Tests d'intégrité d'hyperviseurs de machines virtuelles à distance et assisté par le matériel

Benoît Morgan, Éric Alata,
Vincent Nicomette
benoit.morgan@laas.fr eric.alata@laas.fr
vincent.nicomette@laas.fr

INSA Toulouse, LAAS-CNRS

Résumé Les gestionnaires de machines virtuelles (ou VMM) sont de plus en plus utilisés dans le monde de l'informatique aujourd'hui (notamment, depuis l'engouement pour le Cloud Computing). Du point de vue de la sécurité, ces VMM peuvent constituer une cible de choix pour les attaquants, étant donné leur installation dans des couches logicielles très privilégiées. Afin de les sécuriser, de multiples solutions ont été proposées mais beaucoup de vulnérabilités ont été recensées également. Ces vulnérabilités peuvent cibler le logiciel, mais aussi les couches matérielles sous-jacentes. Nous proposons dans cet article une nouvelle méthode de tests d'intégrité à distance permettant de s'assurer efficacement de l'intégrité d'un logiciel tel qu'un VMM. Ce test ne reposant sur aucun logiciel exécuté directement sur le processeur de la machine, il ne peut être victime de vulnérabilités incluses dans les couches logicielles ou matérielles de la machine. Nous détaillons le principe de cette méthode, son fonctionnement ainsi qu'une implémentation en cours basée sur l'utilisation d'une carte PCI-Expresss dédiée.

1 Introduction

La virtualisation est utilisée massivement aujourd'hui, notamment dans les architectures de type Cloud. Si l'utilisation de machines virtuelles (VMs) est intéressante pour des raisons d'économie du matériel, de simplicité de déploiement et de maintenance, elle entraîne aussi des problèmes de sécurité qu'il faut gérer. Les machines virtuelles s'exécutent sur une même entité physique mais n'appartiennent pas forcement au même client, il est fondamental d'assurer leur intégrité. Le gestionnaire de machines virtuelles, (ou *VMM* ou *hyperviseur*) est chargé d'assurer l'isolation spatiale et temporelle des machines virtuelles et à ce titre, doit assurer leur intégrité.

L'hyperviseur ayant donc un rôle essentiel, il peut être la cible d'attaques. Les machines virtuelles peuvent augmenter leurs privilèges en utilisant des vulnérabilités logicielles ou matérielles dans le but d'obtenir

plus de temps processeur, obtenir plus d'espace mémoire ou encore accéder à l'espace des VMs d'autres clients, comme à celui des périphériques. Les attaques peuvent être exécutées par du code sur le processeur [22] mais aussi par entrées/sorties [18]. Il est donc nécessaire d'inclure des mécanismes permettant de s'assurer de l'intégrité du VMM.

Historiquement, le logiciel qui s'exécute sur un processeur est organisé en différentes couches qui ont de plus en plus de privilèges lorsqu'elles sont proches du matériel, et la protection du logiciel est en partie assurée par cette organisation en couches. Par exemple, sur une architecture de type x86, les applications sont protégées par un système d'exploitation au travers des anneaux (*rings*) et de la virtualisation de la mémoire introduits par la segmentation et la pagination. Ces mêmes OS sont à leur tour protégés par un VMM s'exécutant dans un mode plus privilégié (le mode *VMX Root* en l'occurrence). Il en est de même pour les handlers du mode de management, qui est le mode le plus privilégié d'une machine x86. Cependant, des vulnérabilités ont été découvertes dans chacune de ces couches logicielles [10,28] et de nouvelles apparaîtront probablement au fur et à mesure que ces technologies évolueront et que des couches matérielles ou logicielles, toujours plus privilégiées seront ajoutées. Par conséquent, on ne peut faire aveuglément confiance aux couches logicielles les plus privilégiées telles que les VMM, même lorsqu'elles sont assistées par des technologies matérielles et il faut trouver un moyen de s'assurer de leur intégrité, à l'aide d'une méthode qui n'ajoute pas une autre couche plus privilégiée supplémentaire et qui ne se base sur l'exécution de logiciel sur le processeur lui-même.

Par ailleurs, certains VMM sont très complexes et volumineux (VMWare, KVM ou Xen) et il est particulierement difficile de s'assurer de leur intégrité. C'est d'autant plus difficile qu'on ne dispose pas de leurs sources [25]. Et si un VMM est corrompu par un logiciel malveillant, ce logiciel malveillant peut facilement passer outre l'exécution d'un logiciel de tests d'intégrité local, puisqu'il dispose des privilèges les plus élevés. Des outils matériels tels que TPM et measured late launch (Intel TxT) permettent d'établir une chaîne de confiance au chargement et ainsi de s'assurer de l'intégrité d'un VMM au démarrage de la machine, mais cela n'est pas suffisant durant la phase d'exécution [27].

Nous proposons donc dans cet article une méthodologie permettant d'assurer la protection d'un VMM. Elle est basée sur deux points. Premièrement, le développement d'un petit hyperviseur de sécurité, permettant de détecter les corruptions de ce VMM. Notre hyperviseur utilise les techniques d'assistance à la virtualisation, ceci même si le VMM qu'il virtualise

les utilise déjà. On parle alors de virtualisation récursive ou de *nested virtualization*. Notre hyperviseur étant spécifiquement dédié à la détection de compromission du VMM, la taille de son code demeure modeste. Il est beaucoup plus simple de vérifier qu'il ne possède pas de vulnérabilités, contrairement aux VMM plus conséquents, dont le rôle est la gestion de nombreuses machines virtuelles par exemple. Cependant même si la confiance que l'on peut avoir dans notre hyperviseur est grande, nous savons également qu'il peut contenir des vulnérabilités et être compromis par un bug matériel par exemple [10,28], conformément à ce que nous avons évoqué plus haut. Par conséquent, nous nous assurons de son intégrité à l'aide de tests réalisés par une carte d'entrées/sorties spécifique, et pilotée à distance depuis une machine de confiance. Ces tests nous permettent de nous assurer de l'intégrité de l'hyperviseur, en utilisant du logiciel spécifique, s'exécutant sur une carte dédiée, et non exécutée par le processeur.

Notons que notre hyperviseur de sécurité peut tout à fait être utilisé pour assurer la sécurité d'une couche logicielle quelle qu'elle soit. Nous l'avons développé initialement pour protéger un VMM mais il peut également assurer la protection d'un système d'exploitation par exemple. Son principe de fonctionnement reste le même : détecter les compromissions de la couche logicielle supérieure en la virtualisant. C'est pourquoi, dans la suite de cet article, nous considérerons que la couche logicielle virtualisée par notre hyperviseur de sécurité est quelconque et nous nous focaliserons sur le test d'intégrité à distance de cet hyperviseur, en particulier sur la description du protocole d'exécution de ces tests. C'est la contribution de cet article.

L'article s'organise de la façon suivante. Dans la section 2, nous situons notre travail par rapport aux travaux connexes dans le domaine. La section 3 propose ensuite un rappel des technologies matérielles spécifiques à l'architecture x86 dont nous avons besoin pour la bonne compréhension de cet article. La section 4 présente ensuite rapidement les fonctionnalités de notre hyperviseur. La section 5 propose de caractériser l'intégrité de notre hyperviseur, c'est-à-dire de déterminer l'ensemble des données qui sont à vérifier pour s'assurer de son intégrité lors des tests. La section 6 présente les différentes techniques qui nous permettent d'accéder à ces données et précise le choix que nous avons fait pour y accéder. Les sections 7 et 8 décrivent ensuite le protocole de test d'intégrité lui-même ainsi qu'un prototype en cours de développement. Enfin, la section 9 conclut cet article et propose quelques perspectives.

2 État de l'art

De nombreux hyperviseurs sont disponibles et nous pouvons citer deux grandes catégories. La première catégorie concerne les hyperviseurs destinés à fournir à l'utilisateur un environnement virtualisé performant pour ses machines virtuelles. Les implémentations sont disponibles sous la forme commerciale avec, par exemple, VMWare ESXi et Hyper-Z ou encore sous la forme libre avec, par exemple, VirtualBox, KVM et Xen. La seconde catégorie concerne les hyperviseurs destinés à assurer les propriétés de sécurité. Les hyperviseurs de cette catégorie profitent du positionnement privilégié de l'hyperviseur pour observer le comportement des couches applicatives supérieures. Ces couches applicatives observées peuvent être un système d'exploitation virtualisé voire un autre hyperviseur si les techniques de virtualisation imbriquée sont mises en œuvre. Une autre manière de classer les hyperviseurs consiste à identifier si l'hyperviseur s'installe directement au dessus du matériel (hyperviseur de type 1) ou si il s'exécute avec le support d'un système d'exploitation (hyperviseur de type 2). Dans la suite, nous présentons un tour d'horizon des hyperviseurs qui peuvent être employés pour assurer les propriétés de sécurité.

Hytux [16], est un hyperviseur de type 2 qui a pour but de protéger un noyau Linux. Il utilise directement les extensions de virtualisation Intel (sans profiter des dernières extensions telles que EPT) et il intercepte les modifications des composants du noyau qui ne devraient pas évoluer durant l'exécution du système. Ces modifications sont considérées comme reflétant une tentative d'intrusion dans l'espace du noyau. Cet hyperviseur s'est inspiré des travaux de Invisible Things Lab sur Blue Pill [23], un *rootkit*-hyperviseur de type 1. BMCS [20] est un hyperviseur similaire à Hytux, pour linux, mais implémenté au dessus de KVM. Ramooflax [12,13] est un hyperviseur de type 1 qui n'est pas directement orienté sécurité. Il est compatible Intel et AMD et prend en compte les dernières extensions de virtualisation. Il implémente un outil de surveillance à distance des machines virtuelles au travers d'un *stub* gdb. HIMA [3], est un outil générique d'analyse de l'intégrité de machines virtuelles, indépendant de l'OS virtualisé, basé sur Xen. Il surveille les appels système des machines virtuelles grâce aux registres de débogage du processeur. Il maintient de plus la cartographie des processus d'un système d'exploitation, en interceptant certains appels systèmes importants tels que *mmap* et *execve*, et en calculant l'empreinte mémoire des pages chargées. Il utilise notamment le bit *NX* des tables de pages pour intercepter l'exécution de pages non exécutables.

D'autres hyperviseurs sont spécialisés dans leur propre protection. C'est le cas de SIMA [24] qui est basé sur des capteurs intégrés dans les systèmes d'exploitation virtualisés. Ils servent à extraire des informations pertinentes pour évaluer les intentions des systèmes invités. Les trois objectifs de cet hyperviseur sont de surveiller activement le logiciel invité, détecter des *malwares* et être furtif. Le vecteur d'attaque considéré par SIMA est donc du code exécuté depuis les machines virtuelles. HyperSafe [27] adopte le même principe que HIMA pour la protection de son espace mémoire, mais propose une fonctionnalité supplémentaire basée sur le contrôle de son propre flot d'exécution. Lors de la compilation, son code est statiquement instrumenté au niveau des points d'entrée pour permettre le contrôle, lors de l'exécution, de l'enchaînement de ces points d'entrée. TinyChecker [25], propose une approche basée sur la virtualisation imbriquée pour protéger les données critiques des machines virtuelles, détecter et réparer les fautes de l'hyperviseur virtualisé et enregistrer les communications entre les machines virtuelles et l'hyperviseur virtualisé. Il s'appuye sur Xen pour son implémentation.

Enfin, il existe d'autres types d'hyperviseurs, qui n'ont pas pour but premier de vérifier à l'exécution l'intégrité d'un code (que ce soit un hyperviseur virtualisé, un noyau ou encore une application). Hirano et al. [15,14] proposent d'utiliser un hyperviseur comme *Trusted Computing Base* dans le sens où il va démarrer dans un premier temps un système d'exploitation capable de lire une carte à puce pour authentifier un utilisateur et ainsi l'autoriser à charger un système d'exploitation cible dans un deuxième temps. Cette solution est basée sur BitVisor [1], un hyperviseur léger gérant une seule machine virtuelle. BitVisor est aussi utilisé dans un autre projet, TCvisor [21], qui propose un stockage physique sécurisé grâce à l'hyperviseur et à des pilotes de type *para-passthrough*.

Hypercheck [26], est un outil de tests d'intégrité basé sur une approche différente des précédentes solutions. Il ne se présente pas sous la forme d'un hyperviseur. Il s'agit plutôt d'un composant intégré au mode de gestion de la machine (*System Management Mode* – SMM). Le mode SMM est un mode du processeur distinct des modes d'exécution classiques et il dispose d'un accès total à la machine. En principe, c'est le *BIOS* qui enrichit le code de ce mode en y insérant une partie de son propre code ainsi que le code embarqué dans les *ROM* des composants matériels. Donc, pour permettre à Hypercheck d'être intégré à ce mode, il est nécessaire de l'installer, en l'occurrence, dans la *ROM* d'une carte réseau.

Il existe d'autres projets destinés à controler l'intégrité des composants matériels d'un système, à distance. Par exemple, VIPER [17] est un

outil de test de l'intégrité de *firmwares* de périphériques au travers de défis cryptograhiques. Dans ce cas, le périphérique est mis au défi par le processeur et ce sont deux composants distincts. Un autre exemple de tests à distance a été présenté par Deswarte et al. [9] pour tester l'intégrité de serveurs *web*. Cette solution s'appuye sur la réplication des serveurs et sur des défis cryptographiques basés sur le calcul de l'empreinte de fichiers statiques (code, fichiers de style, etc). Les défis sont envoyés depuis des machines de vérification dédiées.

Cette section a présenté plusieurs approches pour la sécurité. Soit ces approches s'exécutent sur le même processeur que le code de l'attaquant et ne peuvent plus rendre leur service en cas de compromission réussie de l'hyperviseur, soit elles vérifient l'intégrité des *firmwares* des périphériques. A notre connaissance, il n'existe pas d'approche qui teste l'intégrité d'un hyperviseur sans avoir recours à l'exécution d'un code par le processeur. La solution que nous présentons dans cet article est destinée à rendre un tel service de sécurité et, en ce sens, elle nous semble originale.

3 Rappels techniques sur X86 et la Virtualisation

Afin de mieux appréhender les détails concernant la caractérisation de l'intégrité d'un hyperviseur, il est nécessaire de faire un point sur les composants de base d'un ordinateur. En particulier, cette section décrit les concepts fondamentaux des technologies liées à la virtualisation dans l'architecture x86. Aujourd'hui, la grande majorité des gestionnaires de machines virtuelles utilisent les extensions de virtualisation disponibles sur les processeurs Intel (VT-x) ou sur les processeurs AMD (AMD-V). Ces deux technologies sont équivalentes et proposent les mêmes fonctionnalités. Nous avons arbitrairement choisi de mener nos expérimentations sur les extensions de virtualisation Intel VT-x. La suite de cette section présente donc les composants de ces processeurs.

Cette section est structurée de la manière suivante. Dans une première partie, nous présentons les composants de base d'une architecture Intel x86 (mémoire, cache, modes et PCI). La seconde partie est consacrée à la présentation des mécanismes matériels dédiés à la virtualisation. La dernière partie présente le mode de gestion du système (mode SMM).

3.1 Fonctionnalités de base

Modes Les cœurs d'un processeur x86 peuvent s'exécuter selon plusieurs modes : 1) le mode réel, 2) le mode protégé, ou le mode long. Pour des

raisons historiques et de rétro-compatibilité, un cœur débute systématiquement son exécution en mode réel. Il dispose d'un espace d'adressage segmenté de 20 bits, d'un accès logiciel illimité à la mémoire, aux entrées/-sorties ainsi qu'aux périphériques. Le mode protégé, quant à lui, propose un espace d'adressage physique plus important, de 32 bits, et se base sur l'utilisation d'une unité de gestion de mémoire (MMU – figure 1). La MMU permet d'organiser la mémoire en utilisant la segmentation et la pagination. Chaque partie de la mémoire (segment ou page) est protégée via l'utilisation de niveaux de privilèges appelés *rings* (de 0 à 3) ou encore via l'utilisation de droits d'accès. Enfin, le mode long permet d'accéder aux registres généraux de 64 bits et à un espace d'adressage encore plus important.

Virtualisation de la mémoire Le mode protégé et le mode long supportent la virtualisation de la mémoire via les mécanismes de segmentation et de pagination. La traduction d'une adresse virtuelle en adresse physique s'effectue en deux étapes. La mémoire virtuelle est tout d'abord découpée en plusieurs segments, par exemple le segment de données, le segment de code et le segment de la pile. Ce découpage est assuré par l'unité de segmentation. Cette unité permet donc de traduire une adresse virtuelle en adresse linéaire (cf. figure 2). Ces segments sont à leur tour réorganisés en pages qui ne sont pas forcément contiguës [1]. Ce découpage est assuré par l'unité de pagination. Cette unité permet donc de traduire une adresse linéaire en adresse physique (cf. figure 3). L'enchaînement de ces deux mécanismes est présenté à la figure 1.

Segmentation Les segments sont déclarés dans une table de descripteurs nommée la *Global Descriptor Table* (GDT) pointée par le registre du processeur *gdtptr*. Ces descripteurs indiquent le niveau de privilège requis pour accéder au segment ainsi que sa position dans la mémoire. L'association entre le segment et le descripteur se fait par le biais des sélecteurs de segments, *ds* pour les données et *cs* pour le code par exemple. Une addresse virtuelle est donc composée du couple *segment:offset*. Le niveau de privilège courant (CPL) est contenu dans les deux premiers bits de *cs* (figure 2).

1. Lorsque les adresses physiques (après traduction) sont égales aux adresses virtuelles correspondantes, pour tout l'espace d'adressage virtuel, on parle d'*identity mapping*.

Pagination La traduction d'une page [2] de la mémoire linéaire en adresse physique se fait par le biais d'une table avec 4 niveaux de profondeurs (pour le mode long) pointée par le registre du processeur *cr3*. Le parcours de cette table se fait par le biais d'indices obtenus par le découpage de l'adresse linéaire en 5 élements (cf. figure 3). Le registre *cr3* pointe donc sur la première table, la PML4. Le premier indice de l'adresse linéaire permet de récupérer l'entrée correspondante de cette table, qui pointe à son tour sur une seconde table, la PDPT. La traduction se poursuit en utilisant les autres tables, la PD et la PT, pour ainsi obtenir l'adresse physique de la page. Le dernier indice correspond à l'offset dans cette page. A chacune des entrées de ces tables, un bit U/S est disponible pour autoriser ou non l'accès à cette zone mémoire uniquement au superviseur, autrement dit au code s'exécutant avec un CPL de 0 ou ring 0. Aujourd'hui, dans les systèmes d'exploitation modernes, la segmentation est dite "à plat" et seule la pagination est réellement utilisée. Le CPL est tout de même utilisé pour séparer l'espace noyau, ring 0, de l'espace utilisateur, ring 3.

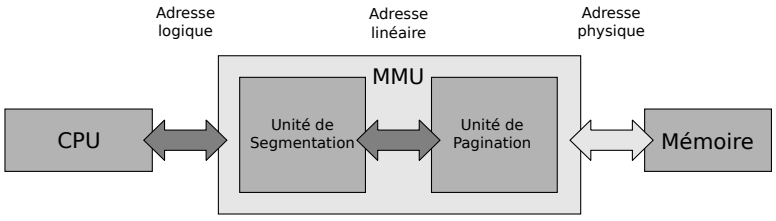

Figure 1. Memory Management Unit

Peripheral Component Interconnect (PCI) Le PCI est un bus permettant de connecter au système des périphériques tels qu'une carte réseau ou une carte graphique. Il a été créé pour permettre une communication haute performance. De plus, la conception de ce bus permet à ces périphériques de dialoguer directement ensemble sans passer par le processeur. La spécification PCI apporte un espace de configuration séparé de manière à ce qu'il soit possible de configurer les périphériques PCI depuis le processeur. Tous les périphériques doivent proposer 256 octets de registres de configuration et les systèmes doivent proposer un accès à cet espace de configuration (il s'agit du travail du *Host Bridge*). Les périphériques sont adressés de la manière suivante : `bus:device:function`. La venue du PCI

2. Espace mémoire d'une taille de 4 Ko (resp. 2 Mo et 1 Go) aligné sur 4 Ko (resp. 2 Mo et 1 Go.

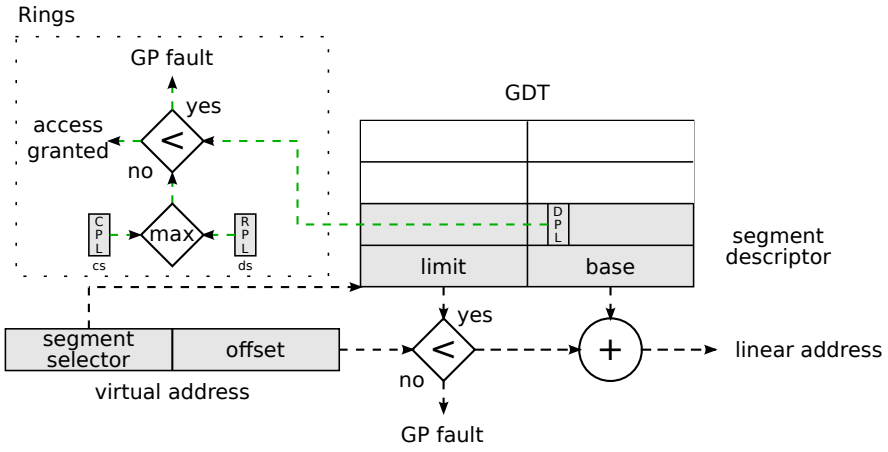

Figure 2. Traduction d'adresse logique en adresse linéaire par l'unité de segmentation

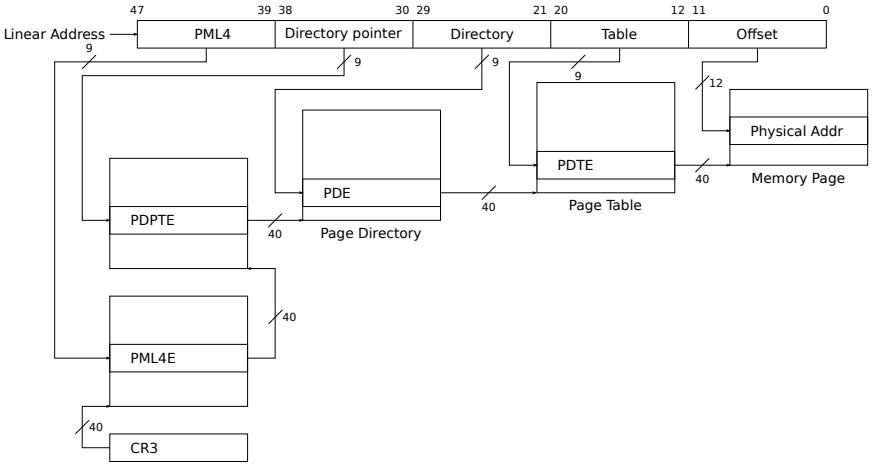

Figure 3. Pagination en mode long et pages de 4Ko

Express étend, entres autres, l'espace de configuration de 256 octets par périphérique à 4 Ko. Du point de vue du processeur, deux modes d'accès à ces registres sont disponibles : PIO (PCI I/O) ou MMIO (*Memory Mapped I/O*). Les accès PIO se font via les instruction assembleur `in` et `out`. Par contre, seuls les 256 premiers octets sont accessibles via ce mode. Avec les accès MMIO, tous les registres de configuration sont *mappés* en mémoire. L'accès à ces registres se fait donc via les instructions de manipulation de la mémoire (telles que `mov`, etc.). Lors de ces accès, le controleur mémoire route les accès vers le *chipset* plutôt que vers la mémoire centrale.

Gestion des caches Lors de l'éxécution d'un logiciel, le processeur réalise de nombreux accès à la mémoire. La vitesse de ces accès est largement moins élevée que la vitesse de fonctionnement du processeur. Aussi, pour

éviter de "sous-charger" le processeur, des mémoires intermédiaires (appelées mémoires cache), fonctionnant à des vitesses proches de celle du processeur, sont utilisées pour stocker des copies des informations de la mémoire. Ce mécanisme, intégré dans les processeurs, nécessite une synchronisation régulière entre le contenu du cache et la mémoire centrale. Le type de synchronisation pour différentes plages d'adresses mémoires peut être configuré via les registres spécifiques du processeurs : les MTRR (*Memory Type Range Registers*), ou une extension de la pagination, PAT (*Page Attribute Table*). des temps de traitement les plus rapides. Or, cette approche peut conduire à des situations d'incohérences. En particulier, la mémoire MMIO correspondant aux périphériques PCI ne doit pas être associée à une politique de cache telle que *Write Back*, pour laquelle le contenu du cache n'est écrit en mémoire que lorsque ce cache est vidé. Effectivement, le périphérique pourrait être placé dans un état incohérent qui pourrait entraîner la défaillance de la machine. Aussi, dans ce cas particulier, il est nécessaire qu'une écriture sur cette zone soit directement répercutée sur les périphériques concernés avec une politique *Uncachable* ou *Write Trough* par exemple.

3.2 VT-x

Les extensions de virtualisation Intel apportent le support matériel pour la création de la gestion des machines virtuelles. L'activation de la virtualisation nécessite la distinction de deux composants logiciels : le *host software* (que nous nommerons hyperviseur par la suite, ou *Virtual Machine Monitor* (VMM)), qui crée et gère les machines virtuelles ; le *guest software* (que nous nommerons machine virtuelle, ou *VM*) qui correspond à la machine virtuelle. L'hyperviseur peut créer des processeurs virtuels et donner accès à ces processeurs virtuels aux machines virtuelles. Une machine virtuelle est donc composée d'un logiciel invité et s'exécute dans un mode moins privilégié que l'hyperviseur.

Le support du processeur pour la virtualisation est fourni par deux modes de fonctionnement appelés *VMX root operations* et *VMX non root operations*. Le premier est le mode d'exécution de l'hyperviseur et le second est le mode d'exécution des machines virtuelles (mode moins privilégié). Les transitions entre ces deux modes sont appelées des *VMX transitions*. Une *VMX transition* depuis l'hyperviseur vers une machine virtuelle est nommée *VM entry* et *VM exit* dans le sens opposé [7] (figure 4).

L'hyperviseur exerce son contrôle sur les machines virtuelles en configurant un cœur virtuel. Cette configuration est effectuée dans des structures appelées VMCS (*Virtual Machine Control Structure*). Dans le cas de

plusieurs machines virtuelles, chacune doit posséder sa propre VMCS. La configuration de ces VMCS est effectuée depuis le mode *VMX root operations* avec les instructions `vmread` et `vmwrite`. Une machine virtuelle peut commencer ou poursuivre son exécution lorsque l'hyperviseur exécute l'instruction `vmlaunch` ou `vmresume` et l'ordonnancement entre plusieurs machines virtuelles peut se faire via l'instruction `vmptrld` qui active une VMCS à la fois.

VMCS Une VMCS est une structure interne au microprocesseur, permettant de configurer un cœur virtuel. Il s'agit d'une zone de 4 Ko 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. Il ne peut donc y avoir qu'une seule VMCS active par cœur logique. L'intérêt de la virtualisation étant de pouvoir exécuter de nombreuses machines virtuelles, une copie des VMCS correspondantes doit être présente en mémoire principale. Ces copies sont appelées *VMCS region* et occupent également 4 Ko. Elles sont rapatriées au sein du processeur via les instructions `vmptrld`, par exemple.

Les deux zones des VMCS réservées aux sauvegardes et aux restaurations des états des machines virtuelles et de l'hyperviseurs sont, respectivement, *Guest-state area* et *Host-state area*. Ces zones sont a priori mises à jour 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 VMS courante et l'état de la VMCS region, qui se situe dans la mémoire centrale. Ce point étant important pour les tests d'intégrité, il sera abordé dans une section ultérieure.

Tables de pages étendues Afin de protéger l'espace mémoire de l'hyperviseur et des machines virtuelles, il est nécessaire que le résultat de la traduction d'une adresse virtuelle d'une machine virtuelle par la MMU du processeur virtuel soit à son tour traduite pour assurer la séparation en mémoire des différents espaces mémoire. Initialement, cette traduction était assurée logiciellement avec les *shadow page tables* [12]. Dans les dernières extensions de virtualisation, les processeurs Intel proposent un

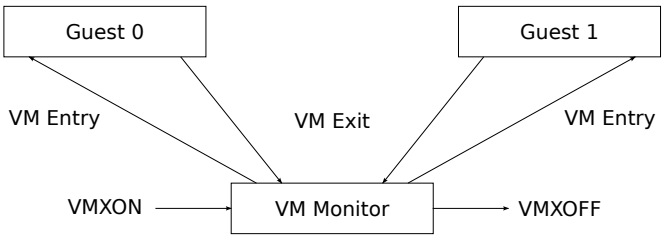

Figure 4. Extensions de virtualisation

mécanisme de pagination étendue, remplaçant les *shadow page tables*. Il s'agit de *Extended Page Table* ou EPT. Cette extension permet de définir des tables de pages permettant de traduire une adresse physique d'une machine virtuelle, appelée *guest physical address*, en *physical address* (figure 5). La structure de EPT et son mécanisme de traduction d'adresse sont similaires à ceux de la pagination (figure 3). EPT permet aussi de redéfinir la politique de cache des MTRRs utilisés par le machine virtuelle mais conserve l'influence de PAT par défault.

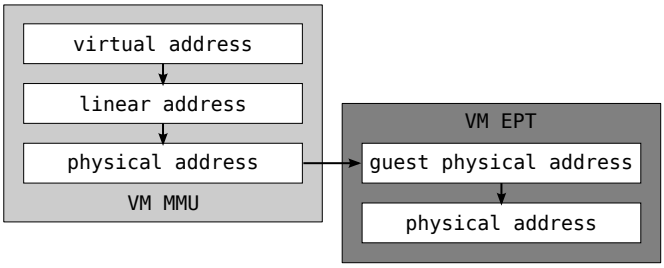

Figure 5. Pagination étendue avec EPT

Autres extensions Les mécanismes de virtualisation s'enrichissent régulièrement. Les *shadow* VMCS ajoutent le support matériel pour la création d'hyperviseurs imbriqués. Notons également que les technologies Intel VT-d apportent l'IOMMU qui virtualise et protège les accès à la mémoire principale depuis les périphériques. Enfin, le *Single Root I/O virtualization*, permet le support de la virtualisation des périphériques par les périphériques eux-mêmes (si cette technologie est supportée).

3.3 System Management Mode (SMM)

Le mode de gestion du système (*System Management Mode* – SMM) a été introduit pour la première fois avec le processeur Intel 386SL. Ce

mode est dédié à la gestion des évènements liés au matériel. Il permet aux systèmes d'exploitation d'être développés sans avoir une connaissance précise de l'implémentation matérielle. Ce mode ne dispose d'aucun mécanisme de sécurité étant donné que le code est développé uniquement par les concepteurs de la carte mère et protégé par le controleur mémoire avant le chargement d'un système d'exploitation. Les logiciels s'exécutant dans ce mode jouissent d'un accès total à l'espace mémoire dans la limite des 4 Go. Ils disposent aussi d'un accès illimité à l'espace mémoire des périphériques en MMIO et PIO [10].

Le mode SMM dispose d'un espace mémoire dédié, la SMRAM. Cet espace mémoire débute à l'adresse pointée par le registre SMBASE (qui est un MSR particulier) du processeur. Ce MSR est lisible uniquement depuis le mode SMM. Le BIOS doit veiller à ce que la SMRAM soit verrouillée avant le chargement du système d'exploitation [8]. Le verrouillage du mode SMM est indispensable pour des raisons de sécurité, rendant la totalité de la machine vulnérable dans le cas contraire [11]. Aujourd'hui, la grande majorité des BIOS verrouillent correctement l'accès à la SMRAM.

3.4 Conclusion

Cette section fournit un résumé des fonctionnalités importantes pour la mise en place d'un hyperviseur. Les mécanismes de gestion et de protection de la mémoire sont présentés, avec les deux types d'accès aux mémoires associées aux composants PCI. Les mécanismes de virtualisation sont également présentés. Ces différentes notions ont été mises en pratique dans le cadre du développement de notre hyperviseur. La présentation de ce développement fait l'objet du chapitre suivant.

4 Hyperviseur de sécurité

L'hyperviseur de sécurité à tester à distance est simple. Son objectif est de virtualiser la couche supérieure dans une seule machine virtuelle correctement isolée par EPT. Il se protège aussi des attaques menées indirectement au travers de périphériques malveillants en contrôlant l'espace PIO et MMIO ainsi qu'en configurant l'IOMMU. Cet hyperviseur se limite à l'implémentation de ses objectifs restreints, la taille de son code demeure donc modeste, le rendant facilement vérifiable, c'est pour cela que la confiance que l'on peut avoir en lui est élevée. De plus sa conception restera figée, il ne configurera en effet qu'une seule VMCS et n'aura qu'une seule *VMCS region*, qu'un seul contexte mémoire en identity mapping,

donc un seul jeu de tables de pages et une seule GDT, enfin un seul jeu de tables EPT.

Comme expliqué dans la section 5, l'hyperviseur se présente sous la forme d'une image UEFI 64-bits. Nous sommes chargés par le *loader* du *firmware* directement en mode long, l'écriture d'un loader PE32+, format d'image UEFI, n'est donc pas nécessaire. Comme le montre l'image 6, notre stratégie de chargement est très simple et s'appuye sur le firmware uniquement jusqu'à la fin de la phase de *pre-boot* [2]. Le loader UEFI nous donne la main, nous configurons le contexte mémoire de notre *host*, activons *VMX operations*, initialisons la VMCS de la VM. Nous renseignons le rip du *handler* de *VM Exit*, et recopions l'état du cœur courant dans la partie *guest state*, En effet, une fois installés comme VMM, nous rendons la main au *firmware* dans une machine virtuelle contrôlée par nos soins, de manière à profiter de services implémentés dans celui-ci pour la suite du démarrage.

Il nous reste à marquer notre espace mémoire comme réservé aux yeux du *guest*. Dans les hyperviseurs s'appuyant sur les *firwmares* historiques, les BIOS, il était obligatoire de modifier la méthode de récupération du mapping mémoire, service délivré par l'interruption 15, méthode e820. Le mapping est retourné sous la forme de structures appelées *System Memory MAP* ou SMAP. Aujourd'hui le firmware UEFI s'en charge pour nous si l'image chargée est un driver de *runtime*, ce que nous avons choisi pour notre hyperviseur de sécurité.

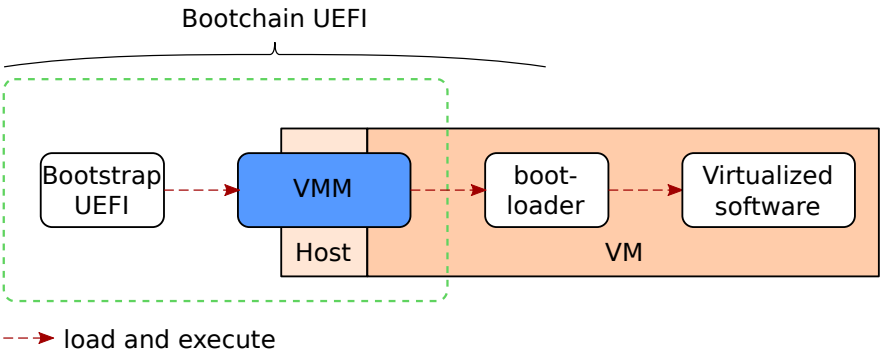

Figure 6. Séquence de démarrage de l'hyperviseur de sécurité UEFI bare metal

Nous connaissons l'architecture de notre hyperviseur de sécurité. Il reste maintenant à extraire quelles sont les parties de son espace mémoire, de sa configuration, caractéristiques de son intégrité.

5 Caractérisation de l'intégrité de l'hyperviseur de sécurité

Dans cette section, nous présentons les éléments dont la modification impromptue reflète une compromission de l'hyperviseur. Ces éléments correspondent au code de l'hyperviseur et à sa configuration. Le code est stocké en mémoire centrale et dans les caches du processeur. Dans cette étude, nous nous concentrons sur le contenu de la mémoire centrale. La configuration de l'hyperviseur est constituée des VMCS, des tables de EPT (figure 7). Nous détaillons dans la suite ces différents éléments.

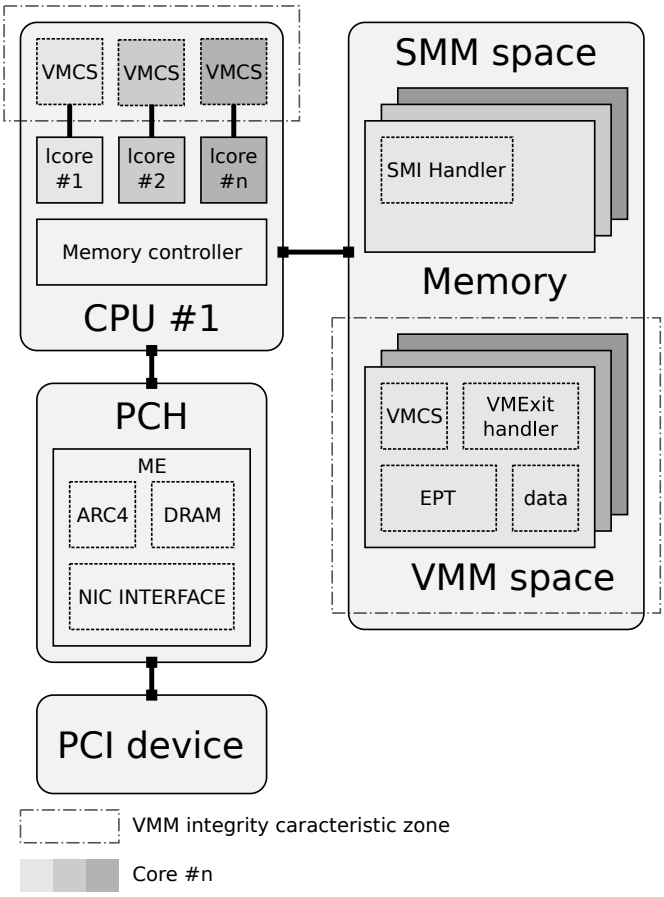

Figure 7. Zones caractéristiques de l'intégrité de l'hyperviseur pour les processeurs Intel de 4$^{\text{ième}}$ génération

5.1 Intégrité du point de vue de la mémoire

Rappelons que l'hyperviseur est chargé en tant que pilote UEFI. À ce titre, l'espace mémoire occupé par le code est contigu en mémoire centrale.

Cet espace mémoire n'a aucune raison d'être modifié durant l'exécution. Aussi, une modification de cet espace mémoire est caractéristique d'une compromission. Il s'agit donc d'une des vérifications à réaliser. Les données utilisées par notre hyperviseur sont aussi stockées en mémoire. Certaines de ces données possèdent des valeurs fixes (par exemple, l'adresse de EPT). Une vérification de ces valeurs est également nécessaire. Par contre, il existe certaines données de l'hyperviseur dont les valeurs peuvent évoluer durant l'exécution. Il est important de vérifier aussi ces valeurs en réalisant un test de vraissemblance. À ce niveau, le test employé doit être dépendant de la donnée elle-même. À titre d'exemple, nous pouvons imaginer que la valeur de rip d'une VM doit toujours se situer dans les zones de mémoire qui ne sont pas marquées comme étant à protéger par le bios UEFI, la même remarque peut s'appliquer pour le rip de l'hyperviseur, qui ne doit pas exécuter du code *VM land*.

Toutefois, un composant malveillant pourrait tout à fait profiter d'une vulnérabilite lui permettant de "déplacer" l'hyperviseur dans une autre zone mémoire tout en laissant intacte la zone de code initiale. Donc, il n'est pas suffisant de se contenter de vérifier les données et le code relatifs directement à l'hyperviseur. Autrement dit, la routine de vérification d'intégrite doit analyser l'intégralite de la mémoire à la recherche de zones de code ou de données qui sont caractéristiques de valeurs manipulées par un hyperviseur (exactement comme un IDS basé sur des signatures identifie des attaques sur un reseau). Bien entendu, cette analyse doit être réalisée avec soin de manière à ne pas engorger le bus mémoire et ainsi pénaliser les VMs au niveau des accès. L'idée retenue vise à analyser la mémoire par morceaux à des moments différents. L'hypothèse forte est que l'attaquant doit réaliser un effort trop important pour anticiper sur ces analyses et cet effort important aura un impact tout aussi important sur les performances de la machine (et sera donc détecté).

5.2 VMCS region

Comme vu dans le section 3, les VMCS sont des structures internes au processeur qui vont être copiées dans la mémoire à une fréquence non connue, dépendante de l'occurence des *VMX transitions* et des `vmread` et `vmwrite`. La zone mémoire où les VMCS sont sauvegardées se nomme la *VMCS region*. Les VMCS conservent l'état du cœur de l'*host*, du *guest*, le pointeur EPT et les contrôles d'exécution. Ces données sont donc présentes dans la mémoire. La fréquence de mise à jour des champs étant ni connue, ni déterministe, nous devons travailler en tenant en compte de l'obsolescence des données. Une action malveillante ne sera écrite que

plus tard en mémoire, d'autant plus tard que la politique de cache de la VMCS region est *Write Back*. Les modifications seront effectives en RAM un certain temps après l'écriture initiale par le processeur, en fonction de l'utilisation du cache. Cet aspect est important si l'accès s'effectue en dehors du contexte du cœur propriétaire du cache modifié, un accès DMA par exemple.

La configuration de l'hyperviseur de sécurité 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*. Le tableau 1 liste les champs d'une VMCS region, représentatifs de l'intégrité d'un hyperviseur.

Nom	**Description**
Configuration de l'hyperviseur de sécurité	
Virtualisation de la mémoire	
es, cs, ss, ds, fs, gs, tr	Valeur des sélecteurs de segment
cr3	Contexte mémoire, pointeur de PML4
fs, gs, tr base	Cache de la base des segments
gdtr base	Pointeur de GDT
IA32_PAT	Gestion du cache dans les tables de pages
VM Exit handler	
rsp	Pile du VMM lors d'un VM Exit
rip	Localistation du VM Exit handler
Mode d'exécution	
IA32_EFER	LME, LMA relatifs au *long mode*
IA32_PERF_GLOBAL_CTRL	Permet d'activer les compteurs de performance
cr0	PE, PG, pagination, contrôle du mode
cr4	VMXE, bit de virtualisation activée, PAE
Gestion d'interruptions	
idtr base	Pointeur d'IDT
Contrôle de la VM	
Address of I/O bitmap A/B	Contrôle d'accès à l'espace des entrées/sorties
Address of MSR bitmaps	Contrôle d'accès aux MSR
EPT pointer (EPTP)	Adresse des tables EPT
Pin-based VM-execution controls	Contrôle de certaines interruptions
Processor-based VM-execution controls	Contrôle certains évènements, chargement du cr3
Exception bitmap	Permet d'être notifié de certaines exceptions
cr0, cr4 guest/host mask	Contrôle l'écriture d'un bit du cr0, cr4 donné
cr0, cr4 read shadow	cr0, cr4 lu par le *guest*

Table 1. Champs de la VMCS region représentatifs de l'intégrité de l'hyperviseur

5.3 Espace de configuration PCI

Les hyperviseurs s'appuient sur les composants PCI pour étendre leur fonctionnalités ou communiquer avec le monde extérieur. On pense à un contrôleur RS232 pour les communications séries, qu'utilise Ramooflax ou encore une carte Ethernet pour communiquer avec une machine de confiance. Si le VMM se repose sur un système de fichiers stocké sur disque, il configure le contrôleur IDE, SATA ou USB. Les registres de configuration de ces composants PCI s'ajoutent à la configuration de l'hyperviseur et sont caractéristiques de son intégrité. À titre d'exemple, un système d'exploitation linux auquel on retire le périphérique hôte de son SquashFS ne survit pas.

Les registres de configuration de ces périphériques configurés par le processeur font partie de l'espace de configuration de type 0. Ils sont disponibles en mémoire depuis le processeur via PIO et MMIO. On peut aussi y accéder depuis les périphériques via des requètes de configuration de type 0.

L'IOMMU s'active via l'espace de configuration PCI. Sa configuration est donc lisible et vérifiable. Nous ne prévoyons pas pour l'instant d'activer l'IOMMU car elle pourrait masquer l'accès aux zones mémoires de la configuration de l'hyperviseur. Si un logiciel ou périphérique malveillant la configure, nous pourrons le vérifier via l'espace de configuration PCI depuis le processeur ou depuis le matériel. Il est difficile de savoir de façon générale si l'espace de configuration de l'IOMMU est lisible depuis un périphérique PCI. Il est probable que cet accès dépende de la plateforme matérielle utilisée et il est donc nécessaire de s'en assurer lors des expérimentations. Un autre moyen de détecter sa présence est lié au fait qu'en la configurant, l'attaquant nous donne une vision de la mémoire physique réduite, il est par exemple possible de le détecter en analysant les similitudes entre les pages lues.

5.4 Tests d'intégrité en fonction du logiciel virtualisé

Quel que soit le moyen d'accès à la configuration de l'hyperviseur, le comportement du logiciel de la machine virtuelle a un impact sur la prise de décision du protocole de tests d'intégrité. En effet, un système d'exploitation simple ne crée pas tout à fait les mêmes structures qu'un hyperviseur (cf. section 3, un hyperviseur virtuel va par exemple créer et configurer des VMCS dans son espace mémoire, s'ajoutant à celles de notre hyperviseur de sécurité de manière tout à fait légitime). Les décisions

concernant nos tests d'intégrité devront donc s'adapter au type de logiciel virtualisé.

Dans la section suivante, nous allons décrire quels sont les moyens d'accès à la configuration de l'hyperviseur et choisir la plus pertinante.

6 Méthode d'accès à la configuration de notre hyperviseur

La section précédente nous a permis de caractériser l'intégrité d'un hyperviseur. Il est maintenant nécessaire d'étudier comment pouvoir accéder à sa configuration. Il existe pour cela différentes techniques possibles. Nous les présentons et justifions le choix de la technique que nous utilisons finalement.

6.1 Accès via le mode de management : SMM

Le mode SMM est un mode intéressant pour effectuer des tests d'intégrité. Suffisamment isolé du reste du monde dans la SMRAM, nous pouvons l'utiliser pour envoyer des défis à l'hyperviseur ou encore lire une partie de sa configuration, sans les VMCS [26]. Le problème du mode de management classique est que, lorsqu'une SMI apparaît dans le guest, le processeur désactive VMX operations avant d'entrer dans le mode SMM, nous interdisant matériellement l'accès aux VMCS. Mais il est possible d'aller plus loin, lire les VMCS internes au microprocesseur en activant le *dual monitor treatment* [7]. Lorsque ce mode est activé, en plus du VMM, un deuxième hyperviseur, le *SMM Transfer Monitor* ou STM, est disponible dans le mode de management. Les SMI survenues dans le guest sont transférées au STM, s'exécutant aussi en VMX root operations. Le STM possède un pointeur de VMCS supplémentaire, propre à son fonctionnement, *SMM-transfer VMCS pointer*. La transfer VMCS courante contient un pointeur vers la VMCS region du guest ou la vmxon VMCS region, contenant toutes les deux la configuration matérielle du VMM. À l'aide de cette méthode, nous pouvons donc installer du code en mode SMM capable de lire la VMCS region pour en tester l'intégrité.

Cependant, une vulnérabilité majeure a été découverte simultanément par Duflot, Levillain [10] et Wojtczuk, Rutkowska [28] concernant la protection de la SMRAM par le MCH, ancien northbridge. Le problème venait de la séparation de la vérification d'une lecture / écriture effective dans la SMRAM vis à vis de celles effectuées dans le cache. Il suffisait de rendre cachable une zone mémoire correspondant à la SMRAM (politique

Write Back) pour suffisamment retarder l'écriture effective de code dans celle-ci jusqu'à la prochaine SMI, ainsi le CPU en mode SMM exécutait le code des lignes de caches, non vérifié par le MCH. Cette vulnérabilité a été corrigée depuis, mais elle prouve encore une fois la nécessité de se séparer le plus possible du CPU pour réaliser des tests d'intégrité.

Les *firmwares* UEFI offrent aujourd'hui un service d'installation d'une image UEFI en tant qu'handler de SMI. Nous avons malheureusement éprouvé des difficultés d'utilisation de ce service qui semble non implémenté sur les firmwares dont nous disposons. Une dernière solution serait d'utiliser l'Expansion ROM d'un device PCI Express afin d'y installer notre handler et une routine de chargement [5]. Nous espérons ainsi être chargés suffisamment tôt par le *firmware* pour que la SMRAM ne soit pas encore verrouillée.

Cette solution, même si elle semble séduisante n'est pas celle que nous avons retenue, à la fois parce que des vulnérabilités récentes ont été identifiés dans le mode SMM et que nous avons éprouvé pour le moment des difficultés d'utilisation.

6.2 Accès via Intel AMT

Active Management Technology ou AMT est un processeur indépendant situé premièrement dans le Northbridge ou MCH, puis aujourd'hui dans le PCH [6]. Son architecture est de type ARC4 RISC, il dispose d'un accès à une plage dédiée de la mémoire principale DRAM et à une interface spéciale avec la carte réseau. Son environnement d'exécution est appelé Management Engine ou ME [29]. Ce CPU est capable d'exécuter des programmes en parallèle du processeur principal. Le programme de ce processeur est sauvegardé dans la mémoire *flash SPI*, protégée contre les attaques de *reflash*. Enfin, il peut accéder à la mémoire du CPU principal en DMA. Les applications de management Intel sont exécutées sur l'OS Temps réel Nucleus.

AMT est un outil intéressant pour accéder à la mémoire principale et lire une partie de la configuration de l'hyperviseur. Il ne dispose néanmoins d'aucun moyen de lire les registres du processeur, e.g. les VMCS.

Dans notre cas, AMT peut exécuter notre application de tests d'intégrité, ou encore s'en servir comme proxy d'accès à la mémoire principale pour la même application de tests s'exécutant sur une machine de confiance, via DMA et l'accès dédié à la carte réseau.

Mais, Tereshkin et Wojtczuk proposent une solution pour accéder à la mémoire dédiée à AMT via memory remapping sur un chipset Q35 [29]

pour y installer un rootkit, exposant une vulnérabilité exploitable pour cette technologie. Cette vulnérabilité semble aujourd'hui colmatée.

AMT est donc totalement séparé du processeur principal, ceci nous séparant de la surface d'attaque. Cependant, il ne dispose d'aucun moyen d'accès direct aux VMCS, seules les VMCS region restant lisibles, sans aucune garantie d'actualisation des données. De plus, il reste fermé, impliquant une instrumentation difficile. C'est la raison principale pour laquelle nous n'avons pas retenu cette solution.

6.3 Accès via VMX root

La solution la plus aisée pour accéder à la configuration d'un VMM est bien évidemment depuis le VMM lui-même. Nous pouvons imaginer ajouter un module de tests d'intégrité ou de lecture de configuration dans celui-ci. Cependant, si l'on suppose que le VMM lui-même peut être corrompu, il est bien évident que l'on ne peut faire confiance en l'exécution de ce test directement par le VMM. Si un attaquant compromet le VMM, il contrôle la totalité de la machine et peut aisément ignorer ces tests s'exécutant dans le même mode. Nous n'avons donc pas retenu non plus cette solution.

6.4 Accès via un périphérique PCI-Express

IronHide [18], est une solution matérielle et logicielle implémentée sur FPGA. Cette carte nous permet d'accéder à la mémoire principale en lecture et écriture via DMA.

Ces accès, totalement détachés du processeur, ne nous permettent pas de lire ses structures internes, notamment les VMCS. Nous ne pouvons pas nous assurer par une simple lecture que l'hyperviseur soit bien intègre, même si son espace mémoire et ses VMCS regions semblent toujours être en place en mémoire principale (attaque par relocalisation). Les problèmes de l'obsolescence de la mémoire liés aux VMCS et au caches, en plus de ceux liés à l'IOMMU (vus en section 6) sont à prendre en compte avec cette solution.

Néannmoins, cette solution est celle que nous avons retenue car elle ne repose sur aucun logiciel pouvant être contrôlé depuis le processeur. L'utilisation d'une carte spécifique faisant des accès direct à la mémoire sans intervention du processeur nous permet de vérifier l'intégrité de notre hyperviseur sans être soumis aux effets de bords liés à la compromission de cet hyperviseur. Un attaquant ayant pris le contrôle de notre hyperviseur aura de grandes difficultés à leurrer les tests d'intégrité s'ils sont réalisés à l'aide de cette technique.

7 Le protocole

Rappelons que l'objectif de cet article est de fournir une architecture permettant d'assurer la sécurité de la couche logicielle la plus privilégiée d'un système. Cette couche peut être soit un noyau d'un système d'exploitation soit un gestionnaire de machines virtuelles. En particulier, nous supposons qu'une compromission de cette couche logicielle implique nécessairement une atteinte à son intégrité. Notre architecture doit donc mettre en place les mécanismes nécessaires aux tests d'intégrité des éléments identifiés dans la section 5.

À l'issue de la section 6, le composant local en charge des tests d'intégrité a été identifié. Il s'agit du composant Electronical Remote Intégrity Checker ou ERIC, basé sur IronHide et développé par nos soins, qui correspond à un FPGA installé sur une carte PCIe. Bien que notre architecture implique l'ajout d'un matériel supplémentaire, nous limitons fortement le contenu de ce composant en terme de logique de fonctionnement. Par conséquent, une grande partie de la logique de fonctionnement est exportée vers une machine de confiance chargée du pilotage des tests d'intégrité à distance. Un lien de confiance est mis en place entre ERIC et la machine de confiance (cf. figure 8).

La manière dont les tests sont réalisés ne doit pas permettre à un code malveillant de les anticiper facilement tout en lui laissant de la marge pour agir. Effectivement, dans un tel cas, le code malveillant pourrait par exemple adopter une politique de migration entre différents composants qu'il peut compromettre et activer les migrations en fonction des composants en cours de test par ERIC. Ou bien, il pourrait très bien reconfigurer des composants matériels tels que l'IOMMU afin de masquer une zone mémoire aux yeux de ERIC. Autrement dit, les tests d'intégrité doivent avoir un caractère difficile à anticiper ou bien difficile à duper voire coûteux à contrer.

Aussi, nous avons choisi de réaliser un balayage aléatoire de la mémoire afin de caractériser le contenu des différentes pages mémoires. Un cycle complet de ce balayage doit permettre de tester toutes les pages mémoire une et une seule fois. Chaque page peut alors être caractérisée en fonction du type de données qu'elle contient (VMCS, PML4, etc.). Cette approche sera mise en place en adoptant des techniques proches des registres à décalage à rétroaction linaire. Quant à la caractérisation des pages elles-mêmes, elle sera réalisée à partir d'un automate dont chaque état indiquera la prochaine zone de la page mémoire à lire et les transitions signaleront les valeurs "masquées" à obtenir pour la parcourir. À titre d'exemple, le premier

état peut indiquer de récupérer les 8 premiers octets de la page et une transition en sortie de cet état peut signaler la valeur 0x1 avec le masque 0xffffffff pour indiquer qu'il y a de fortes chances qu'il s'agisse d'une VMCS. Cette transition pointe alors sur une autre partie de l'automate destinée à vérifier cette hypothèse en poursuivant les lectures et comparaisons. Le résultat des analyses servira à comptabiliser le nombre des différents types de pages (VMCS, PML4, etc.), avec éventuellement l'adresse de certaines de ces pages. Ces informations seront ensuite envoyées à la machine de confiance qui pourra alors identifier si il y a eu atteinte à l'intégrité. En résumé, cette approche vise à mettre en place une approche à base de signature pour la caractérisation de la compromission d'un hyperviseur. La majorité de ces fonctionnalités sera implémentée directement sur le FPGA pour éviter d'aboutir à des temps de traitement élevés. Tous ces tests pourront être accompagnés de tests initiés directement par la machine de confiance, par exemple dans le cas d'une investigation pour identifier plus précisément la nature de certains éléments de la machine.

Tester l'intégrité d'un composant logiciel dont le comportement est inconnu est très délicat. Pour contourner cette difficulté, nous avons également opté pour la mise en place de notre propre hyperviseur de sécurité. Cet hyperviseur minimaliste sera dédié au contrôle d'une unique machine virtuelle. Cette machine virtuelle correspondra à la couche logicielle à protéger. Il ne contiendra donc qu'un nombre limité de VMCS. Par conséquent, il devient plus facile de vérifier que notre hyperviseur n'a pas été compromis étant donné que nous connaissons son comportement. Quant aux couches logicielles de niveau supérieur, la caractérisation de leur compromission pourra être réalisée au moyen des indicateurs envoyés à la machine de confiance.

Avec cette approche, il devient alors difficile pour un attaquant d'anticiper sur la prochaine zone mémoire en cours de test. De plus, étant donné que les tests des pages mémoires sont réalisés fréquemment, une migration du code malveillant aura un impact important sur les performances générales du système et sera alors détectée. De plus, nous espérons que le déroutement des tests (via une IOMMU par exemple), aura forcément un impact sur la comptabilisation de la nature des pages mémoire.

8 Implémentation

Nous avons implémenté un prototype de périphérique PCI-Express, développé sur une carte Xilinx ml-605 hébergeant un FPGA du même constructeur, ERIC. Ce développement, qui s'appuie sur les projets Iron-

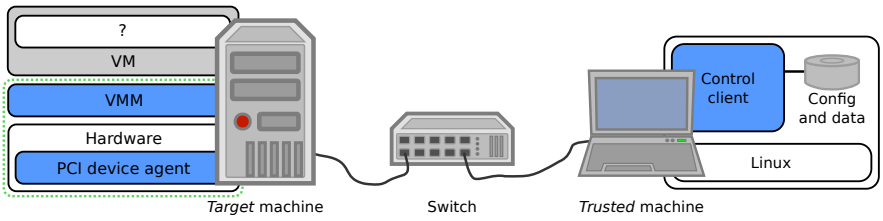

Figure 8. Architecture de l'hyperviseur de sécurité

Hide [18] et Milkimyst [4], est un System On Chip ou SOC, contenant plusieurs composants interconnectés sur un bus wishbone [19], dont un processeur Lattice Micro32 ou LM32, un contrôleur MAC Ethernet et un bridge USB to UART pour le debug (figure 9). La reconnaissance des patterns mémoires (section 5), est calculée à l'aide d'un composant dédié du SOC, Checker, développé par nos soins (figure 10).

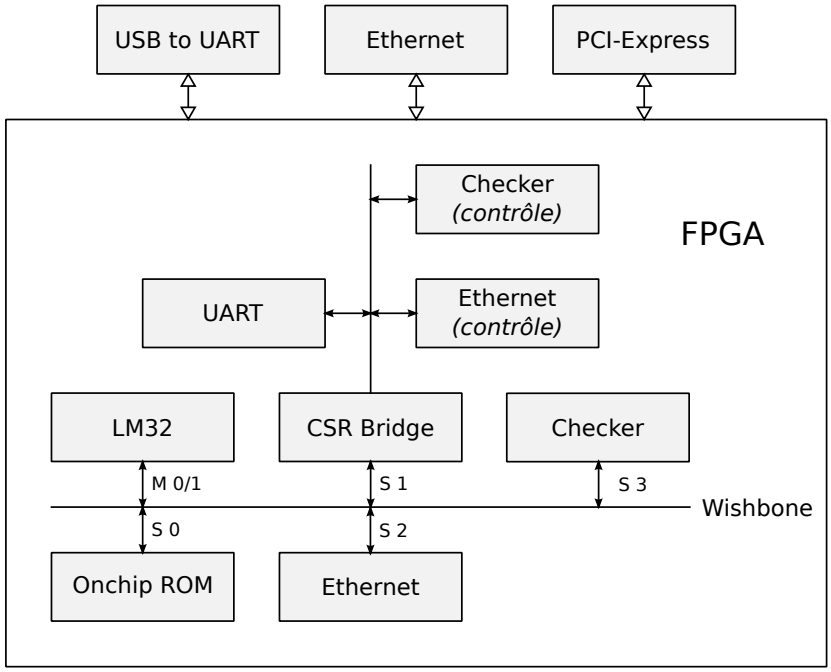

Figure 9. ERIC : System On Chip spécialisé dans l'analyse de zones mémoires

Checker embarque un processeur spécialisé dans le traitement de la mémoire, le Memory Processing Unit ou MPU capable d'exécuter un jeu d'instructions spécifique à l'analyse de données bit à bit, ainsi qu'un composant d'envoi réception de requètes de lecture mémoire hôte PCI-Express. Ce processeur est le cœur de la caractérisation des pages analysées. Il peut interrompre le CPU pour le notifier de certains évènements, comme

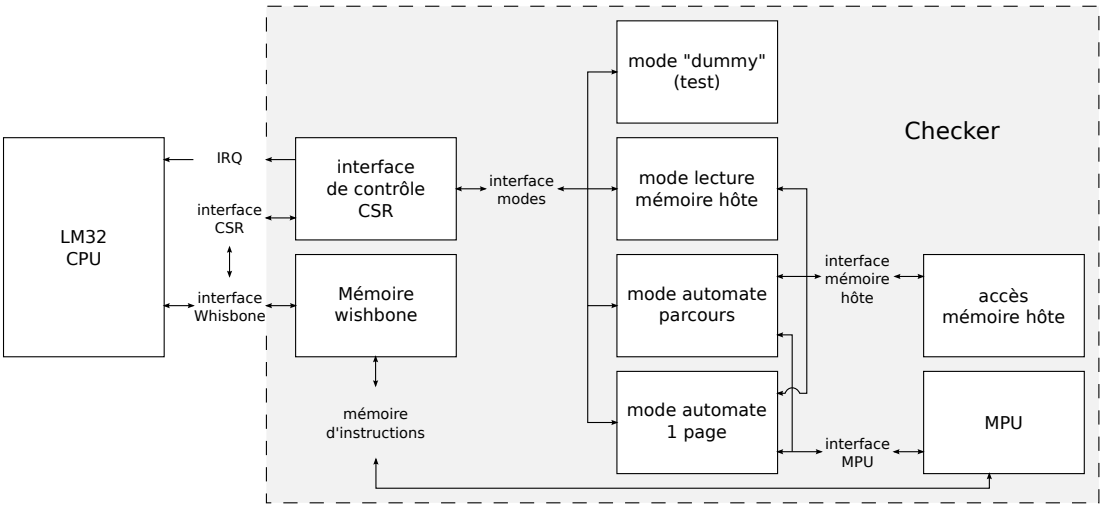

Figure 10. Checker : composant Wishbone

la détection d'une structure en mémoire. Ces évènements seront traités par le processeur principal.

Prenons l'exemple de la détection d'une table de page EPT. Une table de pages est composée de 512 entrées de 64 bits, soit une page de 4 kilo octets. En appliquant successivement un masque sur un nombre suffisant de quadruple mots, nous pouvons affirmer que la page courante contient une table EPT avec un niveau de certitude élevé. L'application de ce masque s'effectue en un seul coup d'horloge grâce à l'instruction `mask` du MPU. Pour le cas d'une machine disposant d'un espace d'adressage physique de 40 bits, les bits 51 à 40 doivent être à zero et 2 à 0 (bits rwx) doivent correspondre aux droits de la VM concernée. Nous pouvons aussi calculer la distance de hamming, grâce à une instruction dédiée du processeur, d'une plage de bits qui n'a pas de valeur connue mais dont deux entrées doivent avoir des valeurs proches. Les bits 39 à 12 de deux entrées d'une table de page en identity mapping doivent avoir une distance de hamming au plus égale à 2 car l'adresse d'une page est incrémentée de 1 à chaque entrée.

9 Conclusion

Dans cet article, nous avons proposé une méthode originale de protection d'une couche logicielle privilégiée (typiquement, un VMM). Pour cela, nous avons proposé le développement d'un petit hyperviseur de sécurité dont nous vérifions à distance l'intégrité à l'aide d'un périphérique spécifiquement dédié, pilotable à distance par une machine de confiance, qui

réalise des analyses de la mémoire périodiquement ou sur demande. Cette machine de confiance détermine l'intégrité de l'hyperviseur en fonction de l'évolution du nombre et du type de motifs trouvés en mémoire.

Nous sommes en cours d'implémentation et notamment, nous allons devoir adapter le périphérique PCI-Express actuel (ERIC) sur le FPGA que nous aurons choisi. Il sera nécessaire d'étendre ses fonctionnalités, notamment ajouter des fonctions de balayage aléatoire et complet de l'espace mémoire. La suite du développement consiste à concevoir et créer l'automate d'identification de motifs dans les pages (VMCS, table de page, données), vis à vis des expérimentations que nous aurons menées. L'utilisation de masques matériels, pour séparer les données caractéristiques d'une page, du bruit, seraient une bonne piste d'optimisation du système. Le nombre et la nature des structures identifiées devront être transférées à la machine de confiance. Celle-ci pourra, dans un client de contrôle, qualifier l'intégrité de l'hyperviseur vis à vis des données reçues, et de la configuration connue.

À plus long terme, une perspective à ce travail consiste à réutiliser le principe de cette implémentation pour développer d'autres outils de sécurité tels que des systèmes de détection d'intrusion sur le bus PCI-Express. Nous pouvons imaginer un périphérique qui capture les trames protocolaires PCI-Express pour les analyser, détecter ces intrusions et alerter un centre système de gestion des évènements de sécurité ou *SIEM* via une liaison de confiance, le cas échéant.

Références

1. Bitvisor : A single-vm lightweight hypervisor. `http://www.bitvisor.org/`.
2. Unified extensible firmware interface specification, version 2.3.1, errata c. Technical report, Unified EFI, Inc., jun 2013.
3. A.M. Azab, Peng Ning, E.C. Sezer, and Xiaolan Zhang. Hima : A hypervisor-based integrity measurement agent. In *Computer Security Applications Conference, 2009. ACSAC '09. Annual*, pages 461–470, 2009.
4. Sébastien Bourdeauducq. A performance-driven soc architecture for video synthesis, 2010.
5. Pierre Chifflier. UEFI et bootkits PCI : le danger vient d'en bas. In *Actes du 11ème symposium sur la sécurité des technologies de l'information et des communications (SSTIC)*, pages 159–190, 2013.
6. Intel Corporation. Intel AMT release 6.0 architecture. `http://software.intel.com/sites/manageability/AMT_Implementation_and_Reference_Guide/WordDocuments/intelamtrelease60architecture1.htm`.
7. 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.

8. 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-1-datasheet.pdf http://www.intel.com/content/dam/www/public/us/en/documents/datasheets/4th-gen-core-family-desktop-vol-2-datasheet.pdf.
9. Yves Deswarte, Jean-Jacques Quisquater, and Ayda Saïdane. Remote integrity checking. In Sushil Jajodia and Leon Strous, editors, *Integrity and Internal Control in Information Systems VI*, volume 140 of *IFIP International Federation for Information Processing*, pages 1–11. Springer US, 2004.
10. Loic Duflot and Olivier 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)*, pages 132–168, 2009.
11. Loïc Duflot, Daniel Etiemble, and Olivier Grumelard. Using CPU system management mode to circumvent operating system security functions, 2007.
12. Stephane Duverger. Ramooflax übervisor. In *Actes du 9ème symposium sur la sécurité des technologies de l'information et des communications (SSTIC)*, pages 195–233, 2011.
13. Stéphane Duverger. sduverger/ramooflax, a pre-boot virtualization tool. https://github.com/sduverger/ramooflax.
14. 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*, pages 1225–1230, 2008.
15. 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*, pages 129–135, 2009.
16. Éric Lacombe, Fernand Lone Sang, Vincent Nicomette, and Yves Deswartes. Une approche de virtualisation assistée par le matériel pour protéger l'espace noyau. In *Actes du 7ème symposium sur la sécurité des technologies de l'information et des communications (SSTIC)*, pages 189–214, 2009.
17. Yanlin Li, Jonathan M. McCune, and Adrian Perrig. Viper : Verifying the integrity of peripherals' firmware. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, pages 3–16, New York, NY, USA, 2011. ACM.
18. Fernand Lone Sang, Vincent Nicomette, and Yves 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)*, pages 237–265, 2012.
19. Silicore Opencores.org. Specification for the : Wishbone system-on-chip (soc) interconnection architecture for portable ip cores. http://cdn.opencores.org/downloads/wbspec_b3.pdf.

20. Wu Qingbo, Wang Chunguang, and Tan Yusong. System monitoring and controlling mechanism based on hypervisor. In *Parallel and Distributed Processing with Applications, 2009 IEEE International Symposium on*, pages 549–554, 2009.

21. M. Rezaei, N.S. Moosavi, H. Nemati, and R. Azmi. Tcvisor : A hypervisor level secure storage. In *Internet Technology and Secured Transactions (ICITST), 2010 International Conference for*, pages 1–9, 2010.

22. Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan 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*, CCS '09, pages 199–212, New York, NY, USA, 2009. ACM.

23. Joanna Rutkowska. Introducing blue pill. `http://theinvisiblethings.blogspot.fr/2006/06/introducing-blue-pill.html`, 2006.

24. 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*, pages 106–112, 2010.

25. Cheng Tan, Yubin Xia, Haibo Chen, and Binyu 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*, pages 1–6, 2012.

26. Jiang Wang, Angelos Stavrou, and Anup Ghosh. Hypercheck : A hardware-assisted integrity monitor. In *Proceedings of the 13th International Conference on Recent Advances in Intrusion Detection*, RAID'10, pages 158–177, Berlin, Heidelberg, 2010. Springer-Verlag.

27. Zhi Wang and Xuxian Jiang. Hypersafe : A lightweight approach to provide lifetime hypervisor control-flow integrity. In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 380–395, 2010.

28. Rafal Wojtczuk and Joanna Rutkowska. Attacking SMM Memory via Intel® CPU Cache Poisoning. `http://invisiblethingslab.com/itl/Resources.html`, March 2009.

29. Rafal Wojtczuk and Alexander Tereshkin. Introducing Ring -3 Rootkits. `http://invisiblethingslab.com/itl/Resources.html`, March 2009.