Table des matières

Introduction

L'utilisation des systèmes informatiques est aujourd'hui en pleine évolution. Le modèle classique qui consiste à associer à chaque utilisateur une machine physique qu'il possède et dont il va exploiter les ressources devient de plus en plus obsolète. Aujourd'hui, les ressources informatiques que l'on utilise peuvent être distribuées n'importe où dans l'Internet et les postes de travail du quotidien ne sont plus systématiquement des machines réelles. Cette constatation met en avant deux phénomènes importants qui sont à l'origine de l'évolution de notre utilisation de l'informatique : le Cloud computing et la virtualisation. Le Cloud computing (ou informatique en nuage en français) permet à un utilisateur d'exploiter des ressources informatiques, de granularités potentiellement très différentes, pendant une durée variable, qui sont à disposition dans un nuage de ressources. L'utilisation de ces ressources est ensuite facturée à l'utilisateur. Ce modèle peut être bien sûr avantageux pour une entreprise qui peut s'appuyer sur des ressources informatiques potentiellement illimitées, qu'elle n'a pas nécessairement à administrer et gérer elle-même. Elle peut ainsi en tirer un gain de productivité et un gain financier. Du point de vue du propriétaire des machines physiques, le gain financier lié à la location des puissances de calcul est accentué par une maximisation de l'exploitation de ces machines par différents clients. L'informatique en nuage doit donc pouvoir s'adapter à la demande et facilement se reconfigurer. Une manière d'atteindre ces objectifs nécessite notamment l'utilisation de machines virtuelles et des techniques de virtualisation associées. Même si la virtualisation de ressources informatiques n'est pas née avec le Cloud, l'avènement du Cloud a considérablement augmenté son utilisation. L'ensemble des fournisseurs d'informatique en nuage s'appuient aujourd'hui sur des machines virtuelles, qui sont beaucoup plus facilement déployables et migrables que des machines réelles.

La virtualisation de ressources informatiques était auparavant essentiellement basée sur des techniques logicielles. Mais l'utilisation massive de machines virtuelles notamment pour le Cloud, a poussé les fondeurs de processeurs à inclure des mécanismes d'assistance matérielle à la virtualisation dans leurs processeurs. Ces extensions matérielles permettent d'une part de rendre plus facile la virtualisation, et d'autre part d'obtenir des gains de performance. Ainsi, un certain nombre de technologies ont vu le jour, telles que VT-x et VT-d chez Intel ou AMD-V chez AMD ou *Virtualization Extensions* chez ARM. Par ailleurs, la virtualisation nécessite l'implémentation de fonctionnalités supplémentaires, capables de gérer les différentes machines virtuelles, de pouvoir les ordonnancer, les isoler et partager les ressources matérielles comme la mémoire et les périphériques. Ces différentes fonctionnalités sont en général prises en charge par un gestionnaire de machines virtuelles, dont le travail peut donc être plus ou moins facilité, en fonction des caractéristiques du processeur sur lequel il s'exécute (assistance matérielle à la virtualisation ou non). De façon globale, ces technologies introduisent des nouveaux modes d'exécution sur les processeurs, de plus en plus privilégiés et de plus en plus complexes.

Ainsi, s'il est indéniable que l'utilisation de la virtualisation apporte un véritable intérêt pour l'informatique d'aujourd'hui, il est par ailleurs évident que sa mise en œuvre ajoute une complexité aux systèmes informatiques, complexité à la fois logicielle (gestionnaire de machines virtuelles) et matérielle (nouveaux mécanismes d'assistance à la virtualisation intégrés dans les processeurs). À partir de ce constat, il est légitime de se poser la question de la sécurité informatique dans ce contexte où l'architecture des processeurs devient de plus en plus

complexe, avec des modes de plus en plus privilégiés. Étant donné la complexité des systèmes informatiques, l'exploitation de vulnérabilités présentes dans les couches privilégiées ne risque-t-elle pas d'être très sérieuse pour le système global ? Étant donné la présence de plusieurs machines virtuelles, qui ne se font pas mutuellement confiance, au sein d'une même machine physique, est-il possible que l'exploitation d'une de ces vulnérabilités soit réalisée par une machine virtuelle compromise ? N'est-il pas nécessaire d'envisager de nouvelles architectures de sécurité prenant en compte ces risques ?

C'est à ces questions que cette thèse propose de répondre. En particulier, nous présentons un panorama des différents problèmes de sécurité dans des environnements virtualisés et des architectures matérielles actuelles. À partir de ce panorama, nous proposons dans nos travaux une architecture originale permettant de s'assurer de l'intégrité d'un logiciel s'exécutant sur un système informatique, quel que soit son niveau de privilège. Cette architecture est basée sur une utilisation mixte de logiciel (un hyperviseur de sécurité développé par nos soins, s'exécutant sur le processeur) et de matériel (un périphérique de confiance, autonome, que nous avons également développé). L'hyperviseur, même s'il est développé par nos soins et s'il s'exécute dans un mode très privilégié du processeur, est susceptible d'être compromis. Ainsi, le périphérique de confiance, totalement autonome et indépendant du processeur principal, est chargé de vérifier l"intégrité de l'hyperviseur. Nous présentons dans cette thèse la conception de cette architecture, ainsi qu'une implémentation qui a été réalisée sur une plateforme à base de processeur Intel. Des tests de performance et d'efficacité sont également présentés.

Cette thèse s'articule ainsi. Dans le premier chapitre, nous présentons le contexte de nos travaux en présentant les principales caractéristiques du cloud computing ainsi que ses acteurs. Nous précisons ensuite les concepts de virtualisation et de machines virtuelles ainsi que de différentes techniques de mise en œuvre associées à ces concepts. Ces techniques sont utilisées par les gestionnaires de machines virtuelles dont nous définissons les fonctionnalités et responsabilités en suivant. Le contexte de cette thèse étant également la sécurité informatique, nous présentons ensuite rapidement la terminologie associée ainsi que les principales définitions utiles à la compréhension de cette thèse. Dans la suite de ce chapitre, nous identifions les problèmes de sécurité apportés par la virtualisation, pour en déduire un modèle de menace. Un panel d'attaques connues sur ces systèmes est commenté pour terminer la section. Puis, nous présentons un état de l'art des solutions de sécurité existantes pour les systèmes virtualisés avant de terminer ce premier chapitre par une énumération des différents objectifs de cette thèse.

Le deuxième chapitre présente une étude, orientée sécurité, des architectures actuelles que nous avons utilisées pour développer notre architecture de sécurité. Dans cette étude nous commençons par donner les caractéristiques des architectures actuelles en termes de communications inter composants. Nous spécialisons nos observations sur le cas des processeurs Intel et donnons les différents mécanismes de contrôle d'accès qu'ils incluent. Nous terminons ce chapitre en donnant différents rappels techniques précis, nécessaires pour la compréhension générale de ce manuscrit de thèse.

Le troisième chapitre est dédié à la présentation de la contribution majeure de cette thèse : une architecture de sécurité permettant de réaliser des tests d'intégrité de logiciel à l'exécution, qui tient compte des observations faites dans les deux premiers chapitres. Après avoir présenté les hypothèses de nos travaux, en accord avec le modèle de menaces du chapitre 1, nous décrivons le principe général de notre architecture ainsi que le protocole de tests d'intégrité que nous mettons en place. Cette architecture inclut un hyperviseur de sécurité, chargé de

tester l'intégrité du logiciel sur le système, ainsi qu'un périphérique matériel de confiance, chargé de tester l'intégrité de l'hyperviseur lui-même. Nous décrivons ensuite comment nous avons été capables de caractériser l'intégrité de l'hyperviseur de sécurité et nous terminons en présentant la conception des différents tests que nous effectuons pour tester chacune de ses caractéristiques.

Le quatrième chapitre concerne la mise en œuvre de notre architecture sur un prototype. Notre architecture nécessite la mise en œuvre du périphérique de confiance et de l'hyperviseur de sécurité dont le développement est présenté dans les deux premières sections. Ce chapitre se termine avec la présentation de la mise en œuvre des différents tests d'intégrité logiciels effectués par les deux composants mis en œuvre.

Enfin, le dernier chapitre de cette thèse présente des expérimentations concrètes réalisées avec notre prototype. Nous commençons par présenter une version virtuelle de notre architecture de sécurité, qui a permis de concevoir plus rapidement les tests d'intégrité de l'hyperviseur. Le reste du chapitre concerne les différentes expérimentations de validation de l'efficacité et de calcul des performances de notre architecture de sécurité.

Chapitre 1

Contexte scientifique et État de l'art

Sommaire

Ce chapitre de thèse présente tout d'abord le *cloud computing* et propose ensuite un état de l'art de différents travaux, qui concernent les techniques de virtualisation et les problèmes de sécurité associés. Ainsi, ce chapitre est composé de cinq parties. Dans la première, nous rappelons tout d'abord les caractéristiques principales du cloud computing et des acteurs associés. Ensuite, nous présentons un aperçu des différents types et techniques de virtualisation qui sont employés aujourd'hui. Dans la troisième partie, nous nous focalisons sur le rôle d'un hyperviseur dans les architectures virtualisées ainsi que sur ses fonctionnalités principales. La quatrième partie est consacrée à la description des différentes vulnérabilités et attaques liées à la virtualisation des systèmes informatiques. La cinquième partie est quant à elle consacrée à la présentation des solutions de sécurité existant aujourd'hui pour faire face à ces problèmes. Enfin, ce chapitre termine par justifier, au vu de cet état de l'art, les travaux de cette thèse.

1.1 Le cloud computing

Ces dernières années, les entreprises ont progressivement transformé leurs parcs informatiques. Initialement, si nous prenons l'exemple des applications webs d'entreprises, les ordina-

teurs personnels clients ainsi que les serveurs étaient hébergés physiquement et gérés par la même entreprise. Elle jouait donc à la fois le rôle de fournisseur de service et de client.

Puis, de plus en plus, l'hébergement des serveurs a été délégué à des entreprises externes (action de mettre en infogérance) pour minimiser les coûts de production. Par la suite, plus de responsabilités ont été données aux entreprises d'infogérance. Notamment, la gestion et la maintenance de la pile logicielle de la machine ont été aussi déléguées à ces acteurs extérieurs, qui proposent par exemple directement la location d'environnements d'hébergement de sites Web ou d'applications diverses accessibles par le réseau Internet. Dans ce cas, les entreprises clientes n'ont plus la possibilité de gérer les couches sous-jacentes aux applications qu'elles utilisent. De plus, la configuration et la mise en œuvre de ces applications ne sont pas entièrement connues et peuvent être spécifiques à l'hébergeur, ce qui peut entraîner des difficultés de migration. L'infogérance entraîne une donc grande dépendance vis-à-vis du fournisseur de service, sans pour autant qu'elle apporte de nouvelles fonctionnalités.

L'avènement de la virtualisation a redonné de la flexibilité et de la maîtrise sur la pile logicielle d'une machine. En effet, les fournisseurs de services se sont appuyés sur les machines virtuelles pour, par exemple, permettre à un utilisateur de garder le contrôle sur le système opératoire d'une machine qu'il veut faire infogérer. Aussi, les hébergeurs obtiennent la possibilité de mutualiser des machines physiques pour héberger plusieurs machines virtuelles, réduisant encore les coûts. Notons que les architectures virtualisées n'interdisent pas non plus les anciennes méthodes d'hébergement de services comme la mutualisation de serveurs Web. Les clients peuvent de plus migrer plus simplement leurs machines virtuelles chez un autre hébergeur en déplaçant simplement les fichiers de configuration des machines virtuelles et disques virtuels. Cette solution est tout de même limitée en matière de flexibilité. Par exemple, la configuration matérielle des machines virtuelles n'est pas extensible sans l'intervention du fournisseur de service. En plus, en cas de défaillance physique, le fournisseur doit effectuer manuellement la migration des machines virtuelles sur d'autres hôtes physiques, ce qui engendre un temps d'arrêt notable même s'il est plus réduit qu'avec les architectures non virtualisées. Enfin, le client ne peut pas lui-même configurer une infrastructure complète avec un contrôle total sur le matériel et le réseau virtuel sans intervention du fournisseur. L'absence de ces fonctionnalités a contribué à la conception des architectures cloud, ou *cloud computing*, qui a émergé il y a quelques années maintenant.

Le *cloud computing* (*cloud*) ou informatique en nuage en Français, est un modèle d'architecture et d'infrastructure de service informatique. Les formes que peuvent prendre les architectures cloud sont si nombreuses et différentes qu'il est très difficile d'en donner une définition ferme et arrêtée. Néanmoins, l'institut national des standards et des technologies, le *National Institute of Standards and Technology* (NIST), propose une définition via les caractéristiques du cloud et les acteurs, souvent reprise par les acteurs du domaine.

1.1.1 Caractéristiques essentielles du cloud

Selon le NIST, le *cloud* est défini par 5 caractéristiques essentielles, 3 types de services pouvant être commercialisés et 4 modèles de déploiement [1, 2].

Le cloud doit proposer un service disposant des caractéristiques suivantes :

— **Libre et accessible à la demande** : le client doit pouvoir unilatéralement modifier la configuration de son infrastructure informatique, par exemple le temps processeur loué ou sa bande passante réseau, sans que cela ne requière l'intervention d'une personne

humaine chez le fournisseur de service.

— **Accessible massivement par le réseau** : les fonctionnalités du cloud doivent être accessibles par le réseau et au travers d'une diversité d'appareils qui ne disposent pas forcément de grandes capacités de calcul tels que des appareils mobiles.

— **Où les ressources sont mises en commun** : les ressources du fournisseur de service sont mises en commun afin de servir une multitude de clients selon un modèle multi tenant, c'est-à-dire mutualisant les ressources matérielles et logicielles. Cette propriété apporte un aspect d'indépendance vis-à-vis de la localisation physique des ressources, sur laquelle le client n'a généralement pas de contrôle.

— **Flexible** : Il doit être possible d'allouer et de supprimer rapidement des ressources informatiques, et ce parfois automatiquement. Du point de vue de l'utilisateur, la quantité de ressources apparaît souvent illimitée. Celles-ci doivent pouvoir être affectées à tout moment et à n'importe quelle quantité.

— **Facturé à l'usage** : les systèmes cloud contrôlent et optimisent l'utilisation de ressources en effectuant des mesures au niveau de contrôle approprié. L'utilisation des ressources peut être surveillée, contrôlée et rapportée, ce qui apporte autant de transparence au fournisseur de service qu'à l'utilisateur.

Les architectures de type cloud proposent à la consommation trois types de services différents :

— *Software As A Service* (**SAAS**) ou logiciel en tant que service : ici, le fournisseur de service offre la possibilité aux utilisateurs d'utiliser le service fourni par une application telle qu'une application WEB, une API réseau ou bien un service de courrier électronique. L'utilisateur ne gère ni ne contrôle l'infrastructure cloud sous-jacente à l'application fournie.

— *Platform As A Service* (**PAAS**) ou plateforme en tant que service : ce type de service donne la capacité à l'utilisateur de déployer dans l'infrastructure cloud, diverses applications, bibliothèques, outils qui sont supportés par le fournisseur de service. L'utilisateur ne gère ni ne contrôle la couche cloud sous-jacente comme le réseau, les serveurs physiques, le système d'exploitation ou le stockage. Toutefois, il peut configurer les applications et l'environnement d'hébergement de celles-ci.

— *Infrastructure As A Service* (**IAAS**) ou l'infrastructure en tant que service : à ce niveau de service, on donne à l'utilisateur la capacité d'allouer du temps processeur, du stockage, configurer le réseau et d'autres ressources informatiques fondamentales dans lesquelles il peut y déployer n'importe quel type de logiciel. Encore une fois, l'utilisateur n'a pas à gérer les couches du cloud sous-jacentes, mais dispose du contrôle sur les systèmes d'exploitation, le stockage, les applications et certains éléments réseau.

Le NIST définit aussi 4 modèles de déploiement caractérisant l'origine des utilisateurs vis-à-vis du fournisseur de service, c'est-à-dire le propriétaire de l'infrastructure cloud.

— **Le cloud privé** : les utilisateurs finaux des services cloud privés appartiennent à la même organisation. L'infrastructure peut être elle aussi propriété de cette même organisation, mais peut aussi être pilotée par une entreprise tierce ou les deux.

— **Le cloud communautaire** : les utilisateurs appartiennent à un groupe d'organisations qui partagent certains centres d'intérêt (du point de vue de la sécurité par exemple). Il est géré par une ou plusieurs organisations membres de la communauté, mais peut l'être aussi encore une fois par un acteur extérieur.

— **Le cloud public** : les clouds publics ont pour vocation d'être massivement ouverts à

tous, ce qui implique que les utilisateurs ne partagent très certainement aucun centre d'intérêt. Ce type de cloud peut être géré par tout type d'organisations.

— **Le cloud hybride** : un regroupement d'au moins deux clouds qui ne suivent pas forcément le même mode de déploiement. Les différents clouds sont connectés par des protocoles définis pour rendre possible la migration de données et d'applications inter cloud ce qui permet notamment d'effectuer de l'équilibrage de charge.

Les 5 propriétés du cloud citées précédemment peuvent paraître très ambitieuses à obtenir en réalité. Prenons quelques exemples illustrant leur difficulté de mise en œuvre. Du point de vue de la flexibilité des ressources, il apparaît par exemple difficile d'opérer et contrôler une éventuelle diminution demandée de la quantité de mémoire vive et de temps processeur, utilisés par le système d'exploitation d'un client, s'il maîtrise complètement les ressources matérielles d'une machine physique. Comment opérer une mise en commun des ressources matérielles d'une même machine entre plusieurs systèmes d'exploitation si ceux-ci ont pour vocation de s'en approprier la totalité ? Comment mettre en application le concept d'indépendance de la localisation physique pour migrer simplement les différents logiciels et données d'un client d'une machine physique à une autre ?

1.1.2 Les acteurs des systèmes cloud

Il existe 4 acteurs principaux dans les systèmes cloud.

Le client Le client est une personne qui va consommer le service d'un cloud. Elle peut être la même que le gérant du cloud dans le cas du cloud privé ou faire partie d'une organisation différente si l'on considère un cloud public. Dans la très grande majorité des cas, ces personnes ne possèdent pas de privilèges sur la partie administrative de l'infrastructure et n'ont pas d'accès physique aux machines du cloud.

L'administrateur du cloud Un administrateur de cloud est une personne physique appartenant à l'organisation gérante du cloud. Afin d'exercer son activité, il possède donc des privilèges élevés sur l'infrastructure et parfois même un accès physique aux machines hôtes qui composent le cloud.

Machines distantes Ce groupe comprend toutes les machines et leurs utilisateurs pouvant communiquer, au travers d'un réseau quelconque, avec une interface physique présente sur une des machines hôtes du cloud. Dans notre étude, cet acteur comprend aussi les autres machines hôtes faisant partie du même cloud.

Les constructeurs de matériel Ce dernier type d'acteur est spécial dans le sens où il ne va pas utiliser / administrer et interagir avec le cloud en production, mais est tout de même essentiel à représenter dans notre modèle de menace. Afin d'exécuter les logiciels et communiquer avec le monde extérieur, les concepteurs de systèmes et fournisseurs de service s'appuient sur ces matériels produits par ce type d'acteur. Ces matériels sont très complexes et leur chaîne de production n'est pas forcément maîtrisée en termes de sécurité, du début à la fin par les constructeurs, à cause de problématiques de sous-traitance par exemple. Pour ces raisons, leur intégration dans l'analyse de sécurité des systèmes cloud est donc, selon nous, indispensable.

Toutes ces caractéristiques, types de service et modèles de déploiement, afin de pouvoir être mis en œuvre de manière réaliste et performante, s'appuient sur le concept de machines virtuelles et de virtualisation dont la définition et les caractéristiques principales sont données dans la section suivante.

1.2 Virtualisation et machines virtuelles

La définition qui fait le plus souvent référence concernant la notion de machine virtuelle est celle de Gerald J. Popek et coll. dans [2].

Definition 1.1. Une machine virtuelle est une copie d'une machine physique qui doit demeurer performante et isolée. Une machine virtuelle s'exécute sous le contrôle exclusif du gestionnaire de machines virtuelles qui va mettre en place un environnement d'exécution proche de la machine physique et qui n'engendre pas de perte significative de performance.

Pour mettre en œuvre cette définition, il existe plusieurs types et techniques de virtualisation dont nous détaillons ici les principaux.

1.2.1 Types de virtualisation

Le concept théorique de machine virtuelle est mis en pratique grâce à la virtualisation. En informatique, la virtualisation désigne l'abstraction des ressources d'une machine physique, et ce à différents niveaux. Il existe deux types de virtualisation qui sont définis par le comportement du logiciel exécuté et virtualisé.

Le premier cas considère que les logiciels sont conscients d'être virtualisés. Dans ce cadre, ils interagissent explicitement avec le **gestionnaire de machines virtuelles** sous-jacent (sur lequel nous reviendrons dans la suite) afin de demander l'exécution d'actions spécifiées, nécessitant par exemple des privilèges plus élevés. On parle dans ce cas de **paravirtualisation**. Un logiciel développé pour un environnement non virtuel doit donc être adapté pour pouvoir s'exécuter correctement dans ce type d'environnement.

Par opposition, la **virtualisation complète** ou *full virtualization* prend en charge des logiciels qui n'ont aucunement conscience d'être virtualisés. Ceux-ci interagissent avec les interfaces logicielles et matérielles qu'ils utilisent habituellement hors contexte de virtualisation. Il incombe donc au gestionnaire de machines virtuelles de mettre en place les interfaces logicielles ou d'émuler le matériel rendant possible cette exécution. Ce type de virtualisation permet donc d'exécuter directement un logiciel développé hors contexte de virtualisation sans lui appliquer de modification.

1.2.2 Techniques de virtualisation

Pour mettre en œuvre la virtualisation, il existe différentes techniques, abstrayant différentes ressources matérielles et logicielles : les périphériques ; l'interface logicielle d'un système d'exploitation et d'un logiciel ; certaines parties du processeur ; le processeur dans son intégralité et enfin une architecture matérielle entière. Ces techniques sont valables dans les architectures virtualisables selon la définition de Popek et ses associés [2], dans lesquelles l'exécution d'un logiciel peut être contrôlée par une entité plus privilégiée, le gestionnaire de machines virtuelles. Ce contrôle est exercé en interceptant l'exécution d'instructions privilégiées pour

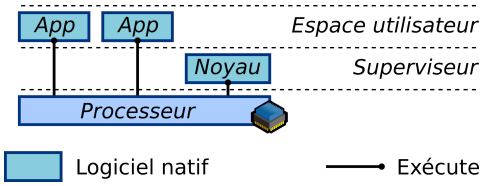

FIGURE 1.1 – Architecture logicielle classique

donner la main au gestionnaire de machines virtuelles (figure 1.1). Sans ces interceptions, non seulement le gestionnaire de machines virtuelles ne peut plus gérer les ressources entre les différentes machines virtuelles, mais en plus, une de ces machines peut alors exécuter des instructions qui auraient pour effet de désinstaller complètement le gestionnaire de machines virtuelles.

Virtualisation de la mémoire Dans les processeurs actuels, les applications possèdent un espace d'adressage virtuel qui est traduit et filtré selon une configuration mise en place par le noyau du système d'exploitation. Cela permet à l'application d'imposer la cartographie de son espace mémoire au chargeur d'application, qui lui met en place la traduction d'adresse demandée. Cela apporte aussi certaines garanties de sécurité, en rendant inaccessible l'espace mémoire d'applications s'exécutant dans un contexte différent. Sur la figure 1.2, on comprend que les processeurs 1 et 2 ont la même structure et peuvent contenir certaines adresses virtuelles identiques. Par contre dans la mémoire physique, les espaces mémoire alloués par les 2 processeurs sont bien différents. Toutefois, certains segments peuvent être partagés, notamment les bibliothèques partagées, mais aussi certaines mémoires de données affectées à la communication inter processus.

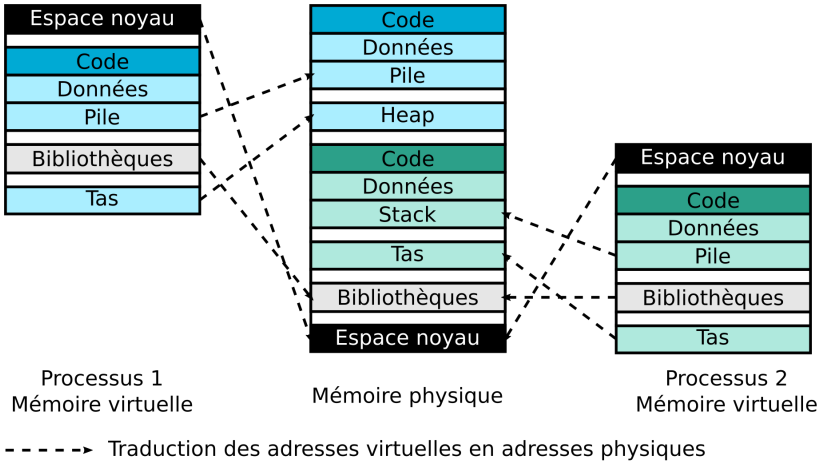

FIGURE 1.2 – Virtualisation de la mémoire des applications sous les systèmes Unix

Émulation de processeurs L'émulation de processeurs est un cas spécial de la virtualisation dans le sens où elle reproduit un environnement d'exécution complet en interprétant logiciellement les instructions d'un programme compilé pour une architecture donnée (figure

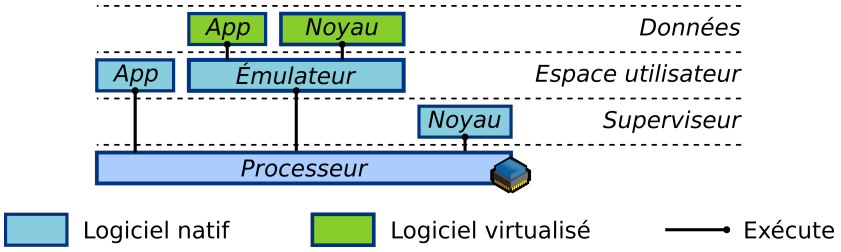

FIGURE 1.3 – Architecture matérielle émulée

1.3). On parle parfois aussi de *binary translation*. Il existe plusieurs types d'émulation, en fonction du niveau choisi pour réaliser la translation.

En ce qui concerne les applications, certains langages de programmation compilés, tels que Java ou la famille .NET, ciblent des architectures et un jeu d'instructions virtuel pour des raisons de portabilité. L'exécution de ces programmes doit être prise en charge par une machine virtuelle fournie par le développeur du langage, qui émule les instructions du programme. La machine virtuelle, quant à elle, est écrite en code natif pour une plateforme donnée. Ainsi, le développeur d'application n'a pas à supporter différents environnements matériels et systèmes d'exploitation hôtes, ce support étant pris en compte par les différentes machines virtuelles.

D'autres émulateurs sont destinés à émuler des systèmes d'exploitation entiers (noyau et applications). Pour ce faire, ils sont amenés à émuler entièrement le comportement du processeur utilisé dans le logiciel. Pour conséquent, il est donc nécessaire de virtualiser également les périphériques (réseau, stockage et affichage). Notons que le code des applications et du noyau font partie des données de l'émulateur.

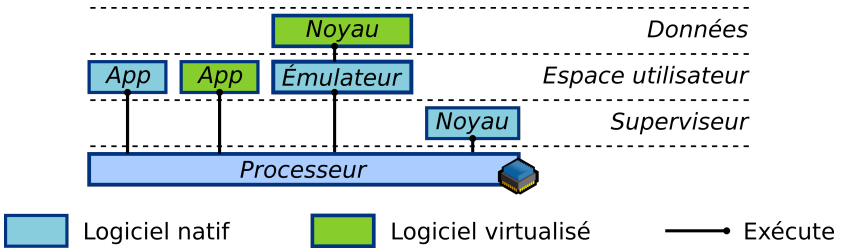

FIGURE 1.4 – Virtualisation hybride

Émulation du noyau Afin d'améliorer les faibles performances liées à l'émulation, certains concepteurs de gestionnaires de machines virtuelles comme VMware [3] et QEMU associé à KQEMU [4], mêlent l'émulation avec l'exécution réelle de code dans le cadre de la virtualisation d'un système complet. Les applications du système virtualisé, non privilégiées, peuvent être exécutées comme des applications classiques. Le noyau est, quant à lui, entièrement émulé, de façon à contrôler entièrement son comportement (figure 1.4). Ce choix se justifie, en ce qui concerne les performances, par le fait que le processeur passe très peu de temps dans l'exécution du noyau par rapport à l'exécution des applications et de ce fait diminue largement l'impact de l'émulation sur les performances du système.

Conteneurs d'applications Le but de ce type de virtualisation est de pouvoir obtenir plusieurs instances d'espaces utilisateurs (applications, systèmes de fichiers, services, confi-

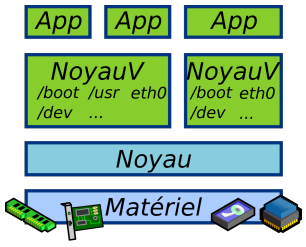

FIGURE 1.5 – Exemple de virtualisation de système de fichier et d'interface réseau

gurations, etc.), tout en gardant un seul noyau de système d'exploitation partagé entre les différentes instances virtuelles. On parle aussi de conteneurs ou conteneurs d'application, *software containers* en anglais [5], [6], [7], [8], [9]. Les interfaces logicielles entre l'application et le noyau du système d'exploitation sont donc instrumentées pour pouvoir mettre en place ce type d'environnement d'exécution d'applications. Notons que les conteneurs rendent possible l'exécution d'applications développées pour un système d'exploitation différent sans leur apporter de modification.

Virtualisation logicielle des périphériques Les bus et périphériques peuvent être aussi virtualisés afin de supporter l'exécution de noyaux de systèmes d'exploitation et de drivers dans un environnement virtuel. Dans ce cas, les interfaces matérielles d'entrées / sorties sont émulées afin de présenter aux logiciels un environnement virtuel en apparence similaire à l'environnement réel. Il existe deux cas de figure : soit le périphérique est entièrement émulé (disque virtuel, carte graphique virtuelle), soit l'interface matérielle vers le périphérique est émulée par un driver et parfois partagée avec d'autres logiciels virtualisés (interface de communication, carte réseau). L'interface émulée ordonnance les accès aux périphériques entre machines virtuelles. Les auteurs de [10] nomment ce type de driver *para pass-through*

Virtualisation matérielle des périphériques De plus en plus, les périphériques embarquent des fonctionnalités d'accélération matérielle de virtualisation de leurs fonctions. Une extension de la spécification du bus PCI Express a été notamment portée par Intel dans ce sens, appelée *Single Root Input Output Virtualization* (SR-IOV) [11]. L'accélération matérielle de la virtualisation de ces fonctions minimise le goulot d'étranglement du débit causé par le processeur dans le cadre de l'émulation de périphérique complet ou de son interface matérielle. Ce support existe pour l'instant uniquement sur le bus PCI Express. Dans le cas des cartes réseau, la technologie SR-IOV permet d'optimiser la surcharge ajoutée par le partage des périphériques physiques par l'hyperviseur. Il est maintenant possible de définir un nombre donné de fonctions virtuelles PCI Express dans la carte réseau, que l'on associe à des machines virtuelles. Ces machines pourront directement communiquer avec les fonctions virtuelles via des accès mémoire directs, pour configurer l'envoi et la réception de trames Ethernet ainsi que les interruptions. Pour la réception d'une trame Ethernet, l'identification de la machine virtuelle à interrompre sera effectuée directement au niveau de la carte réseau, grâce aux adresses MAC associées aux différentes fonctions virtuelles. Ainsi le gestionnaire de machines virtuelles n'a plus à être interrompu en premier pour router logiciellement l'interruption vers la machine virtuelle destinataire. Cette méthode permet d'obtenir une véritable bande passante dix Giga bits dans un environnement virtuel.

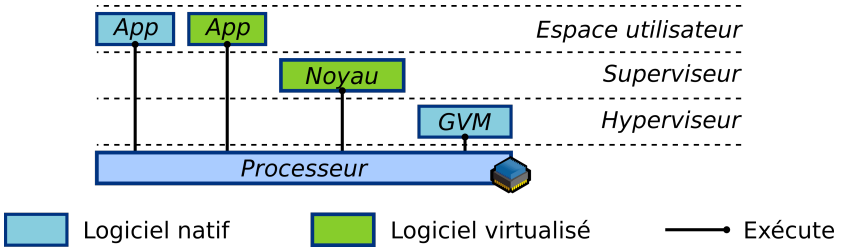

FIGURE 1.6 – Virtualisation matérielle

Virtualisation matérielle de cœurs de processeur La mise en œuvre de la virtualisation complète par émulation peut être extrêmement lente comparée à l'exécution du même logiciel sur un processeur réel. Si nous prenons l'exemple des architectures x86, les premières implémentations de virtualisation des cœurs s'effectuaient via l'utilisation des niveaux de privilèges, les anneaux ou *rings*, apportés par la segmentation. Pour la plupart, les systèmes d'exploitation actuels n'utilisent que 2 des 4 niveaux de privilège disponibles, 0 pour le noyau et 3 pour les applications. Ainsi certains des premiers concepteurs de gestionnaires de machines virtuelles ont eu l'idée de tirer parti de ce constat pour contrôler l'exécution de logiciel privilégié (OS, firmware) dans une machine virtuelle, en l'exécutant en *ring 1* au lieu de *ring 0*, technique appelée compression de *rings*. Ainsi certaines actions, comme la modification du niveau de privilège, l'accès aux registres de contrôle, etc., rendent la main au gestionnaire de machines virtuelles qui peut exercer son contrôle sur l'exécution de la machine virtuelle. Le problème de cette approche est qu'il engendre de nombreuses transitions entre la machine virtuelle et le gestionnaire de machines virtuelles. Aussi, il y a quelques années, Intel a proposé des extensions matérielles permettant de virtualiser un cœur de processeur, de manière plus performante, c'est-à-dire limitant le nombre de changements de contexte inutiles, par rapport à la technique précédemment citée. La virtualisation des cœurs implique aussi la virtualisation de fonctionnalités comme la mémoire, les interruptions, l'exécution de certaines instructions et les entrées / sorties. Contrairement à l'émulation du noyau qui s'effectue à un niveau sémantique plus élevé, l'exécution d'un service (appel système par exemple) va engendrer plusieurs interceptions où pour chacune le gestionnaire de machines virtuelles devra déterminer le contexte de cette interception et autoriser ou émuler l'action privilégiée en cours. C'est pour cette raison que certains développeurs de gestionnaires de machines virtuelles tels que VMWare, ont conservé la technique d'émulation du noyau jusqu'à très récemment, car elle était proche au niveau des performances.

Virtualisation matérielle des interruptions Dans le cas où au moins deux systèmes d'exploitation partagent la même machine, il est nécessaire de distribuer correctement les interruptions entre les deux noyaux. Pour cette raison, les interruptions doivent être gérées par le gestionnaire de machines virtuelles qui redistribue celles-ci au bon système d'exploitation. Plusieurs cas de figure existent : soit le gestionnaire de machines virtuelles s'installe en gestionnaire d'interruptions systématiquement puis les redistribue en temps voulu, soit il existe un support matériel dans la machine physique permettant d'accélérer cette redistribution sans intervention du gestionnaire de machines virtuelles.

Cette section a présenté la plupart des techniques de virtualisation du logiciel et du matériel. Dans la prochaine section, nous présentons comment il est possible de les composer pour mettre en œuvre le concept de machines virtuelles.

1.2.3 Mise en œuvre des machines virtuelles

Selon la définition 1.1 donnée par Popek et ses associés, une machine virtuelle doit posséder certaines caractéristiques qui contraignent les techniques de virtualisation que l'on peut utiliser pour mettre en œuvre ce concept :

1. Correspondre à la copie d'une machine physique ;
2. Être performante ;
3. Son exécution doit être contrôlée par un gestionnaire de machines virtuelles.

Les gestionnaires de machines virtuelles utilisés dans les architectures cloud en production aujourd'hui utilisent donc un sous-ensemble des techniques de virtualisation citées précédemment :

— Dans la plupart des cas, la virtualisation matérielle de cœurs de processeur.
— La virtualisation de périphériques via l'émulation ou un support matériel dans le périphérique lui-même, notamment pour les fonctionnalités d'affichage de stockage et des communications réseau.
— La virtualisation matérielle de la mémoire (cœurs et périphériques).
— La virtualisation matérielle des interruptions.

Nous excluons de nos travaux les machines virtuelles ne rentrant pas dans la catégorie (1), comme les machines liées aux langages Java et .NET.

Enfin, en ce qui concerne le type de virtualisation, la paravirtualisation et la virtualisation complète sont toutes deux utilisées, et ce parfois ensemble dans un même projet (Xen [12]).

Les logiciels qui mettent en œuvre ces techniques afin de supporter l'exécution des machines virtuelles sont les gestionnaires de machines virtuelles, que nous avons déjà abordés à plusieurs reprises dans ce manuscrit. La section suivante présente les différents types existants.

1.3 Gestionnaires de machines virtuelles

Un gestionnaire de machines virtuelles, ou hyperviseur, est un logiciel conçu pour mettre en place l'environnement d'exécution de machines virtuelles en utilisant les techniques présentées en section 1.2.2 et 1.2.3.

1.3.1 Architecture

Ces gestionnaires sont classés en deux catégories, les hyperviseurs de type 1 et de type 2 (figure 1.7).

Un hyperviseur de **type 2** ou **hébergé** va gérer l'exécution des machines virtuelles en s'appuyant sur les fonctionnalités et sur la couche d'abstraction du matériel qu'offre un système d'exploitation. Il existe de multiples hyperviseurs de type 2 disponibles, qui sont à peu près équivalents en termes de type de virtualisation et techniques de virtualisation : virtualisation complète et virtualisation matérielle des cœurs de processeur [13], [14], [15], [16].

Ce type d'hyperviseur doit s'interfacer directement avec le matériel en le configurant et interagissant avec lui tout au long de l'exécution de la machine avec ses propres pilotes [17], [18], [19], [12]. Un hyperviseur de **type 1** ou *bare-metal* est généralement chargé au plus tôt dans l'ordre de démarrage de la machine, à la manière d'un système d'exploitation.

Pour pouvoir aborder les problématiques de sécurité concernant les hyperviseurs, il est nécessaire de définir leur spécification fonctionnelle.

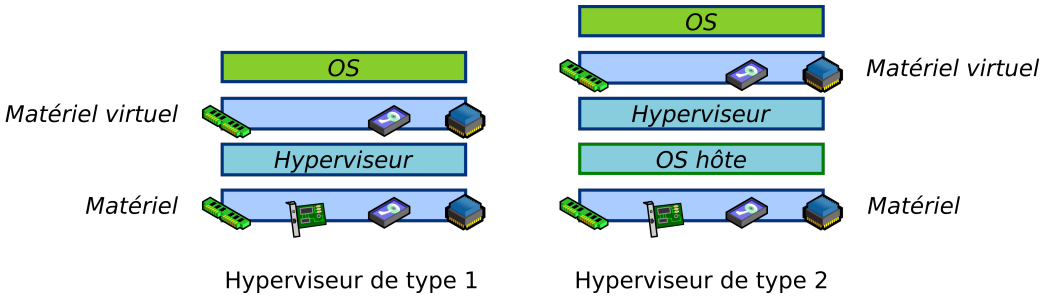

FIGURE 1.7 – Types d'hyperviseurs

1.3.2 Fonctionnement et rôle de l'hyperviseur

Cette section décrit le rôle fonctionnel des hyperviseurs vis-à-vis du matériel et des machines virtuelles qu'il met en place. Notons que la position privilégiée qu'occupe le gestionnaire de machines virtuelles dans les architectures implique des responsabilités similaires aux noyaux de systèmes d'exploitation vis-à-vis du matériel. Ces fonctions peuvent être classées en 4 catégories [20].

Gestion des cœurs du processeur En premier lieu, les machines virtuelles sont exécutées par les cœurs du processeur. Il incombe donc à l'hyperviseur d'en gérer leur utilisation et d'en émuler certaines parties. Ces machines virtuelles sont créées et supprimées de façon dynamique. À la manière des processus, plusieurs machines virtuelles sont allouées à un même cœur, il est donc important de les ordonnancer afin d'offrir du temps processeur à chacune d'elle. De plus, l'hyperviseur lui-même est amené à effectuer ses propres calculs (gestion de timers, de périphériques virtuels, etc.), il va donc inscrire dans cet ordonnancement ses propres processus.

Gestion de la mémoire L'hyperviseur doit aussi gérer l'espace mémoire physique de la machine. Cet espace est partagé en espaces virtuels alloués aux différentes machines virtuelles. Dans le cadre de la virtualisation complète, les systèmes d'exploitation de ces machines peuvent disposer de leur espace mémoire virtuel pour allouer de la mémoire aux applications par exemple, et ce de manière identique à un environnement physique. C'est en général au gestionnaire de machines virtuelles d'apporter ou de configurer le support mettant en œuvre cette virtualisation de l'espace mémoire physique. C'est pour cette raison que, dans ce cas, deux mécanismes de traduction d'adresses sont enchaînés. Tout d'abord les adresses virtuelles de la machine virtuelle sont traduites en adresses physiques de la machine virtuelle, puis elles sont retraduites dans un deuxième temps en adresses physiques de l'hôte hyperviseur.

Gestion des périphériques Les périphériques forment une ressource importante de la machine physique. Ils permettent notamment aux machines virtuelles de communiquer sur des réseaux IP et de rendre le service pour lequel elles ont été conçues. Pour cette raison, l'hyperviseur doit aussi partager l'accès aux périphériques physiques entre les différentes machines virtuelles en utilisant les techniques de virtualisation citées précédemment. La technique la plus couramment utilisée consiste à émuler l'interface physique des bus et périphériques, qui eux sont associés aux machines virtuelles, pour ensuite redistribuer ces entrées / sorties vers différents périphériques physiques.

Gestion des interruptions Chaque système d'exploitation virtualisé configure son gestionnaire d'interruption virtuel de manière à être notifié d'évènements matériels divers qui ont été configurés dans leurs périphériques virtuels (*timers ticks*, traitement terminé sur un périphérique, etc.). Les configurations d'interruptions des différentes machines virtuelles ont un impact sur la configuration du gestionnaire d'interruption physique associé à un cœur. L'hyperviseur doit donc traduire cette configuration. De plus lorsqu'une interruption apparaît lors de l'exécution d'une machine virtuelle donnée, elle doit être gérée par le gestionnaire de machines virtuelles afin que celui-ci décide quelles sont les machines destinataires de celle-ci. Enfin, notons bien qu'à la manière des périphériques et de la mémoire, l'hyperviseur lui-même configure le gestionnaire d'interruptions pour ses propres besoins.

Les hyperviseurs et le support matériel pour la virtualisation ajoutent une nouvelle couche logicielle et par conséquent de nouvelles vulnérabilités que nous décrivons dans la section suivante.

1.4 Vulnérabilités et attaques

Cette section commence par un rappel sur la terminologie de la sécurité informatique, elle présente ensuite les enjeux de sécurité autour des architectures virtualisées, puis fait un focus sur les gestionnaires de machines virtuelles en définissant le modèle de menaces considéré pour les travaux présentés dans ce manuscrit.

1.4.1 Terminologie de la sécurité informatique

La sûreté de fonctionnement définit les termes de sécurité-innocuité (*safety*) et sécurité-immunité (*security*). Nos travaux s'inscrivent dans le cadre de la sécurité-immunité. Cette section donne le vocabulaire nécessaire pour la suite de la lecture de ce manuscrit, en commençant par définir les principaux concepts de la sûreté de fonctionnement, issus de [21] et mis à jour dans [22], puis en se focalisant sur la sécurité-immunité.

1.4.1.1 Sûreté de fonctionnement

La sûreté de fonctionnement d'un système informatique est définie comme « la propriété qui permet aux utilisateurs du système de placer une confiance justifiée dans le service qu'il leur délivre ». Le service délivré correspond au comportement du système perçu par ses utilisateurs. La sûreté de fonctionnement comporte trois axes principaux : les attributs qui la décrivent, les entraves qui empêchent sa réalisation et les moyens d'atteindre celle-ci (figure 1.8).

Les attributs de la sûreté de fonctionnement sont les propriétés complémentaires suivantes :

— **Disponibilité** : capacité d'un système à être prêt à l'utilisation.
— **Fiabilité** : continuité du service.
— **Sécurité-innocuité** : non-occurrence de conséquences catastrophiques pour l'environnement.
— **Confidentialité** : non-occurrence de divulgations non autorisées de l'information.
— **Intégrité** : non-occurrence d'altérations inappropriées de l'information.
— **Maintenabilité** : aptitude d'un système à être réparé ou à subir des évolutions.

On parle de non-sûreté de fonctionnement lorsqu'au moins une partie ces attributs ne peuvent plus être assurés pour cause de fautes, d'erreurs et de défaillances :

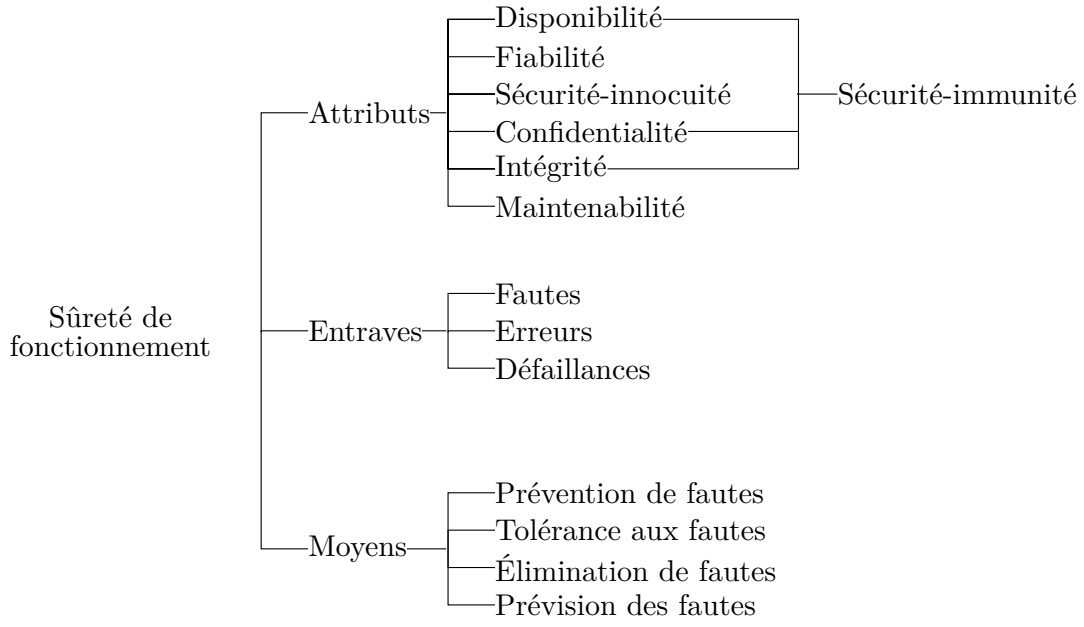

FIGURE 1.8 – Vocabulaire de la sûreté de fonctionnement

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

Une faute est dite **active** lorsqu'elle produit une erreur. Par propagation une erreur crée de nouvelles erreurs. Une défaillance survient lorsque, par propagation, elle affecte le service délivré par le système. La conséquence de la défaillance d'un composant est une faute pour le système qui le contient ou pour les composants qui interagissent avec lui. L'enchaînement de ces entraves permet de compléter la « chaîne fondamentale »suivante :

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

Pour mettre en place la sûreté de fonctionnement dans un système, on dispose de plusieurs **moyens**. Ces moyens sont des méthodes et techniques qui vont permettre de réduire les conséquences des entraves sur les attributs :

— La **prévention de fautes** empêche l'occurrence ou l'introduction de fautes.
— La **tolérance aux fautes** permet de fournir un service à même de remplir la fonction du système en dépit des fautes.
— L'**élimination des fautes** réduit la présence (nombre, sévérité) des fautes.
— La **prévention des fautes** estime la présence, la création et les conséquences des fautes.

Nos travaux s'inscrivent dans le cadre de la sécurité-immunité, qui se définit comme la combinaison des attributs de disponibilité de confidentialité et d'intégrité. Dans la suite, nous décrivons les concepts liés à la sécurité-immunité.

1.4.1.2 Sécurité-immunité

La sécurité-immunité a pour but de protéger un système contre les fautes intentionnelles, dites **malveillances**. Les attributs de la sûreté de fonctionnement définis dans la section précédente sont génériques. Dans cette section, nous allons les appliquer au contexte de la sécurité-immunité. Dans le reste de ce manuscrit, les termes de sécurité ou de sécurité informatique feront référence à la sécurité-immunité, sauf exception explicitement précisée.

Attributs Dans le contexte de la sécurité-immunité, les attributs de la sûreté de fonctionnement peuvent être spécialisés de la manière suivante :

— **Disponibilité** : prévention de rétentions d'information non autorisées.
— **Confidentialité** : prévention de divulgations d'information non autorisées.
— **Intégrité** : prévention de modifications d'information non autorisées.

Malveillances Le projet MAFTIA [23] précise le concept de faute due à l'homme pour la sécurité-immunité. Dans nos travaux, nous nous concentrons principalement sur ces fautes. Il existe deux classes de fautes intentionnelles, ou malveillances : les logiques malignes et les intrusions. **Les logiques malignes** sont des fautes internes intentionnelles, qui sont conçues pour provoquer des dégâts (bombes logiques) ou pour faciliter les futures intrusions par l'ajout de vulnérabilités. Les logiques malignes peuvent être présentes dès la première utilisation du système. [1] ou durant son exploitation par l'installation d'un cheval de Troie ou par une intrusion. Les **intrusions** sont définies conjointement à ses deux causes :

— **Une attaque** est une faute d'interaction externe au système, dont le but est de violer un ou plusieurs des attributs de sécurité. Elle peut être aussi définie comme une tentative d'intrusion.
— **Une vulnérabilité** est une faute qui peut être accidentelle, intentionnelle malveillante ou non malveillante placée dans les exigences, la spécification, la conception ou la configuration du système, ou dans la manière dont il est utilisé.
— Une vulnérabilité peut être exploitée avec une attaque pour créer une **intrusion**. Une intrusion est donc une faute malveillante, initiée depuis l'extérieur pendant l'utilisation du système.

Fautes non intentionnelles dues à l'homme Le projet MAFTIA [23] décrit deux autres types de fautes dues à l'homme, cette fois-ci non intentionnelles, qui peuvent être génératrices de vulnérabilités ou d'intrusions dans les systèmes :

— Les **fautes de conception** sont ajoutées de manière non intentionnelle à la conception du système. Par exemple un firmware peut omettre de configurer certains registres vitaux pour la sécurité [24], l'implémentation d'un protocole peut être vulnérable [25], où certains matériels embarquent encore des fonctions de débogage privilégiées (contrôleur JTAG encore présent).
— Les **fautes d'interactions** sont des fautes externes, dues à la mauvaise utilisation du système. Par exemple, l'utilisateur d'une paire de clés RSA pour le protocole SSH, peut accidentellement envoyer sa clé privée à un administrateur.

Les fautes non intentionnelles ajoutent principalement des vulnérabilités au système ou permettent, par transitivité, des intrusions qui pourront par la suite ajouter à leur tour de

1. C'est à dire ajoutée par le concepteur du système.

nouvelles vulnérabilités. Les fautes d'interaction humaines, bien que très sensibles pour la sécurité des systèmes, ne font pas l'objet de ces travaux, car les traitements associés sont plutôt du domaine de la formation. Pour ce qui est des vulnérabilités ajoutées par les fautes de conception, elles seront potentiellement exploitées de manière malveillante par des attaques. Il est donc indispensable de lutter contre les malveillances pouvant aller à l'encontre des trois attributs de la sécurité-immunité.

1.4.1.3 Moyens de lutte contre les malveillances

Nous pouvons encore une fois spécialiser les méthodes de développement de système sûr de fonctionnement pour les appliquer à la sécurité-immunité.

Prévision des fautes La prévision des fautes peut se dériver en trois méthodes pour les attaques, les vulnérabilités et les intrusions. La **prévision des attaques** consiste à dissuader les utilisateurs malveillants d'attaquer le système. Cela est possible via la loi et la pression sociale par exemple. La **prévision des vulnérabilités** lutte contre l'introduction de vulnérabilités dans la conception du système via l'application de méthodes semi-formelles ou formelles, mais aussi l'éducation de l'utilisateur (choix de mot de passe robuste, utilisation correcte d'un certificat ou d'une paire de clés par exemple). La **prévention d'intrusions** est mise en place via les techniques d'authentification, d'autorisation et des pare-feu. Aussi, par transitivité, la prévision d'attaque et de vulnérabilité va prévenir les intrusions.

Tolérance aux fautes Dans le cas de malveillances, on s'intéresse principalement à la tolérance aux intrusions. Il s'agit d'un ensemble de méthodes que met en place un système pour être capable de détecter une d'intrusion, de se réparer et se reconfigurer tout en continuant de garantir une disponibilité de service et/ou l'intégrité des données pendant une attaque [26]. Dans nos travaux, nous détectons les intrusions à l'aide de tests d'intégrité de l'environnement logiciel et matériel d'une machine. Nous pouvons aussi citer les mécanismes de chiffrement qui conservent la confidentialité, ou encore l'authentification ou l'autorisation.

Élimination des fautes Toujours dans le cadre du projet MAFTIA, les auteurs de [23] avancent que seule l'élimination de vulnérabilités est réellement pertinente. En effet, on ne peut interdire formellement à un utilisateur d'attaquer un système et identiquement, si une vulnérabilité existe, une intrusion peut de toute façon exister. Le développement d'un système est complexe, il est donc très difficile d'éliminer complètement les fautes d'un système. Les concepteurs peuvent donc tenter de réduire leur nombre en appliquant, entre autres, de la vérification formelle ou semi-formelle et des techniques de test.

Prévision des fautes Enfin, la prévision de fautes propose un ensemble de méthodes permettant d'identifier les vulnérabilités, attaques, intrusions potentielles d'un système et de mesurer l'impact des erreurs sur les attributs de la sécurité-immunité sur le système.

Dans section suivante, nous décrivons les problèmes de sécurité qui sont apportés par la virtualisation.

1.4.2 Problèmes de sécurité apportés par la virtualisation

Les architectures virtualisées apportent de nouvelles fonctionnalités pouvant changer radicalement les problématiques de sécurité existantes. Garfinkel et ses associés [27] ont donné une liste de ces fonctionnalités et comment elles peuvent avoir un impact sur la sécurité des systèmes d'information. Nous avons étendu cette liste pour tenir compte du gestionnaire de machines virtuelles, mais aussi des attaques matérielles.

Passage à l'échelle Dans le cadre des infrastructures physiques, l'agrandissement d'un parc informatique pour une société est limité en espace en raison de la place physique que prend une machine ainsi que son équipement d'alimentation, de refroidissement et de stockage, et en temps, car la commande et l'installation avant sa mise en production prennent un temps conséquent. Les machines virtuelles annulent quasiment le temps de mise en production et minimisent la place que celles-ci prennent physiquement dans le parc (fonction du nombre supporté par machine physique). Cette rapide augmentation peut malheureusement provoquer une dégradation de la sécurité. En effet la configuration des outils de sécurité n'est en règle générale pas complètement automatisée, une partie est mise en place par les administrateurs du système eux-mêmes. La responsabilité des utilisateurs est parfois aussi engagée dans le sens où ils ne vont pas systématiquement déclarer l'ajout de machines virtuelles dans le parc. La somme de ces faits augmente la probabilité de présence de machines vulnérables dans le réseau d'entreprise.

Caractère éphémère Par opposition aux machines physiques, les machines virtuelles peuvent apparaître et disparaître tout aussi rapidement sur un réseau. Les réseaux virtuels peinent donc à converger vers un état stable et ils demeurent donc difficiles, voire impossibles à représenter ou modéliser du point de vue des administrateurs système. Pour illustrer ce problème, Garfinkel et al. [27] prennent l'exemple d'un logiciel malveillant, ou *malware*, de type vers. Il est plus aisé de détecter les infections dans un réseau de machines physiques que virtuelles tout simplement du fait que les machines physiques n'ont pas la possibilité d'apparaître sporadiquement sur le réseau pour limiter leur empreinte et donc leur probabilité d'être détectée.

Cycle de vie du logiciel Les logiciels hébergés dans un environnement physique ont un cycle de vie assez linéaire. Ils évoluent incrémentalement dans le temps en termes de version et de manière assez homogène sur les différentes machines du parc. Dans le cas des logiciels exécutés dans une machine virtuelle, sur la totalité du parc informatique, le versionnement est beaucoup plus difficile à contrôler et une multitude de versions différentes d'un logiciel existent en même temps. Ce phénomène est en partie dû à la capacité qu'offrent les techniques de virtualisation aux machines virtuelles de définir des points de contrôle ou *snapshots*, sauvegardant l'état entier d'un disque de machine virtuelle voire de la machine entière. Ces états sont parfois utilisés pour s'assurer du fonctionnement d'un service logiciel donné par une machine virtuelle entre deux redémarrages par exemple. Le problème avec l'utilisation des points de contrôle est que les mises à jour effectuées par le système opératoire durant son exécution seront oubliées au prochain redémarrage d'un point de contrôle, pouvant restaurer certaines vulnérabilités ayant été corrigées. De plus, la restauration d'un *snapshot* peut poser de graves problèmes liés à l'utilisation de protocole nécessitant l'utilisation de nombres aléatoires. De

même, l'utilisation des mots de passe utilisables une seule fois ou *One Time Passwords* (OTP) transmis en clair sur le réseau sera compromise, car ils peuvent éventuellement être rejoués.

Diversité Il est fréquent que la politique de sécurité d'une organisation s'appuie sur l'homogénéisation des machines et logiciels d'un parc informatique. La virtualisation peut mettre en péril cet effort à cause de l'utilisation des points de contrôle, de la non-mise à jour des logiciels (compatibilité ou négligence).

Mobilité Les machines virtuelles ne vont pas forcément effectuer leur cycle de vie entier sur un seul hôte. De ce fait, leur base de confiance informatique ou *Trusted Computing Base* (TCB) va être étendue à tous les hôtes sur lesquels elles vont s'exécuter dans le temps. Les machines virtuelles peuvent être menées à migrer pour différentes raisons. Par exemple une machine virtuelle malveillante peut provoquer la migration de machines virtuelles légitimes d'un hôte si elle parvient à le compromettre et que cette compromission est détectée. Si les machines virtuelles légitimes ont été infectées par un *malware* avant le déclenchement de la migration, ce *malware* peut éventuellement se répandre sur la totalité du réseau. Enfin, les machines virtuelles sont, pour la plupart des systèmes de virtualisation, stockées sous forme de fichiers (disques virtuels et configuration), ce qui simplifie le vol de données dans le sens ou une machine complète peut être dérobée tout simplement en volant un fichier.

Identité d'une machine virtuelle Il est nécessaire d'identifier l'utilisateur responsable d'un évènement sur le réseau pour y appliquer du contrôle d'accès, ou traiter un évènement de sécurité a posteriori. L'identité des machines physiques est en général établie via les identifiants uniques des cartes réseau (adresse MAC). Chaque machine ayant un responsable, il est facile de remonter à un éventuel utilisateur. Le cas des machines virtuelles est beaucoup plus compliqué, car leurs identifiants sont mutualisés entre différents utilisateurs responsables de machines virtuelles. Cela s'aggrave encore si on y applique le paramètre de mobilité, où in fine cet ensemble d'utilisateurs est difficilement identifiable.

Durée de vie des données Limiter la durée pendant laquelle une donnée sensible reste présente dans un système augmente la confidentialité. Les systèmes sécurisés sont aujourd'hui construits dans ce sens. Si nous prenons l'exemple des systèmes cryptographiques, une clé privée est lue uniquement pour chiffrer ou déchiffrer des données et est ensuite effacée de la mémoire. Les hyperviseurs peuvent porter atteinte à la confidentialité en mettant en œuvre les *snapshots* et notamment en créant un point de contrôle. Lors de la création d'un snapshot, la mémoire vive est en partie recopiée dans un fichier (en fonction de l'état des snapshots précédents). Cette copie peut donc rendre persistantes en mémoire des données à caractère volatile. Il est donc important de protéger les points de contrôle des machines virtuelles.

Un composant privilégié supplémentaire Les architectures virtualisées apportent nécessairement une nouvelle couche logicielle qui effectue la séparation en espace et temps entre les machines virtuelles, le gestionnaire de machines virtuelles. Cette couche logicielle est par construction plus privilégiée que les machines virtuelles elles-mêmes. Elle est donc une cible idéale pour des attaques, provenant des machines virtuelles ou du matériel, qui permettent de prendre le contrôle du système ou de rebondir sur d'autres machines virtuelles afin de briser les

propriétés d'isolation. En ce qui concerne les vulnérabilités des gestionnaires de machines virtuelles, identiquement aux systèmes d'exploitation traditionnels, elles peuvent aussi être liées à une mauvaise gestion du cycle de vie du logiciel. Notamment, les processeurs évoluent et embarquent de nouvelles fonctionnalités qui ne sont pas forcément prises en compte. De même les périphériques et chipsets évoluent. La non-configuration ou la configuration par défaut de ces matériels peut engendrer des vulnérabilités exploitables.

1.4.3 Modèle de menaces

Dans cette section, nous proposons un modèle de menace simple et cohérent avec la plupart des gestionnaires de machines virtuelles actuels. Les travaux de Zhang et ses associés proposent un modèle de menace des hyperviseurs, tenant compte du contexte de cloud computing [28]. La figure 1.9 présente ce modèle que nous avons modifié pour considérer les menaces provenant du matériel. Notons que contrairement à notre modèle, l'approche de Zhang et ses associés est plus orientée scénario où l'exploitation d'une même vulnérabilité pouvant atteindre deux composants donne lieu à deux surfaces d'attaque.

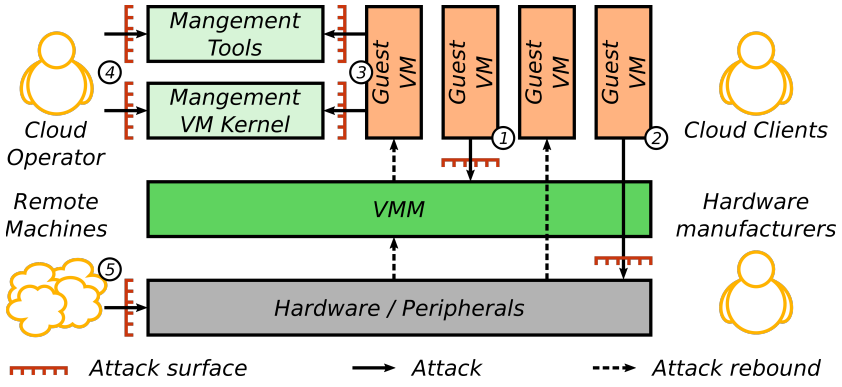

FIGURE 1.9 – Modèle de menaces d'une machine hôte

Les gestionnaires de machines virtuelles sont de logiciels volumineux qui comportent parfois plus d'un million de lignes de code (Xen, KVM, VMware ESXi) et qui s'appuient sur du matériel complexe. Leur surface d'attaque est potentiellement grande et par conséquent, le nombre de vulnérabilités exploitables l'est aussi.

Dans les prochaines sections, nous listons les vulnérabilités identifiées relatives aux surfaces d'attaque que nous considérons dans notre modèle. Nous considérons donc le sous-ensemble de surfaces d'attaques qui a été apporté par la virtualisation et le cloud. Par conséquent, sont exclues de cette étude les vulnérabilités liées aux systèmes opératoires et applications exécutés dans une des machines virtuelles d'un client.

1.4.3.1 Les machines virtuelles du client

Le premier type de vulnérabilités concerne les surfaces d'attaque accessibles depuis les machines virtuelles des clients. Il existe deux surfaces d'attaque principales provenant de ces machines virtuelles. La première est en connexion avec l'hyperviseur (surface d'attaque 1 de la figure 1.9). Elle représente l'interface logicielle implicite (*full virtualization*) ou explicite (*paravirtualization*) entre la machine virtuelle et l'hyperviseur. Les objectifs d'attaque à ce niveau sont divers. Un client du cloud est facturé à l'usage de temps processeur et de bande

passante réseau par exemple. Un utilisateur malveillant peut par exemple tenter de contourner les mécanismes de contrôle d'utilisation de ressources mis en œuvre par l'hyperviseur afin d'alléger la consommation rapportée. L'hyperviseur est le logiciel le plus privilégié de la plupart des architectures supportant la virtualisation. Détourner son fonctionnement en exploitant ses interfaces peut également permettre de porter atteinte à la confidentialité des données d'un autre client stockées dans ses machines virtuelles.

1.4.3.2 Les périphériques

La deuxième surface d'attaque (surface d'attaque 2) concerne les périphériques physiques éventuellement accessibles directement (*pass-through*) ou indirectement par les machines virtuelles (*paravirtualization*). Les périphériques des architectures actuelles sont devenus avec le temps de véritables systèmes dans le système du point de vue de leur complexité. Beaucoup de fonctionnalités sont mises en œuvre par des micrologiciels ou *firmwares*. Les périphériques disposent aujourd'hui de surfaces d'attaques identiques à celles d'un système informatique classique. Ces périphériques disposent d'un accès privilégié sur les bus d'interconnexion des plateformes actuelles. Ils peuvent par ce biais accéder directement à la mémoire principale via le *Direct Memory Access* (DMA). Les périphériques sont donc un moyen efficace, pour les machines virtuelles ayant pu éventuellement contourner leurs protections, d'atteindre les machines virtuelles d'autres clients, ou le gestionnaire de machines virtuelles lui-même.

1.4.3.3 Machines virtuelles et drivers de gestion

Afin de faciliter de développement des systèmes cloud et d'en réduire le coût, les outils de gestion du cloud sont en général déployés sur des systèmes opératoires pris sur étagères (*Component Off The Shelf* ou COTS). Ces systèmes d'exploitation s'exécutent dans des machines virtuelles privilégiées dédiées à l'administration des hôtes du cloud [2] et communiquent via des drivers de paravirtualisation avec l'interface logicielle mise en place par l'hyperviseur pour utiliser les fonctions de l'hôte (démarrer une machine virtuelle, gérer les disques virtuels, etc.). Ces machines virtuelles privilégiées sont une cible idéale pour les attaquants, car elles possèdent toutes les interfaces nécessaires pour porter atteinte aux propriétés de sécurité de toute la plateforme. La première surface d'attaque concerne les opérateurs du cloud, qui peuvent être malveillants ou non légitimes (intrusion dans les bâtiments, vol d'identité numérique, etc.). Depuis les interfaces de gestions de ces machines virtuelles, un opérateur malveillant peut porter atteinte à la sécurité d'un hôte, voire de tous les hôtes du cloud (surface d'attaque 4). Une autre surface d'attaque existe directement entre les machines virtuelles des clients et les machines virtuelles d'administration. Elle existe sous la forme de drivers de noyaux, d'application, etc., exécutés dans une machine virtuelle et dialoguant avec la machine virtuelle de gestion [3] et permettent de faciliter la gestion de la plateforme. Elles sont évidemment maîtrisées par la machine virtuelle privilégiée. Une machine virtuelle de client malveillant peut tenter d'attaquer ces interfaces pour augmenter ses privilèges (surface d'attaque 3).

2. La machine *dom0* pour l'hyperviseur Xen
3. Les VMTools de VMWare par exemple

1.4.3.4 Communications avec l'extérieur

Enfin, une dernière classe d'attaque concerne encore une fois le matériel, non pas son interface avec le bus de la machine hôte, mais plutôt ses interfaces avec le monde extérieur, autrement dit le réseau Internet (point 5). Aujourd'hui les composants matériels de communication, comme les cartes réseaux Ethernet accélèrent la plupart des protocoles réseaux et mettent parfois en œuvre des protocoles d'administration à distance pour décharger le processeur hôte du système. Ses fonctionnalités accessibles depuis des réseaux extérieurs constituent une large surface d'attaque dans le domaine du cloud, car les machines sont largement connectées sur le réseau Internet. Les conséquences de telles attaques en cas de succès peuvent être dramatiques en termes de sécurité à cause de la place privilégiée que prennent les périphériques dans les systèmes actuels.

1.4.4 Attaques des systèmes virtualisés

Dans cette section sont décrites les attaques typiques contre les systèmes virtualisés dans le contexte du cloud. Parmi ces attaques, on retrouve bien sûr celles des architectures classiques, que l'on retrouve aussi sur les systèmes d'exploitation, mais que nous ne décrirons pas ici.

Les attaques contre les gestionnaires de machines virtuelles dans un environnement cloud peuvent être regroupées en différentes classes ([29]) que nous présentons dans la suite.

1.4.4.1 Détection d'un environnement virtualisé

Comme le définissent Popek et ses associés (définition 1.1), une machine virtuelle doit être avant tout une copie fidèle d'une machine physique. Par conséquent, il ne doit pas être possible au système d'exploitation virtualisé de savoir s'il est virtualisé ou non, autrement dit, s'il y a un hyperviseur en cours d'exécution ou non. Ce n'est malheureusement pas le cas, même pour les hyperviseurs actuels effectuant de la virtualisation complète. En règle générale, c'est l'émulation du matériel qui trahit l'hyperviseur. Les changements de contexte induits par l'exécution des instructions privilégiées par la machine virtuelle introduisent tout l'abord des déviations temporelles qui peuvent être détectées même si le matériel dispose aujourd'hui d'extensions pouvant le compenser. De plus, les instructions ne sont pas forcément émulées à l'identique, produisant un écart entre ce qui se passe en réalité sur un processeur donné et sur l'hyperviseur. Un logiciel exécuté dans une machine virtuelle peut tout d'abord prendre conscience de sa virtualisation. Aussi, les mécanismes de virtualisation de l'espace d'adressage des machines virtuelles utilisent des caches de traduction d'adresse [4]. Comme le montre [30], l'exécution de l'instruction `CPUID` sous les processeurs Intel, induit un changement de contexte vers l'hyperviseur qui est détectable temporellement, car ces caches de traduction d'adresses sont vidés pour laisser place à ceux de l'hyperviseur.

1.4.4.2 Identification de l'hyperviseur

Il est possible d'étendre l'attaque précédente en détectant une version précise d'hyperviseur. Ferries, toujours dans [30], cite les recettes de détection de six hyperviseurs pris sur étagère (VMWare, VirtalPC, Parallels, Bochs, Hydra et QEMU). L'auteur montre qu'il existe des différences d'effets de bord engendrés sur le matériel lors de l'exécution d'instructions

4. *Translation Lookaside Buffers* (TLB) pour les architectures x86

spécifiques virtualisées par différents hyperviseurs. Les implémentations spécifiques de chaque projet sur un jeu d'instructions à émuler présentent assez de différences pour identifier très clairement un hyperviseur donné. Parmi celles-ci, les méthodes *RedPill* et *Scoopy Doo* utilisent l'instruction non privilégiée `SIDT` permettant de récupérer l'adresse de la table de handlers d'interruptions sous Intel x86. Cette instruction n'étant pas privilégiée, il est impossible pour l'hyperviseur d'émuler son fonctionnement en prenant la main au moment où elle est exécutée. Le contenu privilégié, maîtrisé par l'hyperviseur, est donc accessible par la machine virtuelle. Comme deux systèmes d'exploitation s'exécutent en parallèle sur un seul cœur, le système invité et l'hyperviseur, cette table d'interruptions doit être partagée par les deux. L'hyperviseur est donc contraint de créer une copie de la table d'interruptions mise en place par la machine virtuelle dans une région mémoire haute afin de garder le contrôle sur l'exécution du cœur et d'aiguiller correctement les évènements matériels en cas d'interruption. C'est cette relocalisation qui permet d'identifier clairement le type d'hyperviseur installé sur la machine. Notons que cette technique ne fonctionne plus sous Intel x86 depuis que VT-x supporte les machines virtuelles de type *Unrestricted Guests* grâce auxquelles la table de handlers d'interruptions n'est plus partagée.

1.4.4.3 Évasion de machine virtuelle : outrepasser l'isolation

Le gestionnaire de machines virtuelles est responsable de l'isolation spatiale et temporelle entre les machines virtuelles : spatiale pour la mémoire, les cœurs de processeur, les interruptions et les périphériques ; temporelle pour la gestion de l'ordonnancement des machines virtuelles sur les cœurs de processeur. Un des objectifs des attaquants de systèmes virtualisés est d'outrepasser cette isolation. Pour ce faire, ils emploient plusieurs méthodes exploitant des vulnérabilités, des problèmes de configuration logicielle ou matérielle. Les attaques contre l'isolation peuvent mener à plusieurs conséquences :

— Dénis de service ou *Denial Of Service* (DOS) : le DOS a pour objectif de s'accaparer ou d'inhiber les ressources processeur d'une machine donnée afin de l'empêcher de rendre correctement son service. Dans le cas des machines virtuelles, l'attaquant va tenter de s'accaparer le temps processeur en détournant la politique d'ordonnancement du gestionnaire de machines virtuelles ou encore en modifiant directement le code de l'hyperviseur de façon à ce qu'il n'effectue plus l'ordonnancement.
— Extinction du système ou *system halt* : cette attaque est un cas particulier du DOS ou l'utilisateur malveillant se rend en mesure d'exécuter la séquence de code privilégiée pouvant éteindre la machine, de façon à empêcher l'exécution de toutes les machines virtuelles.
— Évasion de machine virtuelle ou *VM escape* : cette catégorie d'attaque est la plus invasive des trois et concerne les machines virtuelles malveillantes qui arrivent à briser l'isolation spatiale mise ne place par le gestionnaire de machines virtuelles. L'attaquant arrive à accéder à la mémoire des autres machines, des périphériques physiques et de l'hyperviseur lui-même.

Il existe des cas d'attaques documentés pour les trois grands hyperviseurs pris sur étagère Xen, VMWare [5] et KVM [31], [32], [33].

5. Actuel VMWare workstation, hyperviseur de type 2 par opposition à VMWare ESXi qui est un hyperviseur de type 1

Wojtczuk, dans [31], exploite les capacités DMA de matériels pris sur étagère et systématiquement présents dans les architectures actuelles, pour modifier le code de l'hyperviseur Xen depuis un utilisateur root malveillant du `dom0`, dans le but d'installer une librairie de chargement de code arbitraire dans l'hyperviseur Xen.

Dans le cas de [32], l'auteur remet en cause la sécurité de la carte vidéo virtuelle développée par VMWare. Probablement à cause de la complexité de ce type de logiciel, des vulnérabilités ont été identifiées au niveau du protocole de rendu 2D de la carte virtuelle. Il est possible par exemple de lire et d'écrire en dehors du framebuffer, et de ce fait d'accéder à la mémoire de l'hyperviseur hôte du système, depuis la machine virtuelle. Des vulnérabilités ont aussi été découvertes au niveau du protocole de rendu 3D. L'auteur exploite ces vulnérabilités et montre qu'il est possible, in fine, de contrôler la machine au détriment de l'hyperviseur qui était en place.

En ce qui concerne l'hyperviseur KVM, lors de son utilisation conjointe avec QEMU, l'auteur de [33] arrive à contrôler l'exécution d'un programme *root* s'exécutant dans le système d'exploitation hôte (linux) depuis un programme *root* s'exécutant dans la machine virtuelle. L'attaque s'appuie sur l'exploitation d'une vulnérabilité concernant les périphériques virtuels émulés (voir section 1.2.2, virtualisation logicielle des périphériques).

1.4.4.4 Corésidence

Pour agrandir la surface d'attaque d'une victime dans le cloud, un attaquant peut tenter de détecter la position d'une machine virtuelle dans l'infrastructure physique et lancer une machine virtuelle sur la même machine physique [6].

Les auteurs de [34] étudient le cas du cloud d'Amazon EC2 [35]. Ils démontrent qu'il est possible après avoir cartographié le réseau du cloud et défini des indicateurs déterministes de corésidence d'une machine d'attaque avec une machine victime, d'effectuer des attaques par canaux cachés pour porter atteinte à la confidentialité d'un client depuis la position d'un autre client malveillant.

Dans un premier temps, pour diminuer l'espace de recherche de corésidence, les auteurs démontrent qu'il y a une corrélation forte entre les adresses IP privées des machines que l'on démarre dans le cloud et leur couple (configuration de machine virtuelle / zone géographique sélectionnable à la création). Ils donnent une liste d'heuristiques qui serviront à la prochaine étape de détection de corésidence pour diminuer l'espace de recherche.

La deuxième étape consiste à détecter la corésidence de deux machines virtuelles en utilisant les heuristiques définies précédemment, c'est-à-dire deux machines virtuelles ne sont pas en corésidence si et seulement si leur configuration et zone géographiques ne sont pas les mêmes. Les auteurs arrivent à extraire un couple d'indicateurs déterministes pour détecter très précisément la corésidence de la machine virtuelle d'un attaquant et de sa victime : les deux adresses IP sont numériquement distantes d'au plus 7 et elles partagent le même routeur (ici le *dom0* de Xen).

Ensuite, les auteurs utilisent ce résultat pour élaborer deux méthodes de placement précis d'une machine virtuelle sur une machine physique. La première consiste à deviner par force brute la création / suppression de machine virtuelle jusqu'à ce que la corésidence soit établie. Une deuxième méthode moins gourmande en temps à aussi été proposée. Elle se base sur un résultat intéressant, trouvé empiriquement lors de leurs expériences : lorsqu'une victime

6. Dans le cas de cloud multi tenants

démarre une machine, l'attaquant dispose d'environ une chance sur deux de succès de corésidence avec celle-ci en lançant deux dizaines de machines à ce moment précis. L'auteur ajoute qu'il existe des méthodes que les attaquants peuvent utiliser pour déclencher le démarrage de machines.

Enfin, une fois la corésidence établie, l'attaquant est en mesure de lancer des attaques par canaux cachés sur la machine virtuelle victime pour porter atteinte à son intégrité.

1.4.4.5 Rootkits hyperviseurs

Une dernière catégorie d'attaque regroupe les rootkits hyperviseurs. Elle diffère des attaques d'évasion de machine virtuelle qui exploitent des vulnérabilités dans les mécanismes d'isolation mis en place par l'hyperviseur. Dans le cas des rootkits, l'attaquant exploite des fonctionnalités matérielles [7] normalement utilisées par l'hyperviseur légitime pour s'installer en dessous de celui-ci. En effet, avant la version 3.0 de l'hyperviseur Xen, celui-ci n'utilisait pas les extensions matérielles pour la virtualisation, mais plutôt la technique de virtualisation hybride présentée en section 1.2.2. Ces extensions apportent de nouveaux modes d'exécution, que l'on peut appeler mode hyperviseur, plus privilégié que le mode noyau (ring-0 pour le processeur x86 ou superviseur pour ARM v8). Comme le montre [36], si une machine virtuelle arrive à s'exécuter en ring-0 en exploitant une vulnérabilité dans l'interface avec l'hyperviseur (depuis le `dom0` ou une machine virtuelle non privilégiée), elle pourra activer les extensions matérielles pour la virtualisation des cœurs de processeur au prochain redémarrage avant l'hyperviseur Xen légitime, pour le virtualiser en utilisant les techniques de virtualisation récursive, ce qui permet de contrôler son exécution et celui des machines virtuelles sous-jacentes.

Comme nous l'avons vu dans cette première section, l'hyperviseur est un logiciel privilégié et indispensable aux architectures virtuelles, et au cloud par extension. Il demeure donc une des cibles privilégiées des attaquants. Ce constat est confirmé dans la littérature, car comme nous l'avons également montré, il existe des attaques contre les hyperviseurs depuis que ces architectures virtuelles sont utilisées en production. Ces hyperviseurs, qui sont la plupart du temps très volumineux en termes de lignes de code source, sont malheureusement souvent développés en priorité pour répondre aux besoins fonctionnels, mettant entre parenthèses la question de la sécurité. Il existe donc des projets visant à améliorer la sécurité des hyperviseurs, des systèmes d'exploitation ou des logiciels privilégiés en général. D'autres projets s'intéressent à la sécurité des machines virtuelles elles-mêmes. Nous discutons de ces solutions dans la section suivante.

1.5 La sécurité dans les architectures virtualisées

Il existe de nombreux travaux dans la littérature qui proposent d'améliorer la sécurité des architectures virtuelles. Nous les présentons dans cette section, en insistant sur ceux qui sont les plus proches des travaux de cette thèse. De plus, à la manière de ce qui est présenté par exemple dans [37] et [29], nous séparons ici ces solutions en deux grandes catégories : la surveillance des machines virtuelles et le renforcement de l'isolation.

7. Extension Intel VT-x ou AMD-V

1.5.1 Surveillance des machines virtuelles

Le fait d'avoir un hyperviseur situé en dessous des systèmes d'exploitation invités et possédant tous les privilèges permet d'utiliser ce dernier pour surveiller de façon externe les activités au sein de ces systèmes. Cette partie donne des exemples d'applications d'un tel procédé.

1.5.1.1 Introspection de machines virtuelles

Les techniques dites d'introspection de machine virtuelle (VMI, *Virtual Machine Introspection*) consistent à utiliser le fait que l'hyperviseur ait accès aux zones de la mémoire allouées aux machines virtuelles pour pouvoir analyser le contenu de celles-ci depuis l'extérieur de ces machines virtuelles. L'accès à ces données et leur analyse peuvent se faire directement depuis l'hyperviseur (ce qui recommandé, car l'impact sur le code de l'hyperviseur est moindre) ou depuis une machine virtuelle dédiée à cette tâche. Cependant, l'hyperviseur n'a pas connaissance des abstractions utilisées par les OS invités et ne peut donc pas comprendre le contenu de ces zones de la mémoire. C'est le "fossé sémantique" que les programmes de VMI vont chercher à combler. Il existe de nombreuses implémentations de VMI ou de modules permettant son application (comme LibVMI pour Xen et KVM par exemple) en fonction des besoins.

1.5.1.2 Observation du noyau

Virtualiser un système d'exploitation peut permettre de sortir celui-ci du plus haut niveau de privilèges en l'exécutant au-dessus d'un hyperviseur. Ce dernier peut alors contrôler l'exécution de l'OS depuis l'extérieur.

SecVisor [38] constitue un tel exemple d'hyperviseur spécialement conçu pour assurer l'intégrité du code d'un OS. C'est un gestionnaire de machines virtuelles relativement petit (moins de 5000 lignes de code) dont le but est d'autoriser uniquement l'exécution de code validé par l'utilisateur. La conception de SecVisor a été prévue pour répondre à trois critères :

1. Une petite quantité de code afin de faciliter une vérification formelle ou un audit manuel.
2. Une interface externe limitée afin de réduire la surface d'attaque.
3. Un nombre de changements à apporter au noyau surveillé le plus petit possible pour faciliter la portabilité.

SecVisor n'empêche pas l'ajout de code dans le noyau, mais il ne permet pas son exécution s'il n'a pas été approuvé. Il se base pour cela sur des mécanismes intégrés au processeur, que ce soit en termes de gestion de la mémoire ou de virtualisation (permission *write xor execute* pour le code du noyau et l'IOMMU pour les accès des périphériques).

SIMA[39] propose une approche différente nécessitant l'instrumentation du système d'exploitation invité, basée sur des capteurs "sensoriels", visant à extraire des informations pertinentes qui seront ensuite traitées. Ces informations renseignent l'hyperviseur sur l'état du système invité et peuvent éventuellement conduire à des contre-mesures en cas d'attaque. Les principaux objectifs de SIMA sont : surveiller activement, durant la phase d'exécution, l'OS invité ; détecter les logiciels malveillants dans sa mémoire ; cacher la présence de l'hyperviseur et des capteurs aux yeux de l'OS.

HIMA [40], Hytux [41], [42] et BMCS [43] sont aussi des systèmes de surveillance du système invité pour la phase d'exécution. Ils diffèrent légèrement du précédent, car ils n'instrumentent pas l'OS invité. Ils sont capables de combler seuls le vide sémantique des évènements

bas niveau interceptés lors de la surveillance. De ce fait, ils proposent un agent de surveillance en théorie parfaitement isolé du code des machines virtuelles potentiellement corrompues.

Patagonix [44] est un hyperviseur conçu pour surveiller un système d'exploitation et détecter les rootkits dissimulant leur exécution via des mécanismes indépendants de l'OS surveillé. Pour ce faire, le code de l'OS virtualisé se voit marqué comme non exécutable, ce qui entraîne un appel de l'hyperviseur à chaque fois qu'une exécution est requise (uniquement pour la première exécution ou après une modification). Dès lors, le code est comparé avec une base de binaires connus. Ainsi, en comparant les rapports fournis par Patagonix et ceux fournis par l'OS surveillé, il est possible de détecter la modification de binaire de l'OS ou un programme cherchant à s'exécuter discrètement. Avec une telle manière de traiter le fossé sémantique, Patagonix peut surveiller théoriquement n'importe quel système, dans la mesure où il possède une base de binaires de référence suffisante.

1.5.2 Renforcement de l'isolation

Un autre axe de recherche se focalise sur des moyens de protéger le gestionnaire de machines virtuelles contre d'éventuelles modifications. Là aussi, plusieurs pistes sont explorées.

1.5.2.1 Intégrité au démarrage

L'intégrité de l'hyperviseur chargé au moment du démarrage peut être assurée par les mécanismes classiques proposés par le *Trusted Computing Group* (TCG), comme le démarrage sécurisé (*secure boot*) assisté par la puce *Trusted Platform Module* (TPM), pour garantir l'intégrité d'un système d'exploitation chargé durant la phase de boot[45, chap. 27]. Il n'y a cependant rien de prévu pour garantir l'intégrité des systèmes invités. C'est un des objectifs de l'hyperviseur BitVisor[46], qui va étendre la chaîne de confiance jusqu'au chargement du système invité. Pour ce faire les auteurs ont divisé leur hyperviseur en deux modules : un pour déchiffrer et charger l'image du système invité et un deuxième pour la phase d'exécution. Cette séparation a aussi l'effet bénéfique de minimiser la taille de l'hyperviseur gérant la phase d'exécution.

1.5.2.2 Modules de sécurité

Cette catégorie concerne les extensions développées pour ajouter des fonctionnalités de sécurité au sein de l'hyperviseur. La principale contribution dans cette catégorie est représentée par Xen Security Modules [47], un framework implémentant de nouvelles règles de sécurité pour Xen que l'administrateur peut choisir ou non d'activer.

1.5.2.3 Protection de l'espace de stockage

Les hyperviseurs disposent d'un contrôle total sur le matériel vis-à-vis des machines virtuelles. C'est grâce à cela que Rezaei et al. ont proposé une version modifiée de BitVisor, TCvisor[10], de manière à intégrer des capacités de cryptographie de l'espace de stockage. Le chiffrement des disques durs peut s'effectuer à trois niveaux. 1) au niveau applicatif où l'utilisateur est responsable de la bonne utilisation des applications de chiffrement, ce qui ne garantit pas un niveau de sécurité correct ; 2) au niveau noyau où le processus de chiffrement peut être plus facilement appliqué de manière systématique, cette application restant vulnérable

aux attaques classiques sur les OS ; 3) au dernier niveau, introduit par l'assistance matérielle à la virtualisation, qui garantit une isolation correcte du processus de chiffrement. De plus le mécanisme est générique, c'est-à-dire non dépendant du système d'exploitation. C'est ce mécanisme qui a donc été choisi pour TCvisor.

1.5.2.4 Protection du gestionnaire de machines virtuelles

On peut utiliser un hyperviseur pour surveiller l'exécution de systèmes virtualisés. Cependant, l'hyperviseur peut à son tour être modifié par des attaques. Celui-ci étant l'élément le plus privilégié du système, sa sécurité est donc primordiale. Cependant, s'il est possible de s'assurer de l'intégrité d'un système lors de son lancement (démarrage sécurisé), il devient plus difficile de conserver cette intégrité au cours de son exécution. Ainsi, si l'on veut s'intéresser à la vérification de l'intégrité de l'hyperviseur, une approche similaire à celle vue dans 1.5.1.2 peut être envisagée en plaçant celui-ci au-dessus d'un autre hyperviseur. Mais là encore, rien ne nous garantit a priori que l'hyperviseur le plus privilégié ne peut être corrompu. Des études ont donc cherché à s'assurer de l'intégrité du système le plus privilégié par d'autres moyens.

Par exemple, HyperSentry [48] fournit un environnement de mesure de l'intégrité d'un hyperviseur (ou d'un autre système ayant les plus hauts niveaux de privilège) qui ne cherche pas à se situer en dessous de celui-ci. Pour ce faire, il utilise le *System Management Mode* (SMM) pour protéger le code de l'hyperviseur et les méthodes de démarrage sécurisé (*secured boot*) définies par le TCG pour s'assurer de l'intégrité de son code au démarrage. De plus, l'usage de canaux de communication hors de contrôle du gestionnaire de machines virtuelles (IPMI [49] dans leur exemple) permet de déclencher une analyse sans que l'hyperviseur ne soit averti. Dès lors, HyperSentry peut accéder à l'état de l'hyperviseur à un instant donné sans que ce dernier en ait connaissance et permet donc théoriquement de fournir des données non altérées aux outils d'analyse.

D'autres recherches visent à protéger l'hyperviseur d'attaques malveillantes via la mise en application de certains principes. Ainsi Hypersafe [50] vient modifier des hyperviseurs existants pour implémenter deux mécanismes (*memory lockdown* et *restricted pointer indexing*) visant à garantir l'intégrité de leurs structures de contrôle. Le but est que l'hyperviseur se protège lui-même contre des modifications ou des exécutions non prévues de son code. Hypersafe a déjà été implémenté sur BitVisor et (partiellement) sur Xen. TinyChecker[51] quant à lui virtualise l'hyperviseur à protéger (en utilisant la *nested virtualisation*) et surveille périodiquement la présence et l'intégrité de celui-ci. Il est capable en cas de corruption de le rebooter vers un état temporellement proche. Ce reboot préserve les machines virtuelles en cours d'exécution.

Si de tels outils permettent globalement d'améliorer la sécurité des hyperviseurs et des machines virtuelles qu'ils gèrent, leur utilisation implique en contrepartie une augmentation de la quantité de code s'exécutant avec de hauts privilèges (puisque faisant partie de l'hyperviseur). Une erreur de conception de ces modules peut alors faire apparaître de nouvelles vulnérabilités, par exemple dans [52] où une attaque est rendue possible à cause d'une erreur dans le code du module FLASK de XSM, qui n'est pas installé par défaut dans Xen. Il est donc également important d'envisager des mécanismes permettant aux machines virtuelles de se protéger de l'hyperviseur.

1.5.2.5 Protection des machines virtuelles contre l'hyperviseur

CloudVisor [53] est conçu pour assurer la confidentialité et l'intégrité des données des machines virtuelles, dans le cas où des éléments de l'interface de virtualisation (hyperviseur, machine virtuelle d'administration, autre machine virtuelle invitée) sont corrompus. Le but est que tout accès aux données d'une machine virtuelle ne provenant pas de celle-ci ne puisse en obtenir qu'une version chiffrée. Cependant, CloudVisor ne cherche pas à protéger une machine virtuelle contre des attaques extérieures à la couche de virtualisation. Pour ce faire, CloudVisor va virtualiser l'hyperviseur à surveiller et sortir ce dernier de la zone la plus privilégiée tout en lui donnant l'impression qu'il contrôle toujours complètement les machines virtuelles. Ce faisant, le VMM s'exécute en dehors du mode hyperviseur et seul CloudVisor est dans ce mode (sortant ainsi l'hyperviseur surveillé de la zone la plus critique). Il intercepte alors les accès du VMM à la mémoire d'une machine virtuelle et chiffre le contenu d'une page si l'accès n'est pas demandé par son propriétaire. Un tel concept est très intéressant dans le cadre du cloud computing où des utilisateurs différents vont être amenés à partager les ressources d'une même machine physique, la confidentialité des données représentant la préoccupation principale des clients d'un tel service.

1.5.2.6 Protection des données des machines virtuelles contre les utilisateurs

BitVisor[54] utilise aussi sa position privilégiée afin de garantir une politique d'accès orientée rôles (RBAC) aux machines virtuelles installées sur une même machine physique. L'identification du rôle d'un utilisateur s'effectue via une carte à clé privée contenant le(s) groupe(s) au(x)quel(s) appartient un utilisateur. Ainsi les administrateurs sont capables de définir précisément quels systèmes et applications peuvent être démarrés et utilisés par différents groupes utilisateurs. Le même principe est utilisé pour ce qui est de l'accès au matériel et aux ressources réseau (VPN transparent, chiffrement transparent des disques, etc.). L'utilisateur aura accès à un jeu de machines virtuelles qui correspondent au niveau de privilège de son groupe.

1.5.3 Vulnérabilité du logiciel et confiance dans le matériel

Les approches citées précédemment ne sont pas suffisantes de notre point de vue, car elles ont besoin de faire confiance au processeur, aux divers composants matériels pris sur étagère et aux différents *firmwares* qui constituent leurs environnements d'exécution.

1.5.3.1 Vérification des micrologiciels

Dans le but de renforcer la confiance dans le matériel, d'autres travaux se focalisent sur la vérification des micrologiciels (*firmware*. Cette couche logicielle est essentielle pour l'exécution correcte du système entier et notamment au démarrage. Par exemple, VIPER [55] est un outil dédié dans le test de l'intégrité des *firmwares* embarqués dans les périphériques. VIPER s'exécute au-dessus d'un système d'exploitation et utilise des challenges cryptographiques soumis aux périphériques, de manière à détecter l'altération de leurs *firmwares*. IOCheck [56] est une solution similaire à VIPER, mais plutôt installée dans le mode de management des machines Intel (SMM). Par conséquent, il n'a pas besoin de faire confiance à un système d'exploitation pour rendre son service.

1.5.3.2 Extensions matérielles spécialisées dans la sécurité

Les fabricants de processeurs ont aussi proposé des extensions de leurs architectures qui sont dédiées à la sécurité. ARM a implémenté la technologie TrustZone [57], qui est depuis la troisième génération des processeurs de gamme Cortex-A. TrustZone ségrègue les logiciels en deux domaines d'exécution, le *normal world*, plus privilégié que le *secure world*. Le *secure world* apporte un environnement d'exécution sécurisé pour de petits noyaux de sécurité. Sans compter les potentiels bugs matériels, avec ARM TrustZone, les logiciels hautement privilégiés sont effectivement bien protégés contre les tentatives de corruption directes par des logiciels au niveau de privilèges moins élevés. Or, TrustZone ne protège pas les logiciels contre l'exploitation de vulnérabilités introduites dans le code protégé lui-même, quel que soit le niveau de privilège dans lequel il est exécuté, et ce aussi longtemps qu'il existe une interface pour accéder aux vulnérabilités.

L'extension Intel SGX [58] apporte ensemble de nouvelles instructions qui permettent aux applications logicielles de créer un environnement d'exécution privé qui est aussi appelé enclave de sécurité par Intel. Le problème avec cette technologie est que les applications protégées par SGX font face aux mêmes problèmes qu'avec TrustZone de ARM. Les applications interagissent aussi avec leur environnement, et peuvent être aussi corrompues en raison de ces interactions. Si le code lui-même contient des vulnérabilités, Intel SGX ne peut pas empêcher leur exploitation.

1.5.3.3 Vérification depuis le matériel

Copilot [59], un coprocesseur de sécurité connecté comme un périphérique, aborde le problème de la confiance dans le processeur et le matériel. Il offre une solution de vérification d'intégrité entièrement externe, par le calcul et la vérification des pages mémoires possédées par un système d'exploitation. Cette approche est la plus proche de l'architecture que nous proposons dans le chapitre 3, mais n'est pas suffisante selon notre point de vue, car elle fait aveuglément confiance à son environnement. En effet, la situation suivante peut survenir : le logiciel surveillé est correct, mais un logiciel malveillant a réussi à prendre le contrôle du comportement du logiciel surveillé sans le modifier, en étant exécuté dans un niveau de privilèges plus élevé ou en étant installé ailleurs dans la mémoire. De plus, le fossé sémantique ou *semantic gap* [44], de l'interprétation de la mémoire entre un périphérique externe et un noyau de système d'exploitation est difficile à traiter. Par exemple, certains registres internes du processeur sont essentiels pour comprendre quel composant logiciel est installé comme hyperviseur, et il est impossible d'accéder au contenu de ces registres depuis un périphérique externe. Une approche basée sur la collaboration de logiciels et de composants matériels est, de notre point de vue, plus réaliste pour exécuter des challenges plus complexes et efficaces. Enfin, les solutions basées uniquement sur la vérification d'empreinte mémoire ne peuvent pas passer à l'échelle sur des serveurs possédant une grande quantité de mémoire et exécutant de nombreuses machines virtuelles par exemple.

1.6 Objectifs de la thèse

Dans ce chapitre, nous avons fait un état des lieux des différents problèmes de sécurité qui sont liés à la virtualisation des systèmes informatiques. Nous avons également proposé un aperçu des différentes solutions existant aujourd'hui pour faire face à ces problèmes. Même si

ces solutions sont intéressantes, nous faisons le constat qu'elles se basent massivement sur du logiciel et que le logiciel est vulnérable et assez facilement attaquable, même lorsqu'il s'exécute dans les couches les plus privilégiées du processeur. Aucun logiciel n'est aujourd'hui à l'abri de l'exploitation d'un bug du processeur, ou d'une attaque matérielle provenant d'un périphérique malveillant par exemple. Face à cette constatation, notre proposition est donc de concevoir une architecture de sécurité **utilisant conjointement du logiciel et du matériel**. Nous partons de l'hypothèse qu'un composant matériel est toujours beaucoup difficilement contournable et modifiable que du logiciel, surtout si ce matériel est totalement **indépendant du processeur principal de la machine sur lequel il est connecté**. Cette architecture, son implémentation ainsi que des tests d'efficacité et de performances sont proposés dans les chapitres 3, 4 et 5 de cette thèse, après quelques explications sur les mécanismes de virtualisation des architectures Intel, présentées dans le chapitre 2.

Chapitre 2

Étude des architectures matérielles actuelles

Sommaire

2.1 Introduction

Ce chapitre est dédié à l'étude de l'architecture matérielle Intel sur laquelle nous avons travaillé durant ces travaux de thèse. Dans une première section, nous étudions les composants principaux et leurs interactions pour dans un second temps discuter des différents mécanismes de contrôles d'accès existants. Nous terminons ce chapitre avec des détails techniques nécessaires à la bonne compréhension de ce manuscrit de thèse.

2.2 Les processeurs Intel récents

Dans cette section, nous présentons les composants principaux des machines Intel core de 4^{e} génération, Haswell, associées au *chipset* Intel C220 et étudions les interactions entre eux.

Dans le cas des processeurs Haswell, l'objet commercialisé appelé processeur contient 4 cœurs d'exécution, des périphériques tels que le processeur graphique, et surtout le composant appelé *System Agent*. Ce dernier composant, anciennement appelé *north bridge* puis *Memory Controller Hub* (MCH), a été intégré physiquement dans le processeur depuis l'architecture Intel Nehalem. Nous l'appellerons contrôleur mémoire dans ce chapitre. Il permet la communication inter cœurs via le *ring bus*, avec la mémoire, le processeur vidéo intégré, des périphériques PCI Express hautes performances, mais aussi avec d'éventuels autres périphériques. Ces

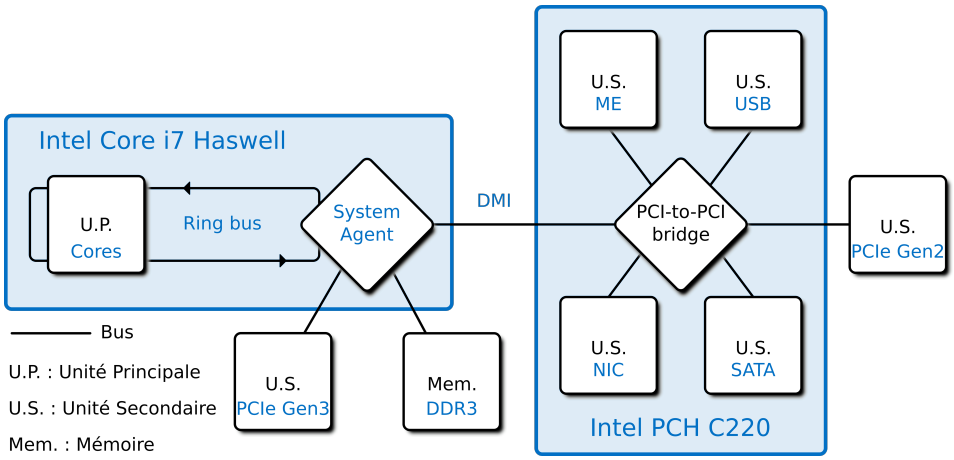

FIGURE 2.1 – Implémentation du modèle sur Intel Haswell et PCH Intel C220

dernières communications sont rendues possibles grâce au bus *Direct Media Interface* (DMI) qui relie le processeur au deuxième circuit principal de la machine, le *Platform Controller Hub* (PCH), remplaçant du *south bridge* (figure 2.1).

Le PCH Intel C220, communiquer avec le processeur grâce à son pont PCI-to-PCI, en traduisant les messages DMI vers et depuis d'autres bus, registres, ou protocoles tels que PCI Express. Il intègre aussi d'autres périphériques comme un contrôleur Ethernet Giga bits, un contrôleur Serial ATA, un contrôleur USB, mais aussi un coprocesseur particulier, le *Management Engine* (ME) et son *firmware*.

Le *Management Engine* est utilisé pour rendre possible l'administration de machines à distance, sans interagir avec le système opératoire exécuté par le processeur, et dans des modes de gestion d'énergie potentiellement dégradés [1].

Enfin, il est possible d'étendre le nombre de périphériques via le protocole PCI Express. Le *chipset* C220 propose 8 ports racines PCI Express 2 permettant de connecter des périphériques supplémentaires, généralement par le biais de slots d'extension.

Une particularité des *chipsets* PCH Intel est l'intégration du contrôleur d'affichage analogique (VGA) qui va effectuer la modulation du signal pour l'écran à partir de la mémoire vidéo (*framebuffer*). Or cette mémoire vidéo est située dans le processeur graphique, lui-même situé dans le processeur, donc hors du PCH. Cette communication est rendue possible par un bus dédié au simple transport du contenu de la mémoire vidéo, le *Flexible Display Interface* (FDI) [60, Vol 2. 1.2.1, 5.28.3, Vol 1. 2.7]. Par souci de simplicité et en raison du peu d'ouverture de cette interface, nous avons décidé de ne pas la prendre en compte dans notre analyse.

En raison de la volonté de rétrocompatibilité d'Intel, l'architecture étudiée est très configurable, supportant des modes de fonctionnement dépréciés qui ne sont plus utilisés à l'exécution. De plus, certaines parties de l'architecture non programmables par le logiciel ne sont peu, plus, voire pas documentées, exigeant de déduire la nature de certains composants et bus de communication. Cette remarque concerne notamment la gestion des interruptions inter-unités d'exécution, plus particulièrement dans le PCH. Comme le montre la figure 2.2, un périphérique comme le contrôleur SATA dispose de multiples routes pour interrompre l'exécution d'un cœur [61, 3.6.2]. En mode opératoire classique, il est possible de directement transmettre une

1. de la suspension de tous les cœurs (S1) à l'hibernation sur disque (S4)

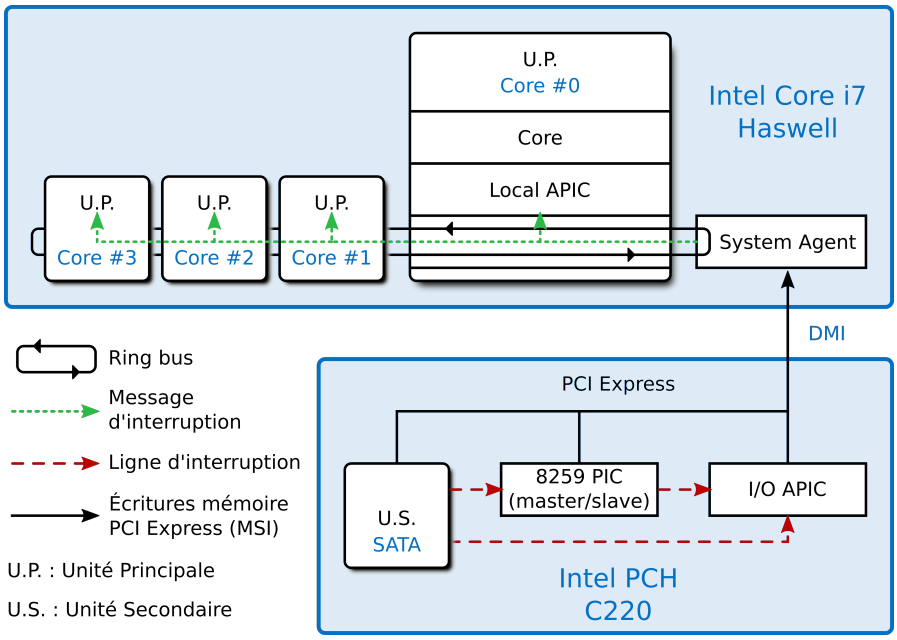

FIGURE 2.2 – Mécanismes internes de gestion des interruptions

interruption PCI Express, appelée *Message Signaled Interrupt* (MSI). Ces messages, prenant la forme d'une écriture à destination du gestionnaire d'interruption local, le *local APIC* d'un ou plusieurs cœurs [2], sont transmis au processeur par le bus DMI. Il peut aussi effectuer une requête d'interruption auprès du contrôleur d'interruption historique, le *Programmable Interrupt Controller 8259* (8259 PIC). Enfin, l'*I/O Advanced Programmable Interrupt Controller* (I/O APIC) est maintenant présent dans les architectures récentes et peut gérer plus d'interruptions. Une fois la requête d'interruption acceptée par le 8259 PIC ou l'I/O APIC, un pont aux bornes de ces composants émet l'écriture PCI Express correspondant au cœur à interrompre in fine [3] [60, 5.10.1]. La route que prend une interruption générée par un périphérique dépend de sa configuration ainsi que de celle du PCH. Notons que l'I/O APIC peut aussi fonctionner en cascade avec le 8259 PIC. Une fois que l'interruption parvient au processeur, elle est traitée par le *System Agent* et émise sur le bus en anneau. Les interruptions inter cœurs y transitent de la même manière.

Cette gestion des interruptions hors processeur est complexe et aboutit dans tous les cas au même type de messages d'écriture PCI Express. Dans cette étude, nous considérons uniquement les MSI et les émissions d'interruption de l'I/O APIC si nécessaire.

Voici les types différents types de communications pour l'implémentation étudiée.

1. Les accès mémoire des cœurs visant la mémoire principale sont traduits en commandes de lecture et d'écriture DDR3 par le contrôleur mémoire intégré au processeur. Ces commandes DDR3 sont directement émises depuis le contrôleur mémoire intégré au processeur.
2. Les accès mémoire des cœurs en direction des périphériques de par la topologie de cette architecture Intel, sont traités de différentes manières en fonction de la destination. Les

2. Dans le cas de l'hyperthreading, chaque cœur physique dispose de deux APIC locaux
3. Uniquement le cœur #0 (ou *Bootstrap Processor* (BSP)) dans le cas du 8259 PIC

accès au processeur graphique intégré sont transmis par le bus en anneau. Les accès aux périphériques PCI Express locaux sont traduits du bus en anneau en requêtes PCI Express. Enfin, les accès aux périphériques hébergés par le PCH vont être traduits en accès DMI, puis PCI Express le cas échéant [60, 9.4].

3. L'architecture étudiée contient plusieurs cœurs d'exécution. La communication entre ces cœurs est rendue possible grâce aux interruptions. Celles-ci sont émises et reçues par les gestionnaires d'interruptions locaux, les *local APIC*, et transmises sur le *ring bus*. Un signal vers un autre cœur est initié par une écriture en mémoire ou dans un registre interne.
4. Il existe deux types d'accès mémoire des périphériques vers la mémoire. Les accès qui proviennent des périphériques gérés par le contrôleur mémoire et ceux qui viennent du PCH, traversant le bus DMI. Dans les deux cas, ils sont traduits en accès mémoire DDR3 par le contrôleur mémoire.
5. Les périphériques vont devoir notifier le cœur lors de la terminaison d'un traitement demandé ou de l'occurrence d'un évènement attendu, par exemple l'arrivée d'une trame Ethernet. Pour ce faire, ils vont générer une requête d'interruption. En fonction du périphérique source, elle est reçue depuis le bus DMI ou traduite depuis les *end points* PCI Express locaux au processeur. Les requêtes seront enfin dirigées vers un des gestionnaires d'interruption (*local APIC*) d'un des cœurs du processeur via le bus en anneau.
6. Sur l'architecture étudiée, les communications directes entre les périphériques intégrés Intel sont plus difficilement identifiables par manque de documentation de par leur caractère propriétaire. Il est cependant possible d'étudier le cas des périphériques PCI Express connectés aux slots d'extension. Les périphériques PCI Express sont organisés en étoile. Enfin, nous pouvons dire qu'en règle générale, les accès mémoires pair-à-pair directs ne sont pas autorisés. S'ils sont nécessaires, ils doivent passer par l'intermédiaire du processeur.

2.3 Mécanismes d'isolation et de contrôle d'accès

L'architecture Intel Core Haswell associée au Intel PCH C220 met en œuvre un nombre considérable de mécanismes de contrôle d'accès qui peuvent être mis à profit pour garantir la sécurité du système. Nous nous intéressons dans ce manuscrit essentiellement à ceux qui sont visibles du point de vue du développeur de logiciels.

2.3.1 Les cœurs

Les cœurs disposent de différents modes de fonctionnement. Les cœurs de processeurs Haswell implémentent l'architecture Intel 64 qui propose le mode d'exécution 64 bits utilisé actuellement à l'exécution. Or, pour des raisons de rétrocompatibilité, ils supportent toujours une multitude de modes plus anciens, moins performants [4], dans lesquels les cœurs démarrent avant d'être reconfigurés en mode 64 bits. Nous allons les étudier dans leur ordre d'apparition historique, qui coïncide avec leur ordre d'utilisation au démarrage.

Les cœurs démarrent dans le mode historique appelé mode réel. Il constitue l'environnement d'exécution des premiers logiciels développés pour le PC IBM qui utilisait un processeur Intel

4. moins de mécanismes d'isolation, jeu d'instruction réduit, etc.

8088 [5]. Ce mode n'est aujourd'hui plus utilisé et ne dispose d'aucun mécanisme de contrôle d'accès.

Le mode protégé apporté par l'Intel 80286 et étendu avec le 80386 [62, Vol. 1 3.1.1] est beaucoup plus intéressant d'un point de vue de l'isolation. Il apporte les mécanismes de segmentation et de pagination qui permettent la virtualisation de l'espace d'adressage du cœur, avec deux phases de traduction effectuées par la *Memory Management Unit* (MMU). Le logiciel manipule des adresses logiques, traduites une première fois en adresses linéaires par la segmentation, puis une deuxième fois en adresses physiques avec les structures arborescentes de la pagination. La segmentation apporte surtout une gestion des niveaux de privilèges appelés anneaux (*rings*) qui influent entre autres sur les instructions et registres accessibles par le logiciel. Le procédé de traduction d'adresse permet aussi d'associer à un segment ou une page mémoire [6] un niveau de privilège requis. Grâce à la segmentation et la pagination, il est possible d'exécuter des logiciels plus ou moins privilégiés mettant en œuvre, pour le cœur, la notion de contextes de mémoire. Un mode de compatibilité, appelé *virtual-8086 mode*, qui permet d'exécuter du code 16 bits dans un environnement protégé est aussi disponible. Notre processeur supportant un mode de fonctionnement 64 bits, il va dès que possible s'y configurer.

Le dernier mode de fonctionnement ajouté par Intel est le mode IA-32e qui contient deux sous mode de fonctionnement [62, Vol. 1 3.1.1], le mode de compatibilité 32 ou 16 bits et le mode 64 bits. Le Mode 64 bits fait disparaître la traduction d'adresses apportée par la segmentation, jugée obsolète, au profit de la pagination qui est bien plus précise. Cependant, ce nouveau mode n'apporte pas de nouveaux mécanismes d'isolation.

Avec l'avènement massif de la virtualisation des architectures, Intel a proposé une extension d'Intel 64 permettant d'accélérer le procédé de virtualisation des cœurs du processeur, à l'aide de nouveaux modes et sous modes opératoires ainsi que d'un jeu d'instruction dédié. Avec cette extension matérielle appelée Intel VT-x [62, Vol. 3 23.1-3], un cœur peut entrer en mode *VMX operation* où un logiciel invité va s'exécuter en *VMX non-root operation* sous la tutelle d'un logiciel privilégié, l'hyperviseur, qui s'exécute en *VMX root operation*. L'hyperviseur est en mesure de reprendre la main sur l'exécution du logiciel invité pour effectuer des contrôles sur des actions privilégiées, comme l'utilisation de certains registres. Un niveau supplémentaire de virtualisation de la mémoire contrôlé par l'hyperviseur est également ajouté. Ce mécanisme de traduction, appelé Extended Page Tables (EPT), très similaire à la pagination du mode IA-32e, simplifie la mise en place de l'espace d'adressage des machines virtuelles. Les signaux entrants et sortants des cœurs sont de plus contrôlés par l'hyperviseur, grâce à la virtualisation du *local APIC*.

Un mode d'exécution transverse, appelé mode de *management* (SMM), est aussi intégré depuis le processeur Intel386 SL [62, Vol. 1 3.1]. Son but est de proposer un environnement d'exécution privilégié pour effectuer des opérations de gestion spécifiques à une plateforme donnée (gestion de l'énergie, configuration du matériel, etc.). Ce mode a été conçu pour accueillir des parties de code du *firmware* de la machine. Il propose un espace d'adressage séparé du reste des modes [7], ainsi que d'un accès total aux ressources de la machine. Le seul évènement pouvant faire basculer un cœur dans ce mode est la réception d'un signal de type *System Management Interrupt* (SMI).

5. le processeur Intel 8086 étant la première implémentation de l'architecture x86
6. unité de découpe de l'espace d'adressage : 4 Kilos octets, 2 Mégas octets ou 1 Gigaoctet
7. certains accès à des segments étant traduits vers des parties réservées de la RAM appelée SMRAM

2.3.2 Interconnexion du processeur

Les mécanismes de contrôle d'accès précédemment cités fonctionnent grâce à la collaboration des cœurs avec le contrôleur mémoire. Les adresses physiques manipulées par les cœurs ne représentent pas forcément une adresse d'un bloc mémoire de la DRAM. Les accès mémoires sont traduits par le contrôleur mémoire et acheminés vers le périphérique ou le bloc mémoire adéquat. Par exemple, les accès au segment [`0xa0000` ; `0xbffff`] sont dédiés à la *Legacy Video Area*. Or, si le cœur est en mode SMM, l'accès est acheminé vers un segment de la DRAM dédié au mode SMM, la SMRAM. Cette vérification effectuée au plus près des cœurs permet notamment de se protéger des attaques du mode SMM par *cache poisoning* [63]. Une région de la DRAM est de plus réservée au coprocesseur *management engine* pour des raisons de sécurité. Les accès venant des cœurs sont donc aussi filtrés par le *System Agent* [64, Vol 2. 2.3, 2.5.2].

Le contrôleur mémoire est au cœur de l'architecture pour les accès mémoire réalisés par le processeur et il prend également en charge le contrôle des accès des périphériques à l'espace mémoire physique. Comme nous l'avons vu dans la partie 2.2, une partie des lectures/écritures mémoires sont traduites vers les périphériques locaux ou distants (bus DMI). C'est aussi le cas dans le sens inverse. Les périphériques peuvent accéder à une certaine partie de la DRAM, mais se voient filtrés pour la SMRAM, le *management engine* et les segments protégés du DMA, etc. [64, Vol. 2 2.5, 2.5.2, 2.15, 1.16].

Chaque périphérique peut être utilisé dans des contextes différents, ce qui pose des problèmes de sécurité. Aussi, Intel a introduit un autre type de composant appelé IOMMU spécifié par les extensions Intel VT-d [65]. Il permet d'effectuer des contrôles d'accès plus précis [8], tenant compte des contextes, sur les lectures/écritures et les signaux émis par les périphériques. Ces contrôles sont respectivement appelés *DMA remapping* et *IRQ remapping* où dans les deux cas, les adresses manipulées sont traduites et filtrées. Sur notre architecture, nous avons une IOMMU pour les périphériques locaux et une autre pour les périphériques distants [64, Vol 2. 4.4, 4.5]. Plusieurs périphériques sont contrôlés par une même IOMMU. Les IOMMU ont été conçues pour bloquer les attaques provenant des périphériques utilisant les lectures/écritures dans des segments accessibles de la mémoire principale [66].

2.3.3 Interconnexion des périphériques

Le PCH est en charge de l'interconnexion des périphériques distants avec le processeur au travers du bus DMI [60, 5.2, 9.4]. Comme nous l'avons vu, les contrôles d'accès en provenance du bus DMI sont faits en amont via l'IOMMU. En ce qui concerne la communication pair-à-pair entre périphériques internes au PCH ainsi qu'avec les périphériques PCI Express, les accès sont tous redirigés vers le PCH et traités par le *PCI-to-PCI bridge*. Ce composant va interdire les accès mémoire des périphériques aux zones *DMI decoded*, c'est-à-dire, les zones où sont mappés les périphériques [60, 5.2, 9.4]. Sur notre architecture, il n'est donc pas possible d'effectuer des requêtes mémoires pair-à-pair entre périphériques.

2.3.4 Démarrage et chaîne de confiance

Comme expliqué dans les sections précédentes, le processeur et ses cœurs démarrent dans un mode dégradé et vont progressivement mettre en place un environnement d'exécution plus

8. en comparaison avec les larges segments énumérés plus haut, avec la même précision qu'une MMU

performant, ainsi que les contrôles d'accès adéquats pour renforcer la sécurité du système. Cette configuration est effectuée par des logiciels. Ce sont les *firmwares*, *bootloaders* et couches basses des hyperviseurs de machines virtuelles et noyaux qui vont s'en charger. Il est donc important d'assurer leur intégrité. L'application du principe de chaîne de confiance, où chaque composant logiciel chargé au fur et à mesure du démarrage va vérifier l'intégrité du prochain, permet d'obtenir la confiance souhaitée [63, 3.1]. Ces vérifications sont préférentiellement assistées par des composants matériels dédiés comme les *Trusted Platform Modules* (TPM) [67] dont notre architecture dispose [60, 5.26].

Ces mécanismes assurent l'intégrité du système au démarrage, mais ne le protègent pas contre l'exploitation de vulnérabilités logicielles et matérielles à l'exécution. Ces exploitations peuvent par exemple viser à désactiver certains mécanismes matériels de contrôle d'accès comme les IOMMU. C'est pour faire face à ce type de menaces qu'Intel a proposé les extensions matérielles *Safer Mode Extensions* (SMX) [62, Vol 2C. 5] spécifiées par *Trusted Execution Technology* (TXT). Avec ces extensions, après un démarrage vérifié, il est possible de verrouiller certaines fonctionnalités et registres matériels pour interdire leur reconfiguration, même dans le cadre d'une attaque, en entrant dans un environnement *Measured Launched Environment* (MLE) [68, 1 et 2].

2.4 Rappels techniques

Cette section entre dans les détails des composants manipulés pour la réalisation de l'architecture de confiance. Le but n'est pas d'être exhaustif, mais de donner un complément sur les éléments nécessaires pour la lecture de ce manuscrit de thèse. Nous commençons par donner des éléments sur la configuration des cœurs de processeur. Nous continuons par décrire succinctement le fonctionnement et la configuration de l'extension matérielle Intel VT-x. Ensuite, nous mettons en avant les éléments importants du fonctionnement du bus d'interconnexion PCI Express ainsi que du composant IOMMU.

2.4.1 Détails sur l'architecture Intel 64

Dans cette sous-section, nous donnons quelques détails sur la configuration de fonctionnalités des cœurs de processeur, notamment sur la configuration des accès à la mémoire et sur la gestion de la configuration des politiques de gestion des caches.

2.4.1.1 Virtualisation de la mémoire des applications avec la MMU

La MMU dispose de plusieurs modes de fonctionnement pour virtualiser la mémoire manipulée par le logiciel qui s'exécute sur un cœur. Le mode le plus élaboré, actif lorsque le cœur s'exécute en mode IA-32e 64 bits, est le mode *IA-32e paging*.

La mémoire virtuelle utilisée par un cœur de processeur est gérée par le composant MMU. Avec ce composant il est possible de découper la mémoire virtuelle en sections de 4 Kilos, 2 Mégas et 1 Gigaoctet, appelées pages mémoire. Chaque section adressée virtuellement pointe vers la section physique qui est associée par la configuration courante de la MMU. En effet, chaque logiciel s'exécutant sur le processeur peut avoir une configuration et donc un langage de traduction de l'espace mémoire différent.

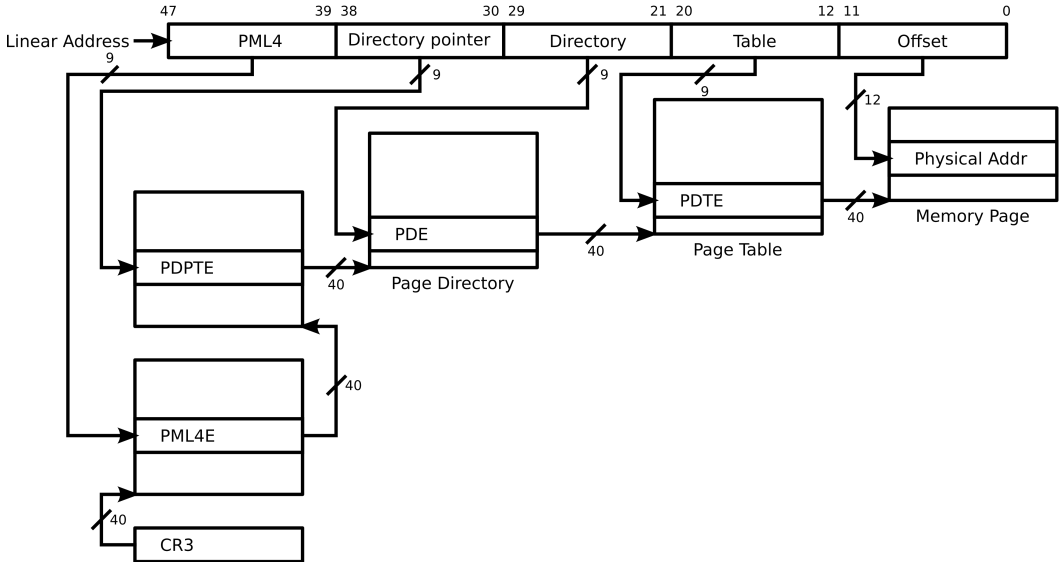

FIGURE 2.3 – Pagination en mode *IA-32e paging* en pages de 4 Kilo octets

En ce qui concerne la configuration de la mémoire virtuelle, le langage de traduction des adresses virtuelles est décrit dans des tables de pages hiérarchiques placées en mémoire. La table racine est pointée par le registre de contrôle `cr3`. Le procédé de traduction est le suivant. Comme le montre la figure 2.3, l'adresse virtuelle de 48 bits manipulée par le processeur est découpée en 4 sections de 9 bits et une de 12 bits, ordonnées des bits de poids forts aux bits de poids faibles. Le début du processus de traduction est donné par l'adresse stockée dans le registre `cr3`, pointant vers la première table de pages (PML4). Cette table, contenant 512 entrées de 64 bits, est indexée avec la première section de 9 bits, nous donnant l'adresse de la prochaine table de traduction fille (PDPT). Le procédé est répété pour les 3 autres sections de 9 bits de l'adresse virtuelle, jusqu'à la dernière table (PT), dont l'entrée indexée ne donne pas d'adresse de table, mais de la page mémoire traduite. Les 12 derniers bits de l'adresse virtuelle sont ajoutés à l'adresse de la page physique pour obtenir l'adresse physique complète. Le procédé décrit précédemment adresse des tables de 4 Kilos octets. Notons qu'il peut s'arrêter plus tôt, pour adresser des pages de plus grande taille, de 2 Mégas octets et 1 Gigaoctet. Dans ces cas-là, le bit *page size* des entrées de tables hiérarchiquement plus élevées est positionné à 1 en amont (respectivement table PD ou PDPT). Dans ces deux cas, les sections de 9 bits non utilisées pour la traduction s'ajoutent aux 12 bits de l'offset.

2.4.1.2 Configuration des caches

Les accès mémoire d'un cœur, en instructions ou données, sont accélérés par plusieurs niveaux de cache : L1, L2 et L3, allant du plus rapide au plus lent et du plus petit au plus gros en capacité. Cependant, ces caches sont configurables avec plusieurs politiques appelées *memory types*, affectant l'ordre dans lequel les accès mémoire sont exécutés, mais aussi le temps pendant lequel la donnée lue ou écrite reste en cache.

Il est possible de configurer la politique de cache par section mémoire. Cette configuration s'effectue par deux mécanismes : les *Memory Type Range Registers* (MTRR) et *Page Attribute Table* (PAT). Les MTRRs sont accessibles via un type de registres spéciaux du processeur,

les *Model Specific Registers* (MSR). Via les MTRRs, il est possible de définir la politique de caches associée à un segment mémoire [62, Vol. 3 11.11]. La pagination offre également un moyen de surcharger la configuration des MTRRs par page mémoire. Une combinaison de bits des entrées de tables de pages indexe la table PAT, définie via des MSRs, pour définir la politique associée à une page mémoire [62, Vol. 3 11.12].

2.4.2 Extension matérielle d'assistance à la virtualisation

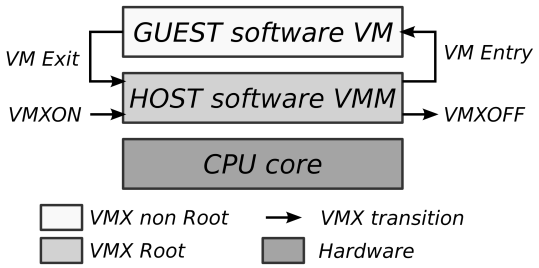

FIGURE 2.4 – VMX operation

VT-x [62], pour Virtual Technology, apporte le support matériel pour la virtualisation de l'architecture x86. Pour ce faire, trois éléments sont ajoutés à l'architecture : 1) un jeu d'instructions appelé VMX incluant 13 instructions, 2) un niveau de virtualisation de la mémoire et 3) un mode d'exécution avec deux sous-modes respectivement appelés *VMX operation*, *VMX root operation* et *VMX non-root operation*.

Des machines virtuelles (*guests*) sont exécutées sur des cœurs virtuels configurés avec des structures internes du processeur appelées *Virtual Machine Control Structures* (VMCS). Les machines virtuelles sont exécutées en mode *VMX non-root operation*, ce qui signifie qu'elles sont sous le contrôle de l'entité logicielle appelée le *Virtual Machine Monitor* (VMM, *host* ou hyperviseur) qui s'exécute en mode *VMX root operation*. L'hyperviseur entre en mode *VMX operation* avec l'instruction `vmxon` et en sort avec l'instruction `vmxoff`. Pour créer et contrôler une machine virtuelle, l'hyperviseur charge une VMCS, avec l'instruction VMX `vmptrld`. Puis, il prépare l'état du *guest* en accédant à la VMCS chargée avec les instructions `vmread` et `vmwrite` pour renseigner à la fois son propre état, celui de la machine virtuelle ainsi que des contrôles d'exécution. Enfin la machine virtuelle est démarrée avec l'instruction `vmlaunch`. L'hyperviseur prend le contrôle sur la machine virtuelle au travers d'interruptions de machines virtuelles appelées *VM Exits*. Après avoir traité correctement le *VM Exit*, dans le cas nominal, l'hyperviseur redonne la main à la machine virtuelle grâce à l'instruction VMX `vmresume` (figure 2.4).

L'hyperviseur protège son propre espace mémoire et isole les régions mémoire de la machine virtuelle en utilisant la couche de virtualisation supplémentaire de la MMU appelée *Extended Page Tables* (EPT).

2.4.2.1 Les structures de contrôle de machines virtuelles

Une VMCS est une structure interne au microprocesseur, permettant de configurer un cœur virtuel. Il s'agit d'un ensemble de registres qui regroupe toutes les informations permettant de sauvegarder ou restaurer l'état de la machine virtuelle et de l'hyperviseur (lors d'un *VM exit* ou

VM entry), de configurer le processeur virtuel (fonctionnalités disponibles) et de spécifier les évènements qui engendreront une transition vers l'hyperviseur. Elle est structurée en plusieurs parties : *Guest-state area* ; *Host-state area* ; *Control fields* ; *VM-exit information fields*.

Un seul processeur virtuel peut s'exécuter à la fois sur un cœur logique. Le principal intérêt de la virtualisation étant de pouvoir exécuter de nombreuses machines virtuelles, la VMCS peut être recopiée en mémoire pour pouvoir ordonnancer plusieurs machines virtuelles. Ces copies sont appelées *VMCS regions* et occupent 4 Kilos octets.

Les deux zones des VMCS réservées aux sauvegardes et aux restaurations des états des machines virtuelles et de l'hyperviseur sont, respectivement, *Guest-state area* et *Host-state area*. Ces zones sont à priori mises à jour dans la *VMCS region* au moment des *VMX transitions*. Notons que les zones mises à jour sont celles de la VMCS courante, contenue dans le processeur. Il est donc important de se poser la question de la cohérence entre l'état de la VMCS courante et l'état de la *VMCS region*, qui se situe dans la mémoire centrale.

La configuration d'un hyperviseur est reflétée seulement partiellement en mémoire, car certains registres, registres de contrôle, debug et *MSRs* sont « partagés » entre le *guest* et le *host* et ne sont pas stockés dans la VMCS region.

2.4.2.2 Virtualisation de la MMU avec EPT

EPT ajoute une couche de virtualisation à la MMU. La configuration la mémoire virtuelle d'une machine virtuelle se fait via le champ de VMCS *EPT pointer*. Ce pointeur indique l'adresse mémoire de la première table de pages de traduction. Le format des tables de pages d'EPT est quasiment identique à celui de la pagination IA-32e classique. Les adresses virtuelles manipulées par une application s'exécutant dans une machine virtuelle sont traduites deux fois, une fois par le fonctionnement classique de la MMU et une autre fois par EPT. Il est important de noter que les accès aux tables de pages IA-32e de cette application seront aussi traduits par EPT.

EPT utilise aussi le cache de traduction d'adresse de la MMU, les *Translation Lookaside Buffers*. Ces caches ont donc dû être remaniés pour éviter les confusions de traduction entre les adresses de machines virtuelles et celles manipulées par EPT. De plus, pour ne pas confondre les adresses traduites entre différentes machines virtuelles, la MMU propose un système d'identifiants appelés *Virtual Process IDentifiers* (VPID), qui permettent de tatouer les addresses traduites par les différentes machines virtuelles et ainsi éviter les confusions.

2.4.3 Interconnexion des périphériques

Plusieurs spécifications de bus comme *Industry Standard Architecture* (ISA) ou *Peripheral Component Interconnect* (PCI) ont été implémentés pour standardiser les communications entre le processeur et leurs périphériques. Aujourd'hui, le bus PCI Express est utilisé dans la plupart des ordinateurs personnels et des serveurs.

2.4.3.1 Bus PCI Express

Les composants PCI Express (*devices*) sont connectés à l'aide de ports et de liens (*links*). Le bus PCI Express interconnecte des composants de manière hiérarchisée. Un port connecté à un composant fils est appelé un port descendant (*downstream port*), tandis qu'un port connecté à

un composant père est appelé un port montant (*upstream port*). Il existe trois types de composants principaux. Le *root complex*, racine de la hiérarchie du bus, est connecté au processeur grâce au pont hôte (*host bridge*) et aux composants fils de premier niveau de hiérarchie. Ces composants peuvent être des périphériques (*endpoints*) ou des ponts *bridges* (figure 2.5). Un pont connecte deux domaines logiques de bus différents avec un port montant et un port descendant. Enfin, un *switch* est une collection de ponts. Notons que le *root complex* peut héberger des périphériques et des ponts. Ces ponts possèdent seulement des ports descendants, appelés ports racines (*root ports*).

2.4.3.2 Communications

Chaque périphérique PCI Express est identifié par un numéro de bus logique PCI, un numéro de composant et un numéro de fonction : `bus:dev.fun`. Cet identifiant est utilisé pour acheminer les messages PCI Express entre les composants.

Il existe un type de message PCI Express pour chaque type de transaction, comme les lectures / écritures mémoire et de configuration. Une transaction peut être soit postée (*posted*) et une réponse, appelée *completion*, est alors nécessaire, soit non postée (*non-posted*) et aucune réponse n'est attendue. Par exemple, une écriture mémoire est postée, car l'émetteur n'attend pas de réponse. Par opposition, une lecture mémoire est non postée et mène nécessairement à l'attente d'une réponse du composant interrogé.

Le destinataire d'un message est identifié soit directement par son identifiant soit par une adresse mémoire. Cette adresse correspond à un emplacement dans la mémoire principale ou au registre d'un autre composant mappé en mémoire. En ce qui concerne un message de lecture

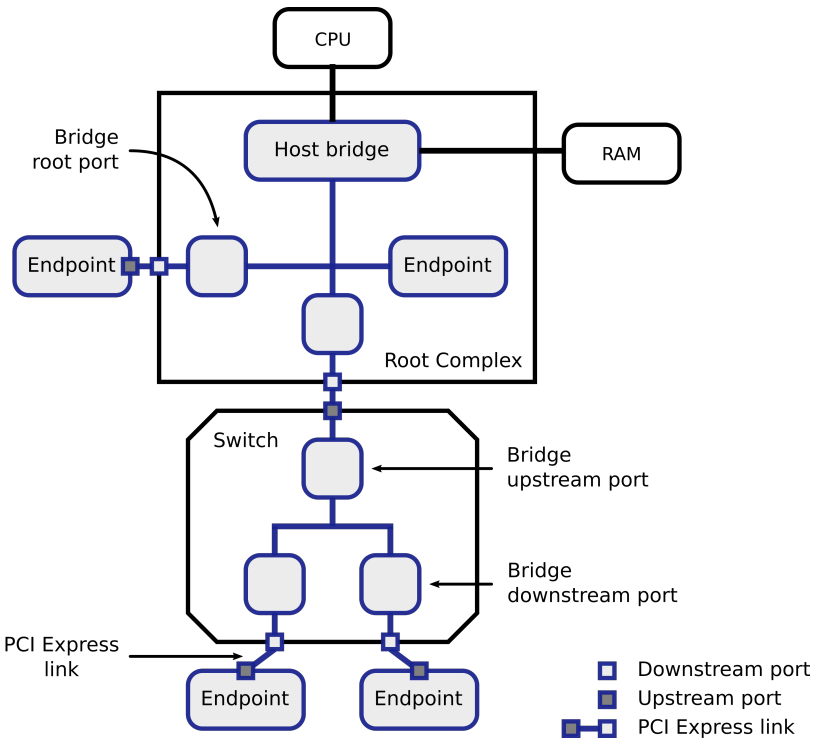

FIGURE 2.5 – Vue logique d'une architecture compatible avec le bus PCI Express

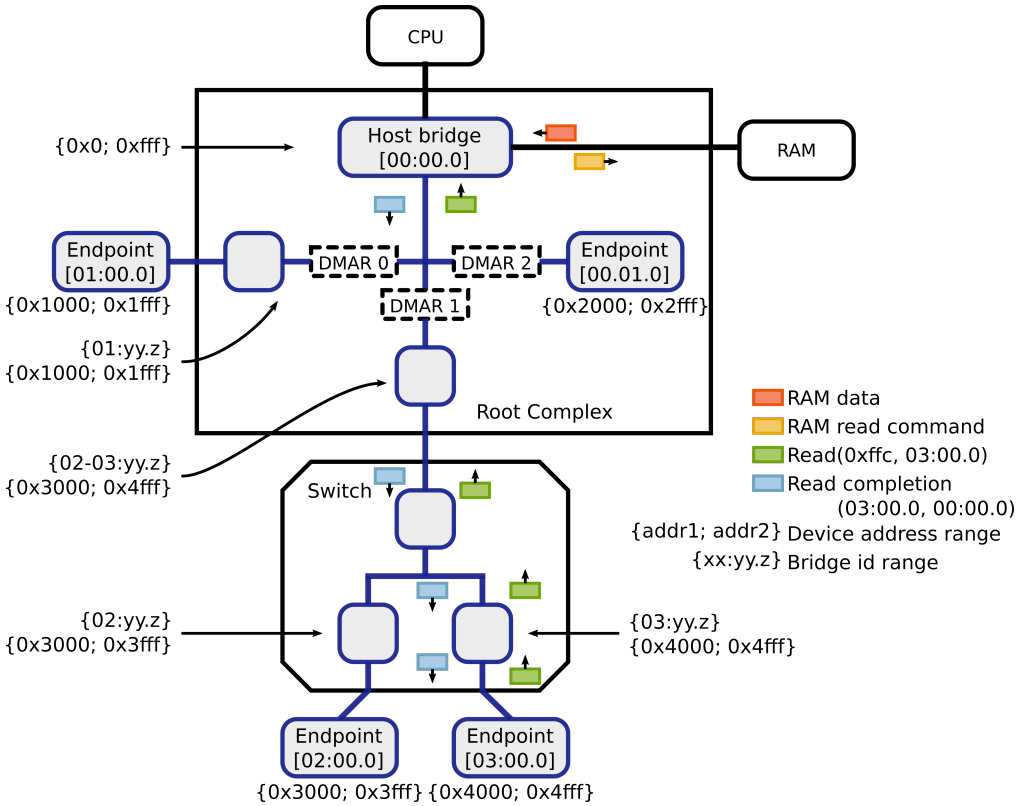

FIGURE 2.6 – Routage d'un message de lecture mémoire

mémoire, il contient une adresse de destination et un identifiant d'émetteur appelé *requester id*. Le composant destinataire de la réponse de lecture mémoire correspondante sera celui pointé par le *requester id* de la requête. Les messages PCI Express sont par conséquent acheminés par adresse ou identifiants PCI. Pour acheminer les messages correctement, les ponts sont configurés par l'hôte (le processeur) pour qu'ils connaissent les identifiants et les intervalles mémoires dont ils sont responsables (figure 2.6).

Au démarrage, les composants ne sont pas configurés (ils ne connaissent pas leur identifiant PCI). En effet, un constructeur ne peut pas savoir sur quel port sera connecté le composant qu'il produit. Cependant, quand le *firmware* de la machine établit la liste de tous les composants connectés, en utilisant par exemple l'instruction `mov` depuis l'espace de configuration PCI Express mappé en mémoire, chaque accès mémoire est traité par le *host bridge*. Le *host bridge* traduit ensuite cet accès en un message de configuration PCI Express qui est acheminé vers le composant correspondant. En particulier, cette requête contient l'identifiant du composant contacté. Donc, si ce composant est présent dans le système, il reçoit la requête et par conséquent connaît son propre identifiant. Cette étape est très importante pour permettre aux périphériques et composants de communiquer entre eux.

Les composants PCI Express sont capables, au travers de requête de lecture mémoire d'accéder aux mémoires d'autres composants, mais aussi à la mémoire principale. On parle dans le dernier cas d'accès DMA. La figure 2.6 présente les différentes étapes d'un accès DMA d'un périphérique, ici d'identifiant `[03:00.0]`. Le message de lecture mémoire est à destination d'une zone mémoire gérée par le *host bridge*, qui va le traduire en commande de

lecture DDR, récupérer la donnée et la retransmettre avec une réponse de lecture à destination du périphérique source de la transaction. Cette fonctionnalité soulève un souci majeur du point de vue de la sécurité si le comportement d'un composant peut être contrôlé par un attaquant. Pour faire face à cette menace, Intel a développé un composant matériel pour virtualiser l'espace mémoire des périphériques et filtrer les messages de lecture et d'écriture mémoire PCI Express. Ce composant, appelé IOMMU, est présenté dans la section suivante.

2.4.3.3 Virtualisation de l'espace mémoire des périphériques

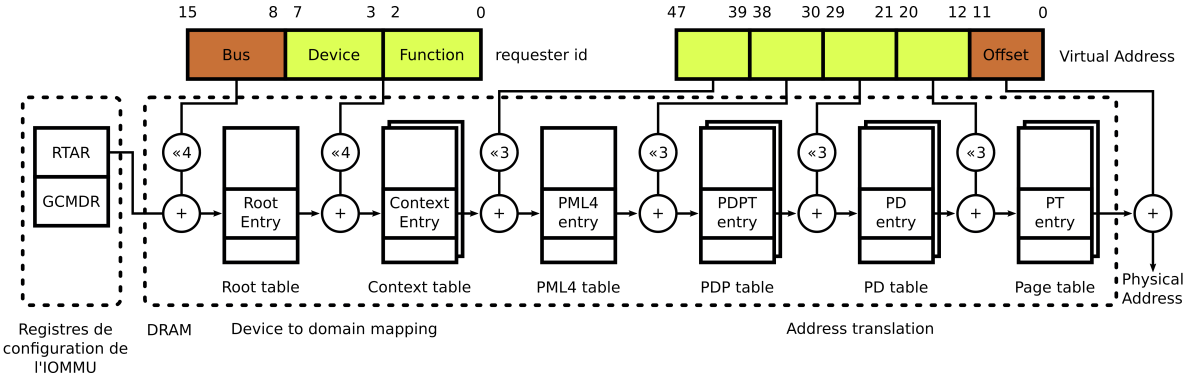

FIGURE 2.7 – DMA Remapping avec des adresses virtuelles de 48 bits en pages de 4 Kilos octets

Les IOMMUs sont notamment conçues pour virtualiser l'espace mémoire via un service appelé *DMA Remapping* (DMAR). Une même architecture peut contenir plusieurs IOMMUs, chacune étant dédiée à un sous ensemble de composants installés (figure 2.6). Les IOMMUs traduisent et filtrent les requêtes en fonction du domaine de protection assigné au composant émetteur. Un domaine de protection est tout simplement un langage de traduction d'adresses virtuelles en adresses physiques. Le procédé est divisé en deux phases. La première sert à identifier le domaine de protection associé au composant émetteur. Cette phase, appelée *device to domain mapping*, est conceptuellement similaire à une traduction d'adresse, mais associe plutôt des identifiants PCI aux domaines de traduction d'adresse. Dans la seconde phase, appelée traduction d'adresse, les adresses utilisées par les accès mémoire des composants sont traduites par les unités de DMAR, avant de traverser le *host bridge* (figure 2.6). Cette traduction est très proche de celle effectuée par les MMUs des cœurs. Des contrôles d'accès sont applicables dans les deux phases de traduction.

Chaque unité de DMAR doit être configurée séparément. Le comportement du DMAR est défini par une page de registres de configuration PCI Express et une structure mémoire arborescente (figure 2.7) configurée au démarrage en RAM. La page de configuration contient le registre de contrôle principal d'une IOMMU appelé *Global Command Register* (GCMDR), qui donne le moyen d'activer le mécanisme de traduction grâce au bit *Translation Enable* (TE) [65, 10]. La page de configuration contient un registre pointant vers la racine de la structure de configuration (c'est-à-dire vers la première *root entry* de la *root table* de la figure 2.7), appelé le *Root Table Address Register* (RTAR). La localisation en mémoire des pages de configuration du DMAR est donnée grâce à des registres de configuration du contrôleur mémoire. L'identifiant d'un message PCI Express est utilisé pour indexer les deux premières

tables de la structure arborescente de traduction (*root table* et *contexte table*) dans la première phase. L'adresse résolue est ensuite utilisée pour localiser les tables de la seconde phase. Ces dernières tables sont indexées avec l'adresse de destination du message PCI Express. Le résultat de la traduction est l'adresse physique correspondant à l'adresse virtuelle accédée. Pendant le parcours des tables pour la traduction, un bit dédié de chaque champ peut indiquer si l'accès est autorisé ou interdit. Enfin, notons que la seconde phase de traduction peut être désactivée (*mode pass through*) grâce aux bits indiquant le type de traduction des entrées de la *context table*.

Les unités de DMAR modernes peuvent aussi être virtualisées. Ceci est mis en place avec l'ajout d'une seconde phase de traduction intervenant après celle mise en place par le noyau d'un système d'exploitation. Ce second niveau est destiné à être utilisé par un hyperviseur, pour appliquer la vision de la mémoire des machines virtuelles (avec EPT) aux composants PCI Express associés.

2.5 Conclusion

Dans ce chapitre nous avons étudié les architectures matérielles modernes et notamment la plus répandue des processeurs Intel, Intel 64. Dans un second temps nous avons présenté les détails techniques qui sont nécessaires pour comprendre les enjeux de cette thèse. Le chapitre suivant met en avant une architecture de sécurité permettant de surveiller l'intégrité de logiciel s'exécutant sur ce type de machines complexes et vulnérables aux attaques présentées dans le chapitre 1.

Chapitre 3

Architecture de sécurité

Sommaire

3.1 Introduction

La tendance actuelle pour sécuriser un système informatique et en particulier ses couches basses, consiste à ajouter des protections dans des niveaux de privilèges élevés du système, basées sur des mécanismes logiciels ou matériels (par exemple ARM TrustZone ou Intel SGX). Ces approches sont intéressantes, mais supposent que le processeur et les couches basses sont des composants de confiance. Or, ces couches basses ou le processeur lui-même peuvent se retrouver en état d'erreur suite à des actions malveillantes. Pour faire face à ce problème,

nous présentons la conception et la mise en œuvre d'une architecture de confiance assistée par le matériel, appelée ici **architecture de sécurité**, dans laquelle il est possible d'exécuter des tests d'intégrité de manière sécurisée, de n'importe quel composant s'exécutant sur cette architecture. Cette architecture est un mélange hybride de composants logiciels et matériels. Elle est décrite dans ce chapitre qui est organisé en deux parties. Dans la première partie, le modèle de menaces et les hypothèses de notre approche sont présentés ainsi que la vue d'ensemble de notre architecture de sécurité et ses deux composants principaux : l'hyperviseur de sécurité et le périphérique de confiance. Dans la seconde partie, nous présentons en détail le protocole de tests d'intégrité qu'il est possible de réaliser dans cette architecture, ainsi que les détails sur la conception des tests eux-mêmes.

3.2 Modèle de menaces et hypothèses

Dans la suite, nous définissons un attaquant comme étant une personne malveillante parmi les acteurs du cloud définis dans la section 1.4.3. Nous faisons l'hypothèse que les attaquants n'ont accès physiquement à aucun des composants de l'architecture (*hypothèse 1*). Notons que si un attaquant obtient un accès physique à une machine, alors il est capable d'effectuer des attaques encore plus graves sur le système que simplement corrompre l'intégrité du logiciel. Nous supposons donc que les attaques sont menées par l'exécution de logiciel malveillant sur le processeur via les surfaces d'attaques présentées dans le modèle de menace du chapitre 1 ou qu'une logique maligne est déjà présente (*hypothèse 2*). De plus, même si l'attaquant effectue seulement des attaques logicielles, nous considérons qu'il est capable de cibler d'autres logiciels, mais aussi des composants matériels (*hypothèse 3*). En particulier, un attaquant est capable de reconfigurer le matériel pour l'utiliser à sa convenance, si le matériel offre directement la capacité de se reconfigurer ou qu'il inclut une vulnérabilité qui le permet.

Suivant ces hypothèses, le modèle de menaces est le suivant. Un logiciel malveillant peut s'exécuter avec un niveau de privilège supérieur, égal ou inférieur à celui qu'il essaie de corrompre. Pour mettre en œuvre son attaque, il a la possibilité de corrompre n'importe quel autre logiciel ayant aussi un niveau de privilège pouvant aussi être plus élevé, égal ou inférieur à son propre niveau. De la même manière, il peut rebondir sur les composants matériels. Par exemple, une application pourrait exploiter des vulnérabilités dans l'interface de certains appels système pour augmenter ses privilèges. Un logiciel malveillant injecté dans l'espace noyau dispose des accès à tout l'espace mémoire physique, et donc jouit de privilèges d'accès élevés pour modifier tous les logiciels s'exécutant sur la machine. On considère également le cas de logiciels pris sur étagère ou *Component Off The Shelf* (COTS) et des composants matériels, qui peuvent exposer des vulnérabilités en raison d'un mauvais développement ou d'une mauvaise configuration.

3.3 Attestation de code et racine de confiance dynamique

Détecter la compromission de l'intégrité d'un logiciel est un problème qui s'apparente à d'autres problèmes plus généraux, comme l'attestation à distance de logiciel, qui permet d'assurer la non-modification du code chargé en mémoire dans une machine distance. Il s'apparente aussi à la mise en place de racine de confiance dynamique (*dynamic root of trust*), qui permet d'assurer à l'utilisateur la non-modification de logiciels à l'exécution, mais aussi l'intégrité de

l'environnement d'exécution.

Les travaux menés dans Pionner [69] se situent notamment dans ce contexte. Pioneer propose un protocole générique et une implémentation sur processeur d'architecture Intel 64, sans extension d'assistance à la virtualisation, dont le but est de mettre en place, à l'exécution, une racine de confiance dynamique (*dynamic root of trust*). Il met en œuvre deux machines. Une première machine de confiance, appelée le *dispatcher* ou vérifieur, possédant la copie intégrale du code à exécuter sur une machine distante non approuvée (*untrusted platform*), ou prouveur. Le cœur du protocole de vérification de Pioneer est la fonction de vérification qui est exécutée sur la plateforme du prouveur. Elle a pour objectif d'exécuter un test d'intégrité de l'exécutable. Pour cela, elle doit tout d'abord s'assurer qu'elle exécute ce test dans un environnement non compromis, en calculant une somme de contrôle de son propre code et en envoyant cette somme au vérifieur. Le vérifieur compare cette somme à une valeur attendue mais aussi vérifie que cette somme a été calculée dans un laps de temps connu. Si ce délai n'est pas respecté, l'environnement d'exécution es compromis.

SWATT [70] est un autre outil qui est lui dédié uniquement à l'attestation de logiciel à distance, dans les dispositifs embarqués. En utilisant une partie des techniques de Pioneer, notamment le calcul d'une somme de contrôle mémoire, par une procédure de vérification embarquée dans le dispositif cible, les auteurs garantissent l'intégrité du contenu attendu de la mémoire. Encore une fois, dans ces travaux la construction de la fonction de vérification utilise un parcours pseudo aléatoire de l'espace mémoire à vérifier, pour complexifier la tâche d'un adversaire. Enfin, les attaques sont détectées grâce à la valeur du *checksum* calculé et du temps d'exécution attendu non respecté.

VIPER [55] est une fonction de vérification de *firmwares* de périphériques PCI Express. La conception du protocole et les propriétés apportées par cette fonction de vérification sont très similaires à celles de Pioneer.

SMART [71] est un autre outil permettant de mettre en place une racine de confiance dynamique comme le propose Pioneer. Contrairement à ce dernier, les auteurs de SMART se focalisent sur les microcontrôleurs simples (ne disposant pas de MMU) comme les MSP430 de Texas Instrument ou les AVR d'AMTEL. Pour pouvoir assurer les propriétés de sécurité attendues, contrairement aux travaux précédents, les auteurs de SMART ajoutent du matériel à ces architectures prêtes à l'utilisation. SMART propose un protocole de type défi/réponse pour mettre en place la racine de confiance dynamique. Contrairement aux projets précédents, l'ajout de matériel spécifique, une ROM protégée contenant un code d'attestation, ainsi qu'une région de stockage sécurisée à l'aide d'une clé privée permettrent de prouver l'authenticité d'une réponse à un défi, sans avoir à mesurer le temps d'exécution de la fonction de vérification. Cette propriété de SMART est en effet très intéressant, car elle s'abstrait de la nécessité de proximité réseau du vérifieur vis-à-vis du prouveur et donne plus de flexibilité quant à la conception de la fonction de vérification.

L'architecture que nous proposons est proche du projet Pioneer dans le sens où elle vise le même type d'architecture et possède un objectif commun, établir une racine de confiance dynamique pour exécuter un logiciel protégé. Nous verrons dans ce chapitre que l'apparition des extensions virtualisation Intel VT-x ainsi que l'utilisation d'un périphérique de confiance comme racine de confiance de l'architecture renforce les propriétés de sécurité déjà apportées par Pioneer, tout en simplifiant leur mise en œuvre logicielle. Nous verrons que les challenges utilisés par notre architecture de sécurité sont dépendants du temps d'exécution, contrairement à SMART. En effet, il n'est pas possible de modifier l'architecture des contrôleurs mémoire

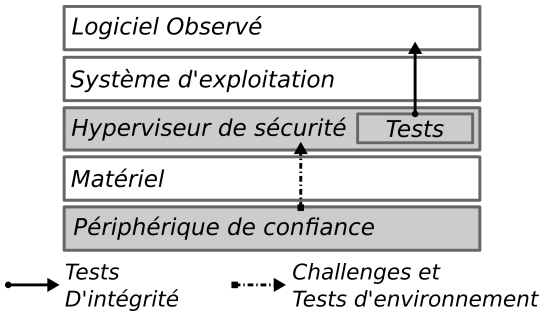

FIGURE 3.1 – Architecture de sécurité

des cœurs de processeurs Intel pour que seulement la procédure de vérification puisse accéder à la clé privée utilisée dans SMART.

3.4 Vue d'ensemble de l'approche proposée

Le but de notre architecture est de sécuriser la vérification de l'intégrité de logiciels s'exécutant sur un processeur. Nous considérons une approche boite noire dans laquelle le développeur apporte un composant logiciel à exécuter (par exemple un module noyau, un serveur WEB, etc.), accompagné d'un ensemble de fonctions de test d'intégrité dédiées à tester ce composant logiciel (figure 3.1). Ce composant logiciel est appelé **logiciel observé**. Les fonctions exécutées pour évaluer l'intégrité de ce composant sont appelées **tests d'intégrité**.

Pour avoir confiance dans les résultats de ces tests, il ne faut surtout pas reposer sur un ensemble de composants connus pour pouvoir être compromis. Aussi, nous avons opté pour intégrer dans l'architecture un composant matériel de confiance, que nous avons appelé **périphérique de confiance**, difficilement reprogrammable depuis le processeur central. Par contre, ce composant constitue seulement le point d'entrée dans cette architecture : il est incapable de mener lui-même les tests d'intégrité du logiciel observé. Aussi, son rôle sera simplement de tester l'intégrité d'un hyperviseur, appelé **hyperviseur de sécurité**, développé par nos soins. Ces tests sont maîtrisés et peuvent largement se satisfaire du niveau de granularité d'information dont dispose le composant matériel de confiance. Ensuite, à ce moment, l'hyperviseur peut alors mener directement les tests d'intégrité.

Le périphérique de confiance est physiquement embarqué dans l'hôte et connecté au bus PCI Express. Il est chargé d'évaluer régulièrement l'intégrité de l'hyperviseur de sécurité (figure 3.1). Effectivement, l'hyperviseur de sécurité peut aussi être corrompu, comme indiqué dans l'hypothèse 2, qui précise que tous les composants logiciels de la machine peuvent être potentiellement corrompus par un logiciel malveillant via l'exploitation d'une vulnérabilité logicielle ou par un rebond sur le matériel (hypothèse 3). Pour cette raison, il est nécessaire de tester l'intégrité de cet hyperviseur, même s'il contrôle le niveau de privilège le plus élevé du processeur. Le périphérique de confiance doit autant vérifier le comportement correct de l'hyperviseur de sécurité que la configuration des composants matériels et logiciels critiques de son environnement d'exécution. En effet, une attaque peut consister à installer un hyperviseur à côté du nôtre. Par conséquent, l'intégrité du code exécutable de notre hyperviseur reste valable alors qu'il est désactivé. Il est donc nécessaire de tester son exécution pour savoir s'il dispose toujours des droits hyperviseur en lui soumettant un code qui utilise ces privilèges. Si

ce code est exécuté dans un temps correct avec une valeur correcte, alors on suppose qu'il est toujours installé en mode hyperviseur et maître du matériel. Si l'exécution fournit un résultat incorrect, alors il est clairement désactivé. Si le code est exécuté dans un temps trop long, cela signifie que notre hyperviseur a perdu son statut d'hyperviseur et il y a de fortes chances qu'un logiciel malveillant émule certaines instructions. Plus précisément, le périphérique de confiance propose régulièrement des **challenges**, présentés dans la section 3.5.1 puis détaillés dans la section 3.7, que l'hyperviseur de sécurité doit exécuter, et vérifie le résultat de cette exécution. Le périphérique de confiance est aussi capable d'effectuer une multitude de tests qui ont pour but de 1) détecter des altérations dans le code ou les données de l'hyperviseur de sécurité et 2) détecter des changements dans la configuration de certains composants de l'environnement de l'hyperviseur de sécurité. Ces vérifications sont appelées **tests d'environnement** et sont présentées dans la section 3.5.2 puis détaillées dans la section 3.7. La détection de l'altération de l'intégrité du logiciel observé, de l'hyperviseur de sécurité ou de son environnement se manifeste par la levée d'une alerte. Les alertes sont traitées par une **machine de monitoring**, au travers d'un protocole de communication dédié, présenté dans le chapitre 4.

L'ensemble des tests d'intégrité, apportés par le développeur, est embarqué dans l'hyperviseur de sécurité, qui est en chargé de les exécuter. Si le logiciel observé est aussi un hyperviseur, notre hyperviseur de sécurité est capable d'activer un mode de virtualisation récursive pour gérer cette situation. Cet hyperviseur de sécurité contrôle le niveau de privilège le plus élevé du processeur. Il a été spécialement conçu et implémenté pour exécuter ces tests d'intégrité.

3.4.1 Infrastructure matérielle

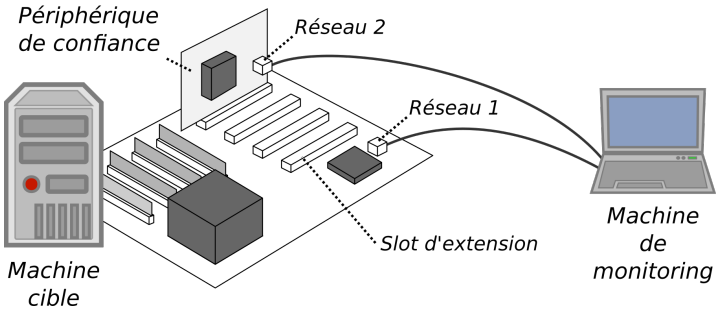

FIGURE 3.2 – Plateforme de l'architecture de sécurité

L'infrastructure matérielle de notre architecture de sécurité est schématisée dans la figure 3.2. Elle est composée de deux machines : la machine de *monitoring* et la machine cible sur laquelle est connecté le périphérique de confiance.

Nous avons fait le choix d'utiliser un système basé sur l'architecture x86 avec des périphériques interconnectés via le bus PCI Express parce que notre solution doit être facilement intégrable dans les architectures existantes. Dans tous les cas, les principes de notre approche sont toujours valides pour tout type d'architecture utilisant des composants interconnectés.

La machine cible exécute le logiciel observé et l'hyperviseur de sécurité. Elle dispose d'un processeur récent supportant l'assistance matérielle pour la virtualisation. Elle inclut aussi un premier lien réseau (figure 3.2, réseau 1), qui est sous le contrôle de l'hyperviseur de sécurité. Ce lien est considéré de confiance tant que l'hyperviseur de sécurité n'est pas corrompu. Les

résultats des tests d'intégrités du logiciel observé sont transmis à la machine de *monitoring* via ce lien.

Le périphérique de confiance est connecté au bus d'interconnexion des périphériques de la machine cible. Ce périphérique est aussi connecté à la machine de *monitoring* au travers de son interface réseau (figure 3.2, réseau 2). Le périphérique de confiance peut donc envoyer le résultat de ses analyses, à la machine de *monitoring* via ce lien de confiance.

Enfin, la machine de *monitoring* exécute un système d'exploitation pris sur étagère. Elle collecte les résultats des challenges, des tests d'environnement ainsi que des tests d'intégrité du logiciel observé. C'est d'ailleurs cette machine de *monitoring* distante qui génère les challenges et les tests d'environnement et les rend disponibles au périphérique de confiance via le réseau 2.

3.4.2 Périphérique de confiance

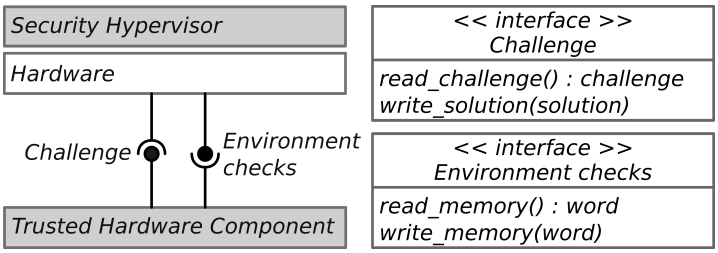

FIGURE 3.3 – Security architecture hardware interfaces

Le périphérique de confiance doit être conçu rigoureusement de manière à mettre en œuvre les challenges et les tests d'environnement tout en restant le plus indépendant du processeur possible. Ainsi, les interfaces du périphérique de confiance doivent être conçues pour prévenir toute exploitation malveillante provenant d'un hyperviseur de sécurité corrompu et par conséquent doivent rester simples (figure 3.3). L'interface proposée par le périphérique de confiance autorise deux interactions : 1) lire le code d'un challenge depuis une mémoire dédiée interne en lecture seule ; et 2) écrire la solution vers une mémoire dédiée interne en écriture seule. Ces interactions ne doivent avoir aucun impact sur le comportement interne du périphérique de confiance. Par exemple, une interaction logicielle malveillante depuis le processeur (*fuzzing*, *flooding*, etc.) ne doit pas empêcher le périphérique de confiance de rendre son service de sécurité.

3.4.3 Hyperviseur de sécurité

L'hyperviseur de sécurité est avant tout un environnement d'exécution de tests d'intégrité sur des logiciels s'exécutant sous sa responsabilité. L'objectif d'intégration de notre architecture vise à conserver la pile logicielle existante de la machine cible en y installant l'hyperviseur de sécurité. Si un système d'exploitation est présent, nous le virtualisons dans une machine virtuelle et s'il s'agit d'un autre hyperviseur, pris sur étagère par exemple, nous devons de la même manière le virtualiser ainsi que ses machines virtuelles dans un mode de virtualisation récursive.

Dans les deux cas, il est nécessaire d'impacter au minimum l'architecture logicielle originale, aussi bien au niveau de la complexité d'intégration, de la reconfiguration des logiciels existants

que des performances proposées aux logiciels. La stratégie que nous adoptons pour atteindre cet objectif consiste à virtualiser et émuler le moins de fonctionnalités possible du processeur, sans risquer de porter atteinte à l'intégrité de l'hyperviseur de sécurité (interruptions provenant des périphériques des machines virtuelles, exceptions non privilégiées, etc.). De plus, nous ne partageons pas les périphériques, ce qui évite l'émulation de leurs interfaces matérielles, pour contrôler les accès depuis la machine virtuelle. En effet, notre hyperviseur de sécurité doit disposer d'une carte réseau qui lui est dédiée, mais autorise l'utilisation directe de tous les autres périphériques par la machine virtuelle.

Enfin, l'hyperviseur de sécurité doit **impérativement** répondre aux interruptions du périphérique de sécurité afin d'exécuter les challenges qui lui sont soumis. Il doit pour cela configurer son contrôleur mémoire et sa mémoire virtuelle afin de rendre accessible l'interface matérielle du périphérique de confiance. Ensuite, il doit configurer son gestionnaire d'interruption ainsi que son vecteur d'interruption pour pouvoir être notifié d'un challenge par le périphérique de confiance.

Après la présentation les objectifs et du fonctionnement général de notre architecture, la prochaine section détaille le protocole de tests. Les tests eux-mêmes sont décrits en section 3.7.

3.5 Cycle de test d'intégrité

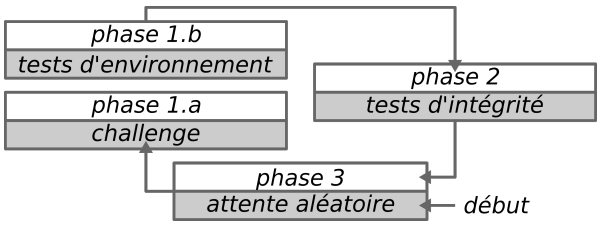

FIGURE 3.4 – Cycle de test d'intégrité

Le cycle de test d'intégrité est constitué de trois phases. La phase 1 correspond au test de l'hyperviseur de sécurité par le périphérique de confiance, au travers de challenges et de tests d'environnement. La phase 2 est dédiée à l'exécution des tests d'intégrité du logiciel observé par l'hyperviseur de sécurité (figure 3.4). La fréquence d'exécution du cycle de test d'intégrité varie. Le temps d'attente entre deux occurrences correspond à la phase 3. Ces trois phases sont décrites dans la suite.

3.5.1 Les challenges

Dans la première méthode (phase 1.a), le périphérique de confiance soumet un challenge à l'hyperviseur de sécurité afin de tester son comportement vis-à-vis du temps d'exécution attendu et de la réponse attendue.

3.5.1.1 Principe général

Chaque challenge consiste en l'exécution d'opérations non triviales qui, à partir de valeurs caractéristiques de l'intégrité de l'hyperviseur de sécurité, calculent une valeur appelée « solution » du challenge, avec un algorithme *non prédictible* (figure 3.5). En effet, la solution ne doit

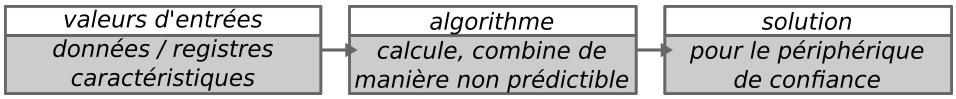

FIGURE 3.5 – Contenu des challenges

pas pouvoir être déduite par une simple analyse du code et nécessite forcément son exécution. Afin d'inhiber la possibilité de pré calcul ou de contournement de l'exécution du challenge par un attaquant de manière à répondre correctement en valeur et en temps, les challenges doivent être conçus avec précaution. Les prérequis suivants sont mis en application dans notre approche :

1. Chaque challenge n'est utilisé qu'une fois.
2. Chaque challenge est composé d'un ensemble d'instructions atomiques dont les bornes du temps d'exécution sont empiriquement connues et maximisées, ce temps étant très inférieur à temps de l'émulation de l'instruction.

Chaque challenge est donc associé à un temps d'exécution attendu. L'hyperviseur de sécurité doit l'exécuter et répondre dans ce délai. La réponse au challenge est collectée et vérifiée dans le périphérique de confiance lui-même. Le délai est contrôlé par un *timer* interne au périphérique, totalement dissocié de l'horloge du processeur principal. De cette manière, manipuler les timers internes du processeur de manière malveillante, ou d'autres compteurs de performance n'affectent pas le mécanisme de détection. Cette approche permet de détecter la virtualisation de notre hyperviseur de sécurité par un hyperviseur malveillant. Les challenges doivent être construits de manière à ce qu'ils nécessitent plus de temps d'exécution s'ils sont exécutés dans un niveau de privilège moins élevé ou s'ils sont interprétés.

3.5.1.2 Algorithme d'exécution des challenges

La figure 3.6 présente les machines à états représentant les différentes étapes de l'exécution d'un challenge du point de vue du périphérique de confiance ainsi que de l'hyperviseur de sécurité. Le périphérique de confiance initie l'algorithme en préparant le prochain challenge à envoyer à l'hyperviseur. Une fois le challenge prêt, il envoie une notification matérielle (interruption) au processeur principal pour interrompre son exécution et donner la main à l'hyperviseur de sécurité. Le périphérique se met alors en attente du téléchargement pour une durée déterminée en fonction de la taille maximum du code d'un challenge. L'hyperviseur prend la main, télécharge le challenge, envoie une alerte dans le cas d'erreur de téléchargement du challenge, le copie dans une zone mémoire exécutable puis l'exécute. Cette action notifie le périphérique de confiance qui relève la date de début d'exécution et se met en attente de fin d'exécution pour la durée maximum d'exécution du challenge. Si la phase de téléchargement n'est pas effectuée dans les temps par l'hyperviseur, le périphérique de confiance stoppe son attente et envoie une alerte à la machine de *monitoring*. Lorsque l'hyperviseur a terminé d'exécuter le challenge, il écrit la solution obtenue dans la mémoire du périphérique de confiance, puis termine l'exécution de son interruption. Si une erreur d'exécution apparaît, l'hyperviseur envoie une alerte. Dans tous les cas, exécution correcte ou erreur, l'hyperviseur réordonnance sa machine virtuelle pour continuer son exécution. Ensuite, le périphérique de confiance, notifié de la fin de l'exécution du challenge, relève la date de fin d'exécution. Enfin, si la durée d'exécution du challenge est située en dehors des bornes temporelles d'exécution minimum et

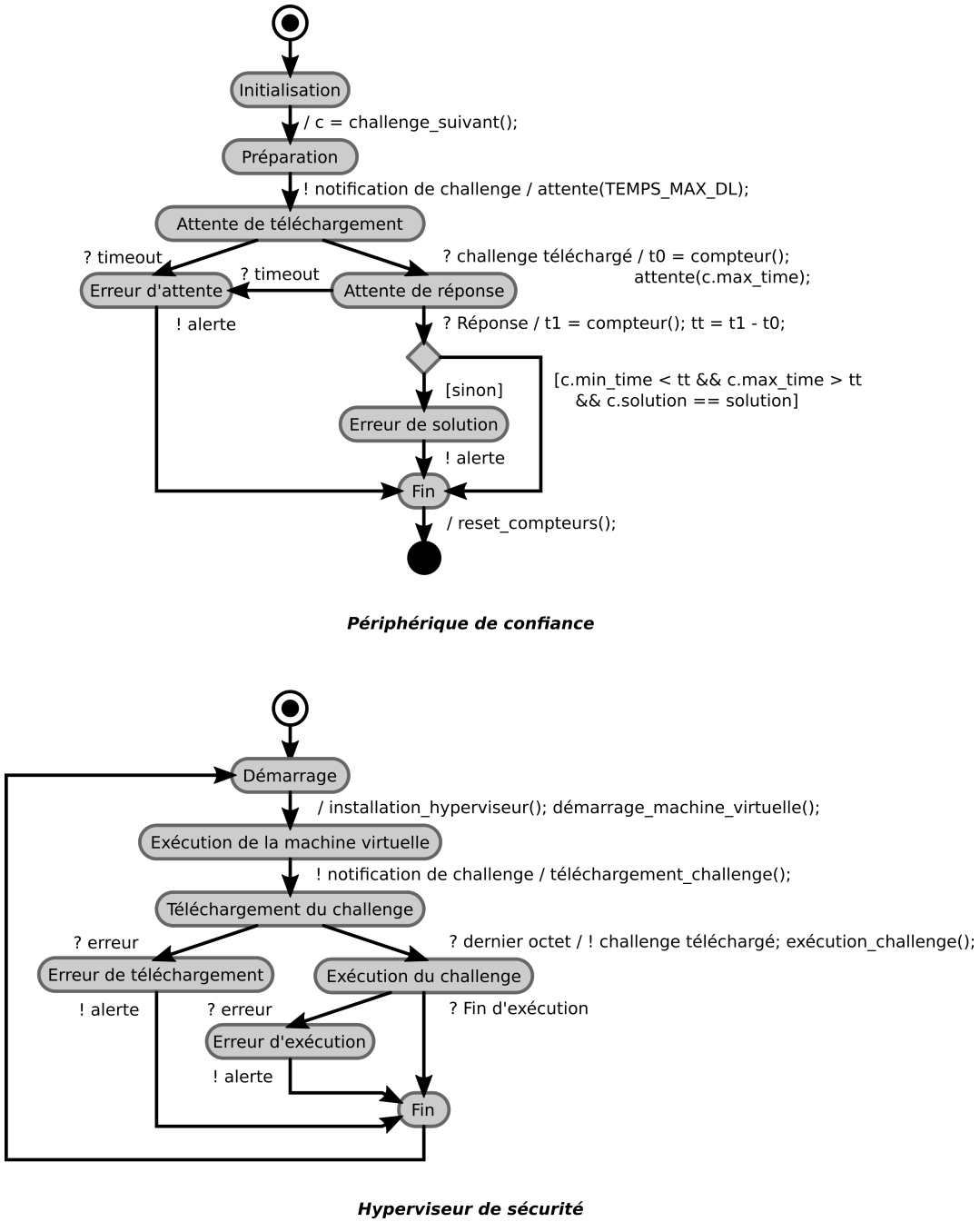

FIGURE 3.6 – Algorithme de traitement d'un challenge

maximum, ou que la solution est différente de la solution attendue, une alerte est envoyée par le périphérique de confiance.

3.5.2 Les tests d'environnement

La seconde méthode (phase 1.b) effectue les tests d'environnement sur la donnée et le code de l'hyperviseur de sécurité comme sur la configuration de certains composants qui appartiennent à l'environnement de l'hyperviseur de sécurité.

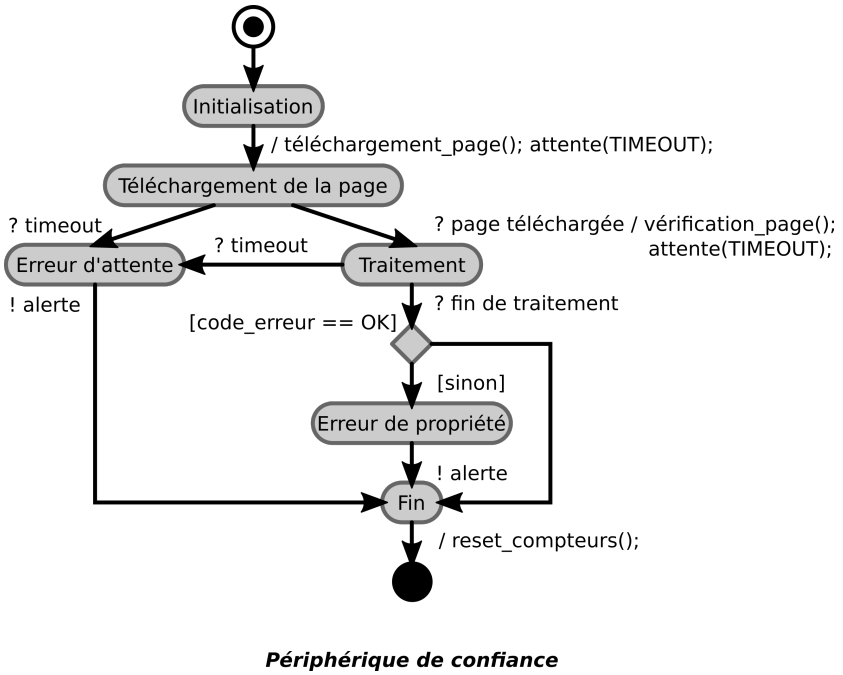

FIGURE 3.7 – Algorithme d'exécution d'un test d'environnement

3.5.2.1 Principe général

Ces tests sont importants, car une section du code ou des données de l'hyperviseur de sécurité pourrait être changée de manière malveillante ou un composant essentiel pour l'exécution correcte de l'hyperviseur de sécurité pourrait être altéré. Ces attaques pourraient avoir un impact non perceptible sur l'exécution des challenges, laissant le périphérique de confiance croire que la sécurité de l'hyperviseur n'est pas erronée. Mais ces altérations peuvent être identifiées en vérifiant le code et les données de l'hyperviseur de sécurité autant que les registres matériels des composants essentiels.

Pour des raisons de performance, ces tests d'environnement peuvent être accélérés par un coprocesseur dédié, que nous avons appelé le *Fast Automata Code* (FAC), inclus dans le périphérique de confiance. Ce coprocesseur est capable d'exécuter des tests mémoires optimisés, avec des instructions dédiées. Ces tests sont conçus avec un langage dédié et un compilateur qui traduit ces tests en automates directement exécutés par le coprocesseur. Le contenu des tests d'environnement est donné en section 3.7.

3.5.2.2 Algorithme d'exécution des tests d'environnement

La figure 3.7 présente la machine à état représentant les différentes étapes de l'exécution d'un test d'environnement du point de vue du périphérique de confiance. Le périphérique commence par télécharger la page mémoire désirée. Dans le cas d'un *timeout* ou d'une erreur, une alerte est levée. Une fois la page téléchargée, le périphérique de confiance va lancer son algorithme de vérification des données. Il pourra utiliser le FAC. Une fois le traitement effectué, si les propriétés vérifiées ne sont pas respectées, une alerte est levée.

3.5.3 Tests d'intégrité du logiciel observé

La phase 2 consiste en l'exécution des tests d'intégrité du logiciel observé par l'hyperviseur de sécurité. Les tests d'intégrité sont mis en œuvre via l'exécution de fonctions d'intégrité. Nous considérons que ces fonctions sont apportées par les concepteurs du logiciel observé et qu'elles sont en dehors du cadre de ces travaux. L'hyperviseur de sécurité doit proposer une architecture de sécurité de manière à les exécuter et envoyer les résultats des tests à la machine de *monitoring*.

Nous avons présenté dans cette section le protocole de tests d'intégrité. Il nous reste maintenant à définir le contenu des challenges et tests d'environnement que nous effectuons sur l'hyperviseur de sécurité afin de garantir son intégrité. Mais avant de pouvoir déterminer ces tests, il nous faut tout d'abord caractériser l'intégrité de l'hyperviseur de sécurité. La section suivante donne une liste finie des caractéristiques de l'intégrité de l'hyperviseur. Enfin, la section 3.7 détaille la vérification, avec les challenges et tests d'environnement, de ces caractéristiques.

3.6 Intégrité de l'hyperviseur de sécurité

Dans cette section nous caractérisons l'intégrité de l'hyperviseur de sécurité. Une fois que les éléments logiciels et matériels clés de l'intégrité de l'hyperviseur de sécurité sont déterminés, nous pouvons concevoir les challenges et tests d'environnement spécifiques associés dans la section suivante.

Notre architecture cible pour l'instant les architectures x86 interconnectées avec le bus PCI Express. D'après l'analyse de ces architectures faite dans le chapitre 2, nous considérons que notre hyperviseur de sécurité est intègre si et seulement si : 1) son code et ses données sont intacts ; 2) tous les registres de configuration du processeur assurent son installation en tant que gestionnaire des évènements privilégiés de la machine et 3) les périphériques sur lesquels il s'appuie pour fonctionner ou communiquer sont aussi intègres. Dans la suite de cette section, nous listons les différents aspects caractérisant l'intégrité de notre hyperviseur de sécurité.

3.6.1 Espace mémoire et chargement

Le comportement de l'hyperviseur de sécurité est en grande majorité défini par son code et ses données chargés au démarrage (caractéristique 1) qui doivent être conservés intacts. Il est aussi indispensable de s'assurer que l'hyperviseur de sécurité est chargé au plus tôt au prochain redémarrage de la machine physique (caractéristique 2).

3.6.2 Modes et niveaux de privilèges

La configuration des modes d'exécution du processeur détermine grandement l'intégrité d'un hyperviseur. Comme expliqué dans le chapitre 2, le support matériel des architectures x86 Intel pour la virtualisation est apporté par les extensions VT-x. L'hyperviseur s'exécute dans le plus privilégié des deux modes apportés par ces extensions, le mode hyperviseur (caractéristique 3). Ce mode d'exécution fonctionne en collaboration avec les niveaux de privilèges apportés par la segmentation du mode protégé. Nous devons nous assurer que nous maîtrisons aussi ces quatre niveaux de privilèges (caractéristique 4). La caractéristique 5 assure également que le code de l'hyperviseur est bien installé comme gestionnaire des interruptions

de machines virtuelles via un *handler* dédié dans son code et qu'un chemin d'exécution depuis ce handler amène bien à l'exécution des challenges.

Parfois, pour gérer la plateforme ou communiquer avec le firmware par exemple, l'hyperviseur doit dégrader son mode pour retourner dans un des modes historiques (par exemple, VT-x → 64 bits → 32 bits → 16 bits). Il est indispensable de s'assurer que le code, les données et les structures de contrôle de ces modes sont toujours intacts. Cela mène à la caractéristique 6.

Il existe d'autres modes de fonctionnement privilégiés que nous ne traitons pas dans notre architecture. Le mode de *management* de la plateforme est un mode à part et très privilégié de la machine. Son code est installé et verrouillé en RAM SMRAM par le firmware (BIOS) au démarrage. Ce mode est en pratique plus privilégié que le mode hyperviseur (*VMX root operation*), mais il est aussi beaucoup plus fermé, car destiné aux développeurs de firmwares et constructeurs. Les différences de privilèges entre les deux modes étant infimes et la difficulté d'instrumentation élevée par rapport au faible apport à notre preuve de concepts, nous avons décidé d'écarter son instrumentation pour le moment. Un autre mode est aussi apporté par les extensions matérielles permettant un démarrage sécurisé avec vérification « dynamique » de la configuration de la plateforme en plus de l'attestation du code et des données chargées : Intel Trusted Execution Technology (TXT). Ces extensions apportent donc des modes de fonctionnement très privilégiés, mais dédiés au démarrage et disposant de peu de moyens de contrôle sur les couches moins privilégiées du logiciel. Il est possible de prendre la main sur l'exécution uniquement lors d'évènements comme la sortie d'hibernation ou l'extinction de la machine. Nous avons donc aussi écarté ce mode de fonctionnement pour notre hyperviseur. Ceci étant, ce mode est configurable et activable tout au long de l'exécution de la machine, nous devons nous assurer qu'il est désactivé (caractéristique 7).

3.6.3 Gestion de la mémoire et des caches

Les accès aux instructions et aux données s'effectuent via les différents niveaux de cache du processeur. Comme indiqué dans le chapitre 2, chaque cœur du processeur peut spécifier la politique de gestion des caches par segment via les registres spéciaux *Memory Type Range Registers* (MTRR) ou par page via *Page Attribute Table* (PAT). Ces configurations permettent d'indiquer la fréquence et les opérations (lecture ou écriture) qui engendrent soit un accès dans les caches soit un accès à la mémoire RAM. Elles sont extrêmement importantes pour des raisons de performances et de bon fonctionnement. De plus, certaines zones mémoire doivent obligatoirement posséder une politique de gestion des caches précise pour s'assurer de l'ordre des différentes lectures et écritures sous peine de dysfonctionnement. C'est le cas des zones mémoire dédiées aux périphériques. Les différentes configurations des caches forment donc un point important de l'intégrité de l'hyperviseur, aboutissant à la caractéristique 8.

Le deuxième point important de l'intégrité de la gestion de la mémoire et des caches est lié à la gestion de la mémoire virtuelle. La segmentation et la pagination sont deux outils de virtualisation de l'espace mémoire que notre hyperviseur configure pour s'installer en tant que gestionnaire de machines virtuelles et constituent la caractéristique 9.

3.6.4 Compteurs et fréquence de fonctionnement

La fréquence de fonctionnement des processeurs récents est configurable grâce à des registres spécifiques. Cette fréquence agit évidemment sur les performances de la machine. Dans

le cadre de l'exécution des challenges, il est très important de conserver cette configuration intacte, notamment pour répondre correctement et à temps (caractéristique 10).

L'hyperviseur s'appuie aussi sur des compteurs matériels pour reprendre la main sur la machine virtuelle et effectuer les tests d'intégrité. La configuration de ces compteurs, qui est aussi dépendante de la caractéristique 10, est indispensable pour notre architecture de sécurité et constitue la caractéristique 11.

3.6.5 Gestion des interruptions externes

L'identification du programme destinataire d'une interruption est configurée dans le cœur lui-même. Or, le cœur destinataire d'une interruption avec le niveau de privilège requis pour le traitement peut être configuré à plusieurs endroits. Cette configuration peut avoir lieu au niveau du gestionnaire d'interruptions programmable local du cœur. Elle peut aussi avoir lieu au niveau d'un gestionnaire d'interruptions partagé entre tous les cœurs. Enfin, elle peut avoir lieu au niveau du périphérique, si ce dernier est capable d'envoyer directement une interruption aux cœurs via des messages (*Message Signaled Interrupt* ou MSI). Chacune de ces méthodes d'interruption influe sur le comportement des cœurs du processeur et sur l'hyperviseur lui-même. Il est donc nécessaire de garantir ces configurations qui constituent la caractéristique 12.

3.6.6 Gestion des périphériques

Notre hyperviseur communique avec la machine de *monitoring* à l'exécution pour envoyer des informations sur son fonctionnement, mais aussi les alertes concernant les tests d'intégrité du logiciel observé. Pour communiquer avec cette machine, il utilise un contrôleur réseau de type Ethernet, présent sur la machine. Comme expliqué dans le chapitre 2, les périphériques sont des unités d'exécution secondaires configurables par le processeur. Dans le cas des contrôleurs réseaux IP/Ethernet Intel, le contrôle de la transmission / réception de trames se fait via des structures de contrôles partagées entre le processeur et le périphérique, placées en RAM. Tous ces emplacements ainsi que le mode de fonctionnement du contrôleur réseau (accélération IP / TCP, mode *promiscuous*) sont configurés dans les registres du périphérique mappés dans l'espace de configuration PCI Express. Cette configuration, formant la caractéristique 13, est indispensable pour l'envoi des alertes et des informations d'exécution de l'hyperviseur et doit donc être conservée pour le bon fonctionnement de notre architecture de sécurité.

3.6.7 Isolation

Les dernières caractéristiques de l'intégrité de l'hyperviseur sont ses configurations d'isolation vis-à-vis des autres composants logiciels et matériels de la plateforme. En effet, l'hyperviseur doit protéger son espace mémoire des accès malveillants des logiciels s'exécutant dans les machines virtuelles en ajoutant une couche de virtualisation pour l'espace mémoire de ces machines virtuelles (caractéristique 14), et des accès malveillants provenant des périphériques en virtualisant aussi leur espace mémoire (caractéristique 15). Aussi, l'hyperviseur contrôle l'exécution des machines virtuelles via des mécanismes matériels, ces contrôles sont aussi indispensables à l'intégrité de l'hyperviseur dans le cas où les machines virtuelles voudraient exploiter une faiblesse (caractéristique 16).

L'ensemble des caractéristiques que nous venons de présenter est résumé dans le tableau 3.1. Cette analyse n'a pas pour but d'être exhaustive, mais est destinée à prouver qu'il est

Numéro	Description
Caractéristique 1	Le code et les données statiques de l'hyperviseur sont intègres
Caractéristique 2	L'hyperviseur sera chargé en premier au prochain redémarrage de la machine
Caractéristique 3	L'hyperviseur doit s'exécuter en mode hyperviseur
Caractéristique 4	L'hyperviseur doit maîtriser les niveaux de privilège du mode hyperviseur
Caractéristique 5	L'hyperviseur est bien gestionnaire des interruptions de machines virtuelles
Caractéristique 6	Les structures de contrôle de changement de mode d'exécution sont intègres
Caractéristique 7	Le démarrage sécurisé Intel TXT est désactivé
Caractéristique 8	La configuration des politiques de cache est compatible avec les exigences de l'hyperviseur
Caractéristique 9	La configuration de l'espace mémoire de l'hyperviseur est respectée
Caractéristique 10	La fréquence de fonctionnement du système n'est pas modifiée
Caractéristique 11	La configuration des compteurs utilisés par l'hyperviseur est intacte
Caractéristique 12	La configuration de la gestion des interruptions externes au cœur est intacte
Caractéristique 13	L'hyperviseur maîtrise toujours la configuration de son contrôleur réseau Ethernet
Caractéristique 14	La mémoire virtuelle imposée aux machines virtuelles par l'hyperviseur est intacte
Caractéristique 15	Le contrôle d'accès sur les accès DMA des périphériques est intact
Caractéristique 16	L'hyperviseur garde son contrôle sur l'exécution d'actions privilégiées par les machines virtuelles

Table 3.1 – Liste des caractéristiques de l'hyperviseur de sécurité

possible d'établir clairement la liste des différentes caractéristiques de l'intégrité d'un logiciel. Après avoir caractérisé l'intégrité de notre hyperviseur de sécurité, nous présentons dans la section suivante la méthode que nous avons adoptée pour vérifier ces caractéristiques.

3.7 Conception des challenges et tests d'environnement

Cette section décrit l'usage des challenges et des tests d'environnement pour nous assurer de l'intégrité des caractéristiques de notre hyperviseur. Pour commencer, il est nécessaire de bien comprendre quelles sont les capacités « d'expression » de nos deux méthodes de test.

3.7.1 Expressivité des tests d'environnement

Les tests d'environnement sont des contrôles effectués à l'intérieur du périphérique de confiance sur des données récupérées depuis les zones mémoires accessibles de l'espace d'adressage physique des périphériques. Sur les cartes mères modernes, l'espace d'adressage physique des périphériques peut être très différent de celui du processeur pour des raisons de sécurité. En effet, la plupart des *chipsets* modernes interdisent certaines plages d'adresses aux périphériques, notamment les plages d'adresses PCI Express. Ces plages d'adresses accueillent deux

types de mémoires (en termes de sémantique associée à la donnée), le premier étant l'espace de configuration PCI Express, mappé à une adresse spécifique configurée dans le processeur et le second étant les mémoires mappées par les périphériques eux-mêmes. Nous pouvons citer par exemple les *buffers* de réception d'un contrôleur réseau ou le *framebuffer* d'une carte graphique. Il est donc, en général, impossible aujourd'hui d'accéder aux mémoires des périphériques PCI Express depuis un périphérique [1]. De plus, les registres internes du processeur ne sont pas accessibles via des accès DMA. Ainsi, nous considérons que les régions mémoires accessibles depuis notre périphérique de confiance sont uniquement des segments de la mémoire RAM principale de la plateforme.

Les tests d'environnement sont donc principalement des tests de vérification de l'intégrité de code, de données ou de structures mémoires placées dans la RAM (tables de pages, ou structures de contrôle de machines virtuelles). Il est très important de noter que ces tests ne couvrent pas tous les types de menaces. Ils doivent donc être complétés avec des challenges, présentés dans la suite.

Enfin, le bus PCI Express permet d'effectuer des accès DMA synchrones avec les caches du processeur, ce qui permet de diminuer les chances de non-détection d'écritures frauduleuses.

Pour conclure sur les tests d'environnement, les vérifications apportées demeurent simplistes et doivent être impérativement complétées par un challenge, mais permettent de soulager le processeur principal de longs calculs d'empreintes en les effectuant directement dans le périphérique de confiance.

3.7.2 Expressivité des challenges

Les challenges sont exécutés directement sur le processeur principal. Un challenge est un algorithme qui produit une solution dans un temps imparti.

Pour produire la solution, un challenge peut donc accéder à tous les registres et à l'espace mémoire complet d'un processeur s'exécutant en mode hyperviseur. Nous pouvons donc par exemple vérifier que le jeu de tables de pages testé via un test d'environnement est bien celui installé dans l'hyperviseur ou alors que la structure de contrôle de machine virtuelle testée via un test d'environnement est bien celle couramment utilisée. Comme indiqué dans les sections précédentes, les challenges doivent être non prédictibles et donc associer des calculs aux données à vérifier en entrée (registres, mémoire) pour pouvoir produire une solution. Un challenge peut par exemple utiliser les champs hôte (configuration de l'hyperviseur) de la structure de contrôle de machine virtuelle courante, comme entrée d'un algorithme de calcul d'empreinte comme SHA-1, pour produire la solution qui prouvera l'intégrité des différentes données utilisées en entrées du challenge et dont les valeurs sont connues par le périphérique de confiance.

Le challenge est la méthode qui possède la plus grande expressivité de nos deux méthodes de tests de l'hyperviseur de sécurité, mais elle est aussi celle qui coûte le plus en performances. Nous allons devoir combiner les deux méthodes pour réduire l'impact des tests d'intégrité de l'hyperviseur de sécurité sur les performances du système.

La solution en valeur d'un challenge est calculable de manière triviale en appliquant les entrées connues à l'algorithme exécuté dans le challenge. Or, le temps d'exécution est aussi un

1. Sauf cas particulier, par exemple les cartes vidéos sur des architectures très spécifiques peuvent effectuer ces accès pair-à-pair pour mettre en œuvre du calcul distribué.

facteur déterminant dans la validation d'un challenge. Pour cette raison, il est indispensable de le définir précisément.

3.7.3 Détermination du temps d'exécution

Un problème important concernant la conception des challenges est l'estimation de leur **temps d'exécution attendu minimum et maximum**. Pour nos travaux, une détermination empirique est suffisante.

Le challenge est premièrement exécuté plusieurs fois hors-ligne sur le même processeur que celui de la plateforme. À chaque exécution, les valeurs des compteurs du processeur, tel que le *timestamp counter*, sont relevées avant et après l'exécution, pour obtenir le **temps d'exécution mesuré**. Nous ajoutons à ce temps une marge correspondant au temps de propagation du message sur le bus PCI Express afin d'obtenir le temps d'exécution maximum attendu, publié avec le challenge. Le temps d'exécution minimum attendu est déterminé grâce à la plus petite valeur du temps d'exécution mesuré.

Le challenge est ensuite exécuté hors-ligne, mais virtualisé avec notre hyperviseur de sécurité, pour obtenir le **temps d'exécution virtualisé mesuré**. Le challenge doit être construit de manière à ce que le temps d'exécution virtualisé mesuré soit au moins 3 ou 4 fois supérieur au temps d'exécution maximum attendu. Cette différence permet alors à notre système de détecter l'émulation ou la virtualisation de notre hyperviseur de sécurité. Pour obtenir une telle différence, nous choisissons des instructions spécifiques pour le challenge, difficiles à émuler, et, ou qui produisent inconditionnellement des *VM Exits* (`cpuid`, `vmread`, `vmptrst`).

Le bus PCI Express et sa couche protocolaire peuvent mener à des variations de débit dues à une surcharge du bus, qui pourrait invalider l'utilisation d'un délai arbitraire dans le calcul du temps d'exécution attendu pour un challenge. Pour faire face à ce problème, la spécification PCI Express propose un mécanisme de qualité de service ou *quality of Service* (QoS) qui permet de ségréguer les messages PCI Express en différentes classes de trafic, les ordonnant par priorité. En utilisant la plus haute classe de priorité pour les messages échangés par le périphérique de confiance, nous obtenons des garanties supplémentaires sur le fait que les messages soient échangés dans un délai limité, même si le bus est surchargé par une classe de trafic de priorité moins élevée. Finalement, même si un attaquant est capable de réaliser une attaque par déni de service au niveau physique, générant une gigue artificielle sur le signal, après avoir gagné le contrôle sur le transmetteur / récepteur d'un périphérique vulnérable par exemple, cette attaque sera détectée par le périphérique de confiance qui ne recevra pas la réponse au challenge dans le délai attendu. Enfin, dans le cas peu probable où la gigue du signal est trop importante, générant trop de délais dans les transmissions, cela se traduira par une fausse alerte. Notons que dans nos expériences, nous n'avons jamais rencontré de telles situations.

3.7.4 Vérification des caractéristiques de l'hyperviseur de sécurité

Cette sous-section présente comment s'assurer de l'intégrité des caractéristiques de l'hyperviseur de sécurité avec les deux méthodes de test dont nous disposons. Les tests détaillés ici font référence à des notions de configuration du processeur présentées dans le chapitre 2. Dans le cas de vérification par un challenge, nous décrivons quelles sont les actions précises à effectuer sur le processeur pour accéder aux informations déterminantes pour l'intégrité,

comme un accès à un registre de configuration ou l'exécution d'une instruction particulière. Si un challenge accède au contenu d'un registre, la valeur lue sera utilisée comme entrée du calcul de la solution du challenge.

3.7.4.1 Espace mémoire et chargement

Caractéristique 1 Le code et les données statiques de l'hyperviseur de sécurité sont chargés en mémoire à une adresse fixée durant la conception de l'hyperviseur de sécurité. Pour s'assurer de leur intégrité, un calcul d'empreinte du code et des données statiques réalisé au moment de la compilation de l'hyperviseur, comparé à l'empreinte de ces mêmes zones mémoires à l'exécution, via un test d'environnement, permet de détecter toute modification.

Caractéristique 2 Notre machine utilise un *firmware* normalisé UEFI. Cette spécification impose l'utilisation d'une mémoire non volatile de type flash pour stocker la configuration de celui-ci et notamment l'ordre de démarrage de la plateforme. Les configurations du *firmware* sont stockées sous forme de dictionnaire dont l'identifiant d'accès est une chaîne de caractères de 2 octets par caractère. La clé identifiant la première entrée de l'ordre de démarrage est de la forme `BOOT_0000`. La donnée associée à cette clé est un chemin au format Windows, de type `fs[0-9]:\EFI\BOOT\BOOTX64.efi` stocké sous la même forme. Il suffit donc de vérifier les données de cette entrée pour vérifier que notre hyperviseur de sécurité sera bien l'image binaire chargée par le *firmware* au prochain redémarrage. Il est aussi nécessaire de vérifier le contenu d'autres entrées de configuration telles que celle qui indique si le démarrage est effectué en mode de compatibilité avec les anciens *firmwares* de type BIOS. Le contenu de la mémoire flash est accessible uniquement depuis le processeur, un challenge est donc nécessaire pour contrôler cette caractéristique.

3.7.4.2 Modes et niveaux de privilèges

Caractéristique 3 Vérifier la maîtrise du mode *VMX root operation* sans exécuter de logiciel sur le processeur nous semble impossible. Aussi, pour vérifier cette caractéristique, nous nous appuyons sur les challenges.

Notre hyperviseur exécute les challenges dans le mode le plus élaboré du processeur, le mode *IA-32e 64 bits*. Ceci implique que les bits `IA32_EFER.lme`, `cr4.pae`, `cr0.pe` et `cr0.pg` doivent être actifs et `rflags.vm` doit être à zéro. Aussi, l'hyperviseur est en mode d'opération *VMX root operation* ce qui implique que le processeur est en mode *VMX operation*, propriété vérifiable avec le bit `cr4.vmxe`.

Si notre hyperviseur est émulé ou virtualisé de manière malveillante, le seul moyen de véritablement confirmer que ces valeurs lues sont correctes est de mesurer puis vérifier le temps d'exécution de l'accès aux différents registres cités dans le paragraphe précédent. En effet, la plupart des instructions du paragraphe précédent sont lentes à émuler ou nécessitent une prise de contrôle inconditionnelle d'un hyperviseur malveillant, multipliant énormément le temps d'exécution d'une simple instruction.

Caractéristique 4 Pour exécuter les challenges de sécurité, toutes les instructions privilégiées doivent être accessibles, incluant les instructions VMX. Pour cette raison, les niveaux de privilèges courants et demandés doivent être *ring 0*. Cela se vérifie respectivement avec les bits

`cs.cpl` et `cs.rpl` qui doivent être à zéro. Enfin, les privilèges d'accès aux entrées / sorties via les instructions `in` et `out` nécessaires sont maximaux. Ainsi, les bits `rflags.iopl` doivent être aussi à zéro.

La maîtrise d'un niveau de privilège est aussi intimement liée à la configuration des *handlers* appelés lors d'exceptions ou d'interruptions. Cette configuration se fait dans la table IDT. Cette table peut être vérifiée via un test d'environnement calculant son empreinte. Cette vérification doit être complétée par un challenge allant lire le pointeur d'IDT courant avec l'instruction `sidt`. Intel a accéléré et sécurisé la gestion des appels système depuis une application via les instructions `sysenter` ou `syscall` qui déclenchent un changement d'état vers le code du noyau. Il est très important de contrôler les registres de configuration prévus à leur configuration via un challenge : `IA32_SYSENTER_CS`, `IA32_SYSENTER_EIP`, `IA32_SYSENTER_ESP` et `IA32_SYSENTER_STAR`.

Caractéristique 5 Notre hyperviseur prend la main sur l'exécution de la machine virtuelle suite à une interruption privilégiée envoyée par le périphérique de confiance. Ce changement d'état du cœur charge l'hyperviseur grâce aux champs dédiés de la structure de contrôle de machines virtuelles couramment chargée qui stocke un « état »du cœur dans lequel l'hyperviseur souhaite reprendre la main lors d'interruptions de machine virtuelle (*VM Exit*). Ces champs caractérisent l'état de l'hôte. Il est nécessaire de vérifier via un challenge l'adresse du code exécuté dans ce cas via les champs `HOST_RIP` et `HOST_RSP`. Le mode d'exécution courant est aussi stocké dans l'état de l'hôte. Les champs correspondant aux registres des caractéristiques 3 et 4 doivent être aussi vérifiés dans l'état de l'hôte.

Une fois que nous sommes assurés de la bonne installation de l'hyperviseur dans le structure de contrôle de machines virtuelles, il est nécessaire de vérifier l'intégrité du code pointé par cette configuration. Pour ce faire, un test d'environnement avec calcul d'empreinte des pages mémoire de code de l'hyperviseur suffit.

Caractéristique 6 La régression du mode *IA-32e 64 bits* vers le mode *16 bits* s'effectue avec une IDT dédiée. On peut y appliquer le même test d'environnement que précédent. En plus de cette IDT, les modes hors *IA-32e 64 bits* utilisent la segmentation, donc une GDT. Afin d'assurer le passage vers ces modes, nous devons aussi vérifier cette GDT via un test d'environnement de type calcul d'empreinte.

Caractéristique 7 Vérifier que les extensions TXT ne sont pas utilisées demeure simple à réaliser avec un challenge. Il suffit de vérifier le bit `cr4.smxe` qui indique si la procédure de démarrage sécurisé a été activée.

3.7.4.3 Gestion de la mémoire et des caches

Caractéristique 8 Notre hyperviseur utilise les MTRR définis par le firmware sans les modifier puis donne la main au système d'exploitation. Nous devons vérifier les valeurs de ces MTRR via un challenge de manière à savoir s'ils sont compatibles ou non avec l'exécution de l'hyperviseur de sécurité. L'espace mémoire de l'hyperviseur doit demeurer en politique *Write Back* et les régions mémoires de notre périphérique de sécurité doivent être *Uncacheable*. La zone pointée par le `BAR0` de notre périphérique, dans laquelle l'hyperviseur écrit la solution du challenge, peut être portée en *Write Through* sans qu'il y ait de conséquences graves sur le

protocole des challenges. Notre hyperviseur ne gère pas les caches avec PAT qui est configuré dans les tables de pages. Cette vérification est faite conjointement avec la caractéristique suivante.

Caractéristique 9 La vision de la mémoire de l'hyperviseur de sécurité est extrêmement importante. Notre hyperviseur de sécurité utilise une configuration de la mémoire virtuelle telle que les adresses virtuelles sont identiques aux adresses physiques (*identity mapping*). Comme expliqué précédemment, notre hyperviseur s'exécute en mode 64 bits. Nous devons donc vérifier l'intégrité des tables de pages mode 64 bits de l'hyperviseur via un test d'environnement de type calcul d'empreinte. Il est nécessaire de compléter cette vérification via un challenge qui lit à la fois dans le registre de contrôle `cr3` ainsi que dans la structure de contrôle de machines virtuelles courante si le registre de contrôle `cr3` stocké dans l'état de l'hyperviseur pointe bien vers les pages vérifiées dans le test d'environnement. En effet, la configuration de l'hyperviseur chargée après une interruption de machine virtuelle est stockée dans la section hôte de la structure de contrôle de machine virtuelle courante. Or, un hyperviseur malveillant peut changer temporairement la vision de la mémoire virtuelle au moment de l'interruption, puis charger la configuration légitime dans le registre `cr3` avant d'exécuter le challenge, laissant l'illusion d'une configuration correcte.

3.7.4.4 Compteurs et fréquence de fonctionnement

Caractéristique 10 La fréquence actuelle du processeur est configurée dans un registre PCI Express du contrôleur mémoire (`MCHBAR`). Cette fréquence a été choisie à la compilation et ne doit pas être modifiée. Seul un challenge peut lire dans les registres du contrôleur mémoire pour vérifier cette fréquence.

Caractéristique 11 L'hyperviseur de sécurité reprend la main régulièrement sur la machine virtuelle, en plus des challenges, de manière à exécuter les tests d'intégrité sur le logiciel observé. Pour cela il s'appuie sur le *timestamp counter*, un compteur précis intégré dans les cœurs du processeur. Ce compteur permet de notifier régulièrement l'hyperviseur. Il est configuré via le champ de la structure de contrôle de machine virtuelle courante `VMX_PREEMPTION_TIMER_VALUE`. L'activation de ce timer s'effectue avec le bit `ACTIVATE_VMX_PREEMPTION_TIMER` de la même structure. La valeur configurée dans le champ précédent est décrémentée toutes les `IA32_VMX_MISC & 0x7` fois que le TSC est incrémenté. Ces bits sont à vérifier via un challenge.

3.7.4.5 Gestion des interruptions

Caractéristique 12 Nous avons vu que les *handlers* associés aux interruptions et autres évènements étaient stockés dans l'IDT. La source de ces interruptions est répartie dans plusieurs configurations. La première est dans le gestionnaire d'interruptions local, le *local APIC*. Sa configuration contient notamment une table de traduction d'évènement en vecteur d'interruption pour les périphériques locaux, la *Local Vector Table* (LVT). Le chipset contient aussi un gestionnaire d'interruption pour ses périphériques locaux, l'*IO APIC*. Notre hyperviseur n'utilise pas les interruptions pour son contrôleur réseau. Enfin, il est nécessaire de vérifier si les ponts PCI Express autorisent les accès « montants » depuis le périphérique de confiance de

manière à ce que les *Message Signaled Interrupts*, prenant la forme d'écritures PCI Express, arrivent bien à destination. Toutes ces configurations doivent être vérifiées via un challenge.

3.7.4.6 Gestion des périphériques

Caractéristique 13 Pour tester l'intégrité du contrôleur réseau dont dépend l'hyperviseur de sécurité, nous devons combiner un test d'environnement, pour lire les descripteurs d'envoi et de réception dans la RAM, avec un challenge, pour lire si la configuration pointée par le `BAR0` dans l'espace mémoire PCI Express du contrôleur est bonne. Le test d'environnement ne peut pas être cette fois-ci de type calcul d'empreinte, car les descripteurs d'envoi et de réception sont dynamiques. C'est pour cette raison que nous devons utiliser dans ce cas le coprocesseur FAC et un automate de vérification.

3.7.4.7 Isolation

Caractéristique 14 Le premier aspect de l'isolation de l'hyperviseur est son isolation vis-à-vis des machines virtuelles. Pour ce faire, il va contrôler leur vision et accès mémoire avec EPT. De plus, les processeurs récents Intel supportent toujours les anciennes entrées / sorties via les instructions `in` et `out`. Il est nécessaire d'effectuer un test d'environnement sur les tables de pages EPT, ainsi que sur les *bitmaps* de contrôle des instructions `in` et `out`. Certains registres spéciaux des architectures x86, les MSR, permettent de configurer les cœurs. Certains de ces registres sont importants pour l'intégrité de l'hyperviseur. Les accès à ces registres depuis la machine virtuelle sont aussi contrôlés à l'aide de *bitmaps* que l'on peut tester avec un test d'environnement. Pour chacun des trois tests précédents, il est nécessaire de confirmer le résultat en allant lire via un challenge les pointeurs respectifs dans la structure de contrôle de machine virtuelle.

Caractéristique 15 Notre hyperviseur de sécurité se protège des accès malveillants de périphériques via le composant IOMMU. Sa configuration est aussi à la fois vérifiable via un test d'environnement (tables de pages) et un challenge pour confirmer ce premier test.

Caractéristique 16 Un autre aspect est celui de la configuration des actions qu'une machine virtuelle ne peut effectuer sans que cela génère une interruption de machine virtuelle. Cette configuration est placée dans la structure de contrôle de machine virtuelle et doit être testée avec un challenge.

Dans cette section nous avons montré comment tester l'intégrité de l'hyperviseur de sécurité à distance depuis notre périphérique de confiance à l'aide des entrées des challenges ou alors de tests d'environnement. Cette étude est résumée dans la table 3.2. Dans la section suivante, nous décrivons succinctement les effets de la virtualisation ou de l'émulation malveillantes sur l'exécution du challenge.

3.7.5 Détection de l'émulation ou de la virtualisation de l'hyperviseur

Dans cette section, nous discutons des effets de l'émulation ou de la virtualisation malveillante de l'hyperviseur sur les temps d'exécution, en particulier celui de l'exécution des challenges. Les challenges récupèrent leurs données d'entrées en exécutant des instructions qui

Numéro	Challenge	Test d'environnement
Caractéristique 1		√
Caractéristique 2	√	
Caractéristique 3	√	
Caractéristique 4	√	√
Caractéristique 5	√	√
Caractéristique 6		√
Caractéristique 7	√	
Caractéristique 8	√	
Caractéristique 9	√	√
Caractéristique 10	√	
Caractéristique 11	√	
Caractéristique 12	√	
Caractéristique 13	√	
Caractéristique 14	√	√
Caractéristique 15	√	√
Caractéristique 16	√	

TABLE 3.2 – Type de test par caractéristique d'intégrité

peuvent être privilégiées et dont leur émulation ou leur exécution influe grandement sur le temps d'exécution par rapport à une exécution classique.

Dans cette section, nous prenons l'exemple de l'exécution d'un challenge dont les valeurs d'entrée sont données par l'exécution de l'instruction `cpuid`. Cette instruction sert à identifier le modèle ainsi que quelques particularités du processeur sur lequel un logiciel s'exécute. Cette instruction n'est pas privilégiée et un programme qui s'exécute dans le niveau de privilège *ring 3* peut donc l'exécuter. Les concepteurs des micro architectures x86 n'ont pas jugé cette instruction critique pour la sécurité de la plateforme. Bien que non privilégiée, cette instruction génère un *VM Exit* lors de son exécution dans une machine virtuelle. Cette instruction fait partie de la liste des instructions générant inconditionnellement une sortie de machine virtuelle sur tous les processeurs Intel supportant les extensions VT-x. Par conséquent, le temps d'exécution de cette instruction par notre hyperviseur de sécurité doit être proche du temps nominal. Un temps d'exécution plus important peut donc correspondre à la génération d'un *VM Exit* et donc à la virtualisation de notre hyperviseur de sécurité.

3.7.5.1 Exécution classique

Comme le montre le diagramme de séquence très simplifié 3.8, dans le cas nominal, le challenge va exécuter cette instruction pour toutes les méthodes `cpuid` existantes. La méthode appelée est paramétrée avec le registre général `rax` et le résultat de l'instruction est stocké dans les registres généraux `r[abcd]x`. Chaque résultat d'exécution est copié en mémoire RAM avec l'instruction `mov`. Dans un challenge réel, les valeurs d'entrée obtenues avec l'instruction `cpuid` seront utilisées comme entrée d'un algorithme choisi pour calculer la solution copiée dans le périphérique de confiance. Cette écriture n'est pas représentée dans le diagramme de séquence de cette section.

Si l'hyperviseur est virtualisé par un hyperviseur malveillant, c'est-à-dire exécuté dans une

machine virtuelle, en dehors du mode d'exécution *VMX root operation*, la propriété sur l'instruction `cpuid` citée plus haut est vraie. C'est-à-dire qu'elle va générer une prise de contrôle par l'hyperviseur malveillant, ce qui dévoilera sa présence du point de vue du temps d'exécution.

3.7.5.2 Exécution virtualisée

Le diagramme de séquence 3.9 montre une exécution très simplifiée dans un contexte malveillant. Une fois le challenge téléchargé par l'hyperviseur de sécurité, il va exécuter le premier `cpuid`. Cette instruction génère la prise de contrôle par l'hyperviseur malveillant qui va exécuter son *handler* de sortie de machine virtuelle classique. Pour déterminer quel type de sortie de machine virtuelle vient de se produire, il est nécessaire d'exécuter l'instruction `vmread` pour aller lire cette information (*exit reason*) dans la structure de contrôle de machine virtuelle courante. Une fois que l'*exit reason* est lu, l'hyperviseur malveillant doit déterminer quelle est la procédure / fonction à exécuter pour pouvoir virtualiser correctement l'évènement. Il va donc exécuter une suite d'instructions `test` et `jmf` pour arriver à la bonne procédure (CPUID).

L'étape suivante est l'exécution de l'instruction `cpuid`. Pour ce faire, l'hyperviseur malveillant doit récupérer le registre général `rax` contenant le paramètre désignant la bonne méthode puis exécuter le `cpuid`. Il est maintenant nécessaire de recopier le résultat de cette exécution dans une structure de sauvegarde afin de la réinjecter dans les registres généraux `r[abcd]x` lors de la reprise de main pour l'hyperviseur de sécurité dans la machine virtuelle.

Pour préparer le réordonnancement de l'hyperviseur de sécurité, il faut mettre à jour son pointeur d'instruction `rip` stocké dans la structure de contrôle de machine virtuelle sur l'instruction suivant le `cpuid` virtualisé en exécutant l'instruction `vmwrite`. Enfin, il faut maintenant mettre en place l'environnement des registres généraux résultant de l'instruction `cpuid` virtualisée en recopiant les quatre registres `r[abcd]x` sauvegardés. Finalement, l'hyperviseur légitime peut être réordonnancé via l'instruction `vmresume` qui rend la main à la machine virtuelle courante. Clairement, toutes ces étapes nécessitent un temps d'exécution bien plus long que la simple exécution de l'instruction `cpuid` par l'hyperviseur de sécurité non compromis.

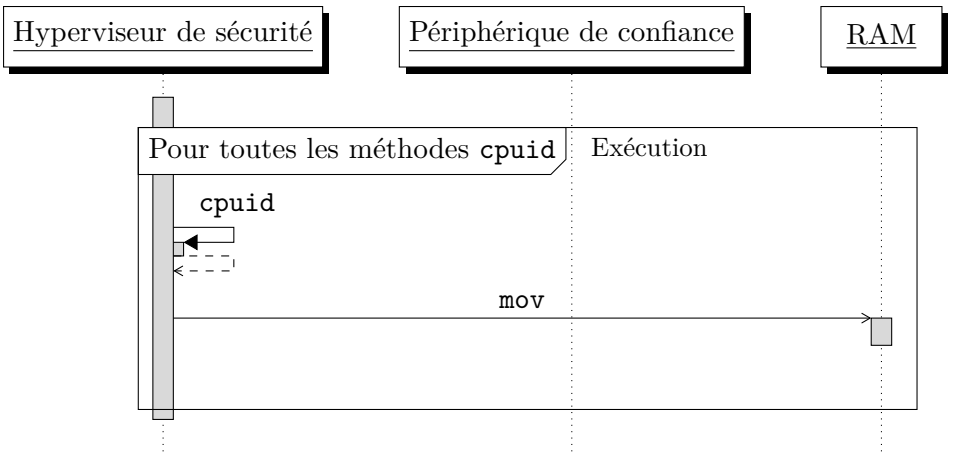

FIGURE 3.8 – Exécution nominale d'un challenge simple CPUID

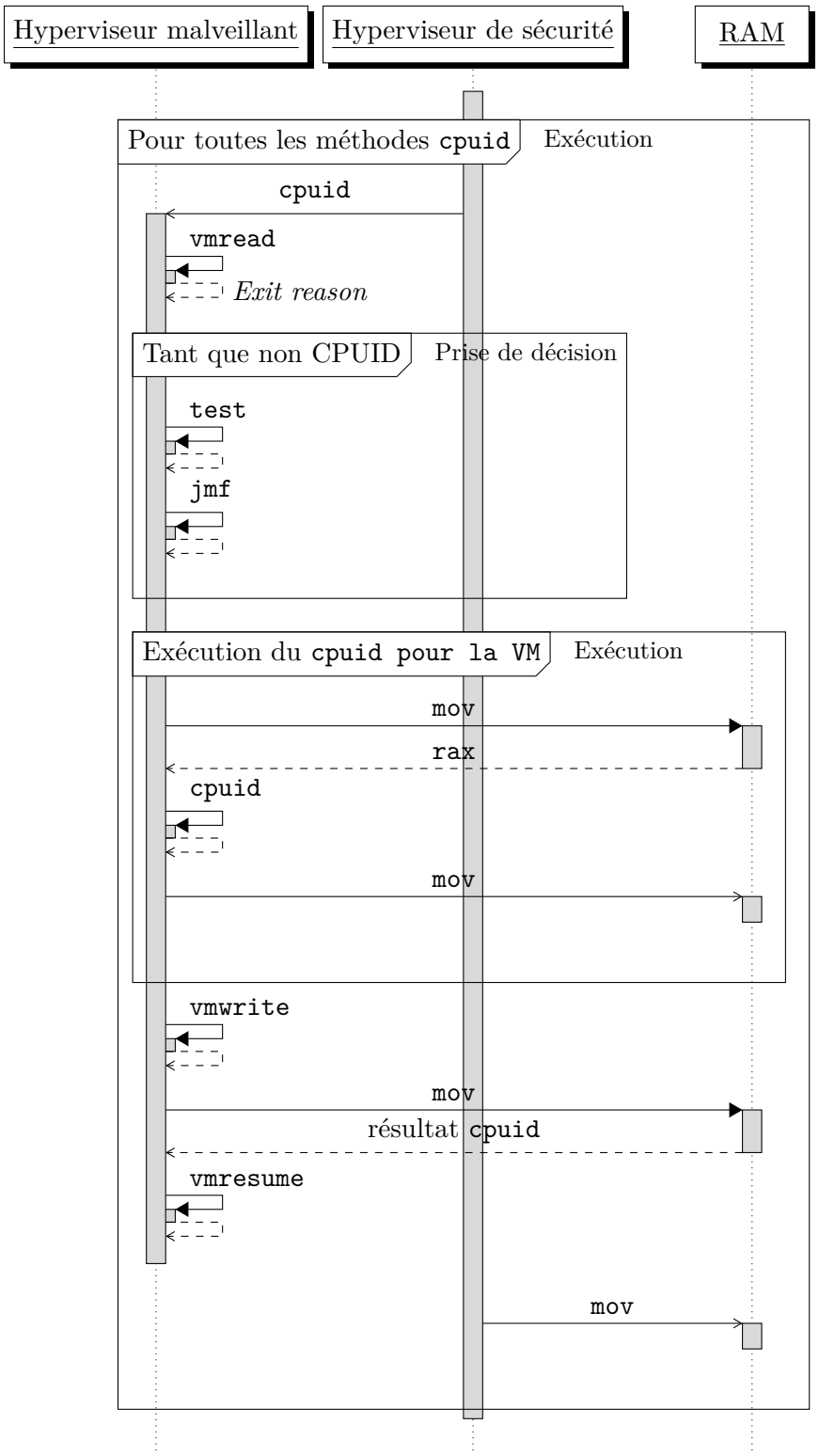

FIGURE 3.9 – Exécution virtualisée d'un challenge simple CPUID

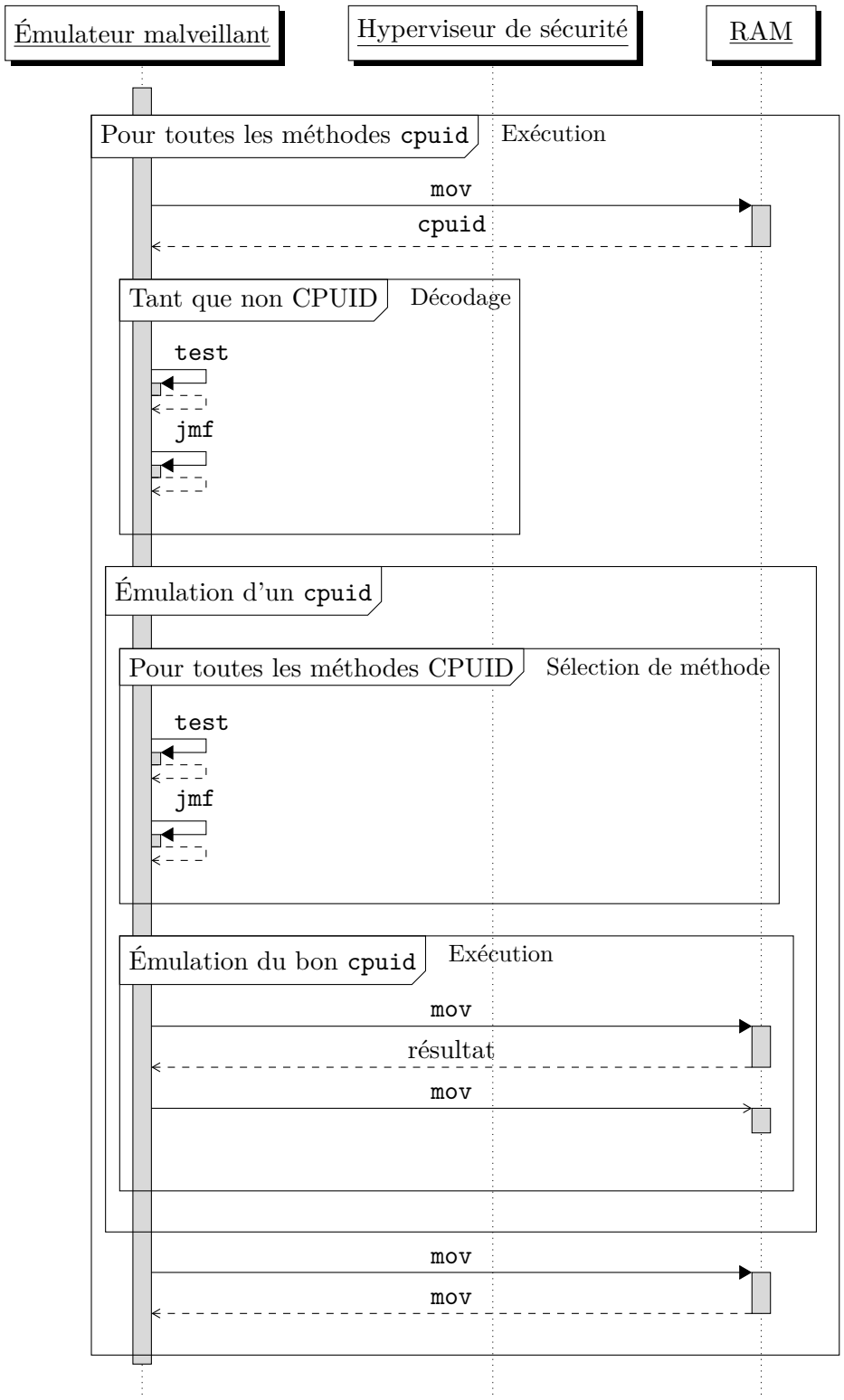

FIGURE 3.10 – Exécution virtualisée d'un challenge simple CPUID

3.7.5.3 Exécution émulée

Le deuxième cas d'exécution malveillante et celui de l'émulation. La figure 3.10 explique cette situation. Nous représentons ici une partie de la séquence de l'émulation de l'exécution

du challenge. En particulier au niveau de la lecture de l'instruction `cpuid` à émuler.

Pour commencer, nous sommes au niveau du début de la boucle d'exécution de l'émulateur malveillant qui va chercher en mémoire la prochaine instruction à exécuter. L'émulateur s'attend à émuler toutes les instructions du jeu d'instructions x86 étendu avec les dernières extensions matérielles (SMX, VT-x, AVX2, etc.). Il va donc décoder l'instruction en exécutant une très grande série de tests, avec les instructions `test` et `jmf`, sur le code opération téléchargé depuis la mémoire, avant de tomber sur le test du code opératoire de `cpuid`.

L'étape suivante appelle la procédure d'émulation d'un `cpuid`. Il existe différentes méthodes paramétrées par le registre général `rax`. Cette deuxième étape va donc encore une fois nécessiter l'exécution d'une grande série de `test` et de `jmf` avant d'identifier la méthode demandée.

L'émulation du `cpuid` va uniquement copier les registres résultats dans la structure mémoire de l'émulateur représentant un cœur de processeur et notamment les registres généraux `r[abcd]x`.

L'émulation de l'instruction `cpuid` est terminée et l'émulateur continue sa boucle d'exécution principale en récupérant la prochaine instruction qui est le `mov` de sauvegarde, deuxième instruction de la figure 3.8. La suite de l'exécution n'est pas représentée par manque de place.

De la même manière que pour la virtualisation, l'exécution des grandes séries de `test` et de `jmf` nécessite un temps d'exécution largement plus long que la simple exécution de l'instruction `cpuid` par l'hyperviseur de sécurité non compromis.

Notons que l'hyperviseur malveillant pourrait tout à fait ordonner ses tests pour identifier au plus tôt l'instruction exécutée par l'hyperviseur de sécurité (ici, en testant le `cpuid` en premier). Par contre, les challenges sont variés, donc un ordre adapté pour un challenge sera clairement mauvais pour un autre challenge.

Pour conclure cette section, nous constatons que l'émulation ou la virtualisation de l'exécution des challenges de l'hyperviseur de sécurité augmente considérablement le temps d'exécution du challenge. Ce qui permet donc de valider la caractéristique 3 qui indique que l'hyperviseur de sécurité doit s'exécuter dans le mode le plus privilégié du processeur, le mode *VMX root operation*.

3.8 Conclusion

Dans ce chapitre nous avons présenté la problématique du test d'intégrité de logiciel dans le cas ou nous ne pouvons pas faire confiance au processeur. Nous avons aussi décrit la solution hybride que nous proposons faisant intervenir à la fois un hyperviseur de sécurité qui joue le rôle d'environnement de tests d'intégrité sur un logiciel observé et un périphérique de confiance dont le but est d'assurer l'installation de l'hyperviseur de sécurité ainsi que son exécution. Afin d'élaborer les challenges et tests d'environnement exécutés sur l'hyperviseur, nous avons caractérisé l'intégrité de celui-ci et proposé pour chaque caractéristique une mise en œuvre d'un challenge et, ou d'un test d'environnement à cet effet. La mise en œuvre de notre architecture de sécurité est l'objet de la section suivante.

Chapitre 4

Mise en œuvre

Sommaire

4.1 Introduction

Ce chapitre présente, dans un premier temps, la conception du périphérique de confiance et de l'hyperviseur de sécurité. Ensuite, le cycle de test d'intégrité est détaillé. Ce chapitre se termine par une réflexion sur les éventuelles améliorations à apporter à cette implémentation logicielle et matérielle.

4.2 Périphérique de confiance

Dans cette section, nous détaillons l'architecture du périphérique de confiance. Nous commençons, en sous-section 4.2.1, par justifier l'utilisation du circuit électronique programmable de type FPGA puis par présenter la carte de développement FPGA que nous avons choisie. Nos travaux sont basés sur le projet open source Milkymist [72], que nous présentons en sous-section 4.2.2. Nous continuons avec le travail réalisé sur l'adaptation des fonctionnalités du *System On Chip* Milkymist et sur le support matériel de notre carte de développement

FPGA (sous section 4.2.3). La conception et la mise en œuvre de notre coprocesseur FAC sont détaillées dans la sous-section 4.2.4. Nous terminons cette section avec la présentation, en sous-section 4.2.5, du *endpoint* PCI Express que nous avons implémenté.

4.2.1 Prototypage de périphérique PCI Express

Il existe de nombreux bus de communication dans les architectures actuelles. Nous avons choisi le bus PCI Express pour développer notre périphérique de confiance, car, dans les machines modernes, ce bus est le bus dit système, c'est-à-dire que la grande majorité des communications entre périphériques et parfois dans les cœurs de processeur eux-mêmes utilisent à minima la couche applicative de cette norme de bus. S'installer en tant que périphérique PCI Express offre donc une position privilégiée dans la machine pour réaliser les tests d'environnement et les challenges tout en préservant l'indépendance de l'exécution du périphérique de confiance vis-à-vis du comportement des autres composants du système (processeur principal inclus).

Réaliser entièrement un périphérique PCI Express classique demande des compétences et du matériel de pointe en microélectronique. Par exemple, le bus PCI express, dans sa version 2, utilise une communication en série haute performance. Le développement de ce type de transmetteurs / récepteurs demande encore une fois un travail conséquent. Enfin, en plus des difficultés de réalisation physique du circuit, les itérations de conception de la logique elle-même du circuit, en plus d'être couteuses, seraient ralenties par les itérations de fonte et de test du circuit.

Pour cette raison, l'utilisation d'une carte de développement possédant déjà les périphériques de communication, tels que les émetteurs / récepteurs Giga bit pour la couche physique PCI Express, testés et fonctionnels, s'impose donc. Pour la réalisation de la logique du périphérique, il s'est avéré qu'il n'existait pas sur le marché, au commencement de cette thèse, de carte de développement PCI Express, de seconde génération [1], hébergeant un micro contrôleur. De plus, l'utilisation d'un micro contrôleur impose une réalisation purement logicielle de la logique des challenges et tests d'environnement, les couches du protocole PCI Express étant mises en œuvre dans un composant non modifiable de la carte de développement. Or, cette approche ne permet pas de connaître précisément la date de réception des messages, qui constitue une information capitale pour l'analyse des challenges. Il est nécessaire de maîtriser la couche supérieure du protocole PCI Express pour réaliser notre périphérique de confiance.

Pour toutes les raisons précédentes, il nous a semblé indispensable d'utiliser une carte de développement hébergeant un circuit programmable. La dernière génération de circuit programmable correspond aux circuits de type *Field Programmable Gate Array* (FPGA), qui ont la particularité d'être reprogrammables indéfiniment avec une logique matérielle sans modifier la microélectronique du circuit. On peut donc parler de virtualisation complète du circuit électronique dans le sens où la logique chargée dans ce circuit aura exactement le même comportement logique et temporel que s'il avait était réalisé comme un circuit classique.

Nous avons donc fait le choix de développer notre périphérique de confiance à l'aide d'une carte électronique de développement de périphérique PCI Express hébergeant un FPGA. Le terme carte de développement désigne en général un circuit imprimé de taille macroscopique sur lequel des connectiques et des composants d'entrées / sorties ou mémoires sont soudés et câblés, le système global étant dirigé par un processeur ou circuit généraliste maître.

1. Nous nous sommes focalisés sur cette version de cette norme, car il s'agit de la plus répandue.

4.2.1.1 Contraintes sur les périphériques

Pour un même modèle de FPGA, plusieurs cartes de développement sont disponibles avec divers périphériques. Notre architecture impose à notre carte de développement de contenir au moins des émetteurs / récepteurs Giga bit et une horloge compatible PCI Express, ainsi que des émetteurs récepteurs Giga bit et une horloge compatible Ethernet pour la communication des alertes du protocole de test d'intégrité.

4.2.1.2 Support de la pile PCI Express

Un autre aspect déterminant pour le choix de la carte de développement pour notre périphérique de confiance a été le niveau de support du PCI Express. En effet plusieurs constructeurs proposent des cartes de développement avec émetteurs / récepteurs Giga bit pour PCI Express. Cependant, un seul d'entre eux, au moment de notre étude de marché, proposait un circuit pour FPGA mettant en œuvre les 2 premières couches du protocole PCI Express ainsi qu'une partie de la troisième couche. Ce constructeur de FPGA est Xilinx, qui propose le cœur *LogiCORE™ IP Virtex®-6 FPGA Integrated Block for PCI Express®* [73] qui est extrêmement configurable et relativement simple d'intégration. Ce circuit propriétaire est distribué avec la carte de développement Xilinx ML 605. Nous avons donc sélectionné cette carte de développement, qui contient un FPGA Xilinx de famille Virtex 6.

4.2.1.3 FPGA et carte de développement Xilinx ML 605

Le modèle exact du FPGA que nous utilisons est `XC6VLX240T-1FFG1156`. Il dispose de 241152 cellules logiques programmables et de 1872 Mégas octets de blocs de RAM double porte de 36 Kilos bits chacune, très efficaces pour la synchronisation entre logiques séquentielles utilisant différents domaines d'horloges ou pour construire des mémoires internes. Ces blocs de RAM peuvent être aussi initialisés au chargement du circuit sur le FPGA, permettant de simuler une mémoire ROM pour stocker un *firmware* de processeur par exemple. Enfin, notre FPGA peut émettre et recevoir à une vitesse maximum de 5 Gigas bits par secondes par émetteur / récepteur Giga bit pour une fréquence maximum de 2.7 Gigas Hertz, ce qui est largement suffisant pour mettre en œuvre notre périphérique PCI Express de seconde génération.

La carte de développement Xilinx ML 605 met à disposition une barrette de mémoire RAM de 512 Mégas octets, utile pour les sections de données d'un programme. Le circuit de notre FPGA est programmé par un contrôleur système appelé *System Advanced Configuration Environment* ou *System ACE*. Ce composant est connecté à un contrôleur JTAG USB, permettant la programmation du circuit à distance via un câble USB, pendant la phase de développement. Ce contrôleur est aussi connecté à un lecteur de carte *CompactFlash* qui peut contenir le circuit électronique à charger au démarrage du FPGA, fonctionnalité utile une fois le circuit électronique développé, en phase de conception des challenges et tests d'environnement. La carte dispose évidemment d'un connecteur PCI Express normalisé au format 8x.

4.2.1.4 Approche SoC sur FPGA

Deux approches sont envisageables pour concevoir le circuit. La première consiste à développer un circuit contenant à la fois le *endpoint* PCI Express et la logique de gestion des

challenges et tests d'environnement. Cette approche minimise la taille du circuit et le temps de conception initial, mais est extrêmement difficile à mettre en œuvre en raison de sa faible capacité à communiquer avec le monde extérieur (par exemple pour le débogage ou encore pour effectuer des expérimentations légèrement déviantes du contexte initial). Le temps de développement de ce type de circuit étant lourd, peu adaptable et non évolutif, nous avons préféré suivre une approche *System On Chip* (SoC).

L'approche SoC constitue, dans notre contexte, un véritable système dans le système. Il fait intervenir un processeur généraliste, maître du système, qui va orchestrer les composants qui sont connectés sur son bus. L'avantage de ce type de circuit est qu'il est, malgré un temps de développement initial plus long, beaucoup plus modulaire et simple à modifier en cas d'évolution des fonctionnalités. En effet, avec cette approche, chaque fonctionnalité est mise en œuvre dans un morceau de circuit dédié, appelé cœur, qui est contrôlé et configuré par le processeur généraliste maître. Cela permet par exemple le développement d'une fonctionnalité dans un cœur indépendant, qui n'engendre pas de régression dans le reste du système durant la phase de développement, notamment dans les composants de communication et de débogage. Le dernier avantage de cette approche est qu'il existe des projets *System On Chip* généralistes *open source* pour FPGA, tel que Milkymist que nous avons choisi d'utiliser et d'adapter à nos besoins.

4.2.2 SoC Milkymist

Milkymist est un *System On Chip* pour FPGA originalement développé par Sébastien Bourdeauducq [74], dédié au *djing* vidéo, mais qui, en raison de sa qualité de conception, a été adapté et utilisé pour de nombreux projets.

Il est développé avec du logiciel et du matériel libres sur FPGA. L'objectif de la plateforme est d'offrir un environnement où l'utilisateur peut programmer ses scènes avec un logiciel appelé *Flickernoise* [75] proche de l'outil de visualisation musicale pour le lecteur audio Winamp, MilkDrop [76].

4.2.2.1 Principaux composants de Milkymist

Milkymist est orchestré par un processeur généraliste *big endian* développé par *Lattice Semiconductor*, le LatticeMico32 (LM32) (figure 4.1). Il utilise deux types de mémoires. Une première mémoire persistante de type *NOR flash* dans laquelle est stocké le *firmware* du LM32. La technologie NOR fournit de bonnes performances pour les accès aléatoires. Ces performances permettent d'exécuter le *firmware sur place*, sans avoir à le déplacer dans le deuxième type de mémoire, la DDR SDRAM 32 bits de 128 Mégas octets. Cette dernière est plus lente d'accès que la *NOR flash* mais elle est accessible en lecture / écriture lors de l'exécution du système et offre plus d'espace de stockage. Enfin, une carte SD est accessible pour du stockage de données de masse. Les communications réseau sont assurées par un contrôleur Ethernet. Une liaison série est disponible via une UART.

Tous ces composants sont interconnectés via différents bus présentés dans la sous-section suivante.

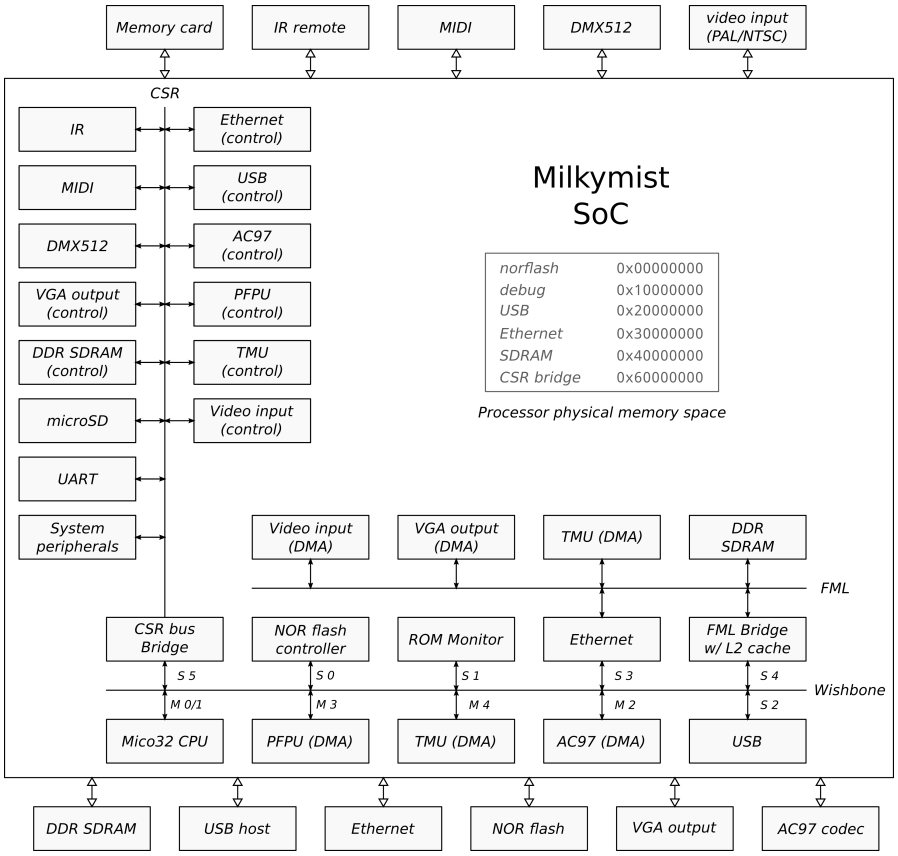

FIGURE 4.1 – SoC Milkymist

4.2.2.2 Bus de communication

La plupart des bus de communication utilisés par Milkymist reposent sur une architecture maître-esclave. Une interface maîtresse est capable d'initier un cycle de lecture ou d'écriture vers une interface esclave. Cette dernière se limite à répondre à ces cycles si elle est interrogée.

Bus système Le bus système utilisé par les périphériques du SoC pour les accès aux données et aux instructions est le bus 32 bits Wishbone. Ce bus est un bus généraliste et bas niveau par rapport au bus PCI Express. La sémantique de l'accès mémoire (lecture ou écriture) est donnée par des signaux dédiés (signal *write enable* pour demander une écriture par exemple), à la manière du bus PCI. Dans la configuration de Milkymist, chaque esclave est responsable d'une plage d'adresse de 256 Mégas octets dans laquelle un maître peut lire.

Bus de contrôle et de statut Deux bus supplémentaires sont utilisés dans Milkymist et connectés en tant qu'esclave sur le bus Wishbone via un pont. Le premier bus est spécialisé dans l'adressage de registres de contrôle et de statut, le bus *Control and Status Register* (CSR). Les 256 Megas octets de mémoire alloués au pont CSR sont partagés entres plusieurs périphériques. Cette approche permet à 16 périphériques de projeter leurs 1024 registres de configuration (de 32 bits chacun) sur le bus Wishbone.

Bus et cache mémoire DDR Le deuxième bus annexe utilisé par Milkymist est un bus dédié aux accès à la mémoire RAM DDR. Ce bus se nomme *Fast Memory Link* (FML). Les accès mémoire DDR étant lents, le pont du domaine Wishbone vers CSR contient un cache. Les accès mémoire effectués par les maîtres du bus FML sont effectués en rafales (*burst*). En effet, chaque maître possède un bus de commande et un bus de données, les deux n'étant pas synchrones, offrant la possibilité d'effectuer une lecture alors qu'une écriture est en cours et inversement. L'objectif de ce bus est de permettre l'accès à la RAM DDR à la fois par le processeur LM32 et les périphériques, en même temps.

Interconnexion des périphériques Le processeur LM32 est connecté au bus système Wishbone. Il utilise deux interfaces maîtresses, une pour les données et une pour les instructions. En plus du processeur, quatre autres composants maîtres sont connectés : les deux composants vidéos, le composant USB ainsi que le composant audio. Le reste des composants du *System On Chip* Milkymist sont des esclaves. Milkymist utilise un processeur généraliste pour contrôler le système. Un *firmware* de démarrage doit être absolument exécuté par le LM32 à l'initialisation du système. Le *firmware* chargé au démarrage du système est stocké dans la mémoire *NOR flash*. Pour les communications réseau, Milkymist utilise la norme Ethernet. Le contrôleur MAC Ethernet est donc aussi connecté au bus Wishbone.

La plupart des composants esclaves du bus Wishbone disposent de registres de configuration qui sont exportés avec le bus CSR. Le composant Ethernet MAC y exporte des registres de contrôle de l'émission et de la réception de trames par exemple. D'autres composants ne sont connectés qu'à ce bus, car ils ne nécessitent pas de partager de larges zones mémoire avec le processeur. C'est le cas de l'UART qui ne dispose que de quelques registres dans le BUS CSR pour contrôler l'envoi et la réception de caractères en série au pont USB to UART. D'autres composants dédiés à l'audiovisuel ont des registres exportés sur le bus CSR (DMX, MIDI, télécommande infrarouge, etc.).

Enfin, certains des composants vidéos, le gestionnaire d'affichage VGA et l'entrée vidéo sont connectés au bus FML, car ils requièrent un accès massif à la RAM pour y copier ou lire de grosses quantités de données.

4.2.2.3 Interruptions

Le processeur LM32 supporte 32 lignes physiques d'interruptions. Ces interruptions sont délivrées directement via ces lignes et ne transitent pas via le bus système Wishbone comme cela peut être le cas avec le bus PCI Express et les SMI.

4.2.2.4 Démarrage et suite logicielle

L'architecture matérielle de Milkymist est relativement peu complexe. De plus, le processeur LM32 ne dispose pas de MMU ni de notion de niveaux de privilèges. Le système d'exploitation supporté par la carte Milkymist est le système d'exploitation temps réel RTEMS [77]. Une adaptation d'une distribution de GNU/Linux a aussi été effectuée a posteriori [78].

En ce qui concerne le démarrage de la plateforme, le premier logiciel exécuté est le *firmware* stocké dans la *NOR flash*. Ce *firmware* propose une interface de type *shell* qui permet de choisir entre différents modes de démarrage pour télécharger le noyau RTEMS (carte SD, série, TFTP et mémoire *NOR flash*) qui prend la main sur la suite de l'exécution, pour ensuite enfin charger l'application Flickernoise.

En ce qui concerne la compilation du code du *firmware* ou de Flickernoise, Milkymist utilise l'adaptation du compilateur GNU GCC pour le processeur LM32, réalisée par les auteurs du noyau RTEMS.

4.2.3 Conception du SoC pour le périphérique de confiance

Milkymist est évidemment un SoC ayant un objectif très différent du périphérique de confiance. Une très grande majorité de ses cœurs ne sont donc pas nécessaires à la réalisation du périphérique de confiance. Dans cette section nous décrivons très brièvement les différentes étapes d'adaptation matérielle et logicielle effectuées sur Milkymist pour développer notre périphérique de confiance. Toutefois, avant de débuter le développement du périphérique de confiance, deux étapes préalables sont indispensables :

1. Restructurer le système et l'épurer des composants logiciels et matériels qui ne seront pas utiles au périphérique de confiance ;
2. Adapter pour notre FPGA et carte Xilinx ML 605 les composants de Milkymist que nous réutilisons.

4.2.3.1 Restructuration du SoC

Du point de vue matériel, notre périphérique de confiance ne nécessite aucun des composants audio et vidéo. Parmi les interfaces de communication, nous préservons donc uniquement le contrôleur Ethernet et l'UART. En ce qui concerne les mémoires, nous avons besoin d'une mémoire ROM de démarrage et d'une mémoire RAM pour l'exécution du logiciel. Nous devons aussi connecter le contrôleur PCI Express et le coprocesseur d'automates de vérification FAC qui sont présentés dans les sous-sections 4.2.4 et 4.2.5. Le cœur propriétaire Xilinx est embar-

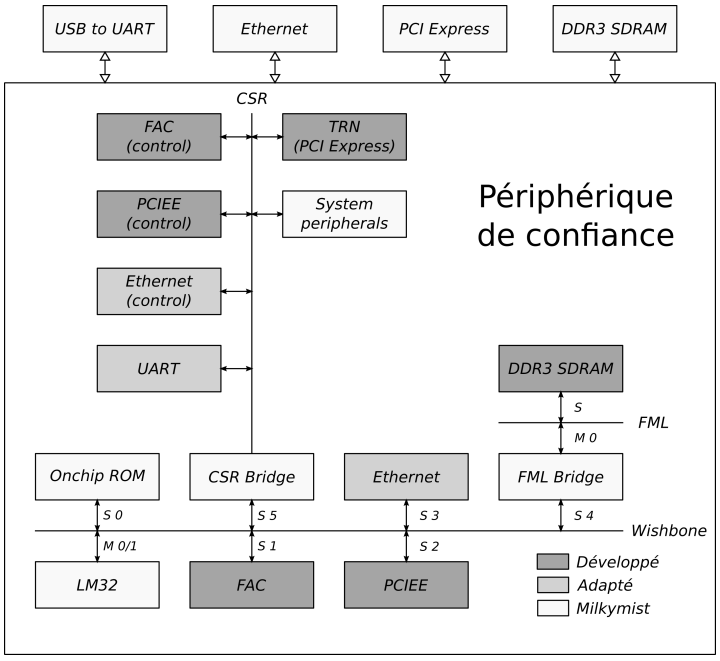

FIGURE 4.2 – System on chip

qué dans le composant TRN. Après restructuration matérielle, nous obtenons l'architecture représentée par la figure 4.2.

L'étape suivante consiste à adapter l'UART de Milkymist pour le pont *USB to UART* de notre carte. Cette étape s'avère anecdotique, car la carte Milkymist possède déjà un pont *USB to UART* similaire au nôtre. Nous ne le détaillons donc pas ici.

4.2.3.2 Adaptation du contrôleur Ethernet

Précédemment au LAAS, le SoC Milkymist a déjà été adapté pour développer une plateforme d'attaque par entrées / sorties PCI Express appelée IronHide [79]. Dans ce cadre, un pont Whishbone vers le contrôleur MAC Ethernet propriétaire Xilinx a été développé. Le FPGA Xilinx supporté par IronHide est un PICO E-17 qui héberge un FPGA Virtex 5, dont le cœur propriétaire qui implémente le contrôleur MAC Ethernet est très similaire à celui du Virtex 6 de notre carte de développement ML 605. Nous avons donc adapté le cœur propriétaire Xilinx qui implémente une couche MAC Ethernet pour notre FPGA et le pont Whishbone MAC Ethernet développé pour IronHide. Notons que nous n'avons pas pu réutiliser IronHide, car il ne permet pas de connaître de manière précise la date de réception des messages PCI Express, qui est une information importante pour les challenges.

4.2.3.3 Adaptation de la mémoire de stockage du firmware

La carte Milkymist possédait une mémoire flash de type NOR, performante pour l'exécution à la volée de code. Notre FPGA ne dispose pas du même type de mémoire flash, le cœur *NOR flash* de Milkymist n'est donc pas immédiatement transposable pour notre FPGA. Or, notre FPGA dispose de 1.8 Mégas octets de mémoire de type *RAM block* interne au FPGA qui sont initialisables au chargement du circuit. Nous avons donc constitué une RAM *Whishbone* avec ces *RAM block*, initialisés à la synthèse avec un fichier de données. Avec cette technique, nous pouvons simuler le comportement de la *NOR flash* de Milkymist en y installant le *firmware* du périphérique de confiance. Ce composant est appelé Onchip ROM (figure 4.2).

4.2.3.4 Adaptation du contrôleur mémoire DDR SDRAM

La mémoire DDR de Milkymist est câblée sur le bus FML. Ce bus est dédié aux accès massifs à la mémoire DDR SDRAM de Milkymist. Les composants de notre périphérique de confiance n'ont pas de tels besoins. Toutefois, en vue de futures améliorations, ce bus a été conservé et adapté à la mémoire DDR3 SDRAM. Cette adaptation s'est avérée non triviale. Effectivement, les domaines de fréquence du SoC et de la RAM DDR3 SDRAM sont différents, ce qui nécessite une synchronisation inter domaines.

4.2.3.5 Cartographie mémoire

Cette réorganisation de Milkymist a bouleversé la cartographie mémoire Wishbone et CSR (figure 4.1). La connexion des composants sur le bus Wishbone s'effectue via un composant d'interconnexion qui inclut un arbitre en cas d'accès simultanés au bus par deux composants maîtres.

Le composant d'interconnexion Wishbone alloue 256 Mégas octet d'espace d'adressage par esclave. Il offre la possibilité de connecter 5 maîtres et 6 esclaves. Notons à nouveau que le *CSR bridge* est un pont vers un autre bus qui a aussi sa propre cartographie mémoire. L'espace mémoire en aval du bridge CSR correspond aux registres des périphériques interfacés sur le bus CSR. Cette organisation aboutit à la cartographie mémoire présentée à la figure 4.3.

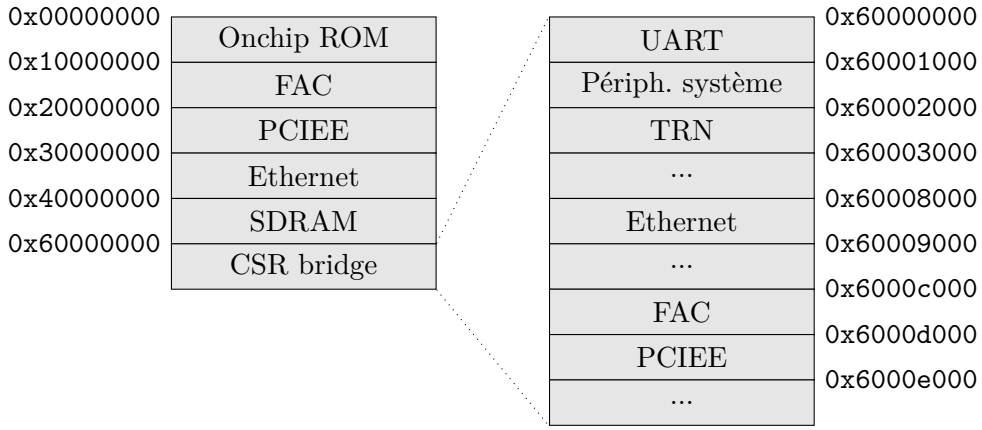

FIGURE 4.3 – Organisation mémoire du périphérique de confiance

4.2.3.6 Démarrage et suite logicielle

Le service rendu par le périphérique de confiance est très simple, car les algorithmes exécutés ne sont pas distribués et ininterruptibles (à l'exception de la gestion de l'UART et du FAC). Il n'est donc pas nécessaire de nous appuyer sur RTEMS, utilisé par Milkymist. Nous avons fait le choix de développer le code métier du périphérique de confiance dans l'environnement logiciel du *firmware* pour en produire une seconde version (le *firmware* métier), destinée à être téléchargée et exécutée à l'initialisation du périphérique.

En ce qui concerne la phase de démarrage, nous avons conservé la chaîne de démarrage initiale qui débute par l'exécution du *firmware* initial, embarqué dans la ROM. Ce *firmware* initialise les composants systèmes, le contrôleur MAC Ethernet et l'UART pour ensuite exécuter un premier *shell* minimaliste accessible via le pont *USB to UART* de la carte ML 605. Nous avons modifié la seconde phase du démarrage pour télécharger par TFTP le *firmware* métier de notre périphérique.

Pour éviter les problèmes d'interdépendance entre les deux *firmwares*, le *firmware* métier initialise tous les composants du périphérique, en incluant également les composants déjà initialisés par le premier *firmware*. Enfin l'unique application exécutée par notre périphérique est un second *shell*, extension du premier, qui permet notamment d'initialiser le cycle de tests d'intégrité. La figure 4.4 décrit les principales étapes du cycle d'exécution logicielle de notre

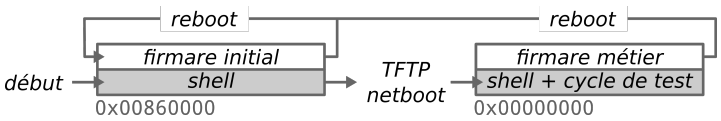

FIGURE 4.4 – Séquence de démarrage du périphérique de confiance

Op. Code	Mnémonique	Description
0x1	mask	Jump if mask not relevant
0x2	equ	Jump if not equal
0x3	inf	Jump if not lower
0x4	add	Add two registers
0x5	xor	Xor two registers
0x6	hamm	Hamming distance
0xc	int	Interruption
0xd	mload	Memory load
0xe	load	Immediate load
0xf	jmp	Jump

TABLE 4.1 – Jeu d'instruction actuel du FAC

périphérique de confiance. Notons que la phase de *netboot* via TFTP est très intéressante pour accélérer la phase de développement du *firmware* métier. En effet, il suffit de recompiler celui-ci et de redémarrer le périphérique de confiance via le *shell* et relancer la commande de *netboot* via le premier shell du *firmware* initial. Des détails sur l'implémentation de l'application du cycle de test d'intégrité et de ses interactions logicielles et matérielles avec l'hyperviseur de sécurité sont donnés dans la section 4.4.

4.2.4 Un coprocesseur d'automates

Le *Fast Automata Core* ou FAC est le composant chargé de réaliser les tests d'environnement. Il s'agit d'un coprocesseur disposant d'un jeu d'instruction dédié. Cette approche révèle des avantages présentés dans la suite.

4.2.4.1 Objectifs

Le coprocesseur FAC possède un jeu d'instruction minimaliste (table 4.1) spécialisé dans la caractérisation de données. Il ne peut pas exécuter de structures de contrôles complexes et les calculs arithmétiques qu'il peut exécuter sont très simples. L'objectif de ce coprocesseur est d'exécuter un automate de vérification de propriétés mémoires, chargé dans son espace mémoire de code et de partager via ses registres ou interruptions les différents codes de retour de l'exécution.

Les automates de vérification sont conçus et compilés sur une machine de développement classique et sont ensuite téléchargés depuis la machine de confiance au démarrage du périphérique de confiance. Chaque automate est dédié à la vérification d'une ou plusieurs propriétés sur un type de structure de données.

Les données à partir desquelles un automate est évalué correspondent au contenu d'une page mémoire téléchargée depuis le bus PCI Express avec PCIEE. Le coprocesseur FAC dispose d'une interface matérielle d'accès à cette page mémoire, dédiée et non concurrente (pas d'interblocage).

L'avantage de ce coprocesseur par rapport au processeur généraliste est qu'il est plus performant dans l'exécution des automates à l'aide de son jeu d'instruction spécialisé et de son unité arithmétique et logique de 64 bits. Ses accès mémoires se font directement sur des

mots de 64 bits. Enfin, les automates peuvent être chargés dynamiquement à l'exécution, ce qui facilite le développement des tests d'environnement.

4.2.4.2 Jeu d'instruction

Le FAC étant dédié à la caractérisation de zone mémoires, son jeu d'instruction est spécialisé en conséquence. Ce jeu d'instruction est adapté aux tests qui nous intéressent à l'issue de cette thèse, mais il peut évoluer. Les instructions sont simples et ont été choisies en se basant sur les particularités du fonctionnement bas niveau de l'OS et du processeur principal (beaucoup de champs de bits ⇒ instruction de masque ; beaucoup de structures arborescentes compactées en mémoire ⇒ instruction de distance de Hamming ; etc.). Dans la suite de cette section, les paramètres d'instruction sont préfixés par `r` et les valeurs immédiates par `$`.

`mask`	`r_val`	`r_m0`	`r_m1`	`r_adr`	L'instruction `mask` vérifie si la valeur du registre `r_val` respecte les contraintes `r_m0` et `r_m1`, passées en paramètres. Ces contraintes indiquent par exemple que certains bits sont autorisés à zéro et autorisés à 1. Si une des conditions n'est pas respectée, le FAC saute à l'adresse `r_adr`.

`equ`	`r_val0`	`r_val1`	`r_adr`	L'instruction `equ` compare deux registres `r_val0` et `r_val1` après les avoir masqués avec le registre `r_m`. Le FAC saute à l'adresse `r_adr` si les deux registres masqués ne sont pas égaux.

`inf`	`r_val0`	`r_val1`	`r_adr`	L'instruction `inf` saute à l'adresse `r_adr` si le registre `r_val0` n'est pas strictement inférieur à `r_val1`.

`add`	`r_res`	`r_val0`	`r_val1`	L'instruction `add` ajoute les registres `r_val0` et `r_val1` et stocke le résultat dans `r_res`.

`xor`	`r_res`	`r_val0`	`r_val1`	L'instruction `xor` calcule le ou exclusif entre les registres `r_val0` et `r_val1` et stoke le résultat dans `r_res`.

`hamm`	`r_res`	`r_val0`	`r_val1`	L'instruction `hamm` calcule la distance de Hamming entre les registres `r_val0` et `r_val1` et stoke le résultat dans `r_res`.

`int`	`r_code`	L'instruction `int` interrompt le processeur principal pour lui faire parvenir un code d'erreur. Le code 0 termine l'exécution.

`mload`	`r_dst`	`r_adr`	L'instruction `mload` charge le registre `r_dst` avec le mot de 64 bits à l'adresse donnée par `r_adr` dans la page mémoire à analyser.

`load`	`r_dst`	`$imm`	L'instruction `load` charge le registre `r_dest` avec la valeur immédiate en paramètre.

`jmp`	`r_adr`	L'instruction `jmp` saute inconditionnellement à l'adresse stockée dans `r_adr`.

4.2.4.3 Format des instructions

La taille des instructions est variable, pour économiser de l'espace mémoire (cf. tableau 4.2). La taille maximale d'une instruction est de 40 bits et les accès du FAC sont alignés à l'octet. Le contrôleur mémoire du bus d'instruction du FAC n'est donc pas trivial.

Le premier octet de l'instruction est son code opération, qui est divisé en trois parties. La première donne le type d'opération, le deuxième permet de spécifier une variante le cas

0		7	8 15	16 23	24 31	32 39
Code Op.						
Op.	Sous op.	Taille oper.	Opér. 1			
Op.	Sous op.	Taille oper.	Opér. 1	Opér. 2		
Op.	Sous op.	Taille oper.	Opér. 1	Opér. 2	Opér. 3	
Op.	Sous op.	Taille oper.	Opér. 1	Opér. 2	Opér. 3	Opér. 4
Op.	Sous op.	0x0	Valeur			
Op.	Sous op.	0x1	Valeur			
Op.	Sous op.	0x2	Valeur			
Op.	Sous op.	0x3	Valeur			

Table 4.2 – Format des instructions du FAC

échéant et le troisième dimensionne la taille des opérandes de l'instruction. En fonction de l'opération, l'instruction peut avoir de un à quatre opérandes ou bien une valeur immédiate de taille variable.

Les opérandes sont découpés en deux parties (cf. table 4.3). La première partie correspond au numéro du registre, sur 5 bits, et la seconde partie identifie un morceau de ce registre. Le banc de registres contient donc 32 registres de 64 bits. La taille du morceau du registre dépend de la valeur du champ correspondant à la taille de l'opérande. La valeur 0 pour la taille de l'opérande correspond à une instruction qui traite un octet. Il peut s'agir d'un des 8 octets du registre. La partie sélecteur de l'opérande peut donc prendre les valeurs entre 0 et 7. De la même manière, la valeur 1 pour la taille de l'opérande correspond à une instruction qui traite deux octets à la fois. Le registre est donc *artificiellement* découpé en 4 morceaux. La partie sélecteur de l'opérande peut donc prendre les valeurs entre 0 et 3 pour sélectionner un de ces morceaux et ainsi de suite.

Le bus d'instruction fait 48 bits. À chaque lecture d'une instruction, 6 octets d'instruction sont présents sur ce bus. Le FAC va traiter une partie de ces octets, en fonction du code opération – et donc de la taille de l'instruction.

0 7	
Opérande	
0 4	5 7
Sélecteur	Registre

Table 4.3 – Format des opérandes du FAC

4.2.4.4 Architecture matérielle

L'architecture matérielle du FAC est présentée à la figure 4.5. Le cœur du FAC comporte quatre composants : un banc de registres, un pointeur d'instruction, un décodeur d'instruction et une unité d'exécution. Le cœur dispose de trois bus : un bus d'instruction, un bus de données et une ligne d'interruption utilisateur avec données.

Le FAC dispose d'un contrôleur mémoire interne composé de huit *RAM blocks* pour un total de 32 Kilos octets, destinés à accueillir le code du FAC chargé par le processeur principal LM32. Cette mémoire interne contient deux bus : un premier connecté en esclave sur le bus *Whishbone* et un deuxième connecté au FAC. Le FAC peut effectuer des accès de 48 bits

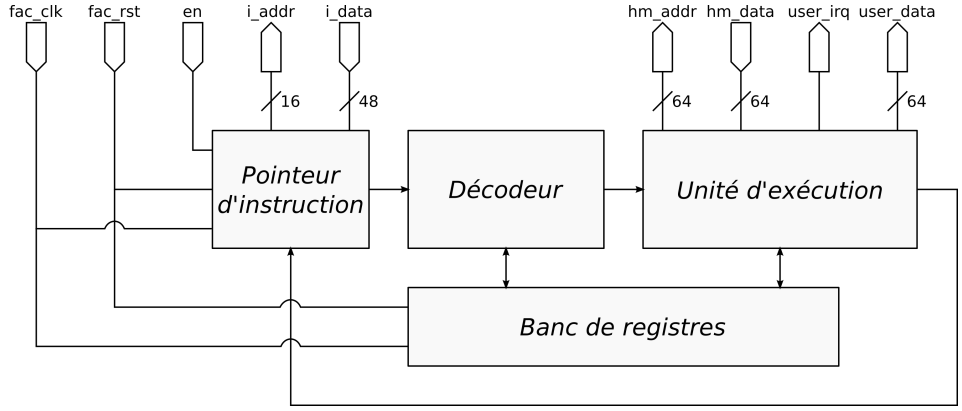

FIGURE 4.5 – Vue simplifiée du cœur du FAC

alignés à l'octet sur ces 32 Kilos octets tandis que le LM32 est aligné au mot de 32 bits. Cette mémoire contient donc également des traducteurs d'adresses.

La ligne d'interruption est connectée directement sur le vecteur d'interruption du processeur LM32, tandis que la donnée utilisateur est copiée dans un registre CSR du FAC. Enfin, la page mémoire PCI Express téléchargée est disponible via un bus mémoire dédié connecté au composant PCIEE décrit dans la section suivante. De manière à exécuter une instruction par front montant d'horloge le cœur ne dispose pas de pipeline et sa fréquence de fonctionnement est deux fois inférieure à celle de son interface mémoire. La figure 4.6 donne un schéma global du composant FAC et de ses interconnexions.

4.2.4.5 Assembleur

Afin de produire l'exécutable des automates pour le FAC nous avons développé un assembleur avec les générateurs de compilateur Lex et Yacc [80]. Le langage d'assemblage est très simple. Il supporte une instruction par ligne. Une instruction commence par une mnémonique prise parmi celles listées dans la table 4.1. Celles-ci sont suffixées avec la taille des opérandes donnée avec les lettres `b`, `w`, `d` et `q` pour 8, 16, 32 et 64 bits. Ensuite viennent les opérandes séparés par des virgules. Les valeurs immédiates commencent par un $ et le format des entiers

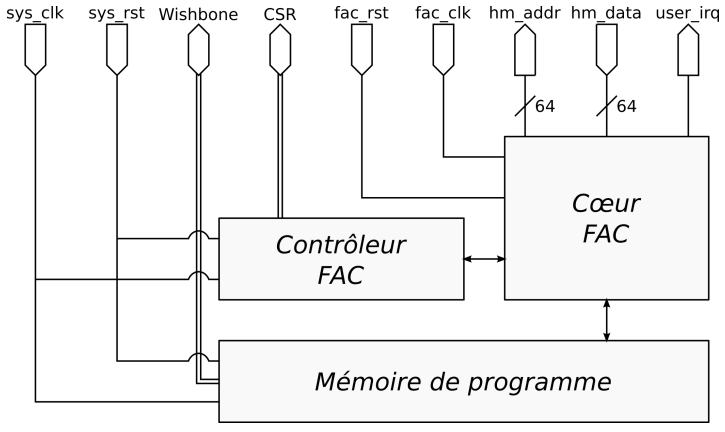

FIGURE 4.6 – Vue simplifiée du composant FAC complet

supporté est hexadécimal, préfixé par `0x`, ou décimal. Les registres sont préfixés par la lettre `r`. Le numéro de registre est ensuite donné en base décimale. Enfin le sélecteur, de 0 à 7, préfixe l'opérande en le séparant du numéro de registre via le caractère *underscore*. Les labels sont supportés et peuvent être utilisés pour charger un registre en vue d'un saut.

Le listing 4.1 montre comment vérifier la propriété de configuration de la mémoire en *identity mapping* pour une table de pages mémoire de 4 Kilos octets. Pour chaque entrée, le FAC vérifie simplement que l'offset contenu dans l'entrée de la page augmente bien de 4 Kilos octets par rapport à l'entrée précédente. Dans les autres cas, un code d'erreur est retourné.

Le FAC et les challenges nécessitent de télécharger des pages de mémoire hôte et de communiquer de manière plus générale sur le bus avec le processeur. La section suivante décrit comment ces communications sont réalisées.

```
/**
 * Page table ID mapping checking
 */
check_id_mapping:
  // Constants
  loadd r31, $0xff8 // 512 entries
  loadd r30, $0x8 // Increment
  loadb r29_1, $0x10 // Page increment 0x1000
  loadd r28, $0x1ff000 // cmp mask
  // Jumps
  loadw r20, $loop // loop start
  loadw r21, $endloop // loop end, ID mapping found
  loadw r22, $dead // No ID mapping found
  // Return codes
  loadb r10, $0x1 // ok
  loadb r11, $0x2 // ko
  // locals
  loadd r1, $0x0 // entry i
  loadd r2, $0x0 // current address
  loadd r3, $0x0 // i
// for (i = 0; i < 0x1000 - 8; i = i + 8)
loop:
  infd r3, r31, r21 // if not (i < 0xff8) => fin
  mloadq r1, r3 // r1 = entry i
  // if (entry i != current addr)
  equd r1, r2, r28, r22 // Error, no ID mapping
  addd r3, r3, r30 // i = i + 8
  addd r2, r2, r29 // i = i + 0x1000
  jmpw r20
dead:
  intq r11 // KO
  intq r0 // end
endloop:
  intq r10 // OK
  intq r0 // end
hamm_end:
```

Listing 4.1 – Vérification de la configuration mémoire ID mapping de l'hyperviseur

4.2.5 Développement de l'interface PCI Express

Le chapitre 2 a présenté le PCI Express comme une spécification de bus avec un modèle à trois niveaux : physique, liaison de données et transactionnelle. Pour développer notre composant PCIEE, nous nous appuyons sur le composant propriétaire de Xilinx présenté en sous-section 4.2.1.2. Ce composant implémente la couche physique et la couche de liaison de

données, et propose une réalisation extensible de la couche transactionnelle. Cette dernière est la plus proche du métier du périphérique développé. Nous nous interfaçons avec cette couche pour mettre en œuvre les services du périphérique de confiance.

4.2.5.1 Le cœur propriétaire Xilinx

Le cœur propriétaire Xilinx implémente toutes les fonctionnalités demandées à minima à un périphérique PCI Express. Par exemple, chaque périphérique ou *endpoint* doit exposer des registres de configuration compatibles avec l'entête de configuration de *type-0*. Ces registres sont accessibles uniquement par le *host bridge*, donc par le processeur, via des messages de lecture et d'écriture de configuration. Ces communications sont prises en charge par le cœur propriétaire.

Le cœur propriétaire ne gère pas les messages de lecture et d'écriture mémoire, car ils sont souvent spécifiques au métier du périphérique et doivent être traités en conséquence.

Trois régions de mémoire sont nécessaires pour notre interface matérielle : une de type *Expansion ROM* en lecture seule qui contient les challenges, et deux en lecture / écriture pointées par le `BAR0` et `BAR1` de l'espace de configuration PCI Express, pour respectivement accueillir la solution des challenges et recueillir des informations diverses sur l'hyperviseur qui nous ont aidés à réaliser les tests d'environnement. La déclaration de ces régions mémoires se fait à la synthèse du cœur propriétaire.

Lors de l'utilisation du périphérique et après l'initialisation de ces régions par un driver exécuté sur le processeur principal de la plateforme x86, le cœur propriétaire nous indique via un bus dédié appelé `bar_hit` si une des trois régions mémoire a été interrogée. Si un `bar_hit` survient, pour une de nos trois régions mémoires, nous devons recevoir et interpréter le message d'accès mémoire et le traiter en générant une réponse (pour une lecture par exemple) ou en mémorisant en interne la valeur (dans le cas d'une écriture). La réception et l'envoi de trames PCI Express transactionnelles s'effectuent via l'interface matérielle transactionnelle TRN proposée par le cœur propriétaire.

4.2.5.2 Architecture du PCIEE

La figure 4.7 propose une vue simplifiée du composant PCIEE. Ce composant contient quatre modules métiers.

Le premier, appelé lecture mémoire, permet d'initier des accès DMA sur l'espace mémoire de la machine hôte et de stocker la page mémoire téléchargée pour un futur accès via le bus *Whishbone* ou pour les accès de données du FAC via le bus `hm`. Le second module, appelé écriture mémoire, permet d'écrire un mot de 32 bits dans l'espace mémoire de la machine hôte. Ces deux premiers modules sont contrôlés et initialisés via des registres CSR exposés au processeur LM32.

Les deux derniers modules gèrent les deux régions mémoire `BAR[01]` et l'*expansion ROM* dont les cycles de fonctionnement sont initiés par le processeur de la plateforme x86, lors de l'écriture de la réponse au challenge et du téléchargement de celui-ci.

Ces 4 modules disposent de mémoires internes accessibles en lecture / écriture par le processeur LM32 et le FAC pour le composant de lecture mémoire. Ces composants s'exécutent en parallèle et initient des cycles de transmissions / réceptions de trames transactionnelles via l'interface TRN. Or, le cœur propriétaire Xilinx ne propose qu'une interface matérielle. Nous

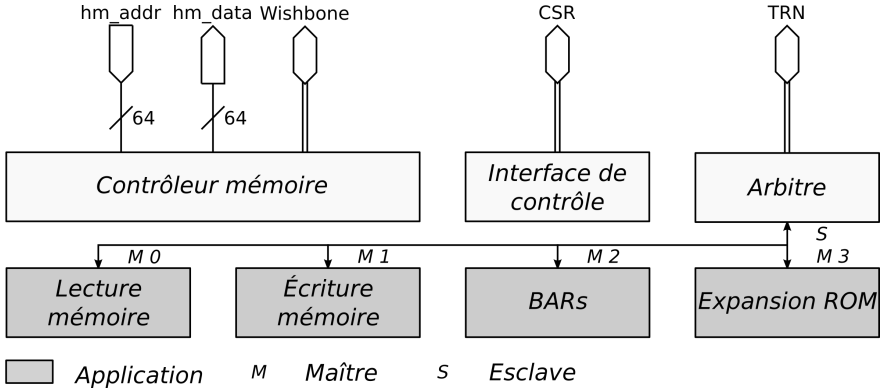

FIGURE 4.7 – Vue simplifiée du composant PCIEE

avons donc dû développer un composant d'interconnexion des quatre modules applicatifs et du bus TRN. Ce composant contient un arbitre matériel appliquant une priorité d'accès fixe fonction du dernier composant ayant eu accès au bus, de manière à éviter le phénomène de famine.

4.2.6 Conclusion

Le développement du périphérique de confiance a été présenté avec ses fonctions : exécuter les tests d'environnement (pouvant être assistés par le FAC), et gérer matériellement les challenges. La gestion logicielle des challenges est réalisée au niveau du processeur principal x86, par l'hyperviseur de sécurité. La conception de cet hyperviseur est présentée dans la suite.

4.3 Hyperviseur de sécurité

Notre hyperviseur de sécurité est conçu pour exécuter les tests d'intégrité sur les logiciels observés. Les logiciels observés peuvent posséder n'importe quel privilège dans le système (hyperviseur pris sur étagère, noyau de système d'exploitation, pilote matériel). L'hyperviseur doit aussi répondre impérativement aux challenges et laisser son espace mémoire accessible aux tests d'environnements. Pour cela il doit être installé en tant que composant le plus privilégié du système, en mode hyperviseur pour virtualiser les couches supérieures. Notre hyperviseur de sécurité doit donc être récursif, c'est-à-dire capable de virtualiser un nombre quelconque d'hyperviseurs potentiellement déjà installés. Il doit aussi communiquer via Ethernet avec la machine de *monitoring* pour envoyer les alertes des tests d'intégrité. Pour mettre en place ces communications, nous avons développé un protocole de contrôle de l'hyperviseur de sécurité, incluant un serveur de débogage de machines virtuelles. Un client de débogage et de contrôle de machine virtuelle est aussi disponible.

L'hyperviseur de sécurité est développé à l'aide d'une bibliothèque d'hyperviseur récursif que nous avons développée et appelée Abyme. Abyme possède deux phases d'exécution, son chargement / initialisation et la phase d'exécution des machines virtuelles. Dans une première partie, nous allons discuter la stratégie de chargement de notre hyperviseur de sécurité et de son initialisation (sous section 4.3.1). Dans une deuxième partie, nous détaillons quelques mécanismes et la conception choisie pour que l'hyperviseur assoie sa position privilégiée à l'exécution des machines virtuelles (sous section 4.3.2). La sous-section 4.3.3 explique comment

nous avons mis en place la virtualisation récursive, notamment en émulant les instructions VMX de VT-x. L'interface de programmation de la bibliothèque Abyme est présentée en sous-section 4.3.5, tandis que le code métier de l'hyperviseur de sécurité est présenté en sous-section 4.3.6. Le protocole de contrôle et sa mise en œuvre font l'objet de la dernière sous-section 4.3.4.

4.3.1 Stratégie de chargement et initialisation

Une des manières de minimiser la taille du code de l'hyperviseur est de s'appuyer au maximum sur les services logiciels déjà présents sur la machine sur laquelle on s'exécute. L'avènement des *firmwares* UEFI est pour cela un avantage, car non seulement ils proposent des services simplifiant la phase de chargement de notre hyperviseur, mais ces services sont également unifiés entre toutes les plateformes supportant cette interface de *firmware*, ce qui nous permet de ne pas perdre en compatibilité.

4.3.1.1 Stratégie de chargement

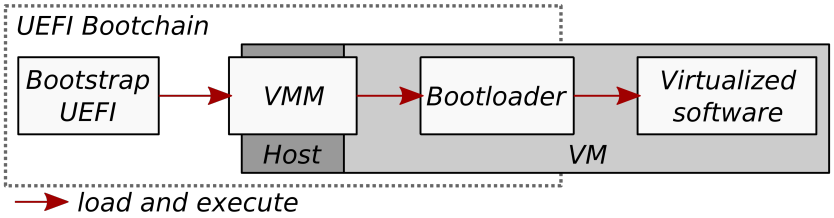

FIGURE 4.8 – UEFI Bootchain

Abyme est un driver de *runtime* UEFI chargé dans la phase de *preboot* de la machine par le chargeur de démarrage du *firmware* UEFI. Par la même occasion, ce chargeur marque l'espace mémoire d'Abyme comme étant occupé. L'hyperviseur active la virtualisation et s'installe comme hôte de la machine. Comme la stratégie adoptée est de limiter l'empreinte du code de l'hyperviseur, le contrôle de l'exécution est directement rendu au *firmware*, **dans une machine virtuelle**, afin de le laisser démarrer la machine comme si l'hyperviseur n'était pas présent (figure 4.8).

Les images UEFI que nous chargeons au démarrage ne sont pas stockées sur la flash interne de la plateforme, mais sur une partition FAT 32 UEFI dédiée sur le disque dur. Afin de faciliter l'enchaînement des opérations de chargement ainsi que pour simplifier la modification des images UEFI chargées, durant la phase de tests, nous utilisons le *shell* UEFI normalisé que nous programmons avec un *script shell*.

4.3.1.2 Initialisation

Lorsque l'hyperviseur prend la main, une fois chargé par le *firmware* UEFI, il exécute l'installation de toutes ses fonctionnalités. Cette étape nécessite l'allocation de pages mémoires. Encore une fois, nous nous appuyons sur le *firmware* pour allouer ces pages mémoires, afin de ne pas a avoir à développer et supporter un allocateur mémoire. La mémoire est donc uniquement allouée durant la phase d'initialisation. Cette zone mémoire est considérée statique dans la suite de l'exécution. Nous avons choisi cette solution pour diminuer la taille de l'image binaire

de l'hyperviseur [2].

Afin d'émettre sur le réseau Ethernet 1, l'hyperviseur s'appuie sur un driver Ethernet que nous avons développé et chargé au préalable via le *script shell UEFI* de démarrage. Deux autres modules sont ensuite chargés : la pile UDP/IP et le serveur de débogage. Ces modules permettent la communication des alertes de détection et le contrôle de l'exécution par le client de débogage et de contrôle exécuté sur la machine de *monitoring* [3].

Ensuite, l'hyperviseur initialise sa mémoire virtuelle, ses *handlers* de gestion des interruptions et sa gestion des caches pour pouvoir supporter une exécution en mode *IA-32e 64 bits*. L'hyperviseur configure également le contrôle d'accès des périphériques via l'IOMMU.

Une fois l'hyperviseur installé, il démarre les extensions de virtualisation et prépare les premières structures de contrôle de machines virtuelles qui servent à continuer l'exécution du démarrage de la machine dans une machine virtuelle. Durant cette étape, l'hyperviseur initialise plusieurs mécanismes matériels de contrôle d'exécution de la machine virtuelle qui sont détaillés dans la section suivante.

4.3.2 Exécution des machines virtuelles

Dans cette section, nous détaillons le comportement d'Abyme durant la phase d'exécution des machines virtuelles.

4.3.2.1 Stratégie de virtualisation

Nous souhaitons réduire l'empreinte mémoire de l'hyperviseur et éviter le développement de mécanismes complexes d'émulation et de virtualisation du matériel. Dans ce sens, l'hyperviseur n'émule aucun périphérique et se borne à virtualiser les mécanismes matériels architecturalement obligatoires ou indispensables pour la sécurité. De plus, il s'accapare la carte réseau Ethernet 1 en masquant son existence aux yeux de la machine virtuelle. De cette manière, le reste du matériel est pris en charge directement par la machine virtuelle sans mettre à contribution l'hyperviseur de sécurité.

L'objectif de l'hyperviseur est d'obtenir un environnement d'exécution plus privilégié que le logiciel observé et qui est présent originalement sur la machine. De ce fait, toute la pile logicielle est exécutée dans une seule machine virtuelle. Cette approche évite la mise en œuvre de l'ordonnancement des machines virtuelles qui peut être complexe.

En ce qui concerne la mémoire de la machine virtuelle, nous sommes aussi dans le cas le plus simple. En effet, comme expliqué dans la sous-section précédente, le *firmware* UEFI charge l'hyperviseur et marque son espace mémoire comme réservé dans la cartographie physique de la mémoire donnée au prochain logiciel chargé. De ce fait, nous n'avons pas à gérer les accès mémoire non désirés à l'espace mémoire de l'hyperviseur puisque le système d'exploitation n'y accédera pas. Enfin, la vision de la mémoire imposée à la machine virtuelle est en *identity mapping*, c'est-à-dire que nous utilisons EPT uniquement pour effectuer les contrôles d'accès et protéger l'espace mémoire de l'hyperviseur [4].

2. Le chargeur de l'UEFI n'est capable de traiter que des fichiers binaires. Disposer d'une zone de données statiques de plusieurs Mégas octets implique donc d'avoir un fichier binaire de plusieurs Mégas octets, ce qui est lourd et long à charger.

3. À l'instar du noyau Linux, le *firmware* UEFI exporte les interfaces permettant de communiquer avec ces modules

4. Un accès à notre espace mémoire serait réalisé en passant outre les consignes du *firmware* UEFI concernant

4.3.2.2 Contrôle d'exécution

Cette sous-section donne quelques éléments sur le contrôle d'exécution des machines virtuelles qu'effectue Abyme.

Abyme est un hyperviseur qui emploie la technique dite de *full virtualization*, mettant en place un environnement d'exécution où le logiciel invité se comporte comme si le matériel n'était pas virtualisé voire n'en est même pas conscient. Pour garder le contrôle sur le matériel, l'hyperviseur doit contrôler, émuler ou exécuter correctement les instructions privilégiées de la machine virtuelle, sans pour autant altérer son fonctionnement. Pour ce faire, il va masquer aux yeux de la machine virtuelle son espace mémoire et le matériel dont il va s'accaparer. Pour éviter de placer le système dans un état incohérent, ce masquage doit être réalisé avant le chargement du système d'exploitation. Dans l'implémentation actuelle, seule une carte réseau est accaparée par Abyme.

La carte réseau accaparée par Abyme est utilisée pour communiquer avec un client de contrôle et de débogage. Pour cela, il doit protéger son espace de configuration en PIO et MMIO. Les registres des cartes réseau d'aujourd'hui étant la plupart du temps *mappés* en mémoire, il doit aussi les protéger pour empêcher un accès direct à ces zones mémoire par un logiciel malveillant. Le masquage de l'espace de configuration permet, comme son nom l'indique, de masquer la présence d'un matériel donné (p. ex., notre carte réseau), alors que la protection des registres empêche sa reconfiguration.

La protection PIO s'effectue grâce au contrôle des deux ports spécifiques `0xcf8` et `0xcfc` de l'espace des entrées/sorties. L'utilisation de ces ports s'effectue en deux étapes. Tout d'abord, le logiciel doit écrire l'adresse du registre PCI à accéder dans le port `0xcf8` via l'instruction `out`. L'adresse PIO est calculée en fonction de l'adresse PCI `bus:device.function` du périphérique. Il va dans un second temps exécuter l'instruction `in` ou `out` pour lire ou écrire le contenu du registre passé en paramètre lors de la première phase.

VT-x propose un contrôle d'accès à l'espace des entrées sorties via les *I/O bitmaps*. Grâce à elles, on peut décider de prendre la main systématiquement sur l'exécution de la machine virtuelle lors de tentatives d'accès aux ports spécifiés dans celles-ci, pour exercer un contrôle adéquat. L'appropriation de la carte réseau se réduit donc au contrôle en lecture et écriture sur le port `0xcfc` qui succède à l'écriture de son adresse PIO sur le port `0xcf8`. Les écritures sont ignorées. Lors de lectures, on retourne un mot de la bonne taille, composé d'octets `0xff`, pour indiquer l'absence de périphérique à l'adresse de la carte réseau.

Pour la protection MMIO, l'hyperviseur configure EPT pour indiquer au logiciel invité que la carte réseau n'est pas présente. Connaissant l'adresse de base des accès MMIO donnée par le registre PCI MMCONFIG, ainsi que l'adresse PCI de notre carte, nous pouvons calculer simplement l'adresse MMIO de son espace mémoire afin de le remapper vers une page contenant des octets à `0xff`.

EPT est aussi utilisée pour protéger l'espace mémoire de l'hyperviseur. Abyme gère une unique machine virtuelle. Il peut donc simplement configurer l'espace mémoire physique en *identity mapping* (sauf MMIO pour la carte réseau), avec tous les droits. Les pages correspondant à son espace mémoire sont par contre interdites d'accès en lecture, écriture et exécution grâce aux attributs des *Page Table Entries* (PTE).

L'hyperviseur est maintenant capable de protéger son espace mémoire et ses périphériques.

---

les zones mémoire occupées (cf. section 4.3.1.1). Un tel comportement pourrait indiquer la compromission du noyau de la machine virtuelle.

Il doit maintenant protéger l'accès à certains registres et à l'exécution par la machine virtuelle de certaines instructions pouvant remettre en cause son bon fonctionnement. Notamment, les accès aux registres de contrôle `cr0` et `cr4` doivent être contrôlés, car certains bits sont nécessaires pour une exécution correcte lorsque les extensions de virtualisation sont activées (*VMX operation*). Pour cela, l'hyperviseur est notifié des accès en écriture aux bits "protégés" grâce aux masques *guest host cr0* et *guest host cr4* et va écrire les modifications dans des versions "fantômes" ou *shadow* stockées dans la VMCS. Ce sont ces versions *shadow* qui seront automatiquement lues plus tard par la machine virtuelle.

Enfin, d'autres instructions génératrices inconditionnelles de VM Exits seront simplement exécutées sur le processeur, car non dangereuses pour le bon fonctionnement de l'hyperviseur de sécurité (c'est-à-dire `cpuid`, `xsetbv`). D'autres instructions sont également génératrices inconditionnelles de VM Exits, celles du jeu d'instruction VMX. Leur gestion par l'hyperviseur est décrite dans la partie suivante.

Les accès DMA provenant des périphériques PCI Express sont contrôlés avec la fonctionnalité de *DMA remapping* des IOMMU d'Intel VT-d [65], qui protègent l'espace mémoire de l'hyperviseur contre les modifications distantes. Notons que les accès du périphérique de confiance sont, eux, autorisés.

4.3.3 Gestion de la récursivité

Le support de la virtualisation récursive par Abyme implique l'émulation des instructions de virtualisation et l'adoption d'une stratégie pour l'ordonnancement des différents niveaux de virtualisation.

4.3.3.1 Virtualisation du jeu d'instructions VMX

La virtualisation du jeu d'instruction VMX est un point technique clé de Abyme. Dans le cadre de la *full virtualization*, un hyperviseur virtualisé, ne sachant pas qu'il s'exécute sur un cœur virtuel, va tenter à son tour d'activer les extensions de virtualisation et utiliser ses 13 instructions. Cette partie décrit le travail qu'un hyperviseur doit réaliser pour virtualiser ces instructions. Ce travail nécessite de reproduire aussi fidèlement que possible le comportement décrit dans le manuel du développeur Intel [62].

Virtualiser les instructions VMX implique de reproduire leur comportement nominal, mais aussi celui de leur gestion des erreurs. La gestion des erreurs pour ce jeu d'instruction est très simple et se comporte de la manière suivante. Quatre états d'erreur ou de succès sont générés par ces instructions : *VMsucceed*, *VMfail* et ses deux sous états *VMfailValid* et *VMfailInvalid*. À chaque état correspond une combinaison des bits du registre RFLAGS. De plus, dans le cas de *VMfailValid*, un code d'erreur est écrit dans le champ *VM-instruction error* de la VMCS courante. La manipulation du registre RFLAGS et du champ *VM-instruction error* suffit à injecter un évènement lié à la virtualisation dans un hyperviseur virtualisé. Dans cette partie nous ne traitons pas le cas des vérifications classiques pouvant être effectuées sur des instructions privilégiées, comme le test du niveau de privilège, du mode du processeur, les exceptions de la gestion des privilèges, etc.

La tentative d'exécution d'une des instructions VMX par le logiciel invité génère systématiquement un VM Exit. Pour chaque VM Exit généré, l'hyperviseur de sécurité émule l'instruction et redonne la main à l'hyperviseur virtualisé avec l'instruction `vmresume`. Pour

chaque instruction prenant en paramètre un pointeur de VMCS, l'alignement de l'adresse de la VMCS et l'identifiant de révision sont testés pour retourner un *VMfailInvalid* en cas d'erreur.

`vmxon(VMCS *)` active le mode *VMX operation*. Elle est la seule instruction VMX exécutable par le processeur en dehors de ce mode. Seul le pointeur VMCS de cette instruction est vérifié.

`vmclear(VMCS *)` passe une VMCS à l'état *clear* pour permettre le lancement de la VM avec `vmlaunch`. Pour l'émuler, Abyme charge avec `vmprtld` la VMCS passée en paramètre, effectue un appel à `vmclear` sur cette VMCS et enfin, recharge la VMCS de l'hyperviseur virtualisé.

`vmptrld(VMCS *)` change la VMCS courante, qui sera implicitement utilisée par les instructions `vmread`, `vmwrite`, `vmlaunch`, `vmresume`, etc. Pour émuler cette instruction, il suffit de copier le pointeur de VMCS. Nous appellerons ce pointeur le *shadow VMCS pointer* pointant vers une *shadow VMCS*, conformément à la documentation Intel. L'hyperviseur n'effectue pas immédiatement de `vmptrld` sur ce pointeur, car il va rendre la main à la VM ayant exécuté le `vmptrld` émulé. La *shadow VMCS* sera chargée plus tard lors de l'émulation d'instructions comme `vmread` et `vmwrite`.

`vmlaunch` démarre une VM configurée par une VMCS dont l'état est *clear*. Dans le cas de l'émulation, cette machine virtuelle correspond à la VMCS que l'hyperviseur virtualisé a précédemment chargée avec l'instruction `vmptrld`. Elle est alors associée au pointeur *shadow VMCS pointer*. Il suffit donc à Abyme de copier cette VMCS dans une VMCS lui appartenant, en exécutant un *vmptrld* sur la *shadow VMCS*, une série de `vmread`, un `vmptrld` pour charger sa VMCS, et enfin une autre série de `vmwrite`, en faisant attention aux champs compromettant son intégrité. Il termine en exécutant `vmlaunch` ou `vmresume`.

`vmresume` poursuit l'exécution d'une VM configurée par une VMCS dont l'état est *launched*. L'hyperviseur virtualisé tente de reprendre l'exécution d'une machine virtuelle qu'il a configurée dans la *shadow VMCS*. Abyme effectue la même copie d'état que pour le `vmlaunch` puis exécute `vmresume`.

`vmwrite(field, value)` accède en écriture à la VMCS courante. Pour ce faire, l'hyperviseur charge la *shadow VMCS*, exécute le `vmwrite(field, value)` et recharge sa VMCS.

`vmread(field)` accède en lecture à la VMCS courante. La même stratégie que pour `vmwrite` est appliquée, les valeurs de retour étant copiées dans sa VMCS.

4.3.3.2 Ordonnancement des niveaux de virtualisation – cas général

Supporter la virtualisation récursive signifie qu'un hyperviseur doit pouvoir virtualiser N hyperviseurs, en ne sachant pas si lui-même ne l'est pas et par combien de couches sous-jacentes. La seule chose qu'il peut déduire étant le nombre de niveaux au-dessus de lui.

Dans tous les cas, lors d'un VM Exit généré par une des machines virtuelles de cette pile, le contrôle sera toujours repris par l'hyperviseur chargé en premier et maître du matériel, l_0. Or, tous les évènements (VM Exits) générés au niveau x par l_x doivent être traités au niveau $x-1$ par l'hyperviseur l_{x-1}. La stratégie employée pour faire face à ce problème est une technique tirée de Turtles [81], indiquant que les évènements générés remontent du niveau zéro vers le niveau responsable des traitements de ces évènements (figure 4.9).

Pour l'implémentation d'Abyme, nous avons choisi de ne considérer que les instructions de

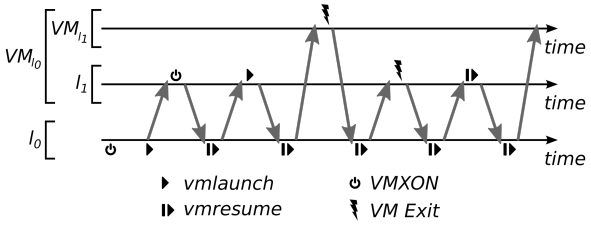

FIGURE 4.9 – Virtualisation récursive

manipulation de contextes et de VMCS. Les instructions de gestion des caches et de paravirtualisation ne sont pas essentielles pour représenter un comportement correct. Les copies de VMCS sont aussi simplifiées pour ne pas surcharger la sortie du modèle, le nombre de champs copiés étant très important.

Pour comprendre la mise en œuvre de cette stratégie de virtualisation, nous prenons le cas simplifié de notre hyperviseur se virtualisant lui-même une fois, en se concentrant sur la phase de démarrage.

4.3.3.3 Ordonnancement des niveaux de virtualisation – avec deux hyperviseurs

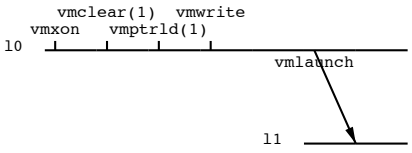

FIGURE 4.10 – Démarrage de l_0

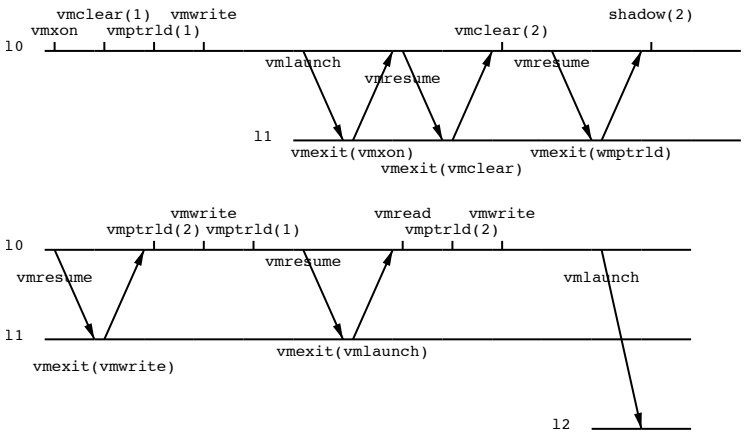

FIGURE 4.11 – Démarrage de l_0 et l_1

La séquence de démarrage d'un seul hyperviseur est présentée à la figure 4.10. La virtualisation est activée avec `vmxon`, passant le processeur en mode *VMX operation*. La première instance de Abyme devient donc l_0. Cette instance effectue ensuite un `vmclear(VMCS *)` pour passer sa VMCS à l'état *clear*, charger cette VMCS avec `vmptrld(VMCS *)` et écrire les attributs de la VM avec une série de `vmwrite` (simplifiée ici à une seule occurrence). Enfin, il lance la VM avec un `vmlaunch`.

Le chargement d'une seconde instance d'Abyme par l_0 permet de créer l_1. l_0 étant déjà installé en tant qu'hyperviseur, il va émuler toutes les instructions VMX exécutées par l_1. La figure 4.11 représente les transitions effectuées lors de ces démarrages successifs. Une analyse visuelle simple montre que le comportement de l'hyperviseur l_1 n'est pas altéré par la présence de l_0 (la séquence de la figure 4.10 est présente dans la figure 4.11) [5].

5. Le diagramme de Hasse pour l'exécution de l_1 donne la figure 4.10 à l'identique, modulo les indices.

Un point intéressant de cette séquence concerne la tentative d'exécution de `vmlaunch` par l_1. Cette tentative entraîne une prise de contrôle et une émulation par l_0. À l'issue de cette émulation, la VM l_2 est bien démarrée. Reprenons étape par étape cette émulation. Pour cela, l_1 tente d'exécuter l'instruction `vmxon`. Comme il est virtualisé, cela déclenche un `vmexit`, qui est représenté sur la figure 4.11 par `vmexit(vmxon)`. L'hyperviseur l_0 ne faisant aucune opération particulière, il rend la main à l_1 à l'aide d'un `vmresume`. l_1 tente ensuite d'exécuter un `vmclear`, ce qui déclenche un nouveau un `vmexit`, représenté par : `vmexit(vmclear)`. l_0 prend la main, effectue alors ce `vmclear`, représenté par : `vmclear(2)` et rend la main à l_1 par un `vmresume`. Ensuite, l_1 veut charger la VMCS correspondant à une machine virtuelle et tente d'exécuter `vmptrld`. Ceci déclenche un `vmexit` représenté par : `vmexit(vmptrld)`. l_0 prend la main et conserve alors l'adresse mémoire de cette VMCS pour pouvoir la lire et la modifier plus tard. Ceci est représenté par : `shadow(2)`. l_0 n'exécute pas tout de suite de `vmptrld` pour l_2 parce qu'il doit rendre la main à l_1 pour qu'il continue son exécution. Il le fera au besoin lors de réels accès à l_2 via des `vmread`, `vmwrite` ou lors de son ordonnancement avec `vmlaunch` ou `vmresume`. l_0 rend ensuite la main à l_1 qui tente d'écrire des champs dans la VMCS qu'il a créée. Ceci déclenche à nouveau un `vmexit` : `vmexit(vmwrite)`. l_0 charge alors la VMCS correspondante (`vmptlrld(2)`), fait les modifications attendues, puis recharge la VMCS associée à l_1 (`vmptlrd(1)`) et lui rend la main. Enfin, comme l_1 a terminé de configurer la VM l_2, il la démarre avec un `vmlaunch`, qui génère un `vmexit(vmlaunch)`. l_0 charge alors sa *shadow VMCS*, étant couramment la VMCS de l_1 qui pointe vers l_2, recopie l'état de l_2 dans une VMCS lui appartement et effectue le `vmlaunch` qui donne ainsi la main à l_2.

Les transitions et évènements générés se complexifient grandement au-delà de 3 hyperviseurs. Pour pouvoir facilement les représenter, il est nécessaire de modéliser le comportement d'Abyme. Un modèle et plus de détails sur l'implémentation d'Abyme sont donnés dans [82].

4.3.4 Contrôle et débogage

Un des défis liés à la mise en œuvre d'un hyperviseur consiste à trouver un moyen efficace et simple pour déboguer les machines virtuelles tout au long de leurs exécutions.

Lors du développement des premières versions de l'hyperviseur, nous avions tout simplement utilisé l'écran et le clavier comme interface de contrôle pour le développeur. Cette solution s'est très vite avérée insuffisante, car le système d'exploitation chargé de la machine virtuelle les reconfigure rapidement et un partage de ces ressources est alors indispensable. Ce partage est compliqué à résoudre et peut augmenter considérablement la taille du code. Pour cette raison, nous avons décidé de nous accaparer un des périphériques de communication de la machine cible pour y installer un serveur de contrôle et de débogage minimaliste.

La machine cible ne disposant pas de contrôleur rs232 câblé sur un port série, il ne restait plus que la carte son et le contrôleur Ethernet. Pour garder une plus grande connectivité, la carte Ethernet semblait la meilleure solution. Nous avons donc développé un driver UEFI de runtime très simple, capable d'émettre et de recevoir, sans utiliser les interruptions, des trames Ethernet. Ce driver est utilisé par un serveur de contrôle et de débogage, intégré à l'hyperviseur, qui implémente un protocole question-réponse simple incluant les fonctionnalités suivantes :

— contrôle d'exécution ;
— lecture et écriture dans la mémoire hôte ;
— accès aux registres du cœur courant ;

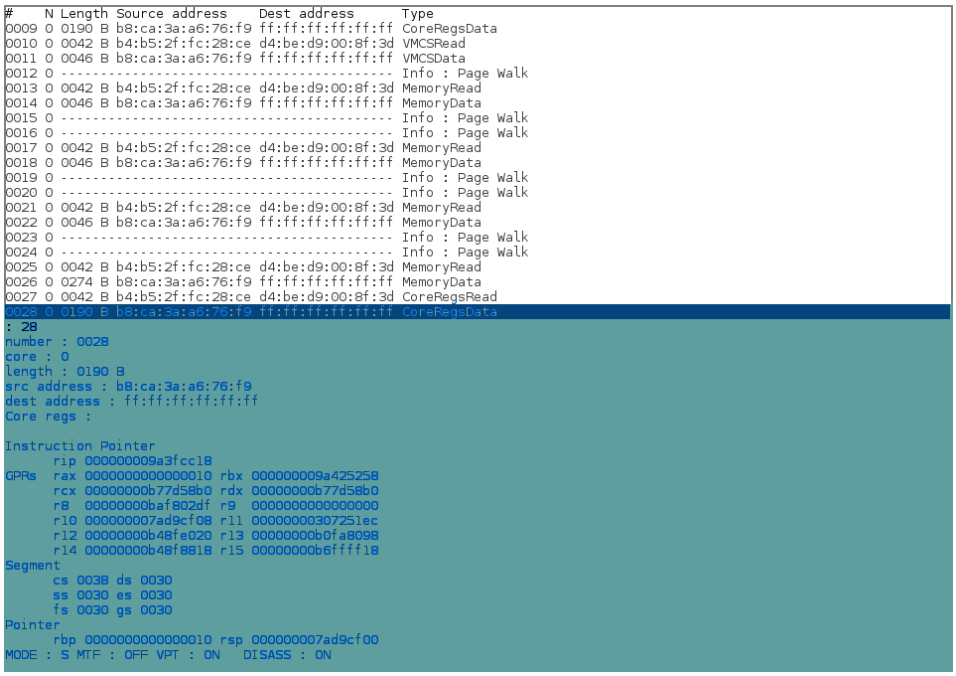

FIGURE 4.12 – Récupération de l'état du cœur courant

— messages d'information pour transporter le texte affiché par l'hyperviseur ;
— messages personnalisés utiles pour les expérimentations ;
— lecture et écriture dans la VMCS courante.

Abyme utilise une bibliothèque minimaliste, que nous avons développée, qui contient entre autres des fonctions d'affichage de chaînes de caractères telles que `printk(const char *fmt, ...)`. Ces chaînes sont affichées à l'écran, mêlées avec celles du *shell UEFI* en phase de pré boot, puis sont redirigées sur le réseau, transportées par les messages d'information pour être traitées par un client de débogage. Ainsi, nous pouvons afficher du texte tout au long de l'exécution de la machine. Le serveur de débogage ainsi que l'affichage à distance peuvent être désactivés à la compilation. De plus la distinction entre les niveaux pour l'affichage se fait au niveau de l'API du driver Ethernet, ne complexifiant pas le code du cœur du projet.

Le client de contrôle et débogage a été écrit en Python. Il met en œuvre le comportement décrit par les PDU de notre protocole (figure 4.12), mais ajoute aussi des fonctionnalités plus pratiques et de plus hauts niveaux comme le désassemblage du code de la VM pointé par son `rip` courant (figure 4.13). Cette fonction nécessite l'exécution de plusieurs étapes :

— récupération des registres de contrôle `cr0` et `cr3` ;
— récupération du MSR IA32_EFER pour son bit LMA indiquant si le processeur est en mode long ;
— détermination du mode du processeur en fonction du registre `cr0` et du msr IA32_EFER ;
— lectures mémoires successives pour récupérer les différents niveaux de Page Entries en fonction du type de pagination (32 bits, PAE, et IA-32e) ;
— lecture de la page contenant le code à l'adresse traduite ;
— invocation de la commande `objdump` en fonction du mode et affichage de sa sortie après post traitement des caractères spéciaux.

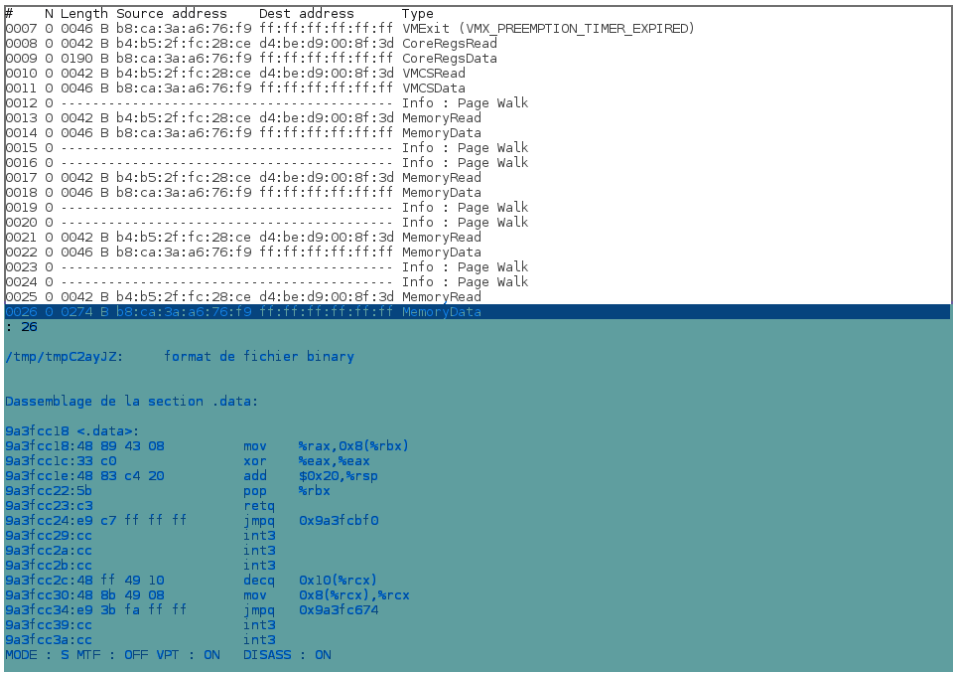

FIGURE 4.13 – Désassemblage du code pointé par le `rip` courant

4.3.5 Bibliothèque d'hyperviseur

Le but de la bibliothèque d'hyperviseur est de simplifier le développement de petits hyperviseurs pour concevoir simplement des preuves de concept. Abyme, une fois compilé, est distribué sous forme d'archive d'objets (bibliothèque statique) et de fichiers d'en-tête des fonctions. Son utilisation impose les deux étapes d'initialisation et d'exécution de la machine virtuelle.

L'API d'Abyme est très simple, mais stricte. Elle impose à un hyperviseur d'implémenter la fonction `void hook_main(void)`. Cette fonction est invoquée par Abyme juste avant le lancement de la machine virtuelle. Elle permet alors à l'hyperviseur de s'initialiser correctement (allocation mémoire) et de s'abonner à des évènements via un système de *hook*.

Chaque hook correspond à un évènement de la machine virtuelle (p. ex., *VM Exits*). Un hyperviseur peut s'abonner à trois types de *hooks* correspondant à trois moments précis de la gestion du *VM Exit* par Abyme, liés à la virtualisation récursive :

— *Boot* : L'hyperviseur prendra la main en premier avant même qu'Abyme n'effectue le moindre traitement. Avec ce type de *hook*, il est possible de surcharger complètement le comportement de la bibliothèque Abyme.

— *Pre* : Ce type de *hook* est déclenché dans le processus de traitement du *VM Exit*, juste avant qu'Abyme ne cède la main à un hyperviseur de plus haut niveau correspondant si la virtualisation récursive est activée. Ce type de *hook* sert principalement à masquer certains évènements privilégiés.

— *Post* : Ce type de *hook* est déclenché après le traitement d'un *VM Exit* par l'hyperviseur de plus haut niveau qui a exécuté l'instruction `vmresume` pour rendre la main à sa machine virtuelle. Grâce à ce type de *hook*, il est possible de surcharger le comportement d'un hyperviseur invité.

Maintenant que nous avons présenté la bibliothèque d'hyperviseur récursif Abyme, la prochaine section présente son utilisation pour la mise en œuvre de l'hyperviseur de sécurité.

4.3.6 Mise en œuvre de l'hyperviseur de sécurité

Cette sous-section présente le développement de l'hyperviseur de sécurité sur la base de la bibliothèque Abyme, via l'utilisation des *hooks*. Dans un premier temps nous présentons le support de l'exécution des challenges et ensuite l'interface logicielle pour le test d'intégrité des logiciels observés.

4.3.6.1 Environnement d'exécution des challenges

Les challenges doivent être exécutés en mode hyperviseur *VMX root operation* (cf. section 3.7.2). De plus, le moment de l'exécution est imposé par la réception d'une interruption provenant du périphérique de confiance (cf. section 3.5.1.2). Nous avons fait le choix d'utiliser les interruptions *Non Maskable Interrupt* (NMI) qui sont envoyées vers le cœur 0. Pour que l'hyperviseur prenne la main sur l'exécution de la machine virtuelle lorsque le cœur reçoit l'interruption, nous devons configurer les VMCS pour que l'évènement *VM Exit NMI Exiting* soit généré. Enfin, nous abonnons l'hyperviseur à cet évènement pour le type de hook *boot* de manière à perdre le moins de temps possible avant d'exécuter le challenge. Si l'hyperviseur était déjà en train de traiter un *VM Exit* à ce moment, un autre *VM Exit* ne sera pas généré, mais une interruption classique sera générée à la place. Ce comportement impose de configurer l'IDT pour traiter ces interruptions classiques.

Pour communiquer avec le périphérique de confiance, nous avons développé un *driver* UEFI similaire au pilote de contrôleur réseau Ethernet 1. Ce *driver* est chargé avant l'hyperviseur avec le *script shell* UEFI. Ce driver nous donne l'adresse mémoire de l'*expansion ROM PCI Express* qui contient les challenges, ainsi que les adresses BAR0 et BAR1.

Lorsque la machine de *monitoring* prépare un challenge, elle n'a pas forcément idée de l'emplacement des fonctions d'entrée/sortie. Toutefois, pour permettre l'usage de ces fonctions durant l'exécution des challenges, l'hyperviseur de sécurité doit constituer une structure de données, appelée *system table*, qui contient les pointeurs vers des fonctions d'affichage pour le débogage et vers les registres matériels du périphérique de confiance pour y écrire la solution. Quand l'hyperviseur de sécurité est interrompu pour exécuter le challenge courant, il doit copier le pointeur vers cette structure dans l'espace mémoire du challenge, qui pourra alors l'utiliser. La sous-section 4.4.1 décrit le format de l'exécutable des challenges et son évolution pendant le cycle de tests d'intégrité.

Une fois toutes ces configurations mises en place durant la phase d'initialisation de l'hyperviseur, l'hyperviseur de sécurité est capable de recevoir les challenges et de les exécuter.

4.3.6.2 Environnement d'exécution des tests d'intégrité

L'environnement d'exécution des tests d'intégrité sur les logiciels observés est très basique. La fonction f de test d'intégrité est découpée en plusieurs composantes logicielles qui sont intégrées dans l'hyperviseur de sécurité à la compilation. Plus précisément, l'interface logicielle proposée par l'hyperviseur de sécurité est découpée en trois évènements qui correspondent à trois fonctions qu'un test d'intégrité doit impérativement définir :

`init` Appelée à l'initialisation du test, durant la phase d'initialisation de l'hyperviseur. C'est à ce moment que le test d'intégrité peut allouer de la mémoire et initialiser des structures.

`call` Fonction appelée après un appel hyperviseur depuis la machine virtuelle. Appelée lors de l'exécution de l'instruction `vmcall` par un test d'intégrité, cette fonction sert d'interface logicielle avec le logiciel observé s'exécutant en niveau de privilège utilisateur, dans la machine virtuelle.

`execute` Cette fonction est appelée au moment où le test d'intégrité doit être réalisé sur le logiciel observé. Les alertes sont transmises si besoin dans ce contexte.

L'enregistrement d'un test d'intégrité est fait par l'hyperviseur de sécurité, dans une liste de tests, au moment de son initialisation. Lors de l'enregistrement d'un test, il est nécessaire de préciser une chaîne de caractères précisant l'identifiant du test qui sera utile au logiciel observé pour dialoguer avec le test via l'interface paravirtualisée.

En effet, le logiciel observé n'est pour l'instant pas chargé par l'hyperviseur de sécurité. Nous laissons pour l'instant ce travail au système d'exploitation ou à l'hyperviseur en place dans l'architecture originale de la machine cible. Dans une version définitive de l'hyperviseur de sécurité, il est indispensable de renforcer cette interface, voire d'intégrer un chargeur d'application, afin d'inhiber les attaques par ce biais. Notre stratégie courante impose au logiciel observé de lui-même indiquer, lors de son chargement, les paramètres indispensables au test d'intégrité qui lui est associé, comme son adresse mémoire de chargement par exemple. Plus précisément, l'interface paravirtualisée du logiciel observé avec l'hyperviseur de sécurité est donnée dans la table 4.4. Les appels à l'hyperviseur de sécurité seront effectués avec l'instruction `vmcall` de VMX. Cette instruction est très intéressante, car elle est utilisable par la machine virtuelle, quel que soit le niveau de privilège (*ring*) couramment utilisé. Le premier appel hyperviseur permet au logiciel observé de lister les noms des tests d'intégrité, afin de récupérer, par comparaison, l'identifiant d'un test particulier. Cet identifiant est utilisé dans un second temps pour utiliser le deuxième appel hyperviseur, qui permet d'effectuer un appel explicite à un test d'intégrité donné. Le deuxième appel hyperviseur, *call*, est en définitive une passerelle vers la fonction `call` d'un test d'intégrité. Le comportement de cette fonction étant spécifique à chaque test d'intégrité. Nous prenons un exemple d'implémentation de test dans la sous-section 4.4.2.

4.3.6.3 Environnement d'exécution des tests d'environnement

Les tests d'environnement ne sont pas exécutés dans l'hyperviseur de sécurité, mais dans le périphérique de confiance. Cependant notre hyperviseur est responsable des accès mémoire des périphériques et doit s'assurer qu'ils sont toujours disponibles pour le périphérique de confiance sous peine de génération d'alerte par celui-ci.

Méthode	[appel|retour]	`rax`	`rbx`	`rcx`	`rdx`	`rsi`
list	appel	1	0	index	-	adr. nom
	retour	1	0	index	présent ?	adr. nom
call	appel	1	1	index	-	-
	retour	1	1	index	-	-

TABLE 4.4 – Interface 64 bits d'interaction avec l'hyperviseur de sécurité

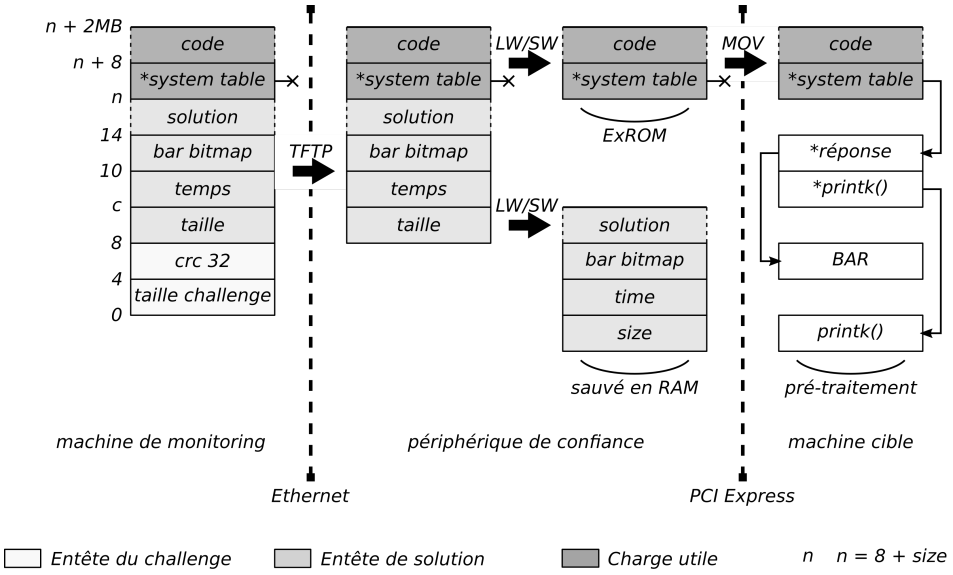

FIGURE 4.14 – Évolution de l'image binaire d'un challenge

L'hyperviseur de sécurité doit s'assurer que la configuration de l'IOMMU mise en place par le système d'exploitation ou l'hyperviseur virtualisé autorise bien les accès pour notre périphérique de confiance. Pour ce faire, nous interceptons l'activation de l'IOMMU par le driver virtualisé et avant que l'IOMMU soit activée, nous ajoutons notre configuration. Par ailleurs, il est aussi possible de bloquer les accès DMA avec des fonctionnalités du contrôleur mémoire, les segments mémoires configurables *DMA Protected Range* (DPR) et les deux segments *Programmable Memory Range* (PMR), qui permettent de se protéger de tout accès DMA. Nous devons nous assurer que notre hyperviseur de sécurité se situe en dehors de ces régions mémoires pour pouvoir être capable d'exécuter les tests d'environnement.

4.4 Le cycle de tests d'intégrité

Dans cette section, nous commençons par donner des détails sur l'implémentation courante des challenges, puis une vue globale sur l'interaction des composants et de la transformation du format binaire d'un challenge tout au long d'un cycle d'exécution. Enfin, nous terminons avec un exemple de test d'intégrité sur un logiciel observé. Nous ne détaillons pas ici l'implémentation des tests d'environnement, car elle a déjà été présentée dans la section 4.2 au travers du périphérique de confiance. Plus de détails sont tout de même donnés sur les différents tests d'environnement dans le chapitre suivant avec la présentation de la version virtualisée de notre architecture de sécurité.

4.4.1 Mise en œuvre des challenges

L'implémentation actuelle des challenges est fonctionnelle bien que l'algorithme de génération ne soit pas encore aléatoire. Cette section détaille le format des fichiers contenant les challenges, la récupération et l'exécution de ces challenges par l'hyperviseur de sécurité et termine par un exemple de challenge.

4.4.1.1 Format d'image binaire

La compilation et l'édition des liens des challenges sont effectuées avec le compilateur GNU GCC et un *script* d'édition des liens dédié. Le code binaire généré est exécutable directement et indépendant de sa position de chargement à la manière des bibliothèques partagées sous Linux. L'image binaire du challenge est décrite dans la figure 4.14. La première section appelée « entête du challenge » est dédiée au contrôle d'intégrité du téléchargement du challenge depuis la machine de confiance. Elle contient deux champs, la taille de l'image et le code de redondance cyclique sur 32 bits (CRC32) pour la vérification de l'intégrité du challenge après son téléchargement via TFTP par le périphérique de confiance. La deuxième partie de l'image binaire est appelée « entête de solution ». Elle contient la solution en valeur et en temps du challenge, mais aussi une *bitmap* qui permet d'indiquer au périphérique de confiance dans quels registres de la zone mémoire BAR0 la solution sera écrite. La dernière partie de l'entête du challenge est appelée « charge utile ». Elle est divisée en deux parties. La première constitue le pointeur vers la structure *system table* qui est mise en place par l'hyperviseur de sécurité avant d'exécuter le challenge (cf. sous-section 4.3.6.1) et la deuxième correspond basiquement aux sections de données et de code concaténées des objets compilés.

Comme le montre la figure 4.14, cette image binaire va évoluer et subir de multiples transformations tout au long de l'exécution du cycle d'un challenge. Comme nous l'avons vu, le challenge est tout d'abord téléchargé depuis la machine de *monitoring* vers le périphérique de confiance. À ce moment, l'entête du challenge est vérifié puis retiré par le périphérique de confiance. Ensuite, il va séparer l'image en deux pour sauvegarder l'entête de solution pour vérifier la future réponse de l'hyperviseur de sécurité. La charge utile est, quant à elle, copiée dans la mémoire PCI Express *Expansion ROM* de notre périphérique de confiance. Enfin, l'hyperviseur de sécurité, une fois interrompu par le périphérique de confiance, va lui aussi copier le challenge dans une page mémoire qu'il a allouée et affecter le pointeur vers la *system table*.

4.4.1.2 Interactions matérielles pour les challenges

Tous les éléments de l'architecture de sécurité ayant été décrits, nous pouvons détailler les interactions des différents éléments du système lors du déclenchement de l'exécution d'un challenge. La figure 4.15 illustre l'architecture pour les challenges dans sa globalité.

Tout d'abord, le challenge est téléchargé par le périphérique de confiance dans sa mémoire DDR3 SDRAM. Le challenge est traité comme expliqué précédemment puis est copié dans la mémoire *expansion ROM* Wishbone du composant PCIEE *(1)*. L'hyperviseur de sécurité est interrompu avec une MSI (écriture PCI Express) via le PCIEE *(2)*. L'hyperviseur reçoit l'interruption sur le cœur zéro et télécharge, en RAM, le challenge depuis la mémoire *expansion ROM* du composant PCIEE, au travers du bus PCI Express *(3)*. L'hyperviseur traite le binaire du challenge et l'exécute *(4)*. Le code du challenge écrit la solution dans la mémoire du module BAR du composant PCIEE du périphérique de confiance au travers du bus PCI Express *(5)*. Enfin, une alerte est levée si besoin par le périphérique de confiance via un message envoyé avec le composant Ethernet MAC au travers du réseau Ethernet 2 *(6)*.

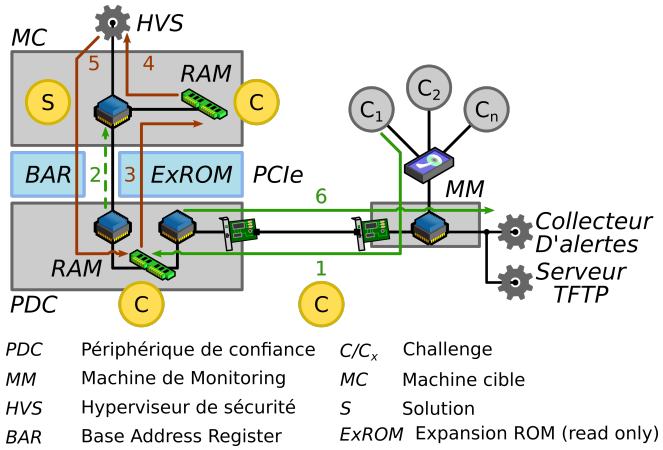

FIGURE 4.15 – Cycle de challenge

4.4.1.3 Exemple de challenge

Comme indiqué en introduction de cette section, nous n'avons pas développé de génération aléatoire des algorithmes des challenges dans cette première version de prototype. Les challenges sont tout de même mis en places dans une version statique.

```
#include "env/challenge.h"
#include "stdio.h"

// CPUID  0 - f and 80000000 - 8000008
static inline cpuid(void) {
  uint64_t rax, rbx, rcx, rdx;
  uint32_t i, j;
  uint64_t xor[4];
  for (j = 0; j < 0x100; j++) {
    xor[0] = 0, xor[1] = 0, xor[2] = 0, xor[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));
      xor[0] ^= rax, xor[1] ^= rbx, xor[2] ^= rcx, xor[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));
      xor[0] ^= rax, xor[1] ^= rbx, xor[2] ^= rcx, xor[3] ^= rdx;
    }
  }
  // MMIO DW Access is mandatory to write the solution
  for (i = 0; i < 4; i++) {
    systab->answer[2 * i] = (xor[i] >> 0x00) & 0xffffffff;
    systab->answer[2 * i + 1] = (xor[i] >> 0x20) & 0xffffffff;
  }
}

void challenge_start(void) {
  cpuid();
}
```

Listing 4.2 – Challenge `cpuid`

Pour illustrer les possibilités offertes pour les challenges, les listings 4.2 et 4.3 représentent respectivement un algorithme basé sur l'exécution de l'instruction `cpuid` qui génère les valeurs caractéristiques d'entrées et la solution associée. L'algorithme calcule 100 fois le ou exclusif de tous les registres `cpuid`. La solution du challenge est donnée dans le second listing.

```
#include "env/challenge.h"
#include "stdint.h"

typedef struct _cpuid_solution {
  solution_header h;
  uint64_t s[4]; // rax, rbx, rcx, rdx
} __attribute__((packed)) cpuid_solution;

cpuid_solution s __attribute__((section(".solution"))) = {
  // Solution header
  .h = {
    .size = sizeof(cpuid_solution), // Size of the solution
    .time_min = 0x240,              // expected execution time in micro fbb
    .time_max = 0x250,              // expected execution time in micro fbb
    .bar_bitmap = (uint32_t)-1,     // The expected BAR
  },
  // Solution : CPUID space xor
  .s = {
    0x000000006d3020dc, // rax
    0x00000000744ef43c, // rbc
    0x00000000139f8de4, // rcx
    0x00000000f64bb2b5  // rdx
  }
};
```

Listing 4.3 – Solution du challenge `cpuid`

4.4.2 Mise en œuvre des tests d'intégrité

Nous illustrons l'environnement de tests d'intégrité de logiciels observés avec le pilote Intel e1000e, distribué avec le noyau Linux. La fonction de test d'intégrité effectuée sur ce logiciel observé est simple et correspond au calcul de l'empreinte du code chargé en mémoire.

Nous avons ajouté un nouveau test d'intégrité dans la liste des tests via l'API des tests d'intégrité de l'hyperviseur de sécurité et implémenté les fonctions correspondant aux divers évènements à traiter par test d'intégrité. Notamment, la table 4.5 donne l'interface paravirtualisée avec le logiciel observé. La méthode `call` permet de donner l'adresse virtuelle de début et de fin du driver e1000e. Cette plage mémoire sera traduite en adresses physiques grâce à l'interprétation du registre `cr3` qui est disponible au moment de l'appel. Les deux autres mé-

Méthode	[appel|retour]	`rax`	`rbx`	`rcx`	`rdx`	`rsi`	`rdi`
call	appel	1	1	index	0	adr. début	adr. fin
	retour	1	1	index	0	adr. début	adr. fin
flip	appel	1	1	index	1	-	-
	retour	1	1	index	1	-	-
unflip	appel	1	1	index	2	-	-
	retour	1	1	index	2	-	-

TABLE 4.5 – Interface 64 bits d'interaction avec le test d'intégrité de e1000e

thodes nous ont permis de vérifier si le test d'intégrité fonctionne en modifiant et restaurant un bit dans l'espace mémoire du driver.

Ensuite, nous avons compilé le *driver* Linux modifié dans le but de récupérer l'identifiant du test d'intégrité, nommé "e1000e", avec la méthode paravirtualisée `list` et de partager les adresses de chargement du driver au test d'intégrité par la suite. Le listing 4.4 donne la fonction d'initialisation du driver modifiée dans ce sens. Les symboles `_e1000e_start` et `_e1000e_end` pointent vers le début et la fin du code du module noyau. La boucle `while` va appeler la méthode `list` jusqu'à ce qu'elle trouve le test d'intégrité au nom de "e1000e". Le driver va ensuite communiquer ses adresses avec le deuxième `vmcall` vers l'identifiant stocké dans la variable `index`. Le chargement normal du driver e1000e continue ensuite.

```
// e1000e guarded software experimentation
void _e1000e_start(void);
void _e1000e_end(void);

/**
 * e1000_init_module - Driver Registration Routine
 *
 * e1000_init_module is the first routine called when the driver is
 * loaded. All it does is register with the PCI subsystem.
 **/
static int __init e1000_init_module(void)
{
    int ret;
  uint64_t index = 0, present = 1;
  char name[256];

  uintptr_t s = (uintptr_t)&_e1000e_start,
            e = (uintptr_t)&_e1000e_end;

  // Get env_e1000e command id
  while (present) {
    // Call list method
    __asm__ __volatile( "vmcall;" : "=d" (present) : "a"((uint64_t)0x1),
        "b"((uint64_t)0x0), "c"((uint64_t)index),
        "S"((uint64_t)(uintptr_t)&name[0]));
    if (strcmp(name, "e1000e") == 0) {
      break;
    }
  }

  if (present == 0) {
    return -1;
  }

  // Call env_e1000e command
  __asm__ __volatile( "vmcall;" : : "a"((uint64_t)0x1),
      "b"((uint64_t)0x1), "c"((uint64_t)index),
      "d"((uint64_t)0x0), "S"(s), "D"(e));

    pr_info("Intel(R) PRO/1000 Network Driver - %s\n",
        e1000e_driver_version);
    pr_info("Copyright(c) 1999 - 2014 Intel Corporation.\n");
    ret = pci_register_driver(&e1000_driver);

    return ret;
}
module_init(e1000_init_module);
```

Listing 4.4 – Initialisation du driver e1000e

4.5 Conclusion

Dans ce chapitre, nous avons présenté l'implémentation des deux entités principales de notre architecture de sécurité. Nous avons aussi détaillé comment composer avec elles pour mettre en œuvre le cycle d'intégrité proposé par notre protocole de tests. Le prototype que nous avons développé est fonctionnel et permet l'exécution complète du protocole de test. Le prochain chapitre expose les expérimentations que nous avons menées avec ce prototype.

Chapitre 5

Efficacité et performances

Sommaire

Ce chapitre présente les diverses expérimentations que nous avons effectuées avec notre architecture de sécurité. Dans une première partie, nous présentons une version virtualisée de l'architecture de sécurité et les raisons de ce développement. Ensuite, nous évaluons l'efficacité de notre cycle de tests d'intégrité, via des cas pratiques. La troisième section discute des performances de notre architecture en mesurant l'impact individuel et cumulé des différents tests lors de l'exécution de tâches classiques sur la machine cible. Enfin, nous prenons du recul sur ces travaux en discutant de l'impact sur la sécurité que peut engendrer l'intégration de notre architecture sur une plateforme.

5.1 Architecture de sécurité virtualisée

Pour mener à bien les expériences de détection que nous présentons dans la section suivante, il est nécessaire d'implémenter les challenges et tests d'environnement conçus dans la section 3.7. Le développement de tests logiciels est fastidieux sur la plateforme matérielle physique. En effet, nous développons ces tests à un niveau d'abstraction très bas, où la moindre erreur

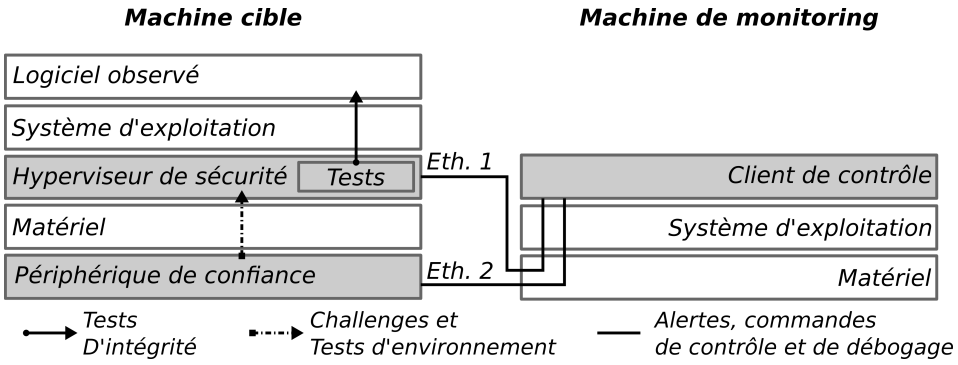

FIGURE 5.1 – Architecture de sécurité physique

logicielle ou matérielle s'avère dramatique pour l'ensemble du système, résultant la majorité du temps en une défaillance critique telle que l'arrêt de la machine. Or, un redémarrage fréquent constitue un frein au développement des challenges et tests d'environnement. Pour cette raison, nous avons décidé de virtualiser notre architecture de sécurité, de façon à pouvoir développer rapidement des tests avant un portage sur l'architecture physique.

5.1.1 Architecture

La figure 5.1 schématise l'architecture de sécurité physique. Nous avons décidé d'exécuter l'hyperviseur de sécurité dans une machine virtuelle qui sera la version virtuelle de la machine cible. Le matériel du périphérique de confiance ne sera pas émulé logiciellement, car une émulation de la logique séquentielle et combinatoire d'un circuit entier est très coûteuse en performances. Par ailleurs, il n'existe pas d'émulateur pour le processeur LM32, sans même tenir compte des composants que nous avons développés. Nous avons donc décidé de reproduire le comportement du périphérique de confiance dans une version logicielle, qui sera connectée comme périphérique virtuel à la machine de confiance. Le contrôleur réseau Ethernet 1 sera remplacé par une carte réseau virtuelle, contrairement au contrôleur réseau Ethernet 2 qui n'aura plus lieu d'être, car le périphérique de confiance est remplacé par une application. Les alertes levées par le périphérique de confiance seront donc transmises via des écritures dans

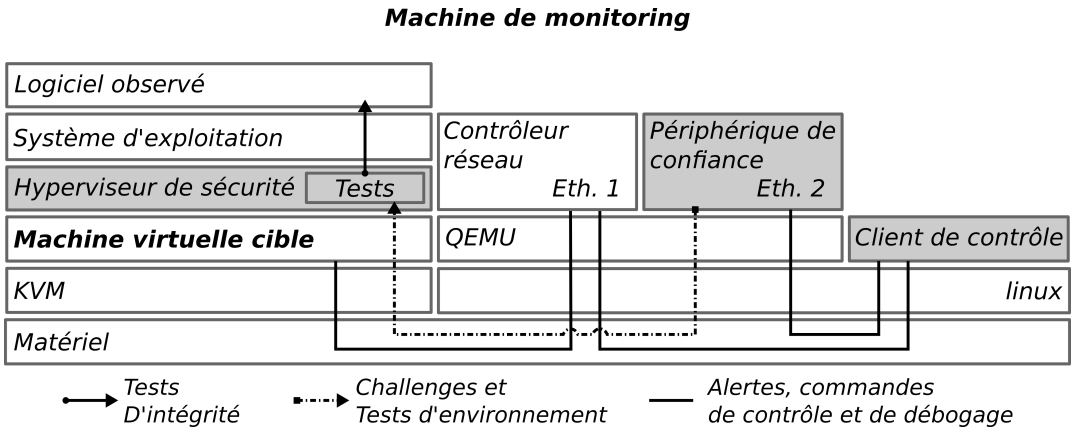

FIGURE 5.2 – Architecture de sécurité virtualisée

des fichiers ou communications inter processus. Toute cette pile logicielle sera hébergée par un hyperviseur exécuté sur la machine de *monitoring* originale.

5.1.2 Implémentation

La figure 5.2 schématise l'architecture de sécurité virtuelle. Pour sa mise en œuvre, nous avons choisi le couple hyperviseur KVM, hébergé dans Linux, et l'émulateur QEMU. Cette combinaison nous permet de préserver au maximum l'architecture de la machine de *monitoring*. Le périphérique de confiance est développé en tant que périphérique virtuel pour QEMU (il est compilé au même moment que QEMU). En ce qui concerne le *driver* Ethernet du contrôleur 1 et l'hyperviseur de sécurité, ceux-ci sont conservés intacts et installés dans la machine virtuelle qui s'exécute dans KVM. Plus de détails sur l'implémentation peuvent être trouvés dans [83].

5.1.3 Avantages

Le principal avantage de la version virtualisée de l'architecture de sécurité se situe au niveau du prototypage des tests. En effet, tester un challenge ou un test d'environnement sur la plateforme virtuelle prend quelques secondes. La durée de compilation d'un challenge ou du logiciel (hyperviseur de sécurité, les *drivers*) reste la même. La différence se situe au niveau du temps de redémarrage de la machine virtuelle et de la compilation du périphérique de confiance virtuel. Originalement, sur la machine physique, cela peut prendre plusieurs minutes, le temps de redémarrer le système, de retélécharger les divers logiciels et de démarrer toute la pile logicielle. De plus, si le circuit du périphérique de confiance doit être regénéré, la mise en place du nouveau test peut nécessiter jusqu'à 20 minutes. Par contre, la plateforme virtuelle est redémarrée et prête à exécuter un cycle d'intégrité en quelques secondes. N'étant pas limité par la quantité de machines et de périphériques de confiance physiques, il est également possible de lancer plusieurs tests en parallèle sur leurs versions virtuelles.

La plateforme virtuelle nous à donc permis de valider rapidement les preuves de concepts que nous avons ensuite portées sur la machine physique. Un autre avantage non négligeable d'une version virtuelle concerne l'injection de fautes pour simuler des attaques et tester notre approche. Cette injection est complexe au niveau physique, mais elle devient triviale dans la version virtuelle. En effet, simuler une attaque DMA en mémoire RAM principale peut être effectuée, par exemple, via un périphérique malveillant virtuel connecté à la machine virtuelle cible, en plus du périphérique virtuel de confiance.

5.2 Mise en place des expérimentations

Cette section présente les attaques que nous avons effectuées pour évaluer l'efficacité de détection de notre architecture de sécurité. Ces attaques nous permettent d'évaluer dans la section suivante la capacité de détection de compromission de notre hyperviseur de sécurité et la pertinence de notre environnement d'exécution de tests d'intégrité du logiciel observé.

5.2.1 Insertion de vulnérabilités

Afin de mettre en avant la capacité de détection de notre architecture, nous prenons l'exemple d'attaques exploitant les surfaces d'attaques 1, 2 et 5 de notre modèle de menaces du chapitre 1. Ces surfaces d'attaques exposent trois vulnérabilités représentatives de l'état

Méthode	[appel|retour]	rax	r8	rbx	rcx	rdx
read	appel	2	1	taille	adresse	-
	retour	code d'erreur	1	taille	adresse	valeur lue
write	appel	2	2	taille	adresse	valeur à écrire
	retour	code d'erreur	2	taille	adresse	valeur écrite

TABLE 5.1 – Interface 64 bits malveillante avec l'hyperviseur

de l'art. Pour la surface 1, une interface de programmation de paravirtualisation non protégée permet d'écrire en mémoire de l'hyperviseur de sécurité. Les surfaces d'attaque numéro 2 et 5 exposent deux vulnérabilités émanant d'un périphérique vulnérable et/ou malveillant, qui une fois exploitée, à distance ou depuis une machine virtuelle, permet d'écrire dans la mémoire de l'hyperviseur de sécurité par DMA.

5.2.1.1 Interface paravirtualisée de l'hyperviseur vulnérable

Certains hyperviseurs utilisent la paravirtualisation pour alléger leur taille et réduire leur charge. Dans le cas de Xen, la machine virtuelle privilégiée d'administration, appelée *Dom0*, effectue des appels réguliers à l'hyperviseur afin d'accomplir des actions privilégiées après avoir effectué les calculs de prise de décision dans un environnement non privilégié. Comme le montre l'article [31], l'auteur a réussi à exploiter cette interface paravirtualisée en injectant une vulnérabilité dans l'hyperviseur avec une attaque DMA. Cette vulnérabilité est instrumentée avec une bibliothèque s'exécutant en mode utilisateur qui permet aux applications d'injecter arbitrairement du code dans l'hyperviseur Xen (chargement de modules).

Pour nos expériences, nous reproduisons ce résultat en injectant directement la vulnérabilité 1 à la compilation, qui sera accessible depuis la surface d'attaque 1. Cette vulnérabilité est composée de deux *hypercalls*, qui permettent respectivement de lire et d'écrire dans l'espace mémoire physique (table 5.1).

5.2.1.2 Périphérique malveillant

Les vulnérabilités introduites dans cette section correspondent au risque lié à la présence d'un périphérique malveillant dans le système. Or, selon les hypothèses établies dans la section 3.2 (*hypothèse 1*), l'attaquant n'a pas un accès physique à la machine. Donc, la seule possibilité correspond à la compromission d'un périphérique déjà présent dans le système, via deux types de vulnérabilités. La première est une vulnérabilité exploitée depuis l'espace logiciel et le périphérique compromis ne peut donc agir qu'après le démarrage du système. La seconde est une vulnérabilité exploitable à distance et durant le démarrage du système et le périphérique peut agir avant le démarrage des différents composants logiciels (hyperviseur, système d'exploitation, etc.). Les exploitations de ces deux vulnérabilités diffèrent donc. Dans la suite, nous supposons que le périphérique compromis ne fait pas partie de l'ensemble des ressources indispensables au bon fonctionnement du système (aucun test d'environnement ne lui est associé). Il n'est donc pas nécessaire d'exploiter effectivement ces attaques. Ce qui est important est de se focaliser sur les actions que peuvent entreprendre les périphériques compromis.

Afin de simuler les attaques par un périphérique compromis, nous utilisons un périphérique PCI Express malveillant pour effectuer des accès DMA. Sur la machine cible physique, nous ne disposons que du périphérique de confiance seul et par conséquent, nous sommes

Nom	Taille	Offset	Description
Commande	32 bits	`0x0`	Permet notamment de déclencher la copie
Statut	32 bits	`0x4`	Informations sur le déroulement de la copie
Adresse source	2×32 bits	`0x8`	Adresse mémoire source de la copie
Adresse destination	2×32 bits	`0x10`	Adresse mémoire destination de la copie

Table 5.2 – Interface DMA PCI Express vulnérable

contraints de l'utiliser dans un seul objectif. Si nous souhaitons donc qu'il effectue des accès DMA malveillants, nous devons adapter le *firmware* que nous avons développé pour lui ajouter un comportement malveillant et mener à bien nos expérimentations. Dans la version virtuelle de l'architecture, nous pouvons facilement utiliser un périphérique malveillant virtuel développé à la manière du périphérique de confiance virtuel.

Nous ajoutons donc deux vulnérabilités. La vulnérabilité 2 est accessible depuis l'espace de configuration PCI Express du périphérique (surface d'attaque 2). Elle est composée de quatre registres PCI Express permettant de déclencher une copie mémoire par DMA. Ces quatre registres sont accessibles depuis l'espace mémoire BAR1 du périphérique (table 5.2). Cette vulnérabilité a été mise en œuvre uniquement sur le périphérique malveillant virtuel. La deuxième vulnérabilité (vulnérabilité 3) est ajoutée sur le périphérique physique. Elle permet de lancer des lectures / écritures DMA à distance via des commandes du *shell* de contrôle du périphérique de confiance dont nous avons modifié le *firmware*. Nous ajoutons deux commandes, `hm_read` qui permet de lire 4 Kilos octets dans la totalité de l'espace mémoire physique accessible et `hm_write` qui permet d'écrire un mot arbitraire de 32 bits.

Dans les sous-sections suivantes, nous détaillons les attaques exécutées contre l'hyperviseur de sécurité ainsi que les caractéristiques de son intégrité qui sont modifiées. Nous précisons également comment notre architecture a détecté leur compromission.

5.2.2 Préparation des Attaques

À notre connaissance, les attaques présentées dans cette section sont représentatives des différentes stratégies d'attaques de l'état de l'art parmi celles qui sont bien connues et documentées. Elles ont été exécutées dans la version physique et virtuelle de notre architecture de sécurité.

5.2.2.1 Modification du code et des données de l'hyperviseur

L'objectif ultime d'un attaquant sur une machine Intel, qui s'exécute avec l'extension VT-x activée, est de s'échapper d'une machine virtuelle pour prendre le contrôle de la plateforme. Une attaque peut consister en l'injection de code et de données dans l'espace mémoire de l'hyperviseur (attaque 1). Pour rendre ce code exécutable, il peut être aussi nécessaire de modifier les tables de pages de l'hyperviseur pour ajouter le droit d'exécution à l'adresse virtuelle correspondante au code malveillant (attaque 2). Nous avons mené les attaques 1 et 2 via la vulnérabilité 2.

5.2.2.2 Modification de la configuration de la carte réseau

Notre architecture s'appuie sur un contrôleur réseau Ethernet dédié pour communiquer les alertes sur le réseau 1. Un logiciel malveillant peut tenter de modifier cette configuration pour inhiber l'envoi d'alertes (attaque 3). La configuration de la carte réseau est accessible sur l'espace mémoire PCI Express, la vulnérabilité 1 est la seule permettant d'accéder directement à ces registres.

5.2.2.3 Attaque par virtualisation complète

Un attaquant peut choisir de virtualiser notre hyperviseur pour espérer ne pas être détecté tout en prenant la main sur le matériel. Cette attaque se déroule en général au plus tôt dans le démarrage de la machine, et peut consister à charger un hyperviseur malveillant avant l'hyperviseur de sécurité légitime (attaque 4). Dans ce cas, sa mise en œuvre la plus simple consiste à modifier l'ordre de démarrage de la machine (attaque 5). Dans le cas des *firmwares* UEFI, celui-ci se trouve dans la mémoire flash de la plateforme, qui contient aussi le code des *firmwares* pour les composants matériels embarqués sur une carte mère, notamment le *firmware* UEFI. La vulnérabilité 1 permettra d'écrire le nouvel ordre dans la mémoire flash (dont l'interface d'accès se trouve dans l'espace PCI Express), ainsi que d'installer l'image binaire malveillante de l'hyperviseur de l'attaquant.

5.2.2.4 Attaque de relocalisation

Un attaquant peut aussi réussir à exploiter une faute matérielle ou une vulnérabilité logicielle pour relocaliser l'hyperviseur de sécurité dans la mémoire et installer un nouvel hyperviseur. Nous utilisons la vulnérabilité 2 pour installer cet hyperviseur malveillant. Il sera nécessaire d'injecter du code dans l'espace mémoire de l'hyperviseur légitime pour dérouter son exécution, puis rétablir immédiatement cette modification via le code de l'hyperviseur malveillant (attaque 6). Ainsi, l'hyperviseur de sécurité et ses données sont restaurés après l'attaque.

5.2.2.5 Modification de la configuration de l'IOMMU

Au démarrage, l'IOMMU n'est pas configurée. La fonctionnalité de contrôle d'accès sur les périphériques appelée *DMA Remapping* (DMAR) n'est pas effective. Nous montrons dans [84] qu'il est possible, avec une machine virtuelle exécutant un noyau Linux, depuis un périphérique malveillant, d'imposer sa configuration de contrôle d'accès durant la phase de configuration de l'IOMMU par le driver `iommu-intel.c` de Linux. Ainsi, après l'activation du service DMAR, le périphérique malveillant dispose d'un accès illimité à l'espace mémoire physique de la machine, bien que le driver Linux pense avoir restreint cet accès (attaque 7). Cette attaque est menée via le périphérique physique (vulnérabilité 3).

5.2.2.6 Attaque du logiciel observé

Pour l'évaluation de l'efficacité des tests d'intégrité du logiciel observé, nous prenons le même exemple du *driver* de contrôleur réseau Intel e1000e, cité dans le chapitre 4. Nous utilisons la vulnerabilité 1 pour modifier le code du driver et obtenir un environnement d'exécution en mode noyau (attaque 8).

Numéro	Vuln.	Description
Attaque 1	2	Corruption des données et du code de l'hyperviseur.
Attaque 2	2	Modification de la mémoire virtuelle de l'hyperviseur.
Attaque 3	1	Modification de la configuration du contrôleur réseau.
Attaque 4	1	Virtualisation malveillante de l'hyperviseur.
Attaque 5	1	Modification malveillante de l'ordre de démarrage.
Attaque 6	2	Relocalisation de l'hyperviseur.
Attaque 7	3	Contournement du contrôle d'accès sur les périphériques.
Attaque 8	1	Modification malveillante du *driver* Linux Intel e1000e.

TABLE 5.3 – Liste des attaques contre l'hyperviseur de sécurité et le logiciel observé

La table 5.3 résume les attaques réalisées durant l'expérimentation.

5.3 Capacité de détection des attaques

Les attaques présentées dans la section précédente impactent directement ou indirectement l'intégrité de l'hyperviseur. Cette section montre comment ces attaques peuvent être détectées par notre architecture de sécurité.

```
#include "env/challenge.h"
#include "stdio.h"

// Challenge identique au chapitre 4
static inline void __attribute__((always_inline)) cpuid(void) {/* ... */}
static inline void __attribute__((always_inline)) order(void) {/* ... */}
static inline void __attribute__((always_inline)) nic(void) {/* ... */}

static inline void __attribute__((always_inline)) vmptr(void) {
  uint64_t addr = 0;
  __asm__ __volatile__("vmptrst (%0)": : "D"(&addr));
  systab->answer[0x8] = (addr >> 0x00) & 0xffffffff;
  systab->answer[0x9] = (addr >> 0x20) & 0xffffffff;
}

static inline void __attribute__((always_inline)) host(void) {
  uint64_t rip, rsp, cr3;
  __asm__ __volatile__("vmread %%rdx, %%rax" : "=a" (rip): "d" (0x6c16));
  __asm__ __volatile__("vmread %%rdx, %%rax" : "=a" (rsp): "d" (0x6c14));
  __asm__ __volatile__("vmread %%rdx, %%rax" : "=a" (cr3): "d" (0x6c02));
  systab->answer[0x9] = (rip >> 0x00) & 0xffffffff;
  systab->answer[0xa] = (rip >> 0x20) & 0xffffffff;
  systab->answer[0xb] = (rsp >> 0x00) & 0xffffffff;
  systab->answer[0xc] = (rsp >> 0x20) & 0xffffffff;
  systab->answer[0xd] = (cr3 >> 0x00) & 0xffffffff;
  systab->answer[0xe] = (cr3 >> 0x20) & 0xffffffff;
}

void challenge_start(void) {
  cpuid();
  vmptr();
  host();
  order();
  nic();
}
```

Listing 5.1 – Challenge d'expérimentation

Pour chaque attaque exécutée, nous déterminons quelles sont les caractéristiques de l'intégrité de l'hyperviseur de sécurité qui sont impactées et nous donnons des détails sur le challenge et/ou le test d'environnement qui a permis de détecter la modification.

5.3.1 Challenge utilisé

Pour détecter les atteintes aux caractéristiques d'intégrité de l'hyperviseur de sécurité, nous avons développé un challenge présenté dans le listing 5.1. Il contient cinq parties. La première est extraite du chapitre 4, et permet de tester si l'hyperviseur n'est pas émulé ni virtualisé grâce au temps d'exécution (fonction `cpuid()`). La seconde vérifie si le pointeur vers la structure de contrôle de machine virtuelle courante est bien celui de l'hyperviseur de sécurité (fonction `vmptr()`). Nous vérifions ensuite si le code de l'hyperviseur de sécurité est toujours installé. La fonction `order()` lit l'ordre de démarrage de la machine stocké dans la mémoire flash de la plateforme via le contrôleur de bus SPI mappé dans la mémoire PCI Express. Pour finir, la dernière fonction `nic()` lit les registres PCI Express du contrôleur réseau de l'hyperviseur de sécurité. Les différentes valeurs obtenues sont écrites dans les `BAR[01]` correspondants. Ce challenge est statique, les algorithmes exécutés par celui-ci sont identiques entre deux exécutions, mais sont représentatifs de ce que pourrait être une version générée automatiquement.

5.3.2 Interface graphique de contrôle

Nous avons développé l'interface illustrée dans la figure 5.3 pour visualiser les résultats des expériences. La fenêtre de gauche affiche l'ordonnancement des processus en cours. Un point représente un changement de contexte et chaque ligne un contexte différent. Les fenêtres de droite représentent l'intégrité du logiciel observé pour celle du haut et de l'hyperviseur de sécurité pour celle du bas. Un pic vers le haut (resp. vers le bas) représente un test d'intégrité réussi (resp. manqué).

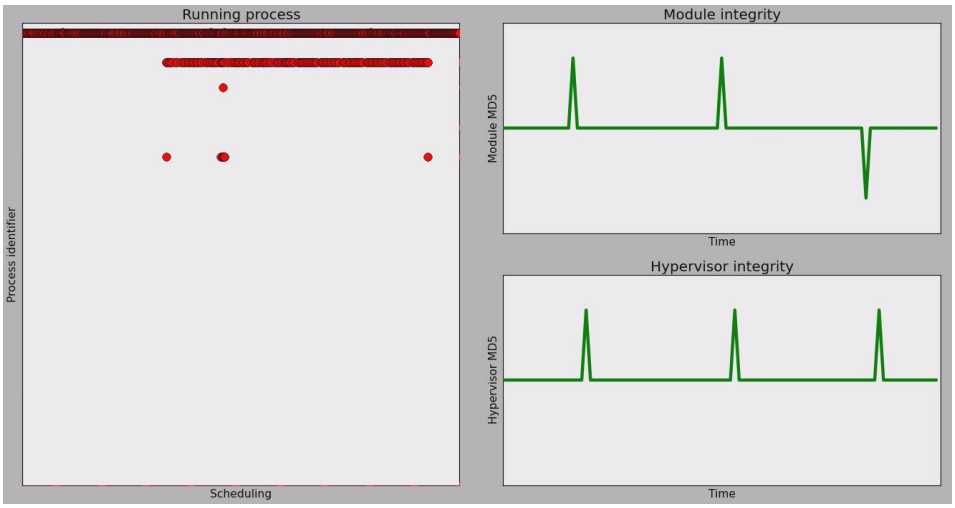

FIGURE 5.3 – Interface graphique utilisateur

Attaque, [V|P]	Carac.	Challenge	Test d'env.	Test d'int.
Attaque 1, V	1		√	
Attaque 2, V	9		√	
Attaque 3, P	13	√		
Attaque 4, P	3,5,9,14	√ (3)		
Attaque 5, P	2	√		
Attaque 6, P	3,5,9,14	√		
Attaque 7, P	15		√	
Attaque 8, P				√

TABLE 5.4 – Détection des attaques

5.3.3 Détection des attaques

Dans cette section nous détaillons les résultats de nos expériences de détection de compromission de notre hyperviseur et du logiciel observé. La table 5.4 résume nos résultats en indiquant si l'expérience a été exécutée sur l'architecture physique ou virtuelle (P,V), en rappelant quelles sont les caractéristiques de l'intégrité de l'hyperviseur de sécurité qui sont violées (le cas échéant) et quelle technique nous a permis de détecter la compromission.

Attaques 1, 2 et 7 Les corruptions engendrées par ces trois attaques sont détectées via trois tests d'environnement. Les empreintes du code de l'hyperviseur de sécurité, de ses tables de page et de la configuration de l'IOMMU sont calculées avec la fonction de hashage `md5sum()` en amont, sur une machine différente de la machine cible. Pour le code, il suffit d'extraire les sections de code du fichier ELF64 de l'hyperviseur de sécurité et d'en calculer l'empreinte. Pour les tables de pages de l'hyperviseur et de l'IOMMU, nous avons reproduit l'algorithme de génération de tables en tenant compte des caractéristiques physiques de la machine cible, comme la taille d'adressage physique par exemple, et reproduit les données de configuration dont nous avons calculé l'empreinte. Les trois tests d'environnement consistent à télécharger du code et des tables de pages durant l'exécution de la machine cible avec le périphérique de confiance, pour calculer leur empreinte courante. L'empreinte courante est comparée à l'empreinte attendue. Ces valeurs n'ont pas à être modifiées après la phase d'initialisation. Pour ces trois attaques, les empreintes ont été détectées comme différentes. Pour le calcul et la vérification de l'empreinte des tables de page, il est nécessaire de masquer certains bits des entrées de tables de pages de l'hyperviseur de sécurité, comme le bit *accessed* ou *dirty*, qui évoluent dans le temps.

Attaque 3 Les contrôleurs réseau Ethernet modernes utilisent des structures mémoire complexes pour configurer l'émission et la réception de trames Ethernet, notamment la localisation en mémoire où le contrôleur réseau doit écrire/lire la trame reçue/à envoyer. Cette configuration est pointée par deux registres de configuration de la première zone mémoire PCI Express du contrôleur Intel (BAR0). Un *driver* doit nécessairement écrire dans ces zones mémoire pour utiliser le contrôleur. Un *driver* malveillant peut modifier cette configuration pour envoyer ses propres trames Ethernet. Notre attaque redirige ces registres vers des adresses à l'extérieur de l'espace mémoire de l'hyperviseur de sécurité. La fonction `nic()` du challenge 5.1 a permis de détecter cette attaque.

Attaques 4 et 5 L'attaque par virtualisation complète est une des plus difficiles a détecter depuis le périphérique de confiance. En effet un hyperviseur malveillant, virtualisant notre hyperviseur en dissimulant sa présence, pourra être détecté uniquement via une analyse du temps d'exécution du challenge. En effet, l'installation d'un tel hyperviseur viole les caractéristiques 3, 5, 9 et 14. Or, il n'est possible de détecter la corruption de la caractéristique 3 que via l'analyse du temps d'exécution. La fonction `cpuid()` du challenge 5.1 est dédiée à cette détection, en forçant fréquemment l'activation d'un éventuel hyperviseur malveillant [1]. La modification de l'ordre de démarrage a été détectée via la fonction `order()`.

Attaque 6 Cette attaque viole les mêmes caractéristiques que l'attaque précédente, mais cette fois-ci, elles sont directement vérifiées par l'exécution de la fonction `host()` et `vmptr()` du challenge 5.1. Si l'hyperviseur est relocalisé ailleurs dans la mémoire, le champ `rip` sera nécessairement modifié dans la structure de contrôle de machine virtuelle. Cette modification sera de plus presque systématiquement accompagnée d'une modification du champ `rsp`, relocalisant la pile de l'hyperviseur et du champ `cr3` changeant le contexte de mémoire virtuelle de l'hyperviseur. Les valeurs lues par la fonction `host()` sont confirmées par la vérification du pointeur de structure de contrôle de machine virtuelle qui sera aussi sûrement différente, car relocalisé ailleurs en mémoire.

Attaque 8 Cette dernière attaque vise le logiciel observé présenté dans le chapitre 4, le *driver* de contrôleur réseau Intel e1000e. Le test d'intégrité effectué sur ce logiciel est un calcul d'empreinte `md5` du code téléchargé dans l'hyperviseur de sécurité et dont la valeur et comparée à l'empreinte calculée hors ligne. Après attaque du *driver*, sa modification est détectée par le test d'intégrité.

L'ensemble des attaques que nous avons effectuées ont donc bien été détectées avec succès, grâce aux challenges, aux tests d'environnement et aux tests d'intégrité du logiciel observé. Même s'il s'agit d'attaques que nous avons nous-mêmes perpétrées, elles sont représentatives d'attaques réelles qui ont été publiées. Leur détection montre par conséquent l'efficacité et la pertinence de nos tests.

5.4 Mesure des performances

Cette section est dédiée à la présentation des expériences d'évaluation de l'impact de notre architecture de sécurité sur les performances du système. Nous commençons par décrire la configuration matérielle sur laquelle l'architecture de sécurité a été testée. Ensuite, nous donnons les résultats d'expériences de *benchmark* qui ont pour but d'évaluer l'impact de l'architecture sur le système.

5.4.1 Configuration matérielle

La machine cible est un Dell precision T1700 avec un processeur Intel i7-4770 associé au chipset PCH Intel c226. Elle embarque 8 Gigaoctets de DDR3 SDRAM. La vitesse du lien PCI Express 2 vers le périphérique de confiance est de 4x. Les interfaces Ethernet de la machine de *monitoring* (Ethernet 2), du périphérique de confiance (Ethernet 1) et de la

1. L'instruction `cpuid` génère systématiquement un VM Exit

PI	Néant	Virt.	5s	1s	100ms
Total	221.10	221.37	222.03	223.71	244.89
Moyen	22.11	22.14	22.20	22.37	24.49
Médian	22.11	22.13	22.20	22.36	24.46
Min.	22.10	22.12	22.19	22.35	24.43
Max.	22.12	22.16	22.25	22.45	24.62
Over.		0.1 %	0.4 %	1.2 %	10 %
Disque	Néant	Virt.	5s	1s	100ms
Total	52.21	51.51	52.18	52.23	52.87
Moyen	5.22	5.15	5.22	5.22	5.29
Médian	4.89	4.84	4.85	4.84	4.98
Min.	4.64	4.71	4.52	4.68	4.67
Max.	8.43	8.14	8.68	8.49	8.70
Over.		-1.3 %	-0.1 %	0.4 %	1.2 %
Réseau	Néant	Virt.	5s	1s	100ms
Total	187.72	187.51	187.92	187.33	186.96
Moyen	46.93	46.88	46.98	46.83	46.74
Médian	46.85	46.89	46.94	46.78	46.69
Min.	46.65	46.68	46.61	46.56	46.67
Max.	47.36	47.06	47.44	47.21	46.91
Over.		-0.1 %	0.1 %	-0.2 %	-0.4 %

TABLE 5.5 – Résultats des benchmarks (valeurs en secondes)

machine de *monitoring* supportent un débit de 1 Giga bits/s. Le connecteur PCI Express 8x du périphérique de confiance est branché à la carte mère via une rallonge (*riser*). Une interface Ethernet supplémentaire ajoutée à la machine cible, utilisée pour les *benchmarks* réseau, est connectée à un réseau 100 Mégas bits qui n'est pas de confiance. Notre hyperviseur de sécurité ne s'installe que sur un seul cœur de processeur pour simplifier son développement. Les tests suivants s'exécutent donc sur une machine ne disposant que d'un seul cœur actif.

5.4.2 Évaluation des performances

Trois ensembles de *benchmarks* ont été exécutés sur l'architecture. Chacun des ensembles est exécuté sur la machine cible, en commençant par une exécution témoin, c'est-à-dire sans notre architecture (colonne *None* dans la table 5.5). Chaque ensemble est ensuite exécuté virtualisé par notre hyperviseur de sécurité seulement. Ensuite, la solution complète est testée, avec les tests d'intégrité du périphérique de confiance, les tests d'environnement et les challenges. Le challenge exécuté est celui présenté dans la section précédente. Le test d'environnement est exécuté une fois durant la phase idle de chaque cycle d'intégrité. Il constitue en une vérification de l'empreinte du code de l'hyperviseur de sécurité de taille de 330 Kilos octets. Enfin, le cycle de test d'intégrité est exécuté à période fixée pour pouvoir mesurer les performances. La période est changée dégressivement de 5s, 1s à 100ms. Une valeur en deçà de 100ms est trop proche de l'unité de temps généralement utilisée par les ordonnanceurs pour être pertinente. À chaque fois qu'un *benchmark* est exécuté, la machine cible est redémarrée afin d'éliminer les éventuels effets de bord.

Le premier ensemble d'expériences calcule 10 fois de suite, dix millions de décimales du nombre PI à l'aide de la bibliothèque GNU Multi Precision (GMP). Le second ensemble effectue 10 fois de suite des copies avec le programme `dd` des systèmes Unix, d'un fichier de 512 Mégas octets, sur le même disque dur dans une partition ext4. Le dernier ensemble effectue 4 fois de suite des transferts via SSH d'un fichier de 512 Mégas octets depuis le disque dur de la machine cible vers une machine distante connectée au réseau local qui n'est pas de confiance.

Concernant les tests d'utilisation du processeur (CPU *burn-in*), la surcharge mesurée (table 5.5) est de moins de 0.4 %, pour une période de 5s et de moins de 1.2 %, pour une période de 1s, ce qui est très acceptable. Cette surcharge atteint 10 % pour une période de 100ms à cause de la nature du *benchmark* PI qui utilise uniquement le CPU. Cette surcharge peut être grandement diminuée (au moins de moitié) en optimisant les communications réseau de l'hyperviseur de sécurité (en désactivant le mode de débogage et la communication des changements de contexte du processeur) ou en optimisant les tests du périphérique de confiance. Notons qu'une période de 100ms est peut-être trop rapide et n'augmentera pas la précision de détection de notre plateforme. Cette expérience est intéressante, mais pas suffisante pour estimer les performances de notre architecture, car un test de *brun in* n'est pas représentatif des différents cas réels d'utilisation d'une machine. Pour cette raison nous avons considéré deux autres séries de *benchmarks* qui se focalisent sur l'utilisation disque et réseau. Ces expériences montrent que les variations de performance dues aux accès disques et à la latence réseau impactent plus le système que notre architecture : nous avons mesuré une surcharge faible (moins de 1,2 %) voire négative, ce qui montre que notre solution conserve des performances quasi identiques.

5.5 Impact global sur la sécurité

L'architecture de sécurité que nous avons présentée dans cette thèse modifie la pile logicielle et matérielle d'une machine cible. Cette section discute de l'impact de l'ajout de ces composants sur la sécurité de la machine cible.

5.5.1 Interface Ethernet de l'hyperviseur de sécurité

Un attaquant peut aussi être en mesure de contrôler le lien Ethernet point à point (Ethernet 1) vers la machine de *monitoring* s'il est capable d'obtenir l'accès au niveau de privilège le plus élevé du CPU (hypothèse 3 du modèle de menaces). Par conséquent, il pourrait aussi envoyer du trafic malveillant ou de fausses alertes au travers de ce lien Ethernet. Les challenges et tests d'environnement effectués par le périphérique de confiance détecteront cette corruption. Le périphérique de confiance peut alors alerter la machine de *monitoring*. Alors, ce trafic et ces alertes seront ignorés par la machine de *monitoring*. Enfin, les attaques initiées depuis la machine de *monitoring* ne sont pas considérées dans nos hypothèses (hypothèse 1 et 2).

5.5.2 Interface PCI Express

Le périphérique de confiance est connecté physiquement à quatre voies PCI Express. Il implémente le support pour la lecture et l'écriture vers des mémoires internes qui sont exposées via les BARs et l'*expansion ROM* PCI Express. Ces deux espaces mémoire n'influencent pas le comportement interne du périphérique de confiance. Un attaquant peut toujours essayer

d'effectuer un déni de service depuis le CPU ou d'autres périphériques matériels (si l'architecture matérielle le permet), pour influencer le temps de propagation des messages. Ces essais résulteront en une alerte par le périphérique de confiance.

5.5.3 Interface Ethernet du périphérique de confiance

L'interface Ethernet du périphérique de confiance est directement connectée à la machine de confiance via un lien Ethernet point à point. Or, nous ne considérons pas d'attaque physique sur ce lien. Donc, du fait que l'interface PCI Express vers le processeur n'apporte aucun accès vers cette interface Ethernet, les attaques depuis le CPU (hypothèse 1 et 2) vers la machine de confiance ne sont pas possibles.

5.5.4 Menaces vers la machine de monitoring

La machine de *monitoring* exécute une distribution GNU/Linux classique et elle inclut les mêmes vulnérabilités logicielles et matérielles et les mêmes surfaces d'attaques que les machines de même type. Pour cette raison, cette machine est totalement isolée de tout le réseau durant l'exécution du cycle d'intégrité et nous ne considérons pas d'attaques contre cette machine.

5.6 Conclusion

Dans ce chapitre, nous avons tout d'abord introduit la conception et le développement d'une version virtuelle de notre architecture de sécurité. Cette plateforme de tests nous a permis de développer plus rapidement les challenges et tests d'environnement pour la plateforme physique. De plus, cette plateforme virtuelle a simplifié la validation des travaux que nous avons effectués par la suite. En effet, ce chapitre continue avec la présentation et la démonstration de la capacité de détection de notre architecture de sécurité. Nous avons pour cela mis en place un jeu de vulnérabilités et d'attaques aboutissant à des modifications de l'intégrité de l'hyperviseur de sécurité et du logiciel observé. Après exécution des attaques, nous avons validé la détection de la modification des différentes caractéristiques de l'intégrité de l'hyperviseur. Les tests de validation des performances ont été effectués et montrent que l'impact de notre architecture sur les performances du processeur est très acceptable, voire non significatif dans certains cas. Enfin, nous avons discuté l'impact sur la sécurité d'une machine lié à l'ajout de notre architecture.

Conclusion

Bilan Étant donné l'ensemble des menaces qui pèsent aujourd'hui sur nos systèmes informatiques, évaluer l'intégrité d'un logiciel en cours d'exécution est fondamental. Comme expliqué dans le premier chapitre de ce manuscrit, cette tâche n'est cependant pas aisée, car il existe de nombreuses attaques qui ciblent toutes les couches d'un système, des moins privilégiées aux plus privilégiées. Ainsi, il n'est pas suffisamment sûr de notre point de vue, de considérer que l'intégrité d'un logiciel peut être évaluée seulement depuis les couches les plus privilégiées du système.

Pour avoir confiance dans les résultats de tests d'intégrité, il est donc nécessaire de ne pas s'appuyer exclusivement sur les composants connus pour pouvoir être compromis. Les travaux de cette thèse proposent une solution basée à la fois sur du logiciel et du matériel. En effet, nous insérons dans notre architecture un observateur matériel qui jouera le rôle de source de confiance. Sa position dans l'architecture le rend très efficace pour tester l'intégrité des couches basses du système. Par contre, il dispose d'un niveau sémantique trop faible pour évaluer l'intégrité de logiciels complexes. Pour cette raison, un logiciel très privilégié et très simple est tout de même ajouté dans les couches basses et privilégiées du processeur, dont le but est d'évaluer périodiquement l'intégrité du logiciel que l'on souhaite protéger. L'intégrité de ce nouveau composant logiciel privilégié est attestée par le composant matériel de confiance, car pour les raisons citées précédemment il pourrait être compromis. Le fait de maîtriser le développement de ce composant logiciel et l'environnement matériel nécessaire à son bon fonctionnement permet de concevoir des tests d'intégrité précis et efficaces qui ont abouti aux notions de challenges et tests d'environnement. Ce composant logiciel est un hyperviseur de sécurité qui ne gère qu'une seule machine virtuelle et dont les fonctionnalités sont réduites au strict minimum. Cet hyperviseur profite des extensions matérielles d'assistance à la virtualisation des processeurs modernes.

Un prototype de cette architecture a été développé pour processeurs Intel et décrit dans ce manuscrit. Ce développement inclut à la fois l'hyperviseur de sécurité et le périphérique de confiance autonome et indépendant du processeur principal, connecté via le bus PCI Express. Ainsi, notre architecture permet de propager la confiance que nous avons dans le périphérique de confiance, via l'hyperviseur de sécurité, jusqu'au logiciel exécuté sur le processeur. Des tests ont été réalisés et ont confirmé la pertinence de l'approche. Du point de vue de l'efficacité des résultats, nous avons effectivement détecté des attaques sur le logiciel observé ou sur notre hyperviseur de sécurité. Du point de vue des performances, la présence de l'hyperviseur de sécurité et l'exécution des tests d'intégrité dégradent très peu les performances du système sur lequel l'architecture de sécurité est installée.

Perspectives Concernant les perspectives de ces travaux, il subsiste encore de nombreuses améliorations pouvant être réalisées sur notre prototype d'architecture de confiance.

Premièrement, les challenges couramment implémentés exécutent un algorithme qui est écrit statiquement durant la phase de conception. Pour introduire de l'aléa dans le cycle d'intégrité, il est pour l'instant nécessaire de compiler manuellement plusieurs versions d'algorithmes pour un challenge donné, produisant des images binaires que la machine de *monitoring* choisira tour à tour aléatoirement. Pour passer notre architecture à l'échelle, il est nécessaire

de concevoir un environnement de développement des challenges qui permettra d'automatiser la phase d'ajout d'aléas dans l'algorithme exécuté sur les valeurs d'entrées caractéristiques choisies en amont, et de produire automatiquement l'image binaire résultante. Ces outils de création d'algorithmes aléatoires devront aussi permettre de déterminer automatiquement la réponse attendue en temps et en valeur. Cette amélioration est indispensable pour empêcher la prédiction, par l'attaquant, du comportement à adopter vis-à-vis des challenges, sans même avoir à les exécuter.

En ce qui concerne le prototype du périphérique de confiance, la capacité d'expression et les performances des tests d'environnement mis en œuvre ne sont pas encore suffisantes pour un passage à l'échelle en production. En effet, concernant l'accès à la mémoire principale de la plateforme via DMA, le débit de téléchargement des pages peut être amélioré en modifiant le composant PCIEE pour qu'il télécharge plusieurs pages en parallèle, dans une zone mémoire redimensionnée en conséquence. Pour les tests d'environnement eux-mêmes, le FAC et les automates qu'il exécute ne sont pas aujourd'hui exploités à leur maximum. Le FAC peut être en conséquence accéléré en ajoutant des instances de cœurs pour pouvoir calculer plusieurs automates en même temps. Son contrôleur mémoire peut être aussi modifié pour qu'il tienne compte des modifications du PCIEE qui contiendra plus d'une page mémoire téléchargée. Aussi, pour pouvoir faciliter l'utilisation du FAC, nous pouvons développer un langage d'expression des automates plus haut niveau que l'assembleur disponible pour l'instant. Le jeu d'instructions du FAC est pour l'instant minimaliste, il peut être étendu pour améliorer sa capacité d'expression. Le prototype de périphérique de confiance est pour l'instant développé sur FPGA. Sa taille physique est conséquente. Cela complique son intégration dans les systèmes existants. Afin de l'industrialiser, il est nécessaire d'en faire un circuit intégré propre à notre application (ASIC), hébergé sur une carte contenant uniquement les périphériques de communication nécessaires et empaqueté en fonction du type de machine qui l'accueillera, comme un serveur en lame ou un ordinateur portable.

Nous avons développé notre prototype d'hyperviseur de sécurité de manière à ce qu'il ait l'empreinte de code la plus faible possible et donc le moins de fonctionnalités inutiles possible. Nous améliorons ses capacités au fur et à mesure qu'il est nécessaire de le faire. Notamment, nous devons améliorer le support de la virtualisation récursive. Pour l'instant notre hyperviseur est capable de s'auto virtualiser un nombre quelconque de fois, mais peine à supporter les gestionnaires de machines virtuelles pris sur étagère, tels que KVM et VMWare ESXi. En ce qui concerne les tests d'intégrité sur le logiciel observé, l'interface d'installation des tests d'intégrité dans l'hyperviseur ainsi que leur interface d'initialisation avec le logiciel observé doivent encore être améliorées, du point de vue de la sécurité et de la complexité d'utilisation. Nous pouvons par exemple intégrer dans l'hyperviseur un chargeur d'application dépendant du système virtualisé, pour supprimer la surface d'attaques qu'est l'interface paravirtualisée actuellement mise en place.

D'un point de vue plus global, nous pouvons aussi augmenter le nombre d'expérimentations effectuées avec notre architecture de sécurité, pour d'autres types de logiciels observés, comme un gestionnaire de machines virtuelles entier par exemple. Aussi, il serait intéressant de valider notre approche sur d'autres architectures matérielles telles que les processeurs AMD et les processeurs et microcontrôleurs ARM. De plus, cette architecture a été pensée au regard des attaques qui constituent la littérature au moment de ces travaux. Il serait intéressant d'évaluer la capacité de cette architecture à détecter les attaques qui apparaîtront dans un futur proche.

Outre les différentes améliorations que nous pouvons encore apporter à nos prototypes, on peut également envisager des perspectives de recherche intéressantes qui pourraient tirer profit d'une telle architecture.

L'architecture que nous avons proposée aborde le problème de la détection de compromission de l'intégrité d'hyperviseur de sécurité via la surveillance de certaines propriétés caractérisant son intégrité. Pour ce faire, nous utilisons essentiellement du calcul d'empreinte de code et de données ainsi que l'exécution de challenges qui permettent de tester le comportement du processeur. Nous pouvons prendre un peu de hauteur dans ces tests en vérifiant plutôt le comportement du logiciel exécuté par l'hyperviseur. En effet, il existe des techniques telles que les graphes de flot de contrôle, que nous pourrions intégrer à la compilation dans le code de notre hyperviseur de sécurité par analyse du code haut niveau. La vérification du comportement de l'hyperviseur pourrait être faite à l'exécution dans notre périphérique de confiance, grâce à la communication du passage du flot d'exécution de l'hyperviseur sur les différents sommets du graphe. Grâce à l'avantage qu'apporte une horloge dédiée, le périphérique de confiance pourra aussi vérifier une version temporelle de graphes de contrôle de l'exécution. Un des avantages de cette approche réside dans l'isolation qu'apporte le périphérique de confiance pour stocker le modèle comportemental attendu et effectuer les vérifications, qui habituellement, sont exécutées sur le processeur principal, et qui peuvent donc être corrompues.

Le développement de logiciel proche du matériel, comme les *firmwares* de machine, les hyperviseurs, ou les chargeurs de démarrages sécurisés ou non, demeure complexe. Un des challenges auxquels les développeurs doivent faire face est la faible capacité de débogage de l'exécution d'un logiciel s'exécutant dans ces modes dégradés. En effet, au niveau du démarrage d'une machine, il n'existe que très peu de périphériques de communication disponibles aidant au débogage. L'utilisation d'un périphérique similaire à notre périphérique de confiance peut s'avérer utile pour améliorer le retour d'informations que peut avoir un développeur de ce type de logiciels. Cependant, il existe déjà de telles cartes électroniques de diagnostic des machines. Mais, développées par les constructeurs, elles sont plutôt dédiées aux techniciens de réparation plutôt qu'aux développeurs de *firmwares* et logiciels bas niveau. De plus ce genre de périphérique ne dispose que de faibles capacités d'expression des erreurs et de communication de l'état de la machine, qui se réduit parfois à un simple code d'erreur prédéfini par le constructeur. Enfin, elles ne sont pas conçues pour aller récupérer l'état d'un cœur de processeur. En raison de sa modularité matérielle (FPGA), logicielle (firmware téléchargeable), ses capacités de communications (pile UDP/IP) et ses différentes interfaces de contrôle et débogage, adapter notre prototype de périphérique de confiance pour le transformer en un débogueur de logiciel proche du matériel, assisté par le matériel, semble pertinent.

Il semble aussi intéressant de pouvoir utiliser notre périphérique pour implanter des mécanismes de points de reprise, qui sont très utilisés dans le domaine de la sûreté de fonctionnement en général. Stocker ces points de reprise dans le périphérique et non sur le processeur principal offre une bien meilleure confiance en leur intégrité et par la même des garanties de reprises de logiciel qui seraient défaillantes pour diverses raisons (fautes accidentelles ou malveillances).

En somme, l'idée même d'un composant matériel de confiance, autonome, connecté au processeur principal, avec la capacité d'exécuter en toute confiance certaines fonctions critiques (du point de vue de la sécurité ou d'autres points de vue) nous semble une approche intéressante et suffisamment générique pour être adaptée et utilisée dans de multiples situations.

Bibliographie

[1] National Institute of Standards and Technology (NIST), "The NIST Definition of Cloud Computing." http://dx.doi.org/10.6028/NIST.SP.800-145, 2011. (Cité en page 6.)

[2] G. J. Popek and R. P. Goldberg, "Formal requirements for virtualizable third generation architectures," *Commun. ACM*, vol. 17, pp. 412–421, July 1974. (Cité en page 9.)

[3] VMWare, "Understanding Full Virtualization, Paravirtualization, and Hardware Assist." http://www.vmware.com/resources/techresources/1008, http://www.vmware.com/files/pdf/VMware_paravirtualization.pdf, 2007. (Cité en page 11.)

[4] "KQemu/Doc - QEMU." http://wiki.qemu.org/KQemu/Doc, 2016. (Cité en page 11.)

[5] "chroot." http://man7.org/linux/man-pages/man2/chroot.2.html. (Cité en page 12.)

[6] "Docker." https://www.docker.com/. (Cité en page 12.)

[7] "Linux Containers." https://linuxcontainers.org/. (Cité en page 12.)

[8] "Linux-VServer." http://linux-vserver.org/. (Cité en page 12.)

[9] "OpenVZ." https://openvz.org. (Cité en page 12.)

[10] M. Rezaei, N. Moosavi, H. Nemati, and R. Azmi, "Tcvisor : A hypervisor level secure storage," in *Internet Technology and Secured Transactions (ICITST), 2010 International Conference for*, pp. 1–9, 2010. (Cité en pages 12 et 29.)

[11] Intel® LAN Access Division, "PCI-SIG SR-IOV Primer." http://www.intel.com/content/www/us/en/pci-express/pci-sig-sr-iov-primer-sr-iov-technology-paper.html, 2011. (Cité en page 12.)

[12] "The Xen Project." http://www.xenproject.org/. (Cité en page 14.)

[13] Oracle, "Virtualbox." https://www.virtualbox.org/, 2016. (Cité en page 14.)

[14] VMware, "VMware workstation." https://www.vmware.com/fr/products/workstation, 2016. (Cité en page 14.)

[15] "Kernel Virtual Machines." http://www.linux-kvm.org/, 2016. (Cité en page 14.)

[16] E. Lacombe, V. Nicomette, and Y. Deswarte, "A hardware-assisted virtualization based approach on how to protect the kernel space from malicious actions," in *Proceedings of the 18th Annual Conference of the European Institute for Computer Antivirus Research (EICAR), Berlin, Allemagne*, 2009. (Cité en page 14.)

[17] S. Duverger, "sduverger/ramooflax, a pre-boot virtualization tool." https://github.com/sduverger/ramooflax. (Cité en page 14.)

[18] "Bitvisor : A single-vm lightweight hypervisor." http://www.bitvisor.org/. (Cité en page 14.)

[19] VMware, "VMware ESXi." https://www.vmware.com/fr/products/esxi-and-esx/overview. (Cité en page 14.)

[20] J. Szefer, E. Keller, R. B. Lee, and J. Rexford, "Eliminating the hypervisor attack surface for a more secure cloud," in *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, (New York, NY, USA), pp. 401–412, ACM, 2011. (Cité en page 15.)

[21] J.-C. Laprie, J. Arlat, J. Blanquart, A. Costes, Y. Crouzet, Y. Deswarte, J. Fabre, H. Guillermain, M. Kaâniche, K. Kanoun, *et al.*, "Guide de la sûreté de fonctionnement," *Toulouse : Cépaduès*, 1995. (Cité en page 16.)

[22] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE transactions on dependable and secure computing*, vol. 1, no. 1, pp. 11–33, 2004. (Cité en page 16.)

[23] D. Powell, R. Stroud, *et al.*, "Malicious-and accidental-fault tolerance for internet applications-conceptual model and architecture," 2001. (Cité en pages 18 et 19.)

[24] L. Duflot, D. Etiemble, and O. Grumelard, "Using cpu system management mode to circumvent operating system security functions," *CanSecWest/core06*, 2006. (Cité en page 18.)

[25] Y.-A. Perez, L. Duflot, O. Levillain, and G. Valadon, "Quelques éléments en matière de sécurité des cartes réseau," in *Actes du 8ème symposium sur la sécurité des technologies de l'information et des communications (SSTIC)*, 2010. (Cité en page 18.)

[26] Y. Deswarte, L. Blain, and J. C. Fabre, "Intrusion tolerance in distributed computing systems," in *Research in Security and Privacy, 1991. Proceedings., 1991 IEEE Computer Society Symposium on*, pp. 110–121, May 1991. (Cité en page 19.)

[27] T. Garfinkel and M. Rosenblum, "When virtual is harder than real : Security challenges in virtual machine based computing environments," in *Proceedings of the 10th Conference on Hot Topics in Operating Systems - Volume 10*, HOTOS'05, (Berkeley, CA, USA), pp. 20–20, USENIX Association, 2005. (Cité en page 20.)

[28] F. Zhang, J. Chen, H. Chen, and B. Zang, "Cloudvisor : Retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, (New York, NY, USA), pp. 203–216, ACM, 2011. (Cité en page 22.)

[29] I. Studnia, E. Alata, Y. Deswarte, M. Kaâniche, and V. Nicomette, "Survey of Security Problems in Cloud Computing Virtual Machines," in *Computer and Electronics Security Applications Rendez-vous (C&ESAR 2012). Cloud and security :threat or opportunity*, (Rennes, France), pp. p. 61–74, Nov. 2012. (Cité en pages 24 et 27.)

[30] P. Ferrie, "Attacks on more virtual machine emulators," *Symantec Technology Exchange*, vol. 55, 2007. (Cité en page 24.)

[31] R. Wojtczuk, "Subverting the xen hypervisor," *Black Hat USA*, vol. 2008, 2008. (Cité en pages 25, 26 et 112.)

[32] K. Kortchinsky, "Cloudburst : A vmware guest to host escape story," *Black Hat USA*, 2009. (Cité en pages 25 et 26.)

[33] N. Elhage, "Virtunoid : Breaking out of kvm," *Black Hat USA*, 2011. (Cité en pages 25 et 26.)

[34] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage, "Hey, you, get off of my cloud : exploring information leakage in third-party compute clouds," in *Proceedings of the 16th ACM conference on Computer and communications security*, pp. 199–212, ACM, 2009. (Cité en page 26.)

[35] "Amazon EC2." `https://aws.amazon.com/fr/ec2/`. (Cité en page 26.)

[36] J. Rutkowska and A. Tereshkin, "Bluepilling the xen hypervisor," *Black Hat USA*, 2008. (Cité en page 27.)

[37] H. Douglas and C. Gehrmann, "Secure virtualization and multicore platforms state-of-the-art report," tech. rep., SICS, Swedish Institute of Computer Science, 2009. (Cité en page 27.)

[38] A. Seshadri, M. Luk, N. Qu, and A. Perrig, "Secvisor : A tiny hypervisor to provide lifetime kernel code integrity for commodity oses," in *ACM SIGOPS Operating Systems Review*, vol. 41, pp. 335–350, ACM, 2007. (Cité en page 28.)

[39] B. Stelte, R. Koch, and M. Ullmann, "Towards integrity measurement in virtualized environments - a hypervisor based sensory integrity measurement architecture (sima)," in *Technologies for Homeland Security (HST), 2010 IEEE International Conference on*, pp. 106–112, 2010. (Cité en page 28.)

[40] A. Azab, P. Ning, E. Sezer, and X. Zhang, "Hima : A hypervisor-based integrity measurement agent," in *Computer Security Applications Conference, 2009. ACSAC '09. Annual*, pp. 461–470, 2009. (Cité en page 28.)

[41] Éric Lacombe, V. Nicomette, and Y. Deswarte, "Une approche de virtualisation assistée par le matériel pour protéger l'espace noyau d'actions malveillantes," in *Actes du 7ème symposium sur la sécurité des technologies de l'information et des communications (SSTIC)*, pp. 321–346, 2009. (Cité en page 28.)

[42] E. Lacombe, *Sécurité des noyaux de systèmes d'exploitation*. PhD thesis, Systèmes, 2009. (Cité en page 28.)

[43] W. Qingbo, W. Chunguang, and T. Yusong, "System monitoring and controlling mechanism based on hypervisor," in *Parallel and Distributed Processing with Applications, 2009 IEEE International Symposium on*, pp. 549–554, 2009. (Cité en page 28.)

[44] L. Litty, H. A. Lagar-Cavilla, and D. Lie, "Hypervisor support for identifying covertly executing binaries.," in *USENIX Security Symposium*, pp. 243–258, 2008. (Cité en pages 29 et 32.)

[45] I. Unified EFI, "Unified extensible firmware interface specification." `http://www.uefi.org/sites/default/files/resources/UEFI_2_3_1_C.pdf`, 2012. (Cité en page 29.)

[46] M. Hirano, T. Shinagawa, H. Eiraku, S. Hasegawa, K. Omote, K. Tanimoto, T. Horie, S. Mune, K. Kato, T. Okuda, E. Kawai, and S. Yamaguchi, "A two-step execution mechanism for thin secure hypervisors," in *Emerging Security Information, Systems and Technologies, 2009. SECURWARE '09. Third International Conference on*, pp. 129–135, 2009. (Cité en page 29.)

[47] G. Coker, "Xen security modules (xsm)," *Xen Summit*, 2006. (Cité en page 29.)

[48] A. Azab, P. Ning, Z. Wang, X. Jiang, X. Zhang, and N. Skalsky, "Hypersentry : enabling stealthy in-context measurement of hypervisor integrity," in *Proceedings of the 17th ACM conference on Computer and communications security*, pp. 38–49, ACM, 2010. (Cité en page 30.)

[49] "IPMI Specification, V2.0, Rev. 1.1 : Document." `http://www.intel.com/content/www/us/en/servers/ipmi/ipmi-second-gen-interface-spec-v2-rev1-1.html`, 2013. (Cité en page 30.)

[50] Z. Wang and X. Jiang, "Hypersafe : A lightweight approach to provide lifetime hypervisor control-flow integrity," in *Security and Privacy (SP), 2010 IEEE Symposium on*, pp. 380–395, IEEE, 2010. (Cité en page 30.)

[51] C. Tan, Y. Xia, H. Chen, and B. Zang, "Tinychecker : Transparent protection of vms against hypervisor failures with nested virtualization," in *Dependable Systems and Networks Workshops (DSN-W), 2012 IEEE/IFIP 42nd International Conference on*, pp. 1–6, 2012. (Cité en page 30.)

[52] R. Wojtczuk, J. Rutkowska, and A. Tereshkin, "Xen 0wning trilogy," in *Black Hat conference*, 2008. (Cité en page 30.)

[53] F. Zhang, J. Chen, H. Chen, and B. Zang, "Cloudvisor : Retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pp. 203–216, ACM, 2011. (Cité en page 31.)

[54] M. Hirano, T. Shinagawa, H. Eiraku, S. Hasegawa, K. Omote, K. Tanimoto, T. Horie, K. Kato, T. Okuda, E. Kawai, and S. Yamaguchi, "Introducing role-based access control to a secure virtual machine monitor : Security policy enforcement mechanism for distributed computers," in *Asia-Pacific Services Computing Conference, 2008. APSCC '08. IEEE*, pp. 1225–1230, 2008. (Cité en page 31.)

[55] Y. Li, J. M. McCune, and A. Perrig, "Viper : verifying the integrity of peripherals' firmware," in *Proceedings of the 18th ACM conference on Computer and communications security*, pp. 3–16, ACM, 2011. (Cité en pages 31 et 51.)

[56] F. Zhang, "Iocheck : A framework to enhance the security of i/o devices at runtime," in *Dependable Systems and Networks Workshop (DSN-W), 2013 43rd Annual IEEE/IFIP Conference on*, pp. 1–4, IEEE, 2013. (Cité en page 31.)

[57] T. Alves and D. Felton, "Trustzone : Integrated hardware and software security," *ARM white paper*, vol. 3, no. 4, pp. 18–24, 2004. (Cité en page 32.)

[58] I. Anati, S. Gueron, S. Johnson, and V. Scarlata, "Innovative technology for cpu based attestation and sealing," in *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, p. 10, 2013. (Cité en page 32.)

[59] N. L. Petroni Jr, T. Fraser, J. Molina, and W. A. Arbaugh, "Copilot-a coprocessor-based kernel runtime integrity monitor.," in *USENIX Security Symposium*, pp. 179–194, San Diego, USA, 2004. (Cité en page 32.)

[60] Intel Corporation, "Intel® 8 Series/C220 Series Chipset Family Platform Controller Hub (PCH)." `http://www.intel.com/content/dam/www/public/us/en/documents/datasheets/8-series-chipset-pch-datasheet.pdf`, 2014. (Cité en pages 36, 37, 38, 40 et 41.)

[61] Intel Corporation, "MultiProcessor Specification." `http://www.intel.com/design/pentium/datashts/24201606.pdf`, 1997. (Cité en page 36.)

[62] Intel Corporation, "Intel® 64 and IA-32 Architectures Software Developer's Manual Combined Volumes :1, 2A, 2B, 2C, 3A, 3B, and 3C." `http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf`, 2015. (Cité en pages 39, 41, 43 et 94.)

[63] L. Duflot and O. Levillain, "ACPI et routine de traitement de la SMI," in *Actes du 7ème symposium sur la sécurité des technologies de l'information et des communications (SSTIC)*, pp. 132–168, 2009. (Cité en pages 40 et 41.)

[64] Intel Corporation, "Mobile 4th Generation Intel ® Core TM Processor Family, Mobile Intel ® Pentium ® Processor Family, and Mobile Intel ® Celeron ® Processor Family." `http://www.intel.com/content/dam/www/public/us/en/documents/datasheets/4th-gen-core-family-desktop-vol-2-datasheet.pdf`, 2013. (Cité en page 40.)

[65] Intel Corporation, "Intel® Virtualization Technology for Directed I/O." `http://www.intel.com/content/dam/www/public/us/en/documents/product-specifications/vt-directed-io-spec.pdf`, 2014. (Cité en pages 40, 47 et 94.)

[66] F. Lone Sang, V. Nicomette, and Y. Deswarte, "Ironhide : plate-forme d'attaques par entrées-sorties," in *Actes du 10ème symposium sur la sécurité des technologies de l'information et des communications (SSTIC)*, pp. 237–265, 2012. (Cité en page 40.)

[67] Trusted Computing Group, "Tpm specification version 1.2 : Design principles." `http://www.trustedcomputinggroup.org/resources/tpm_main_specification`, 2011. (Cité en page 41.)

[68] Intel Corporation, "Intel® Trusted Execution Technology (Intel® TXT) Software Development Guide, Measured Launched Environment Developer's Guide." `http://www.intel.com/content/dam/www/public/us/en/documents/guides/intel-txt-software-development-guide.pdf`, 2015. (Cité en page 41.)

[69] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. van Doorn, and P. Khosla, "Pioneer : verifying code integrity and enforcing untampered code execution on legacy systems," in *ACM SIGOPS Operating Systems Review*, vol. 39, pp. 1–16, ACM, 2005. (Cité en page 51.)

[70] A. Seshadri, A. Perrig, L. Van Doorn, and P. Khosla, "Swatt : Software-based attestation for embedded devices," in *Security and Privacy, 2004. Proceedings. 2004 IEEE Symposium on*, pp. 272–282, IEEE, 2004. (Cité en page 51.)

[71] K. Eldefrawy, G. Tsudik, A. Francillon, and D. Perito, "Smart : Secure and minimal architecture for (establishing dynamic) root of trust.," in *NDSS*, vol. 12, pp. 1–15, 2012. (Cité en page 51.)

[72] S. Bourdeauducq, *A performance-driven SoC architecture for video synthesis*. Skolan för informatons-och kommunikationsteknik, Kungliga Tekniska högskolan, 2010. (Cité en page 75.)

[73] Xilinx, Inc., "LogiCORE™ IP Virtex®-6 FPGA Integrated Block for PCI Express®." `http://www.xilinx.com/support/documentation/ip_documentation/pcie_blk_plus_ug341.pdf`, March 2011. (Cité en page 77.)

[74] S. Bourdeauducq, "Milkymist, an open hardware video synthesis platform," in *26th Chaos Communication Congress*, Chaos Computing Club, December 2009. (Cité en page 78.)

[75] S. Boudeauducq, "Flickernoise." `https://github.com/m-labs/flickernoise`. (Cité en page 78.)

[76] I. Ryan Geiss, Nullsoft, "Milkdrop." `http://www.geisswerks.com/milkdrop/`. (Cité en page 78.)

[77] T. R. Project, "Rtems real time operating system (rtos)." `https://www.rtems.org/`. (Cité en page 80.)

[78] T. MATSUYA and S. Bourdeauducq, "milkymist-linux." `https://github.com/m-labs/linux-milkymist`. (Cité en page 80.)

[79] F. Lone Sang, V. Nicomette, and Y. Deswartes, "Ironhide : plate-forme d'attaques par entrées-sorties," in *Actes du 10ème symposium sur la sécurité des technologies de l'information et des communications (SSTIC)*, pp. 237–265, 2012. (Cité en page 82.)

[80] M. E. Lesk and E. Schmidt, "Lex : A lexical analyzer generator," 1975. (Cité en page 87.)

[81] M. Ben-Yehuda, M. D. Day, Z. Dubitzky, M. Factor, N. Har'El, A. Gordon, A. Liguori, O. Wasserman, and B.-A. Yassour, "The turtles project : Design and implementation of nested virtualization.," in *OSDI*, vol. 10, pp. 423–436, 2010. (Cité en page 95.)

[82] B. Morgan, É. Alata, V. Nicomette, and G. Averlant, "Abyme : un voyage au cœur des hyperviseurs récursifs," in *Symposium sur la Sécurité des Technologies de l'Information et des Communications*, 2015. (Cité en page 97.)

[83] J. Roux, "Caractérisation et contrôle de l'intégrité d'un hyperviseur de sécurité," 2016. (Cité en page 111.)

[84] B. Morgan, E. Alata, V. Nicomette, and M. Kaaniche, "Bypassing iommu protection against i/o attacks," in *LADC 2016 ()*, oct 2016. (Cité en page 114.)

Résumé

L'utilisation des systèmes informatiques est aujourd'hui en pleine évolution. Le modèle classique qui consiste à associer à chaque utilisateur une machine physique qu'il possède et dont il va exploiter les ressources devient de plus en plus obsolète. Aujourd'hui, les ressources informatiques que l'on utilise peuvent être distribuées n'importe où dans l'Internet et les postes de travail du quotidien ne sont plus systématiquement des machines réelles. Cette constatation met en avant deux phénomènes importants qui sont à l'origine de l'évolution de notre utilisation de l'informatique : le Cloud computing et la virtualisation. Le Cloud computing (ou informatique en nuage en français) permet à un utilisateur d'exploiter des ressources informatiques, de granularités potentiellement très différentes, pendant une durée variable, qui sont à disposition dans un nuage de ressources. L'informatique en nuage doit donc pouvoir s'adapter à la demande et facilement se reconfigurer. Une manière d'atteindre ces objectifs nécessite notamment l'utilisation de machines virtuelles et des techniques de virtualisation associées. Même si la virtualisation de ressources informatiques n'est pas née avec le Cloud, l'avènement du Cloud a considérablement augmenté son utilisation. L'ensemble des fournisseurs d'informatique en nuage s'appuient aujourd'hui sur des machines virtuelles, qui sont beaucoup plus facilement déployables et déplaçables que des machines réelles.

Ainsi, s'il est indéniable que l'utilisation de la virtualisation apporte un véritable intérêt pour l'informatique d'aujourd'hui, il est par ailleurs évident que sa mise en œuvre ajoute une complexité aux systèmes informatiques, complexité à la fois logicielle (gestionnaire de machines virtuelles) et matérielle (nouveaux mécanismes d'assistance à la virtualisation intégrés dans les processeurs). À partir de ce constat, il est légitime de se poser la question de la sécurité informatique dans ce contexte où l'architecture des processeurs devient de plus en plus complexe, avec des modes de plus en plus privilégiés. Étant donné la complexité des systèmes informatiques, l'exploitation de vulnérabilités présentes dans les couches privilégiées ne risque-t-elle pas d'être très sérieuse pour le système global ? Étant donné la présence de plusieurs machines virtuelles, qui ne se font pas mutuellement confiance, au sein d'une même machine physique, est-il possible que l'exploitation d'une vulnérabilité soit réalisée par une machine virtuelle compromise ? N'est-il pas nécessaire d'envisager de nouvelles architectures de sécurité prenant en compte ces risques ?

C'est à ces questions que cette thèse propose de répondre. En particulier, nous présentons un panorama des différents problèmes de sécurité dans des environnements virtualisés et des architectures matérielles actuelles. À partir de ce panorama, nous proposons dans nos travaux une architecture originale permettant de s'assurer de l'intégrité d'un logiciel s'exécutant sur un système informatique, quel que soit son niveau de privilège. Cette architecture est basée sur une utilisation mixte de logiciel (un hyperviseur de sécurité développé par nos soins, s'exécutant sur le processeur) et de matériel (un périphérique de confiance, autonome, que nous avons également développé).

Mots clés : Sécurité, Virtualisation, Hyperviseur, Test d'intégrité, Périphérique

Abstract

Computer systems are nowadays evolving quickly. The classical model which consists in associating a physical machine to every users is becoming obsolete. Today, computer resources we are using can be distributed any place on the Internet and usual workstations are not systematically physical machines anymore. This fact is enlightening two important phenomenons which are leading the evolution of the usage we make of computers : Cloud computing and hardware virtualization. Cloud computing enables users to exploit computers resources, with a fine grained granularity and a non predefined amount of time, which are available into a cloud of resources. Cloud systems must be able to adapt quickly to a fluctuating demand and being able to reconfigure themselves quickly. One way to reach these goals is dependant of the usage of virtual machines and the associated virtualization techniques. Even if computer resource virtualization has not been introduced by the cloud, the arrival of the cloud substantially increased its usage. Nowadays, all cloud providers are using virtual machines, which are much more deployable and movable than physical machines.

Thus, even if we can see that virtualization is a real interest for modern computer science, it is either clear that its implementation is adding complexity to computer systems, both at software and hardware levels. From this observation, it is legitimate to ask the question about computer security in this context, with an increased architecture complexity and more and more privileged execution modes. Given the presence of multiple virtual machines, which do not trust each other, and which are executing in the same physical machine, will a compromised virtual machine be able to exploit one vulnerability ? Isn't it necessary to consider new security architectures taking these risks into account ?

This thesis is trying to answer to these questions. In particular, we are introducing state of the art security issues in virtualized environment of modern architectures. Starting from this work, we are proposing an original architecture ensuring the integrity of a software being executed on a computer system, regardless its privilege level. This architecture is both using software, a security hypervisor, and hardware, a trusted peripheral, we have both designed and implemented.

Keywords : Security, Virtualization, Hypervisors, Integrity Checking, Peripherals