A_1 vérifiées.

Malheureusement, aucune spécification formelle n'est fournie par le constructeur de *CoreSight*. Nous devons donc formellement exprimer nous-même le décodage des paquets à partir des données fournies dans la documentation [59]. Cette étape est un processus manuel et nous émettons l'hypothèse que notre expression est correcte. Son résultat est un axiome qui décrit que si certaines propriétés locales P_2 sont vérifiées par le décodeur, et que nous émettons l'hypothèse A_2 que la décompression des paquets est correcte, alors l'adresse décodée est identique à l'adresse contenue dans le paquet.

Nous avons identifié un total de 44 propriétés locales pour P_2. L'annexe C, et plus particulièrement la section C.2, détaille l'architecture du décodeur et donne des exemples de propriétés locales. Nous avons vérifié P_2 par *model-checking* sur le décodeur. Dans cette section, nous décrivons la procédure que nous avons suivie pour déterminer les 44 propriétés.

5.5.4.1 Décodage du jeu d'instructions

Le format des adresses dans les paquets n'est pas constant : la position des bits d'adresse dans le paquet dépend du jeu d'instructions à l'adresse de destination [59]. Nous devons donc, dans un premier temps, exprimer formellement le décodage du jeu d'instructions.

Cette information est présente dans tous les paquets *isync* et dans les paquets *branch* lors d'un changement par rapport à la dernière émission. Ceci sous-entend qu'elle n'est pas répétée à chaque paquet et que notre décodeur doit la mémoriser. Toutes les propriétés que nous

exprimons décrivent donc une mémorisation dans un registre dédié en plus d'une vérification de la valeur. Nous exprimons formellement le décodage et la mémorisation du jeu d'instructions, dans le cas d'un paquet *isync* et dans le cas de paquets *branch* où la valeur est présente. Également, nous exprimons formellement que la valeur mémorisée ne varie pas tant que nous ne recevons pas un paquet contenant une nouvelle valeur.

Nous avons identifié un total de 9 propriétés pour spécifier le décodage du jeu d'instructions. La définition de ces propriétés est rendue disponible publiquement et distribuée sous licence libre [1]. Une fois vérifiées, nous considérons correct le jeu d'instructions décodé et le format des paquets lorsque nous définissons le décodage des adresses de destination.

Ressources : le fichier `5_5_4_mc_decodeur/decoded_isetstate.spec.smv` contient la définition des 9 propriétés que nous avons identifiées pour vérifier la correction du décodage du jeu d'instructions. Notre méthode de vérification formelle automatisée permet de s'assurer, par *model-checking*, que le décodeur vérifie ces propriétés.
Un `Makefile` permet d'automatiser cette vérification.

5.5.4.2 Décodage des adresses de destination

Dans le cas des paquets *isync*, tous les bits d'adresses sont présents et il n'y a pas de dépendance au jeu d'instructions. Nous vérifions donc que l'adresse décodée prend bien la valeur de l'adresse décompressée et que les bits d'adresses sont mémorisés. Dans le cas des paquets *branch*, nous spécifions le décodage des bits d'adresse en fonction du jeu d'instructions (mémorisé) et de la présence des bits de compression. Nous vérifions que les bits d'adresse présents dans le paquet sont égaux dans la valeur décodée et sont mémorisés. Nous vérifions également que les bits d'adresse absents du paquets sont, dans la valeur décodée, égaux à ceux mémorisés. Pour finir, nous vérifions que les valeurs des bits d'adresse mémorisés ne sont pas modifiés si aucun paquet (*isync* ou *branch*) contenant une nouvelle valeur n'est reçu.

Nous avons identifié un total de 35 propriétés pour spécifier le décodage des adresses de destination en fonction de la donnée décompressée. La définition de ces propriétés est rendue disponible publiquement et distribuée sous licence libre [1].

Ressources : le fichier `5_5_4_mc_decodeur/decoded_address.spec.smv` contient la définition des 35 propriétés que nous avons identifiées pour vérifier la correction du décodage des adresses. Notre méthode de vérification formelle automatisée permet de s'assurer, par *model-checking*, que le décodeur vérifie ces propriétés.
Un `Makefile` permet d'automatiser cette vérification.

Une fois les 44 propriétés locales vérifiées (9 + 35), nous considérons que l'adresse décodée est identique à l'adresse décompressée. Aussi, les obligations de preuve A_1 sont vérifiées sous réserve que l'hypothèse A_2 l'est également. L'hypothèse A_2 forme donc une obligation de preuve pour le reste du moniteur, c'est-à-dire le décompresseur (figure 5.3).

5.5.5 Propriétés P_3 et axiomes A_3 : décompresseur

La vérification formelle du décompresseur nous permet de prouver les obligations de preuve A_2. Les propriétés locales que nous vérifions sur celui-ci forment P_3 (figure 5.3). Le décompresseur est le dernier système matériel que nous vérifions formellement, la preuve de A_2 ne

doit donc dépendre que de P_3 et d'axiomes qui représentent notre modèle d'environnement. Nous nommons ces axiomes A_3.

Comme pour le décodeur de traces, nous exprimons formellement nous-même la décompression des paquets à partir des données fournies dans la documentation de *CoreSight* [59]. Le résultat est un axiome qui décrit que, si certaines propriétés locales P_3 sont vérifiées par le décompresseur, alors non seulement l'adresse décompressée est identique à l'adresse contenue dans le paquet émis, mais aussi la décompression correcte permet de vérifier les hypothèses du décodeur (A_2). Nous vérifions donc que, lorsqu'un paquet est prêt, l'adresse décompressée est identique à l'adresse contenue dans le paquet. Nous vérifions également que l'identification des entêtes et des bits de décompression permet de reconstruire tous les paquets et détecter lorsque ceux-ci sont prêts.

Nous avons identifié un total de 91 propriétés locales pour P_3. L'annexe C, et plus particulièrement la section C.3, donne des exemples de propriétés locales. Nous avons vérifié P_3 par *model-checking* sur le décompresseur. Dans cette section, nous décrivons la procédure que nous avons suivie pour déterminer les 91 propriétés.

5.5.5.1 Adresse décompressée

Pour vérifier que l'adresse décompressée est identique à l'adresse transmise par *CoreSight*, nous vérifions que, lorsque les paquets *isync* et *branch* sont prêts, tous les bits des octets contenant les adresses de destination sont présents dans la donnée décompressée. Une égalité est donc nécessaire pour les 5 premiers octets des paquets de type *isync* et *branch* [59].

Les détails de la vérification pour les paquets *branch* sont donnés dans le chapitre 4, plus particulièrement dans la section 4.4. L'expression 4.3 décrit la spécification dans le cas de paquets contenant 5 octets d'adresse. Nous vérifions 5 autres spécifications similaires : une pour les paquets de type *isync* et une pour les paquets de type *branch* contenant 1, 2, 3 et 4 octets d'adresse. Cette spécification est exprimée en fonction des prédicats *ready* et *data_size*, indiquant respectivement si le paquet est prêt paquet et le nombre d'octets présents dans celui-ci. La vérification de ces spécifications n'est donc correcte que dans le cas où le décompresseur identifie correctement ces prédicats.

5.5.5.2 Identification des paquets et de leur taille

Pour déterminer si un paquet est prêt, le décompresseur doit identifier lorsque celui-ci commence : il doit donc pouvoir identifier les en-têtes et décompresser tous les types de paquet (pas seulement *isync* et *branch*). Nous avons identifié 14 propriétés locales pour spécifier l'identification de tous les types de paquets, et 66 pour spécifier la détection des paquets prêts. Un compteur indique sur le vecteur *data_size* le nombre d'octets contenus dans les paquets prêts ; 5 propriétés locales sont nécessaires pour spécifier son comportement. La définition de ces propriétés est rendue disponible publiquement et distribuée sous licence libre [1]. Le total de ces 85 propriétés locales (14 + 66 + 5) vérifie l'hypothèse d'identification correcte des spécifications précédentes (comme celle représentée par 4.3).

Ressources : le dossier `5_5_5_mc_decompresseur/` contient la définition de 85 propriétés que nous avons identifiées pour vérifier la correction de la décompression. Notre méthode de vérification formelle automatisée permet de s'assurer, par *model-checking*, que le décompresseur vérifie ces propriétés.
Un `Makefile` permet d'automatiser cette vérification.

5.5.5.3 Axiomes : modèle d'environnement et théorème

La vérification des propriétés locales P_3 permet également de prouver les obligations de preuve A_2, dans l'hypothèse A_3 où *CoreSight* fonctionne comme décrit dans sa documentation [59] (et où nous avons correctement traduit cette documentation). Or, sans accès aux sources du microprocesseur et de *CoreSight*, nous considérons cette hypothèse comme notre modèle d'environnement axiomatique (voir la section 5.5.1). Nous ne considérons donc pas A_3 comme une obligation de preuve mais comme un axiome défini par notre modèle d'environnement ($A_3 \rightarrow A_3$). Les obligations de preuve nécessaires à la démonstration du lemme intermédiaire L_{enter} (5.17) sont donc satisfaites.

Nous réutilisons ces obligations de preuve et en prouvons de nouvelles pour démontrer les trois autres lemmes intermédiaires (5.18, 5.19 et 5.20). L'annexe C fournit des détails sur ces démonstrations. Les quatre lemmes intermédiaires sont suffisants pour prouver le lemme L_{0_A} et, par extension, la spécification LTL_2. Une fois L_{0_A} démontré, l'établissement du théorème est donc valide.

Preuve complète sur le modèle zéro

Dans cette section, nous détaillons l'établissement d'un théorème pour prouver la spécification LTL_2. La même stratégie de vérification a été employée pour prouver les 9 théorèmes, un pour chaque spécification (5.1). Nous avons pu réutiliser des lemmes pour prouver les spécifications. Par exemple, le lemme L_{0_A} (5.14) a été réutilisé pour prouver les spécifications qui contiennent le prédicat ($PC \in CR$). Comme notre modèle zéro possède la même définition de la sécurité que VRASED (**P1** à **P4**), nous avons réutilisé sa preuve pour démontrer la sécurité sur notre implémentation, à partir des 9 spécifications. Ainsi, nous avons garanti la sécurité de l'attestation à distance sur notre modèle zéro, sans modification du cœur du microprocesseur. La démonstration de l'obtention des 9 spécifications est rendue disponible publiquement et distribuée sous licence libre [1].

Ressources : le dossier `5_5_6_preuve_model_0/` contient un programme écrit en Coq qui établit les 9 théorèmes. Il contient également les définitions des propriétés locales qui constituent les hypothèses nécessaires et suffisantes au programme. Du *model-checking* permet de garantir que les propriétés locales sont vérifiées par le moniteur matériel. La preuve que les spécifications LTL_2 à LTL_{10} sont vérifiées peut être jouée en compilant et en exécutant le programme écrit en Coq.
Un `Makefile` permet d'automatiser ces vérifications.

5.6 Soundness

Notre vérification formelle permet de garantir la sécurité sur le modèle zéro. Cependant, l'architecture que nous proposons doit aussi permettre l'exécution de la fonction d'attestation.

Dans cette section, nous décrivons comment nous définissons et vérifions la *soundness* de cette architecture.

5.6.1 Définition

De la même manière que VRASED, nous définissons la *soundness* comme la possibilité d'exécuter la fonction d'attestation de manière correcte [36]. C'est-à-dire que :

— le périphérique de confiance n'entrave pas l'exécution légitime de la fonction d'attestation ;

— le résultat fourni par celle-ci est correct vis-à-vis du comportement attendu.

En d'autres termes, si la fonction d'attestation calcule, à partir du secret, un HMAC sur la région mémoire à attester et le challenge, alors il est possible d'exécuter ce calcul et, une fois celui-ci terminé, le résultat est bien égal au HMAC donné.

Formellement, l'expression 5.23 fournit la définition de la *soundness* par VRASED [36]. Dans cette définition, M représente une valeur quelconque pour la région mémoire à attester, $\mathcal{C}hal$ représente la valeur du challenge et $\mathcal{K}$ représente le secret tel que stocké dans la région mémoire *KR*. *HMAC*(m, s) représente une fonction mathématique qui à toute région m et secret s calcule un HMAC sur m à partir du secret s.

$$
\begin{aligned}
&\textbf{always}(\\
&\quad (PC = CR_{min}) \wedge (AR = M) \wedge (MR = \mathcal{C}hal) \\
&\quad \wedge (\neg reset \ \textbf{until} \ (PC = CR_{max})) \\
&\rightarrow \\
&\quad \textbf{eventually}(\\
&\quad\quad (PC = CR_{max}) \wedge (MR = HMAC(AR \,||\, \mathcal{C}hal, \mathcal{K})) \\
&\quad) \\
&)
\end{aligned}
\tag{5.23}
$$

Cette définition stipule que, pour toute valeur M présente dans la région mémoire AR, pour tout challenge $\mathcal{C}hal$ présent dans la région mémoire MR, si le pointeur d'instruction est situé à la première instruction de la région critique et qu'aucun *reset* n'est transmis avant que celui-ci n'atteigne la dernière instruction, alors cette dernière instruction sera atteinte et MR contiendra le résultat du calcul du HMAC. Pour notre périphérique de confiance, nous choisissons d'utiliser la même définition de *soundness* en langage naturel mais nous pensons que cette expression formelle est erronée.

5.6.2 Stratégie de vérification

En effet, les auteurs de VRASED ont choisi de prouver la *soundness* 5.23 par réécriture de la conjonction des spécifications (LTL_2 à LTL_{10}) et de la définition du comportement de la fonction d'attestation [36].

Comme notre modèle zéro vérifie les mêmes spécifications (LTL_2 à LTL_{10}), nous pourrions hériter de la preuve. Cependant, nous pensons que cette stratégie n'est pas suffisante pour vérifier la *soundness* du système (même si nous savons que VRASED est *sound*). En effet, l'ensemble des spécifications vérifiées sur le système ne reflète pas l'entièreté de son com-

portement. Ainsi, certaines propriétés locales, qui n'ont pas été vérifiées peuvent empêcher le système de fonctionner correctement alors qu'elles n'interviennent pas dans la preuve.

Nous pouvons illustrer cette faiblesse par un contre-exemple. Prenons le cas d'un système qui ne démarre pas : il n'est donc pas *sound*. La propriété suivante est vérifiée :

$$\textbf{always}(reset) \tag{5.24}$$

Ce système vérifie, par extension, les spécifications LTL_2 à LTL_{10}. Cependant, si le système est toujours en état de *reset*, alors il ne permet pas l'exécution de la fonction d'attestation. Pourtant, il vérifie la définition formelle exprimée par l'expression 5.23.

Ressources : le programme `5_6_2_counterexemple/proof.v` utilise Coq. Il démontre que l'ensemble des spécifications LTL_2 à LTL_{10} peut être prouvé depuis la propriété locale représentée par l'expression 5.24.
Un `Makefile` permet d'automatiser la preuve.

Afin de vérifier la *soundness*, nous pensons qu'il est nécessaire de faire cette vérification sur le modèle concret. En ce sens, nous pouvons tester sur le matériel (implémentation finale du modèle concret) l'exécution de la fonction d'attestation. Pour cette raison, nous savons que VRASED est *sound*, car les auteurs ont réalisé une exécution de la fonction d'attestation sur leur matériel [36].

Nous exécutons donc la fonction d'attestation sur le matériel et observons le résultat fourni par celle-ci. Nous proposons donc l'expérience suivante pour vérifier la *soundness* :

— l'architecture telle que représentée sur la figure 5.1 est implémentée sur une carte de développement Cora Z7-07S ;
— une valeur choisie $\mathcal{K}$ pour le secret est placée dans KR ;
— le code de la fonction d'attestation est placé dans CR ;
— une valeur choisie M est placée dans la DDR à une adresse choisie AR ;
— une valeur choisie $\mathcal{C}hal$ est placée dans la DDR à une adresse choisie MR ;
— une application est chargée dans la DDR, celle-ci effectue un branchement à CR_{min} pour exécuter la fonction d'attestation, puis compare le résultat avec une valeur attendue.

La valeur attendue pour le résultat est calculée en fonction du choix de la fonction d'attestation, de $\mathcal{K}$, de M et de $\mathcal{C}hal$. Lors de la comparaison, si le résultat est égal à la valeur attendue et qu'aucun *reset* n'a été transmis par notre moniteur matériel, alors nous considérons que le système est *sound*.

Nous avons réalisé cette manipulation sur notre carte de développement Cora Z7-07S. Afin de simplifier la manipulation et la vérification du résultat, nous avons remplacé le calcul du HMAC par un simple XOR entre le secret et le challenge. Également, le signal de *reset* est connecté à une LED via une flip-flop. Nous avons observé une égalité entre le résultat du calcul et la valeur attendue ; la LED ne s'est pas allumée. Nous concluons donc que l'architecture que nous proposons est *sound*. Les sources pour reproduire l'expérience sont distribuées sous licence libre et rendues accessible publiquement [1].

Ressources : le dossier `5_6_2_soundness/` contient les sources pour implémenter l'architecture telle que représentée sur la figure 5.1. Il contient également les sources d'un FSBL et d'une application pour vérifier la *soundness* sur la carte de développement. Un script gdb permet d'instrumenter openOCD pour la programmation.
Un Makefile permet d'automatiser toutes ces tâches.

5.7 Faiblesses admises

Notre modèle zéro comporte des faiblesses vis-à-vis de notre modèle de menaces, décrit dans la section 5.1.2. Nous admettons ces faiblesses et tentons d'y remédier dans les prochains chapitres. Ces faiblesses admises sont représentées par nos hypothèses simplifiantes. Nous pouvons les diviser en deux catégories : les faiblesses matérielles, auxquelles une extension du moniteur matériel peut peut-être remédier ; et les faiblesses logicielles, auxquelles une configuration des périphériques du microprocesseur permet de remédier.

5.7.1 Faiblesses matérielles

Tout d'abord, notre solution ne s'applique qu'aux microprocesseurs ne comportant qu'un seul cœur. En effet, les accès aux esclaves AXI-Lite sont autorisés lorsque le pointeur d'instruction se situe dans la région critique. La valeur du pointeur d'instruction est déduite des traces, fournies par *CoreSight*, relatives au cœur du microprocesseur qui exécute la fonction d'attestation. Dans le cas d'un système multi-cœurs, un adversaire peut exécuter du code qui accède au secret depuis un second cœur pendant l'exécution de la fonction d'attestation par le premier cœur. Cette faiblesse est admise et couverte par l'hypothèse simplifiante **H5**.

Également, notre solution n'est valide que lorsque l'adversaire ne dispose pas d'un accès au périphérique de programmation du FPGA. En effet, le microprocesseur a la capacité de reprogrammer le FPGA en utilisant le *Processor Configuration Access Port* (PCAP) [50]. À partir d'un accès distant, un adversaire peut reprogrammer le FPGA et désactiver les fonctionnalités apportées par le moniteur matériel. Il peut également accéder en lecture au résultat de la programmation et lire par ce moyen le contenu des esclaves AXI-Lite (ce qui inclut le secret). Cette faiblesse est admise et couverte par l'hypothèse simplifiante **H8**.

Enfin, nous considérons que les paquets *isync* et *branch* fournis par *CoreSight* sont disponibles à l'état précédent l'exécution de l'instruction de destination. En effet, nous éludons la problématique du temps de transmission du paquet à travers l'interface TPIU. Nous n'avons donc pas de preuve qui garantit qu'un adversaire ne peut pas extraire des informations sensibles durant ce temps de transmission. En particulier, lors d'une levée d'exception, la routine de gestion d'interruption peut potentiellement réaliser une sauvegarde du contenu des registres du microprocesseur avant la transmission de la trace *CoreSight*. Cette faiblesse est admise et couverte par l'hypothèse simplifiante **H9**.

L'annexe E de ce manuscrit apporte des éléments de réponse pour pallier ces différentes faiblesses.

5.7.2 Etat du microprocesseur

Notre solution présente également des faiblesses logicielles. En effet, nous avons considéré que le microprocesseur se trouvait dans un état spécifique avant l'exécution de la fonction d'attestation. Cette supposition implique, entre autres, que la configuration de *CoreSight* et de la MMU permet d'obtenir nos traces d'exécution dans la plage d'adresses souhaitée, avec des adresses virtuelles telles qu'attendues par le transducteur. Également, une autre supposition est que les mémoires cache sont désactivées lors de l'exécution de la fonction d'attestation.

Or, nous n'avons pas de preuve qu'un adversaire ne puisse pas modifier la configuration des périphériques avant une exécution légitime de la fonction d'attestation. Au contraire, nous

n'apportons pas de mécanisme de sécurisation qui garantit cet état spécifique. Cette faiblesse est admise et couverte par les hypothèse simplifiante **H6** et **H7**.

Dans le chapitre 6, nous proposons une nouvelle contribution : une architecture co-conçue et vérifiée formellement pour garantir l'exécution d'un algorithme qui configure les périphériques du microprocesseur comme attendu. Cette architecture constitue une extension de notre périphérique de confiance. L'objectif de cette contribution et de permettre d'attester de la configuration du microprocesseur avant l'exécution de la fonction d'attestation, et ce même dans un environnement potentiellement corrompu.

5.8 Conclusion

Dans ce chapitre, nous avons présenté une architecture co-conçue pour établir une racine de confiance statique sur microprocesseur pris sur étagère. Le défi pour cette architecture est que le cœur du microprocesseur ne peut pas être modifié. Notre solution se base donc sur la conception d'un périphérique de confiance, dans la partie FPGA du *SoC*, dont un moniteur matériel garantit les propriétés de sécurité suffisantes.

Pour vérifier formellement la sécurité de cette solution, nous avons réutilisé la définition de la sécurité de l'attestation à distance, adaptée aux microcontrôleurs, fournie par VRASED. Nous avons également réutilisé des automates de sécurité et la fonction d'attestation, afin de préserver les propriétés de sécurité que ceux-ci garantissent. Nous avons traité progressivement l'augmentation de la surface d'attaque, causée par une extension aux microprocesseurs. Dans un premier temps, nous l'avons donc abstraite pour valider notre choix d'architecture. En outre, des hypothèses simplifiantes nous ont permis de définir notre modèle zéro. Nous raffinerons notre modèle par la suite (chapitre 6 et annexe E) et prouverons une nouvelle définition de la sécurité.

Nous avons modélisé le comportement du microprocesseur (décrit dans sa documentation) et traduit nos hypothèses simplifiantes en une série d'axiomes. Sur notre architecture, nous avons déduit les prédicats nécessaires à l'obtention de la sécurité à partir des traces d'exécution fournies par *CoreSight* et des signaux d'accès sur le bus AXI. En établissant des théorèmes qui attestent de la vérification de 9 spécifications sur notre système, nous avons pu profiter de la preuve de sécurité fournie par VRASED et n'avons pas eu à la rejouer. Nous avons vérifié, par *model-checking*, des propriétés locales sur notre moniteur matériel et, à partir de celles-ci et des axiomes précédemment définis, nous avons prouvé les 9 spécifications.

Nous avons choisi, comme VRASED, d'exécuter une primitive cryptographique logicielle pour la fonction d'attestation. Les hypothèses simplifiantes garantissent la sécurité de cette exécution sur le prouveur. Par exemple, elles suggèrent que la configuration des périphériques est dans un état défini et non modifiable par l'adversaire avant l'exécution de la fonction d'attestation. Également, elles suggèrent que les mémoires cache sont désactivées avant l'exécution des premières instructions de la fonction d'attestation. Malheureusement, de telles hypothèses ne sont pas réalistes vis-à-vis de notre modèle de menaces : l'exécution de la fonction d'attestation se fait dans un environnement potentiellement corrompu. Pour le modèle 1 (la prochaine itération lors du raffinement), nous proposerons donc une solution d'attestation de configuration du microprocesseur et de ses périphériques, afin que l'environnement d'exécution ne mette pas à mal les propriétés garanties par la fonction d'attestation. Cette contribution est décrite dans le chapitre 6.

Chapitre 6

Solution d'attestation de configuration du microprocesseur dans un environnement corrompu

Sommaire

Dans le chapitre précédent, nous avons vérifié formellement la sécurité de l'exécution d'une fonction d'attestation et établi une racine de confiance statique vis-à-vis d'un modèle de menaces simplifié. Notamment, la configuration des périphériques, de la MMU et l'état des mémoires cache sont considérés corrects lors de l'exécution de la fonction d'attestation.

Dans ce chapitre, nous proposons d'intégrer ces éléments d'environnement dans notre modèle de menaces. C'est-à-dire considérer un adversaire qui peut corrompre les mémoires cache, la configuration de l'ensemble des périphériques et la MMU. Nous raffinons donc notre définition de la sécurité de l'attestation à distance : nous nommons le modèle du microprocesseur le *modèle 1*.

Nous introduisons donc deux nouvelles contributions dans ce chapitre :

— la proposition d'une extension logicielle et matérielle de notre périphérique de confiance, pour la vérification de l'intégrité d'environnement d'exécution. Son objectif est de se protéger du modèle de menaces plus fort ;

— une étude de la sécurité de cette extension, avec pour objectif la prévention de vulnératilités face à un adversaire possédant de hauts privilèges, une connaissance de l'architecture et un clone du système physique.

Ces travaux ont fait l'objet d'une publication [89] et d'une présentation à la conférence *Symposium sur la Sécurité des Technologies de l'Information et des Communications* (SSTIC) 2022.

La section 6.1 décrit le contexte des travaux. La section 6.2 présente la stratégie d'attestation de configuration que nous envisageons ainsi que l'extension du périphérique de confiance que nous proposons. La section 6.3 décrit l'étude de sécurité que nous avons conduite sur notre extension. La section 6.4 décrit la vérification formelle de la partie matérielle de cette extension. Finalement, les résultats sont donnés dans la section 6.6.

Dans une optique de rejeu des expériences illustrant nos travaux, les détails de la preuve de sécurité pour l'attestation de configuration sont fournis en annexe D.

6.1 Contexte

Dans ce chapitre, nous réutilisons le périphérique de confiance de nos travaux précédents et nous proposons une extension qui permet de vérifier l'état du microprocesseur avant l'exécution de la fonction d'attestation. L'objectif est de décharger les hypothèses simplifiantes de nos travaux précédents (décrits dans le chapitre 5) : c'est-à-dire de considérer un adversaire pouvant corrompre les mémoires cache, la configuration de *CoreSight* et la MMU. Pour ce faire, nous étendons la fonction d'attestation ainsi que le moniteur matériel avec un test d'intégrité supplémentaire et permettons ainsi de se prémunir de nouvelles classes d'attaques.

6.1.1 Nouvelles classes d'attaques considérées

Une nouvelle classe d'attaques dont nous voulons nous protéger est celle qui requiert une **utilisation des mémoires cache**. En effet, notre moniteur matériel considère qu'une lecture du contenu des esclaves AXI-Lite a lieu lors d'un accès en lecture sur le bus AXI. Dans le cas d'une activation des mémoires cache par un adversaire avant l'exécution de la fonction d'attestation, un accès au secret ou au contenu de la pile exclusive peut être réalisé par la suite sans être visible sur le bus AXI.

Une autre nouvelle classe d'attaques dont nous voulons nous protéger est celle des attaques par **reconfiguration de *CoreSight***. En effet, la configuration de *CoreSight* permet de définir les plages d'adresses où les traces sont transmises au moniteur. Une reprogrammation par un adversaire peut permettre, par exemple, la définition de plages d'adresses où les dernières instructions de la fonction d'attestation ne sont pas contenues. Ainsi, une exécution de la fonction

d'attestation indique, au début, au moniteur matériel une entrée du pointeur d'instruction, les accès au secret sont donc autorisés. Et, à la fin de l'exécution de la fonction d'attestation, le moniteur ne reçoit pas la trace de fin et les accès au secret ne sont pas de nouveau interdits. Un adversaire peut alors accéder au secret (ou à la pile exclusive) par la suite.

La dernière classe d'attaques que nous considérons est celle des attaques par **reconfiguration de la MMU**. En effet, les adresses des lectures et écritures réalisées par la fonction d'attestation sont renseignées dans son code, ce sont donc des adresses virtuelles. Si un adversaire modifie les traductions d'adresses, il devient possible d'utiliser un espace mémoire dans la DDR au lieu de la pile exclusive. Ainsi, une exécution de la fonction d'attestation fournit à l'adversaire un accès indirect au secret. Également, les contrôles d'accès sont garantis par le moniteur matériel en fonction de la valeur du pointeur d'instruction fournie par *CoreSight*. Or, les adresses fournies par *CoreSight* sont des adresses virtuelles. Le moniteur matériel autorise donc les accès au secret depuis une plage d'adresses virtuelles attendue. Si les traductions d'adresses sont modifiées, la lecture du secret devient alors autorisée dans une autre plage d'adresses.

6.1.2 Modèle de menaces

De la même manière que dans le chapitre 5, nous considérons un modèle de menaces fort où l'adversaire n'a pas d'accès physique au système. Ici aussi, notre adversaire peut prendre le contrôle du *prouveur* au travers d'un accès distant et a déjà réalisé des attaques qui lui ont permis d'élever ses privilèges. Il lui est donc toujours possible de lire et écrire dans toutes les mémoires auxquelles le microprocesseur a accès.

Nous nous plaçons dans le même cas d'étude, à savoir que l'adversaire a corrompu l'environnement du *prouveur*, de telle sorte que l'attestation à distance par le *vérifieur* va détecter son intrusion. Son objectif est donc de forger un résultat au test d'intégrité authentifié, de manière à ce que l'attestation à distance réussisse.

Nous ajoutons à l'adversaire, par rapport au chapitre 5, la capacité suivante : **notre adversaire possède un clone du *prouveur*, il peut donc reproduire l'environnement du *prouveur* sur un système où il a physiquement accès**. Cela rend possible la connexion d'un analyseur logique et l'observation des signaux sur le matériel lors d'une exécution de logiciel.

De plus, nous considérons que l'architecture de notre périphérique de confiance est librement accessible et que notre adversaire l'a déjà étudiée. Seul le secret utilisé pour le calcul du test d'intégrité lui est inconnu. En conséquence, notre solution ne peut s'appuyer sur l'offuscation pour garantir la sécurité du système vis-à-vis des classes d'attaques considérées.

6.2 Attestation de configuration

Dans cette section, nous proposons une extension qui permet de décharger le périphérique de confiance (réutilisé) des hypothèses simplifiantes. Concrètement, nous souhaitons attester de la configuration correcte de l'environnement d'exécution de la fonction d'attestation (mémoires cache, *CoreSight*, MMU) pour se protéger de nouvelles classes d'attaques.

En d'autres termes, nous ne pouvons plus nous fier à nos observables avant d'avoir nous-même vérifié la configuration de l'environnement. Nous choisissons de faire confiance au fonctionnement du microprocesseur et de ses périphériques pour une configuration donnée. Ce-

pendant, nous ne faisons pas confiance à leur configuration lors du test d'intégrité. Nous garantissons donc des propriétés qui assurent la conformité.

6.2.1 Positionnement

À la manière de Kovah et al. [34], le modèle de sécurité de notre test d'intégrité se base sur une implémentation optimale en termes de cycles d'horloge. Cela permet d'écarter un adversaire capable d'augmenter sa puissance de calcul. Afin de se protéger des classes d'attaque par *overclocking* [33], nous utilisons des domaines d'horloge indépendants pour le microprocesseur et le moniteur matériel. Comme dans SMART et VRASED [35, 36], un moniteur matériel, implémenté ici dans le FPGA du *SoC*, garantit les propriétés de sécurité et est capable de redémarrer le système en cas de future compromission. La sécurité s'appuie sur un traitement des signaux fournis par l'interface de débug *CoreSight*.

6.2.2 Solution proposée

Nous proposons d'étendre le périphérique de confiance avec un nouveau moniteur matériel, dédié à la configuration de l'environnement d'exécution de la fonction d'attestation.

Nous émettons l'hypothèse que le FPGA est correctement programmé au démarrage du *SoC* (exécution correcte du FSBL) et que l'adversaire ne peut accéder à sa configuration a-posteriori (hypothèse simplifiante **H8**). Cette hypothèse est réaliste car un module matériel peut être ajouté au système et la vérifier. En effet, un accès en lecture ou en écriture par le microprocesseur au contenu du FPGA se traduit par une modification des signaux d'accès au module ICAP [50], situé dans le FPGA. Il est donc possible de procéder à un *reset* du microprocesseur en cas de lecture ou écriture du contenu du FPGA par l'adversaire.

Le nouveau moniteur, que nous proposons d'ajouter, observe les traces d'exécution fournies par *CoreSight* et les signaux sur le bus AXI. Il décide si un *algorithme de confiance* s'exécute. L'algorithme de confiance est un algorithme qui configure les périphériques tel qu'attendu lors de l'exécution de la fonction d'attestation : c'est-à-dire qu'il désactive les mémoires cache et configure *CoreSight* et la MMU (vérification des hypothèses simplifiantes **H6** et **H7**).

L'architecture de notre extension est représentée sur la figure 6.1.

Le nouveau moniteur est lui aussi implémenté dans la partie FPGA du *SoC*, en parallèle du moniteur des travaux précédents.

Une première ROM est ajoutée, sous forme d'un esclave AXI-Lite, et contient le code de l'algorithme de confiance. Celle-ci est accessible par le microprocesseur via le bus AXI.

Une seconde ROM est aussi ajoutée. Elle contient les valeurs des signaux d'accès sur le bus AXI et des traces *CoreSight* telles qu'elles sont attendues par le nouveau moniteur. Cette seconde ROM n'est pas accessible par le microprocesseur. Elle est seulement accessible par le moniteur qui décide, en fonction des signaux observés, si l'algorithme de confiance est bien en train de s'exécuter ou non.

À chaque front d'horloge, le moniteur compare les traces d'exécutions et les signaux d'accès avec les valeurs attendues (disponibles dans la seconde ROM). L'accès à la fonction d'attestation, au secret et à la pile exclusive sera toujours interdit (*reset*) avant la fin de cette exécution. C'est-à-dire que, si, front d'horloge après front d'horloge, les signaux observés par le moniteur ne correspondent pas à ceux attendus, il devient impossible d'exécuter la fonction d'attestation. Cette vérification n'est plus nécessaire lors de l'exécution de la fonction d'attestation car

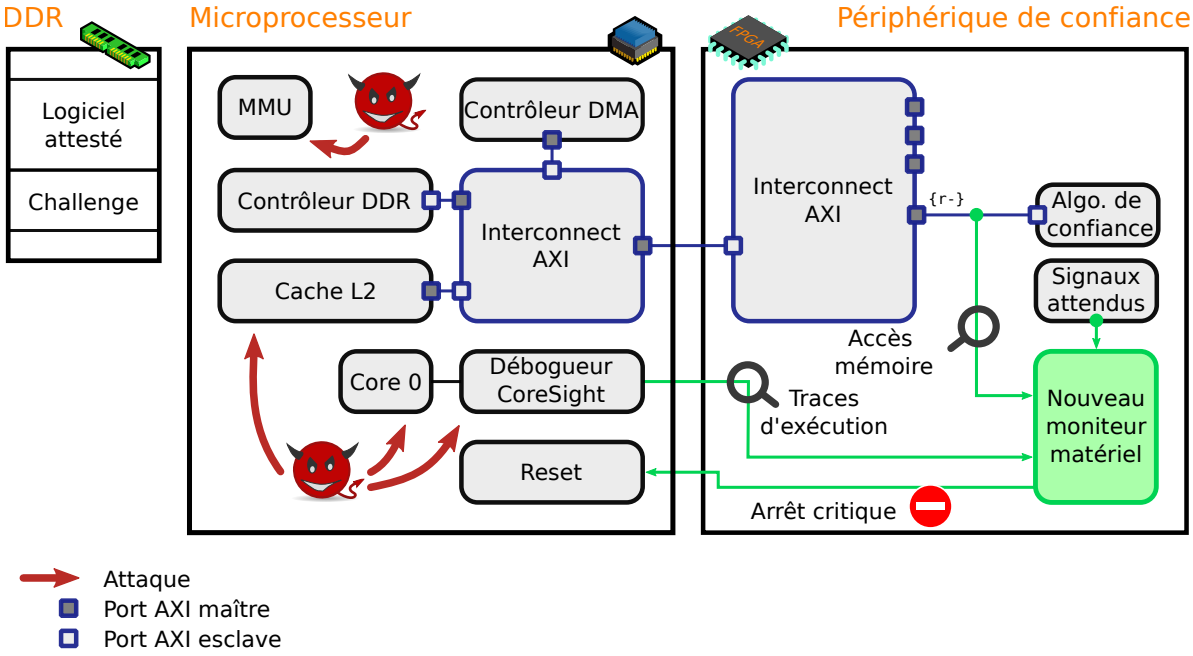

FIGURE 6.1 – Extension du périphérique de confiance

celle-ci est réalisée après celle de l'algorithme de confiance : l'environnement est alors considéré correct.

Notons que nous travaillons sur un microprocesseur qui ne possède qu'un seul cœur (vérification de l'hypothèse simplifiante **H5**). Également, le module d'interconnexion AXI esclave (dans le FPGA) ne possède qu'une seule interface esclave, physiquement connectée l'interface maître du microprocesseur. Tous les signaux observés sur le bus AXI sont donc la conséquence d'une lecture par ce seul cœur et toutes les traces fournies par *CoreSight* décrivent une exécution réalisée par ce même cœur.

L'objectif est le suivant : le microprocesseur doit exécuter l'algorithme de confiance (qui désactive les mémoires cache, configure *CoreSight* et la MMU) pour que les signaux observés par le moniteur correspondent à ceux attendus. **Notre problématique est donc que le moniteur doit parvenir à distinguer, seulement en observant ces signaux, une exécution de l'algorithme de confiance d'une simple lecture de la ROM dans l'esclave AXI-Lite**.

Pour répondre à cette problématique, nous concevons l'algorithme de confiance de manière à ce que, à chaque transaction sur le bus AXI, *CoreSight* émette une trace. Un branchement indirect est donc ajouté, régulièrement, dans l'algorithme de confiance. Ceci permet d'avoir un suivi de l'exécution des instructions accédées sur le bus AXI.

Également, l'algorithme de confiance est optimisé. Il n'est pas optimisé en terme de nombre d'instructions, comme le ferait un compilateur, mais optimisé de sorte que la trace émise par *CoreSight* soit dans une fenêtre de temps très courte après le transfert sur le bus AXI :

— si la trace apparaît rapidement après un accès sur le bus AXI, alors nous pouvons déduire que le moniteur observe bien un *fetch* des instructions par le microprocesseur ;
— si la trace apparaît hors de la fenêtre de temps, après un accès sur le bus AXI, alors

nous pouvons en déduire qu'un adversaire tente de simuler une exécution en effectuant des lectures de l'algorithme de confiance et des branchements indirects depuis un autre programme.

6.2.3 Caractérisation

Sur notre périphérique de confiance, l'horloge (du FPGA) est fournie par un quartz externe : elle est décorrélée du domaine d'horloge du microprocesseur et sa fréquence nominale est de 125MHz. Le périphérique d'interconnexion AXI maître (du microprocesseur) et l'interface de débug *CoreSight* sont synchronisés sur cette horloge externe. Leur fonctionnement est donc ralenti [50].

Pour optimiser notre algorithme de confiance, nous nous basons sur le comportement du microprocesseur lors d'une exécution depuis un esclave AXI. Comme ce comportement est dépendant de notre architecture matérielle et n'est pas décrit dans une documentation, nous procédons à une caractérisation du matériel à l'aide d'un analyseur logique. L'analyseur logique est lui-aussi synchronisé sur l'horloge externe.

La première étape de notre caractérisation consiste à réaliser, depuis le microprocesseur, une lecture et un *fetch* (pour les comparer) depuis un esclave AXI-Lite et observer les signaux sur le bus AXI pour prédire le comportement attendu.

Lors d'une lecture sur le bus AXI, le module d'interconnexion AXI esclave (dans le FPGA) place l'adresse relative sur le vecteur nommé $ARADDR$ et lève le signal $ARREADY$ [60]. Sur notre matériel, une lecture de la donnée par le microprocesseur s'effectue par transactions de mots de 32 bits, où une lecture de 1 mot nécessite 4 périodes d'horloge ; un délai de 15 périodes d'horloge est présent entre deux transactions [1]. Le nombre de mots lus par transaction n'est pas fixe et dépend de la quantité de donnée à lire. Le chronogramme représenté sur la figure 6.2 montre le résultat d'une lecture entre les adresses `0x00` et `0x44` de l'esclave AXI-Lite, par transactions de 8 mots.

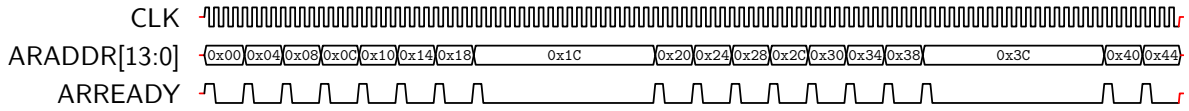

FIGURE 6.2 – Lecture depuis l'esclave AXI-Lite

Lorsque le contenu de l'esclave AXI-Lite est exécutable, un *fetch* par le microprocesseur provoque exactement les mêmes signaux sur le bus AXI que ceux représentés sur la figure 6.2. Nous supposons que ce comportement, que nous observons, reflète la taille de la ligne de cache : $4 \times 8 = 32$ octets [49]. Nous supposons que le microprocesseur soit câblé pour lire 32 octets à la fois (8 mots), même si l'utilisation des mémoires cache est désactivée (comportement identique à celui d'un *cache miss* sans écriture en cache). La taille de la ligne de cache imposerait donc le nombre de mots lus par transaction.

Comme une lecture peut être aussi réalisée par transactions de 8 mots, observer seulement les signaux sur le bus AXI n'est donc pas suffisant pour distinguer une lecture d'un *fetch* des instructions.

Ressources : le dossier `6_2_3_carac_read/` contient les sources pour implémenter un analyseur logique embarqué et connecter le microprocesseur avec un esclave AXI-Lite. Il contient également les sources d'un FSBL et d'une application pour lire dans l'esclave AXI-Lite et observer les signaux sur le bus AXI. Plusieurs transactions sont réalisées, de tailles différentes. Un script gdb permet d'instrumenter openOCD pour la programmation. Un Makefile permet d'automatiser toutes ces tâches.

La seconde étape de notre caractérisation consiste donc à ajouter des branchements indirects et définir l'architecture à adopter pour l'algorithme de confiance.

Comme le microprocesseur procède à une lecture (ou un *fetch*) par transactions de 8 mots, nous pouvons donc construire notre algorithme de confiance sous forme d'une succession de blocs de base (*basic blocks*, en anglais) de 8 mots qui se terminent par un branchement indirect. Ainsi, *CoreSight* émet une trace à chaque transaction sur le bus AXI et, lors d'une exécution linéaire des blocs de base, nous pouvons prévoir le délai entre un accès sur le bus AXI et l'émission des paquets par *CoreSight*.

Notre algorithme de confiance est développé pour le jeu d'instruction ARMv7 / ARM, ce qui signifie que chaque instruction occupe un mot de 32 bits. La dernière instruction de chaque bloc de base est un branchement indirect vers le premier mot du bloc suivant, soit vers l'instruction suivante. Une instruction supplémentaire est nécessaire pour calculer sa destination. En conséquence, pour chaque bloc de base de 8 instructions, l'algorithme de confiance contient 6 instructions utiles et 2 instructions dédiées au branchement indirect.

Le code représenté sur le listing 6.1 schématise son architecture. Ici `pc` représente le registre contenant le pointeur d'instruction (*Program Counter*) et `r3` un registre général. Dans notre implémentation de l'algorithme de confiance, les `nop` sont remplacées par des instructions utiles à la désactivation des mémoires cache, la configuration de la MMU et la configuration de *CoreSight*.

```
nop
nop
nop
nop
nop
nop
add r3, pc, #0  // pc = adresse de l'instruction fetchee (+ 8 octets)
mov pc, r3      // branchement indirect

// <- destination
nop
nop
nop
nop
nop
nop
add r3, pc, #0  //
mov pc, r3      // branchement indirect
```

Listing 6.1 – Branchement indirect régulier

La troisième étape de notre caractérisation consiste à réaliser, depuis le microprocesseur, une exécution de notre algorithme de confiance depuis un esclave AXI-Lite et observer les traces émises par *CoreSight* en fonction des signaux sur le bus AXI.

CoreSight émet un paquet, lorsque sa FIFO contient suffisamment de données. Lors de l'émission d'un paquet, *CoreSight* baisse le signal nommé $TRACE\_CTL$ et place la donnée

sur le vecteur *TRACE_DATA* [62]. L'instant à partir duquel le paquet est émis après le début du *fetch* dépend de l'état de la FIFO de *CoreSight* avant l'exécution du code et de la valeur de l'adresse virtuelle à laquelle l'esclave AXI-Lite est accessible. Pour obtenir une trace dans une fenêtre de temps la plus courte après une transaction sur le bus AXI, nous devons au préalable vider la FIFO de *CoreSight* et choisir une adresse virtuelle faible pour l'esclave AXI-Lite [1]. En effet, le nombre qui encode l'adresse a une influence sur l'instant de transmission des paquets. Nous avons observé qu'un nombre faible provoque une transmission du paquet plus tôt. Le choix de ce nombre n'influe cependant pas sur la taille du paquet : seulement l'instant de début de transmission.

Le chronogramme représenté sur la figure 6.3 montre le résultat le plus optimisé (avec fenêtre de temps la plus courte) d'un *fetch* et de l'exécution de code entre les adresses `0x00200000` et `0x0020003C`, où `0x00200000` représente l'adresse virtuelle permettant d'accéder à l'adresse `0x00` dans l'esclave AXI-Lite. Les instants indiqués par les marqueurs bleus représentent respectivement l'instant du premier accès sur le bus AXI et l'instant du premier décodage de paquet transmis par *CoreSight*. Le signal *DECODED_ADDR* représente la valeur de l'adresse décompressée et décodée par le moniteur depuis le paquet *CoreSight*. Sa valeur n'est disponible qu'après une transmission complète et donne, indirectement, la valeur du pointeur d'instruction. La première valeur obtenue pour l'adresse décodée est `0x00200000`, soit lors de l'entrée du pointeur d'instruction dans la plage d'adresses à tracer, et la seconde valeur est `0x00200020`, soit la destination du premier branchement indirect.

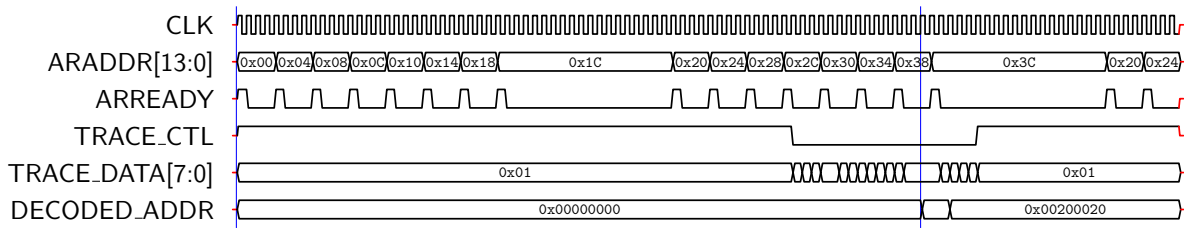

FIGURE 6.3 – Fetch et exécution d'instructions depuis l'esclave AXI-Lite

Les informations qui nous intéressent sont donc :

— le délai (nombre de fronts d'horloge) entre le premier accès sur le bus AXI et l'instant du décodage de paquet transmis par *CoreSight* ;

— la valeur de l'adresse décodée dans le paquet.

Si nous observons les mêmes délais et les mêmes adresses, nous pouvons décider que le microprocesseur effectue un bien un *fetch* et pas une simple lecture.

Ressources : le dossier `6_2_3_carac_fetch/` contient les sources du périphérique de confiance, d'un FSBL et d'une application pour exécuter des branchements indirects situés dans l'esclave AXI-Lite. Ceci permet d'observer les signaux représentés sur la figure 6.3.
Un script gdb permet d'instrumenter openOCD pour la programmation.
Un Makefile permet d'automatiser toutes ces tâches.

6.2.4 Contraintes de conception

En admettant que notre implémentation soit sécurisée, notons que nos choix d'architecture impliquent certaines contraintes.

Une conséquence directe est la réduction des performances du microprocesseur durant l'exécution de l'algorithme de confiance. Tout d'abord, la synchronisation du périphérique d'interconnexion AXI maître sur l'horloge du FPGA ralentit le microprocesseur lors du *fetch*. Notons que ce n'est pas le cas pour la synchronisation de *CoreSight* : l'émission de paquets par *CoreSight* ne ralentit pas l'exécution [61].

Également, la présence de branchements indirects provoque un vidage du *pipeline* et un *fetch* à l'adresse de destination. Comme le microprocesseur réalise ses *fetch* du bloc de base suivant durant l'exécution des instructions du bloc de base courant, atteindre l'exécution de la dernière instruction provoque un second *fetch* du même bloc de base [1]. Le premier chargement est certainement exécuté par le *pre-fetcher*, alors que le second chargement est exécuté à la suite de l'exécution du branchement indirect. En effet, nous rappelons que les mémoires cache sont désactivées. Ainsi, à l'exception de la première, chaque transaction est observée deux fois sur le bus AXI. Le *fetch* qui a un effet sur le fil d'exécution du programme est donc celui de la seconde transaction. Le chronogramme représenté sur la figure 6.4 montre les instants des *fetch* du branchement indirect et de l'instruction placée à sa destination. Dans cet exemple, les instructions placées aux adresses `0x0020001C` et `0x0020003C` sont des branchements indirects vers l'instruction suivante.

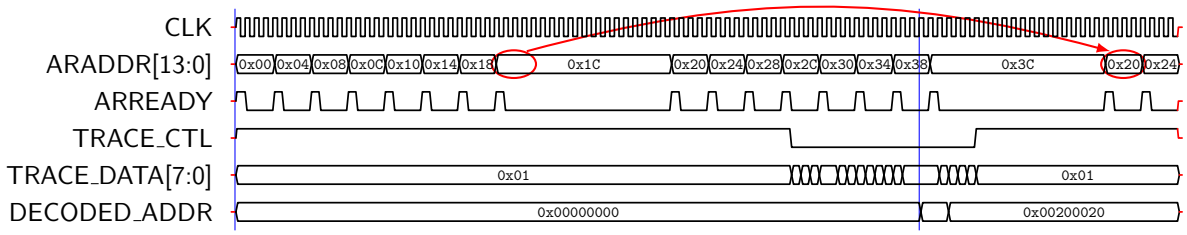

FIGURE 6.4 – Instants des *fetch* : branchement indirect et destination

Une seconde contrainte intervient lors du développement de l'algorithme de confiance : c'est l'alignement des blocs de base sur la taille de la ligne de cache. En effet, si la destination d'un branchement se trouve dans le même bloc de base que l'instruction qui l'a provoqué, alors le vidage du *pipeline* provoque une exception de type *Prefetch Abort* [1]. Notre choix d'architecture pour l'algorithme de confiance satisfait ces contraintes : c'est une succession de blocs de base, de la taille de la ligne de cache, qui se terminent par un branchement indirect vers le bloc suivant.

Une troisième contrainte est que des prérequis sont nécessaires avant l'exécution de l'algorithme de confiance. En effet, le moniteur matériel attend des signaux correspondant à une certaine configuration des périphériques. Ainsi, avant l'exécution de l'algorithme de confiance, les prérequis suivants doivent être appliqués.

1. Comme une lecture de la donnée dans l'esclave AXI-Lite doit être visible sur le bus AXI, de telles données ne doivent pas être présentes en cache. Un vidage des mémoires cache est donc réalisé.
2. Le moniteur matériel attend de *CoreSight* un paquet indiquant une entrée dans une plage d'adresses où les traces sont actives. *CoreSight* est donc configurée de façon à ce que les adresses virtuelles permettant l'accès à l'esclave AXI-Lite soient tracées.

3. Le moniteur matériel vérifie la valeur des adresses données par *CoreSight*. La MMU est donc configurée de telle sorte à ce que les adresses virtuelles soient identiques à celles attendues pour cette plage d'adresses. Les adresses choisies sont suffisamment faibles pour minimiser les délais de transmission des paquets par *CoreSight*.
4. Tout comme montré par Kovah et al. [34] pour les TPM, *CoreSight* n'utilise pas le même nombre de cycles d'horloge pour transmettre les mêmes données en fonction de son état initial. Les FIFO de ses composants sont initialisées en suivant toujours le même protocole, pour obtenir un résultat déterministe et forcer *CoreSight* à transmettre les paquets le plus tôt possible (une calibration du matériel est requise pour réaliser cette étape).

Manquer à un de ces prérequis ne permet pas de reproduire le comportement de l'algorithme de confiance depuis un logiciel malveillant car les signaux d'accès se trouvent alors modifiés. Cependant, cela provoque un refus par le moniteur matériel d'une exécution légitime de l'algorithme de confiance et de la fonction d'attestation.

6.2.5 Nouvelle problématique

Les signaux représentés sur le chronogramme de la figure 6.3 peuvent être modélisés formellement ou sauvegardés dans une mémoire. Cette sauvegarde fournit donc une hypothèse sur le comportement du microprocesseur. Nous pouvons donc effectuer une vérification formelle de la sécurité de la configuration de l'environnement d'exécution pour la fonction d'attestation. Une propriété de sécurité importante, vérifiée par le moniteur, est qu'un accès à la fonction d'attestation (et indirectement au secret et à la pile exclusive) ne sera jamais autorisé si le moniteur n'observe pas des signaux identiques. La vérification formelle de cette propriété est décrite dans la section 6.4.

Néanmoins, nous faisons désormais face à une nouvelle problématique. Notre problématique initiale était de savoir comment distinguer une exécution de l'algorithme de confiance d'une simple lecture. À priori, nous avons montré que cette distinction était possible, en observant à la fois les signaux des transactions sur le bus AXI et des paquets fournis par *CoreSight*. Cependant, **pour garantir la sécurité de la configuration de l'environnement d'exécution de la fonction d'attestation, il nous faut désormais déterminer si un adversaire peut reproduire ces signaux sans exécuter l'algorithme de confiance**.

S'il est possible de reproduire ces signaux sans exécuter notre algorithme de confiance, nous devons malheureusement revoir notre stratégie. En effet, nous ne pourrons pas distinguer une exécution légitime d'une attaque : les propriétés de sécurité vérifiées sur le moniteur ne sont donc pas suffisantes pour garantir la sécurité de la configuration de l'environnement.

En contrepartie, s'il n'est pas possible de reproduire ces signaux sans exécuter notre algorithme de confiance, observer des signaux identiques implique que nous sommes bien en train de l'exécuter. Alors, les propriétés de sécurité vérifiées sur le moniteur sont suffisantes pour garantir que notre périphérique de confiance n'accordera jamais l'accès à la fonction d'attestation sans exécuter notre algorithme de confiance (et avoir configuré l'environnement correctement). Ainsi, nous garantissons, par extension, l'exécution sécurisée de la fonction d'attestation et la confidentialité du secret.

Afin de répondre à cette nouvelle problématique, nous envisageons deux possibilités :

— modéliser de manière exhaustive le comportement du microprocesseur et vérifier, à l'aide de méthodes formelles, l'incapacité d'un adversaire à reproduire les signaux sans

exécuter l'algorithme de confiance.
— ou conduire une étude empirique, en réalisant des attaques sur notre solution, et observer si nous pouvons y accorder un fort degré de confiance, sans toutefois n'apporter de garantie soutenue par une vérification formelle.

Nous avons choisi de cibler les microprocesseurs propriétaires pris sur étagère. Malheureusement, ceux-ci ne sont pas distribués avec leurs modèles ou leurs spécifications formelles. En conséquence, il n'est donc pas réalisable de modéliser de manière exhaustive leur comportement sans effectuer, au préalable, une étude d'ingénierie inverse. Nous proposons donc, dans la section suivante, de conduire une étude empirique en réalisant des attaques sur notre solution, avec pour objectif la prévention de vulnérabilités.

6.3 Prévention de vulnérabilités

Afin de décharger le périphérique de confiance des hypothèses simplifiantes **H6** et **H7**, nous émettons l'hypothèse suivante, que nous nommons **H10** : **nous pouvons écrire notre algorithme de confiance tel qu'il est impossible pour un adversaire de reproduire les signaux d'accès lus par le moniteur sans l'exécuter**.

Pour valider si notre hypothèse **H10** est réaliste ou non, nous devons conduire une étude empirique, en réalisant des attaques sur le matériel concret. Dans cette section, nous proposons de jouer le rôle d'un adversaire et d'attaquer le système. L'objectif est de tenter de reproduire les mêmes signaux depuis un logiciel malveillant.

Nous considérons un adversaire possédant de hauts privilèges, une connaissance de l'architecture et un clone du système physique. Nous observons les signaux internes à l'aide d'un analyseur logique. Pour chaque attaque, notre logiciel malveillant est placé dans la DDR et est développé en assembleur ARM afin d'être au plus près du matériel.

6.3.1 Accès sur le bus AXI

La première attaque de notre étude consiste à trouver le moyen le plus optimisé pour reproduire les accès en lecture sur le bus AXI. Nous effectuons donc une simple lecture dans l'esclave AXI-Lite comme si son contenu était de la donnée. Notre objectif est d'être le plus rapide possible pour tenter d'obtenir les mêmes délais qu'une exécution légitime de l'algorithme de confiance. Nous ne pouvons malheureusement pas nous appuyer sur les mémoires cache pour gagner du temps. En effet, si le contenu de l'esclave AXI-Lite est déjà en cache, effectuer une lecture n'a aucun effet sur le bus AXI. Un prérequis pour jouer cette attaque est donc de désactiver les mémoires cache.

L'instruction de **chargement multiple** *LoaD Multiple* (`ldm`) est une instruction ARM qui provoque un ou plusieurs accès mémoire en lecture. Voici un exemple d'appel de cette instruction : `ldm r0!, {r1, r2}`. L'adresse à laquelle on doit lire est spécifiée dans un registre passé en premier argument ; le nombre de mots à lire est spécifié par le nombre de registres listés en deuxième argument : chaque mot lu est stocké dans un des registres. Si un point d'exclamation est présent sur le premier argument, ce qui est facultatif, alors l'adresse de lecture est incrémentée avec le nombre d'octets lus [49]. Si l'adresse de lecture est localisée dans un composant AXI, alors c'est le module d'interconnexion qui se charge de communiquer avec ce composant.

Réaliser une lecture de 8 mots dans l'esclave AXI-Lite peut donc se faire en chargeant son adresse virtuelle dans un registre et en appelant cette instruction plusieurs fois. Le code représenté sur le listing 6.2 montre une attaque tentant de reproduire les accès sur le bus AXI en utilisant un minimum de registres. La présence du `!` après le nom du registre contenant l'adresse de lecture provoque un incrément égal au nombre d'octets lus [49].

```
mov r0, #0x00200000 // adresse virtuelle
ldm r0!, {r1}       // r0 = r0 + 4 (!)
ldm r0!, {r1}
ldm r0!, {r1}
ldm r0!, {r1}
ldm r0!, {r1}
ldm r0!, {r1}
ldm r0!, {r1}
ldm r0!, {r1}
```

Listing 6.2 – Reproduction des accès sur le bus AXI: 8 lectures de 1 mot

Dans cet exemple, le registre `r1` est utilisé pour stocker le contenu de la donnée lue dans l'esclave AXI-Lite. L'avantage d'une telle approche pour un adversaire est qu'elle ne nécessite l'utilisation que de deux registres (ici `r0` et `r1`) pour réaliser une lecture de 8 mots. L'inconvénient est qu'elle nécessite l'exécution de 8 instructions (une fois l'adresse virtuelle chargée). Or, comme vu précédemment, un accès à l'esclave AXI-Lite synchronisé sur l'horloge du FPGA ralentit l'exécution du microprocesseur. Ainsi, la présence de plusieurs instructions introduit un délai de 15 périodes d'horloge entre chaque lecture de 1 mot. Le chronogramme représenté sur la figure 6.5 montre les signaux d'accès sur le bus AXI. Le signal CLK représente l'horloge du FPGA, il est commun aux deux représentations des signaux $ARADDR$ et $ARREADY$. La première représentation donne les signaux d'accès lors d'un *fetch* tandis que la seconde représentation donne les signaux d'accès lors de lectures de 1 mot.

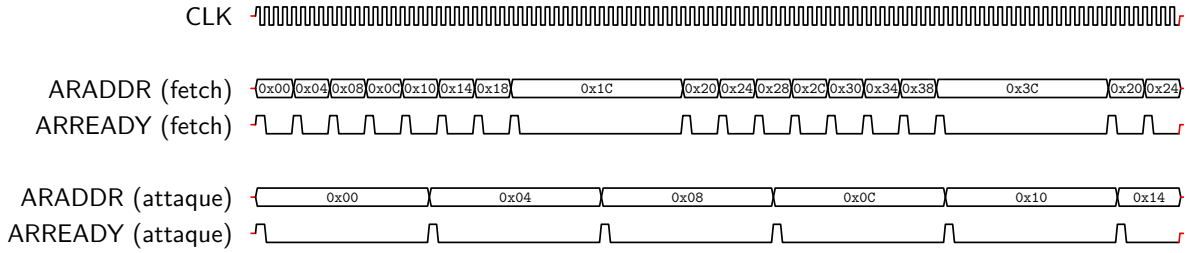

FIGURE 6.5 – Accès sur le bus AXI : *fetch* et lectures de 1 mot

Accéder à la donnée contenue dans l'esclave AXI-Lite par des lectures de 1 mot ne permet pas de reproduire les mêmes signaux qu'un *fetch* sur le bus AXI. Cependant, l'instruction `LDM` permet également d'effectuer plusieurs lectures de donnée en une seule fois si une liste de registres de destination lui est fournie [49]. Ainsi, nous pouvons modifier le code pour réaliser une lecture de 8 mots en stipulant 8 registres de destination.

Ressources : le dossier `6_3_1_attaque_read/` contient les sources d'un FSBL et d'une application pour effectuer 8 lectures de 1 mot par transaction. Ceci permet d'observer les signaux représentés sur la figure 6.5. Un script gdb permet d'instrumenter openOCD pour la programmation.
Un Makefile permet d'automatiser toutes ces tâches.

La présence de branchements indirects, dans l'algorithme de confiance, force une exécution légitime à accéder deux fois au même bloc de base (8 mots de 4 octets). Pour reproduire ce comportement, nous pouvons jouer sur l'utilisation du `!` pour ne pas incrémenter l'adresse de lecture lors du premier accès. Le code représenté sur le listing 6.3 montre une attaque qui reproduit les accès sur le bus AXI lors d'un *fetch*.

```
mov r0, #0x00200000 // adresse virtuelle
ldm r0!, {r1, r2, r3, r4, r5, r6, r7, r8}
ldm r0,  {r1, r2, r3, r4, r5, r6, r7, r8} // pas de ! = 2 lectures
ldm r0!, {r1, r2, r3, r4, r5, r6, r7, r8}
```

Listing 6.3 – Reproduction des accès sur le bus AXI: 1 lecture de 8 mots

Dans cet exemple, la première exécution de l'instruction `LDM` effectue une lecture de 8 mots dans l'esclave AXI-Lite à l'*offset* `0x00`. Les deux autres exécutions effectuent deux fois la même lecture à l'*offset* `0x20`. Comme la troisième instruction `LDM` possède un `!`, l'accès suivant sera réalisé à l'*offset* `0x40`. Notons qu'il est impératif d'avoir désactivé les mémoires cache au préalable sans quoi la deuxième lecture à l'offset `0x20` n'a pas lieu.

Avec cette approche, il est possible pour un adversaire de reproduire les mêmes signaux qu'un *fetch* dans l'esclave AXI-Lite sans exécuter le code qui s'y trouve. Le nombre de registres à utiliser est cependant plus conséquent. Il est nécessaire d'utiliser 9 registres : 1 registre pour stocker l'adresse de lecture (`r0`) et 8 registres pour stocker les 8 mots lus.

Ressources : le dossier `6_3_1_attaque_read/` contient les sources d'un FSBL et d'une application pour effectuer des lectures de 8 mots par transaction et observer les signaux représentés sur la figure 6.2. Un script gdb permet d'instrumenter openOCD pour la programmation.
Un Makefile permet d'automatiser toutes ces tâches.

6.3.2 Signaux de *CoreSight*

Pour compléter notre attaque, nous devons également reproduire les signaux sur l'interface de débug *CoreSight* durant l'exécution de notre logiciel malveillant. Dans un premier temps, nous nous concentrons sur la problématique de l'envoi de traces par *CoreSight* avec des valeurs correctes (identiques à celles de l'algorithme de confiance) pour les adresses de destination. Notre objectif est donc d'ajouter des instructions qui provoquent un branchement indirect et forcer *CoreSight* à envoyer une trace au même instant que lors d'un *fetch*. Un prérequis pour jouer cette attaque est donc de configurer *CoreSight* pour que les adresses virtuelles de notre logiciel malveillant soient tracées. Un autre prérequis est de configurer la MMU pour que les adresses virtuelles permettant l'accès à ce même logiciel malveillant soient identiques à celles attendues par le moniteur matériel. L'adresse virtuelle pour accéder à l'esclave AXI-Lite doit donc également être modifiée pour être différente.

Exactement comme pour l'algorithme de confiance, nous ajoutons dans notre logiciel malveillant deux instructions permettant d'introduire un branchement indirect. La première instruction permet de calculer l'adresse de destination tandis que la seconde effectue le branchement. L'adresse de destination est choisie afin de conserver le même *offset* que l'algorithme de confiance. Ainsi, nous ajoutons des instructions inutiles que le branchement indirect nous permet de passer. Le code représenté sur le listing 6.4 montre une attaque tentant de reproduire les signaux de *CoreSight*.

```
mov r0, #0x40000000 // nouvelle adresse virtuelle (AXI-Lite)
ldm r0!, {r1, r2, r3, r4, r5, r6, r7, r8}
add r9, pc, #16     //
mov pc, r9          // branchement indirect
nop
nop
nop
nop

// <- destination (adresse virtuelle = 0x00200020)
ldm r0,  {r1, r2, r3, r4, r5, r6, r7, r8} // pas de ! = 2 lectures
ldm r0!, {r1, r2, r3, r4, r5, r6, r7, r8}
add r9, pc, #16     //
mov pc, r9          // branchement indirect
// ...
```

Listing 6.4 – Reproduction des signaux de *CoreSight*

Dans cet exemple, le registre `r9` est utilisé pour stocker l'adresse de destination du branchement indirect. L'instruction `add` placée après les accès sur le bus AXI permet de calculer cette adresse. L'avantage d'une telle approche est qu'elle ne nécessite l'utilisation que d'un seul registre (ici `r9`) pour calculer l'adresse de destination, et ce pour tous les branchements indirects nécessaires. L'inconvénient est qu'elle nécessite l'ajout d'une instruction `add`, en plus du branchement indirect, entre deux lectures sur le bus AXI. Or, l'ajout d'instructions entre deux lectures sur le bus AXI introduit un délai entre les accès. Le chronogramme représenté sur la figure 6.6 montre les signaux observés par le moniteur. La première représentation donne les signaux d'accès lors d'un *fetch* tandis que la seconde représentation donne les signaux d'accès lors de notre attaque : à savoir, une lecture, un calcul de l'adresse de destination et un branchement indirect.

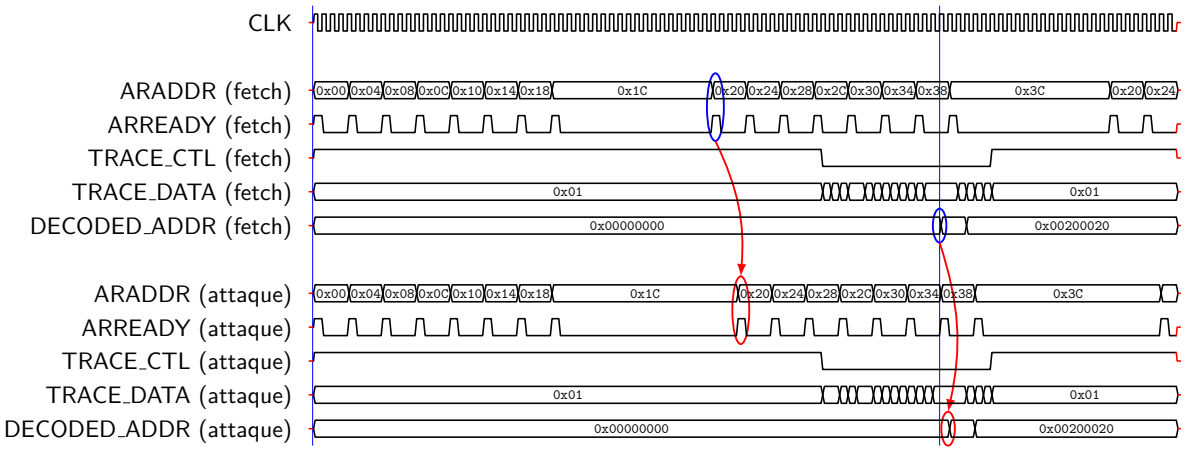

FIGURE 6.6 – Branchement indirect, calcul de la destination entre les lectures

L'ajout de notre branchement indirect dans le logiciel malveillant permet d'obtenir la bonne valeur d'adresse décodée. Cependant, avoir le calcul de l'adresse de destination placé entre deux lectures sur le bus AXI introduit un délai. L'instant de la deuxième lecture, entourée en rouge sur les chronogrammes de *ARADDR* et *ARREADY*, est en retard de 3 périodes d'horloge par rapport au deuxième *fetch*, entouré en bleu sur les chronogrammes des mêmes signaux. Nous avons supposé que le microprocesseur était ralenti car il n'était pas possible de brancher

à l'adresse pointée par `r9` avant d'avoir terminé d'exécuter l'instruction `add`. C'est-à-dire que nous avions une dépendance entre le résultat écrit par le `add` et la destination lue par le `mov` (*read after write data dependency*). Cette supposition sous-entend que le microprocesseur ajoute une bulle dans le *pipeline* afin de terminer le calcul de la destination avant l'exécution du branchement. Or, il s'avère que ce retard est présent même si nous ajoutons une instruction `add` qui stocke le résultat dans un registre différent de `r9`. C'est donc bel et bien l'ajout d'un calcul qui ralentit les accès sur le bus AXI.

La transmission de données par *CoreSight* n'est, quand à elle, en retard que d'une seule période d'horloge. Cela s'explique par le fait que *CoreSight* procède à une collecte des informations de débug sans ralentir le microprocesseur. En réalité, la transmission de la trace s'effectue au même instant que lors d'un *fetch* : le signal $TRACE\_CTL$ est baissé au même front d'horloge. La seule différence entre ces deux transmissions est que *CoreSight* transmet une donnée différente en fonction de l'état de sa FIFO lors de l'émission du paquet. Avoir un logiciel malveillant optimisé peut potentiellement permettre d'obtenir la même donnée transmise par *CoreSight*.

Notons que cette affirmation est seulement vraie dans le cas de la première lecture car, dans le cas où les accès sur le bus AXI diffèrent de lors d'un *fetch*, les FIFO de *CoreSight* ne seront pas dans le même état lors du second branchement indirect. En effet, le délai causé par la première lecture se répercute sur l'instant d'exécution du second branchement indirect.

Ressources : le dossier `6_3_2_attaque_coresight/` contient les sources d'une application pour exécuter l'attaque représentée par le listing 6.4 et observer les signaux représentés sur la figure 6.6. Un script gdb permet d'instrumenter openOCD pour la programmation.
Un Makefile permet d'automatiser toutes ces tâches.

6.3.3 Synchronisation des lectures et de *CoreSight*

Comme montré précédemment, l'ajout de calculs entre les lectures dans l'esclave AXI-Lite introduit un délai. Désormais, nous nous concentrons donc sur la problématique de la synchronisation entre les accès sur le bus AXI et l'envoi de traces par *CoreSight*.

Pour cela, nous retirons le calcul de l'adresse de destination d'entre les lectures sur le bus AXI. Différents registres sont donc préchargés avant l'exécution du logiciel malveillant et seules les instructions de branchement sont conservées. Chaque branchement indirect utilise donc un registre différent pour obtenir l'adresse de destination. Le code représenté sur le listing 6.5 représente cette attaque.

```
// destinations des branchements dans r9, r10, r11 et r12:
movw r9, #0x0020
movt r9, #0x0020    // r9  = 0x00200020
add r10,  r9, #0x20 // r10 = 0x00200040
add r11, r10, #0x20 // r11 = 0x00200060
add r12, r11, #0x20 // r12 = 0x00200080

mov r0, #0x40000000 // adresse virtuelle (AXI-Lite)
ldm r0!, {r1, r2, r3, r4, r5, r6, r7, r8}
mov pc, r9          // branchement indirect
nop
nop
nop
nop
```

```
nop

// <- destination (adresse virtuelle = 0x00200020)
ldm r0,  {r1, r2, r3, r4, r5, r6, r7, r8} // pas de ! = 2 lectures
ldm r0!, {r1, r2, r3, r4, r5, r6, r7, r8}
mov pc, r10          // branchement indirect
// ...
```

Listing 6.5 – Préchargement des adresses de destination

Avec cette approche, les signaux d'accès sur le bus AXI ne sont pas ralentis. Également, le code situé après la première lecture se trouve optimisé. Certaines de ses exécutions permettent de reproduire exactement les signaux d'accès sur le bus AXI et sur les données transmises par *CoreSight*. L'inconvénient, néanmoins, est que cette approche ne permet pas de précharger les adresses de destination à l'infini : le nombre de registres du microprocesseur constitue une limite physique.

Cette attaque nous permet donc de reproduire les mêmes signaux d'accès sans exécuter l'algorithme de confiance. Toutefois, une limitation est que cette reproduction n'est valable que pour les six premiers branchements indirects. En effet, le microprocesseur ARM que nous utilisons possède 13 registres généraux et 3 registres spéciaux, dont le registre `pc` qui contient le pointeur d'instruction [49]. Un total de 15 registres est donc accessible à l'adversaire : 9 registres sont utilisés pour reproduire les signaux d'accès sur le bus AXI et 6 registres peuvent être utilisés pour précharger les adresses de destination des branchements indirects.

Ressources : le dossier `6_3_3_attaque_synchro/` contient les sources d'une application pour exécuter l'attaque représentée par le listing 6.5 et observer les signaux représentés sur la figure 6.3. Un script gdb permet d'instrumenter openOCD pour la programmation. Un Makefile permet d'automatiser toutes ces tâches.

6.3.4 Bilan

Nous considérons notre hypothèse **H10** valide si l'algorithme de confiance, présent dans l'esclave AXI-Lite, contient 7 branchements indirects ou plus. En effet, il devient alors nécessaire pour un adversaire de calculer une nouvelle adresse de destination entre deux accès sur le bus AXI, ce qui introduit un délai qui est détecté par le moniteur. Notre algorithme de confiance est donc impacté en ce sens (ainsi que les signaux attendus par le moniteur). Nous considérons donc que nous pouvons accorder un fort degré de confiance en notre hypothèse : nous pouvons écrire notre algorithme de confiance tel qu'il est impossible pour un adversaire de reproduire les signaux d'accès lus par le moniteur sans l'exécuter.

6.4 Vérification formelle

En considérant que notre hypothèse **H10** est valide, nous pouvons construire notre nouveau moniteur matériel tel que la fonction d'attestation ne peut pas être exécutée si les signaux observés ne sont pas exactement reproduits. Nous réalisons une vérification formelle sur ce moniteur pour garantir la sécurité de la configuration de l'environnement d'exécution sur le modèle 1. Dans cette section, nous décrivons la procédure de cette vérification formelle. L'annexe D fournit des détails techniques sur l'architecture du nouveau moniteur et les spécifications vérifiées.

6.4.1 Modèle d'environnement

Dans cette section, nous décrivons l'environnement du nouveau moniteur matériel, à savoir la communication avec la ROM contenant les valeurs attendues des signaux et l'observation de ces mêmes signaux durant la vérification.

6.4.1.1 Connexions à la ROM

Notre nouveau moniteur matériel accède aux valeurs attendues des signaux depuis une ROM de $8Ko$. Afin d'implémenter cette ROM, nous instancions un bloc RAM sur lequel nous forçons la désactivation d'écriture à l'aide des bits prévus à cet effet. Cette ROM fournit un vecteur *rddata*, de 64 bits de donnée, qui fournit le contenu stocké à l'adresse demandée par le moniteur [50]. Un vecteur d'adresse permet de définir à quel index la donnée est lue. Les valeurs attendues au premier front d'horloge sont stockées à l'adresse 0, celles attendues au second front d'horloge sont stockées à l'adresse 1, etc.

Nous modélisons le comportement de la ROM à partir d'une traduction depuis la documentation du *SoC* [50]. Nous exprimons formellement ce modèle sous forme d'une propriété temporelle. Cette propriété est considérée axiomatique. L'expression 6.1 fournit ce modèle, où i représente l'adresse placée le moniteur sur le vecteur d'adresse, n représente un entier naturel quelconque et $ROM(n)$ représente la valeur stockée dans la ROM à l'adresse n.

$$\textbf{forall } n \textbf{ in } \{0:1023\}: \ \textbf{always}((i = n) \leftrightarrow \textbf{next}(rddata = ROM(n))) \tag{6.1}$$

6.4.1.2 Arrêt de la vérification

Le moniteur observe les signaux et procède à un *reset* du microprocesseur en cas de différence avec les valeurs attendues. Cette vérification n'a pas lieu en permanence : seulement lors de l'exécution de l'algorithme de confiance. Elle débute au premier marqueur bleu représenté sur la figure 6.3. Le vecteur d'adresse est alors incrémenté à chaque front d'horloge. La vérification se termine lorsqu'un index END_ADDR est atteint sur le vecteur d'adresse, choisi lorsque PC atteint la fonction d'attestation (CR_{min}). L'expression 6.2 décrit ce comportement, que nous considérons axiomatique.

$$\textbf{always}(\textbf{next}(i = END\_ADDR) \rightarrow (PC = CR_{min})) \tag{6.2}$$

La vérification est suivie par un état de fin, que le moniteur quitte lorsque le transducteur indique que nous avons terminé d'exécuter la fonction d'attestation ($PC_d = CR_{max}$). Des détails techniques sur cette fonctionnalités sont fournis dans la section C.1 de l'annexe C.

6.4.2 Définition de la sécurité

Notre définition de la sécurité de l'attestation à distance sur le modèle 1 est un raffinement de celle sur le modèle zéro. Nous intégrons à celle-ci une propriété supplémentaire : conformément à notre hypothèse **H10**, **lors d'une exécution depuis l'esclave AXI-Lite, il est impossible d'obtenir une différence entre les signaux observés et les valeurs attendues jusqu'à la fin de l'algorithme de confiance**. Autrement dit, le moniteur émet un *reset* si et seulement si une déviation apparait. Cette extension de la définition se traduit formellement par plusieurs spécifications.

L'expression 6.3 fournit la définition formelle d'une spécification, celle qui stipule qu'il est impossible d'obtenir une différence entre les signaux observés et les valeurs attendues. Cette définition considère que les valeurs attendues sont concaténées dans les bits de poids faible des vecteurs de donnée contenus dans la ROM. Les prédicats qui modélisent les signaux représentés sur la figure 6.3 portent les mêmes noms que ces signaux. Le prédicat $verifying$ indique que le nouveau moniteur procède à la vérification, soit que l'algorithme de confiance s'exécute.

$$
\begin{aligned}
&\textbf{always}(\ verifying \wedge \neg(\\
&\qquad (sw\_araddr = rddata[13:0]) \\
&\quad \wedge (sw\_arready = rddata[14]) \\
&\quad \wedge (trace\_ctl = rddata[15]) \\
&\quad \wedge (trace\_data = rddata[23:16]) \\
&\quad \wedge (decoded\_address = rddata[55:24]) \\
&\quad) \\
&\quad \rightarrow reset \\
&\ )
\end{aligned}
\tag{6.3}
$$

Notre extension de la définition de la sécurité ne se limite pas à cette seule expression, nous exprimons d'autres spécificationss, qui décrivent que :

- le vecteur $rddata$ correspond à la donnée, lue dans la ROM, à l'index du front d'horloge auquel la vérification doit être réalisée.
- le prédicat $verifying$ est vérifié lors de l'exécution de l'algorithme de confiance. Ceci implique de définir formellement les instants de vérification et apporte donc des obligations de preuves supplémentaires.

L'annexe D fournit des détails techniques sur l'environnement et l'architecture du nouveau moniteur matériel. La section D.4 détaille la vérification et inclut la définition formelle de tous les spécifications qui expriment formellement la sécurité sur le modèle 1.

6.4.3 Établissement de théorèmes

Au travers d'une combinaison de *model-checking* et de preuves, nous établissons des théorèmes exprimant que nos spécifications sont vérifiées par le moniteur. Des propriétés locales vérifiées sur le nouveau moniteur permettent de prouver la spécification représentée par l'expression 6.3. La modélisation de la ROM (expression 6.1) permet de prouver que $rddata$ prend la valeur de la donnée lue à l'index du front d'horloge approprié. La modélisation de l'arrêt de la vérification (expression 6.2) et des propriétés locales vérifiées sur le transducteur (section C.1 de l'annexe C) permettent de prouver que la vérification est réalisée durant l'exécution de l'algorithme de confiance. La section D.4 de l'annexe D détaille la vérification formelle de toutes les spécifications qui expriment la sécurité.

Ressources : le programme `6_4_3_preuve_moniteur/proof.v` utilise Coq. Il permet de prouver la spécification représentée par l'expression 6.3 depuis des propriétés locales vérifiées sur le moniteur. Les propriétés locales sont converties à l'aide de notre programme *psl2coq*.
Un `Makefile` permet d'automatiser la conversion des propriétés locales et la preuve.

6.5 Soundness

Dans ce chapitre, nous avons vérifié la sécurité apportée par le nouveau moniteur. Cependant, le périphérique de confiance, avec l'extension que nous proposons ici, ne doit pas entraver l'exécution de la fonction d'attestation. Dans cette section, nous décrivons comment nous définissons et vérifions la *soundness* de cette extension.

6.5.1 Définition

De la même manière que dans le chapitre 5, nous définissons la *soundness* de notre solution comme suit : une exécution complète de la fonction d'attestation est toujours possible, malgré l'ajout de notre nouveau moniteur. Également, nous ne vérifions pas la *soundness* en manipulant les propriétés locales vérifiées sur le nouveau moniteur : nous exécutons une fonction d'attestation sur le système concret et observons le résultat. Pour rappel, une justification de cette stratégie est donnée dans la section 5.6.2.

6.5.2 Stratégie de vérification

Nous réalisons les étapes suivantes pour vérifier la *soundness* de notre solution :

— le FPGA est programmé de telle sorte à ce que le périphérique de confiance représenté sur la figure 5.1, ainsi que son extension représentée sur la figure 6.1, soient implémentées. Une LED témoigne du *reset* transmis par les moniteurs.
— la ROM contenant les valeurs des signaux attendus est programmée avec les valeurs observées lors d'une phase de calibration.
— un algorithme de confiance contenant 7 branchements indirect est présent dans un esclave AXI-Lite, suivi d'une fonction d'attestation qui réalise un XOR entre le secret (situé dans un esclave AXI-Lite) et le challenge (situé dans la DDR).
— une application est chargée dans la DDR, elle réalise une écriture du challenge, une exécution de la fonction d'attestation et une lecture du résultat.

Nous considérons que notre solution est *sound* dans le cas où la LED ne s'allume pas et que le résultat d'une exécution de la fonction d'attestation est égal au XOR entre le challenge et le secret.

La difficulté de la vérification de la *soundness* est que le résultat de la calibration peut varier en fonction de la carte de développement. Deux cartes de développement différentes peuvent fournir des signaux différents, même si les modèles sont identiques [34]. Nous proposons donc deux environnements de travail différents :

— un environnement de calibration où le nouveau moniteur est absent : celui-ci est remplacé par un analyseur logique intégré. Une solution logicielle permet d'extraire les signaux attendus et fournir des valeurs de paramètres *Verilog* pour l'initialisation de la ROM.
— un environnement de vérification où la ROM est initialisée avec les valeurs de paramètres *Verilog* extraites depuis l'environnement précédent.

Nous réalisons donc une vérification après une calibration. Les environnements de développement permettant de réaliser ces tâches sont rendus accessibles publiquement [1].

Ressources : le dossier `6_5_2_calibration/` contient un environnement de développement permettant de calibrer notre nouveau moniteur implémenté sur une carte de développement Digilent Cora Z7-07S. Le résultat de la calibration est un fichier *Verilog* nommé `expected.v` et contenant un modèle de ROM correctement initialisé. Un exemple de résultat est fourni avec les valeurs calibrées sur notre carte de développement.
Un `Makefile` permet d'automatiser l'implémentation, la programmation et l'exécution de la calibration.

Ressources : le dossier `6_5_2_soundness/` contient un environnement de développement permettant de vérifier la *soundness* de notre nouveau moniteur implémenté sur une carte de développement Digilent Cora Z7-07S. Le modèle de ROM contenant les valeurs des signaux attendus est obtenu depuis le résultat de la calibration. L'application chargée dans la DDR permet de vérifier la *soundness*.
Un `Makefile` permet d'automatiser l'implémentation, la programmation et l'exécution de l'application.

6.6 Faiblesses admises

Dans le chapitre 5, nous avons admis des faiblesses pour notre modèle zéro, représentées par des hypothèses simplifiantes numérotées de **H5** à **H9**. Dans cette section, nous rappelons brièvement ces hypothèses simplifiantes et décrivons comment notre nouveau moniteur matériel et notre algorithme de confiance impactent sur celles-ci. Ceci décrit donc les contraintes de conception et faiblesses admises du modèle 1 du microprocesseur.

6.6.1 Contraintes de conception

L'hypothèse simplifiante **H5** considérait qu'un seul cœur du microprocesseur fonctionnait durant l'exécution de la fonction d'attestation. Dans ce chapitre, nous avons travaillé sur un *SoC* contenant un microprocesseur mono-cœur. Nous considérons donc que cette hypothèse est vérifiée dans ce contexte. Cependant, la solution que nous proposons dans ce chapitre ne s'applique elle aussi qu'aux microprocesseurs ne comportant qu'un seul cœur. En effet, notre étude de sécurité conclut que nous pouvons distinguer une lecture d'un *fetch* dans le cas où l'exécution de branchements indirects ralentit les accès sur le bus AXI. Cette affirmation n'est valable que dans le cas d'un système mono-cœur. Dans le cas d'un système multi-cœurs, l'exécution de branchements indirects peut être réalisée par un cœur tandis qu'un second cœur réalise les accès sur le bus AXI : ceux-ci ne sont pas ralentis. Nous modifions donc l'hypothèse simplifiante **H5** afin que celle-ci devienne : **le microprocesseur étendu par notre périphérique de confiance ne possède qu'un seul cœur**. Elle n'est donc plus une hypothèse simplifiante mais une contrainte de conception.

6.6.2 Etat du microprocesseur

Les hypothèses simplifiantes **H6** et **H7** considéraient que le microprocesseur se trouvait dans un état spécifique avant l'exécution de la fonction d'attestation. Dans ce chapitre, nous proposons une solution pour forcer l'état spécifique en question. Nous considérons donc que nous déchargeons notre périphérique de confiance des hypothèses simplifiantes **H6** et **H7**. Cependant, en l'absence de modèle pour le cœur du microprocesseur et ses périphériques, nous

avons dû émettre une hypothèse de non-reproductibilité des signaux d'accès (lors de la configuration de notre environnement). Afin d'argumenter sérieusement en faveur de cette hypothèse, nous avons conduit une étude empirique sur notre système et avons montré que, sous réserve de contraintes liées à l'architecture du microprocesseur, nous sommes dans l'incapacité d'infirmer notre hypothèse. Nous la considérons donc réaliste. Néanmoins, celle-ci reste toujours une hypothèse. Nous nommons donc cette hypothèse **H10**, que nous définissons ainsi : nous pouvons écrire notre algorithme de confiance tel qu'il est impossible pour un adversaire de reproduire les signaux d'accès lus par le moniteur sans l'exécuter.

6.6.3 Faiblesses matérielles

L'hypothèse simplifiante **H8** considérait que l'adversaire n'accédait pas au périphérique de programmation du FPGA. Dans ce chapitre, nous n'avons pas traité cette problématique et avons considéré que nous pouvions sécuriser le module ICAP situé dans la partie FPGA du *SoC* afin de vérifier l'hypothèse. Nous détaillons cette considération dans l'annexe E, sans toutefois n'apporter de garantie soutenue par une vérification formelle. Une faiblesse est donc toujours admise pour notre modèle 1 : à partir d'un accès distant, un adversaire peut reprogrammer le FPGA et désactiver les fonctionnalités apportées par le moniteur matériel.

Enfin, l'hypothèse **H9** considérait que les paquets *isync* et *branch* fournis par *CoreSight* étaient disponibles à l'état précédent l'exécution de l'instruction de destination. En effet, nous faisions abstraction du temps de transmission des paquets par *CoreSight*. Dans ce chapitre, nous proposons la vérification de la transmission des paquets vis-à-vis de valeurs attendues. Notre solution apporte donc un début de réponse concernant la vérification de l'hypothèse **H9**. En effet, le temps de transmission que nous avions abstrait dans le chapitre 5 ne peut pas être consacré à une autre tâche que l'exécution de notre algorithme de confiance. Cette affirmation est valable jusqu'à la réception par le nouveau moniteur de la trace causée par un branchement indirect à l'adresse CR_{min} (première instruction de la fonction d'attestation). Nous pouvons donc considérer que l'hypothèse simplifiante **H9** est vérifiée lors de l'entrée du pointeur d'instruction dans la région critique.

Néanmoins, lors de l'exécution de la fonction d'attestation, rien n'indique que nous pouvons abstraire le temps de transmission des paquets en cas d'une levée d'exception. Nous ne pouvons pas considérer **H9** comme vérifiée dans ce cas. Toutefois, nous pouvons nous interroger sur la possibilité de configurer, à l'aide de l'algorithme de confiance, l'environnement du microprocesseur en cas de levée d'exception. Ainsi, nous pouvons tenter de déterminer si une configuration spécifique peut permettre de placer le flot d'exécution en attente active (*spin lock*) jusqu'à la réception du paquet décompressé par le moniteur. Nous proposons une interrogation à ce sujet dans l'annexe E.

Lors de la sortie de la région critique par le pointeur d'instruction, nous ne pouvons pas abstraire le temps de transmission. Ceci est critique car les accès aux mémoires sensibles sont autorisés par le moniteur matériel jusqu'à la réception de la trace causée par un branchement indirect à l'adresse CR_{max}. Rien n'indique donc qu'un adversaire ne peut procéder à un accès à ces mémoires durant le temps de transmission. Nous considérons néanmoins qu'il est possible de remédier à cette faiblesse à l'aide d'une architecture co-conçue. Nous détaillons cette considération dans l'annexe E, sans toutefois n'apporter de garantie soutenue par une vérification formelle.

6.7 Conclusion

Dans le chapitre précédent, nous avons étendu le domaine d'application de l'attestation à distance formellement vérifiée aux microprocesseurs. Dans ce cadre, nous avons ciblé les microprocesseurs propriétaires pris sur étagères, tels que le ARM Cortex-A9. Cette proposition préliminaire suppose malheureusement que l'état du microprocesseur, avant l'exécution de la fonction d'attestation, est correct et sûr (voir la définition du modèle zéro, dans le chapitre 5). Dans ce chapitre, nous avons introduit une extension du périphérique de confiance afin de vérifier l'intégrité de son environnement et pallier aux précédentes hypothèses. Cela permet de garantir un environnement d'exécution correct pour la fonction d'attestation. Nous garantissons donc la sécurité de la fonction d'attestation sur notre modèle 1, où l'adversaire peut modifier la configuration du microprocesseur et de ses périphériques ainsi que manipuler les mémoires cache.

Notre architecture co-conçue logicielle et matérielle est implémentée sur une carte de développement Digilent Cora Z7-07S. À la manière des précédents travaux de ce domaine de recherche, notre solution s'appuie à la fois sur du support matériel [35, 36] et sur une mesure précise du comportement du microprocesseur (mesure des cycles d'horloge et des signaux associés à l'exécution de certaines instructions) lors de l'exécution d'un logiciel critique de configuration de l'environnement [33, 34]. Nous émettons l'hypothèse que nous pouvons écrire ce logiciel tel qu'il est impossible pour un adversaire de reproduire les signaux d'accès lus par le moniteur matériel sans l'exécuter. Une vérification formelle du moniteur démontre que la fonction d'attestation ne peut pas être exécutée si les signaux observés ne sont pas exactement reproduits. Ainsi, nous avons montré que notre extension au périphérique de confiance garantit la configuration de l'environnement d'exécution de la fonction d'attestation.

Les travaux de ce chapitre valident uniquement notre hypothèse en réalisant différentes attaques sans parvenir à l'infirmer. Nous n'avons donc pas prouvé formellement qu'il était impossible pour un adversaire de reproduire les signaux sans exécuter notre l'algorithme de confiance. De futurs travaux consistent donc a étayer nos modèles formels de façon à pouvoir formellement prouver ou réfuter l'hypothèse. Cette approche se limite donc toujours aux freins de l'absence de modèle et de l'aspect propriétaire de certain composants impliqués. Prouver notre hypothèse nécessite un modèle du microprocesseur ARM Cortex-A9 et de ses périphériques, sur lequel la preuve peut être réalisée (ce qui semble difficile compte-tenu de l'aspect propriétaire de cette architecture). Réfuter notre hypothèse nécessite de fournir un contre-exemple, en exécutant un logiciel malveillant qui reproduit les signaux d'accès. Dans le cas où de futures attaques, dont nous n'avons pas connaissance lors de l'écriture de ce chapitre, réfutent notre hypothèse, notre solution co-conçue devra être adaptée.

Notons qu'il est également possible de faire évoluer l'architecture de l'algorithme de confiance. S'il est possible de reproduire les signaux d'accès sans l'exécuter, il est nécessaire de vérifier que cela n'est pas dû à un défaut dans son implémentation plutôt qu'une invalidité de l'hypothèse. Elle n'est pas invalide si une évolution de l'algorithme de confiance empêche de nouveau la reproduction des signaux par un adversaire.

Enfin, afin de formellement compléter ces travaux, dans l'éventualité où un modèle du microprocesseur et de ses périphériques devient disponible, notre hypothèse pourra alors prendre la forme d'une obligation de preuve pour la sécurité globale de l'attestation à distance sur microprocesseur.

Conclusion et perspectives

Bilan Ces dernières décénies, la puissance de calcul des systèmes informatiques n'a fait qu'augmenter et les coûts de mise en production ont drastiquement réduit. Pour s'adapter, les constructeurs de matériels ont été poussés à la course à la performance, dans laquelle ils peuvent considérer plus rentable de rapidement proposer sur le marché un système imparfait que réaliser des vérifications exhaustives en matière de sécurité. De plus, les ressources apportées par ces matériels performants ont été exploitées par les concepteurs de systèmes pour les mutualiser et les densifier. Désormais, nombre d'applications hétérogènes complexes sont hébergées sur une même infrastructure matérielle, souvent distante et partagée par différents utilisateurs, potentiellement malveillants. Dans un tel contexte, sécuriser l'exécution d'un programme sur ces systèmes devient un problème difficile, même lorque le code source a été prouvé sécurisé.

Dans ce manuscrit, nous avons donc considéré comme problématique la sécurisation de l'exécution d'un algorithme dans un environnement complexe. Nous avons considéré que nous ne pouvons pas vérifier qu'à tout instant, cette exécution est sécurisée, même dans le cadre d'un adversaire distant. Néanmoins, il est possible de détecter une intrusion, une compromission de cet algorithme, à un instant donné et d'établir une racine de confiance dynamique. Pour cela, il existe le protocole d'attestation à distance. La sécurisation de l'exécution du protocole nécessite une racine de confiance statique, vis-à-vis des capacités d'un attaquant. Cette racine de confiance statique, lorsqu'elle est accompagnée d'une vérification formelle, a jusqu'à présent été réalisée sur des microcontrôleurs simples, tels que la famille des MSP430. L'obtenir sur ces systèmes nécessite des composants matériels dédiés, suffisamment simples pour que la vérification formelle reste possible.

Afin de permettre la vérification formelle de systèmes matériels plus complexes, nous avons proposé une méthode de vérification formelle automatique et industrialisable. L'objectif de cette méthode est de minimiser l'effort d'abstraction et ainsi réduire l'impact de l'explosion combinatoire. La solution consiste à créer des modèles abstraits, certifiés corrects, pour un système en fonction de propriétés choisies et adaptées à une preuve de théorème. Nous avons fourni une mise en œuvre de cette méthode sous forme d'un ensemble d'outils, composé uniquement de logiciels libres, que nous avons déployé tout au long de cette thèse. Cette méthode nous a donc permis d'envisager l'établissement d'une racine de confiance statique sur microprocesseur.

En nous appuyant sur les architectures des *SoC* modernes, nous avons donc envisagé d'étendre le domaine d'application de l'attestation à distance vérifiée formellement. Notre cas d'étude visait à sécuriser l'exécution d'une fonction d'attestation sur un microprocesseur pris sur étagère : le ARM Cortex-A9. Nous avons proposé une définition de la sécurité pour la racine de confiance statique sur cette famille de systèmes, en tenant compte de l'augmentation de la surface d'attaque et des capacités d'un adversaire privilégié. Nous avons décomposé ces capacités en des problématiques décorrélées, que nous avons pu traiter indépendamment pour permettre la vérification formelle. Nous avons ensuite réalisé une implémentation, que nous avons confrontée à notre définition de la sécurité et avons apporté la preuve que celle-ci était vérifiée. La solution que nous avons proposée ne nécessite pas de modification matérielle, mais simplement une extension sous forme d'un périphérique de confiance, implémenté dans un

circuit de logique programmable.

Considérer les architectures de microprocesseurs propriétaires apporte cependant une problématique : sans accès aux sources, nous ne pouvons pas modéliser formellement leur comportement. Vis-à-vis de notre modèle de menace fort, nous ne considérons donc pas que le microprocesseur constitue un environnement sûr. Nous avons donc proposé de reconstruire un mécanisme de gestion des privilèges externe à celui-ci, dans la partie matérielle de notre périphérique de confiance. Ce mécanisme s'appuie sur des hypothèses émises vis-à-vis du comportement du microprocesseur, que nous avons validées par une étude empirique pour pouvoir y accorder un fort degré de confiance. Ici aussi, la solution proposée est vérifiée formellement, compte tenu de ces hypothèses.

Perspectives Nos contributions abstraient toutefois certaines capacités de notre adversaire que nous envisageons de traiter lors de futurs travaux. La racine de confiance statique que nous avons établie ici est donc encore vulnérable à deux classes d'attaques, que nous avons identifiées. La première est celle où une reprogrammation partielle de la logique programmable est réalisable depuis le microprocesseur. La seconde est celle où des accès concurrents aux mémoires sensibles peuvent être réalisés à des instants critiques. Pour ces deux classes d'attaques, nous avons envisagé des extensions logicielles et matérielles, que nous pouvons modéliser et dont nous pouvons formellement vérifier la sécurité. Des détails techniques concernant ces extensions sont fournis en annexe E.

Également, nos contributions reposent sur des hypothèses de préservation sémantique lors des transformations de nos modèles. Nous avons vérifié nos propriétés de sécurité sur des modèles abstraits, dont une transformation (compilation, synthèse...) assure une implémentation sur le *SoC* cible. Nous avons donc proposé, pour la préservation des propriétés de sécurité, l'instrumentation d'une infrastructure logicielle spécifique, basée sur des outils existants lors de l'écriture de ce manuscrit. Malheureusement, la préservation sémantique n'est pas prouvée et constitue une hypothèse pour certains de ces outils. Une perspective consiste donc à migrer vers des outils de transformation où celle-ci est vérifiée formellement, voir de les développer si de tels outils n'existent pas. À ce sujet, des détails techniques sont fournis en annexe E.

Une autre perspective importante concerne la définition approfondie des propriétés de sécurité nécessaires pour l'algorithme à attester. En effet, à la manière de précédents travaux à l'état de l'art, nous avons choisi, pour la fonction d'attestation, un calcul de tag d'intégrité depuis le prouveur lui-même. Ceci nous a permis d'éluder la problématique de la définition du test d'intégrité, malgré l'augmentation de la surface d'attaque, car toutes les mémoires accessibles par le prouveur peuvent y être incluses. Nous n'avons donc pas défini les zones mémoires qui devaient être incluses dans le calcul du tag d'intégrité, hormis l'algorithme à vérifier. En ce sens, de futurs travaux consistent à définir un test d'intégrité approprié pour les algorithmes qui s'exécutent dans un environnement complexe. Cette définition doit tenir compte de l'application et des propriétés de sécurité suffisantes pour le vérifieur. Certains registres du microprocesseur et de ses périphériques doivent potentiellement être inclus dans les calculs d'empreinte. Également, selon l'application, un calcul de tag d'intégrité peut ne pas être suffisant.

Concernant l'absence d'accès aux sources du microprocesseur, nous avons modélisé statiquement et précisément son comportement lors de l'exécution d'un logiciel légitime. Notre racine

de confiance statique s'appuie sur cette modélisation : les propriétés de sécurité sont garanties s'il est impossible pour un adversaire de reproduire ce comportement depuis un programme malveillant. Pour s'affranchir de cette hypothèse, une perspective serait de remplacer l'exécution de la fonction d'attestation sur le microprocesseur par l'utilisation d'une implémentation matérielle, à la manière d'un cryptoprocesseur. Une autre perspective envisageable est de considérer les architectures ouvertes, comme par exemple celles de type RISC-V, et proposer une modification pour obtenir des observables suffisants tout en appuyant la vérification sur des modèles disponibles pour le système.

Dans le cas d'un cryptoprocesseur étroitement lié à un microprocesseur pris sur étagère, le test d'intégrité ne sera plus dépendant de l'état du microprocesseur et de ses périphériques. En contrepartie, de nouvelles propriétés de sécurité seront à vérifier, comme la correction ou l'indépendance au secret. En effet, dans nos travaux, nous nous sommes appuyés sur une bibliothèque cryptographique logicielle où celles-ci ont été vérifiées formellement en amont. Également, les accès physiques aux zones mémoires nécessaires au calcul devront être rendus possibles lors de la phase de conception du système. Proposer une implémentation ne sera donc possible qu'après définition du test d'intégrité approprié pour l'application. Si certains registres de configuration des périphériques doivent être inclus, il devient donc aussi nécessaire de définir un moyen sécurisé pour y accéder.

Considérer les architectures ouvertes permettra d'accompagner la vérification d'un modèle et de garantir formellement l'absence de vulnérabilité sans émettre d'hypothèse sur leur comportement. Des modifications matérielles pourront permettre d'obtenir des observables suffisants à l'établissement d'une racine de confiance statique. De plus, cela permettra le calcul du test d'intégrité à l'aide de primitives cryptographiques logicielles vérifiées formellement. Cependant, contrairement aux microprocesseurs pris sur étagère, la problématique tiendra de l'ordre du coût de conception, du maintien des performances et de la reprogrammabilité. Une perspective envisageable est la modification matérielle d'un ASIC : cette solution permettra d'accéder à des performances élevées, même durant l'exécution de la fonction d'attestation. En contrepartie, celle-ci impliquera de satisfaire des lourdes contraintes de placement/routage pour permettre une fréquence de fonctionnement élevée. Une autre perspective envisageable est d'implémenter le système modifié dans un FPGA, à la manière des *softcores*. Dans ce cas, la maintenance du système implique simplement une reprogrammation mais la fréquence de fonctionnement sera limitée par les capacités du FPGA.

Afin de tirer profit de tous les avantages proposés par ces solutions, une dernière perspective consistera à les combiner : proposer la modification d'un ASIC de système ouvert, où un cryptoprocesseur dédié au calcul du tag d'intégrité sera implémenté dans un FPGA. La modification de l'ASIC permettra seulement de rendre possible les accès physiques aux registres et mémoires dont la valeur sera incluse au calcul du test d'intégrité : minimiser les modifications permettra de réduire les coûts de conception. Un cryptoprocesseur existant pourra être implémenté sous forme d'un *softcore* : les performances seront donc réduites, mais seulement pour le calcul du test d'intégrité. Ici aussi, il conviendra de définir le test d'intégrité en premier lieu, puis de proposer une architecture adaptée aux systèmes modernes en conséquence.

Quelque soit la solution envisagée, nos définitions de la sécurité de l'attestation à distance, nos modèles de microprocesseur et de ses périphériques, ainsi que nos preuves formelles, pourront être réutilisés. Le modèle du périphérique de confiance ou du cryptoprocesseur chargé du calcul du test d'intégrité, en revanche, sera à adapter en fonction du cas d'étude. Pour sa vé-

rification formelle, notre méthode de vérification formelle automatisée pourra être de nouveau déployée. À notre connaissance, il n'existe pas, aujourd'hui, d'implémentation qui permette d'établir une racine de confiance statique pour l'attestation à distance sur microprocesseur. Néanmoins, nous espérons que nos travaux ouvriront la voie à la construction de nouvelles architectures.

Annexe A

Ressources complémentaires : détails techniques

Sommaire

Ce manuscrit est accompagné de ressources complémentaires, modèles et outils, publiés sous licence libre et rendus accessibles publiquement [1]. Ces ressources permettent au lecteur une implémentation du système, une étude des modèles que nous proposons et un rejeu des vérifications formelles.

Dans les différents chapitres, nous proposons des expériences illustrant nos travaux. Dans cette annexe, nous fournissons des détails techniques destinés au lecteur souhaitant reproduire ces expériences. La section A.1 détaille l'environnement de développement pour programmer le *SoC* Zynq-7000. La section A.2 fournit des rappels techniques et exemples sur la vérification formelle, prérequis pour apprécier les preuves de sécurité de nos modèles.

A.1 *Systems On Chip* modernes : environnement de développement

Le chapitre 2 fournit des rappels techniques sur le fonctionnement du *SoC* Zynq-7000. Dans cette section, nous détaillons deux méthodes différentes d'implémentation de notre système sur celui-ci. La première méthode consiste à créer une image disque automatiquement chargée au démarrage du *SoC*. Cette solution devra être adoptée lors de la mise en production. La seconde méthode consiste à utiliser l'interface de test JTAG [90] et de contrôler la programmation et l'exécution depuis une connexion USB. Cette solution propose un environnement de travail adapté au développement. Nous appliquons ces méthodes à la programmation du FPGA et l'exécution de programmes sur la carte de développement Digilent Cora-Z7.

A.1.1 Image disque

Lors de la mise sous tension du *SoC*, une image, un FSBL, un *bitstream* et une ou plusieurs applications doivent être chargés en mémoire pour permettre le démarrage. L'ensemble de ces fichiers forme une image disque qui, si elle possède le format adapté, place le *SoC* dans un état sûr. Sur la carte de développement Cora-Z7, cette image disque peut être stockée dans une mémoire accessible depuis le *SoC*, soit une mémoire flash placée physiquement à coté de celui-ci, soit une carte SD dont le lecteur est physiquement accessible, etc.

Le constructeur fournit un programme, nommé `bootgen`, qui permet la génération d'images disques depuis un *bitstream* et plusieurs applications compilées et utilisant le format *Executable and Linkable Format*. Un fichier de configuration permet d'informer `bootgen` des fichiers à charger dans l'image et définit l'exécutable qui contient le FSBL. Dans cette thèse, nous utilisons `bootgen` pour construire nos images disque.

Ressources : le dossier `a_1_1_sdcard/` contient un exemple de fichier de configuration permettant la création d'une image disque avec `bootgen`. Les fichiers chargés dans l'image sont respectivement le *bitstream* généré à partir des ressources de la section 2.1.2, le FSBL des ressources de la section 2.2.1 et l'application des ressources de la section 2.2.2.
Le `Makefile` permet d'automatiser l'appel à `bootgen`. Le fichier généré peut ensuite être copié sur une carte SD à insérer dans la carte de développement Digilent Cora-Z7. Si le cavalier est positionné pour un démarrage sur la carte SD, alors les périphériques sont initialisés, le FPGA est programmé et l'application est chargé puis exécutée.

Notons qu'il est également possible de s'affranchir de l'outil `bootgen` dans le cas où nous utilisons le *boot loader* universel *Das U-Boot*. *Das U-Boot* est un logiciel libre remplaçant le FSBL et compatible avec le Zynq-7000. Nous ne l'utilisons pas dans cette thèse mais il représente une alternative libre pour la programmation du *SoC* lors de la mise en production.

A.1.2 OpenOCD

La seconde solution pour mettre en place un environnement fonctionnel sur le Zynq-7000 est d'utiliser l'interface JTAG. Dans ce cas, nous avons recours à un logiciel libre de développement, nommé OpenOCD [91], qui communique à l'aide d'USB avec un composant dédié à la programmation du *SoC* et présent sur la carte de développement (FT2232HQ). OpenOCD implémente les protocoles de communication avec les composants utilisés pour le débug des systèmes embarqués. Il propose une interface de programmation pour automatiser les tâches de débug (lecture mémoire, écriture dans les registres du microprocesseurs, création de points d'arrêt, etc.) et supporte la communication avec les microprocesseurs d'architecture ARMv7 Cortex-A9 [91]. Un fichier de configuration est nécessaire pour initialiser la communication avec chaque *SoC*. OpenOCD est livré avec un script d'initialisation du Zynq-7000 comportant un processeur ARMv7 avec deux cœurs. Nous fournissons une version modifiée de ce script où un seul cœur est initialisé [1].

OpenOCD possède également une interface de communication avec le débogueur de la suite GNU : `gdb` [91]. Lors de cette communication, l'exécution des commandes par l'outil `gdb` (*break*, *jump*, etc.) est traduite en un appel de fonctions de l'interface de programmation d'OpenOCD. Il devient donc possible de réaliser des sessions de débug depuis une machine connectée en USB, directement sur le microprocesseur ARMv7 implémenté dans le *SoC*. C'est

la solution que nous avons choisie pour implémenter les ressources fournies avec ce manuscrit dans un environnement de développement. Cette solution n'est pas adaptée à la mise en production car elle sous-entend que la machine utilisée pour la configuration du SoC est une machine de confiance.

Lors de la phase de développement, afin de placer le *SoC* dans l'état souhaité, nous utilisons un script `gdb`. Ce script contient une succession de commandes pour communiquer avec OpenOCD. Le début de chaque script suit en général les mêmes étapes :

1. le microprocesseur est placé dans un état de *reset*,
2. le FPGA est programmé avec un fichier *bitstream*,
3. le FSBL est chargé dans la DDR,
4. un point d'arrêt est ajouté à l'instruction de fin du FSBL,
5. un saut dans l'exécution est provoqué vers le point d'entrée du FSBL,
6. l'application est chargée dans la DDR,
7. des points d'arrêt sont ajoutés en fonction des tests à réaliser,
8. un saut dans l'exécution est provoqué vers le point d'entrée de l'application.

Pour chaque ressource nécessitant un script `gdb`, celui-ci est rendu disponible publiquement et est distribué sous licence libre [1].

Ressources : le dossier `a_1_2_openocd/` contient le script d'initialisation de la connexion entre OpenOCD et la carte de développement Digilent Cora-Z7. Également, un script `gdb` permet de démarrer une session de débug une fois la connexion initalisée. Les fichiers chargés via `gdb` sont respectivement le *bitstream* généré à partir des ressources de la section 2.1.2, le FSBL des ressources de la section 2.2.1 et l'application des ressources de la section 2.2.2.
Le `Makefile` permet d'automatiser ces tâches. La commande `make` appelle OpenOCD pour initialiser la connexion. La commande `make start` appelle `gdb` pour démarrer la session de débug. Les deux commandes doivent être exécutées dans deux terminaux différents.

A.2 Vérification formelle : rappels techniques et exemples

Le chapitre 3 introduit la stratégie de vérification formelle que nous proposons pour établir notre racine de confiance statique. Également, il introduit les outils, disponibles à l'état de l'art, sur lesquels nous nous appuyons pour appliquer cette stratégie de vérification. Dans cette sections, nous apportons des détails techniques sur la vérification formelle, plus particulièrement sur la théorie des automates et les méthodes de vérification.

A.2.1 Automates finis déterministes

Un automate fini est une construction mathématique abstraite, susceptible d'être dans un nombre fini d'états, mais étant à chaque instant dans un seul état à la fois. Le changement d'état d'un automate est contraint par une relation de transition. La notion d'instant est donc présente dans la théorie des automates : l'automate peut changer d'état au cours d'une suite d'instants comme décrit par la relation de transition. L'état dans lequel se trouve un

automate à un instant donné est appelé *état courant*. Un changement successif d'état au cours de différents instants est appelé une *exécution* de l'automate. La représentation d'une exécution de l'automate sous forme d'une suite d'états parcourus est appelée la *trace d'exécution*.

A.2.1.1 Automates finis

Dans cette thèse, nous représentons les automates finis sous la forme d'un quintuplet $A = \langle \mathcal{Q}, \Sigma, \mathcal{R}, \mathcal{I}, \mathcal{F} \rangle$, où :

— $\mathcal{Q}$ est un ensemble fini d'états
— Σ est un alphabet
— $\mathcal{R} \subseteq \mathcal{Q} \times \Sigma \times \mathcal{Q}$ est une relation de transitions
— $\mathcal{I} \subseteq \mathcal{Q}$ est un ensemble d'états initiaux
— $\mathcal{F} \subseteq \mathcal{Q}$ est un ensemble d'états finaux

L'ensemble des **états** dans lequel l'automate peut se situer est défini par $\mathcal{Q}$. À chaque instant, l'état courant est un élément de $\mathcal{Q}$.

Les automates possèdent également le concept d'**alphabet**. Un alphabet Σ est un ensemble quelconque dont les éléments sont appelés *symboles* ou *lettres*. Une suite de lettres, c'est-à-dire un changement de lettre au cours des instants, représente un *mot*.

La **relation de transition** $\mathcal{R}$ d'un automate détermine son état de destination en fonction de son état courant et d'une lettre de l'alphabet.

Le premier état dans lequel se trouve un automate fini est un membre des **états initiaux** $\mathcal{I}$. $\mathcal{I}$ est un sous-ensemble de $\mathcal{Q}$, les états de l'automate.

Certains états de $\mathcal{Q}$ sont qualifiés d'**états finaux**, ou états d'acceptation. Lorsque l'automate se trouve dans un état final, le mot qui a permis les transitions successives d'un état initial à cet état final est dit *accepté* par l'automate. L'ensemble de tous les mots acceptés par l'automate constitue le *langage* reconnu par l'automate.

La figure A.1 donne un exemple d'automate fini. Dans cette exemple, une lettre est un prédicat exprimé en fonction des variables a et b, représentant des booléens pouvant respectivement prendre les valeurs 0 ou 1.

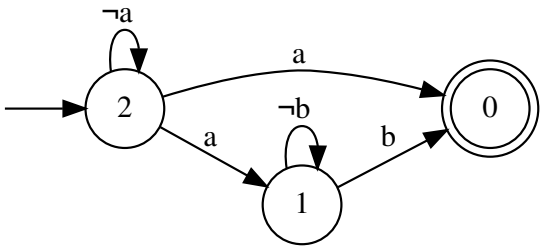

FIGURE A.1 – Exemple d'un automate fini non-déterministe

Cet automate possède trois états (0, 1, 2), dont un état initial (2) et un état final (0). Son alphabet est composé de quatre lettres ($(a \wedge b)$, $(\neg a \wedge b)$, $(a \wedge \neg b)$, $(\neg a \wedge \neg b)$), soit l'ensemble des n-uplets pour toutes valeurs des variables a et b. L'automate est non-déterministe, car sa relation de transition donne deux états de destination différents (0 et 1) depuis l'état 2 avec les lettres $(a \wedge b)$ et $(a \wedge \neg b)$.

Les mots suivants peuvent être acceptés par l'automate non-déterministe, selon l'exécution :

— une suite finie de lettres où a vaut 0, suivie d'une lettre où a vaut 1 (cas de la transition de l'état 2 vers l'état 0)
— une suite finie de lettres où a vaut 0, suivie d'une lettre où a vaut 1, puis une suite finie de lettres où b vaut 0, puis une lettre où b vaut 1 (cas de la transition de l'état 2 vers l'état 1, puis vers l'état 0)

A.2.1.2 Automates finis déterministes

Un automate fini est dit déterministe si et seulement si il ne possède qu'un seul **état initial** d'une part et, toutes les transitions à partir de chaque état sont déterminées de manière unique par une **fonction de transition** d'autre part. La relation $\mathcal{R}$ devient donc fonctionnelle dans le sens où :

$\forall p, q, q' \in \mathcal{Q}$ et $\forall s \in \Sigma$, si $(p, s, q) \in \mathcal{R}$ et $(p, s, q') \in \mathcal{R}$, alors $q = q'$.

Nous pouvons donc définir la fonction de transition Δ telle que :

$\forall (p, s, q) \in \mathcal{R}$, $\Delta(p, s) = q$.

Ainsi, un automate fini déterministe ne possède pas de relation de transition $\mathcal{R}$ et d'ensemble d'états initiaux $\mathcal{I}$: il possède une fonction de transition Δ et un unique état initial q_0. Il est donc défini sous la forme d'un quintuplet $A = \langle \mathcal{Q}, \Sigma, \Delta, q_0, \mathcal{F} \rangle$, où :

— $\mathcal{Q}$ est un ensemble fini d'états
— Σ est un alphabet
— $\Delta : \mathcal{Q} \times \Sigma \to \mathcal{Q}$ est une fonction de transition
— $q_0 \in \mathcal{Q}$ est un état initial
— $\mathcal{F} \subseteq \mathcal{Q}$ est un ensemble d'états finaux

La figure A.2 donne un exemple d'automate fini déterministe. Comme sur l'exemple donné sur la figure A.1, une lettre est un prédicat exprimé en fonction des variables a et b.

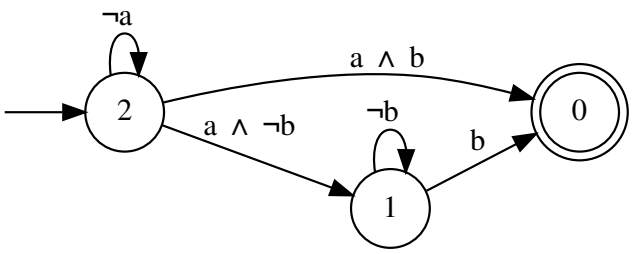

FIGURE A.2 – Exemple d'un automate fini déterministe

Cet automate possède les mêmes états et le même alphabet que celui représenté sur la figure A.1. Il est déterministe, car sa fonction de transition ne donne qu'un état de destination possible pour chaque état avec chaque lettre de l'alphabet. Les mots suivants sont acceptés par l'automate déterminisme :

— une suite finie de lettres où a vaut 0, suivie d'une lettre où a vaut 1 et b vaut 1 (cas de la transition de l'état 2 vers l'état 0)
— une suite finie de lettres où a vaut 0, suivie d'une lettre où a vaut 1 et b vaut 0, puis une suite finie de lettres où b vaut 0, puis une lettre où b vaut 1 (cas de la transition de l'état 2 vers l'état 1, puis vers l'état 0)

A.2.1.3 Automates de Büchi

Les automates de Büchi sont des automates finis particuliers : ils opèrent sur des mots infinis. Une trace d'exécution sur un automate de Büchi est donc elle aussi infinie. Ils peuvent être déterministes ou non-déterministes. La représentation mathématique des automates de Büchi est identique à celle des automates finis décrite précédemment. La différence avec ceux-ci est la condition d'acceptation des mots du langage. En effet, comme les mots sont infinis, un mot est accepté par l'automate si et seulement si la trace d'exécution passe infiniment souvent par un état final. Le langage d'un automate de Büchi est alors appelé un langage *rationnel*. Une expression définissant ce langage est appelée une expression *rationnelle* ou *ω-régulière*.

La figure A.3 donne un exemple d'automate de Büchi déterministe. La signification des variables a et b est la même que pour les automates représentés sur les figures A.1 et A.2.

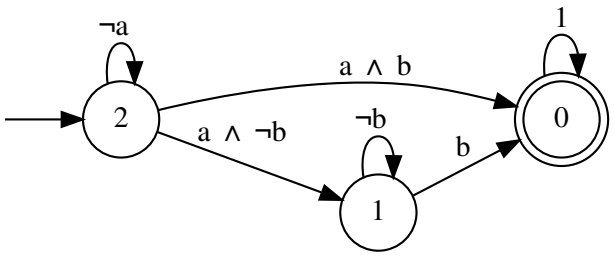

FIGURE A.3 – Exemple d'un automate de Büchi déterministe

Cet automate possède les mêmes états et le même alphabet que ceux représentés sur les figures A.1 et A.2. Il est déterministe, car sa fonction de transition ne donne qu'un état de destination possible pour chaque état avec chaque lettre de l'alphabet.

Cet automate est presque identique à l'automate fini déterministe représenté sur la figure A.2. La différence est qu'une exécution ne s'arrête pas lorsqu'un état final est atteint : la fonction de transition contient également une transition de l'état 0 vers lui-même. Comme toutes les lettres sont acceptées pour cette transition, les mots acceptés par l'automate de Büchi déterminisme sont donc tous les mots infinis commençant par les mots finis acceptés par l'automate représenté sur la figure A.2.

Ressources : la bibliothèque Spot fournit des fonctions pour manipuler les automates de Büchi déterministes.
Le script Python `a_2_1_automate_de_Buchi/buchi.py` utilise Spot, il permet de créer l'automate représenté sur la figure A.3 et de l'exporter au format *dot*. Le `Makefile` permet d'automatiser son exécution et une conversion au format *pdf*.

A.2.2 Logiques temporelles

Les propriétés exprimées en logique temporelle permettent la vérification d'automates. Pour toute propriété LTL ou PSL ϕ, on peut associer une expression rationnelle décrivant la forme que doit respecter une exécution si elle veut satisfaire ϕ [29]. Également, un langage est définissable par une expression rationnelle si et seulement si c'est un langage reconnaissable par un automate fini [92]. Il existe donc un automate de Büchi non-déterministe B ayant un alphabet Σ, où la propriété ϕ est vérifiée. Or, pour tout automate non-déterministe B, il existe

un automate déterministe B_d qui possède le même langage [68]. En conséquence, pour toute propriété LTL ou PSL ϕ, nous pouvons construire un automate de Büchi déterministe B_d qui accepte le langage associé à ϕ. Les automates de Büchi sont donc utilisés pour modéliser à la fois des systèmes à états/transitions et des propriétés de logique temporelle [68].

La propriété décrite par l'expression A.1 donne une spécification formelle du langage d'un automate. Pour qu'un mot fasse partie de ce langage, dans un état futur, le prédicat a doit être vérifié à l'état courant et le prédicat b doit être vérifié dans un nouvel état futur. Notons que, formellement, l'état courant fait également partie des états futur : c'est l'état qui est atteint après zéro instant durant l'exécution d'un automate.

$$\phi = \mathbf{eventually}\,(a \wedge \mathbf{eventually}(b)) \tag{A.1}$$

Plus exactement, pour qu'un mot fasse partie du langage décrit par la propriété ϕ de l'expression A.1, les caractéristiques suivantes doivent être vérifiées :

- le prédicat a peut ne pas être vérifié pour un nombre fini de lettres
- le prédicat a doit être vérifié pour une lettre ; pour cette lettre, le prédicat b peut être vérifié :
  - si b est vérifié en même temps que a, alors le mot fait partie du langage.
  - si b n'est pas vérifié en même temps que a, alors :
    - le prédicat b peut ne pas être vérifié pour un nombre fini de lettres,
    - le prédicat b doit être vérifié pour une lettre.

La figure A.4 représente un automate de Büchi déterministe qui accepte le langage décrit par la propriété ϕ de l'expression A.1. Pour qu'un état final de l'automate soit atteint infiniment souvent par une exécution depuis l'état inital, les mêmes caractéristiques sur le mot doivent être vérifiées.

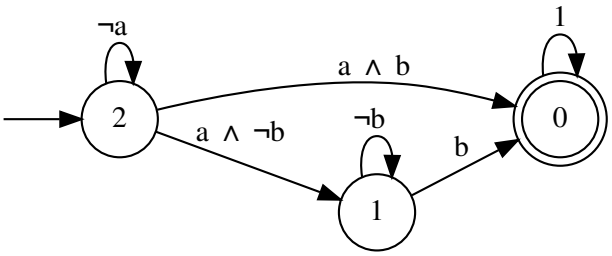

FIGURE A.4 – Automate de Büchi déterministe acceptant le langage décrit par ϕ

Cet automate possède les mêmes états, le même alphabet et la même condition d'acceptation que celui représenté sur la figure A.3. L'ensemble des mots acceptés par cet automate vérifient donc la propriété ϕ de l'expression A.1.

Ressources : la bibliothèque Spot fournit des fonctions pour convertir les propriétés LTL et PSL en automates de Büchi déterministes.
Le script Python `a_2_2_logique_temporelle/ltl.py` utilise Spot, il permet de convertir la propriété ϕ de l'expression A.1 vers l'automate représenté sur la figure A.4 et de l'exporter au format *dot*. Le `Makefile` permet d'automatiser son exécution et une conversion au format *pdf*.

A.2.3 Model-checking

Dans cette thèse, nous utilisons le model-checker *NuSMV* [74, 75] pour la vérification de nos propriétés locale. Pour la manipulation de propriétés temporelles, nous utilisons *Spot* [48], une bibliothèque C++ dont les fonctionnalités peuvent être appelées depuis un interpréteur Python. *Spot* peut également être utilisée pour réaliser indépendamment les étapes du *model-checking*. Dans cette section, nous rappelons les concepts du *model-checking* et proposons un exemple de vérification par *model-checking* avec *Spot* ou *NuSMV*.

A.2.3.1 Concepts du *model-checking*

Dans cette thèse, nous utilisons le *model-checking* dans le but de réaliser des vérifications sur des automates représentant des circuits électroniques numériques, c'est-à-dire des automates finis déterministes possédant un ensemble vide d'états finaux. La présence d'états finaux sur un automate est ici la conséquence d'avoir une propriété de logique temporelle à vérifier.

Pour un automate fini déterministe A donné, dont l'ensemble des états finaux est vide, et une propriété de logique temporelle ϕ donnée, la stratégie de vérification par un model-checker suit ces différentes étapes :

1. Comme vu précédemment, pour toute propriété ϕ, nous pouvons associer une expression rationnelle décrivant la forme que doit respecter une exécution pour satisfaire ϕ.

   Le model-checker peut donc construire un automate de Büchi non-déterministe B ayant pour alphabet Σ, où ϕ est vérifiée. Cet automate possède donc au moins un état final.

2. Or, pour tout automate de Büchi B ayant pour alphabet Σ et pour langage $L(B)$, il existe un automate complémentaire $\overline{B}$ tel que $L(\overline{B}) = \Sigma - L(B)$ [68].

   Le model-checker construit donc un automate complémentaire $\overline{B}$ acceptant exactement les exécutions qui ne satisfont pas ϕ, soit un automate qui satisfait $\neg\phi$ [29].

3. Comme vu précédemment, pour tout automate non-déterministe, il existe un automate déterministe qui possède le même langage.

   Un automate de Büchi déterministe $\overline{B_d}$ est donc construit à partir de $\overline{B}$.

4. Également, pour tout couple d'automates A_1 et A_2, il existe un automate $A_p = A_1 \times A_2$ tel que $L(A_p) = L(A_1) \cap L(A_2)$ [68]. Cet automate A_p est appelé le produit carthésien des automates A_1 et A_2.

   Pour vérifier si une propriété de logique temporelle est satisfaite par un automate A, le model-checker réalise donc le produit carthésien entre l'automate A et l'automate déterministe $\overline{B_d}$ [29].

5. Le problème de *model-checking* "est-ce que l'automate A satisfait la spécifiation ϕ ?" est donc ramené au problème "est-ce que le langage reconnu par $A \times \overline{B_d}$ est vide ?" [29].

   Trouver un mot reconnu par l'automate de Büchi déterministe $A \times \overline{B_d}$, c'est-à-dire trouver un chemin qui passe infiniment souvent par un état final, fournit un contre exemple qui viole la propriété ϕ. Si le langage de l'automate $A \times \overline{B_d}$ est vide, alors la propriété ϕ est vérifiée.

En théorie la seule limitation du *model-checking* est que le système à états/transitions à vérifier doit être fini, ce qui est toujours le cas dans cette thèse, car nous manipulons des modèles de circuit électroniques numériques. En pratique, cependant, la limitation majeure du *model-checking* est le *problème d'explosion combinatoire du nombre d'états du système*, où la mémoire nécessaire pour modéliser l'automate de manière exhaustive excède la mémoire physique de la machine utilisée pour le *model-checking*. Pour donner un ordre de grandeur [93] :

— un système manipulant 10 variables codées sur 8 bits est représenté par un automate de $2^{(10\times8)}$ états, soit environ 10^{24} états ; le nombre de secondes écoulées depuis le Big-Bang est de l'ordre de 10^{17}.

— un système manipulant 10 variables codées sur 32 bits est représenté par un automate de $2^{(10\times32)}$ états, soit environ 10^{96} états ; le nombre de particules dans l'univers est de l'ordre de 10^{80}.

Ce problème d'*explosion combinatoire* est donc causé, d'une part, par le nombre de variables décrivant les états du système. D'autre part, il est lié au nombre d'opérateurs temporels dans la propriété à vérifier : lors de la conversion de sa négation en automate de Büchi déterministe, le nombre d'états résultant impacte directement la taille du produit des automates.

De plus, la vérification de propriétés temporelles requiert un dépliage du produit des automates. Ce dépliage est la création d'un arbre dont la racine est l'état initial de l'automate produit et dont chaque nœud a pour successeurs les états de destination définis par la fonction de transition. La différence entre l'automate produit déplié et l'automate produit non-déplié est que l'on n'identifie plus les nœuds correspondants aux mêmes états : un même état peut apparaitre plusieurs fois dans l'arbre, car il peut être successeur de plusieurs nœuds. Néanmoins, les techniques de *model-checking symbolique* par *diagrammes de décision binaires* (BDD : de l'anglais *Binary Decision Diagrams*) [94] permettent de représenter de manière compacte les systèmes de transitions et réduire ainsi l'explosion combinatoire.

A.2.3.2 Vérification avec *Spot*

La figure A.5 représente un automate fini déterministe sur lequel nous pouvons vérifier des propriétés temporelles. Ses différentes transitions sont exprimées en fonction des valeurs de deux variables booléennes a et b. Il possède un ensemble vide d'états finaux.

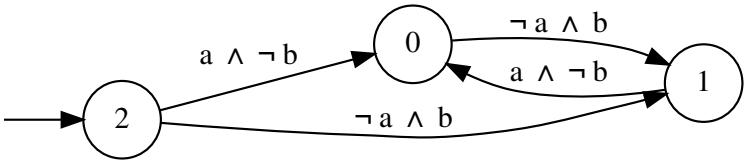

FIGURE A.5 – Automate fini déterministe

Cet automate possède un état initial 2, à partir duquel il peut se placer dans un état 0, où seule la variable a est vraie, ou se placer dans un état 1, où seule la variable b est vraie. Depuis chacun des états 0 et 1, l'automate oscille indéfiniment : les valeurs des variables a et b oscillent donc également entre vrai et faux. Sur cet automate, les outils de model-cheking nous permettent de vérifier des propriétés temporelles telles que la propriété ϕ de l'expression A.1 ou la propriété ψ de l'expression A.2.

$$\phi = \textbf{eventually}\,(a \wedge \textbf{eventually}(b))$$
$$\psi = \textbf{always}\,(a \wedge \textbf{eventually}(b)) \tag{A.2}$$

La figure A.6 représente deux automates de Büchi déterministes construits par le model-checker. Ceux-ci représentent respectivement les automates complémentaires aux constructions qui satisfont ϕ et ψ. Ce sont donc des automates qui vérifient respectivement $\neg\phi \equiv \textbf{always}\,(\neg a \vee \textbf{always}(\neg b))$ et $\neg\psi \equiv \textbf{eventually}\,(\neg a \vee \textbf{always}(\neg b))$.

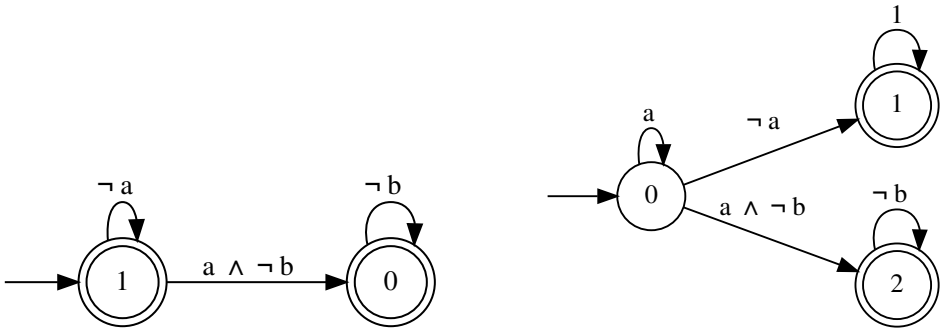

FIGURE A.6 – Automates complémentaires à ceux construits depuis les propriétés ϕ et ψ

À partir de l'automate de la figure A.5, le model-checker réalise deux produits, respectivement avec les deux automates de la figure A.6. Le résultat de ces deux produits est représenté sur la figure A.7. Le premier automate représente le résultat du produit avec l'automate vérifiant $\neg\phi$ tandis que le second représente celui du produit avec l'automate vérifiant $\neg\psi$.

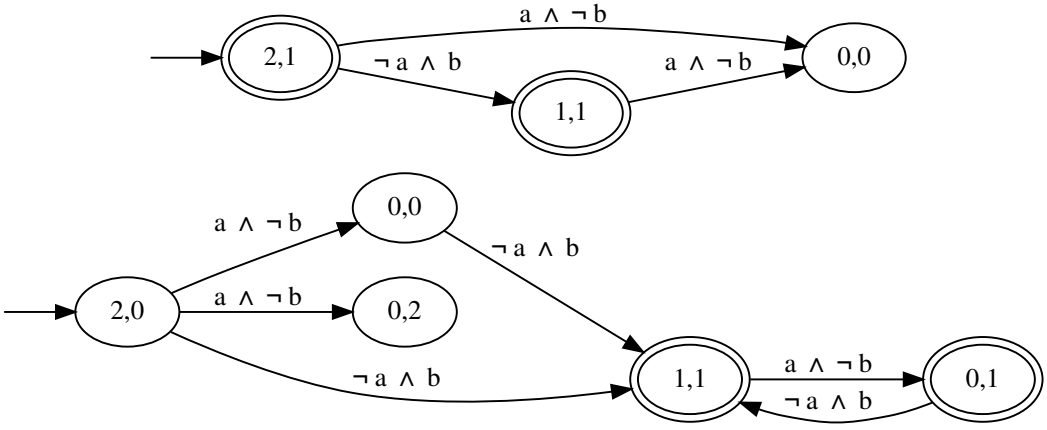

FIGURE A.7 – Produits entre l'automate à vérifier et les automates de $\neg\phi$ et $\neg\psi$

Sur le premier automate, il n'existe pas d'exécution qui passe infiniment souvent par un état final : une fois que l'état $(0, 0)$ est atteint, il n'est plus possible d'en sortir. Le langage de cet automate est donc vide et la propriété ϕ est vérifiée par l'automate.

Sur le second automate, il existe au moins une exécution qui passe infiniment souvent par un état final : une fois que l'état $(1, 1)$ est atteint, nous pouvons atteindre l'état $(0, 1)$ et inversement. Cette opération peut être répétée infiniment souvent et les exécutions qui la

réalisent passent infiniment par les états $(1, 1)$ et $(0, 1)$. Ces exécutions représentent des contre-exemples et démontrent que la propriété ψ n'est pas vérifiée par l'automate fini déterministe représenté sur la figure A.5.

Ressources : la bibliothèque Spot fournit des fonctions pour réaliser les compléments d'automates et les produits d'automates.
Le script Python `a_2_3_model_checking/print/mc.py` utilise Spot, il permet d'obtenir les automates produits représentés sur la figure A.7 et de les exporter au format *dot*. Le `Makefile` permet d'automatiser son exécution et une conversion au format *pdf*.

A.2.3.3 Vérification avec *NuSMV*

Le model-checker *NuSMV* automatise les différentes tâches de vérification de propriétés temporelles sur un automate. Il est simplement nécessaire de lui transmettre les définitions de l'automate et des propriétés temporelles, toutes deux exprimées en langage SMV. Après vérification, *NuSMV* atteste de la vérification d'une propriété ou produit un contre-exemple en cas de falsification.

Ressources : Le modèle SMV `a_2_3_model_checking/nusmv/main.smv` contient la définition de l'automate représenté sur la figure A.5 et des propriétés ϕ et ψ. Le `Makefile` permet d'automatiser l'exécution de *NuSMV* et d'observer un contre-exemple qui viole la propriété ψ.

A.2.4 Preuve de théorème

Dans cette thèse, nous instrumentons les fonctionnalités de la bibliothèque *Spot* pour également réaliser des preuves de théorème automatiques. Afin d'éviter de se heurter à l'explosion combinatoire lors de la preuve d'un théorème, nous utilisons l'assistant de preuves Coq. Les concepts de preuve de théorème sont décrits dans la section 3.3.2. Dans cette section, nous proposons un exemple de preuve de théorème, où une spécification est vérifiée dès lors que la conjonction de propriétés locales l'est aussi.

A.2.4.1 Preuve automatique

Spot peut également être utilisée pour réduire une implication de spécification à partir de propriétés locales. Dans ce cas, nous utilisons ses fonctionnalités de filtrage. Si cette réduction permet d'aboutir à une tautologie, alors nous obtenons une preuve automatique de théorème. Dans cette section, nous proposons un exemple de preuve de théorème automatique à partir de propriétés locales et d'une spécification.

Pour illustrer la méthode automatique de filtrage des propriétés, l'expression A.3 donne un exemple de deux propriétés locales ϕ_1 et ϕ_2 et de deux spécifications ϕ et ψ. Nous considérons que les propriétés locales ϕ_1 et ϕ_2 ont été vérifiées sur un automate par *model-checking* : elles représentent donc nos hypothèses. Nous souhaitons prouver que l'automate vérifie alors les spécifications ϕ et ψ.

$$\begin{aligned}\phi_1 &= \textbf{eventually}\,(a)\\ \phi_2 &= \textbf{always}\,(a \rightarrow \textbf{eventually}(b))\\ \phi &= \textbf{eventually}\,(a \wedge \textbf{eventually}(b))\\ \psi &= \textbf{always}\,(a \wedge \textbf{eventually}(b))\end{aligned} \tag{A.3}$$

Nous écrivons donc une formule composée de la conjonction des propriétés locales ϕ_1 et ϕ_2 et d'une implication de la spécification (ϕ ou ψ, selon le cas). Les algorithmes de filtrage fournis par Spot permettent une minimisation de l'automate résultant et une réécriture automatique de la formule. L'expression A.4 donne les résultats d'un tel filtrage.

$$\begin{aligned}&\phi_1 \wedge \phi_2 \rightarrow \phi \equiv 1\\ &\phi_1 \wedge \phi_2 \rightarrow \psi \equiv \textbf{always}(a \wedge \textbf{eventually}(b)) \vee \textbf{always}(\neg a) \vee \textbf{eventually}(a \wedge \textbf{always}(\neg b))\end{aligned} \tag{A.4}$$

Si nous ne procédons pas au *model-checking* des spécifications ϕ et ψ sur notre automate, nous pouvons néanmoins conclure que :

- la spécification ϕ est également vérifiée par l'automate : son implication par la conjonction des propriétés locales ϕ_1 et ϕ_2 est une tautologie.
- la vérification par *model-checking* des propriétés ϕ_1 et ϕ_2 ne semble pas suffisante pour garantir que la spécification ψ est elle aussi vérifiée : son implication n'est réduite qu'à une expression formée de prédicats et d'opérateurs temporels. Nous ne pouvons donc pas conclure que ψ est vérifiée.

Ressources : le script Python `a_2_4_preuve/spot/minimize.py` utilise Spot, il permet de reproduire l'expérience et obtenir les résultats décrits par l'expression A.4.
Dans ce même script, un exemple de propriété provoquant une explosion combinatoire est proposé : il illustre le cas où Spot ne trouve pas de solution et où nous ne pouvons pas conclure. Le `Makefile` permet d'automatiser son exécution.

A.2.4.2 Preuve manuelle

Si nous souhaitons éviter de se heurter à l'explosion combinatoire lors de la preuve d'un théorème, nous pouvons avoir recours à une preuve manuelle à l'aide d'un assistant de preuves. Dans cette thèse, nous avons choisi l'assistant de preuves Coq pour démontrer nos théorèmes.

Pour reprendre l'exemple de l'expression A.3, si nous considérons les propriétés ϕ_1 et ϕ_2 comme des hypothèses, il est possible de prouver la spécification ϕ. Nous utilisons pour cela un lemme qui démontre l'involution de la négation sur une propriété temporelle quelconque. En revanche, nous ne pouvons pas prouver la spécification ψ car nos hypothèses ne sont pas suffisantes.

Lorsque nous prouvons manuellement un théorème, la différence majeure avec une méthode automatique de filtrage des propriétés est que nous pouvons obtenir une preuve qui s'applique à des formules temporelles quelconques et non à de simple prédicats. Avec Spot, nous pouvons prouver la spécification ϕ à partir des propriétés ϕ_1 et ϕ_2 dans le cas où a et b sont de simples prédicats : nous ne pouvons donc pas réutiliser notre théorème dans le cas où ϕ_1 et ϕ_2 sont

exprimées en fonction de prédicats supplémentaires. Une nouvelle preuve est alors nécessaire et nous risquons à nouveau de nous heurter à une explosion combinatoire. Avec Coq, nous pouvons prouver la spécification ϕ à partir des propriétés ϕ_1 et ϕ_2 dans le cas où a et b sont des formules temporelles quelconques. Ainsi, nous pouvons réutiliser notre théorème et réduire l'effort de preuve dans le cas où plusieurs de nos propriétés locales et spécifications ont la même forme.

Ressources : le programme `a_2_4_preuve/coq/proof.v` utilise Coq, il permet de reproduire l'expérience et obtenir la preuve de ϕ depuis les propriétés ϕ_1 et ϕ_2 de l'expression A.3. Lors de cette preuve, les termes a et b font référence à des formules temporelles quelconques.
Dans ce même programme, deux lemmes sont également prouvés, nécessaires à l'obtention de la preuve de ϕ, dont l'involution de la négation sur une propriété temporelle quelconque.
Le `Makefile` permet d'automatiser sa compilation et son exécution.

Annexe B

Outils associés à la méthode de vérification formelle automatisée : détails techniques

Sommaire

Notre méthode de vérification formelle automatisée, détaillée dans le chapitre 4, est mise en œuvre sous forme de plusieurs outils indépendants. Ces outils réalisent entre autres des conversions de propriétés exprimées en PSL vers du LTL. Dans cette annexe, nous fournissons des détails techniques sur les équivalences entre PSL et LTL. Également, nous introduisons brièvement les fonctionnalités de chacun des outils. La combinaison de ceux-ci permet une réduction par localisation et une automatisation complète de la vérification formelle.

B.1 Équivalences entre PSL et LTL

Dans cette section, nous détaillons des équivalences entre PSL et LTL. Nous proposons des exemples de définitions pour les fonctions de traduction des opérateurs temporels étendus. Également, nous fournissons les recommandations à suivre pour tirer profit au maximum des abstractions implicites lors d'une traduction automatique de PSL vers LTL.

B.1.1 Opérateurs temporels étendus

Nous proposons ici des exemples de définitions pour les fonctions de traductions des opérateurs étendus **next_event** et **next_event_a**. Les algorithmes 2 et 3 fournissent de telles définitions.

L'algorithme 2 décrit comment *NuSMV* traduit l'opérateur temporel étendu **next_event**. Cet algorithme est défini pour toute propriété temporelle ψ et ϕ et tout entier naturel i supérieur ou égal à 1. Il contient une récursion dans le cas où i est strictement supérieur 1.

Algorithme 2 : **next_event**$(\psi)[i](\phi)$

Entrée:
ψ : propriété temporelle,
i : entier naturel ≥ 1,
ϕ : propriété temporelle
Sortie:
propriété temporelle
1: **si** (i = 1) **alors**
2: **retourne** $(\neg\psi$ **until** $(\psi \wedge \phi) \vee$ **always**$(\neg\psi))$
3: **sinon**
4: **retourne** $(\neg\psi$ **until** $(\psi \wedge$ **next** $($**next_event**$(\psi)[i-1](\phi))) \vee$ **always**$(\neg\psi))$
5: **fin si**

— Dans le cas où i est égal à 1, l'algorithme retourne une propriété temporelle exprimée en fonction de ψ et ϕ et des opérateurs temporels de base **always** et **until**.
— Dans le cas où i est strictement supérieur 1, l'algorithme contient une récursion et retourne une propriété temporelle exprimée en fonction de ψ et des opérateurs temporels de base **next**, **always** et **until**.

Algorithme 3 : **next_event_a**$(\psi)[i:j](\phi)$

Entrée:
ψ : propriété temporelle,
i : entier naturel ≥ 1,
j : entier naturel $\geq i$,
ϕ : propriété temporelle
Sortie:
propriété temporelle
1: **si** (j = i) **alors**
2: **retourne** $($**next_event**$(\psi)[i](\phi))$
3: **sinon**
4: **retourne** $($**next_event_a**$(\psi)[i:(j-1)](\phi) \wedge$ **next_event**$(\psi)[j](\phi))$
5: **fin si**

L'algorithme 3 décrit comment *NuSMV* traduit l'opérateur temporel étendu **next_event_a**. Cet algorithme est défini pour toute propriété temporelle ψ et ϕ et tout entier naturel i et j où i est supérieur ou égal à 1 et j est supérieur où égal à i. Il fait appel à l'algorithme de traduction de l'opérateur temporel étendu **next_event** décrit par l'algorithme 2. Il contient une récursion dans le cas où j est strictement supérieur i.

— Dans le cas où j est égal à i, l'algorithme retourne une propriété temporelle résultante d'un appel à **next_event**$(\psi)[i](\phi)$.
— Dans le cas où j est strictement supérieur i, l'algorithme contient une récursion et retourne une conjonction entre un appel à lui-même et un appel à **next_event**$(\psi)[j](\phi)$.

Dans tous les cas, l'opérateur temporel étendu **next_event** est exprimé en fonction des opérateurs temporels de base **next**, **always** et **until** comme décrit par l'algorithme 2.

Ressources : le programme `b_1_x_strategie_preuve/coq_next_event/compute.v` utilise Coq, il propose une implémentation des algorithmes 2 et 3. Son exécution affiche le résultat d'un calcul pour certaines valeurs des entiers naturels i et j, où ψ et ϕ sont des prédicats quelconques. Le `Makefile` permet d'automatiser sa compilation et son exécution.

B.1.2 Abstractions de fonctions non-interprétées

Une fois que nous sommes en possession de propriétés temporelles reconnaissables par *Spot*, nous construisons un script *Python* qui permet un filtrage. Le filtrage est réalisé sur une propriété construite depuis la conjonction des propriétés locales et une implication des spécifications. *Spot* est instrumenté pour produire un automate de Büchi déterministe qui reconnait le langage décrit par cette propriété et pour appliquer ses algorithmes de minimisation.

Cette méthode est entièrement automatique, mais elle requiert un dépliage de l'automate résultant et se heurte à l'explosion combinatoire. Afin de réduire l'impact du dépliage, nous fournissons des abstractions de fonctions non interprétées lors de la construction de la propriété à filtrer. Ces abstractions sont axiomatiques et implicites, elles sont donc à considérer par le concepteur lors de l'expression des propriétés locales.

Dans cette section, nous proposons des exemples d'abstractions de fonctions non-interprétées et listons les recommandations à suivre pour prouver des spécifications complexes.

Dans le chapitre 4, l'équivalence entre les expressions 4.1 et 4.2 donne un exemple d'axiome implicite. Dans cet exemple, la propriété représentée par l'expression 4.1 est vérifiée par *model-checking* ; la propriété représentée par l'expression 4.2 est celle qui est considérée par *Spot* lors du filtrage, durant l'exécution du script *Python* que nous construisons.

Dans cet exemple, la conjonction de 256 propriétés est abstraite en une seule propriété où le prédicat $in_1\_equals\_i$ implique toujours que le prédicat out_equals_i sera éventuellement vérifié. D'un côté, cette abstraction réduit la complexité de la minimisation de manière drastique : si les prédicats $in_1\_equals\_i$ et out_equals_i apparaissent des deux côtés de l'implication, nous pouvons prouver un théorème valable quelque soit les tailles des vecteurs in_1 et out. D'un autre côté, il devient de la responsabilité du concepteur de s'assurer que l'intervalle $\{0 : 255\}$ de l'expression 4.1 est suffisante pour couvrir toutes les valeurs possibles pour les vecteurs in_1 et out. Également, il devient nécessaire de bien choisir les noms des indices des quantificateurs : le nom des indices apparait dans le nom du prédicat créé par notre script *Python* ; un prédicat différent doit être construit pour chaque quantificateur si les propriétés paramétrées diffèrent. En effet, si deux propriétés locales différentes contiennent des quantificateurs et que, dans les deux cas, l'indice est nommé i, alors *Spot* considère que nous traitons la même propriété paramétrée contenant une conjonction. Il y a donc deux cas possibles lorsque nous traitons une conjonction d'au moins deux propriétés locales contenant des quantificateurs :

1. soit la conjonction des quantificateurs est équivalente à un quantificateur de la conjonction des propriétés paramétrées. Dans ce cas, nous avons une équivalence des indices et le concepteur peut utiliser les mêmes noms pour les différents indices.
2. soit la conjonction des quantificateurs n'est pas équivalente à un quantificateur de la conjonction des propriétés paramétrées. Dans ce cas, les indices sont différents et le concepteur ne peut pas utiliser les mêmes noms. Il lui est cependant possible d'ajouter

des propriétés axiomatiques afin de définir une relation entre les différents indices des quantificateurs.

Pour illustrer le premier cas de figure, imaginons que nous considérons les propriétés décrites par l'expression B.1. Les propriétés ϕ_1 et ϕ_2 représentent les propriétés locales que nous vérifions sur le système par *model-checking* et la propriété ϕ représente la spécification que nous souhaitons prouver. Nous traitons le cas d'un système qui possède un vecteur d'entrée in_1 encodé sur 8 bits et un vecteur de sortie *out* encodé lui aussi sur 8 bits.

$$\begin{aligned}
\phi_1 &\equiv \textbf{forall } i \textbf{ in } \{0:255\} : \textbf{eventually}(in_1 = i) \\
\phi_2 &\equiv \textbf{forall } i \textbf{ in } \{0:255\} : \textbf{always}[(in_1 = i) \rightarrow \textbf{eventually}(out = i)] \\
\phi &\equiv \textbf{forall } i \textbf{ in } \{0:255\} : \textbf{eventually}[(in_1 = i) \wedge \textbf{eventually}(out = i)]
\end{aligned} \tag{B.1}$$

Dans le cas des propriétés locales ϕ_1 et ϕ_2, l'égalité $(in_1 = i)$ qui apparait dans les deux propriétés peut être intervertie : il y a une équivalence entre ces deux égalités dans les deux propriétés locales. Nous pouvons donc écrire la conjonction des propriétés locales ϕ_1 et ϕ_2 comme une propriété contenant un quantificateur de la conjonction des propriétés paramétrées. L'expression B.2 donne une représentation de cette propriété.

$$\begin{aligned}
\phi_1 \wedge \phi_2 \equiv \textbf{forall } i \textbf{ in } \{0:255\} : \\
(\textbf{eventually}(in_1 = i) \wedge \textbf{always}[(in_1 = i) \rightarrow \textbf{eventually}(out = i)])
\end{aligned} \tag{B.2}$$

Nous sommes donc face au cas où nous devons utiliser le même indice i pour les différents quantificateurs. La seule restriction est qu'il est nécessaire de s'assurer que l'intervalle $\{0:255\}$ couvre bien toutes les valeurs possibles pour les vecteurs in_1 et *out* lors du *model-checking*. Si les vecteurs in_1 et *out* sont encodés sur 8 bits, alors la preuve que nous réalisons avec *Spot* est correcte.

Pour illustrer le second cas de figure, imaginons que nous considérons les propriétés décrites par l'expression B.3. Ici aussi, les propriétés ϕ_1 et ϕ_2 représentent les propriétés locales et la propriété ϕ représente la spécification que nous souhaitons prouver. Nous traitons le cas d'un système qui possède deux vecteurs d'entrée in_1 et in_2 encodés sur 8 bits et un vecteur de sortie *out* encodé lui aussi sur 8 bits. Les données reçues sur les deux vecteurs d'entrée sont restituées sur le vecteur de sortie à des instants différents, dépendants d'une demande de lecture indiquée par le bit *read*.

$$
\begin{aligned}
\phi_1 \equiv\ & \textbf{forall}\ \ i_1\ \ \textbf{in}\ \ \{0:255\}:\ \textbf{always}\{(in_1 = i_1) \rightarrow \textbf{next\_event}(read)[1](out = i_1)\} \\
\phi_2 \equiv\ & \textbf{forall}\ \ i_2\ \ \textbf{in}\ \ \{0:255\}:\ \textbf{always}\{(in_2 = i_2) \rightarrow \textbf{next\_event}(read)[2](out = i_2)\} \\
\phi \equiv\ & \textbf{forall}\ \ i\ \ \textbf{in}\ \ \{0:65535\}:\ \textbf{always}\{ \\
& \quad (in_1 = (i >> 0)) \\
& \quad \wedge (in_2 = (i >> 8)) \\
& \rightarrow \\
& \quad \textbf{next\_event}(read)[1](out = (i >> 0)) \\
& \quad \wedge \textbf{next\_event}(read)[2](out = (i >> 8)) \\
& \}
\end{aligned}
\tag{B.3}
$$

Dans le cas de la propriété ϕ_1, les 8 bits lus sur le premier vecteur d'entrée seront restitués sur le vecteur de sortie lors du premier évènement où le prédicat *read* est vérifié. Dans le cas de la propriété ϕ_2, les 8 bits lus sur le second vecteur d'entrée seront restitués sur le vecteur de sortie lors du second évènement où le prédicat *read* est vérifié. Nous souhaitons prouver que, comme exprimé par la propriété ϕ, la concaténation des deux vecteurs d'entrée est restituée sur le vecteur de sortie lors des demandes de lecture et ce, pour toute valeur possible des 16 bits d'entrée. Ici, nous ne pouvons pas écrire la conjonction des propriétés locales ϕ_1 et ϕ_2 comme une propriété contenant un quantificateur de la conjonction des propriétés paramétrées. En effet, les deux indices i_1 et i_2 ne font pas toujours référence à la même valeur.

Nous sommes donc face au cas où nous ne pouvons pas utiliser le même indice i pour les différents quantificateurs. Avec notre abstraction des quantificateurs **forall**, les propriétés locales ϕ_1 et ϕ_2 fournissent des prédicats créés à partir des deux indices i_1 et i_2. Le concepteur n'a donc pas d'autre choix que d'utiliser un troisième indice pour faire référence à toutes les valeurs des 16 bits. En conséquence, avec seulement les trois propriétés décrites par l'expression B.3, *Spot* ne nous permet pas de prouver la spécification. Pour prouver cette spécification, il est donc nécessaire de renseigner *Spot* sur les relations que nous définissons entre les différents indices. Le concepteur doit donc ajouter des propriétés axiomatiques, c'est-à-dire des propriétés qui sont ajoutées à la conjonction des propriétés locales, mais qui ne sont pas vérifiées par *model-checking*. Celles-ci définissent une relation entre les prédicats en fonction des différents indices des quantificateurs. Les axiomes décrits par l'expression B.4 permettent de prouver la spécification ϕ de l'expression B.3.

$$
\begin{aligned}
& \textbf{always}\{(in_1 = i_1) \leftrightarrow (in_1 = (i >> 0))\} \\
& \textbf{always}\{(in_2 = i_2) \leftrightarrow (in_2 = (i >> 8))\} \\
& \textbf{always}\{(out = i_1) \leftrightarrow (out = (i >> 0))\} \\
& \textbf{always}\{(out = i_2) \leftrightarrow (out = (i >> 8))\}
\end{aligned}
\tag{B.4}
$$

Ces axiomes expriment, pour chacun des indices i_1 et i_2, une équivalence entre des égalités de vecteurs exprimées ou non en fonction de l'indice. Cette équivalence est valide pour les vecteurs d'entrée et le vecteur de sortie. Comme notre outil abstrait les opérations logiques entre vecteurs par des prédicats, définir une relation entre ces prédicats implique définir une

relation entre les indices i, i_1 et i_2.

Notons également, qu'en plus de définir des axiomes, le concepteur doit ici aussi s'assurer, lors du *model-checking*, que l'intervalle $\{0 : 255\}$ couvre bien toutes les valeurs possibles pour les vecteurs in_1, in_2 et out. L'intervalle $\{0 : 65535\}$ n'a pas d'impact sur la preuve de théorème automatique, mais elle doit être choisie de manière cohérente : la spécification ϕ constitue un résultat qui peut être utilisé comme hypothèse pour une seconde preuve.

Ressources : les dossiers `b_1_x_strategie_preuve/psl2spot_same_i/` et `b_1_x_strategie_preuve/psl2spot_different_i/` contiennent des définitions formelles de propriétés locales et de spécifications. Le programme *psl2spot* permet d'automatiser la construction d'un script pour la preuve de théorème automatique. Dans chaque dossier, un `Makefile` permet d'automatiser la construction d'un script et son exécution.
Le dossier `b_1_x_strategie_preuve/psl2spot_same_i/` contient la preuve de théorème illustrée par l'expression B.1 : c'est-à-dire de la spécification ϕ à partir de la conjonction des propriétés locales ϕ_1 et ϕ_2. De la même manière, le dossier `b_1_x_strategie_preuve/psl2spot_different_i/` contient la preuve de théorème illustrée par l'expression B.3.

B.2 Mise en œuvre de la méthode : outils associés

Dans cette section, nous introduisons brièvement les fonctionnalités de chacun des outils composant la mise en œuvre de la méthode. La combinaison de ceux-ci permet une réduction par localisation et une automatisation complète de la vérification formelle, comme décrit dans le chapitre 4.

B.2.1 Construction du *wrapper*

La construction du *wrapper Verilog* est automatisée par un générateur dédié. Il suffit simplement de lui fournir le nom du module *Verilog* à instancier dans le *wrapper*, la liste de ses ports d'entrée, la liste de ses ports de sortie ainsi que la liste des ports de sortie à filtrer au niveau du *wrapper*. Nous fournissons cet outil de construction du *wrapper* sous forme d'un programme nommé *createVerilogWrapper*. Il est développé en TCL, est distribué sous licence libre et est rendu accessible publiquement [1]. Une documentation utilisateur est disponible en exécutant le programme avec l'option `--help` et une documentation développeur peut être générée à partir de l'outil *Doxygen*.

Ressources : le dossier `b_2_x_reduction_localisation/createVerilogWrapper/` contient l'outil de construction du *wrapper Verilog*. Il peut être exécuté pour créer un *wrapper* à partir de données fournies en argument. Un `Makefile` permet d'automatiser la construction d'un *wrapper* en filtrant des sorties.

L'identification des informations à fournir au programme *createVerilogWrapper*, concernant les entrées/sorties du modèle, est automatisée par un module VPI dédié. Il suffit simplement de fournir le module VPI et le circuit décrit en *Verilog* à un simulateur. Notre mise en œuvre de la méthode utilise *Icarus Verilog* [95] comme simulateur, une suite de simulation et synthèse *Verilog* publiée sous licence libre.

Nous fournissons les sources du module VPI, son chargement dans le simulateur extrait le nom du module *Verilog* à instancier dans le *wrapper*, la liste de ses ports d'entrée ainsi que la liste de ses ports de sortie. Ces informations sont formatées sous forme d'une chaîne de caractères contenant les arguments à transmettre au programme *createVerilogWrapper*. Il est développé en C, est distribué sous licence libre et est rendu accessible publiquement [1]. Une documentation développeur peut être générée à partir de l'outil *Doxygen*.

Ressources : le dossier `b_2_x_reduction_localisation/vpi_listIO/` contient les sources du module VPI. Le dossier parent contient la description *Verilog* d'un circuit. Le module VPI peut être compilé et chargé par *Icarus Verilog* pour identifier automatiquement les informations à fournir au programme *createVerilogWrapper* et ainsi créer un *wrapper* pour ce circuit. Un `Makefile` permet d'automatiser la compilation du module, la compilation du circuit décrit en *Verilog* et le chargement du module dans le simulateur.

L'identification des informations concernant les sorties à filtrer, à fournir au programme *createVerilogWrapper*, est extraite depuis les prédicats des propriétés locales par *NuSMV*. Il suffit simplement de lui fournir le fichier où sont définies les propriétés locales, accompagné par la définition d'un modèle vide, pour que la liste des prédicats soit donnée. Aucun outil supplémentaire n'est requis pour cette tâche.

Ressources : le dossier `b_2_x_reduction_localisation/filter/` contient les fichiers nécessaires à l'extraction de prédicats depuis des propriétés locales. Le dossier parent contient le fichier où sont définies les propriétés locales. Un `Makefile` permet d'automatiser l'appel à *NuSMV* et l'extraction des prédicats depuis la liste que celui-ci fournit.

Dans certains cas, le concepteur peut souhaiter conserver certains registres internes qui ne sont pas des sorties du circuit. Dans ce cas, pour éviter une optimisation de ces registres, les sorties du système dépendantes de ces registres doivent apparaître dans la propriété locale à vérifier. Nous devons donc ajouter la conjonction d'une expression, fonction de la sortie à conserver et équivalente à une tautologie. Par exemple, la disjonction entre cette expression et sa négation. L'expression B.5 fournit un exemple de propriété locale exprimée en fonction du registre interne `i_packets_decompresser.r_data_size`. Ici, la sortie `data_size` est dépendante de ce registre : sa présence dans la propriété locale permet de conserver le registre interne.

$$\begin{aligned}&\textbf{always}(\\&\quad((data\_size = 0) \vee \neg(data\_size = 0)) \wedge\\&\quad(clk \wedge reset) \rightarrow (i\_packets\_decompresser.r\_data\_size = 0)\\&\quad)\end{aligned} \tag{B.5}$$

Configurer l'algorithme d'optimisation différemment pour la construction du modèle abstrait (pour préserver les registres internes) n'est pas une solution viable. En effet, le même algorithme d'optimisation et les mêmes configurations doivent être utilisés pour la synthèse du modèle concret et la construction du modèle abstrait. Si nous optons pour cette solution, tous les modèles abstraits sont contraints de conserver les registres internes en question et la réduction par localisation est moins efficace lorsque ceux-ci ne sont pas requis.

Ressources : le dossier `b_2_1_preserver_registres/` contient la définition du décompresseur en langage *Verilog* et la propriété B.5 exprimée en langage SMV. Nos outils peuvent être appelés pour observer la préservation des registres internes. Un `Makefile` permet d'automatiser l'appel à ceux-ci et d'observer le résultat.

B.2.2 Traduction en automates de Büchi

Verilog2SMV se charge de la traduction du circuit décrit en langage *Verilog* vers un automate de Büchi. Pour générer un modèle abstrait du circuit, il suffit de réaliser une synthèse de celui-ci lorsqu'il est encapsulé dans un *wrapper* ; pour générer un modèle concret, il suffit de réaliser une synthèse lorsque le circuit est défini seul. Des détails sur la traduction du circuit en automate de Büchi par *Verilog2SMV* sont fournis dans le chapitre 4, dans la section 4.2.4.

Ressources : le dossier `b_2_x_reduction_localisation/traduction_buchi/` contient la définition d'un circuit et de son *wrapper*, nécessaires à la construction d'un modèle abstrait et du modèle concret. *Verilog2SMV* peut être appelé pour générer chacun de ces deux modèles, selon le choix du circuit à synthétiser. Un `Makefile` permet d'automatiser l'appel à *Verilog2SMV* et la création des deux modèles.

B.2.3 Découpage des propriétés

Le découpage des propriétés locales contenant des quantificateurs est automatisé par un outil dédié. Il suffit simplement de lui fournir le nom du fichier contenant l'ensemble des propriétés. Nous fournissons cet outil de découpage des propriétés sous forme d'un programme nommé *splitLtl*. Il est développé en TCL, est distribué sous licence libre et est rendu accessible publiquement [1]. Une documentation utilisateur est disponible en exécutant le programme avec l'option `--help` et une documentation développeur peut être générée à partir de l'outil *Doxygen*.

L'avantage d'un découpage des propriétés est de permettre la parallélisation lors de la vérification par *model-checking* et la réduction du BDD résultant. En effet, la complexité de la vérification augmente exponentiellement avec la taille de l'automate modélisant le système et la taille de l'automate modélisant la négation de la propriété. Dans le cas d'une propriété possédant un quantificateur, la conjonction des propriétés paramétrées est séparée en plusieurs uniques propriétés non-paramétrées. Ainsi, nous séparons un problème d'augmentation exponentielle de la complexité en plusieurs problèmes, où la nouvelle complexité augmente linéairement avec le nombre de problèmes. Par exemple, pour un automate modélisant le système contenant 2^m états, pour une propriété locale possédant un quantificateur de n propriétés paramétrées, dont chacune induit un automate possédant 2^k états, le *model-checking* consiste à résoudre un problème sur un automate dont le nombre d'états maximum peut atteindre $2^{(m \times n \times k)}$. Dans le cas où nous découpons les propriétés locales, nous réduisons cette vérification et le *model-checking* consiste à résoudre n problèmes sur des automates dont le le nombre d'états maximum peut atteindre $2^{(m \times k)}$.

Notons cependant que les chiffres donnés ici sont majorés au nombre maximum d'états mais que le model checker peut réduire ce nombre en construisant un BDD. Dans certains cas, où l'automate modélisant le système est petit et où chaque propriété locale paramétrée est petite, cette méthode peut se trouver contre-productive. Nous n'avons pas réalisé d'étude de

complexité sur cette approche mais avons observé, avec les automates que nous manipulons dans cette thèse, que nous obtenions de meilleurs résultats avec un découpage des propriétés.

Ressources : le dossier `b_2_x_reduction_localisation/splitLtl/` contient l'outil de découpage de propriétés. Le dossier parent contient la description formelle de propriétés locales. L'exécution de *splitLtl* permet le découpage des propriétés locales en propriétés non-paramétrées uniques. Un `Makefile` permet d'automatiser le découpage.

B.2.4 Conversion de contre-exemples en VCD

NuSMV est utilisé pour la vérification de chaque propriété découpée sur le modèle abstrait. Des détails sur la vérification formelle par *model-checking* sont fournis dans le chapitre 3, dans la section 3.3.1. Dans le cas où une propriété locale est falsifiée, nous complétons le contre-exemple fourni par *NuSMV* par un traitement permettant une conversion en VCD. Le résultat est un chronogramme observable avec un outil de visualisation [1] exactement comme lors d'une simulation de circuit. Nous fournissons cet outil de conversion sous forme d'un programme nommé *smvTrace2vcd*. Il est développé en TCL, est distribué sous licence libre et est rendu accessible publiquement [1]. Une documentation utilisateur est disponible en exécutant le programme avec l'option `--help` et une documentation développeur peut être générée à partir de l'outil *Doxygen*.

Notons que la différence majeure entre un contre-exemple fourni par *NuSMV* et un chronogramme issu d'une simulation de circuit est la présence de boucles. En effet, les contre-exemples que nous manipulons décrivent des traces d'exécution d'un automate de Büchi, soit des traces d'exécution qui passent infiniment souvent par un état final. *NuSMV* symbolise cette infinité par la présence de boucles [75] : un sous-ensemble de la trace répété infiniment. En plus de représenter au format VCD l'état du système et les différentes variables d'états, *smvTrace2vcd* ajoute donc également un chronogramme supplémentaire, pour chaque boucle, indiquant si la trace d'exécution se situe à l'intérieur ou hors de celle-ci.

Ressources : le dossier `b_2_x_reduction_localisation/smvTrace2vcd/` contient l'outil de conversion de contre-exemples au format VCD et un fichier contenant un contre-exemple tel que fourni par *NuSMV*. L'exécution de *smvTrace2vcd* permet la création d'un chronogramme au format VCD qui peut être visualisé par le logiciel libre *GTKwave*. Un `Makefile` permet d'automatiser la conversion et la visualisation du chronogramme.

B.2.5 Automatisation complète

Notre mise en œuvre de la méthode de vérification formelle automatisée regroupe plusieurs outils décrits dans les précédentes sections. Nous accompagnons ceux-ci d'un outil d'automatisation qui permet d'exécuter toutes les étapes de notre approche. Celui-ci est baptisé *nusmvall*, est distribué sous licence libre et est également rendu accessible publiquement [1]. Une documentation utilisateur est disponible en exécutant l'outil avec l'option `--help` et une documentation développeur peut être générée à partir de l'outil *Doxygen*.

1. Dans cette thèse, nous utilisons le logiciel libre *GTKWave* pour visualiser les chronogrammes.

Ressources : le dossier `b_2_5_nusmvall/` contient l'ensemble des outils qui composent notre mise en œuvre de la méthode de vérification formelle automatisée. Dans cette thèse, les ressources complémentaires qui proposent une vérification formelle automatisée utilisent *nusmvall* présent à cet emplacement.

Annexe C

Théorème pour le modèle zéro : détails de la vérification

Sommaire

Le chapitre 5 décrit la méthode de modélisation et de vérification formelle de notre racine de confiance statique. La preuve de théorème est détaillée dans la section 5.5. Dans cette annexe, nous fournissons des informations techniques sur l'architecture du transducteur, du décodeur et du décompresseur de traces. Nous complétons avec les détails de la vérification de propriétés locales (par *model-checking*) sur ces architectures et de la preuve des différentes obligations de preuve. La section C.1 détaille la vérification formelle du transducteur, la section C.2 celle du décodeur et la section C.3 celle du décompresseur. Les sous-sections C.5.1 et C.5.2 détaillent la preuve manuelle de spécification complexes et la vérification d'invariants, qui constituent des perspectives pour notre méthode de vérification formelle, décrite dans le chapitre 4.

C.1 Transducteur

Le transducteur est le cœur de notre moniteur matériel : il traduit le résultat du décodage et de la décompression des traces d'exécutions en des signaux qui représentent les variables d'entrée des automates de sécurité. Cette traduction vérifie des propriétés locales P_1, qui ne sont pas suffisantes pour prouver le lemme L_{0_A}. La preuve de L_{0_A} nécessite donc d'établir des obligations de preuves A_1 pour le décodeur et le décompresseur. Dans cette section, nous commençons par définir les prédicats que nous introduisons (nécessaires à l'expression des obligations de preuves), nous décrivons les propriétés vérifiées par *model-checking* sur le transducteur, puis nous détaillons la preuve.

C.1.1 Architecture

Notre transducteur traduit les adresses de destination décodées et des informations sur la présence d'exception dans un langage accepté par les automates de sécurité. La figure C.1 donne une représentation des entrées/sorties du transducteur.

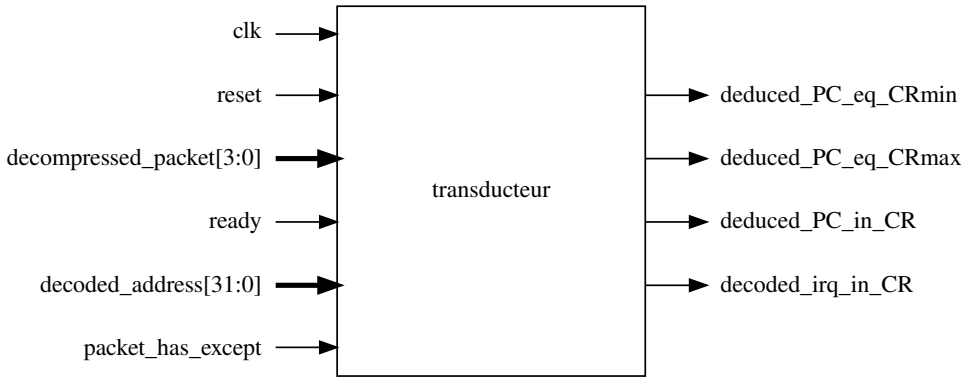

FIGURE C.1 – Entrées/sorties du transducteur

Le transducteur obtient les informations sur les traces depuis le décompresseur et le décodeur. Le décompresseur fournit l'information sur le type de paquet décompressé depuis un vecteur de 4 bits (*decompressed_packet*) et un bit d'activation (*ready*). Le décodeur fournit l'adresse de destination depuis un vecteur de 32 bits (*decoded_address*) et un bit informant de la présence d'un exception. Le transducteur possède également une entrée *reset* qui est connectée à un bouton poussoir et n'est utilisé qu'à des fins de débug.

L'information est accédée lorsque le décompresseur active le bit *ready*. À cet instant, le transducteur déduit la présence du pointeur d'instruction (PC) dans notre région critique (CR), c'est-à-dire la région mémoire où est stocké le code de la fonction d'attestation. Cette déduction se manifeste par la variation du signal *deduced_PC_in_CR*. Le transducteur déduit également si le pointeur d'instruction se situe à la première instruction de la fonction d'attestation (CR_min) ou à la dernière (CR_max). Cette déduction se manifeste par la variation des signaux *deduced_PC_eq_CRmin* et *deduced_PC_eq_CRmax*. Il transmet aussi le signal *decoded_irq_in_CR* qui indique si une exception a été levée durant l'exécution de la fonction d'attestation.

Une spécification importante est que notre transducteur doit réaliser une déduction correcte pour ces informations, qui constituent les variables d'entrée des automates de sécurité. Un autre aspect important du transducteur est qu'il n'introduit aucune latence : la déduction des prédicats est immédiate lorsque le décompresseur indique que la donnée est prête (c'est-à-dire à la réception du dernier octet du paquet). Les spécifications suivantes sont vérifiées par le transducteur :

- Lorsqu'un paquet *isync* ou *branch* est disponible, que *ready* est levé, sur front montant de *clk*, le signal *deduced_PC_in_CR* est levé si l'adresse de destination se situe dans la région critique.
- Lorsqu'un paquet *isync* ou *branch* est disponible, que *ready* est levé, sur front montant de *clk*, le signal *deduced_PC_in_CR* est baissé si l'adresse de destination se ne situe pas dans la région critique.
- Le signal *deduced_PC_in_CR* ne varie pas si aucun paquet *isync* ou *branch* n'est disponible.
- Lorsqu'un paquet *isync* ou *branch* est disponible, que *ready* est levé, sur front montant de *clk*, le signal *deduced_PC_eq_CRmin* est levé si l'adresse de destination se situe à la première instruction de la fonction d'attestation.
- Lorsqu'un paquet *isync* ou *branch* est disponible, que *ready* est levé, sur front montant de *clk*, le signal *deduced_PC_eq_CRmax* est levé si l'adresse de destination se situe à la dernière instruction de la fonction d'attestation.
- Lorsqu'un paquet *branch* est disponible, que *ready* est levé, sur front montant de *clk*, le signal *decoded_irq_in_CR* est levé si le signal *packet_has_except* est levé.
- Les signaux *deduced_PC_eq_CRmin*, *deduced_PC_eq_CRmax* et *decoded_irq_in_CR* ne restent pas actifs plus d'une période d'horloge.

Ressources : le dossier `c_1_1_transducteur/` contient la description du transducteur en langage *Verilog*. Il est possible de faire appel à *Verilog2SMV* pour construire le modèle concret en langage SMV. Un `Makefile` permet d'automatiser la construction et de donner le nombre de lignes de code.

C.1.2 Notations

Les entrées/sorties du transducteur sont représentées sur la figure C.1. Pour prouver le lemme L_{0_A}, nous n'utilisons pas de prédicats représentant les signaux de sortie *deduced_PC_eq_CRmax*, *deduced_PC_eq_CRmin* et *decoded_irq_in_CR*, respectivement utiles à la preuve des spécifications LTL_3, LTL_4 et LTL_5. Nous utilisons simplement le prédicat représentant le signal de sortie *deduced_PC_in_CR*, que nous notons $(PC_d \in CR)$ pour plus de lisibilité.

La plupart des variables d'entrée du transducteur sont utiles à la preuve du lemme L_{0_A}. Néanmoins, nous faisons abstraction de toutes les valeurs possibles que peuvent prendre les adresses décodées, les adresses contenues dans les paquets décompressées et les adresses contenues dans les paquets en cours d'émission par *CoreSight*. À la place, nous considérons simplement les cas où ces adresses sont contenues dans CR ou extérieures à CR. De la même manière, nous ne considérons pas toutes les valeurs possibles pour le type de paquet décompressé et le type de paquet en cours d'émission par *CoreSight*. À la place, nous considérons simplement les cas où ces paquets sont de type *isync*, *branch* ou aucun des deux.

Nous définissons les prédicats suivants afin d'établir nos obligations de preuves pour le décodeur et le décompresseur et de prouver le lemme L_{0_A} :

— ($decompressed\_packet = branch$) signifie que la variable d'entrée *decompressed_packet* du transducteur prend une valeur qui représente que le décompresseur indique une décompression d'un paquet de type *branch*.

— ($decompressed\_packet = isync$) signifie que la variable d'entrée *decompressed_packet* du transducteur prend une valeur qui représente que le décompresseur indique une décompression d'un paquet de type *isync*.

— ($decoded\_address \in CR$) signifie que la variable d'entrée *decoded_address* du transducteur prend une valeur qui représente que le décodeur fournit une adresse décodée située dans CR.

— ($decompressed\_address \in CR$) signifie que la variable d'entrée *data* du décodeur est un paquet qui contient une adresse décompressée située dans CR. Bien entendu, tous les bits d'adresses ne sont pas forcément contenu dans la donnée *data*, certains peuvent être implicites.

C.1.3 Propriétés locales

Une description des propriétés locales est donnée dans la section 5.5.3.1. Dans cette section, nous donnons les expressions formelles des propriétés P_{enter}, P_{in}, P_{exit}, P_{out} et P_{start}. Celles-ci sont respectivement décrites par les expression C.1, C.2, C.3, C.4 et C.5 ci-après.

$$
\begin{aligned}
P_{enter} : \mathbf{always}(& \\
& clk \wedge (\\
& ready \wedge (\\
& \quad ((decompressed\_packet = isync) \wedge (decoded\_address \in CR)) \\
& \quad \vee ((decompressed\_packet = branch) \wedge (decoded\_address \in CR)) \\
&)) \\
& \rightarrow \\
& (\mathbf{next}(PC_d \in CR)) \\
&)
\end{aligned}
\tag{C.1}
$$

$$
\begin{aligned}
P_{in} : \mathbf{always}(& \\
& clk \wedge (PC_d \in CR) \wedge \neg(\\
& ready \wedge (\\
& \quad ((decompressed\_packet = isync) \wedge \neg(decoded\_address \in CR)) \\
& \quad \vee ((decompressed\_packet = branch) \wedge \neg(decoded\_address \in CR)) \\
&)) \\
& \rightarrow \\
& (\mathbf{next}(PC_d \in CR)) \\
&)
\end{aligned}
\tag{C.2}
$$

$$
\begin{aligned}
P_{exit} : \textbf{always}(& \\
& clk \wedge (\\
& ready \wedge (\\
& \quad ((decompressed\_packet = branch) \wedge \neg(decoded\_address \in CR)) \\
&)) \\
& \rightarrow \\
& (\textbf{next}(\neg(PC_d \in CR))) \\
&)
\end{aligned}
\tag{C.3}
$$

$$
\begin{aligned}
P_{out} : \textbf{always}(& \\
& clk \wedge \neg(PC_d \in CR) \wedge \neg(\\
& ready \wedge (\\
& \quad ((decompressed\_packet = isync) \wedge (decoded\_address \in CR)) \\
& \vee ((decompressed\_packet = branch) \wedge (decoded\_address \in CR)) \\
&)) \\
& \rightarrow \\
& (\textbf{next}(\neg(PC_d \in CR))) \\
&)
\end{aligned}
\tag{C.4}
$$

$$P_{start} : \neg(PC_d \in CR) \tag{C.5}$$

Ressources : le fichier `5_5_3_mc_transducteur/transducer.spec.smv` contient la définition des propriétés locales P_{enter}, P_{in}, P_{exit}, P_{out} et P_{start} en langage SMV. Notre méthode de vérification formelle automatisée permet de vérifier, par *model-checking*, que le transducteur vérifie ces propriétés.
Un `Makefile` permet d'automatiser cette vérification.

C.1.4 Lemmes intermédiaires

Les lemmes intermédiaires, permettant de prouver le lemme L_{0_A} depuis les propriétés locales P_1, sont décrits dans la section 5.5.3.2. Dans cette section, nous indiquons les propriétés et axiomes nécessaires à leur démonstration.

Chacun de ces quatre lemmes intermédiaires est respectivement obtenu à partir de chacune des quatre premières propriétés locales vérifiées sur le transducteur (expressions C.1, C.2, C.3 et C.4). Le lemme L_{enter} est prouvé à partir de la propriété P_{enter}, le lemme L_{in} à partir de la propriété P_{in}, etc. Des axiomes et des obligations de preuve sont également nécessaires à la démonstration.

En ce qui concerne les axiomes, notre modélisation du fonctionnement du matériel est nécessaire : *clk* est toujours actif (expression 5.9). Également, sont nécessaires les axiomes qui modélisent le comportement de *CoreSight* et des branchements indirects :

— l'émission de paquet par *CoreSight* lors d'une entrée de PC dans CR (expression 5.10) sert à prouver L_{enter} ;
— la non-émission de paquet par *CoreSight* lorsque PC reste dans CR (expression 5.11) sert à prouver L_{in} ;
— l'émission de paquet par *CoreSight* lors d'une sortie de PC de CR (expression 5.12) sert à prouver L_{exit} ;
— la non-émission de paquet par *CoreSight* lorsque PC reste hors de CR (expression 5.13) sert à prouver L_{out}.

Dans le cas où la preuve se base sur une émission de paquet par *CoreSight* (lemmes L_{enter} et L_{exit}), nous utilisons notre hypothèse simplifiante **H9** pour abstraire le temps de transmission du paquet et considérer que les paquets sont prêts dès leur émission (expressions 5.5 et 5.6).

C.1.5 Obligations de preuve

La conjonction des obligations de preuve forme A_1 (figure 5.3). Nous pouvons les séparer en deux catégories :

— une levée d'exception lorsque PC est localisé dans CR peut être détectée.
— la décompression et le décodage des paquets est correcte.

En ce qui concerne la détection d'une levée d'exception, cette obligation de preuve est une conséquence de l'axiome décrivant la non-émission de paquet lorsque PC reste dans CR (expression 5.11). Cet axiome stipule que nous n'avons pas de paquet émis si aucune exception n'est levée lorsque PC reste dans CR (présence du prédicat irq).

Nous choisissons pour cette obligation de preuve d'ajouter la spécification LTL_5 (expression 5.1). Nous prenons garde de vérifier que cette spécification puisse être prouvée sans utiliser notre lemme L_{0_A}, sans quoi elle ne pourra pas être prouvée plus tard. Intuitivement, ceci est le cas car :

— lors d'une levée d'exception lorsque PC se trouve dans CR, *CoreSight* émet un paquet de type *branch* contenant des informations d'exception ;
— le décompresseur transmet la donnée au décodeur ;
— le décodeur lève le signal $packet\_has\_except$;
— le transducteur lève le signal $decoded\_irq\_in\_CR$;
— l'automate de sécurité procède à un arrêt critique du microprocesseur.

Il n'est pas nécessaire de déduire la présence de PC dans CR. En effet, selon notre hypothèse simplifiante **H6** et la configuration de la plage d'adresses à tracer (décrite dans la section 5.5.1.3), avoir un paquet de type *branch* suffit pour déterminer que PC se trouve : soit dans CR, soit au niveau du branchement indirect avant CR, soit à CR_{max}. Notre moniteur matériel interdit les exception dans toute la plage d'adresses à tracer, qui inclut CR.

En ce qui concerne la décompression et le décodage des paquets, cette obligation de preuve est une conséquence des axiomes décrivant l'émission de paquet par *CoreSight* (expressions 5.10, 5.11, 5.12 et 5.13). Ces axiomes sont exprimées en fonction des prédicats $available\_packet$ et $packet\_address$. Ces prédicats représentent respectivement le type de paquet émis par *CoreSight* et la valeur de l'adresse contenue dans celui-ci.

Les propriétés locales P_{enter}, P_{in}, P_{exit} et P_{out} que nous vérifions sur le transducteur (expressions C.1, C.2, C.3 et C.4) sont exprimées en fonction des prédicats

decompressed_packet et *decoded_address*. Ces prédicats représentent respectivement le type de paquet identifié par le décompresseur et l'adresse décodée par le décodeur. Nous devons donc définir des obligations de preuve, nous permettant de prouver les lemmes intermédiaires L_{enter}, L_{in}, L_{exit} et L_{out} tout en modélisant le comportement du décompresseur et du décodeur.

Nous avons choisi de définir un total de 16 obligations de preuve : 8 obligations de preuve pour le décodeur et 8 obligations de preuve pour le décompresseur. Ces obligations de preuve traduisent réciproquement un décodage correct des paquets décompressés et une décompression correcte des paquets émis par *CoreSight*. Elles sont regroupées par pairs, et toutes les pairs sont semblables : seuls les prédicats diffèrent d'une pair à une autre.

Les expressions C.6 et C.7 donnent un exemple de pair d'obligations de preuve $A_{in/branch_1}$ et $A_{in/branch_2}$. Cette pair modélise le comportement du décodeur, elle traduit le décodage correct d'un paquet décompressé de type *branch* lorsque l'adresse de destination est située dans CR.

$$
\begin{aligned}
A_{in/branch_1} : \textbf{always}(& \\
& ready \wedge (decompressed\_packet = branch) \wedge (decompressed\_address \in CR) \\
\rightarrow & \\
& (decoded\_address \in CR) \\
)&
\end{aligned}
\tag{C.6}
$$

$$
\begin{aligned}
A_{in/branch_2} : \textbf{always}(& \\
& ready \wedge (decompressed\_packet = branch) \wedge (decoded\_address \in CR) \\
\rightarrow & \\
& (decompressed\_address \in CR) \\
)&
\end{aligned}
\tag{C.7}
$$

L'obligation de preuve $A_{in/branch_1}$ exprime que, si un paquet décompressé de type *branch* est prêt et que l'adresse de destination dans celui-ci est dans CR, alors le décodeur fournit une adresse décodée dans CR.

Réciproquement, l'obligation de preuve $A_{in/branch_2}$ exprime que, si un paquet décompressé de type *branch* est prêt et le décodeur fournit une adresse décodée dans CR, alors le paquet décompressé contient une adresse dans CR.

Pour le décodeur, nous avons quatre pairs semblables à ($A_{in/branch_1}$, $A_{in/branch_2}$), où l'on remplace le type de paquet décompressé (*branch* devient *isync*) et où l'on remplace la présence de l'adresse de destination dans CR (l'adresse de destination est soit dans CR, soit hors de CR ; ceci s'applique à l'adresse décompressée et à l'adresse décodée).

Pour le décompresseur, nous avons également quatre pairs d'obligations de preuve semblables. Les expressions C.8 et C.9 donnent un exemple de pair d'obligations de preuve $B_{in/branch_1}$ et $B_{in/branch_2}$. Cette pair traduit la décompression correcte d'un paquet de type

branch émis par *CoreSight* lorsque l'adresse de destination est située dans CR.

$$B_{in/branch_1} : \textbf{always}(\quad (C.8)$$
$$ready \wedge (available\_packet = branch) \wedge (packet\_address \in CR)$$
$$\rightarrow$$
$$(decompressed\_packet = branch) \wedge (decompressed\_address \in CR)$$
$$)$$

$$B_{in/branch_2} : \textbf{always}(\quad (C.9)$$
$$ready \wedge (decompressed\_packet = branch) \wedge (decompressed\_address \in CR)$$
$$\rightarrow$$
$$(available\_packet = branch) \wedge (packet\_address \in CR)$$
$$)$$

Nos quatre pairs d'obligations de preuve sont également semblables à ($B_{in/branch_1}$, $B_{in/branch_2}$), où l'on remplace le type de paquet émis et le type de paquet décompressé (*branch* devient *isync*) et où l'on remplace la présence de l'adresse de destination dans CR (ceci s'applique à l'adresse dans le paquet émis et dans le paquet décompressé).

Ressources : le dossier `5_5_3_lemmes_intermediaires/` contient le programme qui démontre les 4 lemmes intermédiaires tels que décrits dans la section C.1.4. Cette preuve utilise notre bibliothèque Coq et la bibliothèque `LTL_Coq`.
Dans ce dossier, le fichier `axioms.v` contient les définitions des axiomes telles que décrites dans les sections 5.4.3 et 5.5.1. Le fichier `proof_obligations.v` contient les définitions des obligations de preuve telles que décrites dans la section C.1.5. Les propriétés locales vérifiées sur le transducteur, telles que décrites dans la section C.1.3, sont converties à l'aide de notre programme *psl2coq*.
Un `Makefile` permet d'automatiser la conversion des propriétés locales et la preuve des lemmes intermédiaires.

C.2 Décodeur de traces

C.2.1 Architecture

À partir des traces d'exécution fournies par *CoreSight*, notre décompresseur se charge de reconstruire les paquets non-compressés. Notre décodeur extrait les informations utiles de ces paquets conformément à leur format. La figure C.2 donne une représentation des entrées/sorties du décodeur.

Le décodeur obtient la donnée transmise par le décompresseur depuis un vecteur d'entrée de 15 octets (*data*) et l'information sur le type de paquet décompressé depuis un vecteur de 4 bits (*decompressed_packet*), il possède une entrée de synchronisation via une horloge (*clk*) et un bit d'activation (*ready*). Il possède également une entrée *reset* qui est connectée à un bouton poussoir et n'est utilisé qu'à des fins de débug. La donnée est lue par le décodeur sur l'entrée *data* lorsque le décompresseur active le bit *ready*. À cet instant, le décodeur fournit un vecteur *decoded_address* et un signal *packet_has_except* contenant respectivement l'adresse

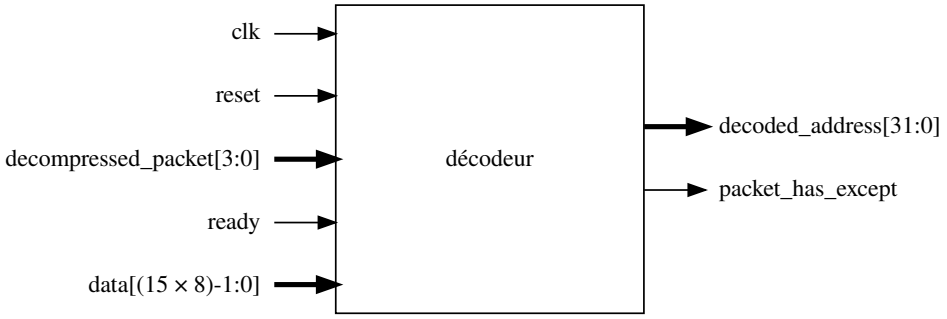

FIGURE C.2 – Entrées/sorties du décodeur

de destination et un booléen indiquant la présence d'une exception. Seuls les paquets *isync* et *branch* sont décodés par le décodeur car ce sont ceux qui contiennent les adresses de destination pour le pointeur d'instruction et les informations d'exception [59].

Une spécification importante est que notre décodeur doit fournir une adresse de destination correcte vis-à-vis des paquets fournis par *CoreSight*. En particulier, dans le cas où *CoreSight* compresse les paquets, une mémorisation des bits d'adresse compressés doit avoir lieu. Un autre aspect important du décodeur est qu'il n'introduit aucune latence : le décodage de l'information est immédiat lorsque le décompresseur indique que la donnée est prête (c'est-à-dire à la réception du dernier octet du paquet). Les spécifications suivantes sont vérifiées par le décodeur :

- Lorsqu'un paquet *isync* ou *branch* est disponible, que *ready* est levé, sur front montant de *clk*, l'adresse de destination est mémorisée depuis *data*.
- Lorsqu'un paquet *isync* est disponible, que *ready* est levé, sur front montant de *clk*, la valeur de *decoded_address* correspond à celle présente dans *data* et le signal *packet_has_except* est baissé.
- Lorsqu'un paquet *branch* est disponible, que *ready* est levé, sur front montant de *clk*, la valeur des bits de poids faibles de *decoded_address* correspond à celle présente dans *data*.
- Lorsqu'un paquet *branch* est disponible, que *ready* est levé, sur front montant de *clk*, la valeur des bits de poids fort de *decoded_address* (qui correspondent à ceux absents de *data*) correspond à celle mémorisée.
- Lorsqu'un paquet *branch* est disponible, que *ready* est levé, sur front montant de *clk*, *packet_has_except* indique la présence de donnée d'exception non nulle dans *data*.

Ressources : le dossier `c_2_1_decodeur/` contient la description du décodeur en langage *Verilog*. Il est possible de faire appel à *Verilog2SMV* pour construire le modèle concret en langage SMV. Un `Makefile` permet d'automatiser la construction et de donner le nombre de lignes de code.

C.2.2 Obligations de preuve

Une description des obligations de preuve est donnée dans la section 5.5.4. Dans cette section, nous décrivons notre expression formelle de ces obligations de preuve.

C.2.3 Décodage du jeu d'instruction

La première difficulté à laquelle nous faisons face est que le format des adresses dans les paquets n'est pas constant : la position des bits d'adresse dans le paquet dépend du jeu d'instruction à l'adresse de destination [59]. Nous devons donc, dans un premier temps, exprimer formellement le décodage du jeu d'instruction.

Cette information est présente dans tous les paquets *isync* et dans les paquets *branch* lors d'un changement par rapport à la dernière émission. Ceci sous-entend qu'elle n'est pas répétée à chaque paquet et que notre décodeur doit la mémoriser. Toutes les propriétés que nous exprimons décrivent donc une mémorisation dans un registre dédié en plus d'une vérification de la valeur.

L'expression C.10 donne un exemple de propriété qui spécifie le décodage et la mémorisation du jeu d'instruction dans le cas d'un paquet *isync* et du jeu d'instruction ARM. Dans cet exemple, le bit de poids faible (index 0) du deuxième octet (index 1) de la donnée prend la valeur zéro dans le paquet. Le prédicat $decoded\_isetstate$ représente la valeur du vecteur qui encode le jeu d'instruction ; celui peut prendre les valeurs énumérées arm, $thumb$ ou $jazelle$. Le prédicat $r\_decoded\_isetstate$ représente la valeur d'un registre de même taille que ce vecteur, dédié à la mémorisation.

$$
\begin{aligned}
&\textbf{always}(\\
&\quad clk \wedge ready \wedge (decompressed\_packet = isync) \\
&\quad \wedge \neg(data[(8 \times 1) + 0 : (8 \times 1) + 0]) \\
&\rightarrow \\
&\quad (decoded\_isetstate = arm) \wedge (r\_decoded\_isetstate = decoded\_isetstate) \\
&\quad)
\end{aligned}
\tag{C.10}
$$

Dans le cas d'une réception d'un paquet *branch*, une condition est ajoutée sur chacun des bits de compression nécessaire à la présence du cinquième octet, qui contient l'information sur le jeu d'instruction.

Afin de compléter la vérification formelle du jeu d'instruction, nous vérifions également deux propriétés. La première est que la valeur du registre ne varie pas tant que nous ne recevons pas un paquet contenant une nouvelle valeur. La seconde est que la valeur du jeu d'instruction considérée par le décodeur est égale à celle du registre dans le cas où nous recevons un paquet et que celui-ci ne contient pas de nouvelle valeur.

L'expression C.11 donne un exemple de propriété qui spécifie que la valeur du registre n'est pas modifiée dans le cas du jeu d'instruction ARM.

$$
\begin{aligned}
&\textbf{always}(\\
&\quad clk \wedge (r\_decoded\_isetstate = arm) \wedge (\\
&\qquad \neg(ready \wedge (decompressed\_packet = isync)) \wedge \neg(ready \wedge (decompressed\_packet = branch) \\
&\qquad\quad \wedge (data[(8 \times 0) + 7 : (8 \times 0) + 7]) \\
&\qquad\quad \wedge (data[(8 \times 1) + 7 : (8 \times 1) + 7]) \\
&\qquad\quad \wedge (data[(8 \times 2) + 7 : (8 \times 2) + 7]) \\
&\qquad\quad \wedge (data[(8 \times 3) + 7 : (8 \times 3) + 7]) \\
&\qquad) \\
&\quad) \\
&\rightarrow \\
&\quad \textbf{next}(r\_decoded\_isetstate = arm) \\
&\ )
\end{aligned}
\tag{C.11}
$$

L'expression C.12 donne la propriété qui spécifie que le décodeur considère la valeur du registre dans le cas de la réception d'un paquet sans information sur le jeu d'instruction.

$$
\begin{aligned}
&\textbf{always}(\\
&\quad clk \wedge ready \wedge (\\
&\qquad \neg((decompressed\_packet = isync)) \wedge \neg((decompressed\_packet = branch) \\
&\qquad\quad \wedge (data[(8 \times 0) + 7 : (8 \times 0) + 7]) \\
&\qquad\quad \wedge (data[(8 \times 1) + 7 : (8 \times 1) + 7]) \\
&\qquad\quad \wedge (data[(8 \times 2) + 7 : (8 \times 2) + 7]) \\
&\qquad\quad \wedge (data[(8 \times 3) + 7 : (8 \times 3) + 7]) \\
&\qquad) \\
&\quad) \\
&\rightarrow \\
&\quad (decoded\_isetstate = r\_decoded\_isetstate) \\
&\ )
\end{aligned}
\tag{C.12}
$$

Nous avons identifié un total de 9 propriétés pour spécifier le décodage du jeu d'instruction. La définition de ces propriétés est rendue disponible publiquement et distribuée sous licence libre [1]. Une fois vérifiées, nous considérons correcte la valeur du prédicat *decoded_isetstate* et que nous pouvons l'utiliser pour définir des propriétés qui spécifient le décodage des adresses de destination.

Ressources : le fichier `5_5_4_mc_decodeur/decoded_isetstate.spec.smv` contient la définition des 9 propriétés que nous avons identifiées pour vérifier la correction du décodage du jeu d'instruction. Notre méthode de vérification formelle automatisée permet de s'assurer, par *model-checking*, que le décodeur vérifie ces propriétés.
Un `Makefile` permet d'automatiser cette vérification.

C.2.4 Décodage des adresses de destination

Le décodage des adresses de destination est spécifié en fonction du type de paquet décompressé et du jeu d'instruction considéré par le décodeur. Dans le cas des paquets *isync*, il n'y a aucune condition : tous les bits d'adresses sont présents et il n'y a pas de dépendance au jeu d'instruction. Nous vérifions que l'adresse décodée prend bien la valeur de l'adresse décompressée et que les bits d'adresses sont mémorisés.

Dans le cas des paquets *branch*, nous spécifions le décodage des bits d'adresse en fonction du jeu d'instruction (considéré par le décodeur) et de la présence des bits de compression. Nous vérifions que les bits d'adresse présents dans le paquet sont égaux dans la valeur décodée et sont mémorisés. Nous vérifions également que les bits d'adresse absents du paquets sont, dans la valeur décodée, égaux à ceux mémorisés.

Pour finir, nous vérifions que les valeurs des bits d'adresse mémorisés ne sont pas modifiés si aucun paquet (*isync* ou *branch*) contenant une nouvelle valeur n'est reçu.

Les propriétés PSL résultantes vérifient les valeurs des bits de donnée de la même manière que les propriétés représentées par les expressions C.10, C.11 et C.12.

Nous avons identifié un total de 35 propriétés pour spécifier le décodage des adresses de destination en fonction de la donnée décompressée. La définition de ces propriétés est rendue disponible publiquement et distribuée sous licence libre [1]. Une fois vérifiées, nous considérons que l'adresse décodée est identique à l'adresse décompressée. Nos quatre pairs d'obligations de preuve pour le décodeur sont donc vérifiées.

Ressources : le fichier `5_5_4_mc_decodeur/decoded_address.spec.smv` contient la définition des 35 propriétés que nous avons identifiées pour vérifier la correction du décodage des adresses. Notre méthode de vérification formelle automatisée permet de s'assurer, par *model-checking*, que le décodeur vérifie ces propriétés.
Un `Makefile` permet d'automatiser cette vérification.

Notons néanmoins que cette vérification apporte de nouvelles obligations de preuves pour le décompresseur, celles-ci forment A_2 (figure 5.3). En effet, le décodage des adresses n'est valide que lorsque les paquets *isync* et *branch* sont correctement identifiés et considérés prêts par le décompresseur (présence des prédicats *ready* et *decompressed_packet* dans nos propriétés). Or comme l'identification est réalisée à partir des entêtes des paquets [59], il devient nécessaire de distinguer à quels instants un paquet *isync* ou *branch* débute. En conséquence, une obligation de preuve pour le décompresseur est de réaliser une décompression et une identification correcte de tous les types de paquets.

C.2.5 Détection de la présence d'exception

Un dernier point concernant la vérification formelle du décodeur est que la détection de la présence d'exception dans les paquets de type *branch* constitue aussi une obligation de preuve. En effet, même si cette vérification n'est pas directement utilisée dans la preuve du lemme L_{0_A} et donc de la spécification LTL_2, celle-ci est nécessaire pour la preuve de la spécification LTL_5 (représentée par l'expression 5.1), comme discuté dans la section C.1.5.

Nous ajoutons donc 2 propriétés qui vérifient la valeur du prédicat *packet_has_except* en fonction des bits de compression présents dans les paquets *branch*. La première propriété

spécifie que le prédicat *packet_has_except* est vérifié lorsque les informations d'exception sont contenues dans le paquet. La seconde propriété spécifie que ce prédicat est falsifié lorsque les informations d'exception sont absentes du paquet.

Ressources : le fichier `5_5_4_mc_decodeur/packet_has_except.spec.smv` contient la définition des 2 propriétés que nous avons identifiées pour vérifier la détection de la présence d'exception. Notre méthode de vérification formelle automatisée permet de s'assurer, par *model-checking*, que le décodeur vérifie ces propriétés.
Un `Makefile` permet d'automatiser cette vérification.

C.3 Décompresseur de traces

C.3.1 Architecture

Une description du décompresseur de traces est donnée dans le chapitre 4, dans la section 4.3.1. Nous invitons le lecteur à se référer à cette section pour obtenir des informations sur les spécifications. La figure C.3 donne de nouveau une représentation des entrées/sorties du décompresseur.

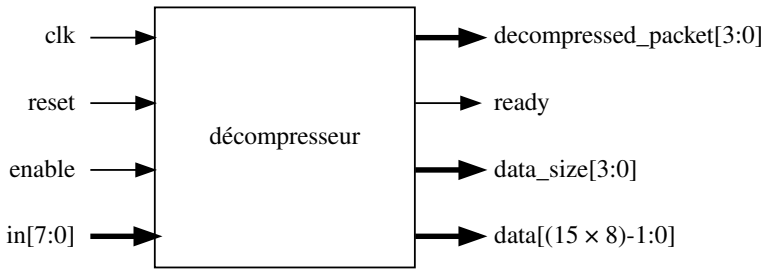

FIGURE C.3 – Entrées/sorties du décompresseur

Notons que dans cette annexe, nous nous intéressons également à l'identification du paquet décompressé, qui avait été omise lors du chapitre 4. Nous ajoutons donc sa spécification ici :

— La sortie *decompressed_packet* donne un identifiant unique pour le paquet en cours de décompression. Cette valeur est placée dès la réception du premier octet et conservée jusqu'à la réception du dernier octet. L'identifiant de valeur nulle indique qu'aucun paquet n'est en cours de décompression.

Notons également que l'entrée *reset* est connectée à un bouton poussoir et n'est utilisé qu'à des fins de débug.

C.3.2 Obligations de preuve

Notre preuve du lemme L_{0_A} requiert quatre pairs d'obligations de preuve pour le décompresseur. Ces obligations de preuve décrivent que, lorsqu'un paquet *isync* ou *branch* est prêt, alors l'adresse décompressée se trouve (ou non) dans CR lorsque l'adresse transmise par *CoreSight* se trouve (ou non) dans CR. Comme pour le décodeur, nous vérifions une propriété plus forte : que lorsqu'un paquet *isync* ou *branch* est prêt, alors l'adresse décompressée est identique à l'adresse transmise par *CoreSight*. Également, nous vérifions que l'identification de

tous les types de paquets est correcte, afin de satisfaire l'obligation de preuve introduite par la vérification formelle du décodeur. La conjonction des obligations de preuve forme P_3 (figure 5.3).

Nous employons la même méthode de vérification que celle du décodeur, c'est-à-dire que nous traduisons manuellement la documentation de *CoreSight* en propriétés PSL, que nous vérifions sur le décompresseur.

C.3.3 Identification de tous les types de paquets

Afin de décompresser correctement les paquets de type *isync* ou *branch*, le décompresseur doit d'abord pouvoir identifier un paquet. Le résultat de cette identification est donné par la sortie *decompressed_packet*. Afin de spécifier le comportement de cette sortie selon la documentation de *CoreSight*, nous vérifions les propriétés suivantes :

- à l'état initial, le décompresseur identifie qu'aucun paquet est en cours de décompression ($decompressed\_packet = none$).
- si $decompressed\_packet = none$ et que le décompresseur reçoit un en-tête à l'état suivant, alors le décompresseur n'identifie pas que qu'aucun paquet est en cours de décompression ($decompressed\_packet \neq none$) dès l'état suivant.
- si $decompressed\_packet \neq none$ et que le paquet n'est pas prêt ($\neg ready$), alors la valeur de *decompressed_packet* est identique à l'état suivant.
- si un paquet est prêt (*ready*) et que le décompresseur ne reçoit pas un entête à l'état suivant, alors le décompresseur identifie qu'aucun paquet est en cours de décompression ($decompressed\_packet = none$) dès l'état suivant.
- pour tous les en-têtes de tous les paquets (par exemple, *isync*), si un paquet est prêt (*ready*) ou qu'aucun paquet est en cours de décompression ($decompressed\_packet = none$), que l'en-tête est reçu à l'état suivant, alors le décompresseur identifie le paquet en cours de décompression (par exemple, $decompressed\_packet = isync$).

Nous avons identifié un total de 14 propriétés pour spécifier l'identification de tous les types de paquets. La définition de ces propriétés est rendue disponible publiquement et distribuée sous licence libre [1]. Notons que les paquets de synchronisation (de type *async*) sont ignorés par le décompresseur : les sous-systèmes permettant l'analyse des traces dans des conditions réelles d'utilisation sont dédiés à cette tâche et ne sont pas décrits dans ce manuscrit.

Ressources : le fichier `5_5_5_mc_decompresseur/decompressed_packet.spec.smv` contient la définition des 14 propriétés que nous avons identifiées pour vérifier la correction de l'identification de tous les paquets. Notre méthode de vérification formelle automatisée permet de s'assurer, par *model-checking*, que le décompresseur vérifie ces propriétés.
Un `Makefile` permet d'automatiser cette vérification.

Nous considérons suffisantes ces spécifications pour garantir une identification correcte de tous les types de paquets. Néanmoins, ceci n'est vérifié que dans le cas où le décompresseur détecte correctement lorsqu'un paquet est prêt (présence du prédicat *ready*). Une nouvelle obligation de preuve consiste alors à vérifier cette détection.

C.3.4 Détection d'un paquet prêt

Pour déterminer si la donnée transmise par *CoreSight* est un en-tête, le décompresseur doit pouvoir détecter lorsque le paquet précédent se termine : lorsqu'il est prêt. Le résultat de cette détection est donné par la sortie *ready*. Afin de spécifier le comportement de cette sortie selon la documentation de *CoreSight*, nous procédons comme suit :

— un registre interne sauvegarde l'index de l'octet attendu dans le paquet. Un index 0 signifie que le décompresseur attend un en-tête.
— pour tout paquet en cours de décompression, lors de la réception d'un octet de donnée, le décompresseur observe les bits de compression et détermine quel octet sera le prochain attendu. La valeur du registre interne est modifiée en conséquence.
— si l'octet reçu est le dernier du paquet, alors la valeur du registre interne est remise à 0 et le signal *ready* est levé. Sinon, le signal *ready* est baissé.

Nous définissons donc une propriété PSL pour chaque octet de chaque paquet que *CoreSight* peut transmettre, soit un total de 64 propriétés. Les expressions C.13 et C.14 donnent deux exemples parmi ces 64 propriétés.

L'expression C.13 donne l'exemple de la détection lors de la décompression du premier octet (index 0) d'un paquet de type *atom*. Dans cet exemple, le bit de compression est le septième bit (numéro 6). Si celui est levé, alors un nouvel octet est attendu : *ready* doit être baissé et le prochain octet attendu est l'octet d'index 1. Réciproquement, si le bit de compression est baissé, alors *ready* doit être levé et le prochain octet attendu sera un entête.

$$
\begin{aligned}
&\textbf{always}(\\
&\quad clk \wedge (decompressed\_packet = atom) \wedge (index = 0) \wedge enable \\
&\rightarrow \\
&\quad (\ \ in[6] \rightarrow (\neg ready \wedge \textbf{next}(index = 1))) \\
&\quad \wedge (\neg in[6] \rightarrow (\ \ ready \wedge \textbf{next}(index = 0))) \\
&\quad)
\end{aligned}
\tag{C.13}
$$

Dans le cas de certains octets, déterminer le prochain octet attendu est plus délicat. C'est le cas du deuxième octet (index 1) d'un paquet de type *branch*. L'expression C.14 donne la spécification de la détection d'un paquet prêt dans ce cas. Dans cet exemple, un nouvel octet sera toujours attendu donc *ready* doit être baissé. Le bit de compression est le huitième bit (numéro 7). Si celui-ci est levé, alors le prochain octet attendu est l'octet d'index 2. Si celui-ci est baissé, alors le septième bit (numéro 6) doit être considéré comme un bit de compression. Si celui-ci est levé, alors le prochain octet attendu est l'octet d'index 5. Sinon, le prochain octet attendu est l'octet d'index 7.

Nous ajoutons également deux propriétés qui expriment que :

— la valeur de l'index est 0 à l'état initial ;
— si aucune donnée n'est transmise par *CoreSight*, alors le signal *ready* est baissé à l'état courant et la valeur de l'index reste inchangée à l'état suivant.

La définition de l'ensemble de ces propriétés est rendue disponible publiquement et distribuée sous licence libre [1].

$$
\begin{aligned}
&\textbf{always}(\\
&\quad clk \wedge (decompressed\_packet = branch) \wedge (index = 1) \wedge enable \\
&\rightarrow \\
&\quad \neg ready \\
&\quad \wedge (\ \ in[7] \rightarrow \textbf{next}(index = 2)) \\
&\quad \wedge (\neg in[7] \rightarrow (\\
&\qquad (\ \ in[6] \rightarrow \textbf{next}(index = 5)) \\
&\qquad \wedge (\neg in[6] \rightarrow \textbf{next}(index = 7)) \\
&\quad)) \\
&\quad)
\end{aligned}
\tag{C.14}
$$

Ressources : le fichier `5_5_5_mc_decompresseur/ready.spec.smv` contient la définition des 66 propriétés que nous avons identifiées pour vérifier la correction de la détection des paquets prêts. Notre méthode de vérification formelle automatisée permet de s'assurer, par *model-checking*, que le décompresseur vérifie ces propriétés.
Un `Makefile` permet d'automatiser cette vérification.

Nous considérons suffisantes ces spécifications pour garantir une détection correcte des paquets prêts. Nous obtenons donc notre obligation de preuve requise pour l'identification correcte de tous les types de paquets. La dernière étape consiste donc à vérifier les quatre pairs d'obligations de preuve requises pour la preuve du lemme L_{0_A}.

C.3.5 Adresses identiques

Comme décrit plus tôt, nous vérifions que l'adresse décompressée est identique à l'adresse transmise par *CoreSight*. Pour cela, nous vérifions que tous les bits des octets contenant les adresses de destination sont bien présents dans la donnée décompressée lorsque les paquets *isync* et *branch* sont prêts. Cette vérification est donc nécessaire pour les 5 premiers octets des paquets de type *isync* et les 7 premiers octets des paquets de type *branch* (si l'on souhaite vérifier la décompression des informations d'exception ; sinon, seuls les 5 premiers octets suffisent).

Les détails de la vérification sont donnés dans le chapitre 4, plus particulièrement dans la section 4.4. L'expression 4.3 décrit la spécification dans le cas des paquets de type *branch*. Nous vérifions une propriété similaire dans le cas des paquets de type *isync*. Notons que cette spécification est exprimée en fonction du prédicat *ready*, dont la correction a été vérifiée, mais également en fonction du prédicat *data_size*, qui est un vecteur indiquant le nombre d'octets utiles présents dans le paquet (soit la taille du paquet).

La vérification de ces spécifications n'est donc correcte que dans le cas où la taille du paquet est correctement identifiée. Une nouvelle obligation de preuve consiste alors à vérifier cette identification.

C.3.6 Taille du paquet

Afin de déterminer la taille du paquet transmis par *CoreSight*, nous utilisons la même stratégie que lors de la détection d'un paquet prêt. À savoir :

— un registre interne sauvegarde le nombre de paquets que le décompresseur a reçu. Il est initialisé à la valeur 0.
— pour tout paquet en cours de décompression, lors de la réception d'un octet de donnée, s'il n'est pas le dernier du paquet (le signal *ready* est baissé), alors la valeur du registre interne est incrémentée.
— si l'octet reçu est le dernier du paquet (le signal *ready* est levé), alors le registre interne est réinitialisé.
— si aucun octet n'est reçu (le signal *enable* n'est pas actif), alors la valeur du registre interne est inchangée.

À partir de ces spécifications, nous vérifions que le vecteur *data_size* prend la valeur du registre interne ajoutée à 1 lorsque le signal *ready* est levé. Nous considérons donc notre registre interne comme un compteur que nous réinitialisons à la réception du dernier octet ; le vecteur *data_size* prend la valeur que ce compteur devrait avoir s'il était incrémenté.

Nous avons identifié un total de 5 propriétés pour spécifier l'identification de la taille des paquets. La définition de ces propriétés est rendue disponible publiquement et distribuée sous licence libre [1].

Ressources : le fichier `5_5_5_mc_decompresseur/data_size.spec.smv` contient la définition des 5 propriétés que nous avons identifiées pour vérifier la correction de l'identification de la taille des paquets. Notre méthode de vérification formelle automatisée permet de s'assurer, par *model-checking*, que le décompresseur vérifie ces propriétés.
Un `Makefile` permet d'automatiser cette vérification.

C.4 Assemblage

Les vérifications décrites dans les sections C.2.2 et C.3.2 forment notre traduction manuelle de la documentation de *CoreSight* [59]. Nous considérons axiomatique que cette traduction manuelle est correcte. Ainsi, nos spécifications sont suffisantes pour garantir que l'adresse décompressée est identique à l'adresse transmise par *CoreSight* lors de la réception du dernier octet (lorsque *ready* est levé) des paquets *isync* et *branch*. En d'autres termes, nous considérons axiomatique une implication des 8 pairs d'obligations de preuve, requises par la vérification du transducteur et décrites dans la section C.1.5.

Les 8 pairs d'obligations de preuve décrivent une détection que l'adresse décodée est contenue dans CR. Comme l'adresse décompressée est identique à celle transmise, nous pouvons considérer, de la même manière, une détection que l'adresse décodée est égale à CR_{min} ou à CR_{max}. De telles propriétés sont utilisées lors de la preuve des spécifications LTL_3 et LTL_4 (décrites par l'expression 5.1).

Les différents sous-systèmes (transducteur, décodeur, décompresseur) sont assemblés dans notre moniteur matériel afin de garantir la sécurité de notre racine de confiance statique. Deux sous-systèmes supplémentaires sont également présents. Ils permettent l'analyse des traces dans des conditions réelles d'utilisation ; c'est-à-dire pour permettre l'analyse des traces de passer à l'échelle. Ces deux sous-systèmes ne sont pas décrits dans ce manuscrit car leur fonctionnement n'a pas d'impact sur la preuve de sécurité. Leur objectif est d'implémenter un flot de décompression tel que décrit dans l'annexe *Overall PFT trace decompression flow* des

spécifications de *CoreSight* [59]. Néanmoins, les sources de ces sous-systèmes sont disponibles publiquement, distribuées sous licence libre et ceux-ci ont été également implémentés [1].

Ressources : le dossier `5_6_2_soundness/` contient les sources pour implémenter l'intégralité de notre périphérique de confiance, incluant tous les sous-systèmes cités précédemment. Il contient également les sources d'un FSBL et d'une application pour exécuter une fonction d'attestation depuis le microprocesseur. Le *reset* du microprocesseur n'est pas connecté : le signal est relié à une LED pour témoigner d'un arrêt critique par le moniteur. Un script gdb permet d'instrumenter openOCD pour la programmation et l'exécution d'un programme malveillant.
Un Makefile permet d'automatiser toutes ces tâches.

C.5 Passage à l'échelle

Le chapitre 4 décrit une stratégie de vérification formelle avec une preuve de théorème automatique. Cette stratégie est potentiellement soumise à l'explosion combinatoire. La solution proposée pour s'en affranchir et de réaliser une preuve manuelle des spécifications et une vérification d'invariants. Dans cette section, nous proposons un exemple d'application de cette solution à la vérification du décompresseur de traces.

C.5.1 Preuve manuelle des spécifications

Malgré plusieurs abstractions, démontrer de manière automatique la spécification 4.3 par la conjonction des propriétés locales peut nécessiter un temps de calcul ou une quantité de mémoire qui est trop élevé. Une solution simple, comme montré sur la figure 4.5, est de démontrer des lemmes intermédiaires où des propriétés temporelles, plus larges que nos propriétés locales mais plus petites que la spécification, sont impliquées. Ensuite, la preuve est séparée en plusieurs étapes : une étape pour chaque lemme intermédiaire.

Une autre solution est de recourir à une preuve manuelle à l'aide d'un assistant de preuves afin de ne pas dépendre de la mémoire de la machine sur laquelle le théorème est démontré. Cette méthode est adaptée à toute vérification formelle lorsque le concepteur maitrise l'assistant de preuves. Des théorèmes génériques, adaptés à n'importe quelle propriété temporelle, peuvent être démontrés et appliqués à chaque propriété locale. Également, il n'est plus nécessaire d'introduire des prédicats pour abstraire des expressions PSL non-interprétées. Dans cette section, nous proposons une stratégie de preuve adaptée au cas d'étude, qui remplace celle décrite dans la section 4.4.2. Elle peut être adaptée à n'importe quel autre cas d'étude.

Dans le cas de notre décompresseur de traces, la preuve manuelle de la spécification se déroule en deux étapes :

— d'abord, nous montrons que nos sept propriétés locales sont valables pour toute valeur de *data_size* supérieure ou égale à 7.
— ensuite, nous montrons que la spécification est vérifiée en effectuant une fusion des expressions PSL contenant les opérateurs **next_event** (étape 2 représentée en rouge sur la figure 4.4).

À ces fins, nous utilisons la bibliothèque `LTL_Coq`, qui fournit les définitions des opérateurs temporels, et une bibliothèque construite pour notre besoin, qui fournit une définition des vecteurs de bits et une extension à la bibliothèque `LTL_Coq`. Cette nouvelle bibliothèque,

majoritairement développée par Guillaume Dupont est distribué sous licence libre et est rendue accessible publiquement [1]. Elle permet, entre autres, la conversion entre entiers naturels et vecteurs de bits d'une taille donnée, la comparaison de vecteurs et la création de formules de logique temporelle à partir du résultat des comparaisons. Cette bibliothèque contient également des théorèmes importants sur les propriétés garanties par les vecteurs de bits. Par exemple, un théorème d'intérêt pour prouver notre spécification stipule que, pour tout entier naturel n, i et j, si j est inférieur à 2^n et i est inférieur à j, alors le vecteur de bit de taille n et de valeur i est inférieur au vecteur de bit de taille n et de valeur j. En d'autres termes, la conversion d'entier naturel en vecteur de bit préserve la monotonie de l'inégalité stricte sous réserve que les valeurs soient inférieures à deux puissance la taille du vecteur.

La première étape de notre preuve consiste à établir un théorème, générique à chacune de nos 7 propriétés locales et conditionné par des comparaisons entre des valeurs d'entiers naturels, puis de l'appliquer à chacune des propriétés locales. Ce théorème permet de remplacer la valeur de tout entier naturel auquel *data_size* doit être supérieur ou égal par 7, sous réserve que nous possédons une propriété temporelle dont cet entier est strictement inférieur à 7. L'expression C.15 représente le dit théorème. Celui-ci est appliqué à chacune des 7 propriétés temporelles avec une valeur pour l'entier i comprise entre 1 et 6 (pour chaque application, l'entier j est égal à 7 et l'entier n est égal à 4, taille du vecteur *data_size*).

Dans cette expression, les symboles n, i et j sont des entiers naturels quelconques ; les symboles ϕ_0, ϕ_1, ϕ_2 et ϕ_3 sont des expressions PSL quelconques ; le symbole bv est un vecteur de bit quelconque de taille n. L'opérateur **uwconst**(i, n) est une fonction mathématique qui transforme tout entier naturel i en un vecteur de bit de taille n et de valeur $i \pmod{2^n}$.

Lors de l'application de ce théorème sur les propriétés locales, les symboles ϕ_0, ϕ_1, ϕ_2 et ϕ_3 sont remplacées par les expressions PSL qui correspondent. Pour que cette application soit valable, les conditions $(i < j)$ et $(j < 2^n)$ forment une obligation de preuve.

$$
\begin{array}{l}
\textbf{forall } n,\ i,\ j : \\
\textbf{forall } \phi_0,\ \phi_1,\ \phi_2,\ \phi_3 : \\
\textbf{forall } bv : \\
(i < j) \wedge (j < 2^n) \\
\quad \rightarrow \quad (\\
\qquad \textbf{always}(\\
\qquad\quad (\phi_0 \wedge \textbf{next}(\phi_1 \wedge \textbf{next\_event}(clk \wedge ready)[1](bv \geq \textbf{uwconst}(i, n)) \wedge \phi_2) \rightarrow \phi_3) \\
\qquad\quad \rightarrow \\
\qquad\quad (\phi_0 \wedge \textbf{next}(\phi_1 \wedge \textbf{next\_event}(clk \wedge ready)[1](bv \geq \textbf{uwconst}(j, n)) \wedge \phi_2) \rightarrow \phi_3) \\
\qquad) \\
\qquad)
\end{array}
\tag{C.15}
$$

La seconde étape de notre preuve consiste à établir un théorème, générique à deux propriété locales quelconques et conditionné par la forme des opérateurs temporels, puis de l'appliquer à chacune de nos propriétés locales. Ce théorème permet de réaliser une fusion de plusieurs expressions PSL dans une implication et dans un opérateur **always**. L'expression C.16 représente le dit théorème. Celui-ci est appliqué six fois : pour fusionner chacune des expressions PSL **next_event** et **next_event_a** présentes dans nos propriétés temporelles.

$$
\begin{aligned}
&\textbf{forall } H_0,\ H_1,\ H_2,\ P_1,\ P_2 : \\
&\quad \textbf{always}(\\
&\qquad (H_0 \wedge \textbf{next}(H_1) \rightarrow \textbf{next}(P_1)) \\
&\qquad \wedge (H_0 \wedge \textbf{next}(H_2) \rightarrow \textbf{next}(P_2)) \\
&\quad \rightarrow \\
&\qquad (H_0 \wedge \textbf{next}(H_1 \wedge H_2) \rightarrow \textbf{next}(P_1 \wedge P_2)) \\
&\quad)
\end{aligned}
\tag{C.16}
$$

Dans cette expression, les symboles H_0, H_1, H_2, P_1, P_2, sont des expressions PSL quelconques. Lors de l'application de ce théorème sur les propriétés locales, ils sont remplacées par les expressions PSL qui correspondent.

Au final, en appliquant simplement les deux théorèmes précédents et avec quelques réécritures basées sur l'associativité de la conjonction, nos différentes fusions permettent de démontrer, en un temps réduit (quelques secondes) et une très faible mémoire, la spécification 4.3.

Ressources : le programme `c_4_1_preuve_coq/proof.v` utilise Coq et les deux bibliothèques, il permet de démontrer que la spécification 4.3 est vérifiée. Lors de cette preuve, les théorèmes décrits par les expressions C.15 et C.16 sont démontrés et appliqués à nos propriétés locales. Le `Makefile` permet d'automatiser sa compilation et son exécution.

C.5.2 Vérification d'invariants

Afin d'illustrer la procédure de vérification d'invariants, nous proposons ici un exemple de vérification dans le but de prouver une propriété temporelle.

Dans le cas de notre décompresseur de traces, nous pouvons raisonner sur les registres internes qui encodent l'état de la machine à états finis, sur le compteur auquel la sortie `data_size` est affectée, ou encore sur la mémoire à laquelle est connectée la sortie `data`.

Pour reprendre l'exemple du registre interne que nous préservions dans la section 4.5.1, nous pouvons par exemple caractériser le comportement de la sortie `data_size` en vérifiant quelques invariants décrivant le comportement du compteur `i_packets_decompresser.r_data_size` :

- la valeur du compteur est remise à zéro après la réception d'un *reset* synchrone (expression C.17) ;
- la valeur du compteur est remise à zéro après que le signal `ready` soit levé (expression C.18) ;
- la valeur du compteur ne change pas après que l'entrée `enable` soit baissée (expression C.19) ;
- la valeur du compteur est incrémentée après que l'entrée `enable` soit levée (expression C.20) ;
- la sortie `data_size` est supérieure à la valeur du compteur lorsque le signal `ready` est levé (expression C.21).

Une spécification formelle de ces différents invariants est décrite par les expressions C.17, C.18, C.19, C.20 et C.21.

$$\mathbf{always}((clk \wedge reset) \rightarrow \mathbf{next}(i\_packets\_decompresser.r\_data\_size = 0)) \tag{C.17}$$

$$\mathbf{always}((clk \wedge ready) \rightarrow \mathbf{next}(i\_packets\_decompresser.r\_data\_size = 0)) \tag{C.18}$$

$$\begin{aligned}\mathbf{always}(&\neg(clk \wedge reset) \wedge \neg(clk \wedge enable) \rightarrow \\ &\mathbf{next}(i\_packets\_decompresser.r\_data\_size) = \\ &\quad i\_packets\_decompresser.r\_data\_size)\end{aligned} \tag{C.19}$$

$$\begin{aligned}\mathbf{always}(&\neg(clk \wedge reset) \wedge \neg(clk \wedge ready) \wedge (clk \wedge enable) \rightarrow \\ &\mathbf{next}(i\_packets\_decompresser.r\_data\_size) = \\ &\quad i\_packets\_decompresser.r\_data\_size + 1)\end{aligned} \tag{C.20}$$

$$\begin{aligned}\mathbf{always}(&\neg(clk \wedge reset) \wedge (clk \wedge ready) \rightarrow \\ &\quad data\_size \geq i\_packets\_decompresser.r\_data\_size)\end{aligned} \tag{C.21}$$

À partir de ces différents invariants, il est possible de prouver une propriété temporelle qui décrit le comportement de la sortie `data_size`. En suivant un raisonnement analogue, nous pouvons établir des invariants décrivant le comportement des registres internes qui affectent la sortie `data` et prouver une propriété temporelle qui décrit son comportement.

Notons que la vérification par *model-checking* de chacun de ces invariants ne requiert que quelques secondes de calcul avec *NuSMV* : une durée grandement inférieure aux temps de calcul listés dans le tableau 4.3.

Plusieurs propriétés temporelles peuvent être démontrées ; à partir de celles-ci, nous pouvons établir une preuve de la propriété temporelle décrite par l'expression 4.4. Notons cependant que *NuSMV* possède une expressivité différente de celle fournie par la bibliothèque `LTL_Coq` que nous sommes obligés d'utiliser dans l'expression de certains invariants. Dans ce cas, une traduction de paradigme devient nécessaire. Par exemple, pour décrire que la valeur d'un vecteur est incrémentée après un évènement, *NuSMV* permet l'écriture de l'expression suivante :

$$\mathbf{next}(bv) = (bv + 1)$$

Dans cette expression, bv est un vecteur de bits et $\mathbf{next}(bv)$ n'est pas une propriété temporelle. En effet, $\mathbf{next}(bv)$ est également un vecteur de bit dont la valeur est celle du vecteur bv à l'état suivant. Lorsque nous utilisons la bibliothèque `LTL_Coq`, l'opérateur **next** ne s'applique que sur des propriétés temporelles et son résultat est lui aussi une propriété temporelle. Afin de convertir un invariant *NuSMV* possédant une telle écriture en une propriété temporelle exprimée à l'aide de la bibliothèque `LTL_Coq`, il devient nécessaire de raisonner sur la valeur à laquelle le vecteur bv est égale : l'égalité entre deux vecteurs représente un prédicat. Nous pouvons par exemple traduire l'expression précédente par :

$$\begin{aligned}&\mathbf{forall}\ n, i : \\ &\quad \mathbf{forall}\ bv : \\ &\qquad (bv = \mathbf{uwconst}(i, n)) \rightarrow \mathbf{next}(bv = \mathbf{uwconst}(i + 1, n))\end{aligned}$$

Dans cette expression, les symboles n et i sont des entiers naturels quelconques et le

symbole bv est un vecteur de bit quelconque de taille n. Celle-ci revient donc à dire que, pour toute valeur i, lorsque le vecteur bv prend la valeur i, il prend la valeur $i + 1$ à l'état suivant.

Ressources : le dossier `c_4_2_invariants/` contient la définition de plusieurs invariants permettant de raisonner sur les registres internes du décompresseur qui affectent la sortie `data`. Notamment, certains invariants aident à décrire le comportement de son neuvième octet dont la vérification pour une valeur prend plusieurs heures (voir le tableau 4.3). Également, les invariants décrits par les expressions C.17, C.18, C.19, C.20 et C.21 sont présents. Le `Makefile` permet d'automatiser la vérification de ces invariants et d'observer un temps de calcul réduit.

Annexe D

Théorème pour l'attestation de configuration : détails de la vérification

Sommaire

Le chapitre 6 décrit la méthode d'attestation de configuration du microprocesseur dans un environnement corrompu. La vérification formelle est détaillée dans la section 6.4. Dans cette annexe, nous fournissons des informations techniques sur l'architecture du nouveau moniteur. Nous complétons avec les détails de la vérification de propriétés locales (par *model-checking*) sur cette architecture et de la preuve de sécurité. Également, nous apportons des considérations sur la validité de l'hypothèse simplifiante **H10**. La section D.1 rappelle le contexte, la section D.2 décrit le modèle d'environnement, la section D.3 décrit l'architecture du nouveau moniteur et la section D.4 détaille la vérification. Finalement, la section D.5 apporte des considérations sur la non-reproductibilité des signaux et de potentielles évolutions de l'architecture proposée.

D.1 Contexte

Nous plaçons de manière continue l'algorithme de confiance et la fonction d'attestation dans le même esclave AXI-Lite. L'exécution du dernier branchement indirect de l'algorithme de confiance a pour destination la première instruction de la fonction d'attestation (référencée par CR_{min} dans le chapitre 5). Comme décrit dans le chapitre 5, la dernière instruction de la fonction d'attestation est elle aussi un branchement indirect dont la destination est référencée par CR_{max}.

D.2 Modèle d'environnement

Dans cette section, nous décrivons l'environnement du nouveau moniteur matériel, à savoir la communication avec la ROM contenant les valeurs attendues des signaux et l'observation de ces mêmes signaux durant la vérification.

D.2.1 Connexions à la ROM

Notre nouveau moniteur matériel accède aux valeurs attendues des signaux (traces d'exécution fournies par *CoreSight* et les signaux observés sur le bus AXI) depuis une ROM de $8Ko$. Afin d'implémenter cette ROM, nous instancions un bloc RAM sur lequel nous forçons la désactivation d'écriture via les bits prévus à cet effet. Cette ROM fournit un vecteur de 64 bits de donnée, nommé *rddata*, stockée à l'adresse demandée par le moniteur. Le vecteur d'adresse, nommé i, est de 10 bits : il peut donc prendre $2^{10} = 1024$ valeurs possibles.

La valeur de l'adresse i permet d'accéder aux valeur attendues des signaux pour chaque front d'horloge : le vecteur stocké à l'adresse 0 contient les signaux attendus au premier front d'horloge, celui stocké à l'adresse 1 contient les signaux attendus au second front d'horloge, etc. Lorsqu'une adresse est placée par le moniteur sur i, alors la ROM place la donnée sur *rddata* au prochain front d'horloge. Réciproquement, si la ROM place la donnée sur *rddata* au prochain front d'horloge, alors c'est que l'adresse est placée sur i. L'expression D.1 fournit un modèle du comportement de la ROM que nous considérons axiomatique, où $ROM(n)$ représente la valeur stockée dans la ROM à l'adresse n.

$$\textbf{forall } n \textbf{ in } \{0:1023\}: \ \textbf{always}((i = n) \leftrightarrow \textbf{next}(rddata = ROM(n))) \tag{D.1}$$

D.2.2 Observation des signaux

Le moniteur observe les signaux et procède à un *reset* du microprocesseur en cas de différence avec les valeurs attendues. Cette vérification ne doit cependant pas avoir lieu en permanence : seulement lors de l'exécution de l'algorithme de confiance. Elle est arrêtée lors de l'exécution de la fonction d'attestation, même si les signaux varient toujours sur le bus AXI. Nous pouvons donc considérer trois phases de fonctionnement pour le nouveau moniteur matériel :

- une phase d'attente, où aucune vérification n'est réalisée, car le microprocesseur n'accède pas à l'esclave AXI-Lite (contenant l'algorithme de confiance et la fonction d'attestation) ;
- une phase de vérification, où la vérification est effective car le microprocesseur exécute l'algorithme de confiance ;
- une phase de fin, où la vérification n'est pas effective malgré des accès sur le bus AXI, car le microprocesseur exécute la fonction d'attestation.

La figure 6.2 représente les signaux à vérifier au début d'une exécution de l'algorithme de confiance. Cette exécution est initiée par une levée du signal $ARREADY$ qui indique un *fetch* des instructions par le microprocesseur. La levée du $ARREADY$ de l'esclave AXI-Lite (contenant l'algorithme de confiance) met donc fin à la phase d'attente et démarre la phase de vérification. Nous nommons ce signal $sw\_arready$.

Durant la phase de vérification, front d'horloge après front d'horloge, le nouveau moniteur incrémente le vecteur d'adresse i et provoque un *reset* du microprocesseur si la valeur d'un

signal observé ne correspond pas à celle attendue. Le nombre de fronts d'horloge fixant la durée de la phase de vérification est choisi à la conception. Nous notons END_ADDR cette valeur. Lorsque ce nombre est atteint par le vecteur i, alors la phase de vérification se termine et la phase de fin débute. La valeur END_ADDR choisie doit être suffisamment élevée pour couvrir les 7 branchements indirects. Par exemple, si l'algorithme de confiance nécessite 7 paquets de 6 instructions et que chaque paquet est *fetché* 2 fois sauf le premier, alors 611 fronts d'horloge sont nécessaires (un *fetch* nécessite $(4 \times 8 + 15)$ fronts d'horloge). Nous choisissons END_ADDR tel que, lorsque i atteint cette valeur, alors le dernier branchement indirect est exécuté et sa destination est CR_{min}. Ceci se traduit par une déduction du transducteur de $(PC_d \in CR)$ à l'état précédent (voir la section 5.5.3). L'expression D.2 décrit ce comportement, que nous considérons axiomatique.

$$\mathbf{always}(\mathbf{next}(i = END\_ADDR) \rightarrow (PC_d \in CR)) \tag{D.2}$$

Durant la phase de fin, la levée du signal *sw_arready* ne provoque pas d'entrée en phase de vérification. Nous considérons que les mémoires cache, MMU et *CoreSight* sont dans l'état attendu par notre racine de confiance statique et que nous pouvons désormais faire confiance aux traces *CoreSight* pour fournir une adresse de destination correcte. La phase de fin se termine donc lorsque le transducteur indique que la valeur déduite du pointeur d'instruction est à CR_{max}. Le moniteur entre alors en phase d'attente : la levée du signal *sw_arready* provoquera de nouveau une entrée en phase de vérification.

D.3 Nouveau moniteur matériel

À partir des signaux observés et des informations transmises par le transducteur, le nouveau moniteur matériel adapte la phase dans laquelle il se trouve. Le cas échéant, il décide si l'algorithme de confiance s'exécute. La figure D.1 donne une représentation des entrées/sorties du nouveau moniteur.

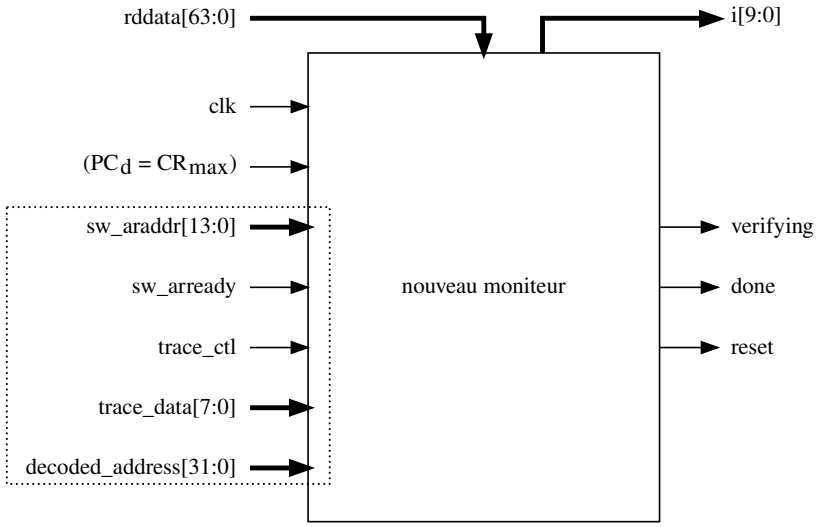

FIGURE D.1 – Entrées/sorties du nouveau moniteur

Le nouveau moniteur communique avec la ROM à l'aide des vecteurs *rddata* et i représentés en haut de la figure D.1. Les entrées encadrées en pointillés représentent les signaux observées, dont les valeurs sont comparées avec le contenu du vecteur *rddata* durant la phase de vérification. En cas de différence, le signal *reset* est levé. La présence dans cette phase est indiquée par le nouveau moniteur avec une levée du signal $verifying$. Lorsque celle-ci se termine, le nouveau moniteur baisse le signal $verifying$ et lève le signal *done* pour indiquer qu'il passe en phase de fin. Le transducteur varie la variable d'entrée ($PC_d = CR_{max}$) pour indiquer au moniteur lorsque la phase d'attente peut reprendre. À cet instant, le nouveau moniteur baisse le signal *done*.

Pour garantir la sécurité de l'exécution de l'algorithme de confiance, le nouveau moniteur doit correctement entrer dans les phases d'attente, de vérification et de fin. De plus, la valeur du vecteur i doit correctement varier pour que la lecture dans la ROM des valeurs attendues corresponde bien à celle du prochain front d'horloge. Les spécifications suivantes sont garanties par le nouveau moniteur :

- lorsque le moniteur est en phase d'attente, si le signal sw_aready est levé au prochain état, alors le moniteur entre en phase de vérification ; sinon, il reste en phase d'attente.
- lorsque le moniteur est en phase de vérification, si le vecteur i atteint le nombre indiquant la fin de la vérification, alors le moniteur entre en phase de fin ; sinon, il reste en phase de vérification.
- lorsque le moniteur est en phase de fin, si le transducteur indique que le pointeur d'instruction atteint CR_{max}, alors le moniteur entre en phase d'attente ; sinon, il reste en phase de fin.
- lorsque le moniteur est en phase d'attente, la valeur du vecteur i est nulle.
- lorsque le moniteur est en phase de vérification, la valeur du vecteur i est incrémentée à chaque front d'horloge.
- lorsque le moniteur est en phase de vérification, toute différence entre les valeurs des signaux observés et leur valeur présente sur le vecteur *rddata* entraîne un *reset* immédiat.

Ressources : le dossier `d_3_0_moniteur/` contient la description du nouveau moniteur en langage *Verilog*. Il est possible de faire appel à *Verilog2SMV* pour construire le modèle concret en langage SMV. Un `Makefile` permet d'automatiser la construction et de donner le nombre de lignes de code.

D.4 Détails de la vérification

Conformément à notre hypothèse **H10**, notre définition de la sécurité est la suivante : il est impossible d'obtenir une différence entre les signaux observés et les valeurs attendues. Cette définition se traduit formellement par un invariant que nous vérifions sur le moniteur par une combinaison de *model-checking* et de preuve. L'expression D.3 fournit cette définition formelle dans le cas où les valeurs des signaux attendues sont concaténées dans les bits de poids faible des vecteurs de donnée contenus dans la ROM.

$$\begin{aligned}
&\textbf{always}(\ verifying \wedge \neg(\\
&\qquad (sw_araddr = rddata[13:0]) \\
&\quad \wedge (sw_arready = rddata[14]) \\
&\quad \wedge (trace_ctl = rddata[15]) \\
&\quad \wedge (trace_data = rddata[23:16]) \\
&\quad \wedge (decoded_address = rddata[55:24]) \\
&\quad) \\
&\quad \rightarrow reset \\
&\ )
\end{aligned} \tag{D.3}$$

Ressources : le programme `6_4_3_preuve_moniteur/proof.v` utilise Coq. Il permet de démontrer que la spécification représentée par l'expression D.3 est impliquée par des propriétés locales vérifiées sur le moniteur. Les propriétés locales sont converties à l'aide de notre programme *psl2coq*.
Un `Makefile` permet d'automatiser la conversion des propriétés locales et la preuve.

Notre définition de la sécurité implique cependant des obligations de preuves. En effet, notre définition est exprimée en fonction des prédicats $rddata$ et $verifying$. Des spécifications supplémentaires doivent donc être également vérifiées ; en particulier :

- lors des accès à la ROM, le vecteur $rddata$ doit correspondre à la donnée lue dans la ROM à l'index du front d'horloge auquel la vérification doit être réalisée.
- la phase de vérification doit être correctement détectée afin que le prédicat $verifying$ soit vérifié lors de l'exécution de l'algorithme de confiance. Ceci implique donc que les phases d'attente et de fin soient elles aussi correctement détectées.

D.4.1 Accès à la ROM

La première obligation de preuve impliquée par notre définition de la sécurité est que la valeur du vecteur lu dans la ROM corresponde au bon index du front d'horloge. Nous spécifions donc les variations du vecteur $rddata$ en fonction de l'entrée dans la phase de vérification. Cela aboutit à deux spécifications formelles :

- la première décrit que le vecteur $rddata$ est égal à la donnée située à l'index 0 de la ROM lors de l'entrée dans la phase de vérification.
- la seconde décrit que, pour toute égalité du $rddata$ avec la donnée située à l'index n dans la ROM durant la phase de vérification, alors nous avons une égalité avec la donnée située à l'index $n+1$ au prochain front d'horloge.

Les expressions D.4 et D.5 représentent ces spécifications exprimées en langage PSL.

$$\begin{aligned}
&\textbf{always}(\\
&\qquad \neg verifying \wedge \neg done \wedge \textbf{next}(verifying) \\
&\quad \rightarrow \\
&\qquad \textbf{next}(rddata = ROM(0)) \\
&\ )
\end{aligned} \tag{D.4}$$

$$\begin{aligned}&\textbf{forall } n \textbf{ in } \{0:1023\}: \\ &\quad \textbf{always}(\\ &\qquad verifying \wedge (rddata = ROM(n)) \\ &\quad \rightarrow \\ &\qquad \textbf{next}(rddata = ROM(n+1)) \\ &\quad)\end{aligned} \tag{D.5}$$

Afin de prouver ces spécifications, nous nous appuyons sur la modélisation du comportement de la ROM, décrite par l'expression D.1, et des propriétés locales vérifiées par *model-checking* sur le nouveau moniteur. Les propriétés locales décrivent le comportement du vecteur i en fonction de l'entrée dans la phase de vérification. Le point important à noter est que la donnée est transmise par la ROM une fois l'adresse positionnée sur le vecteur i et après un front d'horloge. Les propriétés locales décrivent donc que le vecteur i est positionné à l'adresse souhaitée à l'instant précédent la nécessité d'obtenir la donnée.

Concrètement, cela se traduit par trois propriétés que nous pouvons décrire comme suit :

— à l'instant précédent le début de la phase de vérification, l'adresse positionnée sur i doit toujours être nulle ;

— durant la phase de vérification, si l'adresse positionnée sur i est inférieure à END_ADDR, alors celle-ci est toujours incrémentée.

— à l'état initial du système, le nouveau moniteur ne se trouve pas en phase de vérification.

À partir de ces trois propriétés locales et de notre modèle de ROM, nous prouvons, à l'aide de l'assistant de preuve Coq, les spécifications représentées par les expressions D.4 et D.5.

Ressources : le programme `d_4_1_rom/proof.v` utilise Coq. Il permet de démontrer, à partir de l'axiome D.1, que les spécifications représentées par les expressions D.4 et D.5 sont impliquées par les propriétés locales vérifiées sur le nouveau moniteur. Les propriétés locales sont converties à l'aide de notre programme *psl2coq*. Un `Makefile` permet d'automatiser la conversion des propriétés locales et la preuve.

D.4.2 Phase de vérification

La seconde obligation de preuve, conséquence de notre définition de la sécurité, est de garantir que les phases d'attente, de vérification et de fin soient correctement détectées (présence du prédicat $verifying$ dans la spécification D.3). Nous vérifions cette correction en spécifiant la détection de chacune des phases séparément.

Phase d'attente : le nouveau moniteur indique qu'il est en phase d'attente lorsque les signaux $verifying$ et *done* sont baissés.

Détecter correctement la phase d'attente n'est pas un prérequis pour garantir la sécurité de l'exécution de l'algorithme de confiance. Cependant, c'est un prérequis pour pour garantir la *soundness* de notre nouveau moniteur : celui-ci ne doit pas transmettre un *reset* au microprocesseur lorsqu'aucune vérification n'est nécessaire.

Nous pouvons vérifier formellement la détection de la phase d'attente à l'aide des invariants D.6 et D.7. L'expression D.6 décrit que le moniteur reste en phase d'attente si le signal

$sw\_arready$ n'est pas levé. L'expression D.7 décrit que seule une levée de $sw\_arready$ peut mettre fin à la phase d'attente. Du *model-checking* seul est suffisant pour vérifier ces invariants.

$$\begin{aligned}&\textbf{always}(\\&\quad \neg verifying \wedge \neg done \wedge \textbf{next}(\neg sw\_arready)\\&\rightarrow\\&\quad \textbf{next}(\neg verifying \wedge \neg done)\\&)\end{aligned} \tag{D.6}$$

$$\begin{aligned}&\textbf{always}(\\&\quad \neg verifying \wedge \neg done\\&\rightarrow\\&\quad (\neg verifying \wedge \neg done)\ \textbf{until}\ sw\_arready\\&)\end{aligned} \tag{D.7}$$

Phase de vérification : le nouveau moniteur indique qu'il est en phase de vérification lorsque le signal $verifying$ est levé.

Détecter correctement la phase de vérification est un prérequis pour garantir la sécurité de l'exécution de l'algorithme de confiance. Celle-ci doit démarrer lorsque le signal $sw\_arready$ est levé et doit se terminer lorsque le vecteur i a atteint la valeur finale.

L'expression D.8 décrit que le moniteur passe en phase de vérification si le signal $sw\_arready$ est levé. L'expression D.9 décrit que seule une égalité entre i et END_ADDR peut mettre fin à la phase de vérification. Du *model-checking* seul est suffisant pour vérifier ces spécifications.

$$\begin{aligned}&\textbf{always}(\\&\quad \neg verifying \wedge \neg done \wedge \textbf{next}(sw\_arready)\\&\rightarrow\\&\quad \textbf{next}(verifying)\\&)\end{aligned} \tag{D.8}$$

$$\begin{aligned}&\textbf{always}(\\&\quad verifying\\&\rightarrow\\&\quad verifying\ \textbf{until}\ (i = END\_ADDR)\\&)\end{aligned} \tag{D.9}$$

Notons que ces spécifications garantissent que, en phase de vérification, le moniteur passe bien en phase de fin après une égalité entre i et END_ADDR. Une spécification supplémentaire est nécessaire pour garantir que celui-ci ne passe pas en phase de fin lors de ce même évènement s'il se trouve en phase d'attente. Cette spécification est exprimée par l'expression

D.10.

$$\begin{aligned}&\mathbf{always}(\\&\quad \neg verifying \wedge \neg done \wedge \mathbf{next}(i = END\_ADDR)\\&\rightarrow\\&\quad \mathbf{next}(\neg done)\\&\quad)\end{aligned} \tag{D.10}$$

Phase de fin : le nouveau moniteur indique qu'il est en phase de fin lorsque le signal *done* est levé.

Détecter correctement l'entrée dans la phase de fin n'est pas un prérequis pour garantir la sécurité de l'exécution de l'algorithme de confiance. En effet, l'exécution se termine à cet instant. Cependant, c'est un prérequis pour garantir la *soundness* de notre nouveau moniteur : celui-ci doit céder la sécurité à notre premier moniteur (décrit dans le chapitre 5) durant l'exécution de la fonction d'attestation. Nous pouvons donc vérifier des propriétés de *soundness* semblables à celles exprimées par les expressions D.6 et D.7.

La sécurité de notre nouveau moniteur dépend néanmoins de la détection d'une sortie de la phase de fin. En effet, une nouvelle phase de vérification, critique à la sécurité de l'exécution de l'algorithme de confiance, ne peut démarrer que si le nouveau moniteur ne se trouve pas en phase de fin. Nous vérifions donc que la levée du signal *deduced_PC_eq_CRmax* produise une sortie de la phase de fin. L'expression D.11 décrit formellement cette spécification.

$$\begin{aligned}&\mathbf{always}(\\&\quad done \wedge deduced\_PC\_eq\_CRmax\\&\rightarrow\\&\quad \mathbf{next}(\neg done)\\&\quad)\end{aligned} \tag{D.11}$$

Du *model-checking* permet de vérifier une propriété locale équivalente, seulement si le vecteur i n'atteint pas END_ADDR à l'état suivant où le transducteur lève *deduced_PC_eq_CRmax*. Nous utilisons cette propriété locale et l'axiome décrit par l'expression D.2 pour prouver la spécification telle que décrite par l'expression D.11.

Ressources : le programme `d_4_2_fin/proof.v` utilise Coq. Il permet de démontrer, à partir de l'axiome D.2, que la spécification représentée par l'expression D.11 est impliquée par des propriétés locales vérifiées sur le moniteur et sur le transducteur. Les propriétés locales sont converties à l'aide de notre programme *psl2coq*.
Un `Makefile` permet d'automatiser la conversion des propriétés locales et la preuve.

Phase interdite : les obligations de preuve décrivant la détection des différentes phases sont exprimées en fonction des signaux $verifying$ et *done*. Notons que si les deux signaux sont levés en même temps, cela ne représente aucune des trois phases dans lesquelles le nouveau moniteur peut se trouver. Une dernière vérification consiste donc à s'assurer que le nouveau moniteur ne lève jamais ces deux signaux en même temps. L'expression D.12 décrit formellement cette

spécification. Du *model-checking* seul est suffisant pour la vérifier.

$$\textbf{always}(\neg(verifying \wedge done)) \tag{D.12}$$

Ressources : le dossier `d_4_2_mc_moniteur/` contient la définition des 27 propriétés que nous avons identifiées pour vérifier la correction du nouveau moniteur. Cela inclut toutes les spécifications définies dans la section D.4 ainsi que toutes les propriétés locales nécessaires à l'établissement des preuves. Notre méthode de vérification formelle automatisée permet de s'assurer, par *model-checking*, que le nouveau moniteur vérifie ces propriétés. Un `Makefile` permet d'automatiser cette vérification.

D.5 Considérations sur la non-reproductibilité des signaux

Dans le cas où notre hypothèse **H10** est valide, nous implémentons un algorithme de confiance tel qu'il est impossible pour un adversaire de reproduire les signaux d'accès lus par le moniteur sans l'exécuter. Nous proposons une implémentation dans le chapitre 6, que nous avons évaluée au travers d'une étude de la sécurité.

Toutefois, il est probable que l'implémentation que nous proposons soit vulnérable à des attaques que nous n'avons pas considérées. Dans ce cas, une fois les vulnérabilités découvertes, nous pouvons faire évoluer l'architecture d'attestation de configuration du microprocesseur que nous proposons. L'objectif de cette section est de présenter des classes d'attaque que nous avons considérées mais n'avons pas réussi à mettre en œuvre et, en cas de vulnérabilités avérées, de proposer des contre-mesures afin d'y remédier.

Les classes d'attaques que nous avons considérées dans le chapitre 6 sont celles des attaques logicielles, où un adversaire développe un programme malveillant pour reproduire les signaux d'accès. Nous pouvons également considérer les attaques matérielles, où un adversaire modifie la configuration des périphériques afin d'obtenir un comportement différent pour l'exécution du même programme malveillant. Seulement le contrôleur AXI maître et l'interface *CoreSight* sont d'intérêt pour ces classes d'attaque car ce sont les périphériques dont notre nouveau moniteur compare les signaux avec des valeurs attendues.

D.5.1 Vecteur de donnée de *CoreSight*

Une classe d'attaque que nous n'avons pas réussi à mettre en œuvre relève des attaques matérielles sur l'interface *CoreSight*. Notre proposition d'extension matérielle considère que le vecteur de donnée fournit par *CoreSight* est large de 8 bits. Le nouveau moniteur vérifie une égalité entre ces 8 bits et leurs valeurs attendues. Or, rien n'indique qu'un adversaire ne peut modifier la configuration de *CoreSight* pour fournir un vecteur de donnée de taille différente. Il est possible de modifier la configuration pour transmettre un vecteur de 16 ou 32 bits [50].

Ainsi, la classe d'attaque en question consiste à configurer *CoreSight* pour fournir un vecteur de donnée plus grand, et adapter la configuration de la MMU et l'architecture du programme malveillant. L'objectif est que les 8 bits de poids faibles du vecteur de donnée correspondent à ceux attendus par le nouveau moniteur. Cela semble possible avec des valeurs d'adresses de destination astucieusement choisies, voir même en modifiant les bits de poids fort de celles-ci pour forcer *CoreSight* à transmettre plus de donnée. Le nombre de branchements indirects à réaliser diffère alors du nombre de branchements indirects présents dans l'algorithme

de confiance. Si celui-ci est inférieur, alors le nombre de registres nécessaire pour précharger les adresses de destination devient plus faible.

Nous n'avons pas réussi à confirmer ou infirmer la possibilité de reproduire les signaux via cette classe d'attaque. La difficulté réside dans le fait que *CoreSight* transmette plusieurs paquets en une fois lorsque sa FIFO contient suffisamment de données. Ainsi, modifier la taille du vecteur de sortie influe sur la variation de l'état de la FIFO au cours du temps. Du *model-checking* permettrait de vérifier si cette attaque peut ou non être réalisée, à condition de posséder un modèle du composant *CoreSight*. À ces fins, une étude nécessitant de l'ingénierie inverse est requise.

Si toutefois notre architecture est vulnérable à cette classe d'attaque, nous pouvons tout de même proposer une contremesure. La solution consiste simplement à **observer les bits de poids fort du vecteur de donnée de *CoreSight***. Notre architecture suppose que ceux-ci ne sont pas utilisés, donc leur valeur doit être fixe. Pour obtenir la sécurité, si une modification est observée, alors un *reset* est transmis au microprocesseur.

Notons que nous n'avons pas vérifié si *CoreSight* modifie ou non les valeurs des bits de poids fort durant une utilisation normale. C'est-à-dire même si ceux-ci ne sont pas utilisés. Une étude est nécessaire afin de s'assurer que le système reste *sound* : l'ajout d'une telle vérification ne doit pas empêcher l'algorithme de confiance de s'exécuter.

D.5.2 Fréquences d'horloges

Une seconde classe d'attaque que nous n'avons pas réussi à mettre en œuvre relève des attaques matérielles par *overclocking*. C'est-à-dire les attaques où la fréquence de fonctionnement des périphériques est modifiée (généralement augmentée). Cette classe d'attaque peut être envisagée dans le but de synchroniser les signaux fournis par *CoreSight* avec les accès sur le bus AXI, qui ralentissent le microprocesseur (comme montré dans la section 6.3.1).

La reconfiguration du contrôleur AXI ne semble cependant pas une classe d'attaque valable car celui-ci communique avec un contrôleur AXI esclave implémenté dans le FPGA. Or c'est le contrôleur AXI esclave qui décide de la fréquence de transfert de données en levant le signal $ARREADY$ lorsque celui-ci est prêt [60]. Une augmentation de la fréquence de fonctionnement du contrôleur AXI n'accélère donc pas les transactions.

Cependant, nous pouvons envisager d'augmenter la fréquence de fonctionnement de l'interface *CoreSight* afin d'accélérer la transmission. Le nouveau moniteur vérifie la correspondance des signaux avec leur valeur attendue à chaque front d'horloge. Cette horloge est externe au microprocesseur. Si la fréquence de fonctionnement de *CoreSight* est augmentée, alors certains octets du paquet transmis sont manqués par le nouveau moniteur. La classe d'attaque en question consiste donc à adapter la configuration de la MMU et l'architecture du programme malveillant afin que les paquets observés par le nouveau moniteur forment la même trace d'exécution que celle attendue.

Nous n'avons pas réussi à confirmer ou infirmer la possibilité de reproduire les signaux via cette classe d'attaque. La difficulté réside ici aussi dans le fait que *CoreSight* transmette plusieurs paquets en une fois lorsque sa FIFO contient suffisamment de données. Ainsi, modifier la taille des paquets influe sur la variation de l'état de la FIFO au cours du temps.

Si toutefois notre architecture est vulnérable à cette classe d'attaque, nous pouvons tout de même, ici aussi, proposer une contremesure. *CoreSight* transmet sur le TPIU une copie de son horloge interne divisée par deux. Cette copie est accessible via le signal $TRACE\_CLK$

et permet la synchronisation [50, 62]. Nous ne l'utilisons pas dans nos travaux car nous nous synchronisons sur une horloge externe. La solution consiste donc à **compter et comparer le nombre de fronts d'horloge fournis par TRACE_CLK et ceux fournis par notre horloge externe**. Tous les deux fronts d'horloge externe, le rapport entre les valeurs des deux compteurs doit être de 1/2. Pour obtenir la sécurité, si le rapport diffère de 1/2 à ces instants, alors un *reset* est transmis au microprocesseur.

Annexe E

Futurs travaux : détails techniques

Sommaire

Dans les chapitres 5 et 6, nous avons proposé un périphérique de confiance, permettant l'attestation à distance vérifiée formellement sur microprocesseur. Ces premières contributions s'appuient néanmoins sur trois hypothèses simplifiantes pour notre définition de la sécurité.

Également, nous avons proposé, dans le chapitre 4, des méthodes de vérification formelle de notre solution en nous appuyant sur une modélisation dans un formalisme haut niveau. Entre les modèles haut niveau et l'implémentation finale, nous avons émis des hypothèses de préservation sémantique pour les outils de transformation.

Dans cette annexe, nous proposons des extensions complémentaires pour notre périphérique de confiance et nous apportons des réflexions sur les hypothèses que nous avons émises. La section E.1 présente les extensions complémentaires que nous envisageons pour vérifier les hypothèses simplifiantes et la section E.2 apporte des considérations sur les transformations que nous réalisons.

E.1 Vérification des hypothèses simplifiantes

À ce stade de nos travaux, nous considérons toujours trois hypothèses simplifiantes :
- **H8** : l'adversaire n'accède pas au périphérique de programmation du FPGA.
- **H9** : les paquets *isync* et *branch* émis par *CoreSight* sont disponibles à l'état précédant l'exécution de l'instruction de destination.
- **H10** : nous pouvons écrire notre algorithme de confiance tel qu'il est impossible pour un adversaire de reproduire les signaux d'accès lus par le moniteur sans l'exécuter.

Nous considérons réaliste de pouvoir vérifier les hypothèses simplifiantes **H8** et **H9**. Dans cette section, nous décrivons des pistes d'extensions pouvant être vérifiées formellement.

E.1.1 Sécurisation du FPGA via le port ICAP

Afin de vérifier l'hypothèse simplifiante **H8**, il est possible de mettre en place une architecture matérielle qui interdit les accès aux FPGA depuis le microprocesseur. Cette interdiction

est valable à tout instant après la première programmation du FPGA, indépendamment de l'état dans lequel le microprocesseur se trouve.

Afin de décrire la solution que nous envisageons, nous devons tout d'abord introduire le fonctionnement des *SoC* Zynq-7000 concernant l'accès à la mémoire de configuration du FPGA [50] (*CRAM*). Un composant non-programmable, appelé *PL configuration module* et présent dans la partie FPGA du *SoC*, donne accès à cette mémoire. Le FPGA possède 3 ports d'accès permettant de communiquer avec le *PL configuration module* :

— le TAP (*Test Access Port*), un accès depuis le composant externe JTAG. Il est utilisé pour programmer la mémoire de configuration du FPGA depuis un accès physique au *SoC*, généralement durant le développement. C'est le port qui est utilisé par openOCD.

— le PCAP, un accès depuis le microprocesseur. Il est utilisé pour programmer la mémoire de configuration depuis un *bitstream*, durant l'exécution du FSBL. C'est le port qui peut être utilisé par un adversaire pour accéder au FPGA, dont nous souhaitons interdire l'utilisation.

— l'ICAP (*Internal Configuration Access Port*), un accès depuis le FPGA lui-même. Il est utilisé pour programmer la mémoire de configuration depuis un *bitstream* stocké dans une mémoire accessible depuis le FPGA. Une première programmation est nécessaire pour utiliser ce port et rendre la mémoire accessible.

La connexion entre chacun de ces trois 3 ports et le *PL configuration module* est mutuellement exclusive. C'est-à-dire qu'il est impossible d'utiliser plus d'un port à la fois pour communiquer avec le *PL configuration module*. Des multiplexeurs permettent la sélection du port en question. Le microprocesseur modifie la sélection en fonction de la configuration de registres dédiés.

Nous envisageons d'utiliser le port ICAP pour détecter un accès depuis le PCAP. Le port ICAP est utilisé pour réaliser une lecture du contenu du FPGA en permanence. Une erreur est reçue par l'ICAP en cas de perte d'accès au *PL configuration module*. **Un *reset* est alors immédiatement transmis au microprocesseur**. Notons que cette solution empêche également une reprogrammation légitime du FPGA depuis le bus JTAG, il est donc nécessaire de la désactiver lors de la phase de développement.

E.1.2 Verrou de l'exécution

L'hypothèse simplifiante **H9** sert à prouver que nous déduisons bien l'évolution du pointeur d'instruction et les signaux d'interruptions. Elle stipule que, lors d'un branchement indirect ou d'une levée d'exception, l'instruction de destination ne sera exécutée qu'après la réception des paquets. Cette hypothèse doit être vérifiée lors d'une entrée ou d'une sortie de PC dans CR, pour empêcher l'exécution d'instructions illégitimes avant le décodage des paquets. Nous devons considérer plusieurs cas de figure pour sa vérification. Nous considérons les cas suivants :

1. PC entre dans CR sans exécuter l'algorithme de confiance au préalable.
2. PC entre dans CR à la fin de l'algorithme de confiance ($PC = CR_{min}$).
3. PC sort de CR à la suite d'une exception durant la fonction d'attestation.
4. PC sort de CR à la fin de la fonction d'attestation ($PC = CR_{max}$).

Pour rappel, la fonction d'attestation se termine toujours par un branchement indirect ayant pour destination l'adresse CR_{max} (voir la section 5.5.1.3).

Les cas de figure 1 et 2 ne posent pas de problème de sécurité. En effet, notre solution d'attestation de configuration du microprocesseur (décrite dans le chapitre 6) assure une exé-

cution de l'algorithme de confiance jusqu'au décodage du paquet indiquant que PC atteint CR_{min}. Il n'est donc pas possible (selon l'hypothèse simplifiante **H10**) pour un adversaire de modifier le flot d'exécution durant cet intervalle de temps.

Les cas de figure 3 et 4, en revanche, introduisent un délai entre la sortie de CR par PC et le décodage du paquet qui l'indique. Notre moniteur matériel n'effectue un *reset* du microprocesseur qu'une fois le paquet décodé. Durant ce délai, un adversaire peut procéder à une exécution d'instructions illégitimes, comme par exemple un accès au secret, dont le résultat est sauvegardé avant le *reset* du microprocesseur.

L'hypothèse simplifiante **H9** doit donc être vérifiée dans les cas de figure 3 et 4. Il est donc necessaire de mettre en pause le flot d'exécution de la fonction d'attestation lors de l'émission du paquet par *CoreSight*, jusqu'au décodage du paquet transmis. Pour ce faire, nous envisageons une extension logicielle (au niveau de l'algorithme de confiance) et une extension matérielle (au niveau du moniteur).

E.1.2.1 Extension logicielle : interruption de la fonction d'attestation

Pour traiter le cas de figure 3, nous envisageons de configurer des *handlers* d'interruption pour mettre le flot d'exécution en attente active (*spin lock*). Ainsi, la valeur de PC reste fixe jusqu'au *reset* du microprocesseur, levé par le moniteur matériel, au décodage du paquet contenant les informations d'exception. En effet, ces *handlers* d'interruptions ne peuvent être reconfigurés que par exécution de logiciel. Or, l'adversaire ne peut précisément pas, à ce stade, exécuter de code sur le processeur autrement qu'en interrompant l'exécution de la fonction d'attestation. L'algorithme de confiance sera en charge de la configuration de ces *handlers*. Enfin, afin de préserver le comportement fonctionnel de la fonction attestée, un épilogue de la fonction d'attestation se chargera de restaurer la configuration originale des *handlers* d'exception.

E.1.2.2 Extension matérielle : fin d'exécution de la fonction d'attestation

Pour traiter le cas de figure 4, où PC sort de CR à la fin de l'exécution de la fonction d'attestation ($PC = CR_{max}$), nous envisageons une extension matérielle.

Notre moniteur matériel, décrit dans le chapitre 5, interdit les accès aux mémoires sensibles une fois le paquet décodé (indiquant que $PC = CR_{max}$). Comme décrit dans la section 5.5.1, et plus particulièrement sur la figure 5.4, l'instruction située à l'adresse de destination est un retour à la fonction appelante. Cette fonction appelante peut donc effectuer un accès aux mémoires sensibles avant le décodage du paquet et ce, sans avoir à interrompre la fonction d'attestation.

Pour pallier ce problème, nous envisageons de modifier l'architecture de l'esclave AXI-Lite contenant le code de la fonction d'attestation. Les 8 dernières instructions sont séparées dans une ROM dédiée. Cette ROM n'est pas considérée comme une mémoire sensible. Les 8 instructions constituent la destination du branchement indirect à CR_{max} : le retour à la fonction appelante et 7 instructions inutilisées. Le listing E.1 représente les instructions qui sont séparées dans la ROM dédiée.

```
// <- destination = CR_max
pop {fp, pc}  // retour a la fonction appelante
nop
nop
```

```
nop
nop
nop
nop
nop
```

Listing E.1 – Séparation du retour à la fonction appelante

En plus de cette séparation, nous envisageons d'ajouter une seconde ROM, de même taille, qui contient également 8 instructions. Celle-ci est accessible par le microprocesseur exactement à la même adresse que la première, c'est-à-dire à CR_{max}. C'est **le moniteur matériel** qui **décide, lors d'un *fetch*, de permettre au contrôleur AXI une lecture de la première ROM ou de la seconde**.

Dans cette seconde ROM, 8 instructions réalisent une boucle infinie. Par exemple, nous pouvons envisager 7 instructions inutilisées et un branchement à CR_{max}. Le listing E.2 représente les instructions qui sont placées dans la seconde ROM.

```
// <- destination = CR_max
nop
nop
nop
nop
nop
nop
nop
b CR_max  // pc = pc - 28
```

Listing E.2 – Boucle infinie à CR_{max}

Comme notre algorithme de confiance désactive l'utilisation des mémoires cache lors de l'exécution de la fonction d'attestation, un *fetch* à l'adresse CR_{max} sera toujours traduit par un accès sur le bus AXI. Un multiplexeur permet de sélectionner la première ou la seconde ROM lors d'un tel accès. Le branchement indirect à CR_{max}, exécuté à la fin de la fonction d'attestation, force *CoreSight* à émettre un paquet. Tant que le paquet n'a pas été transmis, décompressé et décodé, le multiplexeur sélectionne la seconde ROM, bloquant le microprocesseur dans une attente active. Une fois le paquet décodé, c'est-à-dire après que le moniteur interdise de nouveau les accès aux mémoires sensibles et autorise les levées d'exception, le multiplexeur sélectionne la première ROM et permet le retour à la fonction appelante.

E.2 Vérifications formelles à considérer

En admettant que l'ensemble des hypothèses simplifiantes soient vérifiées, notre périphérique de confiance permet l'établissement d'une racine de confiance statique sur microprocesseur. En effet, nous nous sommes appuyés sur des modèles de celui-ci pour prouver la sécurité de l'exécution de la fonction d'attestation et le maintien de la confidentialité du secret. Le modèle du périphérique de confiance est décrit en *Verilog* au niveau RTL et le modèle de la fonction d'attestation est décrit en C.

Afin de réaliser notre implémentation finale, nous avons cependant transformé ces modèles à l'aide d'outils dédiés. En effet, nous utilisons, pour l'implémentation, un ensemble d'outils de transformation (compilation, synthèse, etc.) de nos modèles dans des formats adaptés au *SoC* sur lequel nous travaillons (*elf*, *bitstream*, etc.). Nous utilisons également, pour la modélisation,

un ensemble d'outils de traduction de nos modèles (*Verilog*) et propriétés locales (PSL) dans des formalismes adaptés aux outils de *model-checking* et de preuve (SMV, Gallina).

Pour que notre preuve reste valide sur l'implémentation finale de notre périphérique de confiance, notre méthode d'implémentation et de modélisation doit être reconsidérée. Nous décrivons dans le chapitre 3 les méthodes d'implémentation et de modélisation que nous avons choisies. Nous y émettons des hypothèses sur la préservation sémantique des outils de traduction. Également, nous décrivons dans le chapitre 4 notre méthode de vérification formelle automatisée, qui permet des traductions automatiques pour les modèles et propriétés locales. La préservation sémantique n'y est pas abordée. Dans cette section, nous proposons des poursuites de travaux dans le but de s'affranchir des hypothèses de préservation sémantique lors des transformation et de vérifier formellement les traductions.

E.2.1 Implémentation

Dans la section 3.1, nous établissons des hypothèses sur la préservation sémantique des outils de transformation. En particulier :

— l'hypothèse de transformation **T1** suggère la préservation des propriétés de sécurité durant l'édition des liens par les outils de la suite `gcc`.
— l'hypothèse de transformation **T3** suggère la préservation de la sémantique du circuit durant le *place-and-route* et la génération du *bitstream* par *Vivado*.

Dans cette section, nous proposons des axes d'amélioration pour s'affranchir de ces hypothèses et renvoyons vers les travaux de recherche à l'état-de-l'art qui traitent du même sujet.

E.2.1.1 Implémentation du logiciel

Dans ce manuscrit, nous proposons de compiler avec `CompCert` la fonction d'attestation, située dans la région critique. En revanche, pour le reste du projet, FSBL et application, nous utilisons la bibliothèque logicielle dédiée au Zynq-7000 fournie par Xilinx [96]. Cette bibliothèque a été développée pour être compilée avec `gcc` et des modifications doivent être réalisées avant de pouvoir la compiler avec `CompCert`, qui ne supporte pas l'intégralité du langage C [22].

Or, la préservation sémantique garantie par `CompCert` n'est prouvée que dans le cas où l'ensemble du projet repose sur une compilation et une édition des liens réalisées par `CompCert` [24]. Un axe d'amélioration, **afin de profiter de la preuve établie par `CompCert`, est de l'utiliser pour la compilation et l'édition des liens de tout le projet**. Ainsi, nous pourrions nous affranchir de l'hypothèse de transformation **T1**. La difficulté de cette tâche est qu'elle requiert une migration de la bibliothèque logicielle dédiée au Zynq-7000 fournie par Xilinx.

À notre connaissance, il n'existe pas, aujourd'hui, de projet de recherche visant à rendre compatible la bibliothèque logicielle dédiée au Zynq-7000 avec `CompCert`.

E.2.1.2 Implémentation du matériel

Dans ce manuscrit, nous proposons de synthétiser nos extensions matérielles à l'aide de *yosys*, la suite libre de synthèse *Verilog*. La vérification formelle des propriétés locales est réalisée sur le résultat de cette synthèse. Cependant, entre la synthèse et l'implémentation finale, nous utilisons la suite logicielle *Vivado* pour réaliser le *place-and-route* et la génération

du *bitstream*. Or la préservation sémantique lors de ces opérations n'est pas prouvée par *Vivado*. De plus, cette suite logicielle est propriétaire et il n'est pas possible d'accéder aux sources afin de vérifier une telle préservation sémantique.

Un nouvel axe d'amélioration consiste à réaliser une suite logicielle équivalente à `CompCert` mais pour le matériel, c'est-à-dire **réaliser un ensemble d'outils de synthèse, *place-and-route* et génération du *bitstream* ciblant un FPGA particulier, dont la préservation sémantique est prouvée**. Ainsi, nous pourrions nous affranchir de l'hypothèse de transformation **T3**. La difficulté de cette tâche est qu'elle requiert une grande quantité de formalisation, développement et vérification.

Toutefois, des projets de recherche sur le sujet existent, malheureusement sans vérification formelle. Nous pouvons par exemple citer le projet `nextpnr` [66], qui a pour objectif de proposer une solution de *place-and-route* et génération du *bitstream* pour les FPGA commerciaux. Il propose aujourd'hui un ensemble d'outils libres adaptés à l'implémentation sur les FPGA *Latice*. Également, le projet `nextpnr-xilinx` [55] propose depuis 2021 une extension aux FPGA Xinlinx. Celui-ci s'appuie sur l'ingénierie inverse des formats de *bitstream* générés par *Vivado* au sein du projet *Project X-Ray* [55].

E.2.2 Modélisation

Dans la section 3.1, nous établissons également des hypothèses sur la préservation sémantique des outils de modélisation. En particulier, l'hypothèse de transformation **T2** suggère la préservation sémantique de la description *Verilog* vers un automate de Büchi déterministe décrit en SMV. De plus, dans le chapitre 4, nous proposons deux outils de traduction automatique pour les propriétés locales, *psl2spot* et *psl2coq*, que nous considérons corrects. Dans cette section, nous proposons des axes d'amélioration pour s'affranchir de ces hypothèses et renvoyons également vers les travaux de recherche à l'état-de-l'art qui traitent du même sujet.

E.2.2.1 Génération des automates de Büchi

Dans ce manuscrit, nous adoptons une approche ascendante : nous concevons nos modèles en langage *Verilog* et générons des automates de Büchi à l'aide de l'outil *Verilog2SMV*. La vérification formelle des propriétés locales est réalisée sur le résultat de cette génération. Or, la préservation sémantique lors de cette opération n'est pas prouvée par *Verilog2SMV*.

Durant nos travaux, nous avons de plus observé des résultats de génération qui semblent proposer une sémantique différente. Ceci est particulièrement le cas lors d'une affectation d'un vecteur par une concaténation de plusieurs entrées d'un module. Nous avons pu néanmoins contourner ce problème en affectant des *wire* intermédiaires aux entrées en question. Un exemple de génération montrant ce comportement est distribué sous licence libre et est rendu accessible publiquement [1].

> **Ressources :** le dossier `e_2_2_verilog2smv/` contient une définition de plusieurs modules dont l'automate de Büchi généré peut présenter une sémantique différente. L'outil *Verilog2SMV* peut être utilisé pour générer les automates et *NuSMV* peut être utilisé pour vérifier la sémantique. Le `Makefile` permet d'automatiser la génération des automates et la vérification.

Un axe d'amélioration consiste donc à **vérifier formellement la préservation sémantique lors de la génération d'un automate de Büchi depuis un modèle écrit en**

Verilog. Ainsi, nous pourrions nous affranchir de l'hypothèse de transformation **T2**. La difficulté de cette tâche réside dans le fait que *Verilog2SMV* fournit un résultat exprimé en fonction de la transformation réalisée par *yosys* et que la vérification formelle peut être lourde. De plus, le résultat de la génération dépend de nombreuses options d'optimisation durant la synthèse, ce qui alourdit d'autant plus la vérification formelle.

À notre connaissance, il n'existe pas d'outil de transformation de *Verilog* vers SMV dont la préservation sémantique est vérifiée formellement. D'autres outils que *Verilog2SMV* existent, comme par exemple EBMC [97], le successeur de VCEGAR [98], ou Cadence-SMV [99] mais nous n'avons pas trouvé de travaux qui traitent de préservation sémantique.

E.2.2.2 Expression des propriétés locales

Nos contributions incluent deux outils de traduction des propriétés locales vers des langages cibles. L'outil *psl2spot* traduit les expressions PSL en LTL dans un langage compréhensible par *Spot*. L'outil *psl2coq* traduit les expressions PSL dans un langage compréhensible par *Coq* sous réserve d'utiliser la bibliothèque qui l'accompagne. Nous n'avons pas vérifié formellement la préservation sémantique de ces traductions. Nous n'avons pas non plus évalué les potentielles différences de sémantique de la logique temporelle LTL entre les définitions fournies par *NuSMV*, *Spot* et la bibliothèque `Coq_LTL`.

Dans nos travaux, nous avons considéré axiomatique la correction des traductions que nous proposons. Un axe d'amélioration consiste donc à **réaliser une étude de ces traductions et proposer une vérification formelle de la préservation de la sémantique**. Notons également que les travaux VRASED nécessitent une traduction manuelle des propriétés LTL, exprimées en F⋆ et vérifiées par la communauté du projet HACL⋆ [36]. Leur traduction a pour cible un langage compréhensible par *Spot*. Un second axe d'amélioration consiste donc à **proposer une traduction automatique de propriétés LTL exprimées en F⋆ et vérifier formellement la préservation sémantique**. Notons que le langage cible d'une telle traduction peut correspondre à celui de *Spot* ou de *Coq*, selon le cas.

Dans ce domaine, nous pouvons citer les travaux de Tuerk et Schneider, qui apportent la preuve formelle de la préservation sémantique d'un sous-ensemble de PSL vers LTL [71]. Cette preuve est réalisée avec l'assistant de preuves HOL4, qui incorpore la sémantique de PSL [100]. Ces travaux sont compatibles avec la vérification par *model-checking* en SMV [101].

E.2.3 Correct par construction

Dans ce manuscrit, nous avons considéré l'attestation à distance sur microprocesseur vérifiée formellement comme une poursuite des travaux réalisés par VRASED [36]. À ce titre, nous avons opté pour la même stratégie en termes de modélisation et de vérification (que nous avons dû adapter pour passer à l'échelle). De ce fait, nos modèles sont décrits dans un langage de description matérielle (*Verilog*) et la vérification formelle est réalisée *a-posteriori*, par du *model-cehcking*.

Une autre stratégie peut être envisagée pour la vérification formelle de systèmes matériels, c'est celle du correct par construction. L'idée est de **décrire les modèles de manière abstraite sur lesquels les spécifications sont vérifiées puis, par raffinements successifs, d'aboutir au modèle concret qui peut être traduit en implémentation**. Cette méthode permet de s'affranchir de la vérification formelle par *model-checking*, où nous sommes

hautement dépendants de la mémoire de la machine utilisée pour la vérification. Seulement, la dernière étape : la traduction en implémentation, doit être prouvée pour préserver la sémantique.

Nous pouvons citer dans ce domaine la méthode B [102], dont le projet ComenC [103] permet dès aujourd'hui la traduction en langage C. Le projet *Forcoment* [104] a pour objectif de permettre la traduction en langage VHDL synthétisable. Ce projet ne semble pas encore abouti. Nous pouvons également citer *Fe-Si* (*FEatherweight SynthesIs*) [105], une implémentation d'un outil de synthèse formellement vérifié. La description du modèle est décrite en Coq et le langage cible est un sous-ensemble de VHDL ou de Verilog synthétisable.

Bibliographie

[1] J. Certes and B. Morgan, "Thesis materials," *https://gitlab.irit.fr/these-jonathan-certes-public/ressources/thesis-materials*, 2022. (Cité en pages 3, 53, 57, 60, 63, 67, 70, 72, 90, 97, 98, 99, 100, 102, 110, 112, 113, 123, 133, 134, 135, 152, 153, 154, 155, 167, 168, 170, 171, 173, 174, 175 et 196.)

[2] P. Kocher, J. Horn, A. Fogh, , D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks : Exploiting speculative execution," in *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019. (Cité en pages 6 et 12.)

[3] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown : Reading kernel memory from user space," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018. (Cité en pages 6 et 12.)

[4] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping bits in memory without accessing them : An experimental study of dram disturbance errors," in *ACM SIGARCH Computer Architecture News*, vol. 42, pp. 361–372, IEEE Press, 2014. (Cité en pages 6 et 12.)

[5] D. Gruss, M. Lipp, M. Schwarz, R. Fellner, C. Maurice, and S. Mangard, "KASLR is dead : Long live KASLR," in *Engineering Secure Software and Systems - 9th International Symposium, ESSoS 2017, Bonn, Germany, July 3-5, 2017, Proceedings* (E. Bodden, M. Payer, and E. Athanasopoulos, eds.), vol. 10379 of *Lecture Notes in Computer Science*, pp. 161–176, Springer, 2017. (Cité en pages 6 et 12.)

[6] I. Studnia, E. Alata, Y. Deswarte, M. Kaâniche, and V. Nicomette, "Survey of Security Problems in Cloud Computing Virtual Machines," in *Computer and Electronics Security Applications Rendez-vous (C&ESAR 2012). Cloud and security :threat or opportunity*, (Rennes, France), pp. p. 61–74, Nov. 2012. (Cité en page 7.)

[7] P. Ferrie, "Attacks on more virtual machine emulators," in *Symantec Technology Exchange*, 2007. (Cité en page 7.)

[8] R. Wojtczuk, "Subverting the xen hypervisor," in *Black Hat USA*, Sept. 2008. (Cité en pages 7 et 12.)

[9] K. Kortchinsky, "Cloudburst : A vmware guest to host escape story," in *Black Hat USA*, Sept. 2009. (Cité en pages 7 et 12.)

[10] N. Elhage, "Virtunoid : Breaking out of kvm," in *Black Hat USA*, Aug. 2011. (Cité en pages 7 et 12.)

[11] J.-C. Laprie, J. Arlat, J. Blanquart, A. Costes, Y. Crouzet, Y. Deswarte, J. Fabre, H. Guillermain, M. Kaâniche, K. Kanoun, *et al.*, "Guide de la sûreté de fonctionnement," *Toulouse : Cépaduès*, 1995. (Cité en page 7.)

[12] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE transactions on dependable and secure computing*, vol. 1, no. 1, pp. 11–33, 2004. (Cité en page 7.)

[13] D. Powell, R. Stroud, *et al.*, "Malicious-and accidental-fault tolerance for internet applications-conceptual model and architecture," 2001. (Cité en pages 9, 10 et 11.)

[14] L. Duflot, D. Etiemble, and O. Grumelard, "Using cpu system management mode to circumvent operating system security functions," *CanSecWest/core06*, 2006. (Cité en page 10.)

[15] Y.-A. Perez, L. Duflot, O. Levillain, and G. Valadon, "Quelques éléments en matière de sécurité des cartes réseau," in *Actes du 8ème symposium sur la sécurité des technologies de l'information et des communications (SSTIC)*, 2010. (Cité en page 10.)

[16] Y. Deswarte, L. Blain, and J. C. Fabre, "Intrusion tolerance in distributed computing systems," in *Research in Security and Privacy, 1991. Proceedings., 1991 IEEE Computer Society Symposium on*, pp. 110–121, May 1991. (Cité en page 11.)

[17] M. A. Wahab, P. Cotret, M. N. Allah, G. Hiet, V. Lapotre, and G. Gogniat, "Armhex : A hardware extension for DIFT on arm-based socs," in *27th International Conference on Field Programmable Logic and Applications, FPL 2017, Ghent, Belgium, September 4-8, 2017* (M. D. Santambrogio, D. Göhringer, D. Stroobandt, N. Mentens, and J. Nurmi, eds.), pp. 1–7, IEEE, 2017. (Cité en page 13.)

[18] Y. Lee, I. Heo, D. Hwang, K. Kim, and Y. Paek, "Towards a practical solution to detect code reuse attacks on ARM mobile devices," in *Proceedings of the Fourth Workshop on Hardware and Architectural Support for Security and Privacy, HASP@ISCA 2015, Portland, OR, USA, June 14, 2015* (R. B. Lee, W. Shi, and J. Szefer, eds.), pp. 3 :1–3 :8, ACM, 2015. (Cité en page 13.)

[19] D. Arsenault, A. Sood, and Y. Huang, "Secure, resilient computing clusters : Self-cleansing intrusion tolerance with hardware enforced security (SCIT/HES)," in *Proceedings of the The Second International Conference on Availability, Reliability and Security, ARES 2007, The International Dependability Conference - Bridging Theory and Practice, April 10-13 2007, Vienna, Austria*, pp. 343–350, IEEE Computer Society, 2007. (Cité en page 13.)

[20] D. Ahman, C. Hritcu, K. Maillard, G. Martínez, G. D. Plotkin, J. Protzenko, A. Rastogi, and N. Swamy, "Dijkstra monads for free," in *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017* (G. Castagna and A. D. Gordon, eds.), pp. 515–529, ACM, 2017. (Cité en page 13.)

[21] J. Protzenko, J. K. Zinzindohoué, A. Rastogi, T. Ramananandro, P. Wang, S. Z. Béguelin, A. Delignat-Lavaud, C. Hritcu, K. Bhargavan, C. Fournet, and N. Swamy, "Verified low-level programming embedded in F," *Proc. ACM Program. Lang.*, vol. 1, no. ICFP, pp. 17 :1–17 :29, 2017. (Cité en page 14.)

[22] X. Leroy, "A formally verified compiler back-end," *Journal of Automated Reasoning*, vol. 43, no. 4, pp. 363–446, 2009. (Cité en pages 14, 42 et 195.)

[23] J. K. Zinzindohoué, K. Bhargavan, J. Protzenko, and B. Beurdouche, "Hacl* : A verified modern cryptographic library," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017* (B. M. Thuraisingham, D. Evans, T. Malkin, and D. Xu, eds.), pp. 1789–1806, ACM, 2017. (Cité en pages 14, 17, 22, 42, 78 et 82.)

[24] X. Leroy, "The CompCert C compiler," *https://compcert.org/compcert-C.html*, 2021. (Cité en pages 14, 42 et 195.)

[25] E. M. Clarke and E. A. Emerson, "Design and synthesis of synchronization skeletons using branching-time temporal logic," in *Logics of Programs, Workshop, Yorktown Heights, New York, USA, May 1981* (D. Kozen, ed.), vol. 131 of *Lecture Notes in Computer Science*, pp. 52–71, Springer, 1981. (Cité en pages 14 et 45.)

[26] E. M. Clarke, R. P. Kurshan, and H. Veith, "The localization reduction and counterexample-guided abstraction refinement," in *Time for Verification, Essays in Memory of Amir Pnueli* (Z. Manna and D. A. Peled, eds.), Springer, 2010. (Cité en pages 14 et 19.)

[27] A. Pnueli, "The temporal logic of programs," in *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*, pp. 46–57, IEEE Computer Society, 1977. (Cité en pages 14 et 45.)

[28] S. Berezin, S. V. A. Campos, and E. M. Clarke, "Compositional reasoning in model checking," in *Compositionality : The Significant Difference, International Symposium, COMPOS'97, Bad Malente, Germany. Revised Lectures* (W. P. de Roever, H. Langmaack, and A. Pnueli, eds.), Springer, 1997. (Cité en pages 14, 19 et 59.)

[29] O. collectif Coordination Philippe Schnoebelen, *Vérification de logiciels : techniques et outils du model-checking.* Vuibert informatique., Paris : Vuibert, 05 1999. (Cité en pages 14, 138 et 140.)

[30] G. Coker, J. D. Guttman, P. Loscocco, A. L. Herzog, J. K. Millen, B. O'Hanlon, J. D. Ramsdell, A. Segall, J. Sheehy, and B. T. Sniffen, "Principles of remote attestation," *Int. J. Inf. Sec.*, vol. 10, no. 2, pp. 63–81, 2011. (Cité en page 15.)

[31] B. Morgan, E. Alata, V. Nicomette, M. Kaaniche, and G. Averlant, "Design and implementation of a hardware assisted security architecture for software integrity monitoring," pp. 189–198, 11 2015. (Cité en page 16.)

[32] A. Ghosh, A. Sapello, A. Poylisher, C. J. Chiang, A. Kubota, and T. Matsunaka, "On the feasibility of deploying software attestation in cloud environments," in *2014 IEEE 7th International Conference on Cloud Computing*, pp. 128–135, 2014. (Cité en page 16.)

[33] A. Seshadri, M. Luk, A. Perrig, L. van Doom, and P. K. Khosla, "Pioneer : Verifying code integrity and enforcing untampered code execution on legacy systems," in *Malware Detection* (M. Christodorescu, S. Jha, D. Maughan, D. Song, and C. Wang, eds.), vol. 27 of *Advances in Information Security*, pp. 253–289, Springer, 2007. (Cité en pages 16, 108 et 126.)

[34] X. Kovah, C. Kallenberg, C. Weathers, A. Herzog, M. Albin, and J. Butterworth, "New results for timing-based attestation," in *IEEE Symposium on Security and Privacy, SP 2012, 21-23 May 2012, San Francisco, California, USA*, pp. 239–253, IEEE Computer Society, 2012. (Cité en pages 16, 108, 114, 123 et 126.)

[35] K. Eldefrawy, G. Tsudik, A. Francillon, and D. Perito, "Smart : Secure and minimal architecture for (establishing dynamic) root of trust.," in *NDSS*, vol. 12, pp. 1–15, 2012. (Cité en pages 16, 17, 108 et 126.)

[36] I. D. O. Nunes, K. Eldefrawy, N. Rattanavipanon, M. Steiner, and G. Tsudik, "VRASED : A verified hardware/software co-design for remote attestation," in *28th USENIX*

Security Symposium (USENIX Security 19), (Santa Clara, CA), pp. 1429–1446, USENIX Association, Aug. 2019. (Cité en pages 16, 17, 22, 24, 25, 42, 43, 47, 48, 52, 76, 77, 78, 79, 82, 83, 84, 101, 102, 108, 126 et 197.)

[37] H. Krawczyk, M. Bellare, and R. Canetti, "Hmac : Keyed-hashing for message authentication," 1997. (Cité en pages 16, 22 et 78.)

[38] C. Wang, G. D. Hachtel, and F. Somenzi, *Abstraction Refinement for Large Scale Model Checking*. Series on Integrated Circuits and Systems, Springer, 2006. (Cité en pages 18 et 20.)

[39] R. Milner, "An algebraic definition of simulation between programs," in *Proceedings of the 2nd International Joint Conference on Artificial Intelligence. London, UK, September 1-3, 1971* (D. C. Cooper, ed.), pp. 481–489, William Kaufmann, 1971. (Cité en page 19.)

[40] D. L. Dill, A. J. Hu, and H. Wong-Toi, "Checking for language inclusion using simulation preorders," in *Computer Aided Verification, 3rd International Workshop, CAV '91, Aalborg, Denmark, July, 1-4, 1991, Proceedings* (K. G. Larsen and A. Skou, eds.), vol. 575 of *Lecture Notes in Computer Science*, pp. 255–265, Springer, 1991. (Cité en page 20.)

[41] S. Bensalem, A. Bouajjani, C. Loiseaux, and J. Sifakis, "Property preserving simulations," in *Computer Aided Verification, Fourth International Workshop, CAV '92, Montreal, Canada, Proceedings* (G. von Bochmann and D. K. Probst, eds.), Springer, 1992. (Cité en pages 20 et 59.)

[42] H. Jain, D. Kroening, N. Sharygina, and E. M. Clarke, "Word-level predicate-abstraction and refinement techniques for verifying RTL verilog," *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 2008. (Cité en pages 21, 53 et 54.)

[43] Y. Ho, P. Chauhan, P. Roy, A. Mishchenko, and R. K. Brayton, "Efficient uninterpreted function abstraction and refinement for word-level model checking," in *2016 Formal Methods in Computer-Aided Design, FMCAD, Mountain View, CA, USA* (R. Piskac and M. Talupur, eds.), IEEE, 2016. (Cité en pages 21, 53 et 54.)

[44] Z. S. Andraus and K. A. Sakallah, "Automatic abstraction and verification of verilog models," in *Proceedings of the 41th Design Automation Conference, DAC 2004, San Diego, CA, USA* (S. Malik, L. Fix, and A. B. Kahng, eds.), ACM, 2004. (Cité en pages 21, 53 et 54.)

[45] A. Irfan, A. Cimatti, A. Griggio, M. Roveri, and R. Sebastiani, "Verilog2smv : A tool for word-level verification," in *2016 Design, Automation & Test in Europe Conference & Exhibition, DATE 2016, Dresden, Germany, March 14-18, 2016* (L. F. andJürgen Teich, ed.), pp. 1156–1159, IEEE, 2016. (Cité en pages 21, 24, 58, 59 et 60.)

[46] I. C. Society, *1364-2001 IEEE Standard Verilog Hardware Description Language*. IEEE, 2001. (Cité en pages 21, 52, 58 et 59.)

[47] C. Wolf, "Yosys open synthesis suite." `http://www.clifford.at/yosys/`. (Cité en pages 21, 32, 43 et 72.)

[48] A. Duret-Lutz and D. Poitrenaud, "SPOT : an extensible model checking library using transition-based generalized büchi automata," in *12th International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS 2004), 4-8 October 2004, Vollendam, The Netherlands* (D. DeGroot, P. G. Har-

rison, H. A. G. Wijshoff, and Z. Segall, eds.), pp. 76–83, IEEE Computer Society, 2004. (Cité en pages 24, 47, 48 et 140.)

[49] *ARM Architecture Reference Manual - ARMv7-A and ARMv7-R edition*, no. ARM DDI 0406C.c ID051414, 2014. (Cité en pages 25, 35, 110, 115, 116 et 120.)

[50] *Zynq-7000 AP SoC Technical Reference Manual*, no. UG585 (v1.12.1), 2017. (Cité en pages 25, 33, 34, 37, 90, 103, 108, 110, 121, 187, 189 et 192.)

[51] J. Zhao, B. Korpan, A. Gonzalez, and K. Asanovic, "Sonicboom : The 3rd generation berkeley out-of-order machine," May 2020. (Cité en page 25.)

[52] C. Wolf, "Picorv32 - a size-optimized risc-v cpu." `https://github.com/YosysHQ/picorv32`, 2020. (Cité en page 25.)

[53] C. Domas, "Breaking the x86 isa," in *Black Hat USA*, 2017. (Cité en page 26.)

[54] *Zynq-7000 SoC Product Selection Guide*, no. XMP097 (v1.3.2), 2019. (Cité en page 31.)

[55] f4pga, "prjxray." `https://github.com/f4pga/prjxray/`, 2022. (Cité en pages 32 et 196.)

[56] *Vivado Design Suite Tcl Command Reference Guide*, no. UG835 (v2019.1), 2019. (Cité en page 32.)

[57] *Vivado Design Suite 7 Series FPGA and Zynq-7000 SoC Libraries Guide*, no. UG953 (v2022.1), 2022. (Cité en pages 33 et 52.)

[58] *Vitis Unified Software Platform Documentation : Embedded Software Development*, no. UG1400 (v2019.2), 2020. (Cité en page 34.)

[59] *ARM Coresight PFT architecture specification - PFTv1.0 and PFTv1.1*, no. ARM IHI 0035B ID060811, 2008-2011. (Cité en pages 35, 37, 38, 53, 61, 62, 89, 97, 99, 100, 165, 166, 168, 173 et 174.)

[60] *AMBA AXI and ACE Protocol Specification*, no. ARM IHI 0022D ID102711, 2003-2011. (Cité en pages 35, 36, 81, 110 et 188.)

[61] *CoreSight PTM-A9 Technical Reference Manual - Revision : r1p0*, no. ARM DDI 0401C ID073011, 2008-2011. (Cité en pages 37 et 113.)

[62] *CoreSight TPIU-Lite Technical Reference Manual - Revision : r0p0*, no. ARM DDI 0317A, 2006. (Cité en pages 38, 89, 112 et 189.)

[63] K. Bhargavan, B. Bond, A. Delignat-Lavaud, C. Fournet, C. Hawblitzel, C. Hritcu, S. Ishtiaq, M. Kohlweiss, K. R. M. Leino, J. R. Lorch, K. Maillard, J. Pan, B. Parno, J. Protzenko, T. Ramananandro, A. Rane, A. Rastogi, N. Swamy, L. Thompson, P. Wang, S. Z. Béguelin, and J. K. Zinzindohoue, "Everest : Towards a verified, drop-in replacement of HTTPS," in *2nd Summit on Advances in Programming Languages, SNAPL 2017, May 7-10, 2017, Asilomar, CA, USA* (B. S. Lerner, R. Bodík, and S. Krishnamurthi, eds.), vol. 71 of *LIPIcs*, pp. 1 :1–1 :12, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017. (Cité en page 42.)

[64] *Vitis Unified Software Platform Documentation - Embedded Software Development*, no. UG1400 (v2019.2), January 2020. (Cité en page 42.)

[65] A. Irfan, A. Cimatti, A. Griggio, M. Roveri, and R. Sebastiani, "Verilog2smv : A tool for word-level verification," in *2016 Design, Automation & Test in Europe Conference & Exhibition, DATE 2016, Dresden, Germany, March 14-18, 2016*, pp. 1156–1159, 2016. (Cité en page 43.)

[66] D. Shah, E. Hung, C. Wolf, S. Bazanski, D. Gisselquist, and M. Milanovic, "Yosys+nextpnr : An open source framework from verilog to bitstream for commercial fpgas," in *27th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM 2019, San Diego, CA, USA, April 28 - May 1, 2019*, pp. 1–4, IEEE, 2019. (Cité en pages 43 et 196.)

[67] gatecat, "nextpnr-xilinx." `https://github.com/gatecat/nextpnr-xilinx/`, 2022. (Cité en page 43.)

[68] M. Y. Vardi, "An automata-theoretic approach to linear temporal logic," in *Logics for Concurrency - Structure versus Automata (8th Banff Higher Order Workshop, Banff, Canada, August 27 - September 3, 1995, Proceedings)* (F. Moller and G. M. Birtwistle, eds.), vol. 1043 of *Lecture Notes in Computer Science*, pp. 238–266, Springer, 1995. (Cité en pages 44, 139 et 140.)

[69] K. Y. Rozier, "Linear temporal logic symbolic model checking," *Comput. Sci. Rev.*, vol. 5, no. 2, pp. 163–203, 2011. (Cité en page 45.)

[70] I. C. Society, *1850-2010 IEEE Standard for Property Specification Language (PSL)*. IEEE, 2010. (Cité en page 45.)

[71] T. Tuerk and K. Schneider, "From PSL to LTL : A formal validation in HOL," in *Theorem Proving in Higher Order Logics, 18th International Conference, TPHOLs 2005, Oxford, UK, Proceedings* (J. Hurd and T. F. Melham, eds.), Springer, 2005. (Cité en pages 45, 54 et 197.)

[72] C. Eisner and D. Fisman, *A Practical Introduction to PSL*. Springer, 2006. (Cité en page 45.)

[73] C. Baier and J. Katoen, *Principles of model checking*. MIT Press, 2008. (Cité en page 46.)

[74] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella, "Nusmv version 2 : An opensource tool for symbolic model checking," in *Proc. International Conference on Computer-Aided Verification (CAV)*, (Copenhagen, Denmark), Springer, July 2002. (Cité en pages 46, 58 et 140.)

[75] *NuSMV 2.6 User Manual*, FBK-irst - Via Sommarive 18, 38055 Povo (Trento) – Italy, 2015. (Cité en pages 46, 47, 55, 140 et 155.)

[76] Z. Manna and A. Pnueli, *The temporal logic of reactive and concurrent systems - specification*. Springer, 1992. (Cité en page 47.)

[77] A. Biere, "Picosat essentials," *J. Satisf. Boolean Model. Comput.*, vol. 4, no. 2-4, pp. 75–97, 2008. (Cité en page 48.)

[78] S. Baarir and A. Duret-Lutz, "Mechanizing the minimization of deterministic generalized büchi automata," in *Formal Techniques for Distributed Objects, Components, and Systems - 34th IFIP WG 6.1 International Conference, FORTE 2014, Held as Part of the 9th International Federated Conference on Distributed Computing Techniques, DisCoTec 2014, Berlin, Germany, June 3-5, 2014. Proceedings* (E. Ábrahám and C. Palamidessi, eds.), vol. 8461 of *Lecture Notes in Computer Science*, pp. 266–283, Springer, 2014. (Cité en page 48.)

[79] S. Baarir and A. Duret-Lutz, "Sat-based minimization of deterministic \omega - automata," in *Logic for Programming, Artificial Intelligence, and Reasoning - 20th International Conference, LPAR-20 2015, Suva, Fiji, November 24-28, 2015, Proceedings*

(M. Davis, A. Fehnker, A. McIver, and A. Voronkov, eds.), vol. 9450 of *Lecture Notes in Computer Science*, pp. 79–87, Springer, 2015. (Cité en page 48.)

[80] Y. Bertot and P. Castéran, *Interactive theorem proving and program development : Coq'Art : the calculus of inductive constructions*. Springer Science & Business Media, 2013. (Cité en page 49.)

[81] J. Certes and B. Morgan, "An Automated Framework Towards Widespread Formal Verification of Complex Hardware Designs," in *11th European Congress Embedded Real Time Systems (ERTS2022)*, (Toulouse, France), June 2022. (Cité en page 52.)

[82] W. Khan, M. Kamran, S. R. Naqvi, F. A. Khan, A. S. Alghamdi, and E. Alsolami, "Formal verification of hardware components in critical systems," *Wireless Communications and Mobile Computing*, 2020. (Cité en page 52.)

[83] J. Hogan, "Hogan on cadence, mentor, onespin, real intent, synopsys formal." `https://www.deepchip.com/items/0558-05.html`, 2016. (Cité en page 52.)

[84] V. Taraate, *Advanced HDL Synthesis and SOC Prototyping : RTL Design Using Verilog*. Springer Singapore, 1st ed. ed., 2019. (Cité en page 52.)

[85] S. Busard and C. Pecheur, "Pynusmv : Nusmv as a python library," vol. 7871 of *LNCS*, pp. 453–458, Springer-Verlag, 2013. (Cité en page 55.)

[86] J. Certes and B. Morgan, "Attestation à distance de microprocesseurs vérifiée formellement," in *Rendez-vous de la Recherche et de l'Enseignement de la Sécurité des Systèmes d'Information (RESSI 2020)*, (En ligne), Dec. 2020. (Cité en page 76.)

[87] J. Certes and B. Morgan, "Remote attestation of bare-metal microprocessor software : A formally verified security monitor," in *Database and Expert Systems Applications - DEXA 2021 Workshops - BIOKDD, IWCFS, MLKgraphs, AI-CARES, ProTime, AISys 2021, Virtual Event, September 27-30, 2021, Proceedings* (G. Kotsis, A. M. Tjoa, I. Khalil, B. Moser, A. Mashkoor, J. Sametinger, A. Fensel, J. M. Gil, L. Fischer, G. Czech, F. Sobieczky, and S. Khan, eds.), vol. 1479 of *Communications in Computer and Information Science*, pp. 42–51, Springer, 2021. (Cité en page 76.)

[88] Y. Yarom and K. Falkner, "FLUSH+RELOAD : A high resolution, low noise, L3 cache side-channel attack," in *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014* (K. Fu and J. Jung, eds.), pp. 719–732, USENIX Association, 2014. (Cité en page 88.)

[89] J. Certes and B. Morgan, "Attaque et sécurisation d'un schéma d'attestation à distance vérifié formellement," in *Symposium sur la sécurité des technologies de l'information et des communications*, (Rennes, France), June 2022. (Cité en page 106.)

[90] I. C. Society, *1149.1-2013 IEEE Standard for Test Access Port and Boundary-Scan Architecture*. IEEE, 2013. (Cité en page 133.)

[91] *Open On-Chip Debugger : OpenOCD User's Guide*, no. for release 0.11.0+dev, 28 June, 2021. (Cité en page 134.)

[92] *Representation of events in nerve nets and finite automata*, Princeton Univ. Press, Princeton, N. J., 1996. (Cité en page 138.)

[93] S. Bardin, "Introduction au model checking." `http://sebastien.bardin.free.fr/mc-saclay-poly.pdf`, 2022. (Cité en page 141.)

[94] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang, "Symbolic model checking : 10^20 states and beyond," in *Proceedings of the Fifth Annual Symposium on Logic in Computer Science (LICS '90), Philadelphia, Pennsylvania, USA, June 4-7, 1990*, pp. 428–439, IEEE Computer Society, 1990. (Cité en page 141.)

[95] S. Williams, "Icarus verilog." `http://iverilog.icarus.com/`. (Cité en page 152.)

[96] Xilinx, "Xilinx Embedded Software (embeddedsw) Development," *https://github.com/Xilinx/embeddedsw*, 2014. (Cité en page 195.)

[97] D. Kroening and M. Purandare, "Ebmc - a model checker for verilog designs," *http://www.cprover.org/ebmc/*, 2017. (Cité en page 197.)

[98] H. Jain, D. Kroening, N. Sharygina, and E. M. Clarke, "VCEGAR : verilog counterexample guided abstraction refinement," in *Tools and Algorithms for the Construction and Analysis of Systems, 13th International Conference, TACAS 2007, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2007 Braga, Portugal, March 24 - April 1, 2007, Proceedings* (O. Grumberg and M. Huth, eds.), vol. 4424 of *Lecture Notes in Computer Science*, pp. 583–586, Springer, 2007. (Cité en page 197.)

[99] K. McMillan, "The cadence smv model checker," *https://mcmil.net/smv.html*, 2000. (Cité en page 197.)

[100] M. Gordon, "Psl semantics in higher order logic," *Workshop on Designing Correct Circuits (DCC)*, 2004. (Cité en page 197.)

[101] T. Tuerk, K. Schneider, and M. Gordon, "Model checking PSL using HOL and SMV," in *Hardware and Software, Verification and Testing, Second International Haifa Verification Conference, HVC 2006, Haifa, Israel, October 23-26, 2006. Revised Selected Papers* (E. Bin, A. Ziv, and S. Ur, eds.), vol. 4383 of *Lecture Notes in Computer Science*, pp. 1–15, Springer, 2006. (Cité en page 197.)

[102] J. Abrial, *The B-book - assigning programs to meanings*. Cambridge University Press, 1996. (Cité en page 198.)

[103] C. S. Engineering, "Comenc - b0 implementation translation into c language." `http://www.comenc.eu/`, 2022. (Cité en page 198.)

[104] C. S. Engineering, "Forcoment project : Event-b for the synthesizable vhdl." `https://www.clearsy.com/en/references/forcoment/`, 2022. (Cité en page 198.)

[105] T. Braibant and A. Chlipala, "Formal verification of hardware synthesis," in *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings* (N. Sharygina and H. Veith, eds.), vol. 8044 of *Lecture Notes in Computer Science*, pp. 213–228, Springer, 2013. (Cité en page 198.)

Résumé

Dans ce manuscrit, nous considérons comme problématique la sécurisation de l'exécution d'un algorithme dans un environnement complexe. Nous partons du principe que nous ne pouvons pas vérifier qu'à tout instant, cette exécution est sécurisée, même dans le cadre d'un adversaire distant. Néanmoins, il est possible de détecter une intrusion, une compromission de cet algorithme, à un instant donné et d'établir une racine de confiance dynamique. Pour cela, il existe le protocole d'attestation à distance. Sécuriser l'exécution du protocole nécessite d'établir une racine de confiance statique, vis-à-vis des capacité d'un adversaire. Cette racine de confiance statique, lorsqu'elle est accompagnée d'une vérification formelle, a jusqu'à présent été établie sur des microcontrôleurs simples, tels que la famille des MSP430. L'obtenir sur ces systèmes nécessite des composants matériels dédiés, suffisamment simples pour que la vérification formelle reste possible.

Afin de permettre la vérification formelle de système matériels plus complexes, nous proposons une approche de vérification formelle automatique et industrialisable. L'objectif de cet approche est de minimiser l'effort d'abstraction et ainsi réduire l'impact de l'explosion combinatoire.

Ensuite, en nous appuyant sur les architectures des *SoC* modernes, nous envisageons d'étendre le domaine d'application de l'attestation à distance vérifiée formellement. Notre cas d'étude vise à établir une racine de confiance statique sur un microprocesseur pris sur étagère : le ARM Cortex-A9. Pour cela, nous devons sécuriser l'exécution d'une fonction d'attestation et maintenir la confidentialité d'un secret. Nous proposons une définition de la sécurité sur cette famille de systèmes, en tenant compte de l'augmentation de la surface d'attaque et des capacités d'un adversaire privilégié. Nous décomposons ces capacités en des problématiques décorrélées, que nous pouvons traiter indépendamment pour permettre la vérification formelle.

Nous réalisons une implémentation, que nous confrontons à notre définition de la sécurité et apportons la preuve que celle-ci est vérifiée. La solution que nous proposons ne nécessite pas de modification matérielle, mais simplement d'une extension sous forme d'un périphérique de confiance, implémenté dans un circuit de logique programmable.

Considérer les architectures propriétaires apporte cependant une problématique : sans accès aux sources, nous ne pouvons pas modéliser formellement leur comportement. Également, vis-à-vis de notre modèle de menace fort, nous ne considérons pas que le microprocesseur constitue un environnement sûr. Nous proposons donc de reconstruire un mécanisme de gestion des privilèges externe à celui-ci, dans la partie matérielle de notre périphérique de confiance. Ce mécanisme s'appuie sur des hypothèses émises vis-à-vis du comportement du microprocesseur, que nous avons validées par une étude empirique pour pouvoir y accorder un fort degré de confiance.

Mots clés : Sécurité, Attestation à distance, Vérification formelle, FPGA, *CoreSight*

Abstract

In this thesis, we deal with how to secure the execution of an algorithm in a complex environment. We consider that it is impossible to verify that this execution is secure at every instant, even if the adversary accesses the system remotely. Nevertheless, it is possible to detect an intrusion, an alteration of this algorithm, at a specific instant : this allows to establish a dynamic root of trust. To do so, we rely on the remote attestation protocol. To secure the execution of this protocol, a static root of trust must be established, according to the threat model. This static root of trust, when formally verified, has formerly been established for simple embedded devices, such has microcontrollers from the MSP430 family. Obtaining it on these systems requires dedicated hardware support, with enough simplicity so that formal verification remains possible.

In order to allow formal verification to scale and to be able to target more complex hardware systems, we propose an automated formal verification approach that is compatible with the industrial requirements. The purpose of this approach is to minimize the abstraction effort and then to reduce the impact of the state-space explosion problem.

Then, we take advantage of modern *SoC* : we leverage the architecture to extend formally verified remote attestation to more complex devices. Our case study aims at establishing a static root of trust for a of-the-shelf microprocessor : the ARM Cortex-A9. To do so, we must provide secure execution for an attesting function and maintain confidentiality for a secret. We propose a definition for the security when targeting this family of systems, taking into account the rise of attack surface and capabilities of an adversary with high privileges. We separate these capabilities into uncorrelated problems, so that we can deal with them independently during formal verification.

We propose a design that we implement, and we confront our definition of security to this design and prove that it is verified. The solution we verify does not require any hardware modification : it is simply an extension for the microprocessor, a trusted peripheral, that is implemented in a field-programmable gate array.

Although, targeting proprietary architectures comes with an issue : with no access to the source, we cannot formally model their behaviour. Also, according to our strong threat model, we do not consider that the microprocessor represents a secure environment for the execution of the attesting function. As a consequence, we propose to enforce access controls and secure execution from outside the microprocessor : in the trusted peripheral. This mechanism relies on some hypothesis regarding the behaviour of the microprocessor, which we validated so that a reasonable amount of trust can be achieved.

Keywords : Security, Remote attestation, Formal verification, FPGA, *CoreSight*
