Ministère de l'enseignement supérieur et de la recherche

Institut National des Sciences Appliquées
Toulouse

Département de Génie Électrique & Informatique

Mémoire de fin d'études

Développement d'une architecture de confiance pour l'informatique en nuage

---

IRIT 2 rue Charles Camichel 31000 Toulouse

Paul FLORENCE
Génie Informatique - Sécurité informatique
perso@florencepaul.com

13 septembre 2020

RÉSUMÉ

Durant les 23 semaines qui composent ce stage de fin d'étude j'ai travaillé à l'amélioration de l'architecture de sécurité développée au sein de l'IRIT [1]. Cette architecture de sécurité est dirigée vers l'usage *Cloud* de l'informatique et propose une amélioration de la sécurité des serveurs hébergeant des machines virtuelles. Les Hyperviseur [2] sont des logiciels historiquement très vulnérables et présentant une surface d'attaque importante. Cette architecture de sécurité est basée sur un hyperviseur de sécurité mettant en place un mécanisme d'attestation à distance afin de vérifier l'intégrité de l'ordinateur. Pour cela un périphérique de confiance est utilisé afin d'assister l'hyperviseur dans sa preuve d'intégrité. J'ai tout d'abord porté le périphérique de confiance sur une nouvelle architecture électronique, avant de le remplacer par un micro-contrôleur. En deuxième partie de stage, j'ai formalisé et amélioré l'épreuve cryptographique utilisée dans la phase d'attestation en me basant sur des publications scientifiques.

1. Institut de Recherche en Informatique de Toulouse.
2. Une plate-forme de virtualisation permettant d'exécuter des machines virtuelles sur une seule machine physique.

Les travaux présentés dans ce manuscrit ont été réalisés à l'Institut de Recherche en Informatique de Toulouse (IRIT).

REMERCIEMENTS

Je tiens à exprimer ma gratitude envers Benoît Morgan qui m'a encadré durant ce stage et a toujours su trouver du temps pour m'aider malgré sa vie de famille et son emploi du temps chargé. Je remercie aussi tout particulièrement Annabelle Sansus qui a su faciliter mon arrivée administrative à l'IRIT et a été d'une gentillesse à toute épreuve.

Je remercie aussi toute l'équipe pédagogique du Département Génie Électronique et Informatique de l'INSA Toulouse et tout particulièrement l'équipe enseignante à TLS-SEC. Merci à Eric Alata, Vincent Nicomette ainsi que Daniela Dragomirescu qui dirige le département en portant une attention particulière au bien être des étudiants et des étudiantes.

Merci à mes parents qui m'ont toujours apporté soutien et réconfort quelles que soient les circonstances.

Enfin, je n'aurai pas pu réaliser ce stage sans le soutien de mes formidables amis à qui je dois beaucoup : merci Julien, Célia, Benjamin, Kapau, Jérôme, Lucien, Mélanie et Quentin, vous êtes incroyables !

TABLE DES MATIÈRES

TABLE DES FIGURES

3. System on Chip
4. Field Programmable Gate Array
5. Peripheral Component Interconnect Express

GLOSSAIRE

ACADIE Assistance à la Certification d'Applications DIstribuées et Embarquées

ANSSI Agence Nationale de Sécurité des Systèmes d'Informations : L'agence assure la mission d'autorité nationale en matière de sécurité des systèmes d'information. À ce titre elle est chargée de proposer les règles à appliquer pour la protection des systèmes d'information de l'État et de vérifier l'application des mesures adoptées. Dans le domaine de la défense des systèmes d'information, elle assure un service de veille, de détection, d'alerte et de réaction aux attaques informatiques, notamment sur les réseaux de l'État [21].

AXI Advanced eXtensible Interface : un bus de communication haute performance parallèle, synchrone conçu principalement pour la communication entre les composants d'un même circuit intégré. C'est un bus sous licence libre.

Bootloader Bootloader : Programme informatique mettant en place le contexte d'exécution d'un système d'exploitation. Il s'occupe de charger en mémoire les composants principaux de celui-ci, de détecter les périphériques existants et de lancer le noyau du système d'exploitation.

CSR Control and Status Register : un registre d'état et de contrôle permettant de configurer et de surveiller un périphérique dans un système embarqué (micro-contrôleur, FPGA, etc.).

FPGA Field Programmable Gate Array : Un réseau de portes logiques programmables. Il est possible de reconfigurer les cellules et les bascules logiques d'un FPGA afin d'implémenter le circuit électronique de son choix.

Hypercall Hypercall : L'équivalent d'un appel système, mais pour de la virtualisation. Un système d'exploitation conçu pour la virtualisation utilise des hypercall pour communiquer avec l'hyperviseur.

Hyperviseur Une plate-forme de virtualisation permettant d'exécuter des machines virtuelles sur une seule machine physique.

IDS Intrusion Detection System : Un système de détection d'intrusion (antivirus, routeur intelligents, etc.).

IRIT Institut de Recherche en Informatique de Toulouse.

MMU Memory Management Unit : l'unité de gestion de la mémoire ajoute une couche d'indirection entre la mémoire physique et la mémoire manipulée par les programmes. Cela permet de découpler les espaces d'adresses et facilite la programmation, car tous les programmes peuvent considérer qu'ils s'exécutent

sur une zone mémoire contiguë de grande taille même si la réalité physique est toute autre.

MSI — Message Signaled Interrupts : une interruption implémentée en utilisant un message dédié dans le protocole de communication, par opposition à une ligne d'interruption qui nécessite un câble dédié pour signaler l'interruption. Elles sont supportées par le bus PCIE.

PCIE — Peripheral Component Interconnect Express : Un bus de communication présent dans les ordinateurs basés sur les processeurs *x86_64*. Il spécifie un protocole de communication, les connecteurs physiques a utilisé ainsi que la topologie du bus.

PRNG — Pseudo Random Number Generator : Générateur de nombres pseudo-aléatoires.

QEMU — QUick EMulator : Un logiciel sous licence libre de machine virtuelle permettant notamment d'exécuter des systèmes d'exploitations à travers l'hyperviseur KVM.

RISC — Reduced Instruction Set Computer : Une architecture matérielle pour les processeurs basés sur un jeu d'instruction simple composée de peu d'instructions rapides à décoder.

ROM — Read Only Memory : mémoire en lecture seule.

RTL — Register Transfer Level : c'est une méthode de description des architectures microélectroniques.

Secure Boot — Système de démarrage sécurisé mettant en place une racine de confiance et prouvant par petites étapes l'intégrité du système. Le processeur valide d'abord le BIOS à l'aide d'un secret stocké dans le silicone du processeur. Puis le BIOS valide le chargeur de démarrage (Bootloader) qui vérifie ensuite l'intégrité du noyau chargé. À chaque étape les composants logiciels sont signés par une clé issue d'un algorithme de cryptographie asymétrique (par exemple RSA) dont la partie publique est stockée en physiquement dans des fusibles non-reprogrammables se trouvant dans une TPM

SMM — System Management Mode : un mode du processeur octroyant encore plus de privilège que le mode VMX Root car dédié à la gestion de l'énergie.

SoC — System on Chip : Système composé d'un cœur de processeur et de périphériques annexes (contrôleurs mémoires, bus de communication, etc.).

SPI — Serial Peripheral Interface : bus de donnée série synchrone avec un modèle de communication maître/esclave.

TLP — Transaction Layer Packet : les paquets de la couche 3 du modèle PCIE (cf Figure 4.4).

TPM — Trusted Platform Module : Cryptoprocesseur permettant de stocker des secrets de manière sécurisée.

UART	Universal Asynchronous Receiver Transmitter : il s'agit d'un composant matériel permettant la communication série.
UEFI	Unified Extensible Firmware Interface (ou interface micro-logicielle extensible unifiée) : c'est la spécification de l'interface entre le micrologiciel (*firmware*) de la plateforme et le système d'exploitation. Successeur du BIOS, UEFI assure l'indépendance entre le système d'exploitation et la plate-forme matérielle sur laquelle il fonctionne.
Verilog	Langage de description matériel permettant de concevoir des circuits logiques.
VMCS	Virtual Machine Control Structure : la structure de données permettant de représenter le contexte d'une machine virtuelle (l'état du processeur lors de son arrêt, etc.). Elle est stockée dans la mémoire principale du processeur (la RAM).
Wishbone	Un bus de communication sous licence libre, utilisé dans les cœurs de processeurs pour sa simplicité. Il est conçu pour permettre aux différents composants d'un circuit intégré de communiquer entre eux.

Première partie

INDUSTRIALISATION D'UN PÉRIPHÉRIQUE DE CONFIANCE

1 INTRODUCTION

Durant ce projet de fin d'étude, j'ai travaillé sur une architecture de sécurité mettant en place de l'attestation à distance afin de garantir son intégrité. Dans cette partie, vous trouverez trois chapitres. Le chapitre 1 que vous lisez actuellement permet d'étudier l'intérêt de cette architecture par rapport au contexte économique et sociétal actuel. Le chapitre 2 sera consacré à la définition de cette architecture et au cadre dans lequel j'ai travaillé tandis que le chapitre 3 présentera l'implémentation existant à mon arrivée dans le laboratoire.

1.1 L'IMPORTANCE DE L'ATTESTATION À DISTANCE DANS L'INFORMATIQUE EN NUAGE

L'informatique en nuage (ou *Cloud Computing* en anglais) est un terme vague désignant généralement la disponibilité à la demande de ressources informatiques, tels que du temps processeur, du stockage ou encore des applications. Ces ressources sont mises à disposition sur le réseau, à travers Internet [19].

On y retrouve plusieurs types de produits , tels que le logiciel à la demande (plus connu sous le nom de *Software as a Service - SaaS*), la plate-forme à la demande (*Platform as a Service - PaaS*) et le *back-end* mobile à la demande (*Mobile Backed as a Service - MBaaS*).

Ce sont ces services que les plate-formes comme Amazon Web Services ou Microsoft Azure proposent aux entreprises.

C'est un service intéressant pour une entreprise car elle peut considérer qu'elle a accès à des ressources informatiques illimitées et de payer uniquement en fonction de son usage. Cela octroie une grande flexibilité dans la consommation des ressources informatiques et a permis le développement de la notion d'élasticité . Ces avantages en font une technologie en pleine croissance économique, avec des revenus en augmentation pour les différents acteurs du secteur [6].

L'élasticité est la capacité d'un système à s'adapter à la charge, aux besoins et aux demandes d'approvisionnement rapidement.

Dans une infrastructure permettant l'informatique en nuage, les programmes de deux clients peuvent s'exécuter sur le même ordinateur. Il est donc crucial d'isoler ces deux applications afin de garantir l'intégrité, la confidentialité et la disponibilité de ces applications et des données dont elles dépendent. Pour cela, l'informatique en nuage tire parti des technologies de virtualisation afin de permettre à deux programmes informatiques de cohabiter sur la même ressource matérielle. Cependant, ces technologies complexes offrent une surface d'attaque qui n'existait pas auparavant. En effet, si l'on considère chaque fonctionnalité d'un hyperviseur comme un vecteur d'attaque, on en dénombre plus d'une dizaine (MMU logicielle,

mécanismes de chronomètre et d'interruptions, émulation du processeur, Hypercall [1], gestion des machines virtuelles, etc.) [20].

Il est donc nécessaire de sécuriser ces couches logicielles. Dans le cadre de ce mémoire, nous allons nous concentrer sur la détection d'intrusion.

On distingue deux familles d'IDS [2] : la recherche de signature et la détection d'anomalie. Cependant dans le contexte de la virtualisation ces méthodes peuvent ne pas détecter un attaquant si celui-ci utilise le matériel pour se dissimuler. C'est sur ce principe que le logiciel malveillant *bluepill* a été conçu : en virtualisant le système d'exploitation cible, *bluepill* devient invisible aux yeux de celui-ci et des programmes de défense [11]. Il faut donc concevoir un IDS spécifique pour les hyperviseurs.

1.2 L'attestation à distance comme mécanisme de détection d'intrusion

Le sujet de l'attestation à distance, c'est-à-dire de « *déterminer si un système informatique distant exécute la version correcte d'un programme informatique* » [12], est bien connu dans la littérature scientifique. Dans notre cas, nous utilisons les mécanismes d'attestation à distance afin d'obtenir la preuve qu'un programme informatique a été exécuté sans modification sur un système informatique distant potentiellement compromis.

Pour cela, on retrouve dans la littérature deux grands types d'approches :

— L'approche basée sur un périphérique matériel qui sers de racine de sécurité. Souvent nommé TPM [3], la preuve de sécurité réside dans la difficulté à compromettre ce composant matériel. En effet, on y retrouve des secrets cryptographiques gravés dans le matériel (souvent à l'aide de fusibles) qui ne sont pas exposés aux composants principaux de la machine. L'interface de communication de la *TPM* expose des primitives cryptographiques de base, mais ne dévoile jamais le secret. Un attaquant souhaitant compromettre ce type de périphérique doit alors s'attaquer au matériel. Dans la pratique cette approche à comme défaut la nécessité de remplacer le matériel sur tout le parc informatique en cas de bogue entraînant une faille de sécurité. De plus, cette approche est aussi sécurisée que le plus faible de ses maillons, et toute la chaîne doit être sécurisée. C'est l'approche choisie par Intel avec la technologie *Intel SGX* qui souffre de nombreuses vulnérabilités [22] [24] [25].

— L'approche basée sur les canaux auxiliaires utilise les propriétés de certains calculs, comme la consommation d'énergie ou le temps d'exécution, pour s'assurer que le programme n'a pas été modifié. En mesurant ces propriétés physiques, on vérifie que l'exécution du programme correspond bien aux mesures enregistrées sur une

1. Hypercall : L'équivalent d'un appel système, mais pour de la virtualisation. Un système d'exploitation conçu pour la virtualisation utilise des hypercall pour communiquer avec l'hyperviseur.
2. Intrusion Detection System
3. Trusted Platform Module

machine non vérolée. Le principal problème de cette approche est que de nombreux éléments peuvent venir perturber la mesure.

Les technologies développées et améliorées pendant ce stage appartiennent à la deuxième catégorie et permettent en théorie d'attester à distance la non compromission d'un ordinateur. Elles sont donc potentiellement très intéressantes pour les fournisseurs de systèmes informatiques en nuage car elles offrent une nouvelle façon de sécuriser leurs services, plus sûre qu'Intel SGX.

1.3 TECHNOLOGIES ABORDÉES PENDANT LE STAGE

Ce stage a été pour moi l'occasion de travailler avec des nombreux outils et technologies.

En effet, le périphérique de confiance a été tout d'abord implémenté sur un FPGA en utilisant le langage de description matériel Verilog [4]. Le travail sur ce périphérique de confiance a demandée la compréhension du standard PCIE et des spécificités du travail sur un FPGA (utilisations des cœurs propriétaires fournis par *Xilinx*, blocs de RAM, domaines d'horloges, etc.).

Par la suite, une deuxième version basée sur un micro-contrôleur Arduino a été réalisé en *C*. Puis, la version finale du périphérique de confiance a été implémentée sur un micro-contrôleur *STM32F103* en utilisant le langage de programmation sûr *Rust*. L'hyperviseur de sécurité étendu pendant le stage est quant à lui implémenté en *C* et en assembleur, et la compilation est réalisée à l'aide de fichier *Makefile*. Le travail sur cet hyperviseur a nécessité d'apprendre des notions nouvelles en micro-architecture *x86_64* .

Rust est un langage de programmation compilé conçu pour la programmation système et la sûreté. Développé par Mozilla depuis 2010, il garantit l'absence de fautes de mémoires et de concurrence critique à la compilation. Il existe aussi un guide de programmation édité par l'ANSSI [8].

L'implémentation de l'épreuve cryptographique a tout d'abord été réalisée en *C* avec certaines parties en assembleur en ligne, puis dans un deuxième temps l'implémentation a été portée sur de l'assembleur *x86_64* pour y être étendue. Cette implémentation a nécessité une bonne compréhension de la micro-architecture des processeurs *Intel*, ainsi que des extensions matérielles de virtualisation. Le traitement des mesures effectuées pendant le stage a été réalisé en utilisant *Python* avec la librairie de calcul numérique *numpy*.

Enfin ce stage a nécessité de lire de nombreux articles scientifiques traitant de l'attestation à distance afin de maintenir l'architecture de sécurité à l'état de l'art, notamment sur lors de mon travail sur l'épreuve cryptographique.

1.4 IMPACT DE LA PANDÉMIE DE COVID-19

Un élément important du contexte de ce stage est la terrible pandémie de *COVID-19* qui a frappé notre civilisation. En effet, celle-ci a complément bouleversée l'organisation de ce stage de fin d'étude. Bien que j'eusse commencé mon stage à temps, il s'agissait du 16 mars 2020, le lendemain

4. Langage de description matériel permettant de concevoir des circuits logiques.

de la déclaration du confinement total de la population française par Emmanuel Macron. J'ai donc passé 21 des 23 semaines de mon stage en télétravail. Ces semaines passées en télé-travail ont rendus impossible la communication informelle et les nombreuses interactions nécessaires pour permettre de travailler efficacement. J'ai ainsi eu de nombreuses et récurrentes difficultés. Je n'ai pas été épargné par la période de confinement total qui a eu une influence négative sur mon moral et ma motivation. Au final, j'estime avoir perdu beaucoup de temps ce qui a impacté la quantité et la qualité des réalisations effectuées durant ce stage. J'ai toutefois continué à développer le projet et la recherche de solution et j'ai persévéré afin d'avancer le plus possible dans la finalisation de mon stage.

De plus la pandémie a aussi affecté les fournisseurs du laboratoire, et nous n'avons pas pu obtenir le FPGA sur lequel je devais travailler. Dans la section 4.1 je présente donc les modifications réalisées, mais elles n'ont jamais étés testées sur le matériel. Le stage s'effectuant dans un laboratoire de recherche j'ai eu une grande liberté dans mes choix de conceptions et j'ai pu par la suite m'éloigner des idées initialement prévues pour proposer une autre démarche basée sur un micro-contrôleur. Ce travail est présenté en section 4.2.

2

CONTEXTE ET ARCHITECTURE DE SÉCURITÉ POUR L'INFORMATIQUE EN NUAGE

Afin de bien comprendre les objectifs et la manière dont est développée l'architecture de sécurité sur laquelle j'ai travaillé au sein de l'IRIT il faut d'abord s'intéresser aux objectifs de l'équipe ACADIE [1] qui s'occupe de son développement.

L'IRIT est une Unité Mixte de Recherche Occitane comptant environ 700 membres.

2.1 L'ÉQUIPE DE RECHERCHE ACADIE

L'IRIT est composé de 24 équipes de recherches différentes et ACADIE est l'une d'entre elle. ACADIE est structurée en trois groupe ; théorie des types et des catégories, développement de logiciels critiques, et vérification de systèmes distribués critiques, mon travail s'est inséré dans celui effectué par le groupe travaillant sur les logiciels critiques.

FIGURE 2.1 – Le logo de l'équipe ACADIE

Au sein de l'antenne de l'IRIT dans laquelle j'ai travaillé, la plupart des membres de l'équipe travaille principalement sur la conception de système sûr par construction, en utilisant des méthodes formelles. Il y a une vingtaine de membres permanents, et autant de non-permanents.

Enfin, mon stage s'inscrit dans une volonté de mettre en place des transferts de technologies du laboratoire vers l'industrie (voir la citation de Michel Daydé en Figure 2.2).

1. Assistance à la Certification d'Applications DIstribuées et Embarquées

« *Le laboratoire travaille à instaurer un continuum allant de la recherche à la valorisation, malgré les difficultés et les limitations inhérentes aux dispositifs existants (propriété industrielle, brevet), en imaginant et développant des formes de collaboration innovantes autour des notions de laboratoire commun ou de consortium. La fertilisation du tissu économique local et national via les grands groupes mais aussi en resserrant nos relations avec les PME et PMI est devenue pour l'IRIT un devoir.* »

FIGURE 2.2 – Michel Daydé, directeur de l'IRIT [7]

C'est donc dans le but d'industrialiser les travaux réalisés par Benoît Morgan durant sa thèse [17] que la création de ce stage a été décidée, afin de mettre en valeur les travaux réalisés par l'équipe dans le but de les proposer au monde de l'entreprise.

Au cours de ce stage j'ai donc eu un double rôle au sein du projet : rénover, nettoyer, remanier les différents projets tout en concevant un nouveau prototype mettant en place de l'attestation à distance permettant de réaliser une démonstration. Pour cela j'ai travaillé sur différents composants logiciels et matériels qui sont décrit dans les sections suivantes.

2.2 L'ARCHITECTURE DE SÉCURITÉ

2.2.1 *Définition du modèle de menace*

Les modèles de menaces de cette architecture de sécurité sont originalement définis dans [17] avec les surfaces d'attaques des différents composants. Pour les besoins de ce stage nous n'avons cependant besoin que des capacités de l'attaquant. Au vu des vulnérabilités apparues ces dernières années [14] [16] nous considérons qu'un attaquant est capable d'exploiter le matériel et de passer outre les protections mises en place par les processeurs tels que la MMU [2]. Nous considérons aussi qu'un attaquant est capable d'effectuer des accès en lecture et en écriture sur toute la mémoire physique et qu'il est capable d'exécuter du code arbitraire dans le mode *VMX Root* à partir d'une machine virtuelle. L'architecture de sécurité est donc conçue pour résister à un modèle d'attaquant particulièrement puissant, mais réaliste au vu de l'état de la sécurité des composants logiciels et matériels que l'on trouve dans un serveur.

Le mode VMX Root est l'un des modes les plus privilégiés sur les processeurs x86_64 . Une définition plus précise de ce mode se trouve en sous-section 3.1.1.

2.2.2 *Aperçu de l'architecture de sécurité*

Comme décris par Benoît Morgan dans sa thèse [17], l'architecture de sécurité est constituée de trois composants : l'hyperviseur de sécurité *Abyme*, le matériel assistant *Abyme* sous la forme d'un périphérique de confiance (nommé *Eric*) et le serveur d'attestation. Pour un schéma de cette architecture, voir la Figure 2.3.

Afin de prouver la non-compromission des machines virtuelles hébergées sur l'ordinateur fournissant les services de l'informatique en nuage le processus suivant est utilisé :

1. Le serveur d'attestation contacte le périphérique de confiance avec une épreuve cryptographique à faire exécuter par *Abyme*.
2. *Eric* envoie l'épreuve à *Abyme* et mesure le temps d'exécution. La mesure et le résultat sont combinés pour vérifier l'intégrité de l'hyperviseur.

2. Memory Management Unit

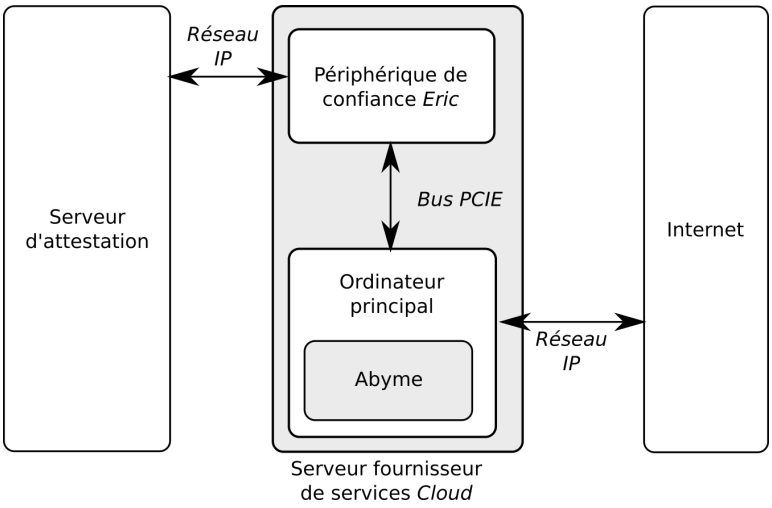


FIGURE 2.3 – L'architecture de sécurité composée d'*Abyme* et d'*Eric*

3. *Eric* remonte toutes ces informations au serveur d'attestation qui s'occupent de les traiter et de lever les alertes nécessaires.

2.2.3 *Le serveur de virtualisation*

Le serveur de virtualisation est l'ensemble des composants matériels constituant l'ordinateur exécutant l'hyperviseur. Cet hyperviseur utilise les technologies de virtualisation pour permettre le déploiement à la demande de machines virtuelles selon les besoins des clients de l'entreprise offrant des services d'informatique en nuage. Pour cela, il s'appuie sur les extensions de virtualisations fournies par les processeurs *x86_64* .

Dans cette architecture, l'hyperviseur est séparé en deux composants logiciel, l'un développé au sein de l'équipe ACADIE, l'hyperviseur de sécurité *Abyme* et un autre hyperviseur plus généraliste pris sur étagère comme *Xen*, *VMWare ESXi* ou encore *Hyper-V*.

En effet, l'hyperviseur *Abyme* met en place les mécanismes d'attestation à distance pour assurer son intégrité. Une fois celle-ci prouvée, il s'occupe ensuite de vérifier l'intégrité de l'hyperviseur qu'il virtualise.

L'hyperviseur généraliste est virtualisé par *Abyme*. Il est utilisé pour ses fonctionnalités de gestion des machines virtuelles. En effet, *Abyme* est un hyperviseur très spécialisé et n'offre que peu de fonctionnalités par rapport à un hyperviseur est pris sur étagère. On utilise donc les capacités d'un hyperviseur généraliste pour offrir des services d'informatique en nuage, *Abyme* offrant une couche de sécurité supplémentaire.

Comme les technologies de virtualisation ont avancé depuis les premières implémentations, il est désormais possible de virtualiser un hyperviseur avec un coût faible en performance. En utilisant un hyperviseur pris sur étagère nous nous épargnons donc le développement d'un hyperviseur complet et nous pouvons nous concentrer sur les fonctionnalités indispensables.

Cependant *Abyme* n'est pas le seul élément de cette architecture de sécurité. En effet, l'intégrité d'*Abyme* ne peut être vérifié par celui-ci, car un attaquant ayant les capacités décrites dans le modèle de menaces (voir sous-section 2.2.1) est capable de virtualiser *Abyme*. L'un des moyens de s'en rendre compte étant de concevoir une épreuve cryptographique permettant de vérifier l'intégrité d'*Abyme* et de vérifier qu'il n'est pas virtualisé. Pour cela, nous avons donc besoin d'un composant matériel dédié ainsi que d'un mécanisme cryptographique.

Ce composant matériel est appelé périphérique de confiance, car il s'agit de la racine de confiance du système.

Pour les lecteurs familiers avec le fonctionnement des systèmes de Secure Boot [3], le périphérique de confiance joue un rôle semblable à la TPM dans le Secure Boot.

2.2.4 *Le périphérique de confiance*

Le périphérique de confiance est la racine de confiance de l'architecture de sécurité. C'est-à-dire qu'il est considéré inattaquable et inviolable. Cela permet de se servir de lui pour prouver que le reste du système est non compromis. Cette hypothèse étant très forte, il est nécessaire d'apporter un soin particulier à ce que l'implémentation présente une surface d'attaque la plus petite possible.

Le rôle de ce périphérique est de recevoir les demandes d'attestation du serveur d'attestation et de les transmettre à l'hyperviseur de sécurité. Il est aussi chargé d'effectuer lui-même des vérifications de l'environnement de l'hyperviseur en vérifiant l'intégrité de certaines zones mémoires. Pour les besoins de l'épreuve d'attestation à distance il doit être capable de mesurer précisément le temps d'exécution de l'épreuve sur le serveur de virtualisation. Enfin, il doit être capable de communiquer avec le serveur d'attestation à travers le réseau, il doit donc posséder au moins une interface réseau.

Le périphérique de confiance joue un rôle clé dans l'architecture de sécurité puisqu'il est chargé de mettre en place l'attestation à distance. Tout la preuve d'intégrité repose sur son bon fonctionnement et sa non-compromission.

2.2.5 *Le serveur d'attestation*

Le serveur d'attestation est le composant logiciel qui se charge de déterminer à quel moment envoyer au périphérique de confiance une épreuve et qui pré-calcule les résultats de celle-ci. Dans le cadre de ce stage, le serveur d'attestation n'a pas été modifié.

2.2.6 *L'épreuve d'attestation à distance*

Cette épreuve est composée de deux primitives :

— Un protocole de communication entre le serveur d'attestation, *Eric* et *Abyme*.

— L'implémentation d'une épreuve cryptographique permettant de s'assurer que le résultat de l'épreuve provient bien de son exécution, et qui demande un temps mesurablement plus long à s'exécuter quand elle est dans un contexte virtualisé.

2.3 RENDRE L'ARCHITECTURE DE SÉCURITÉ UTILISABLE EN PRODUCTION

Ce stage s'ancrant dans la démarche de développement des relations entre l'IRIT et les entreprises et de valorisation du travail produit au sein du laboratoire, son but final est la production d'un prototype permettant de réaliser des démonstrations pour permettre d'attirer l'attention d'acteurs industriels sur le projet.

Initialement, mon stage devait se porter uniquement sur la modernisation du périphérique de confiance. Cependant, à cause de la pandémie de *COVID-19* (voir section 1.4 pour plus de détails) je me suis aussi occupé de la conception, de l'implémentation et de la validation de l'épreuve cryptographique.

2.3.1 *Migrer l'implémentation du périphérique de confiance Eric*

Le périphérique de confiance existant à mon arrivée était implémenté sur un FPGA coûteux (quelques milliers d'euros par unité) et basé sur l'architecture *Virtex 6* de Xilinx commercialisée en 2009. Afin de réduire la barrière d'entrée pour utiliser cette architecture de sécurité, mon objectif était de migrer l'implémentation du périphérique de confiance sur un FPGA moins cher, plus compact et plus performant. Le FPGA choisi est le modèle *Aller Artix-7 FPGA Board with M.2 Interface* (*cf Figure 2.4*) pour plusieurs raisons :

— Le FPGA embarqué est basé sur l'architecture *Artix-7* de *Xilinx*, plus récente, plus performante et moins énergivore.
— La carte présente un connecteur PCIE au format *M.2* qui est un format compact présent sur la majorité des ordinateurs récents. Conçus pour les appareils mobiles, il permet de réduire l'encombrement du périphérique.
— Le FPGA comporte des émetteurs-récepteurs PCIE haute-vitesse facilitant le support du protocole : *Xilinx* fournis des modèles Verilog gérant les couches basses du protocole PCIE.
— La carte embarque une TPM permettant de stocker des secrets cryptographiques et de mettre en place un système cryptographique robuste à l'état de l'art.
— Enfin, la carte est peu chère (*400* €).

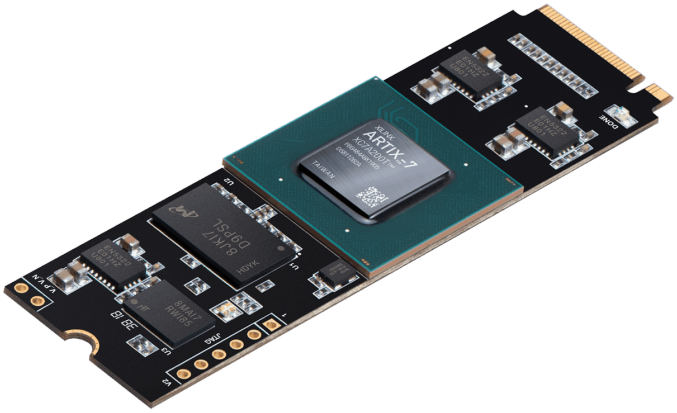


FIGURE 2.4 – La carte électronique embarquant le FPGA du périphérique de confiance [1]

2.3.2 *Conception d'une épreuve d'attestation à distance pour Abyme*

Afin de permettre l'utilisation de cette architecture de sécurité par un acteur commercial, il a fallu augmenter la confiance que l'on peut porter dans les systèmes cryptographiques mis en place dans le protocole d'attestation à distance proposé par Benoît Morgan dans sa thèse. En effet, les primitives cryptographiques proposées dans [17] ne sont pas assez fortes pour un usage réel et doivent être combinées avec les résultats d'autres travaux dans le domaine de l'attestation à distance.

L'objectif final étant l'implémentation en assembleur *x86_64* d'une épreuve cryptographique permettant de vérifier l'intégrité d'*Abyme*.

3

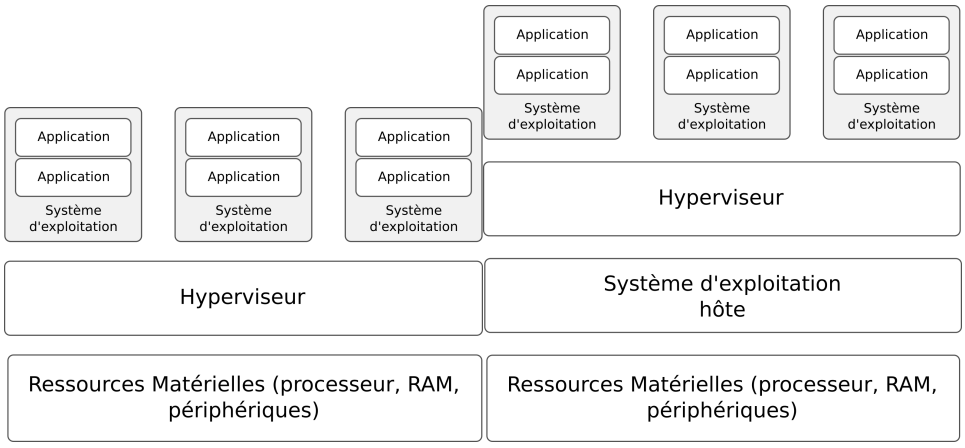
TRAVAUX PRÉCÉDENTS

Les réalisations effectuées pendant mon stage sont basées sur les travaux réalisés précédemment par Benoît Morgan et ses collaborateurs. Dans cette partie, il s'agit de décrire précisément le fonctionnement des différents composants de l'architecture de sécurité présentés dans le chapitre 2. Dans la première section je présente les composants matériels et logiciels constituant cette architecture, et dans la deuxième section je réalise un état de l'art des épreuves d'attestation à distance.

3.1 L'HYPERVISEUR DE SÉCURITÉ

Abyme est un hyperviseur de sécurité dont le développement a débuté aux environs de 2013 pour les besoins des travaux de Benoît Morgan dans le cadre de sa thèse [18] [17]. Il s'agit d'un hyperviseur de type 1 mettant en place une virtualisation totale en utilisant les extension matérielles proposées par Intel (*Intel-VT*) [9].

Un hyperviseur de type 1 est un hyperviseur s'exécutant directement sur le matériel sans couches intermédiaires (cf *Figure 3.1*). On parle de virtualisation totale car le processus de virtualisation est totalement transparent pour le logiciel virtualisé : il s'exécute comme s'il avait accès directement au matériel. Des mécanismes sont mis en place pour assurer le partage des ressources avec les autres logiciels virtualisés.



(a) Hyperviseur de type 1 (b) Hyperviseur de type 2

FIGURE 3.1 – Architecture des hyperviseurs de type 1 et de type 2

3.1.1 *Support des extensions de virtualisation*

Ces fonctionnalités sont appelées Intel VT-X et AMD-V

Avec le développement du support de la virtualisation par les processeurs *x86_64* , Intel et AMD ont rajoutés des fonctionnalités facilitant le travail des fabricants d'hyperviseur. Ces fonctionnalités sont identiques dans leur conception chez les deux fabricants. Pour faciliter le partage des ressources entre plusieurs systèmes d'exploitations pensant être maître de machine, un niveau de privilège supérieur a été introduit dans les processeurs *x86_64* : le mode *VMX Root* (cf Figure 3.2). Dédié aux hyperviseurs, il offre accès à des instructions de virtualisation.

Avec ces extensions de virtualisation, il devient donc possible pour un hyperviseur d'exposer au processeur un contexte de machine virtuelle (par la suite nommé en utilisant la nomenclature Intel : VMCS [1]) et d'utiliser des instructions dédiées pour les manipuler. Ils représentent l'état d'une machine virtuelle vu depuis le processeur à un instant t. On peut notamment charger et reprendre l'exécution d'une machine virtuelle à travers les VMCS. Ils sont placés en mémoire *RAM* et servent de pont entre le mode d'exécution *VMX Root* et les modes d'exécutions moins privilégiés : à chaque changement de contexte (exécution d'une instruction nécessitant un changement de privilège par une machine virtuelle, faute d'exécution, faute de page, etc.) le processeur met à jour le VMCS associé à la machine virtuelle en question. Une fois l'origine du changement de contexte traitée par l'hyperviseur, celui-ci met à jour le VMCS si nécessaire, et il sera utilisé par le processeur pour restaurer le contexte d'exécution de la machine virtuelle la prochaine fois qu'elle sera exécutée.

Un programme hôte est donc capable de configurer les zones mémoires, les périphériques et les ressources accessibles par les invités.

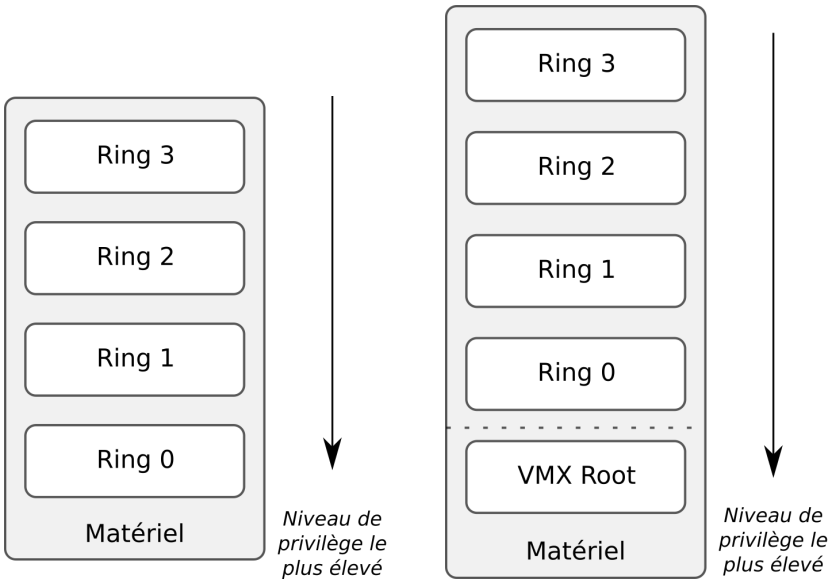
Le programme s'exécutant en mode *VMX Root* est appelé hôte, tandis que les programmes s'exécutant en mode non *VMX Root* sont appelé invités.

3.1.2 *Support de la virtualisation récursive*

Abyme est capable de virtualisation récursive (ou virtualisation imbriquée), c'est-à-dire de se virtualiser lui-même en utilisant les extensions de virtualisation. Pour cela, l'hyperviseur hôte attrape toutes les fautes générées par l'exécution des instructions de virtualisations par les invités (comme les instructions *CPUID, vmread, vmwrite, vmxon*, etc.). Ces fautes doivent ensuite être traitées afin de permettre le bon fonctionnement des hyperviseurs invités.

Enfin, *Abyme* supporte QEMU [2] et uniquement les processeurs Intel récents (pas de support d'AMD).

1. Virtual Machine Control Structure
2. QUick EMulator



(a) Niveaux de privilèges en mode non virtualisé

(b) Niveau de privilèges en mode virtualisé

FIGURE 3.2 – Comparaison des modes de privilèges sur un processeur supportant la virtualisation

3.1.3 *Test d'intégrité*

Le test d'intégrité est déclenché par le périphérique de confiance à travers une MSI [3]. L'arrivée d'une MSI sur le processeur interrompt son fonctionnement, et celui-ci appelle la routine enregistrée dans la table des interruptions. Cette routine exécute l'épreuve envoyée par le périphérique de confiance et lui transmet le résultat avant de reprendre son exécution. Comme expliqué dans la section 3.3, le temps d'exécution fait partie des résultats de l'épreuve. Il est donc important de minimiser le délai entre la réception d'une MSI et le traitement de celle-ci. Si le périphérique de confiance juge le temps de calcul trop long, ou reçoit un résultat invalide il remonte cette information au serveur d'attestation qui pourra prendre la décision adéquate (redémarrer la machine, etc.).

3.1.4 *Chargement de l'hyperviseur par le système UEFI*

L'hyperviseur n'est pas le premier logiciel exécuté par le processeur. En effet, celui-ci commence par exécuter le micrologiciel embarqué dans la ROM [4]. Dans notre cas, il s'agit de la couche logicielle UEFI [5]. Celle-ci s'occupe entre autres d'offrir des services de détection du matériel à l'hyperviseur afin de faciliter sa mise en place.
Le système UEFI permet de concevoir deux types de services : les services

3. Message Signaled Interrupts .
4. Read Only Memory
5. Unified Extensible Firmware Interface (ou interface micro-logicielle extensible unifiée)

de démarrage (*Boot Services*) qui sont accessibles uniquement avant le démarrage de l'OS et les services d'exécution (*Runtime Services*) qui restent accessibles une fois l'OS démarré. Le démarrage de l'hyperviseur commence par le chargement d'un pilote réseau pour les cartes réseau Intel. Celui-ci se présente sous la forme d'un *runtime service* et expose un protocole UEFI pouvant être utilisé par d'autres services afin d'utiliser la carte réseau.

Un protocole UEFI est une interface logicielle permettant la communication entre deux services UEFI.

Une fois le pilote de la carte réseau chargé, *Abyme* est chargé à son tour et démarre. Il prend la forme d'un service d'exécution UEFI et lors de son exécution il s'installe en mémoire, démarre le mode *VMX Root* du processeur et met en place la virtualisation pour ensuite rendre la main au logiciel UEFI qui est désormais virtualisé. Il est ensuite possible d'utiliser un Bootloader[6] afin de charger un système d'exploitation. Cette séquence est spécifiée dans un script de démarrage exécuté par une console UEFI (cf Annexe A.1)

Pour conclure sur *Abyme*, il est conçu pour être la première couche de virtualisation dans le cadre d'une infrastructure dédiée à l'informatique en nuage. En effet, il sert de démonstrateur permettant de mettre en application les innovations apportées par les membres d'ACADIE et à terme il pourra exécuter des hyperviseurs standards du monde de l'industrie (Xen, VMWare, etc.).

3.2 LE PÉRIPHÉRIQUE DE CONFIANCE ERIC

Eric est un périphérique de confiance connecté via un bus PCIE à *Abyme* (cf Figure 2.3). Complètement séparé de la machine hôte, il embarque son propre processeur et sa propre mémoire. Cette isolation permet de placer une confiance importante dans ce périphérique puisque la surface d'attaque depuis le processeur principal est très faible car elle est constituée uniquement de l'interface PCIE et du port réseau relié directement au serveur d'attestation. Cela permet de justifier l'hypothèse établissant ce périphérique comme étant inviolable et permet d'en faire la racine de confiance de notre système comme expliqué dans la sous-section 2.2.4.

Conçu en Verilog, il est adapté à partir du SoC *Milkymist* et il possède les composants matériels suivants :

- Un cœur de processeur basé sur le jeu d'instruction *LM32* et les travaux de Sébastien Bourdeauducq sur *Milkymist*[4]. Il se présente sous la forme d'un module Verilog pris sur étagère. C'est un cœur RISC[7] *big-endian* sans MMU. Il est possible de compiler nativement des programmes pour le jeu d'instruction *LM32* car il est supporté par le compilateur *gcc*.
- Un contrôleur mémoire sous la forme d'une boîte noire fournie par Xilinx permettant d'utiliser les blocs de RAM installés sur la carte.

L'endiannes ou boutisme en français désigne l'ordre dans lesquels sont placés les octets représentants une donnée.

6. Bootloader
7. Reduced Instruction Set Computer

— Un contrôleur PCIE permettant de communiquer avec le serveur de virtualisation.
— Un contrôleur réseau permettant de communiquer avec le serveur d'attestation.
— Un contrôleur UART [8] permettant d'avoir une sortie de deboguage sur le périphérique à travers une connexion série.
— Un bus Wishbone [9] reliant les différents modules du SoC. C'est sur ce bus que transitent les données entre les différents cœurs lors d'une opération (réception d'un message PCIE, émission d'une trame ethernet, etc.).
— Un pont Wishbone vers CSR [10] permettant d'exposer les registres de contrôle et de surveillance des périphériques. Cela permet de configurer les périphériques (ethernet, UART, contrôleur PCIE, etc.) via des opérations mémoires sur des adresses spécifiques.

Les différents modules matériels sont tous cadencés à 100 MHz, sauf le cœur propriétaire PCIE qui nécessite une cadence de 250 MHz.

La Figure 3.3 illustre plus précisément l'architecture matérielle du SoC.

3.3 DÉVELOPPEMENT D'UNE ÉPREUVE CRYPTOGRAPHIQUE POUR L'ATTESTATION À DISTANCE

L'épreuve cryptographique proposée par Benoît Morgan est basée sur l'instruction assembleur *x86_64 CPUID*. L'instruction *CPUID* (*CPU IDentifaction*) est utilisé pour obtenir des informations sur le modèle de processeur et déterminer quelles fonctionnalités sont disponibles. Elle ne prend aucun argument, et utilise le registre *eax* afin de déterminer l'information demandée. La plupart des processeurs *x86_64* exposent 26 entrées : la valeur fournie à *CPUID* par le registre *eax* doit être comprise entre 0 et 16, ou entre 0x80000000 et 0x80000008. Cependant la spécification indique qu'il vaut mieux commencer par interroger le processeur sur l'argument maximal possible avant d'appeler *CPUID* avec une valeur différente de 0. Exécutée dans un contexte non *VMX-Root* elle déclenche une sortie inconditionnelle de la machine virtuelle (on parle de *vmexit*) et l'hyperviseur doit traiter cette instruction. Un *vmexit* est un processus très coûteux en temps car il s'agit d'un changement de contexte et le processeur doit nettoyer les zones de mémoires tampons (TLB, caches,) ainsi que restaurer le contexte d'exécution de l'hyperviseur qui traite ce *vmexit*. Il doit ensuite à nouveau changer de contexte pour permettre à la machine virtuelle interrompue de reprendre son exécution ce qui demande à nouveau beaucoup de temps.

Cette sortie inconditionnelle n'est vraie que chez Intel, AMD permet à un hyperviseur d'indiquer qu'il ne souhaite pas modifier le comportement de l'instruction CPUID.

Ainsi, si lors de l'exécution de l'épreuve *Abyme* n'est plus l'hyperviseur de plus haut niveau de la machine, les multiples *vmexit* engendrés par l'exécution de *CPUID* sont détectables par un observateur extérieur mesurant

8. Universal Asynchronous Receiver Transmitter
9. Un bus de communication sous licence libre, utilisé dans les cœurs de processeurs pour sa simplicité.
10. Control and Status Register

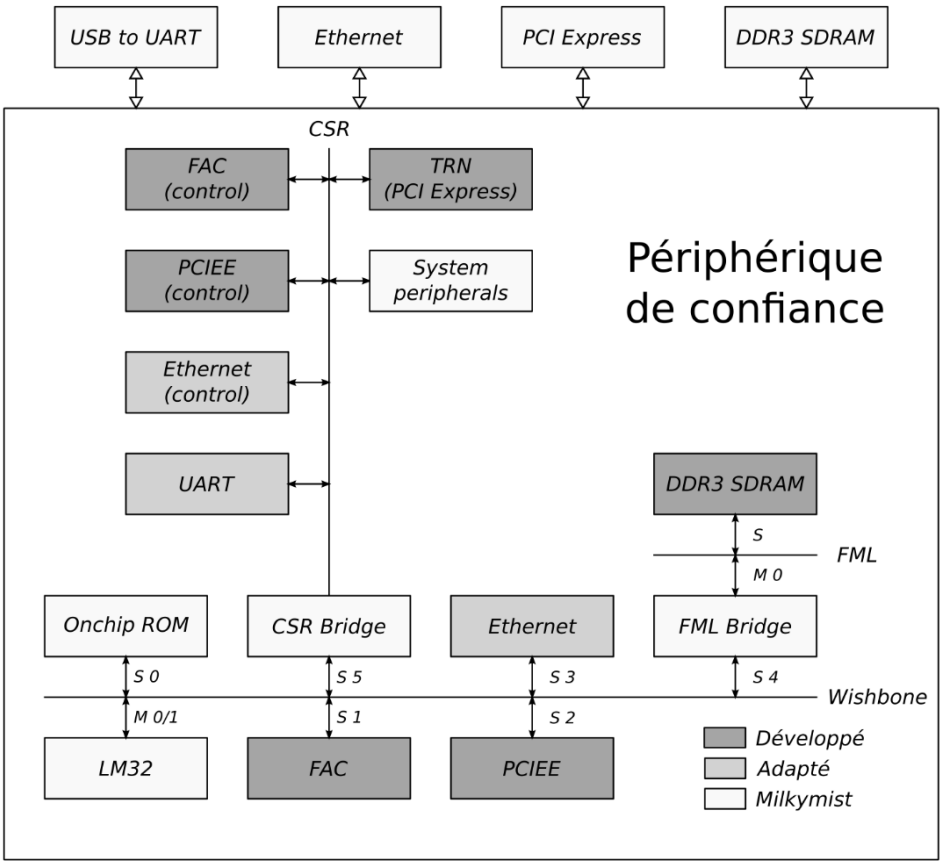


FIGURE 3.3 – L'architecture du SoC développé par Benoît Morgan. Schéma issu de sa thèse [17].

```
// CPUID  0 - f
//        80000000 - 8000008
void challenge_start(void) {
  uint64_t rax, rbx, rcx, rdx; // Resultat de l'appel a CPUID
  uint32_t i, j;
  uint64_t result[4]; // Resultat de l'epreuve
  for (j = 0; j < 0x100; j++) {
    // init
    result[0] = 0, result[1] = 0, result[2] = 0, result[3] = 0;
    // 0 to f
    for (i = 0; i < 0x10; i++) {
      rax = i, rbx = 0x0, rcx = 0x0, rdx = 0x0;
      __asm__ __volatile__("CPUID" : "=a"(rax), "=b"(rbx), "=c"(rcx),
          "=d"(rdx) :
          "a"(rax), "b"(rbx), "c"(rcx), "d"(rdx));
      result[0] ^= rax, result[1] ^= rbx, result[2] ^= rcx, result[3]
          ^= rdx;
    }
    // 80000000 - 80000008
    for (i = 0x80000000; i < 0x9; i++) {
      rax = i, rbx = 0x0, rcx = 0x0, rdx = 0x0;
      __asm__ __volatile__("CPUID" : "=a"(rax), "=b"(rbx), "=c"(rcx),
          "=d"(rdx) :
          "a"(rax), "b"(rbx), "c"(rcx), "d"(rdx));
      result[0] ^= rax, result[1] ^= rbx, result[2] ^= rcx, result[3]
          ^= rdx;
    }
  }
}
```

FIGURE 3.4 – Algorithme de l'épreuve dans le cadre d'une attestation à distance visant à détecter la virtualisation de l'exécutant proposé par [17]

le temps d'exécution de l'épreuve. Une première version d'une épreuve visant à vérifier qu'*Abyme* est bien toujours *VMX Root* a été proposée dans [17] et est décrite dans la Figure 3.4.

Cependant, au vu du modèle de menace décrit dans la sous-section 2.2.1 cette épreuve présente plusieurs vulnérabilités.

3.3.1 *Limites de l'épreuve basée uniquement sur CPUID*

Cette épreuve est vulnérable à plusieurs classes d'attaques.

Tout d'abord, il est possible d'effectuer des attaques par rejeu. Comme le résultat de l'exécution de l'épreuve est toujours le même, un attaquant connaissant l'épreuve ou pouvant observer le réseau est capable de déterminer le résultat de l'épreuve sans l'exécuter. Il lui est alors possible de se camoufler aux yeux d'*Eric*. Ce serait le cas d'un attaquant ayant réussi à s'installer en *VMX Root* et virtualisant notre hyperviseur de sécurité car il

serait capable d'attraper la MSI générée par *Eric* et de répondre à la place d'*Abyme*.

Ensuite, un attaquant se trouvant en position d'homme du milieu entre *Abyme* et *Eric* peut choisir de ne pas transmettre l'épreuve entraînant un déni de service. Cependant ce déni de service est détecté par l'architecture de sécurité car le périphérique de confiance ne reçoit aucun résultat pour l'épreuve.

Dans le cas où un attaquant est en position d'homme du milieu, il peut aussi effectuer une attaque par modification de l'épreuve. Cela consiste en la modification des instructions qui entraînerait la détection de l'attaquant par des instructions plus rapides. Ainsi, en remplaçant l'instruction *CPUID* par une opération de consultation dans une table, un attaquant est capable de créer une version de l'épreuve dont le temps d'exécution est le même dans un contexte *VMX Root* et non *VMX Root*. Il peut alors se dissimuler aux yeux du périphérique de confiance.

Dans la sous-section 4.3.2 je définirai les propriétés que doit vérifier une épreuve répondant à notre modèle de menace. Puis en se basant sur les travaux d'autres équipes je proposerai une implémentation d'une telle épreuve et vérifierai son comportement expérimental.

Deuxième partie

RÉALISATIONS

4

AMÉLIORATION DE L'ARCHITECTURE DE SÉCURITÉ

4.1 MIGRATION DU PÉRIPHÉRIQUE DE CONFIANCE *eric* SUR LA PLATEFORME *virtex-7*

La première tâche que j'ai effectuée a été la migration de l'implémentation du périphérique de confiance sur un nouveau FPGA. Comme expliqué en section 1.4, l'absence de matériel a ensuite entraîné un changement d'orientation des objectifs du stage.

4.1.1 *État de l'art des technologies de conceptions matérielles automatisées*

Eric est actuellement écrit en *Verilog*, un langage de description du matériel permettant d'abstraire la complexité de la conception d'un circuit électronique. Basé sur de nombreuses couches d'abstractions, les outils permettant de transformer une description *Verilog* en transistors et en portes logiques gèrent de nombreux problèmes (agencement des portes, transistors, RTL [1], etc.) ce qui permet aux concepteurs de se concentrer sur le développement du circuit intégré. Cependant, *Verilog* est un langage évoluant très lentement et ayant reçu peu d'améliorations au cours des années. Cette lenteur en fait un langage difficile à appréhender, et ne tirant pas parti des avancées dans le domaine de la conception des langages de programmation [10]. C'est pourquoi je me suis intéressé aux nouvelles méthodes de conceptions matérielles afin de réaliser un projet maintenable plus facilement.

S'agissant d'un SoC, c'est un projet complexe comportant de nombreux composants reliés entre eux. *Verilog* est un langage très basique offrant peu de constructions et très limitant d'un point de vue expressivité. *SystemVerilog* apparu en 2009 en reprend la syntaxe et l'étends afin de corriger ces limitations. Cependant tous les outils utilisés pour développer *Abyme* ne supporte pas *SystemVerilog*. Il est donc impossible de migrer vers ce langage.

Les SoC sont des produits de plus en plus courants, et suivent toujours la même architecture (pour un exemple, voir la Figure 4.1). En effet, on retrouve toujours différents composants matériels (processeur, contrôleur mémoire RAM et flash, etc.) reliés entre eux par un ou deux bus de communication (souvent du Wishbone ou de l'AXI [2]). Des outils de générations de SoC sont donc apparus, et permettent de faciliter le travail du concepteur. Ces outils fonctionnent presque tous de la même manière :

1. Register Transfer Level
2. Advanced eXtensible Interface

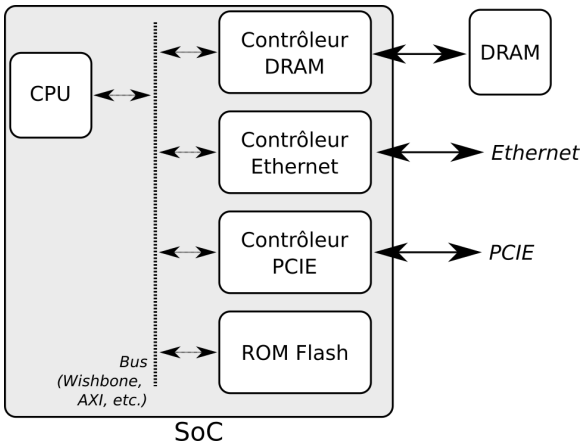


FIGURE 4.1 – Exemple de SoC générique

— Les différents composants (processeur, contrôleurs mémoires, etc.) sont écrits en *Verilog* et suivent une nomenclature très précise au niveau de leur interface avec le monde extérieur afin de permettre à l'outil de génération de SoC d'interpréter correctement les ports d'entrées/sorties.
— Un langage dédié est utilisé afin de décrire les composants utilisés, leurs paramètres et les connexions.
— L'outil génère un fichier *Verilog* contenant toute l'implémentation du SoC. Ce fichier n'est pas conçu pour être modifié à la main.

RISC-V est un jeu d'instruction libre apparus il y a quelques années et qui cherche à proposer une alternative au monopole de l'entreprise ARM sur le marché des processeurs embarqués.

Enfin, avec l'apparition et l'adoption des plate-formes basées sur *RISC-V* des nouvelles méthodes de développement matériels sont apparues. Au cœur de ces méthodes on trouve la réutilisation de composants matériels et logiciels visant à réduire les coûts, les risques et le temps de développement. Ces nouvelles méthodes entraînent l'apparition de nouveaux langages de description du matériel plus faciles d'accès et plus confortables [10].

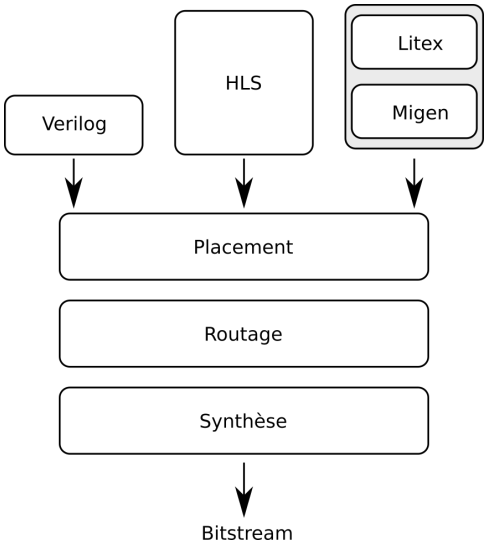


FIGURE 4.2 – Les trois approches étudiées pour la conception sur FPGA

4.1.1.1 *Litex & Migen*

Migen (pour *Milkymist Generator*) est un langage dédié à la description du matériel basée sur *Python*. C'est une suite d'outils composée, d'une librairie *Python* et d'un système de compilation qui permet de transformer le *Python* en un produit final prêt à être programmé sur FPGA. Il est donc possible d'utiliser des paradigmes de langages de programmation moderne tels que la programmation orientée objet, la méta-programmation ou encore les librairies issues de l'écosystème *Python* dans la conception d'un circuit intégré ce qui en fait une technologie de choix pour la conception matérielle.

FIGURE 4.3 – Le logo de *Migen*

Litex est un outil permettant de créer des SoC basés sur *Migen*. Grâce à une librairie variée de composants (plusieurs CPU, contrôleur ethernet, PCIE, DRAM, SATA, USB, HDMI, etc.) le travail de l'ingénieur est facilité. Enfin, *Litex* embarque un système de plate-forme permettant d'implémenter un SoC indépendamment du FPGA utilisé. On obtient alors un code *Python* portable entre les différentes plate-formes. Il est très simple d'instancier et de relier des composants complexes car *Litex* s'occupent automatiquement de nombreuses tâches qui incombaient auparavant au concepteur : gestion de la taille des bus de communication, séparation des domaines d'horloge , adaptation des composants aux ressources matérielles (RAM, pins, etc.)

Un domaine d'horloge est l'ensemble des composants utilisant une même horloge pour se synchroniser. Il existe en général plusieurs domaines d'horloge sur un circuit intégré.

4.1.1.2 *FuzeSoc*

FuzeSoc est un système de compilation et un gestionnaire de dépendances pour la conception matérielle. Le but de *FuzeSoc* est de favoriser la réutilisation des composants matériels et de faciliter la conception des SoC. Les fichiers de conception (*Verilog* ou *VHDL*) n'ont pas à être modifiés pour être utilisé avec FuzeSoc, par contre il est nécessaire d'adjoindre à chaque module *Verilog* un fichier de description d module dans un format qui rappel beaucoup le *YAML*. Ces fichiers de description contiennent principalement des métadonnées (auteur, date, nom, origine du cœur, dépendances, outils de synthèse supportés, etc.) mais aussi des informations sur les ports d'entrées/sorties.

Enfin, il est possible d'intégrer des générateurs de composants avec FuzeSoc. Ces générateurs sont des programmes en ligne de commande acceptant en argument un fichier indiquant les propriétés du module à générer et ils créent en sortie un module FuzeSoc (la description du module et son implémentation). Les générateurs sont la fonctionnalité la plus intéressante de FuzeSoc car ils peuvent être listés dans les dépendances d'un module. Parmi les cas d'usages des générateurs on retrouve beaucoup de composants qui sont pénibles à implémenter en *Verilog* car ils sont très

répétitifs : la génération des bus de connexions (*interconnect*), la génération de blocs de RAM, la génération de franchiseurs de domaine d'horloge, etc.

Cependant, nous avons choisi de maintenir l'implémentation actuelle en *Verilog* pur. En effet, basculer sur Litex ou FuzeSoc est une solution intéressante mais en l'absence du FPGA final j'aurai livré un travail non testé sur matériel ce qui le rend inutile.

4.1.2 *Stratégie de migration et choix de conception*

La migration d'*Eric* a lieu de l'architecture *Virtex 6* vers l'architecture *Artyx 7*. Avant de commencer le processus de migration j'ai donc commencé par lister les changements nécessitant une adaptation :

Les IP-Core sont des cœurs propriétaires fournis par les fabricants de FPGA permettant de s'interfacer avec différents modules (RAM, PCIE, mémoire flash, etc.). Ils facilitent le travail de l'ingénieur.

— Le cœur PCIE propriétaire (*IP-Core*) de Xilinx a changé d'interface et est passé sur le standard AXI.
— La carte est totalement différente : il n'y a plus de port Ethernet, pas d'écran, pas de sortie vidéo. On retrouve sur la nouvelle carte seulement un connecteur PCIE au format M.2, une TPM et un bloc de RAM.
— L'outil de développement fourni par Xilinx pour l'architecture *Virtex 6* a été changé lors du passage à *Artyx 7*. En effet le logiciel *ISE* a été remplacé par le logiciel *Vivado*.
— Quelques primitives provenant de la bibliothèque Xilinx ont évoluées et nécessite une adaptation.

4.1.2.1 *Migration d'ISE à Vivado*

Afin de faciliter le travail sur le projet, une solution à base de *Makefile* a été conçue pour pouvoir se séparer de l'interface graphique d'*ISE*. Cette volonté de minimiser le temps passé sur l'interface graphique viens du fait qu'historiquement les logiciels fournis par Xilinx sont assez lourds à manipuler, peu ergonomiques, lents et ont une fâcheuse tendance à s'arrêter brutalement. Comme les différentes fonctionnalités proposées par le logiciel sont aussi exposées sous la forme d'outils en ligne de commande il est possible d'automatiser les différentes étapes de l'élaboration du *bitstream* et de concevoir un environnement de travail totalement découplé d'*ISE*. Cependant Xilinx a fait le choix de supprimer ces outils en ligne de commande lors de la migration vers *Vivado*. La première étape de la migration a donc été de concevoir un nouveau système permettant de se détacher au maximum de l'interface graphique.

Le bitstream est le produit final d'un projet visant à être programmé sur FPGA.

Vivado propose un système de script base sur le langage *TCL*. Chaque interaction sur l'interface graphique se traduit en une commande *TCL*. Grâce à ce système il est possible d'utiliser uniquement le moteur de script de *Vivado* et de se détacher de l'interface graphique. J'ai donc conçu un script *TCL* et adapté les *Makefile* afin de maximiser l'ergonomie du projet. Pour cela, lors de la compilation, un projet *Vivado* est généré dans un dossier temporaire puis *Vivado* est démarré en mode script afin de transformer

les fichiers sources en un bitstream prêt à être envoyé sur le FPGA. Cette solution n'est pas parfaite car il faut lancer l'interface graphique de *Vivado* mais en l'état actuel des choses il est impossible de faire mieux. Ce travail a aussi été l'opportunité de documenter le fonctionnement des *Makefile* existants.

4.1.2.2 *Interfaçage avec l'IP-Core PCIE*

Dans le SoC on trouve un composant nommé *Host Memory Access* (HM) qui est chargé de gérer la partie communication PCIE. Cela inclut la gestion de la lecture et l'écriture sur les zones mémoires exposées par *Eric*, la gestion des lectures de la mémoire de la machine hôte et enfin la gestion des registres de configuration. Pour cela, le module s'appuie sur le cœur propriétaire Xilinx qui s'occupe des couches physiques et liaison de donnée du modèle de communication de PCIE (voir la Figure 4.4). HM n'a plu qu'à traiter les TLP[3] transmis par l'IP-Core. Dans la centaine de signaux exposés par l'interface de l'IP-Core, tous ne sont pas intéressant pour le module *HM* et seul une quinzaine sont vraiment utilisés.

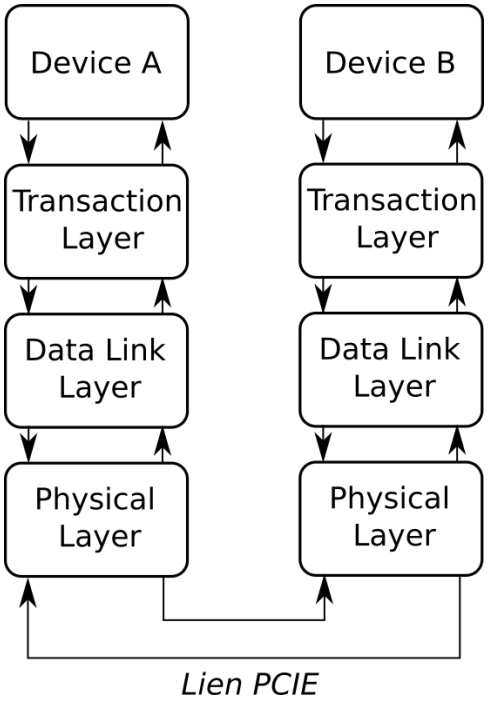


FIGURE 4.4 – Le modèle en couche de PCIE

Ma démarche initiale a été de porter intégralement le module *HM* vers la nouvelle interface AXI exposée par l'IP-Core, ce qui a entraîné un remaniement important de tous les modules interagissant avec le module *HM*. C'est une approche prône aux erreurs. En l'absence d'une suite de tests complète j'ai finalement préféré une approche modifiant le moins de code possible.

3. Transaction Layer Packet

J'ai décidé de développer une couche d'adaptation qui transforme les signaux AXI vers l'ancienne interface en me basant sur le manuel fourni par Xilinx. Cette approche est bien plus intéressante car elle permet de garder le support de l'ancien FPGA, elle évite de modifier le code existant et entraîne donc moins de bogues tout en étant plus rapide à implémenter. La Figure 4.5 présente le nouveau module et ses interactions avec l'IP-Core Xilinx. Ce travail de mise à jour a été accompagné par un rafraîchissement des tests afin de mieux tester certaines subtilités de l'implémentation tels que le respect du boutisme des données, l'ordre des doubles-mots dans les TLP ou encore le bon usage des signaux contrôlant l'IP-Core.

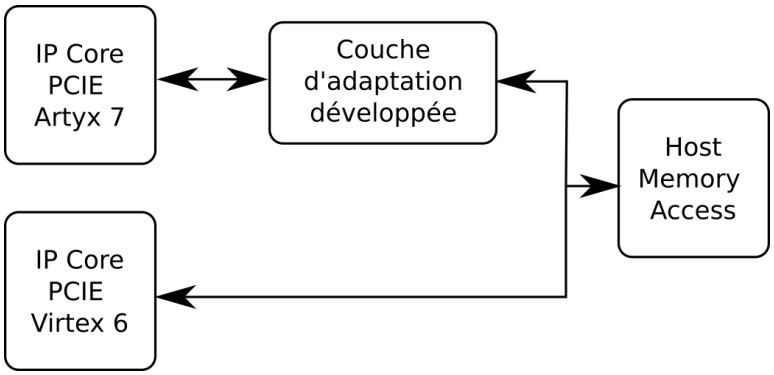


FIGURE 4.5 – La couche d'adaptation du cœur PCIE Artyx 7

4.1.2.3 *Autres travaux de migration*

Afin de terminer la migration j'ai aussi mis à jour les primitives utilisées pour générer la mémoire du module *HM*. La plupart des FPGA embarquent des composants courant utilisés dans les circuits logiques (multiplicateurs, diviseurs, blocs de RAM, etc.). Ils sont interconnectables avec les cellules logiques composant le tissu du FPGA. Cependant, dans l'implémentation d'un module ces blocs proviennent des librairies propriétaires fournies par le fabricant du FPGA. Ainsi, ces types sont dépendants de l'architecture cible et nécessitent un remaniement en cas de changement d'architecture de FPGA.

Comme *HM* expose de la mémoire par PCIE, les blocs de RAM sont utilisés dans l'implémentation du module ont nécessité une mise à jour afin de supporter la nouvelle interface exposée par Xilinx.

En l'absence de matériel, mon travail de modernisation s'est arrêté ici. J'ai cependant proposé d'utiliser un micro-contrôleur pour effectuer les mesures, la prochaine section présente donc ce travail.

4.2 DÉVELOPPEMENT D'UN PÉRIPHÉRIQUE CONFIANCE SUR MICRO-CONTRÔLEUR

Devant rendre ce document avant la réception du matériel nécessaire à la validation de l'implémentation je ne peux présenter que la partie conception et implémentation.

Le passage sur micro-contrôleur présente de nombreux avantages par rapport au FPGA. Il est tout d'abord motivé par un souci de simplicité,

car maintenir une implémentation sur FPGA est un travail complexe et minutieux nécessitant des compétences de spécialiste. La programmation sur micro-contrôleur est nettement plus simple car la partie matérielle est déjà réalisée. Par conséquent les itérations sur micro-contrôleur sont plus rapides à implémenter et moins coûteuses. De plus, un micro-contrôleur est beaucoup moins cher qu'un FPGA (5€ pour l'un, 500€ pour l'autre). Cependant, en l'absence de micro-contrôleurs grand public proposant un point d'accès PCIE, il est nécessaire de se tourner vers des puces dédiées offrant une interface série/parallèle et gérant la partie PCIE comme le composant MCS9901 ou d'accepter de perdre la capacité d'effectuer des lectures mémoires directement depuis le périphérique de confiance.

Afin d'implémenter le périphérique de confiance j'ai choisi d'utiliser des technologies que je maîtrisais déjà à travers mon expérience associative au club robot de l'INSA. En effet, j'ai choisi d'utiliser des micro-contrôleurs *stm32f103* qui sont ceux utilisé en cours à l'INSA et au club robot. J'ai choisi de réaliser la partie programmation en *Rust* car j'ai déjà eu de une bonne expérience avec ce langage de programmation au club robot. *Rust* offre de nombreux avantages par rapport à une implémentation en C, comme l'absence d'erreur de mémoire garantie à la compilation, la présence d'un gestionnaire de dépendance permettant d'importer des librairies conçues par la communauté ou encore un écosystème de développement moderne (compilateur, *linter*, documentation, etc.).

FIGURE 4.6 – Le contrôleur réseau W5500

Les micro-contrôleurs *stm32f103* ne proposant pas de périphérique supportant *Ethernet* nativement il est nécessaire de se tourner vers un composant dédié. J'ai choisi d'utiliser la puce W5500 qui est un contrôleur réseau supportant les protocoles TCP/UDP/IPv4/ICMP/ARP/IGMP. Le contrôleur réseau communique avec le micro-contrôleur grâce à un bus de communication SPI[4]. Côté logiciel embarqué j'utilise une librairie proposant une implémentation du protocole de communication indiqué dans la *datasheet* du fabricant ce qui m'offre un gain de temps conséquent. Enfin, afin de communiquer à la fois avec le serveur d'attestation et l'hyperviseur j'utilise deux interfaces réseau *w5500*.

Le programme embarqué sur le micro-contrôleur contient peu de code car les tâches à réaliser sont assez simples. Après avoir configuré le matériel, le micro-contrôleur traite toutes les trames réseaux reçues et utilise une machine à états finis pour déterminer l'action adéquate. La mesure de temps est implémentée à l'aide d'une minuterie programmée pour déclencher une interruption toutes les micro-secondes. À chaque interruption, un compteur est incrémenté et permet d'obtenir le nombre de micro-secondes écoulées depuis le démarrage du micro-contrôleur. La seule donnée nécessitant

4. Serial Peripheral Interface

d'être protégée des effets de la concurrent est la variable comptant le nombre de micro-secondes. Les accès à celle-ci sont donc protégés par un *mutex*. Le verrouillage du *mutex* entraîne l'entrée dans une section critique qui désactive les interruptions. À la sortie de la section critique elles sont réactivées.

Pour conclure, le système de type de *Rust* m'a beaucoup facilité l'implémentation de la machine à états utilisée pour la logique du micro-contrôleur. De même, il m'a permis de garantir une initialisation correcte des périphériques du micro-contrôleur.

4.3 CONCEPTION D'UNE ÉPREUVE CRYPTOGRAPHIQUE POUR L'ARCHITECTURE DE SÉCURITÉ

Dans l'architecture de sécurité, afin de vérifier l'intégrité d'*Abyme*, on utilise une épreuve cryptographique. Cette section définie les propriétés que doit vérifier cette épreuve, puis une implémentation est proposée en se basant sur un état de l'art des travaux académiques réalisés durant les 15 dernières années. Enfin, la viabilité de ce modèle d'attestation à distance sera vérifiée par des mesures expérimentales.

4.3.1 *Formalisation des propriétés*

L'épreuve se présente sous la forme d'un programme informatique produisant un résultat (la solution). Elle doit permettre de vérifier que l'hyperviseur de sécurité s'exécute toujours sur l'ordinateur sans modifications. Pour cela, le résultat dépend de valeurs caractéristiques d'*Abyme*, telles que les zones mémoires contenant les instructions de l'hyperviseur, la configuration de la MMU ou encore le niveau de privilège dans lequel s'exécute l'hyperviseur. Cependant, comme un hyperviseur malveillant peut utiliser les extensions matérielles de virtualisation pour se camoufler ; par exemple *Nested VMX* peut permettre de faire croire à l'épreuve qu'*Abyme* est toujours *VMX Root* alors qu'un hyperviseur malveillant est présent et virtualise *Abyme*. Il est donc nécessaire de trouver un deuxième mécanisme pour compléter l'épreuve. Nous avons décidé d'utiliser la mesure du temps d'exécution comme canal auxiliaire afin de compléter la solution fournie par l'épreuve. Afin de réussir l'épreuve, un candidat doit donc trouver le bon résultat dans le temps imparti.

À l'aide du travail réalisé par les auteurs de [2] et de [17], il a été possible de définir les propriétés suivantes sur l'épreuve.

Propriété 1 Le temps d'exécution de l'épreuve doit être prévisible, sinon il est impossible de mettre en place des bornes temporelles pour vérifier l'ordinateur.

Propriété 2 Afin d'éviter les attaques par rejeu, le résultat de l'épreuve doit dépendre d'un paramètre unique à chaque tentative (un *nonce*).

Propriété 3 La manière la plus rapide d'obtenir la solution de l'épreuve est de l'exécuter. Par conséquent, les calculs réalisés pendant l'épreuve ne doivent pas pouvoir être parallélisables. Pour cela, le résultat final dépend de l'ordre de calcul des résultats intermédiaires.

Propriété 4 Comme nous utilisons un générateur de nombres pseudos-aléatoires, il faut que les nombres générés soient uniformes et imprévisibles [2].

Propriété 5 De même, la somme de contrôle doit résister aux attaques par secondes pré-images affaiblies : il s'agit de résister aux attaques par seconde pré-image quand l'attaquant ne connais pas la pré-image [2].

Propriété 6 Un mécanisme d'auto-vérification est présent dans l'épreuve afin d'empêcher les attaques par ré-écriture du binaire comme décrit dans la sous-section 3.3.1. Pour cela l'épreuve s'auto-vérifie.

Enfin, l'épreuve a été conçue afin de pouvoir être étendue, par exemple en y intégrant les tests d'environnement décrits dans [17].

4.3.2 *Travaux précédents*

Le problème de l'attestation à distance est étudié depuis plusieurs dizaines d'années par la communauté scientifique. Un des premiers papiers fondateur présente une méthode nommée *Pioneer* [23]. Ce dernier propose un cadre formel pour le problème de l'attestation à distance (définition du problème, hypothèses et modèle d'attaquant) ainsi qu'une première solution sous la forme d'un protocole et d'une fonction de vérification. Les principales idées de *Pionneer* reprise dans l'implémentation de l'épreuve sont :

— L'épreuve vérifie l'intégrité de son propre code durant l'exécution et l'intègre à la solution.
— La somme de contrôle est fortement ordonnée, c'est-à-dire que le résultat de la somme de contrôle à une grande probabilité d'être différent si les opérations la composant sont exécutées dans un ordre différent.
— La somme de contrôle parcourt l'espace mémoire de manière pseudo-aléatoire : si l'adversaire veut substituer sa valeur à celle lue (pour se dissimuler) il doit vérifier chaque lecture mémoire ce qui se traduira par un coût en temps important et croissant avec le nombre de lecture mémoire.
— L'exécution du challenge est pseudo-aléatoire afin d'éviter le pré-calcul des solutions et leur rejeu.

Un autre papier ayant eu beaucoup d'influence sur mon implémentation est *Checkmate* [15]. En effet, les auteurs étendent *Pionneer* en proposant une implémentation et une architecture de sécurité utilisable en production. Notamment, on y retrouve une structuration de l'épreuve en blocs indépendants chacun vérifiant une partie du système. À la fin d'un bloc, un PRNG [5]

5. Pseudo Random Number Generator

est utilisé pour choisir le prochain bloc exécuté. Les auteurs proposent aussi d'intégrer le flot d'exécution de l'épreuve au résultat de l'éoreuve. Ainsi, des blocs dédiés sont utilisés pour vérifier le pointeur d'instruction du bloc appelant ainsi que du bloc appelant l'appelant de l'appelant (grand-père.). De même l'état du PRNG est intégré à la solution régulièrement.

Dans les approches intéressantes, mais que je n'ai pas conservées on trouve notamment *Atrium* [27] qui mesure à la fois la solution de l'épreuve, son temps d'exécution mais aussi son flot de contrôle à l'aide d'un composant matériel intégré au processeur. Cependant, comme il faut modifier le pipeline du processeur afin d'y intégrer un composant matériel je ne l'ai pas retenu. On trouve aussi *MT-SROT* [26] qui propose un algorithme pour mettre en place une épreuve sur plusieurs cœurs grâce à une division et une répartition du travail particulière. *MT-SROT* présente aussi un mécanisme anti-interruption permettant de s'assurer que l'épreuve s'exécute de manière ininterrompue : en utilisant la pile pour stocker les valeurs intermédiaires, on s'assure qu'elles seront ré-écrites en cas d'interruption, rendant la manipulation de l'épreuve par les interruptions impossible. Enfin, les auteurs de [12] expliquent les détails auxquels il faut faire attention dans le cas d'une implémentation multi-cœurs (non-déterminisme et gestion des cœurs). Pour finir, *HyperSentry* [3] est une brique logicielle résidant dans le mode SMM [6] du processeur et mettant en place un mécanisme de contrôle d'intégrité non-intrusif utilisant une TPM. Cependant aucune source n'est disponible et il est par conséquent impossible de ré-utiliser le travail des auteurs...

4.3.3 *Sélection des primitives cryptographiques*

À partir du travail réalisé par les auteurs de [2] j'ai obtenu les propriétés que doivent vérifier les primitives cryptographiques (les propriétés 4 et 5 de la sous-section 4.3.1).

Cependant je n'ai pas réalisé les preuves permettant de montrer que les primitives choisies respectent ces propriétés.

Il faut également que ces primitives soient suffisamment rapides pour que leur temps d'exécution soit négligeable devant le temps de calcul de l'épreuve. Enfin, comme dans l'architecture de sécurité initiale le binaire de l'épreuve était transmis à chaque attestation, la taille finale de l'épreuve était un critère à minimiser. Il fallait donc choisir des primitives dont l'implémentation est compacte.

4.3.3.1 *Génération de nombres pseudo-aléatoires*

Au vu de ces contraintes, utiliser un algorithme de hachage cryptographique comme générateur n'est pas envisageable car le temps d'exécution est trop important et l'implémentation trop lourde. Enfin, même l'utilisation d'un algorithme rapide non cryptographique comme *XXHash* [5]

6. System Management Mode

n'est pas envisageable. En effet, pour des raisons de performance il est préférable que le code machine de l'épreuve tienne sur une seule page mémoire voir même une seule entrée de cache L1. Cela permet d'avoir un temps d'exécution plus déterministe que si des morceaux de l'épreuve doivent être récupéré en RAM régulièrement par le processeur. Au final, j'ai décidé de garder le PRNG utilisé par *Pionneer* et *Checkmate* qui est basé sur la *T-Function* [13] suivante :

$$f(x) = x + (x^2 \vee 5) \mod 2^n$$

En effet, l'implémentation assembleur de cette fonction est très légère et rapide. Il faudrait cependant prouver qu'elle respecte les propriétés énumérées en sous-section 4.3.3.

4.3.3.2 *Somme de contrôle*

Comme expliqué pour le PRNG (cf sous-sous-section 4.3.3.1) l'implémentation de la somme de contrôle doit être rapide, et le code machine doit être aussi petit que possible. Il faut aussi y intégrer le mécanisme de protection anti-interruption provenant de *MT-SROT* [26]. Enfin, il faut que la somme de contrôle soit fortement ordonnée afin que le résultat de l'épreuve dépende non seulement des données d'entrée, mais aussi de l'ordre d'exécution des blocs de l'épreuve. Cela permet d'éviter qu'un attaquant puisse manipuler l'épreuve à son avantage en pré calculant des morceaux. L'implémentation utilisée est basée sur une succession de xor et d'addition, qui sont combinés de manière non symétriques :

$$\text{Résultat} = x_1 \oplus (x_2 + (x_3 \oplus (\cdots + x_i)))$$

Il s'agit de la somme de contrôle proposé dans le papier *Pionneer* que j'ai choisi pour sa simplicité et la compacité de son implémentation.

4.3.4 *Implémentation de l'épreuve sur x86_64*

Dans notre architecture de sécurité l'attestation à distance est mise en place ainsi :

1. Le superviseur contacte le périphérique de confiance afin de demander une preuve d'intégrité.
2. Le périphérique génère un *nonce* aléatoire, fixe un nombre d'itérations pour l'épreuve et pré-calcule la solution de l'épreuve.
3. Les paramètres de l'épreuve sont envoyés à *Abyme* et un chronomètre est lancé.
4. *Abyme* exécute l'épreuve avec les paramètres fournis et envoie la solution le plus rapidement possible au périphérique de confiance.
5. Si la solution est valide et que le temps de calcul est dans l'intervalle de confiance, le périphérique de confiance indique au superviseur que l'attestation est réussie, sinon c'est un échec.

Ce fonctionnement est dessiné sous la forme d'un diagramme de séquence sur la Figure 4.7.

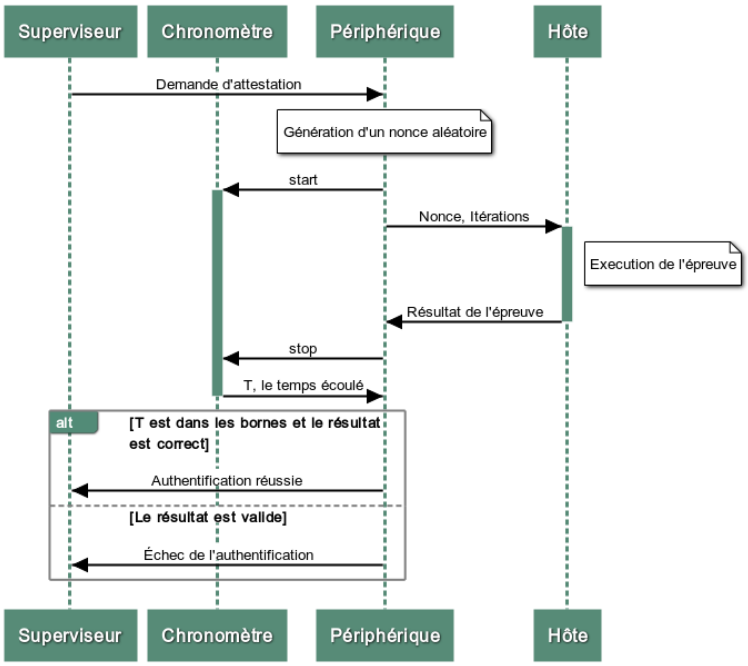


FIGURE 4.7 – Diagramme de séquence d'une attestation à distance

L'épreuve est divisée en bloc, vérifiant chacun une partie du système. Le code source de l'implémentation est disponible en Annexe B. Chaque bloc est composé d'un prologue qui ajoute la valeur du registre RIP à un accumulateur. Ensuite, les données des vérifications sont intégrées à l'accumulateur. Enfin, l'épilogue s'occupe d'intégrer l'accumulateur à l'une des quatre entrées de la somme de contrôle puis de sauter dans le bloc suivant de manière pseudo-aléatoire (voir Figure 4.8).

Le registre DR7 est le registre contrôlant le fonctionnement des points d'arrêt pour le débogage.

Bloc 0 : Ce bloc vérifie la valeur contenue dans le registre *RFLAGS* ainsi que *DR7*.

Bloc 1 : Ce bloc utilise l'instruction CPUID afin de ralentir les attaquants essayant de se camoufler en virtualisant *Abyme*. Le résultat de l'instruction est aussi vérifié.

Bloc 2 : Ce bloc ajoute l'état du PRNG à la somme de contrôle.

Bloc 3 : Ajoute une adresse aléatoire comprise entre le début et la fin du code de l'épreuve au résultat.

Bloc 4 : Ajoute à la somme de contrôle l'adresse de chargement de l'épreuve.

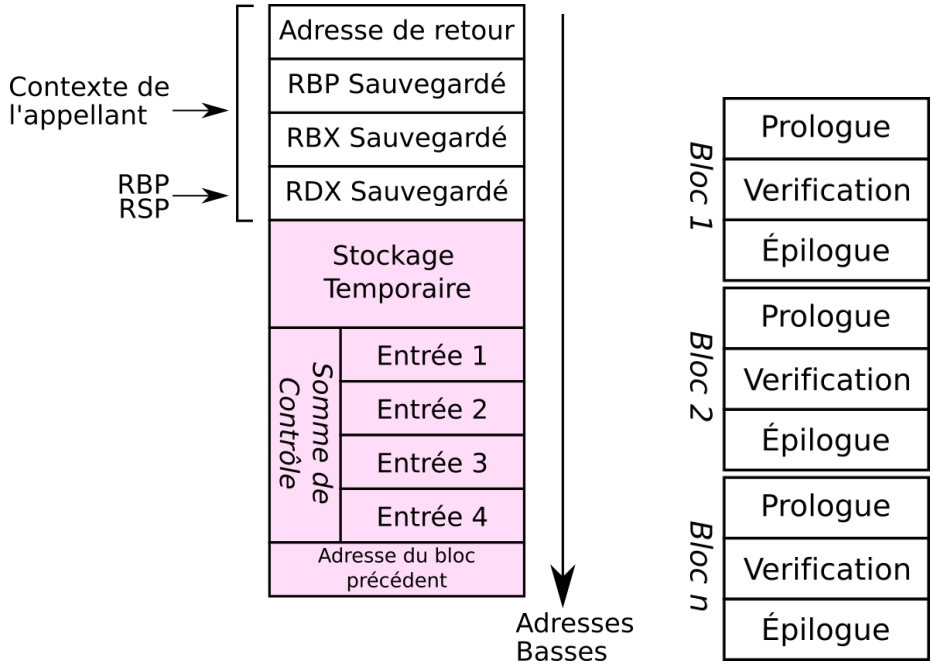
Bloc 5 & 6 : Ajoute à la somme de contrôle la valeur lue à une adresse aléatoire dans l'espace mémoire où est chargé l'épreuve.

Bloc 7 : Ajoute l'adresse de l'appelant ou du père de l'appelant au résultat.

Afin de se protéger des interruptions, j'utilise la pile pour stocker les résultats intermédiaires (cf Figure 4.8) cependant je porte une attention particulière au fait que RSP et RBP se trouvent au dessus des résultats

intermédiaires. Comme l'épreuve est exécutée en ring 0, l'arrivée d'une interruption entraîne la sauvegarde du contexte en-dessous de RSP, ce qui écrase les résultats intermédiaires. En effet l'épreuve s'exécute en mode 64 bits ring 0 et le processeur utilise la pile courante pour sauvegarder le contexte. Ainsi, un attaquant ne peut se baser sur les interruptions pour manipuler l'exécution de l'épreuve.

Sur x86_64 la pile descends vers les adresses basses.



(a) Structure de la pile. En violet, les données qui sont écrasées lors de l'arrivée d'une interruption

(b) Structure de l'épreuve

FIGURE 4.8 – Structure de l'épreuve et de la pile durant son exécution

4.3.5 *Tests, mesures et validation*

Afin de valider la conception et l'implémentation de l'épreuve j'ai réalisé des tests d'abord sur machine virtuelle puis sur machine réelle. Pour cela, l'épreuve est exposée sous la forme d'une librairie C ce qui permet de l'inclure dans un binaire Linux standard, mais aussi dans un module noyau ou encore sous la forme d'une application *EFI*. Afin d'effectuer les mesures dans des conditions qui se rapprochent le plus possible de la réalité, les mesures ont été réalisées à travers une application *EFI* car :

— L'épreuve s'exécute alors en ring 0 dans le même contexte que l'hyperviseur
— Afin d'effectuer les mesures dans un contexte virtualisé, *Abyme* a été utilisé, et le chargement d'une application *EFI* est le moyen le plus simple d'exécuter un programme dans ce contexte

J'ai réalisé 4 séries de mesures différentes visant à mettre en évidence l'efficacité de l'instruction *CPUID* pour détecter la virtualisation de l'hyperviseur. Pour cela, j'ai mesuré à chaque fois le temps d'exécution de

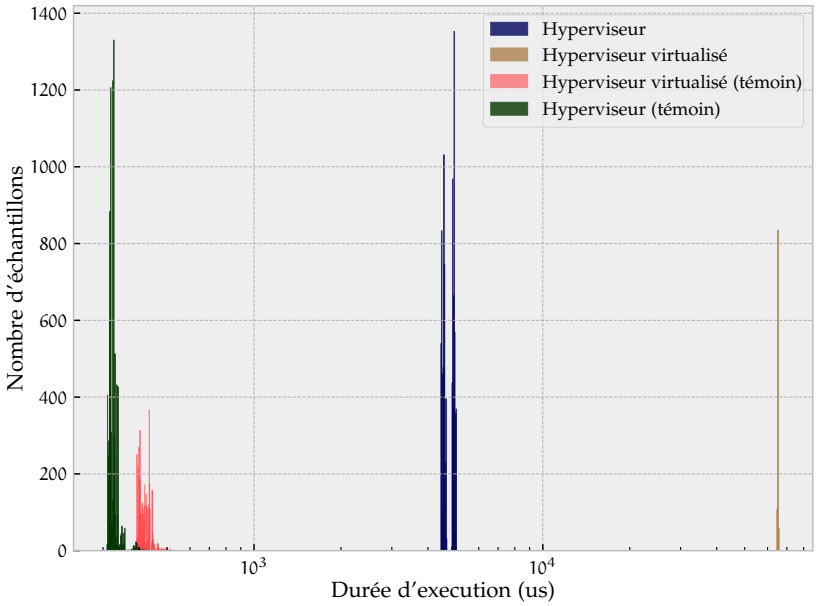


FIGURE 4.9 – Distribution du temps d'exécution de l'épreuve dans un contexte virtualisé et VMX Root

l'épreuve avec la présence de l'instruction *CPUID* et sans la présence de cette instruction (expérience témoin). Ces deux mesures ont été réalisées à la fois dans un contexte virtualisé et dans un contexte *VMX Root*. On obtient la Figure 4.9 (l'axe des abscisses utilise une échelle logarithmique). Chaque série de mesure comporte 10 000 échantillons, sauf la série virtualisée avec l'instruction *CPUID* qui n'en comporte que 1 000 car le temps d'exécution de l'épreuve est plus long. L'épreuve est exécutée avec 65 535 itérations et toujours le même *nonce*.

On peut voir sur cette distribution que l'utilisation de l'instruction *CPUID* au sein de l'épreuve entraîne bien une différence de temps mesurable (un ordre de grandeur de différence). De plus, les deux mesures témoins montrent que cela provient bien de l'instruction *CPUID* et non des effets temporels de la virtualisation.

Enfin, une intégration a été réalisée dans *Abyme*, cependant des difficultés techniques sont apparues, notamment des régressions, au cours du stage et à cette date elle ne marche plus. J'ai aussi eu beaucoup de mal à m'approprier le code de l'hyperviseur.

4.3.6 *Limites de l'épreuve*

Les choix de conception et d'implémentation réalisé pendant ce stage ont permis d'atteindre les objectifs techniques, cependant ils sont aussi limités. En effet, comme la preuve d'intégrité est basée sur le temps d'exécution, il faut que l'implémentation de l'épreuve soit la plus optimale possible. Cependant, outre le fait qu'il est impossible de déterminer qu'une

implémentation est optimale, il est nécessaire de tenir compte de la micro-architecture du processeur sur lequel l'épreuve s'exécutera afin de s'assurer que le processeur est utilisé au maximum. Autrement un attaquant pourrait gagner du temps en utilisant une implémentation tirant pleinement parti du processeur. Cela impose donc de connaître la micro-architecture du processeur ciblé et d'avoir une implémentation optimisée pour celle-ci.

De plus, la preuve d'intégrité se base sur la vérification d'un certain nombre de propriétés. Cependant il est difficile d'évaluer la couverture qu'offrent ces vérifications et un travail plus important est nécessaire afin d'identifier tous les attributs du système devant être vérifiés.

Enfin, avec le passage sur micro-contrôleur nous avons perdu la possibilité de lire directement la mémoire de la machine attestée depuis le périphérique de confiance. Ce sont donc des vérifications qui doivent être intégrées à l'épreuve, rallongeant la durée d'exécution.

5 CONCLUSION

Bien que ce stage se soit déroulé dans le contexte unique de la crise sanitaire du *COVID-19* je ressors satisfait des différents travaux que j'ai accompli et des évolutions apportées à l'architecture de sécurité : le portage du périphérique de confiance sur la nouvelle architecture *Artyx 7*, l'adaptation sur micro-contrôleur et la formalisation de l'épreuve cryptographique. Travailler sur l'hyperviseur *Abyme* est difficile, car il s'agit d'une technologie que je maîtrise mal, exploitée dans un projet de recherche avec le peu de documentation que cela implique. J'aurais aimé pouvoir arriver au bout des objectifs que nous nous étions fixés, hélas les aléas de l'approvisionnement et la pandémie de *COVID-19* ont fait que je n'ai pu tester ni le FPGA, ni l'implémentation sur micro-contrôleur. Ayant passé la majeure partie du stage en télé-travail, il est indéniable que cela a eu un impact négatif sur mes réalisations car je n'ai pu avoir un encadrement aussi efficace qu'en présentiel. Les quelques jours que j'ai passés en laboratoire en Juillet ayant été très productif, j'aurai pu pousser les accomplissements techniques plus loin dans le contexte adéquat.

Après ce stage, l'équipe ACADIE dispose désormais des premières briques fonctionnelles afin d'assembler un prototype permettant de réaliser des démonstrations de leur architecture de sécurité. Ce stage est donc un premier pas vers un potentiel transfert de technologie, même s'il reste du travail à réaliser pour finaliser un prototype.

Grâce à ce stage en laboratoire j'ai pu découvrir le monde de la recherche, bien que d'une manière très particulière. Cela m'a permis d'alimenter mon projet professionnel et d'affiner la direction que je souhaite donner à ma carrière. En effet, bien que j'ai apprécié la liberté dont j'ai disposé durant mon stage, je me suis rendu compte que la direction politique que prend la recherche publique ne correspond pas à mon besoin d'engagement social et humain dans mon travail.

Troisième partie

ANNEXES

A

SCRIPT DE DÉMARRAGE UEFI

```
echo Startup !
fs1:
# Chargement du driver ethernet
cd \EFI\drivers\82579LM
echo Loading ethernet driver
echo press any key
load 82579LM.efi
echo Ethernet runtime driver loaded !
# Chargement de l'hyperviseur
cd \EFI\drivers\vmm_rec_env
echo Loading recursive vmm
echo press any key
load vmm_rec_env.efi
echo Recursive vmm runtime driver loaded !
pause
```

FIGURE A.1 – Script de démarrage UEFI permettant de charger le pilote réseau puis *Abyme*

B

IMPLÉMENTATION DE L'ÉPREUVE D'ATTESTATION À DISANCE

```
# Checkmate challenge extension to support VMX-Root detection
#
# Challenges are made of a pseudo-random execution of blocks checking various
# properties. Each block is structured in the same way :
# 1) Add RIP to the accumulator
# 2) Update the pseudo random number generator
# 3) Perform checks
# 4) Use those checks values to feed a checksum function
# 5) Pseudo-randomly jump to the next block until the provided number of
#        iterations has been reached
#
#
# Registers usage :
# * R9  : Loop counter
# * R10 : Accumulator
# * RSI : PRN Value
#
#
# Stack structure :
#
#              +- - - - - - - - - - - - - - - - -+ +
#                                                 |
#              |                ...              | |
#                                                 |
#              +---------------------------------+ |
#          +-> |Return Address (grand parent RIP) | |
#          |   +---------------------------------+ |
#          +-> |Saved RBP                        | |
# Caller --+   +---------------------------------+ |
# Context  +-> |Saved RBX (ABI)                  | |
#          |   +---------------------------------+ |
#          +-> |Saved RDX (@result)              | |
#              +---------------------------------+ |
#    0x18(RBP) |Checksum Entry 1                 | |
#              +---------------------------------+ |
#    0x10(RBP) |Checksum Entry 2                 | |
#              +---------------------------------+ |
#    0x08(RBP) |Checksum Entry 3                 | |
#              +---------------------------------+ |
#    RBP +---> |Checksum Entry 4                 | |
#              +---------------------------------+ |
#    RSP +---> |Return Address (parent RIP)      | |
#              +---------------------------------+ |
#                                                  v
#                                                Low
#                                             Addresses
.section .text

# The challenge entry point is the symbol <__challenge_start>
.globl __challenge_start
.type __challenge_start, "function"

.start_addr :

# Challenge entry point (3 arguments)
# rsi : number of iterations
# rdi : prng seed
# rdx : pointer to an array of 4 dqword which will hold the checksum results
__challenge_start:
        xorq %rax, %rax
        movq %rsi, %r9                          # Fetch loop counter
              from arguments
        movq %rdi, %rsi                         # Move seed to PRN
              register
        pushq %rbp                                      # Save RBP on
              stack to restore after challenge execution (this eases
              address calculation)
        pushq %rbx                                      # Save context
        pushq %rdx                                      # Save result
              address on stack
        movq %rsp, %rbp # Set RBP above checksum. RBP should not change
              during
        # challenge execution

        # Add free space on stack to store temp variables (pushfq, etc.)
        pushq %rsi

        # Push nonce on the stack to initialize the checksum
        pushq %rsi
        pushq %rsi
        pushq %rsi
        pushq %rsi
        movq %rbp, %rsp                         # Set RSP above checksum
               so that any interrupts overrides
        # it. RBP is already above it.
        callq .block_0                          # We use call instead of
               jump so that RIP is pushed on the stack


# Pop caller RIP and add it to the accumulator
.macro block_prolog
        add (%rsp), %r10                        # Add caller RIP to
              accumulator
        add $8, %rsp                            # Discard RIP
.endm

# Update the PRNG value (RSI)
#
# The PRNG is based on the following T function :
# x -> x + (x^2 v 5) mod 2^n
.macro update_prn
        mov %rsi, %rax
        mul %rax
        or $5, %rax
        add %rsi, %rax
        xor %r10, %rsi
.endm

# Add to the accumulator the address where the challenge code is loaded
.macro check_challenge_load_addr
        lea (%rip), %rbx                        # RBX <- Current address
        addq $-(. - .start_addr), %rbx  # RBX += offset to challenge
              start

        # Combine the results
        addq %rbx, %rsi
        addq %rax, %rsi
        addq %rax, %rbx
        addq %rbx, %rsi
.endm


# Generate a random address within the range of the memory region where the
# challenge is loaded
#
# The resulting value is found in RDI
.altmacro
.macro generate_challenge_addr nb
        movq %rsi, %rax                         # RAX <- SEED
        movq %rsi, %rbx                         # Backup PRN value
        xorq %rdx, %rdx                         # Zero extend RAX
        movq $(.end_addr - .start_addr), %rsi           # RSI <- Address
              range
        divq %rsi                                       # RAX = Quotient
              (SEED /RSI) | RDX = Remainer(SEED /RSI)
.challenge_addr_\nb:
        lea (%rip), %rsi # Mov RIP into RSI
        subq $(.challenge_addr_\nb - .start_addr), %rsi

        leaq (%rsi, %rdx), %rax         # RAX = Challenge start address
              + random point
        # in 0..chal_len (exclusive)
```

```
        movq %rax, %rdx
        movq %rbx, %rsi                         # Restore PRN value
.endm

# Add to the accumulator a random address in the range
# challenge_start...challenge_end (exclusive)
.altmacro
.macro check_challenge_addr nb
        generate_challenge_addr nb
        xorq %rdx, %r10                         # Add address to
              accumulator
.endm

# Add to the accumulator the content of a random address in the range
# challenge_start..challenge_end (exclusive)
.altmacro
.macro check_challenge_addr_val nb
        generate_challenge_addr nb
        addq (%rdx), %r10                       # Add address value to
              accumulator
.endm

# Add DR7 and RFLAGS to the accumulator
.macro check_regs
        /* DR7 (TODO : needs ring 0)
        mov %dr7, %eax
        addq %rax, %r10 */
        # RFLAGS
        pushfq                                          # Push RFLAGS on
              the stack (64 bits)
        addq (%rsp), %r10                       # Mix RFLAGS with the
              accumulator
        addq $8, %rsp                           # Restore the stack
.endm

# Add the prng value to the accumulator
.macro check_prng
        addq %rsi, %r10
.endm

# Add the father RIP to the accumulator
.macro check_father
        test %r10, %r10
        movq -0x8(%rsp), %rax           # RAX = father RIP
        jp .father                                      # Use parity bit
              of the accumulator to choose between the father and the
              grand-father
        movq 0x18(%rsp), %rax           # RAX = grandfather RIP
.father:
        xorq %rax, %rsi                         # Mix father/grand-
              father RIP with PRN
        addq %rsi, %r10                         # Add to the accumulator
.endm

# Calls CPUID with a random argument and adds the result to the
      accumulator
.macro check_cpuid
# Seed CPUID from the PRN : generate numbers between 0 and 25, map them
      to 0->16
# and 0x80000000->0x80000008
.get_cpuid_index:
        update_prn

        # rax = (prn >> 24 & 0x1F)
        movq %rsi, %rax
        andq $0x1F, %rax

        cmp $25, %rax                           # if RAX < 25 : OK
        jb .cpuid_idx_map
        andq $0x19, %rax                        # else, shrink it
.cpuid_idx_map :
        # If rax < 16 use it as is
        cmp $16, %rax
        jbe .cpuid_idx_done
        addq $0x7FFFFFEF, %rax # The magic is 0x80000000- 17
        .cpuid_idx_done :
        mov %rax, %r11                          # Backup RAX
        cpuid

        # Some cpuid bits need masking when RAX == 0x1 or RAX == 0xb
        cmpq $1, %rax
        je .cpuid_idx_0x1
        cmpq $0xb, %rax
        jne .cpuid_finish

# cpuid_idx == 0xb -> RDX = RDX & ~(0xF)
.cpuid_idx_0xb:
        andq $-16, %rdx
        jmp .cpuid_finish

# cpuid_idx == 0x1 -> RBX = RBX & 0xFFFFF
.cpuid_idx_0x1:
        andq $0xFFFFF, %rbx
        jmp .cpuid_finish

.cpuid_finish:
        # Process results : r11 = ((RAX ^ RBX) + RCX) ^ RDX
        movq %rax, %r11
        xorq %rbx, %r11
        addq %rcx, %r11
        xorq %rdx, %r11
        addq %r11, %r10 # Add result to the accumulator
.endm

# Update the checksum using data from the accumulator (R10)
#
# 1) Select one of the 4 checksum slot using the loop counter
# 2) Xor the slot value with the accumulator
# 3) Bit shift all entries to the right by 1, using the lsb of the
     previously
#        updated entry as the carry
.macro update_chk
        mov %r9, %rax                           # RAX <- r9 (loop
              counter)
        and $3, %rax                            # Masks bottom bits to
              use as an index (4 values)
        addq $-0x28, %rbp
        xorq %r10, (%rbp,%rax,8)        # [RBP + RAX * 8] ^= r10 :
              update the given checksum entry with the accumulator, cf
              stack structure diagram
        bt $1, (%rbp, %rax, 8)          # Use bottom bits of the updated
               checksum entry to indicate carry value for the following
              bit shifting
        addq $0x28, %rbp

        # Update checksum values through bitshifting
        rcrq $1, -0x10(%rbp)
        rcrq $1, -0x18(%rbp)
        rcrq $1, -0x20(%rbp)
        rcrq $1, -0x28(%rbp)
.endm


# Holds the loop logic and jumps to the next block if we are not done
.macro epilog
        sub $1, %r9                                     # Decrement loop
              counter
        test %r9, %r9
        jz .finish
        jmp __call_next_block
.endm

# This macro calls `block_<nb>` if $rbx == <nb>
# Otherwise it does nothing
.altmacro
.macro jump_to_block_if nb
        cmp $\nb, %rbx
        jne .block_case_\nb
        call .block_\nb
.block_case_\nb :
.endm

# Call the next block
__call_next_block :
        mov %rsi, %rbx                          # PRN is used as a seed
        and $7, %rbx                            # Mask : we need 3 bits
              as we only have 7 blocks

        # """Switch""" on rbx to jump to the next bloc
        jump_to_block_if 0
        jump_to_block_if 1
        jump_to_block_if 2
        jump_to_block_if 3
        jump_to_block_if 4
        jump_to_block_if 5
        jump_to_block_if 6
        jump_to_block_if 7

# This macro generates the assembly code for a challenge bloc
#
# The bloc is labelled as block_<block_number>, and it will perform the
     checks
# provided as a macro argument.
# The prolog and epilog of the bloc are automatically added, ie the code
      to :
# -> mix RIP with the accumulator
# -> update the PRNG
# -> update the checksum
# -> jump to the next bloc
# is added to the check code
```

```
.altmacro
.macro make_block block_number check
        .begin_\block_number:
        .block_\block_number:
        block_prolog
        update_prn
        \check
        update_chk
        epilog
.endm

# This is where the assembly code is generated
make_block 0 check_regs
make_block 1 check_cpuid
make_block 2 check_prng
make_block 3 check_father
make_block 4 <check_challenge_addr 4> # We use <> to use checks have
     arguments
make_block 5 <check_challenge_addr_val 5>
make_block 6 <check_challenge_addr_val 6>
make_block 7 check_challenge_load_addr

# Clean up the challenge environnement, store the results and restore
     context
.finish :
        movq %rbp, %rsp
        popq %rdx                                       # Pop result
             addr from the stack

        # Copy checksum values to the provided address
        movq -0x10(%rbp), %rax
        mov %rax, (%rdx)
        movq -0x18(%rbp), %rax
        movq %rax, 0x8(%rdx)
        movq -0x20(%rbp), %rax
        movq %rax, 0x10(%rdx)
        movq -0x28(%rbp), %rax
        movq %rax, 0x18(%rdx)
        popq %rbx                                       # Restore RBX
        popq %rbp                                       # Restore RBP
        ret
        leave

.end_addr:
```

BIBLIOGRAPHIE

[1] *Aller Artix-7 FPGA Board with M.2 Interface*. en-US. Publication Title : Numato Lab. URL : https://numato.com/product/aller-artix-7-m-2-fpga-module (visité le 01/07/2020).

[2] Frederik ARMKNECHT, Ahmad-Reza SADEGHI, Steffen SCHULZ et Christian WACHSMANN. « Towards Provably Secure Software Attestation ». en. In : (2013), p. 19.

[3] Ahmed M. AZAB, Peng NING, Zhi WANG, Xuxian JIANG, Xiaolan ZHANG et Nathan C. SKALSKY. « HyperSentry : enabling stealthy in-context measurement of hypervisor integrity ». en. In : *Proceedings of the 17th ACM conference on Computer and communications security - CCS '10*. Chicago, Illinois, USA : ACM Press, 2010, p. 38. ISBN : 978-1-4503-0245-6. DOI : 10.1145/1866307.1866313. URL : http://portal.acm.org/citation.cfm?doid=1866307.1866313 (visité le 09/06/2020).

[4] Sébastien BOURDEAUDUCQ. « A performance-driven SoC architecture for video synthesis ». Anglais. Mém. de mast. Stockholm : Royal Institue of Technology, 2010. URL : http://m-labs.hk/thesis/thesis.pdf (visité le 01/07/2020).

[5] Yann COLLET. *Cyan4973/xxHash*. original-date : 2014-04-30T23 :32 :49Z. Juil. 2020. URL : https://github.com/Cyan4973/xxHash (visité le 22/07/2020).

[6] Louis COLUMBUS. *Public Cloud Soaring To $331B By 2022 According To Gartner*. en. Publication Title : Forbes. URL : https://www.forbes.com/sites/louiscolumbus/2019/04/07/public-cloud-soaring-to-331b-by-2022-according-to-gartner/ (visité le 11/06/2020).

[7] Michel DAYDÉ. *Présentation de l'IRIT*. fr-FR. Publication Title : IRIT. Juin 2020. URL : https://www.irit.fr/le-laboratoire/presentation-du-laboratoire/ (visité le 29/06/2020).

[8] *Développement sécurisé en Rust*. Fr. T. 1. Agence Nationale de Sécurité des Systèmes d'Informations (ANSSI), 2020. URL : https://anssi-fr.github.io/rust-guide/.

[9] *Intel® 64 and IA-32 Architectures Software Developer's Manual : System Programming Guide*. en. T. 3A, 3B, 3C, 3D. Intel, mai 2020. URL : https://software.intel.com/content/www/us/en/develop/download/intel-64-and-ia-32-architectures-sdm-combined-volumes-3a-3b-3c-and-3d-system-programming-guide.html.

[10] Florent Kermarrec, Sebastien Bourdeauducq, Jean-Christophe Le Lann et Hannah Badier. « LiteX : an open-source SoC builder and library based on Migen Python DSL ». en. In : *OSDA'2019 Open Source Hardware Design* (mai 2020), p. 6.

[11] S King et P Chen. « SubVirt : implementing malware with virtual machines ». In : IEEE, 2006. isbn : 0-7695-2574-1. doi : 10.1109/SP.2006.38.

[12] Michael Kiperberg, Amit Resh et Nezer J. Zaidenberg. « Remote Attestation of Software and Execution-Environment in Modern Machines ». en. In : *2015 IEEE 2nd International Conference on Cyber Security and Cloud Computing*. New York, NY, USA : IEEE, nov. 2015, p. 335-341. isbn : 978-1-4673-9300-3. doi : 10.1109/CSCloud.2015.52. url : http://ieeexplore.ieee.org/document/7371504/ (visité le 09/06/2020).

[13] Alexander Klimov et Adi Shamir. « A New Class of Invertible Mappings ». In : t. 2523. Août 2002, p. 470-483. doi : 10.1007/3-540-36400-5_34.

[14] Paul Kocher et al. « Spectre Attacks : Exploiting Speculative Execution ». In : *40th IEEE Symposium on Security and Privacy (S&P'19)*. 2019.

[15] Xeno Kovah, Corey Kallenberg, Chris Weathers, Amy Herzog, Matthew Albin et John Butterworth. « New Results for Timing-Based Attestation ». en. In : *2012 IEEE Symposium on Security and Privacy*. San Francisco, CA, USA : IEEE, mai 2012, p. 239-253. isbn : 978-1-4673-1244-8 978-0-7695-4681-0. doi : 10.1109/SP.2012.45. url : http://ieeexplore.ieee.org/document/6234416/ (visité le 09/06/2020).

[16] Moritz Lipp et al. « Meltdown : Reading Kernel Memory from User Space ». In : *27th USENIX Security Symposium (USENIX Security 18)*. 2018.

[17] Benoît Morgan. « Protection des systèmes informatiques vis-à-vis des malveillances : un hyperviseur de sécurité assisté par le matériel ». fr. PhD Thesis. INSA de Toulouse, 2016.

[18] Benoît Morgan, Éric Alata, Vincent Nicomette et Mohamed Kaâniche. « IOMMU protection against I/O attacks : a vulnerability and a proof of concept ». In : *Journal of the Brazilian Computer Society* 24.1 (jan. 2018), p. 2. issn : 1678-4804. doi : 10.1186/s13173-017-0066-7. url : https://doi.org/10.1186/s13173-017-0066-7.

[19] *Overview of Amazon Web Services*. English. Amazon, jan. 2020. url : https://d1.awsstatic.com/whitepapers/aws-overview.pdf (visité le 09/06/2020).

[20] Diego Perez-Botero, Jakub Szefer et Ruby B. Lee. « Characterizing hypervisor vulnerabilities in cloud computing servers ». en. In : *Proceedings of the 2013 international workshop on Security in cloud computing - Cloud Computing '13*. Hangzhou, China : ACM Press, 2013, p. 3. ISBN : 978-1-4503-2067-2. DOI : 10.1145/2484402.2484406. URL : http://dl.acm.org/citation.cfm?doid=2484402.2484406 (visité le 26/06/2020).

[21] Guillaume Poupard. *L'édito du Directeur général*. fr-FR. Publication Title : ANSSI. 2020. URL : https://www.ssi.gouv.fr/agence/missions/ledito-du-dg/ (visité le 01/07/2020).

[22] Stephan van Schaik, Andrew Kwong, Daniel Genkin et Yuval Yarom. *SGAxe : How SGX Fails in Practice*. 2020. URL : https://sgaxeattack.com/.

[23] Arvind Seshadri, Mark Luk, Elaine Shi et Adrian Perrig. « Pioneer : Verifying Code Integrity and Enforcing Untampered Code Execution on Legacy Systems ». en. In : (2005), p. 16.

[24] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom et Raoul Strackx. « Foreshadow : Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution ». In : *Proceedings of the 27th USENIX Security Symposium*. USENIX Association, août 2018.

[25] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lipp, Marina Minkin, Daniel Genkin, Yarom Yuval, Berk Sunar, Daniel Gruss et Frank Piessens. « LVI : Hijacking Transient Execution through Microarchitectural Load Value Injection ». In : *41th IEEE Symposium on Security and Privacy (S&P'20)*. 2020.

[26] Qiang Yan, Jin Han, Yingjiu Li, Robert H. Deng et Tieyan Li. « A software-based root-of-trust primitive on multicore platforms ». en. In : *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security - ASIACCS '11*. Hong Kong, China : ACM Press, 2011, p. 334. ISBN : 978-1-4503-0564-8. DOI : 10.1145/1966913.1966957. URL : http://portal.acm.org/citation.cfm?doid=1966913.1966957 (visité le 09/06/2020).

[27] Shaza Zeitouni, Ghada Dessouky, Orlando Arias, Dean Sullivan, Ahmad Ibrahim, Yier Jin et Ahmad-Reza Sadeghi. « ATRIUM : Runtime attestation resilient under memory attacks ». en. In : *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. Irvine, CA : IEEE, nov. 2017, p. 384-391. ISBN : 978-1-5386-3093-8. DOI : 10.1109/ICCAD.2017.8203803. URL : http://ieeexplore.ieee.org/document/8203803/ (visité le 09/06/2020).

COLOPHON

Ce rapport a été écris durant l'été 2020 pour mon diplôme d'ingénieur en informatique et réseau spécialisé en sécurité informatique. J'ai utilisé le modèle `classicthesis` développé par André Miede et Ivo Pletikosić.