Attaque et sécurisation d'un schéma d'attestation à distance vérifié formellement

Jonathan Certes † et Benoît Morgan † ‡
jonathan.certes@irit.fr
benoit.morgan@irit.fr

† IRIT - ENSEEIHT, Université de Toulouse
‡ Intel Corporation

Résumé. Dans le cadre de l'attestation à distance sur microprocesseurs ARM, nous émettons l'hypothèse qu'un adversaire, privilégié mais distant, ne peut reproduire la trace d'exécution de notre algorithme de confiance optimisé. Compte-tenu de cette hypothèse, du support matériel garantit la sécurité d'une architecture de vérification de l'intégrité d'environnement d'exécution.

Nous validons notre hypothèse en auditant notre système, au travers d'attaques de bas niveau, en tentant de reproduire les signaux d'accès sans exécuter notre algorithme de confiance. Sous réserve que notre algorithme de confiance respecte certaines contraintes liées à l'architecture matérielle du microprocesseur, nous montrons que nous pouvons accorder un fort degré de confiance en notre hypothèse.

1 Introduction

Garantir la sécurité de l'exécution de logiciel sur un système complexe et distant 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, voir impossible, d'appliquer raisonnablement efficacement un schéma de vérification formelle complet.

De plus, l'utilisation de l'informatique distribuée, telle que l'informatique en nuage, contraint à considérer des modèles de menaces de plus en plus forts. Dans de tels paradigmes informatiques, les utilisateurs ne maîtrisent plus l'intégralité de leur infrastructure, en plus de la partager avec d'autres entités qui sont potentiellement malveillantes et / ou privilégiées. Les risques de compromission sont donc plus probables.

Par conséquent, afin d'avoir raisonnablement confiance en un environnement d'exécution, et ce même en cas de compromission, il est devenu nécessaire de pouvoir mesurer à distance l'intégrité d'un algorithme.

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 composant 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.

L'attestation à distance permet d'établir une racine de confiance dynamique, c'est-à-dire à l'exécution, même en présence d'un attaquant capable de corrompre l'intégralité d'une machine distante, à l'exception de sa racine de confiance statique. À l'issue de l'exécution du protocole, un utilisateur pourra décider de la confiance à accorder à l'algorithme distant, avant de lui transmettre par exemple des données sensibles.

Une racine de confiance statique de taille modeste simplifie sa vérification formelle, et ainsi la sécurité globale d'une méthode d'attestation proposée à défaut de la sécurité d'un système complet. Malheureusement, même si l'on limite la taille et la complexité d'un système à vérifier, certains éléments de spécification ne sont pas formalisables ou non disponibles sous forme de modèle car trop complexes ou propriétaires. Dans ce cas des hypothèses de fonctionnement sont posées et remises en question à l'aide d'audits de sécurité.

Les constructeurs de matériel ont déjà proposé des solutions d'environnement d'exécution de confiance, ou *Trusted Execution Environments* (TEE), qui peuvent aider à construire des racines de confiance statiques ou dynamiques (ARM *Trustzone*, *Intel Trusted eXecution Technology*). Certaines de ces solutions propriétaires proposent déjà des implémentations de l'attestation à distance 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.

Cet article s'inscrit dans le cadre de nos précédents travaux proposant une mise en œuvre de l'attestation à distance vérifiée pour microprocesseurs [7]. Nous présentons dans cet article une extension de notre architecture

de vérification de l'intégrité d'environnement d'exécution. Son objectif est d'attester localement l'exécution d'une fonction de configuration de l'environnement. Cette extension est à la fois vérifiée formellement et auditée lorsque la vérification formelle n'est pas possible.

Dans la continuité de nos précédents travaux [7], nous ciblons des systèmes complexes tels que les microprocesseurs modernes. En effet, nous avons choisi un microprocesseur *ARMv7 Cortex A9*. Ceci implique une large surface d'attaque pour l'adversaire : le système possède des mémoires cache, une unité de gestion mémoire (MMU : *Memory Management Unit*) ainsi que différents périphériques.

Initialement, nous avons vérifié formellement la sécurité de l'attestation à distance vis-à-vis d'un modèle de menaces simplifié [7]. Notamment, la configuration des périphériques, de la MMU et l'état des mémoires cache sont considérés corrects lors de l'exécution du protocole.

Dans cet article, 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 introduisons donc deux nouvelles contributions dans cet article :

— la proposition d'une extension de notre architecture co-conçue, logicielle et matérielle, de vérification de l'intégrité d'environnement d'exécution [7]. Son objectif est de se protéger du modèle de menaces plus fort ;

— une étude de la sécurité de cette extension, au travers d'un audit, face à un adversaire possédant de haut privilèges, une connaissance de l'architecture et un clone du système physique.

Dans la section 2, nous présentons le contexte de nos travaux et fournissons des rappels techniques. L'état de l'art est donné dans la section 3. La section 4 résume nos précédents travaux : une architecture de vérification de l'intégrité d'environnement d'exécution vérifiée formellement. La section 5 détaille notre première contribution, à savoir l'architecture de notre extension visant à protéger d'un adversaire plus fort. La section 6 détaille notre deuxième contribution, à savoir l'audit de sécurité de cette extension. Finalement, les résultats de notre étude sont donnés dans la section 7.

2 Contexte

2.1 Attestation à distance

Le protocole cryptographique d'attestation à distance consiste à vérifier si une machine *prover* possède, à un instant donné, les propriétés de sécurité

acceptables pour une machine de confiance distante appelée *verifier* [9]. Ce protocole s'appuie sur le paradigme question-réponse comme illustré sur la figure 1.

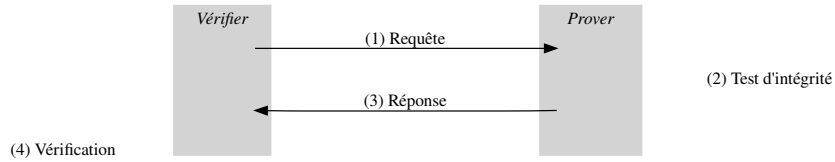

Fig. 1. Protocole d'attestation à distance

Dans un premier temps, le *verifier* envoie une requête ainsi qu'un challenge au *prover* (1) ; généralement, le challenge est un *nonce* qui permet d'éviter le re-jeu. Le *prover* calcule ensuite un test d'intégrité authentifié sur son environnement et le challenge (2) et retourne au *verifier* le résultat (3) ; cette étape sous-entend un prérequis : le *verifier* et le *prover* partagent un secret permettant l'authentification du résultat. A partir du résultat, le *verifier* décide alors si l'état du *prover* est valide ou non (4). Nous appelons fonction d'attestation la fonction qui calcule le test d'intégrité authentifié sur l'environnement du *prover* et le challenge à l'aide du secret partagé.

L'attestation à distance est donc un protocole de tolérance aux intrusions : elle permet la détection d'intrusion et le recouvrement dans le sens où l'on a un non-envoi des données sensibles et potentiellement une maintenance du *prover*.

2.2 Modèle de menaces

Afin d'illustrer nos travaux, nous considérons un modèle de menaces fort où l'adversaire n'a pas d'accès physique au système. Il peut donc prendre le contrôle du *prover* au travers d'un accès distant, via le réseau public.

L'adversaire a, au préalable, réalisé des attaques sur le *prover* et a élevé ses privilèges. Il lui est donc possible de lire et écrire dans toutes les mémoires où le microprocesseur a accès, de re-configurer les périphériques, d'empoisonner les mémoires cache, etc.

Egalement, l'adversaire possède un clone du *prover* : il peut donc reproduire l'environnement du *prover* sur un système où il a physique-

ment accès. Cela rend possible la connexion d'un analyseur logique et l'observation des signaux sur le matériel lors d'une exécution de logiciel.

Dans notre cas d'étude, l'adversaire a corrompu l'environnement du *prover* de telle sorte que l'attestation à distance par le *verifier* va détecter son intrusion. L'objectif de l'adversaire est donc de masquer cette corruption de manière à ce que l'attestation à distance réussisse. Pour cela, l'adversaire doit forger un résultat au test d'intégrité authentifié. Nous envisageons deux méthodes pour cela :

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

Vis-à-vis de notre modèle de menaces, pour garantir la sécurité de l'architecture de vérification de l'intégrité d'environnement d'exécution, nous devons donc :

1. assurer l'exécution sécurisée de la fonction d'attestation
2. maintenir la confidentialité du secret

2.3 Rappels techniques

Dans cette section, nous apportons des rappels techniques spécifiques à l'architecture que nous utilisons.

CoreSight est un périphérique des microprocesseurs ARM : une interface de *debug* qui permet de réaliser des traces. Une trace est une collecte non-invasive de données retraçant l'activité du microprocesseur sans le ralentir [3]. Les traces de *CoreSight* peuvent être communiquées à un périphérique matériel extérieur au microprocesseur.

Advanced eXtensible Interface (**AXI**) est un protocole de communication synchrone, avec une horloge unique, qui suit le paradigme maître-esclaves : un maître AXI peut donc communiquer avec plusieurs esclaves. Le protocole AXI fait partie de la spécification des bus de communication *Advanced Microcontroller Bus Architecture* (AMBA), le standard ouvert de ARM [4].

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 [5].

Le terme **branchement** désigne un saut dans l'exécution, c'est le terme employé par ARM. 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 [2].

3 État de l'art

Dans cette section, nous commençons par poser le contexte scientifique de l'attestation à distance et la sécurisation des *SoC* modernes, puis nous fournissons les détails techniques concernant nos travaux.

3.1 Attestation à distance

Les premiers travaux d'attestation à distance de processeurs complexes utilisent seulement du logiciel et basent l'authenticité de la réponse sur un temps d'exécution attendu.

Seshadri et al. définissent les bases de l'attestation à distance en proposant Pioneer [15] : une solution de test d'intégrité basé sur l'exécution d'une épreuve optimale en un temps attendu. L'épreuve est pensée de telle sorte qu'un adversaire ne peut modifier son environnement d'exécution ou son résultat en un temps inférieur ou égal au temps 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 (*overclocking*) pour réduire la durée du calcul [15].

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 [11]. 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 [11]. Une calibration du matériel est donc nécessaire pour le paramétrage du protocole d'attestation.

Des travaux plus récents utilisent la cryptographie pour mesurer l'authenticité de la réponse. Cette méthode implique le maintien d'un secret et nécessite donc du support matériel pour les contrôles d'accès.

Eldefrawy et al. proposent *SMART* [10], une extension matérielle d'un processeur Texas Instrument MSP430, couplée à un secret et une fonction

d'attestation logicielle. La fonction d'attestation calcule un test d'intégrité authentifié sur une région mémoire à attester. L'extension matérielle définit une région mémoire protégée où un secret est stocké. La sécurité de l'architecture de vérification de l'intégrité d'environnement d'exécution dépend dans ce cas de la confidentialité du secret. Lugou et al. [13] ont tenté de proposer une méthode unifiée de vérification d'architectures de sécurité co-conçues. Ils ont appliqué cette méthode sur *SMART* et modélisé leur système avec *Proverif*. Dans ces travaux, les propriétés de sécurité sont assurées par l'extension matérielle de *SMART*, capable de redémarrer le système en prévention de la compromission du secret.

De Oliveira Nunes et al. [14] ont formellement défini la sécurité de l'attestation à distance dans un modèle qui dépend de la confidentialité d'un secret partagé. Ils ont de plus trouvé une faille de sécurité dans les travaux précédents qui se base sur le masquage des interruptions. Ils proposent VRASED, un *framework* d'attestation à distance co-conçu et prouvé. L'implémentation est également réalisée sur un microcontrôleur MSP430, lourdement modifié, à l'aide d'une extension matérielle similaire à celle de SMART. L'extension matérielle est de type moniteur de sécurité : le cœur est modifié de façon à accéder à des signaux tels que le pointeur d'instruction et les lignes d'interruption, qui sont autant d'observables nécessaires au moniteur.

3.2 Sécurisation des *Systems on chip* modernes

Les *Systems on Chip* (*SoC*) modernes tels que le Xilinx Zynq-7000 proposent des microprocesseurs ARM étroitement intégrés avec un circuit logique programmable de type *Field-Programmable Gate Array* (FPGA) [6]. Ces *SoC* allient les performances d'un ASIC (*Application-Specific Integrated Circuit*) avec la flexibilité et le parallélisme d'un FPGA.

Wahab et al. tirent profit de l'interface de *debug CoreSight* des microprocesseurs ARM en la couplant à un FPGA [16]. Ils proposent une extension matérielle dédiée au *Dynamic Information Flow Tracking* haute-performances. La sécurité de cette solution dépend des contrôles d'accès matériel et du mode moniteur du TEE *TrustZone*.

Lee et al. utilisent *CoreSight* à des fins de sécurité [12]. Ils 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 *SoC*. Dans ces travaux, le décodage des traces permet l'obtention de la valeur du pointeur d'instruction à certains instants de l'exécution d'un programme. L'état de la configuration de la MMU et de *CoreSight* est supposé correct.

Dans nos précédents travaux [7], nous avons prouvé la sécurité de l'attestation à distance de VRASED [14] sur un modèle de microprocesseur sans modification du cœur. Les propriétés de sécurité ont été garanties par un moniteur matériel similaire à VRASED mais externe au processeur : implémenté dans la partie FPGA d'un *SoC* Xilinx Zynq-7000. Nous avons utilisé *CoreSight* pour obtenir les observables nécessaires à notre moniteur, tels que le pointeur d'instruction ou les signaux d'interruption, à des instants critiques de l'exécution d'une fonction d'attestation. Nos précédents travaux considèrent des hypothèses simplifiantes, vis-à-vis du modèle de menaces, pour aider la preuve. Ils supposent que l'état de configuration du microprocesseur est correct au début de l'exécution de la fonction d'attestation. En particulier, l'état des mémoires cache ainsi que les configurations de la MMU et de *CoreSight* ne sont pas corrompus par l'adversaire. Ceci est une faiblesse du modèle qui est admise.

3.3 Positionnement

Dans cet article, nous ré-utilisons l'architecture 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 [7] : c'est-à-dire de considérer un adversaire pouvant corrompre les caches, la configuration de *CoreSight* et de 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.

A la manière de Kovah et al. [11], 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* [15], nous utilisons des domaines d'horloge indépendants pour le microprocesseur et le moniteur matériel. Comme dans SMART et VRASED [10, 14], 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 *debug CoreSight*.

4 Architecture de vérification de l'intégrité d'environnement d'exécution sur microprocesseur

Dans cette section, nous résumons nos travaux précédents et détaillons l'architecture du système, vérifiée formellement, qui garantit, selon des hypothèses simplifiantes, l'intégrité d'environnement d'exécution sur microprocesseur.

4.1 Architecture du système

Le *SoC* envisagé est équipé d'un microprocesseur ARM Cortex-A9 mono-cœur étroitement lié avec un FPGA Artix-7. Nous implémentons une extension matérielle, dans la partie FPGA du *SoC*, qui communique avec le microprocesseur.

La région mémoire à attester contient un logiciel devant posséder les propriétés de sécurité suffisantes pour le *verifier*. Ce logiciel 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 l'ASLR 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 2 montre une vue d'ensemble du système.

Nous procédons à une ségrégation spatiale du contenu sensible. C'est-à-dire que nous créons trois zones mémoires protégées dans le FPGA :

- 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 [4]. Elles sont accessibles par le microprocesseur au

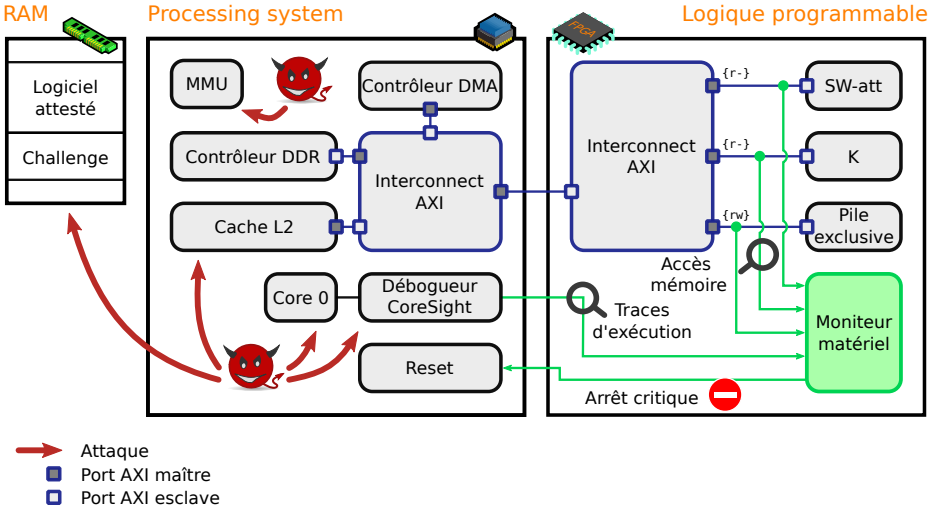

Fig. 2. Vue d'ensemble du système

travers d'un module d'interconnexion AXI. Elles sont donc indépendantes du contrôleur DDR principal.

Les zones mémoires protégées sont étendues avec un moniteur matériel. Ce moniteur matériel est un module qui observe les transaction sur le bus AXI. Une transaction a lieue 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 *debug CoreSight* pour garantir une exécution correcte de la fonction d'attestation. Des instructions spécifiques, par exemple 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.

La fonction d'attestation est *stand-alone* : elle est indépendant 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 la région mémoire à attester et le challenge, avec le secret. Une fois le calcul terminé, le résultat est placé à l'adresse du challenge, dans la DDR. La fonction d'attestation procède à un vidage des mémoires cache et des registres du microprocesseur avant et après son exécution.

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.

4.2 Vérification formelle et faiblesses admises

Nous avons vérifié formellement cette architecture vis-à-vis d'un modèle de menaces simplifié [7].

Les capacités de l'adversaire sont modélisées à haut niveau : seules les valeurs du pointeur d'instruction et de l'adresse de lecture du microprocesseur sont considérées. Ces valeurs ont un fort degré de liberté : il est possible d'effectuer des branchements et des lectures sans restriction.

Le modèle du microprocesseur est plus ou moins abstrait. D'un côté, comme son architecture est propriétaire, nous n'avons pas d'autre choix que d'abstraire son comportement. D'un autre coté, nous modélisons ce même comportement au niveau matériel, pour une configuration donnée (de *CoreSight* et de la MMU), au cycle d'horloge près.

Des propriétés de sécurité sont vérifiées formellement sur un modèle du moniteur matériel, au niveau registres (RTL). Comme nous avons accès à ses sources, le modèle est donc concret.

L'architecture proposée présente des faiblesses qui ont été admises. Ainsi, si nous la ré-utilisons sans modification, celle-ci est vulnérable à certaines attaques.

En effet, sa conception se base sur des hypothèses simplifiantes. Pour réduire les contraintes, le modèle émet l'hypothèse que l'adversaire n'accède pas à la configuration de la MMU ou de *CoreSight*. Cette hypothèse n'est plus réaliste vis-à-vis de notre nouveau modèle de menaces. Egalement, le modèle émet l'hypothèse qu'il est impossible pour un adversaire de procéder à une re-programmation partielle du FPGA et ainsi retirer des fonctionnalités du moniteur.

D'autres vulnérabilités sont potentiellement aussi présentes vis-à-vis du caractère propriétaire de l'architecture du microprocesseur. Ceci est une supposition : comme nous n'avons pas accès à l'architecture du microprocesseur (nous n'aurons probablement jamais accès), le modèle se base sur des hypothèses. Ces hypothèses sont des hypothèses réalistes, par exemple : "le microprocesseur fonctionne comme décrit dans sa documentation". Néanmoins, ce sont des hypothèses car nous ne pouvons pas les vérifier formellement.

Lors des vérifications, les hypothèses ont été validées par une étude empirique : elles ont été auditées sur le matériel concret. Par exemple : si une hypothèse est que *CoreSight* émet une trace lors d'un branchement indirect, alors un branchement indirect a été exécuté sur le microprocesseur et un analyseur logique a permis d'observer la trace.

Si les hypothèses qui ont été émises sont vérifiées, alors la vérification formelle apporte la preuve de la sécurité de l'architecture de vérification de l'intégrité d'environnement d'exécution. Si les hypothèses ne sont pas vérifiées, alors la vérification formelle n'est pas suffisante pour garantir la sécurité : elle ne garantit que les propriétés vérifiées sur le moniteur.

5 Attestation de configuration

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

5.1 Nouvelles classes d'attaques

Une nouvelle classe d'attaques dont nous voulons nous protéger est celle des attaques par re-configuration 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.

Une autre nouvelle classe d'attaques est celle des attaques par reconfiguration de la MMU.

En effet, les lectures et écritures réalisées par la fonction d'attestation sont réalisées à des adresses écrites dans son code, soit 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.

5.2 Solution proposée

Nous proposons d'étendre le système 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 de l'environnement de démarrage) et que l'adversaire ne peut accéder à sa configuration a-posteriori. 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 [6], 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. Un 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 configure *CoreSight* et la MMU.

L'architecture de notre extension est représentée sur la figure 3.

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

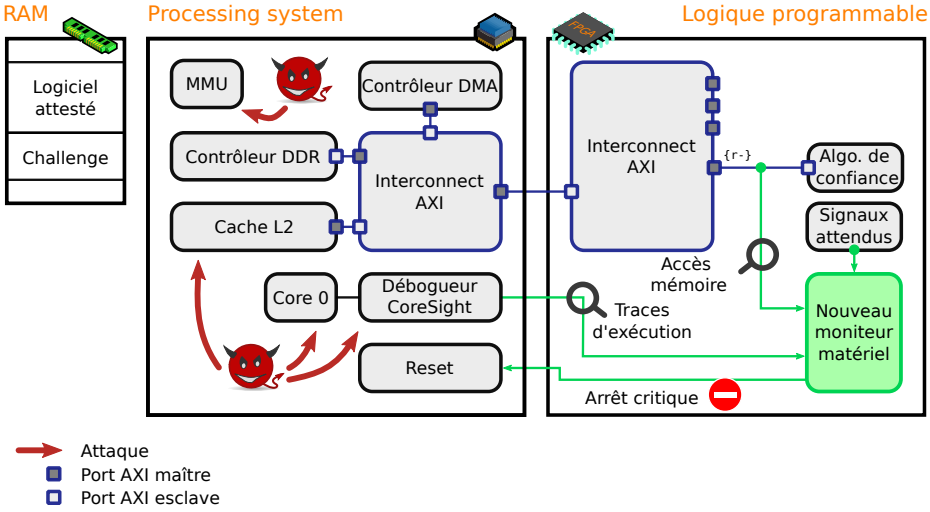

Fig. 3. Extension avec un nouveau moniteur matériel

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 nous sommes bien en train d'exécuter l'algorithme de confiance ou non.

A 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.

Notons que nous travaillons sur un microprocesseur qui ne possède qu'un seul cœur. Egalement, le module d'interconnexion AXI esclave (dans le FPGA) ne possède pas de connexion DMA avec un autre périphérique. 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 (ce qui 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**.

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.

Egalement, 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.

5.3 Caractérisation

Dans notre architecture, l'horloge du FPGA est fournie par un quartz externe : elle est décorrélée du domaine d'horloge du microprocesseur et sa fréquence est divisée jusqu'à atteindre la fréquence nominale de $100MHz$. Le périphérique d'interconnexion AXI maître (du microprocesseur) et l'interface de *debug CoreSight* sont synchronisés sur cette horloge externe. Leur fonctionnement est donc ralenti [6].

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 un vecteur nommé *ARADDR* et lève le signal *ARREADY* [4]. Sur notre matériel, une lecture de la donnée par le microprocesseur s'effectue par paquets de 8 mots de 32 bits, où chaque lecture nécessite 4 périodes d'horloge ; un délai de 15 périodes d'horloge est présent entre les lectures de deux paquets [8]. Le chronogramme représenté sur la figure 4 montre le résultat d'une lecture entre les adresses `0x00` et `0x44` de l'esclave AXI-Lite.

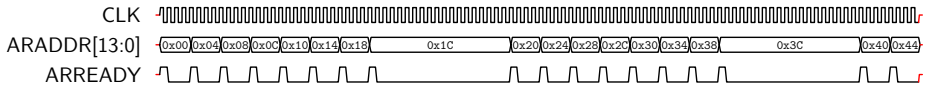

Fig. 4. Lecture depuis l'esclave AXI-Lite

Lorsque le contenu de l'esclave AXI-Lite est exécutable, un *fetch* par le microprocesseur provoque exactement les même signaux sur bus AXI que ceux représentés sur la figure 4. Observer ces signaux seuls n'est donc pas suffisant pour distinguer une lecture d'un *fetch* des instructions.

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 paquets de 8 mots, nous pouvons ajouter un branchement indirect tous les 8 mots pour que *CoreSight* émette une trace à chaque transaction sur le bus AXI. Ainsi, lors d'une exécution de code linéaire, nous pouvons prévoir le délai entre un accès sur le bus AXI et l'émission de paquets par *CoreSight*.

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

Le code représenté sur le listing 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 configuration de la MMU et de *CoreSight*.

```
1  nop
2  nop
3  nop
4  nop
5  nop
6  nop
7  add r3, pc, #0  // pc = pc + 4;   r3 = pc + 4 + 0;
8  mov pc, r3      // branchement indirect
9
10 // <- destination
11 nop
12 nop
13 nop
14 nop
15 nop
16 nop
17 add r3, pc, #0  // pc = pc + 4;   r3 = pc + 4 + 0;
18 mov pc, r3      // branchement indirect
```

Listing 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 $TRACE\_DATA$ [1]. 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 un transfert sur le bus AXI, nous devons au préalable vider la FIFO de *CoreSight* et choisir une adresse virtuelle faible pour l'esclave AXI-Lite [8].

Le chronogramme représenté sur la figure 5 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.

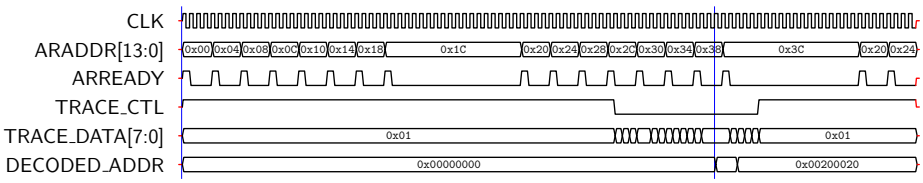

Fig. 5. Fetch et exécution d'instructions depuis l'esclave AXI-Lite

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.

Les signaux représentés sur le chronogramme de la figure 5 peuvent être modélisés formellement. Cette modélisation 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 de 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.

Néanmoins, nous faisons désormais face à une nouvelle problématique. Notre problématique initiale était de savoir comment distinguer, en observant les signaux, une exécution de l'algorithme de confiance d'une simple lecture. A priori, nous venons de montrer que cette distinction était possible. 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 d'exécution de la fonction d'attestation.

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 architecture 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 l'exécution sécurisée de la fonction d'attestation et la confidentialité du secret.

5.4 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*, en revanche, ne ralentit pas l'exécution [3].

Egalement, la présence de branchements indirects provoque un vidage du *pipeline* et un *fetch* à l'adresse de destination. Comme le microprocesseur réalise ses *fetch* de paquet de 8 instructions durant l'exécution des instructions précédentes, atteindre l'exécution de la dernière instruction provoque un second *fetch* du même paquet [8]. Ainsi, à l'exception du premier paquet de 8 instructions, chaque paquet est accédé deux fois sur le bus AXI. Le *fetch* qui a un effet sur le fil d'exécution du programme est donc celui du second accès. Le chronogramme représenté sur la figure 6 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.

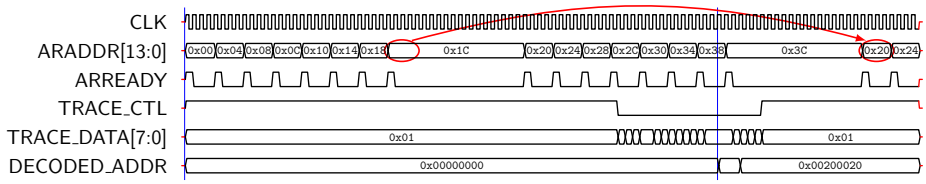

Fig. 6. Instants des *fetch* : branchement indirect et destination

Une seconde contrainte est l'alignement lors du développement de l'algorithme de confiance. En effet, si la destination d'un branchement se trouve dans le même paquet que l'instruction qui l'a provoqué, alors le vidage du *pipeline* provoque une exception de type *prefetch abort* [8]. L'algorithme de confiance est donc conçu de manière linéaire : c'est une suite d'instructions sans aucun branchement à l'exception des branchements indirects réguliers.

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 que nous entrons 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 la taille des paquets transmis par *CoreSight*.
4. Tout comme montré par Kovah et al. [11] 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 trans-

mettre 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 Audit de sécurité

Afin de vérifier formellement la sécurité de l'attestation à distance, nous émettons l'hypothèse suivante : **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 est réaliste ou non, nous devons conduire un audit de celle-ci 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 haut 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.

6.1 Accès sur le bus AXI

La première étape de notre audit 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 `LDM` permet d'effectuer une lecture de donnée à l'adresse définie dans un registre et incrémenter cette adresse [5]. 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 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 [5].

```
1  mov r0, #0x00200000 // adresse virtuelle
2  ldm r0!, {r1}       // r0 = r0 + 4 (!)
3  ldm r0!, {r1}
4  ldm r0!, {r1}
5  ldm r0!, {r1}
6  ldm r0!, {r1}
7  ldm r0!, {r1}
8  ldm r0!, {r1}
9  ldm r0!, {r1}
```

Listing 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 7 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 d'une lecture par paquets de 1 mot.

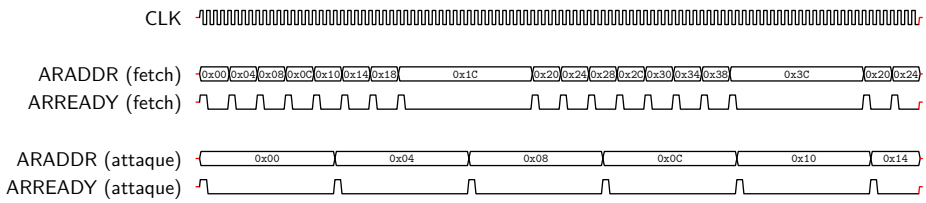

Fig. 7. Accès sur le bus AXI : *fetch* et lecture par paquets de 1 mot

Accéder à la donnée contenue dans l'esclave AXI-Lite par paquets 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 [5]. Ainsi, nous pouvons modifier le code pour réaliser une lecture par paquets de 8 mots en stipulant 8 registres de destination.

La présence de branchements indirects, dans l'algorithme de confiance, force une exécution légitime à accéder deux fois au même paquet de 8 mots. Pour reproduire ce comportement, nous pouvons également jouer sur l'utilisation du `!` pour ne pas incrémenter l'adresse de lecture lors du premier accès. Le code représenté sur le listing 3 montre une attaque qui reproduit les accès sur le bus AXI lors d'un *fetch*.

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

Listing 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 dans le paquet.

6.2 Signaux de *CoreSight*

Pour compléter notre attaque, nous devons également reproduire les signaux sur l'interface de *debug 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 4 montre une attaque tentant de reproduire les signaux de *CoreSight*.

```
1  mov r0, #0x40000000 // nouvelle adresse virtuelle (AXI-Lite)
2  ldm r0!, {r1, r2, r3, r4, r5, r6, r7, r8}
3  add r9, pc, #16     // pc = pc + 4; r9 = pc + 4 + 16;
4  mov pc, r9          // branchement indirect
5  nop
6  nop
7  nop
8  nop
9
10 // <- destination (adresse virtuelle = 0x00200020)
11 ldm r0,  {r1, r2, r3, r4, r5, r6, r7, r8} // pas de ! = 2 lectures
12 ldm r0!, {r1, r2, r3, r4, r5, r6, r7, r8}
13 add r9, pc, #16     // pc = pc + 4; r9 = pc + 4 + 16;
14 mov pc, r9          // branchement indirect
15 // ...
```

Listing 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 8 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.

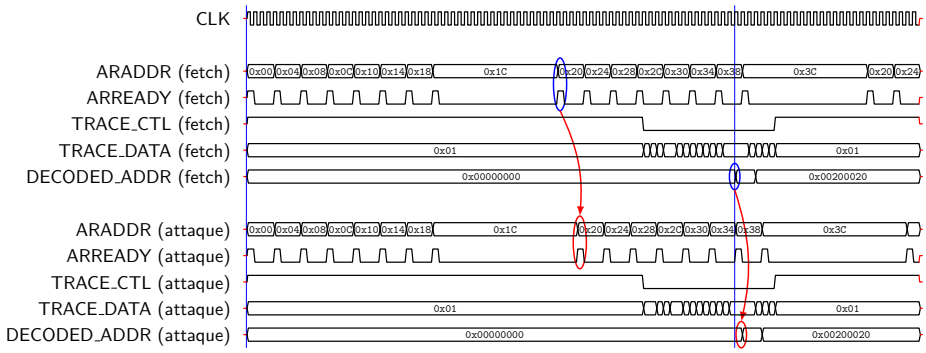

Fig. 8. Branchement indirect, calcul de la destination entre les lectures

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.

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 *debug* 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.

6.3 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 pré-chargés 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 5 représente cette attaque.

```
1  // destinations des branchements dans r9, r10, r11 et r12:
2  movw r9, #0x0020
3  movt r9, #0x0020     // r9  = 0x00200020
4  add r10,  r9, #0x20 // r10 = 0x00200040
5  add r11, r10, #0x20 // r11 = 0x00200060
6  add r12, r11, #0x20 // r12 = 0x00200080
7
8  mov r0, #0x40000000 // adresse virtuelle (AXI-Lite)
9  ldm r0!, {r1, r2, r3, r4, r5, r6, r7, r8}
10 mov pc, r9          // branchement indirect
11 nop
12 nop
13 nop
14 nop
15 nop
16
17 // <- destination (adresse virtuelle = 0x00200020)
18 ldm r0,  {r1, r2, r3, r4, r5, r6, r7, r8} // pas de ! = 2 lectures
19 ldm r0!, {r1, r2, r3, r4, r5, r6, r7, r8}
20 mov pc, r10         // branchement indirect
21 // ...
```

Listing 5. Pré-chargement des adresses de destination

Avec cette approche, les signaux d'accès sur le bus AXI ne sont pas ralentis. Egalement, 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 pré-charger 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 [5]. 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 pré-charger les adresses de destination des branchements indirects.

6.4 Bilan

Nous considérons notre hypothèse 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.

7 Résultats et futurs travaux

Notre architecture co-conçue logicielle et matérielle est implémentée sur une carte de développement Digilent Cora Z7-07S. Le *SoC* est un Zynq-7000 XC7Z007S comportant un microprocesseur ARM Cortex-A9 mono-cœur cadencé à 667MHz et un FPGA Xilinx Artix-7.

L'environnement permettant de reproduire les attaques précédemment listées est disponible publiquement [8]. Sont inclus également les descriptions matérielles du circuit implémenté dans le FPGA (esclave AXI-Lite, décompresseur et décodeur de traces *CoreSight* et analyseur logique embarqué) ainsi que le code de l'algorithme de confiance. Le circuit est synthétisé avec Xilinx Vivado v2019.2, le logiciel est compilé avec `gcc` et la carte est programmée avec `openOCD`.

Une vérification formelle du moniteur matériel (qui ne fait pas partie de l'objet de cette publication) démontre que la fonction d'attestation ne peut pas être exécutée si les signaux observés ne sont pas exactement reproduits. Ainsi, compte-tenu de notre hypothèse, nous avons montré que notre architecture co-conçue garantit la configuration de l'environnement d'exécution de la fonction d'attestation.

Cependant, nous avons seulement validé notre hypothèse en réalisant différentes attaques et en montrant que nous ne parvenons pas 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 peuvent donc consister a étayer nos modèles

formels de façon à pouvoir formellement prouver ou réfuter notre 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.

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. L'hypothèse est que nous pouvons écrire notre algorithme de confiance tel qu'il est impossible pour un adversaire de reproduire les signaux. Elle n'est pas invalide si une évolution de l'algorithme de confiance empêche de nouveau la reproduction des signaux par un adversaire.

Egalement, nous pouvons étendre notre solution afin de permettre à d'autres applications complexes d'en tirer profit. En effet, notre solution modifie les configurations de *CoreSight* et de la MMU avant l'exécution de la fonction d'attestation. Une sauvegarde et une restauration de l'environnement doivent donc être réalisées, respectivement en amont et en aval de l'attestation à distance. A ces fins, le développement et la publication d'une bibliothèque logicielle sont requis.

8 Conclusion

L'attestation à distance vérifiée formellement a jusqu'à présent été réalisée sur des microcontrôleurs simples, tels que la famille des MSP430 [10, 14]. Nous avons précédemment proposé, en nous appuyant sur les architectures des *SoC* modernes, d'étendre son domaine d'application aux microprocesseurs propriétaires pris sur étagères, tel que le ARM Cortex-A9. Nous avons dans ces travaux préliminaires, pu prouver la sécurité de l'attestation à distance sur microprocesseur, malheureusement au prix de suppositions concernant l'état du microprocesseur avant l'exécution du protocole d'attestation [7].

Dans cet article, nous introduisons une extension d'architecture conçue de vérification de l'intégrité d'environnement pour pallier à ces suppositions. Cela permet de garantir un environnement d'exécution correct pour la fonction d'attestation. À la manière des précédents travaux de ce domaine de recherche, notre solution s'appuie à la fois sur du

support matériel [10, 14] 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 [11, 15].

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 un audit technique 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. Compte tenu de cette hypothèse, notre architecture garantit alors une racine de confiance statique pour l'attestation à distance sur microprocesseur.

Dans le cas où de futures attaques, dont nous n'avons pas connaissance lors de l'écriture de cet article, réfutent notre hypothèse, notre solution co-conçue devra être adaptée. Enfin, afin de formellement compléter ces travaux, dans l'éventualité où un modèle du microprocesseur et de ses périphériques devient disponible, notre hypothèse pourra prendre alors la forme une obligation de preuve pour la sécurité globale de l'attestation à distance sur microprocesseur.

Références

1. *CoreSight TPIU-Lite Technical Reference Manual - Revision : r0p0*, number ARM DDI 0317A, 2006.
2. *ARM Coresight PFT architecture specification - PFTv1.0 and PFTv1.1*, number ARM IHI 0035B ID060811, 2008-2011.
3. *CoreSight PTM-A9 Technical Reference Manual - Revision : r1p0*, number ARM DDI 0401C ID073011, 2008-2011.
4. *AMBA AXI and ACE Protocol Specification - AXI3, AXI4, and AXI4-Lite ACE and ACE-Lite*, number ARM IHI 0022D ID102711, 2011.
5. *ARM Architecture Reference Manual - ARMv7-A and ARMv7-R edition*, number ARM DDI 0406C.c ID051414, 2014.
6. *Zynq-7000 All Programmable SoC - Technical Reference Manual*, number UG585 (v1.12.1) December 6, 2017, 2017.
7. Jonathan Certes and Benoît Morgan. Remote attestation of bare-metal microprocessor software : A formally verified security monitor. In *Database and Expert Systems Applications - DEXA 2021 Workshops - IWCFS, Virtual Event, September 27-30, 2021, Proceedings*, volume 1479 of *Communications in Computer and Information Science*, pages 42–51. Springer, 2021. `https://hal.archives-ouvertes.fr/hal-03576711`.

8. Jonathan Certes and Benoît Morgan. Verification materials : source code and examples. `https://gitlab.irit.fr/these-jonathan-certes-public/ressources/sstic_2022`, 2022.
9. George Coker, Joshua D. Guttman, Peter Loscocco, Amy L. Herzog, Jonathan K. Millen, Brian O'Hanlon, John D. Ramsdell, Ariel Segall, Justin Sheehy, and Brian T. Sniffen. Principles of remote attestation. *Int. J. Inf. Sec.*, 10(2) :63–81, 2011.
10. Karim Eldefrawy, Gene Tsudik, Aurélien Francillon, and Daniele Perito. Smart : Secure and minimal architecture for (establishing dynamic) root of trust. In *NDSS*, volume 12, pages 1–15, 2012.
11. Xeno Kovah, Corey Kallenberg, Chris Weathers, Amy Herzog, Matthew Albin, and John Butterworth. New results for timing-based attestation. In *IEEE Symposium on Security and Privacy, SP 2012, 21-23 May 2012, San Francisco, California, USA*, pages 239–253. IEEE Computer Society, 2012.
12. Yongje Lee, Ingoo Heo, Dongil Hwang, Kyungmin Kim, and Yunheung Paek. Towards a practical solution to detect code reuse attacks on ARM mobile devices. In Ruby B. Lee, Weidong Shi, and Jakub Szefer, editors, *Proceedings of the Fourth Workshop on Hardware and Architectural Support for Security and Privacy, HASP@ISCA 2015, Portland, OR, USA, June 14, 2015*, pages 3 :1–3 :8. ACM, 2015.
13. Florian Lugou, Ludovic Apvrille, and Aurélien Francillon. Toward a methodology for unified verification of hardware/software co-designs. *Journal of Cryptographic Engineering*, pages 1–12, 2016.
14. Ivan De Oliveira Nunes, Karim Eldefrawy, Norrathep Rattanavipanon, Michael Steiner, and Gene Tsudik. VRASED : A verified hardware/software co-design for remote attestation. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1429–1446, Santa Clara, CA, August 2019. USENIX Association.
15. Arvind Seshadri, Mark Luk, Adrian Perrig, Leendert van Doom, and Pradeep K. Khosla. Pioneer : Verifying code integrity and enforcing untampered code execution on legacy systems. In *Malware Detection*, pages 253–289. 2007.
16. Muhammad Abdul Wahab, Pascal Cotret, Mounir Nasr Allah, Guillaume Hiet, Vianney Lapotre, and Guy 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*, pages 1–7, 2017.