Abyme : un voyage au cœur des hyperviseurs récursifs

Benoît Morgan, Éric Alata,
Vincent Nicomette, Guillaume Averlant
prenom.nom@laas.fr

LAAS-CNRS, 7 av. du Colonel Roche, 31400 Toulouse
INSA Toulouse, 135 av. de Rangueil, 31400 Toulouse
Université Toulouse III - Paul Sabatier, 118 route de Narbonne, 31400 Toulouse

Résumé Les gestionnaires de machines virtuelles (ou VMMs) sont de plus en plus utilisés dans le monde de l'informatique aujourd'hui (notamment, depuis l'engouement pour le Cloud Computing). Ils reposent principalement sur des techniques d'assistance matérielle à la virtualisation proposées par les différents fondeurs de processeurs. Par ailleurs, il est de plus en plus courant de rencontrer des couches de virtualisation imbriquée ou *nested virtualization*, également supportée par des techniques matérielles. Cette approche a pour effet d'augmenter le nombre de couches de virtualisation. En partant de ce constat, nous nous sommes penchés sur la conception et le développement d'un hyperviseur "récursif", permettant de supporter un nombre quelconque de couches de virtualisation. Nous décrivons son architecture, ainsi qu'un modèle formel décrivant de façon simple comment gérer ces couches de virtualisation. Quelques tests permettant d'évaluer ses performances sont également proposés à la fin de cet article.

1 Introduction

Les gestionnaires de machines virtuelles (ou VMM pour *Virtual Machine Monitor*) sont aujourd'hui couramment utilisés dans le monde informatique, en particulier depuis l'engouement pour le *Cloud Computing*. Leur objectif principal est de contrôler l'exécution de machines virtuelles en profitant d'un niveau de privilège elevé du processeur et ils comportent de plus en plus de fonctionnalités. La majorité des VMMs du marché profitent aujourd'hui des techniques matérielles d'assistance à la virtualisation pour exécuter et contrôler des machines virtuelles. L'installation d'un hyperviseur destiné à virtualiser un VMM et utilisant pour cela lui-aussi les techniques matérielles d'assistance à la virtualisation nécessite donc la gestion de la virtualisation imbriquée (ou *nested virtualization*). Les processeurs récents aujourd'hui fournissent des fonctionnalités pour faciliter la mise en place de la nested virtualization. Mais, par ailleurs,

les VMMs du marché utilisent déjà pour beaucoup d'entre eux ces mécanismes de nested virtualization car ils peuvent être amenés eux-mêmes à virtualiser des VMMs. Les VMMs sont donc devenus de plus en plus complexes au fil du temps. Cette complexité s'accompagne d'une forte probabilité d'introduction de vulnérabilités. Ils deviennent donc des cibles très intéressantes pour un attaquant. Effectivement, leur compromission assure à l'attaquant une prise de contrôle sur un niveau d'exécution très privilégié du processeur.

Il est donc fondamental aujourd'hui de pouvoir détecter la compromission de VMMs. Cette détection peut se faire notamment en installant un logiciel capable de virtualiser ce VMM. Cette virtualisation permet ainsi d'observer et contrôler les différentes opérations effectuées par le VMM, afin de caractériser son comportement "normal" et détecter les compromissions. Aussi, si l'on désire créer un hyperviseur destiné à virtualiser un VMM, on réalise rapidement qu'il doit être capable de supporter un nombre quelconque de couches de virtualisation. A partir de ce constat, nous nous sommes donc intéressés à la création d'un hyperviseur capable de gérer ce problème de multiples couches de virtualisation. Pour cela, nous avons expérimenté un hyperviseur "récursif", au sens où il est capable de se virtualiser un nombre quelconque de fois.

De la même manière que les processeurs gèrent plusieurs anneaux de protection pour séparer les privilèges, la sécurité liée à l'usage de la virtualisation doit également passer par une séparation des privilèges. Cependant aujourd'hui, un seul niveau de privilèges supplémentaire est ajouté aux processeurs récents pour gérer la virtualisation. Il est donc important de penser une architecture logicielle pour gérer cette séparation des privilèges au sein de l'hyperviseur. On pourrait ainsi imaginer l'implémentation de fonctionalités de sécurité plus ou moins critiques dans ces différents niveaux de privilèges. Pour implémenter cette séparation des privilèges au sein de ce même hyperviseur, la virtualisation imbriquée est une des solutions envisageables, l'hyperviseur de plus bas niveau étant dédié aux fonctions les plus critiques et celui de plus haut niveau étant dédié aux fonctions les moins critiques.

Concevoir et implémenter un tel hyperviseur de sécurité implique donc la maîtrise de la gestion d'un nombre quelconque de niveaux de virtualisation. C'est le but de l'expérimentation réalisée dans cet article.

Nous nous sommes donnés, dans un premier temps, les objectifs suivants. Concevoir et implémenter un hyperviseur :

— léger pour ne pas entraîner une surcharge trop importante du système ;

— générique et facile à étendre pour implémenter des fonctions de sécurité ;
— récursif pour permettre l'instanciation de plusieurs fonctions de sécurité avec des niveaux de criticité différents ;
— dont le comportement est facilement modélisable pour vérifier son fonctionnement ;
— supportant la *full-virtualisation* pour ne pas avoir à modifier le comportement des systèmes virtualisés ;
— dont les performances sont facilement évaluables.

D'autres articles se sont focalisés sur la virtualisation imbriquée [3,1,6,7]. Mais, à notre connaissance, il n'existe pas d'implémentation d'un hyperviseur minimaliste, récursif, *bare-metal* et générique sur laquelle des tests de perfomance peuvent être réalisés et des fonctions de sécurité peuvent être instanciés sur differents niveaux. Cet article est un premier pas vers cette architecture et propose en particulier un modèle, une implémentation préliminaire sur x86 et des tests. Cet hyperviseur est baptisé Abyme.

Le plan de l'article est donc le suivant. La section 2 présente l'architecture de notre hyperviseur tandis que la section 3 détaille le modèle sur lequel est basé le fonctionnement de cet hyperviseur. La section 4 propose quelque détails d'implémentation et la section 5 donne les résultats de quelques tests de performance que nous avons réalisés. Finalement, la section 6 conclut cet article.

2 Architecture

Cette section décrit l'architecture globale d'Abyme. Les détails d'implémentation sont abstraits autant que possible. Il est néanmoins important de connaître le fonctionnement de base de l'architecture x86 ainsi que celui des extensions de virtualisation Intel VT-x. Après des rappels technologiques présentés dans la section 2.1, la stratégie de virtualisation classique du matériel est décrite dans la partie 2.2 et la stratégie de virtualisation du jeu d'instruction VMX des extensions de virtualisation Intel est décrite dans la partie 2.3. Enfin, dans la partie 2.4, les principaux concepts de virtualisation récursive sont introduits.

2.1 Intel VT-x

VT-x [2], 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 (VM ou *guests*) sont exécutées sur des cœurs virtuels configurés avec des structures internes du processeur appelées *Virtual Machine Control Structures* (VMCS). Les VMs 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 ou *host*) qui s'exécute en mode *VMX root operation*. Pour créer et contrôler une VM, le VMM 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 VM ainsi que des contrôles d'exécution. Enfin la VM est démarrée avec l'instruction `vmlaunch`. Le VMM prend le contrôle sur la VM au travers d'interruptions de machines virtuelles appelées VM Exits. Après avoir traité correctement le VM Exit, dans le cas nominal, le VMM redonne la main à la VM grâce à l'instruction VMX `vmresume` (figure 1).

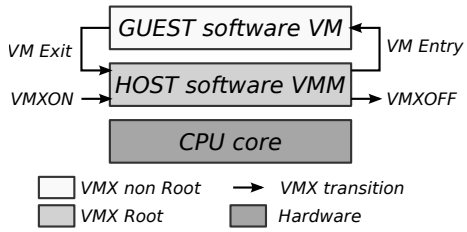

Figure 1. VMX operation

Le VMM 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). Le principe d'EPT est quasiment identique à celui de la pagination IA-32e classique : découpage d'une adresse en indices utilisés pour naviguer dans une arborescence, sachant que la feuille de l'arbre contient le résultat de la traduction. Avec EPT, les adresses physiques de la VM appelées *guest physical addresses* sont traduites en *host physical addresses*, utilisées par le contrôleur mémoire.

Pour plus de détails sur les points techniques, et l'architecture x86, le lecteur peut se référer à [4].

2.2 Virtualisation du matériel

Abyme est un hyperviseur employant 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 intructions privilégiées de la VM, sans pour autant altérer son fonctionnement. Pour ce faire, il va masquer aux yeux de la VM son espace mémoire et le matériel qu'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. Aussi, Abyme est un hyperviseur qui se présente sous la forme d'un pilote UEFI de *runtime*. Sa stratégie de chargement est expliquée dans [4]. 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 réseaux d'aujourd'hui étant la plupart du temps *mappés* en mémoire, il doit aussi les protéger directement pour prévenir 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 (e.g. 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 à protéger. 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 VM 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, avec ses octets à `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. Il doit maintenant protéger l'accès à certains registres et à l'exécution par la VM 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 VM.

Enfin, d'autres instructions génératrices inconditionnelles de VM Exits seront simplement exécutées sur le processeur car non dangeureuses pour le bon fonctionnement d'Abyme (i.e. `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.

2.3 Virtualisation du jeu d'instructions VMX

La virtualisation du jeu d'instruction VMX est un point technique clé de cet article. 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 les 13 instructions tout au long de son exécution. Dans cette partie, nous décrivons le travail qu'un hyperviseur doit réaliser pour virtualiser ces instructions. Une très grande partie du travail d'implémentation de cette fonctionnalité a constitué à reproduire un comportement aussi proche que possible de celui décrit dans le manuel du développeur Intel [2]. Le fonctionnement

et l'utilité des instructions principales sont rappelés avant d'expliquer comment les virtualiser de manière simple. Certains tests matériels très précis ne sont pas décrits ici.

Abyme a pour but de virtualiser les intructions VMX. Cela signifie qu'il doit reproduire leur comportement nominal, mais aussi celui de leur gestion des erreurs. La gestion des erreurs pour ce jeu d'instructions 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. Dans le cas de *VMfailValid*, un code d'erreur est inscrit dans le champ *VM-instruction error* de la VMCS courante. La manipulation du registre RFLAGS et du champ *VM-instruction error* suffisent 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.

Pour chaque instruction prenant en paramètre un pointeur de VMCS, les vérifications suivantes sont effectuées : si l'adresse de la VMCS passée en paramètre n'est pas alignée sur 4ko ou si son identifiant de révision est mauvais, nous retournons l'état *VMfailInvalid*.

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

Nous passons à présent en revue chacune des intructions VMX pour indiquer comment nous pouvons les émuler.

`vmxon(VMCS *)` est la première instruction du jeu VMX utilisée par un hyperviseur pour activer le mode *VMX operation*. Elle est aussi la seule exécutable par le processeur en dehors de ce mode. Mis à part les vérifications sur le pointeur, il n'y a rien à effectuer pour cette instruction.

`vmclear(VMCS *)` permet de passer une VMCS à l'état *clear* pour pouvoir lancer sa VM associée avec `vmlaunch`. Pour l'émuler, nous allons charger avec `vmprtld` la VMCS passée en paramètre, appeler `vmclear` sur cette VMCS et enfin, recharger la VMCS de l'hyperviseur virtualisé.

`vmptrld(VMCS *)` permet de changer la VMCS courante, qui sera implicitement utilisée par les instructions `vmread`, `vmwrite`, `vmlaunch`, `vmresume`, etc. Pour émuler cette instruction, il suffit simplement de copier le pointeur de VMCS. Nous appellerons ce pointeur le *shadow VMCS pointer* pointant vers une *shadow VMCS*, conformément à la documenta-

tion Intel. L'hyperviseur n'effectue immédiatement pas 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` permet de démarrer l'exécution d'une VM configurée par une VMCS dont l'état est *clear*. Dans le cas de l'émulation, l'hyperviseur virtualisé tente de démarrer l'exécution d'une machine virtuelle. Cette machine virtuelle correspond à la VMCS que l'hyperviseur virtualisé a précédement chargé avec l'instruction `vmptrld`. Cette VMCS 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` (cf. section 4).

`vmresume` permet de continuer 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 configuré dans la *shadow VMCS*. Abyme effectue la même copie d'état du `vmlaunch` puis exécute `vmresume` (cf. section 4).

`vmwrite(field, value)` permet d'accéder en écriture au champ field de la VMCS courante. Pour ce faire, l'hyperviseur charge la `shadow VMCS`, i.e. la VMCS courante de l'hyperviseur virtualisé, exécute le `vmwrite(field, value)` et recharge sa VMCS.

`vmread(field)` permet d'accéder en lecture au champ field de la VMCS courante. La même stratégie que pour `vmwrite` est appliquée, les valeurs de retours étant copiée dans sa VMCS.

2.4 Virtualisation récursive

Supporter la virtualisation récursive signifie qu'un hyperviseur doit pouvoir virtualiser son propre code N fois, 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 [1], indiquant que les évènements remontent du niveau zéro vers le niveau dont la responsabilité de traitement incombe.

Une différence demeure, Abyme virtualise son propre code et doit rester fonctionnel à chaque niveau. La stratégie adoptée pour cela et son modèle comportemental associé sont décrits dans la partie suivante.

3 Abyme, Hyperviseur récursif

Cette section présente le modèle de l'hyperviseur Abyme ainsi que les résultats obtenus après sa simulation. Le but de ce modèle est de pouvoir représenter les transitions, interruptions et actions effectuées par le processeur lors de l'exécution de N niveaux d'hyperviseurs ($N \in \mathbb{N}$) et ainsi valider le comportement d'un hyperviseur au niveau x, virtualisé ou non par d'autres hyperviseurs. La trace du comportement attendu a aussi permis de faciliter la compréhension et le développement d'Abyme.

Seules les instructions de manipulation de contextes et de VMCS sont présentées dans une version minimaliste. 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.

3.1 Avec deux hyperviseurs : N = 2

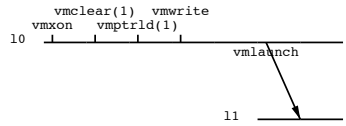

Figure 2. Démarrage de l_0

Afin de mieux comprendre le fonctionnement de l'hyperviseur, nous avons représenté sa séquence de démarrage simplifiée à la figure 2. La virtualisation est activée avec `vmxon`, passant le processeur en mode *VMX operation*. l_0 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`.

Abyme étant récursif, il est possible de le charger une deuxième fois au niveau 1 en tant que l_1. l_0 étant déjà installé en tant qu'hyperviseur, il va

émuler toutes les intructions VMX exécutées par l_1. La figure 3 représente les transitions effectuées lors de ces démarrages successifs.

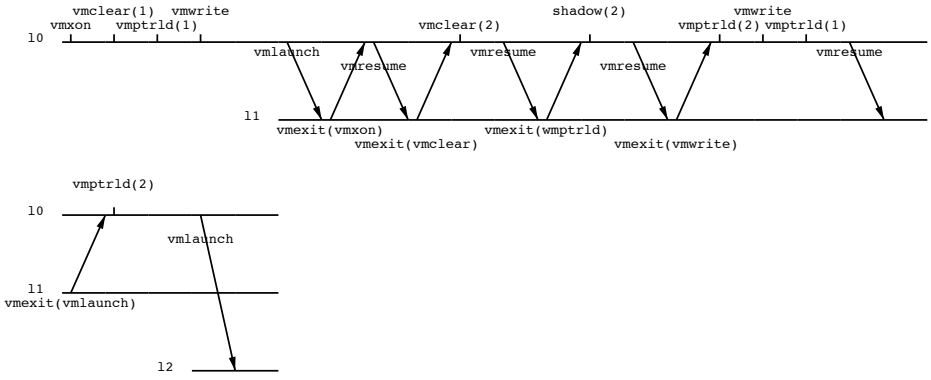

Figure 3. Démarrage de l_0 et l_1

Une analyse visuelle simple montre que le comportement de l'hyperviseur l_1 n'est pas altéré par la présence de l_0. Si l'on s'attarde sur l'instruction `vmlaunch` que l_1 tente d'exécuter, on peut voir qu'après prise de contrôle et émulation par l_0, la VM l_2 est bien démarrée. Pour cela, l_1 tente d'exécuter l'instruction `vmxon`. Comme il est virtualisé, cela déclenche un `vmexit`, qui est représenté sur la figure 3 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 sa propre

VMCS (`vmptlrd(1)`). Enfin, comme l_1 a terminé de configurer la VM l_2, il la démarre avec un `vmlaunch`, qui génére un `vmexit(vmlaunch)`. l_0 charge alors sa *shadow VMCS*, pointant vers l_2, effectue le `vmlaunch` et donne ainsi la main à l_3.

Tracer le diagramme de Hasse pour l'exécution de l_1 donne la figure 2 à l'identique, soit l'exécution de l_0.

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.

3.2 Généralisation : Modélisation d'Abyme

Cette section présente le travail de modélisation en notation fonctionnelle du comportement de l'hyperviseur récursif. Ce travail n'a pas pour but de modéliser, comme l'a fait Popek et al. [5], toute une architecture matérielle de manière formelle, mais plutôt de modéliser le comportement spécifique de Abyme associé aux transitions générées par le mode *VMX opration*. Certaines transitions du modèle, notamment celles résultant de la gestion des VMCS, sont simplifiées car elles génèrent trop d'évènements, ce qui compromet la lisibilité des diagrammes. Nous les explicitons cependant dans la partie 4.

Le modèle est découpé en 4 parties :

— les terminaux ;
— le comportement du processeur ;
— le comportement d'Abyme ;
— la séquence de démarrage d'Abyme.

Terminaux Les fonctions suivantes représentent les instructions et évènements réellement traités sur le processeur. Elles consituent les terminaux de notre language de modélisation. Elles sont classées en trois catégories : instructions, interruptions et actions.

Instructions

$$\begin{cases} ON_i = \texttt{vmxon} \\ P_i(a) = \texttt{vmptrld(a)} \\ C_i(a) = \texttt{vmclear(a)} \\ L_i = \texttt{vmlaunch} \\ R_i = \texttt{vmresume} \\ WR_i = \texttt{vmread} \\ RD_i = \texttt{vmwrite} \\ I_i \Leftrightarrow \text{Instruction privilégiée (}\texttt{cpuid}\text{, etc.)} \end{cases}$$

Interruptions

$E_i(x, I) = \texttt{vmexit(x, I)} \Leftrightarrow l_x$ tente d'exécuter I et a généré un VM Exit

Actions

$$\begin{cases} S(x, a) = \texttt{shadow(x, a)} \\ S(x) = \texttt{shadow(x)} \end{cases}$$

La fonction S met à jour la *shadow VMCS* de l'hyperviseur x, ou retourne celle configurée précédemment.

$$\begin{cases} GL(x, b) = \texttt{guest\_launched(x, b)} \\ GL(x) = \texttt{guest\_launched(x)} \end{cases}$$

Dans la partie 4, nous montrerons qu'il est possible d'utiliser une même VMCS pour exécuter plusieurs niveaux d'hyperviseur, ce qui permet de ne pas avoir recours à de l'allocation dynamique de VMCS. La fonction *GL* permet à un hyperviseur l_x de savoir s'il a déjà effectué un `vmlaunch` sur sa VMCS d'exécution des niveaux $> x + 1$. Le cas échéant, il va exécuter un `vmresume` à la place. Des détails d'implémentation concernant l'utilisation des VMCS sont donnés dans la partie 4.

Comportement du processeur Les fonctions suivantes définissent le comportement du processeur en fonction du niveau d'exécution x. Si $x > 0$, l'exécution se déroule dans un processeur virtuel et génère un VM Exit. Dans le cas ou $x = 0$, il s'agit de l'hyperviseur l_0 et le processeur va effectivement exécuter l'instruction I_i.

$$ON(x) = \begin{cases} ON_i \text{ si } x = 0 \\ E_i(x, ON).E(ON, x, 0, \epsilon) \text{ si } x > 0 \end{cases}$$

$$P(x,a) = \begin{cases} P_i(a) \text{ si } x = 0 \\ E_i(x,P).E(P,x,0,a) \text{ si } x > 0 \end{cases}$$

$$C(x,a) = \begin{cases} C_i(a) \text{ si } x = 0 \\ E_i(x,C).E(C,x,0,a) \text{ si } x > 0 \end{cases}$$

$$L(x) = \begin{cases} L_i \text{ si } x = 0 \\ E_i(x,L).E(L,x,0,\epsilon) \text{ si } x > 0 \end{cases}$$

$$R(x) = \begin{cases} R_i \text{ si } x = 0 \\ E_i(x,R).E(R,x,0,\epsilon) \text{ si } x > 0 \end{cases}$$

$$RD(x) = \begin{cases} RD_i \text{ si } x = 0 \\ E_i(x,RD).E(RD,x,0,\epsilon) \text{ si } x > 0 \end{cases}$$

$$WR(x) = \begin{cases} WR_i \text{ si } x = 0 \\ E_i(x,\ WR).E(\ WR,x,0,\epsilon) \text{ si } x > 0 \end{cases}$$

$$I(x) = \begin{cases} I_i \text{ si } x = 0 \\ E_i(x,I).E(I,x,0,\epsilon) \text{ si } x > 0 \end{cases}$$

A titre d'exemple, la première équation signifie que si l_0 effectue l'opération `vmxon`, alors elle est réellement exécutée sur le processeur. Si c'est un hyperviseur d'un niveau strictement supérieur à 0 qui exécute cette instruction, alors, le processeur déclenche en réalité un `vmexit`. Le traitement du VM Exit est représenté par le fonction $E(I,x,y,p)$, ou I est l'instruction qui a déclenché le VM Exit, x le niveau de l'hyperviseur source de l'évènement, y l'hyperviseur courant en train de gérer l'évènement et p un paramètre le cas échéant. Après un VM Exit, y est toujours égal à 0 car l_0 est maître du matériel. L'évènement remontera aux autres niveaux jusqu'à l'hyperviseur responsable du traitement. Les autres équations s'interprètent de la même façon.

Comportement de l'hyperviseur Les fonctions suivantes décrivent le comportement d'Abyme en terme de virtualisation du jeu d'intructions VMX. Comme introduit dans la partie précédente, lorsqu'un évènement se produit au niveau x avec $x > 0$, la responsabilité du traitement de cet évènement incombe au niveau $x - 1$. La fonction $E(I,x,y,p)$ modélise ce comportement. L'hyperviseur courant, y, va traiter le VM Exit courant pour I si et seulement si $y = x - 1$. Ceci est représenté ci-dessous par la

notation $H(I, x, y, p)$. Cette fonction contient l'implémentation de l'émulation de l'intruction I. Dans les autres cas, il va le déléguer au niveau supérieur $y + 1$, grâce à $P(y, y + 1)$ et $R(y)$. Cette séquence signifie que y charge le contexte de $y + 1$ (`vmptrld`), lui donne la main (`vmresume`) et fait suivre le contexte du VM Exit avec $E(I, x, y + 1, p)$.

$$E(I, x, y, p) = \begin{cases} H(I, x, y, p) \text{ si } y = x - 1 \\ P(y, y + 1).R(y).E(I, x, y + 1, p) \text{ si } x > 0 \end{cases}$$

Nous arrivons maintenant à la modélisation de l'émulation des instructions du jeu VMX, conformément à la partie 2.

$$\begin{cases} H(ON, x, y, \epsilon) &= GL(y, F).R(y) & (1) \\ H(L, x, y, \epsilon) &= P(y, S(y)).R(y) \text{ si } GL(y) = T & (2.1) \\ H(L, x, y, \epsilon) &= P(y, S(y)).GL(y, T).L(y) \text{ si } GL(y) = F & (2.2) \\ H(P, x, y, a) &= S(y, a).R(y) & (3) \\ H(R, x, y, a) &= P(y, s(y)).R(y) & (4) \\ H(C, x, y, a) &= C(y, a).R(y) & (5) \\ H(RD, x, y, \epsilon) &= P(y, S(y)).RD(y).P(y, x).R(y) & (6) \\ H(WR, x, y, \epsilon) &= P(y, S(y)).WR(y).P(y, x).R(y) & (7) \\ H(I, x, y, \epsilon) &= I(y).R(y) & (8) \end{cases}$$

L'équation (1) représente le comportement de Abyme lors du traitement d'un VM Exit causé par l'instruction `vmxon` au niveau x. L'hyperviseur note que sa VMCS pour les niveaux $> y + 1$ n'a pas encore été lancée, en inscrivant F (FALSE) avec la fonction $GL(y, F)$. Il rend la main à l_x, avec $R(y)$.

Les équations (2.1) et (2.2) fonctions précédentes représentent les deux versions de l'émulation du lancement d'une VM en fonction du booléen retourné par $GL(y)$. Dans tous les cas on charge la *Shadow VMCS* que l_x désire lancer avec $P(y, S(y))$. Si $GL(y)$ est faux, on note que la VM est lancée avec $GL(y, T)$ (T pour TRUE), et on effectue le `vmlaunch` avec la fonction $L(y)$. Dans le cas contraire, on exécute juste un `vmresume`, avec $R(y)$, pour ne pas avoir d'erreur, puisque la VMCS à déjà été lancée.

L'équation (3) indique que pour un `vmresume` commandité par l_x, le traitement est strictement identique à celui du `vmlaunch` dans le cas où la VMCS est déjà lancée.

Dans L'équation (4), pour les VM Exit générés par l'instruction `vmptrld`, l'hyperviseur va sauvegarder le pointeur en paramètre a en tant que shadow VMCS avec $S(y, a)$ et rendre la main à l_x.

De la même manière, les équations (5), (6), (7) et (8) représentent la modélisation de l'émulation de l'instruction VMX associée, telle que décrite dans la partie 2.

Modélisation du démarrage Abyme est un hyperviseur récursif. Il va être chargé plusieurs fois en séquence pour N niveaux de virtualisation. La séquence de chargement est écrite dans un script shell UEFI, exécutée par le shell UEFI chargé au préalable.

Script shell

$$start(x) = load(x, 0)$$

$$load(x, y) = \begin{cases} setup(x) \text{ si } x = y \\ setup(x).load(x, y + 1) \text{ si } x > 0,\ y > 0 \end{cases}$$

Démarrage d'Abyme

$$setup(x) = ON(x).C(x, x + 1).P(x, x + 1).\, WR(X).L(x)$$

Maintenant que les parties représentatives du travail d'Abyme pour supporter la virtualisation récursive N sont modélisées, il est possible d'implémenter ce modèle pour analyser les résultats en fonction de N.

3.3 Avec N hyperviseurs

Nous avons implémenté le modèle décrit précédemment en Prolog. Ce programme Prolog nous génère une sortie au format texte représentant tous les évènements qui se sont produits. Ces évènements sont traités ensuite par un *parser* écrit en Python qui va déduire les transitions entre les hyperviseurs afin de générer une sortie au format SVG. La sortie pour N = 2 est représentée dans la figure 4. Ce principe peut être repris pour N niveaux d'hyperviseurs. Pour l'exécution du chargement de chaque niveau, la génération du diagramme de Hasse associé donne systématiquement la trace du chargement du niveau zéro.

Notons que l'impact du niveau d'hyperviseurs sur la génération d'évènements est exponentiel. Par conséquent, pour des raisons de place, nous ne pouvons pas représenter dans cet article la séquence de démarrage pour plus de niveaux, même si elle est parfaitement opérationnelle.

La modélisation formelle d'Abyme masque volontairement certains aspects techniques non représentatifs du comportement récursif et d'émulation des instructions. Ces aspects, comme le nombre de VMCS par niveau, mais encore la gestion des caches, sont par contre indispensables pour une implémentation fonctionnelle.

4 Mise en œuvre

Cette partie développe certains points et choix du développement de Abyme qui nous ont permis de garder une taille de son image assez réduite.

4.1 Détails techniques

Chaque hyperviseur l_x lance une machine virtuelle l_{x+1} à la fin de son installation. Il va donc configurer une VMCS pour celle-ci durant sa phase de démarrage, qui servira aussi à sauvegarder l'état de l_{x+1} lors de VMX transitions. Cette VMCS permettra de sauvegarder par transitivité les états de tous les hyperviseurs des N niveaux, l_x sauvegardant l'état de l_{x+1}. On appelera cette VMCS : `vmcs0`.

Un hyperviseur l_x va être aussi amené à devoir lancer ou reprendre l'exécution d'une machine virtuelle de niveau $> x + 1$. Pour ne pas perdre l'état de l'exécution du niveau $x + 1$, dont il est reponsable, il va devoir utiliser une deuxième VMCS, nommée `guest_vmcs`, qui sera utilisée uniquement pour supporter l'éxécution des niveaux dont il n'a pas la responsabilité directe. Lors du traitement d'un `vmresume` ou `vmlaunch`, commandité par l_{x+1}, Abyme va donc systématiquement recopier l'état de la *shadow VMCS*, configurée par les niveaux supérieurs, dont il a sauvegardé le pointeur, dans la partie *guest* de sa `guest_vmcs`, en faisant attention aux champs à contrôler pour garder effectivement la main sur le matériel. Cette copie est indispensable car la VM de l_{x+1} doit s'executer tout en laissant à l_x le même contrôle sur le matériel. Il est donc nécessaire de créer une nouvelle VMCS dans laquelle la partie host correspond a l_x et la partie guest correspond à la VM de l_{x+1}. La copie se déroule en deux étapes. Premièrement la *shadow VMCS* est chargée via l'intruction `vmptrld` et chacun des champs à rappatrier sont recopiés dans un tableau. Ensuite, la `guest_vmcs` est chargée et tous les champs sauvegardés dans le tableau sont recopiés directement ou mergés avec une configuration non dangereuse pour l'intégrité de l_x. Si nous prenons l'exemple de la figure 4, lors du traitement par l_0 du VM Exit généré par un `vmresume` de l_1, l'état de la *shadow VMCS* de l_0, pointant vers la `vmcs0` de l_1, est recopiée dans

la `guest_vmcs` de l_0. Cette copie systématique qui sera effectuée par les N niveaux, pour tous les traitements de `vmlaunch` et de `vmresume`, fera "descendre" l'état de la VM de niveau $x+1$, lancée par l_x, jusqu'à la `guest_vmcs` de l_0 et ainsi l'exécuter réellement sur le processeur (figure 5). Ce tranfert d'état est donc générateur de beaucoup de VM Exit, c'est pour cela qu'il a été choisi de ne pas le représenter dans la modélisation.

De la même manière, lors de VM Exits, si l_x n'est pas responsable de l'évènement généré par l_y, il va le faire remonter au niveau supérieur, avec l'état d'exécution de la VM l_y, tenant ainsi à jour la `vmcs0` du niveau responsable. Ainsi, l'état de la VM l_{x+1}, exécutée par l_x garde sa cohérence quelque soit le nombre d'hyperviseurs la virtualisant (figure 6).

Abyme est maintenant capable d'exécuter les N niveaux de machines virtuelles. Cependant, chaque l_x, lors de son installation, va configurer la virtualisation de la MMU avec EPT.

EPT est la propriété réelle uniquement du niveau 0. Pour pallier à ce problème, sachant que chaque niveau configure EPT en identity mapping [1], protège son espace mémoire qui est totallement disjoint du niveau sous-jacent et charge une seule fois son pointeur après configuration complète, il est simplement possible que le niveau x merge les attributs de ses tables avec ceux du niveau $x+1$, tout en gardant la trace de la propriété des pages du niveau supérieur dans une structure de données très simple. Cette stratégie, de la même manière que pour l'état des VMs, fera "redescendre" jusqu'au niveau 0 la configuration d'EPT des N niveaux. Pour terminer, lors de l'ordonnancement de l_{x+k}, l_x devra appliquer les droits des pages de l'espace mémoire des niveau $[x; k-1]$ et flusher les TLBs des régions correspondantes avec l'instruction `invvip` (figure 7). Ainsi le niveau $x+k$ pourra écrire dans son espace mémoire et ceux supérieurs, mais pas dans les inférieurs. Lors d'une faute de page EPT (*EPT violation*), il suffit de regarder dans la structure de sauvegarde si nous devons la traiter ou la déléguer au niveau supérieur.

L'utilisation d'une seule `guest_vmcs` par niveau et non autant qu'il y a de VMs $> x+1$, permet de ne pas avoir d'allocation dynamique de VMCS dans notre code (l_x ne sachant pas combien il virtualisera d'hyperviseurs in fine). Cela implique que toutes les VMs utilisent l'intégralité des TLBs. Pour éviter des problèmes d'incohérence du cache, il faudrait à priori les vider lors de l'ordonnancement des VM, ce qui ralentirait les performances. Or, sachant que chaque niveau exécute le même hyperviseur et traduit de la même manière les adresses virtuelles en adresses physiques, une incohérence du cache TLB ne peut pas survenir. Donc, dans notre cas, il

1. Avec la même granularité (4K, 2M, 1G) pour chaque segement mémoire

n'est pas nécessaire de vider ces caches. Par contre, lors de l'exécution de linux, la fonction de translation peut être différente de celle utilisée par les hyperviseurs. Donc, lors d'une transition qui concerne la VM linux, il est nécessaire de vider les caches. EPT apporte de plus une ségrégation des caches par identifiant de cœur virtuel, *Virtual Processor IDentifier* (VPID). En utilisant cette technologie, il suffit dans notre cas de donner un identifiant différent à linux et aux hyperviseurs, pour ne plus avoir à vider les caches.

En admettant l'allocation dynamique de VMCS, il est possible d'appliquer en plus la solution apportée par Turtles [1], qui utilise simplement une VMCS par niveau de virtualisation (figure 8), et qui permet de diminuer le nombre de champs à copier de la VMCS shadow vers la `guest_vmcs` d'exécution. Les performances en fonction des différentes stratégies de gestion des VMCS et du niveau N de virtualisation sont présentées dans la partie suivante.

Enfin, en ce qui concerne les interruptions, Abyme ne les utilise pas. Il configure son IDT pour sauvegarder les évènements qui se sont passés pour les transmettre au niveaux supérieurs jusqu'à l'OS qui les a configurées. Il procède pour cela de la même manière que pour l'état des VMs.

4.2 Contrôle et débogage d'Abyme à distance

Un des défis liés à la mise en œuvre à consisté à trouver un moyen efficace et simple permettant de 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 locaux comme interface de contrôle pour le développeur. Cette solution s'est très vite avérée non viable vis à vis du fait que le système d'exploitation chargé en haut de pile, va très rapidement les reconfigurer avec des drivers hautes performances et implique un partage du matériel avec l'hyperviseur. Une problématique compliquée à résoudre et pouvant augmenter considérablement la taille du code. C'est pour cela que 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 cablé 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 met en œuvre un protocole question réponse très simple incluant les fonctionnalités suivantes :

— contrôle d'exécution ;
— lecture et écriture dans la mémoire hôte ;
— accès aux registres du cœur courant ;
— messages info ;
— messages personnalisés utiles pour les expérimentations ;
— lecture et écriture dans la VMCS courante.

Abyme utlise une libc minimaliste, que nous avons développée, qui contient entre autre des fonctions d'affichage de chaînes de caractères telles que `printk(const char *fmt, ...)`. Ces chaînes sont affichées à l'écran, mélée avec celles du shell UEFI en phase de pré boot, puis sont redirigées sur le réseau, transportées par les messages info pour être traitées par un client de débogage. Ainsi, nous pouvons afficher du texte tout au long de l'exécution de la machine. Abyme étant récursif, chaque niveau de virtualisation va utiliser `printk` et envoyer des messages info. C'est pour cela que chaque PDU contient un champ donnant le niveau x de l'hyperviseur ayant envoyé le message de débogage. Ainsi nous pourrions exécuter en théorie le serveur de débogage aux N niveaux de virtualisation, ce qui n'est pas forcément utile. C'est pour cela que seule la fonctionnalité d'affichage est supportée aux N niveaux, par opposition au serveur de contrôle et débogage, qui lui s'exécute uniquement au niveau 0. Cette différence de comportement en fonction du niveau est la seule que nous ayons identifiée pour le moment, mais n'impacte pas le cœur de Abyme, le serveur de débogage ainsi que l'affichage à distance pouvant 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 polluant pas le code du cœur du projet.

Le client de contrôle et débogage à été écrit en Python. Il met en œuvre le comportement décrit par les PDU de notre protocole (figure 9) mais ajoute aussi des fonctionnalités plus pratiques et de plus haut niveau comme le désassemblage du code de la VM pointé par son `rip` courant (figure 10). Le développement de cette fonction nécessite l'exécution de plusieurs étapes indispensables et conditionnées par le résultat de la précédente :

— 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 mémoire successives pour récupérer les différents niveaux de Page Entries en fonction du type de pagination (32 bit, PAE, et IA-32e)
— lecture de la page contenant le code à l'adresse traduite ;
— invocation du *binutil* `objdump` en fonction du mode et affichage de sa sortie après post traitement des caractères spéciaux.

Abyme peut donc s'exécuter sur N niveaux avec une machine virtuelle exécutant un système d'exploitation tel que *GNU Linux* en haut de pile. De plus, la version actuelle de notre hyperviseur est composée de moins de 4400 lignes de code C et GNU Assembler (headers inclus). Il est intéressant de maintenant le mettre à l'épreuve du point de vue de ses performances.

5 Résultats

Dans cette partie, des résultats de *benchmarks* sont présentés en fonction du nombre de couches de virtualisations, mais aussi en faisant varier certains détails d'implémentation vus précédemment.

La machine cible est un Dell precision T1700 équipée d'un processeur Intel I7-4770 et d'un chipset c226. Elle embarque 8 Go de mémoire DDR3 SDRAM. Sa carte Ethernet Broadcom NetXtreme BCM5722 est connectée à la machine de contrôle via un switch gigabit D-Link DGS-1008D.

Trois ensembles de *benchmarks* sont exécutés par la couche la moins privilégiée (en haut de la pile de virtualisation). Chaque ensemble est d'abord exécuté sans Abyme, puis avec $N = 0$, $N = 1$, etc. Le premier ensemble de *benchmarks* calcule 10 millions de décimales de π avec la librairie GNU Multi Precision, pour 10 itérations. Le deuxième *benchmark* copie, grâce à la commande Unix `dd`, un fichier de 512 Mo, sur le même disque, dans une partition ext4, 10 fois de suite. Et enfin le troisième effectue une copie via SSH d'un fichier de 512 Mo depuis la machine cible vers la machine de contrôle connectée au réseau local, 4 fois d'affilée.

Configuration 1 : Abyme a été configuré avec autant de VMCS par niveau que de couches virtualisées, sans *VMCS shadowing* matériel des Intel Haswell et avec le serveur de débogage désactivé.

Si on regarde les résultats, pour le test de *burn-in*, la surcharge mesurée (tableau 5) est à 0.1 %, pour $N = 0$ (1 hyperviseur), de 0.2 % pour $N = 1$ et 0.9 pour 4 hyperviseurs, ce qui est très acceptable. Ce résultat est intéressant mais pas suffisant pour estimer les performances de notre architecture car un *burn-in* du processeur n'est pas représentatif des différents cas d'utilisation d'une machine. C'est pour cela qu'il a été complété par deux autres ensembles de *benchmarks*, se focalisant sur le disque et le

réseau. Les résultats montrent que la variation des performances dues à la latence d'accès au disque et au taux d'utilisation du réseau impactent plus les performances que notre pile d'hyperviseurs pour $N = 0$ et $N = 1$. C'est pour cela qu'une surcharge faible voire négative à été mesurée, ce qui montre que notre solution est très performante. On note tout de même une augmentation de l'overhead pour le test réseau avec $N = 4$. Le temps de démarrage augmente progressivement en fonction du nombre d'hyperviseurs et est très élevé pour 4 hyperviseurs. Cette augmentation s'explique

Config	None	N=1	N=2	N=3	N=4	SN=1	SN=2	SN=3	SN=4	SN=5
Boot	20	21	22	25	469	21	21	25	47	240
Over.		5%	10 %	25 %	2245 %	5 %	5 %	25 %	135 %	1100 %
PI	221.25	221.53	221.62	221.73	223.17	221.58	221.51	221.72	221.62	222.53
Moyen	22.13	22.15	22.16	22.17	22.32	22.16	22.15	22.17	22.16	22.25
Médian	22.12	22.15	22.16	22.16	22.29	22.15	22.15	22.16	22.16	22.21
Min	22.10	22.13	22.13	22.15	22.27	22.14	22.12	22.14	22.13	22.20
Max	22.20	22.21	22.23	22.26	22.58	22.23	22.22	22.29	22.19	22.48
Over.		0.1 %	0.2 %	0.2 %	0.9 %	0.1 %	0.1 %	0.2 %	0.2 %	0.6 %
Disque	47.01	46.86	47.25	47.53	46.19	46.88	47.13	47.38	46.50	46.76
Moyen	4.70	4.69	4.73	4.75	4.62	4.69	4.71	4.74	4.65	4.68
Médian	4.75	4.73	4.77	4.81	4.68	4.74	4.75	4.86	4.69	4.71
Min	4.12	4.00	4.21	4.95	4.18	3.78	3.85	3.92	4.07	3.95
Max	4.91	4.99	2.99	5.39	4.90	5.07	5.15	5.01	5.29	5.40
Over.		-0.32 %	% 0.51	1.11 %	-1.74 %	-0.28 %	0.26 %	0.79 %	-1.08 %	-0.53 %
Réseau	190.86	190.82	190.60	191.33	196.19	190.81	190.35	189.48	190.36	191.51
Moyen	47.72	47.70	47.65	47.83	49.04	47.70	47.59	47.37	47.59	47.88
Médian	47.67	47.57	47.60	47.87	49.43	47.59	47.49	47.35	47.49	47.90
Min	47.15	47.16	47.15	47.26	47.06	47.12	47.25	46.95	47.12	47.30
Max	48.36	48.52	48.24	48.32	50.27	48.52	48.12	47.82	48.16	48.42
Over.		-0.02 %	-0.14 %	0.25 %	2.79 %	-0.03 %	-0.27 %	-0.72 %	-0.26 %	0.34 %

— Config : configuration du test
— Résultats en secondes sauf indiqué
— None : témoin sans virtualisation
— N = x : nombre d'hyperviseurs
— S : avec Intel VMCS shadowing
— Over. : overhead

Table 1. Résultats des *benchmarks* pour les configurations 1 et 2

par les transistions générées à cause des accès matériels (et de l'exécution d'instructions privilégiées), qui sont importantes au démarrage.

Configuration 2 : Avec le support du *VMCS shadowing* matériel.

De manière générale et sans surprises, les performances sont améliorées par la supression des VM Exits générés par les lectures et écritures dans la plupart des champs des VMCS. Nous pouvons donc charger plus d'hyperviseurs avant de voir les performances diminuer.

6 Conclusion

Dans cet article, nous avons présenté la conception et le développement d'un hyperviseur "récursif", baptisé Abyme, capable de s'auto-virtualiser un nombre quelconques de fois. Ce développement nous a permis de mieux comprendre et maîtriser les technologies de virtualisation imbriquées, particulièrement utiles aujourd'hui dans le développement de différents types d'hyperviseurs, notamment dédiées aux problèmes de sécurité. Cet hyperviseur est léger, *bare-metal* et peut être utilisé comme squelette pour l'implémentation de fonctions de sécurité tout en respectant le principe de séparation des privilèges.

Les perspectives à court terme de ces travaux concernent les tests de performances, que nous devons mener sur d'avantage de niveaux de virtualisation. Les performances diminuent drastiquement au-delà d'un certain nombre d'hyperviseurs du fait du nombre exponentiel de transitions générées entre les différents niveaux d'hyperviseurs. Une optimisation simple et efficace consiste à ne pas propager les VM Exits concernant l'instruction permettant d'ordonnancer les niveaux d'hyperviseurs (`vmresume`) et à la traiter directement au niveau zéro. En acceptant de priver les hyperviseurs virtualisés des évènements d'ordonnancement des hyperviseurs dont ils ont la charge, il est possible d'obtenir un nombre de transitions qui ne sera plus à caractère exponentiel. Ainsi, Abyme pourra supporter un nombre plus grand de couches de virtualisation sans perte significative de performances.

A plus long terme, nous souhaitons étudier comment il est possible de déployer et de faire collaborer des fonctions de sécurité dans les différents niveaux de virtualisation.

7 Annexes

```
echo changing terminal mode
mode 100 31
```

```
echo Startup !
fs1:
rec.nsh
echo Launching grub
echo press any key
pause
fs0:\EFI\arch_grub\grubx64.efi
```

Listing 1. Script shell UEFI primaire

```
cd \EFI\drivers\82579LM
echo Loading ethernet driver
load efi.efi
echo Ethernet runtime driver loaded !
cd \EFI\drivers\vmm_rec
echo Loading recursive vmm
load efi.efi
echo Recursive vmm runtime driver loaded !
cd \EFI\drivers\vmm_rec
echo Loading recursive vmm
load efi.efi
echo Recursive vmm runtime driver again loaded !
cd \EFI\drivers\vmm_rec
echo Loading recursive vmm
load efi.efi
echo Recursive vmm runtime driver again again loaded !
cd \EFI\drivers\vmm_rec
echo Loading recursive vmm
load efi.efi
echo Recursive vmm runtime driver again again again loaded !
cd \EFI\drivers\vmm_rec
echo Loading recursive vmm
load efi.efi
echo Recursive vmm runtime driver again again again again loaded !
```

Listing 2. Script shell UEFI de chargement des hyperviseurs

```
/**
 * I/O
 */
file(F) :- asserta(fout(F)).
fopen :- fout(F), open(F, write, ID), asserta(fdout(ID)).
fclose :- fdout(ID), close(ID).

/**
 * Main
 */
start(X) :- fopen, load(X, 0), fclose.

/**
 * Intructions Executees
 */
format_out(F, V) :- fdout(ID), format(ID, F, V).
```

```
pi(A) :- format_out("p(~d)\n", [A]).
ri :- format_out("r\n", []).
ei(X, I) :- format_out("e(~d, ~a)\n", [X, I]).
si(X, A) :- format_out("s(~d, ~d)\n", [X, A]).
rdi :- format_out("rd\n", []).
wri :- format_out("wr\n", []).
ci(A) :- format_out("c(~d)\n", [A]).
li :- format_out("l\n", []).
oni :- format_out("on\n", []).

/**
 * Processeur
 */
p(0, A) :- pi(A).
p(X, A) :- ei(X, p), e(p, X, 0, A).
r(0) :- ri.
r(X) :- ei(X, r), e(r, X, 0, _).
rd(0) :- rdi.
rd(X) :- ei(X, rd), e(rd, X, 0, _).
wr(0) :- wri.
wr(X) :- ei(X, wr), e(wr, X, 0, _).
c(0, A) :- ci(A).
c(X, A) :- ei(X, c), e(c, X, 0, A).
l(0) :- li.
l(X) :- ei(X, l), e(l, X, 0, _).
on(0) :- oni.
on(X) :- ei(X, on), e(on, X, 0, _).

/**
 * Comportement de l'UEFI script shell
 */

load(X, X) :- vmm_setup(X).
load(X, Y) :- Y1 is Y + 1, vmm_setup(Y), load(X, Y1).

/**
 * Comportement de l'hyperviseur
 */

/* Installation */

vmm_setup(X) :- X1 is X + 1, on(X), c(X, X1), p(X, X1), wr(X), l(X).

/* Nested VMX */

e(p, X, Y, A) :- X_1 is X - 1, X_1 == Y, sw(X_1, A), r(X_1).
e(p, X, Y, A) :- Y1 is Y + 1, p(Y, Y1), r(Y), e(p, X, Y1, A).

e(r, X, Y, _) :- X_1 is X - 1, X_1 == Y, rd(X_1), sr(X_1, S),
    p(X_1, S), wr(X_1), r(X_1).
e(r, X, Y, _) :- Y1 is Y + 1, p(Y, Y1), r(Y), e(r, X, Y1, _).

e(rd, X, Y, _) :- X_1 is X - 1, X_1 == Y, sr(X_1, S), p(X_1, S),
    rd(X_1), p(X_1, X), r(X_1).
e(rd, X, Y, _) :- Y1 is Y + 1, p(Y, Y1), r(Y), e(rd, X, Y1, _).

e(wr, X, Y, _) :- X_1 is X - 1, X_1 == Y, sr(X_1, S), p(X_1, S),
```

```
    wr(X_1), p(X_1, X), r(X_1).
e(wr, X, Y, _) :- Y1 is Y + 1, p(Y, Y1), r(Y), e(wr, X, Y1, _).

e(c, X, Y, A) :- X_1 is X - 1, X_1 == Y, c(X_1, A), r(X_1).
e(c, X, Y, A) :- Y1 is Y + 1, p(Y, Y1), r(Y), e(c, X, Y1, A).

e(l, X, Y, _) :- X_1 is X - 1, X_1 == Y, glr(X_1, V), V == true,
    rd(X_1), sr(X_1, S), p(X_1, S), wr(X_1), r(X_1).
e(l, X, Y, _) :- X_1 is X - 1, X_1 == Y, glr(X_1, V), V == false,
    glw(X_1, true), rd(X_1), sr(X_1, S), p(X_1, S), wr(X_1), l(X_1).
e(l, X, Y, _) :- Y1 is Y + 1, p(Y, Y1), r(Y), e(l, X, Y1, _).

e(on, X, Y, _) :- X_1 is X - 1, X_1 == Y, glw(X_1, false), r(X_1).
e(on, X, Y, _) :- Y1 is Y + 1, p(Y, Y1), r(Y), e(on, X, Y1, _).

sw(X, A) :- si(X, A), asserta(s(X, A)).
sr(X, A) :- s(X, A).

glw(X, V) :- asserta(gl(X, V)).
glr(X, V) :- gl(X, V).
```

Listing 3. Modèle Prolog de l'hyperviseur récursif

Références

1. Muli Ben-Yehuda, Michael D Day, Zvi Dubitzky, Michael Factor, Nadav Har'El, Abel Gordon, Anthony Liguori, Orit Wasserman, and Ben-Ami Yassour. The Turtles Project : Design and Implementation of Nested Virtualization. In *OSDI*, volume 10, pages 423–436, 2010.
2. 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`.
3. Qing He. Nested virtualization on xen. *Xen Summit Asia*, 2009.
4. Benoît Morgan, Éric Alata, and Vincent Nicomette. Tests d'intégrité d'hyperviseurs de machines virtuelles à distance et assisté par le matériel. In *Actes du 11ème symposium sur la sécurité des technologies de l'information et des communications (SSTIC)*, pages 355–382, 2014.
5. Gerald J. Popek and Robert P. Goldberg. Formal requirements for virtualizable third generation architectures. *Commun. ACM*, 17(7) :412–421, July 1974.
6. Joanna Rutkowska. Subverting vistatm kernel for fun and profit. *Black Hat Briefings*, 2006.
7. 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. IEEE, 2012.

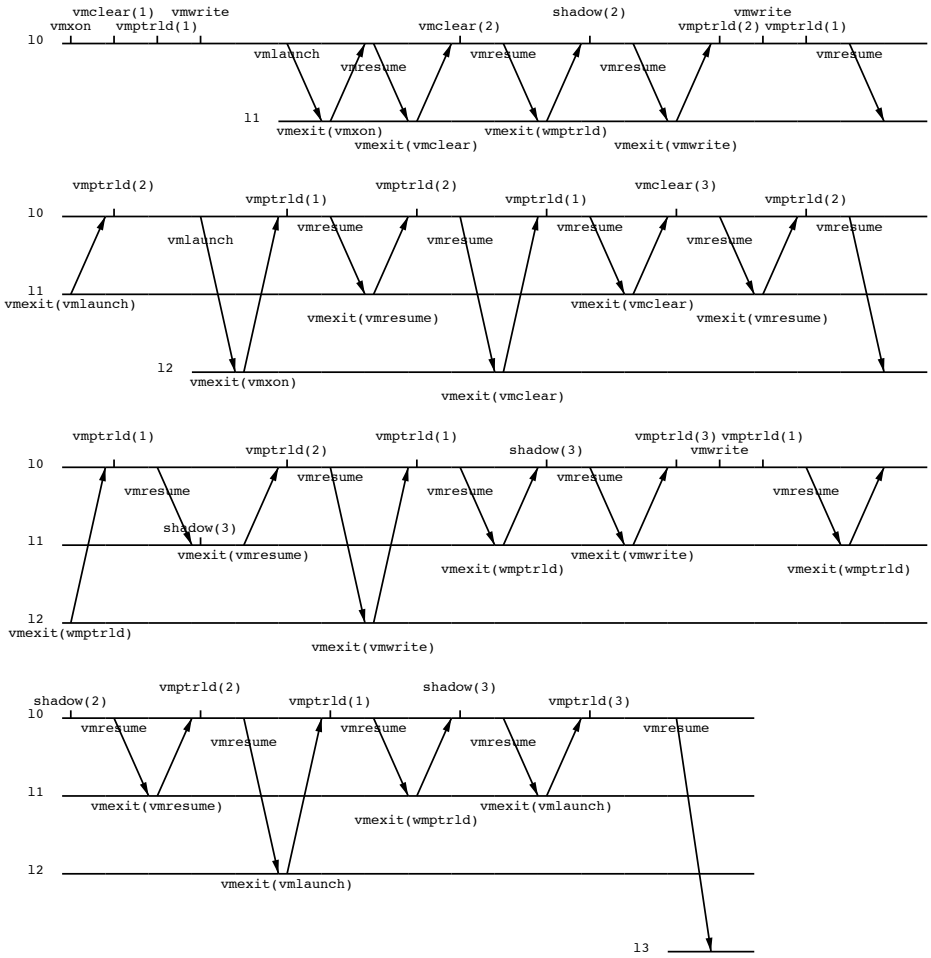

Figure 4. Démarrage de l_0, l_1 et l_2

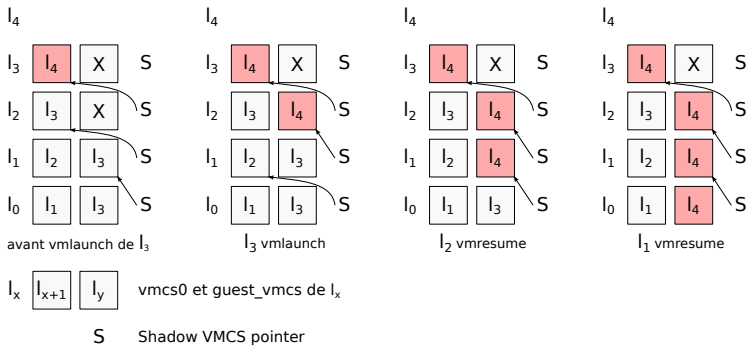

Figure 5. Démarrage simplifié de l_4 par l_3

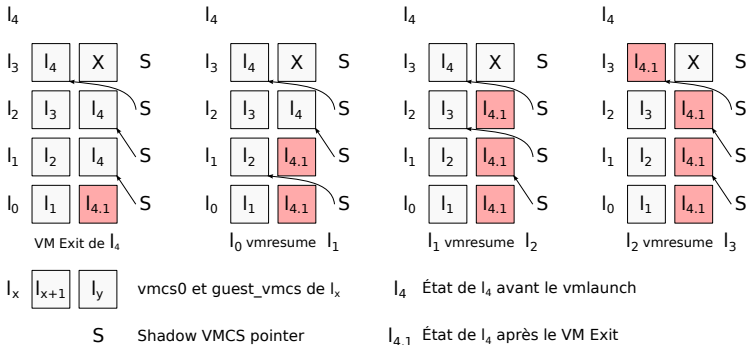

Figure 6. Remontée simplifiée de l'état à jour d'une VM à la suite d'un VM Exit jusqu'à l_3

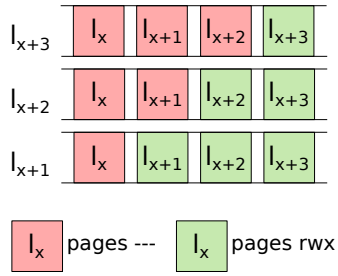

Figure 7. Configuration d'EPT par l_x en fonction du niveau ordonnancé

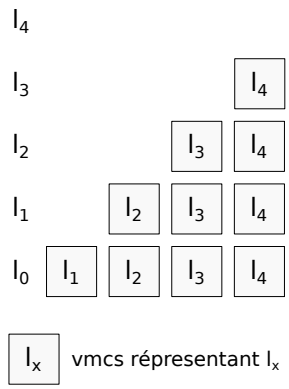

Figure 8. utilisation d'une VMCS par niveau virtualisé

```
#    N Length Source address    Dest address      Type
0009 0 0190 B b8:ca:3a:a6:76:f9 ff:ff:ff:ff:ff:ff CoreRegsData
0010 0 0042 B b4:b5:2f:fc:28:ce d4:be:d9:00:8f:3d VMCSRead
0011 0 0046 B b8:ca:3a:a6:76:f9 ff:ff:ff:ff:ff:ff VMCSData
0012 0 ---------------------------------------- Info : Page Walk
0013 0 0042 B b4:b5:2f:fc:28:ce d4:be:d9:00:8f:3d MemoryRead
0014 0 0046 B b8:ca:3a:a6:76:f9 ff:ff:ff:ff:ff:ff MemoryData
0015 0 ---------------------------------------- Info : Page Walk
0016 0 ---------------------------------------- Info : Page Walk
0017 0 0042 B b4:b5:2f:fc:28:ce d4:be:d9:00:8f:3d MemoryRead
0018 0 0046 B b8:ca:3a:a6:76:f9 ff:ff:ff:ff:ff:ff MemoryData
0019 0 ---------------------------------------- Info : Page Walk
0020 0 ---------------------------------------- Info : Page Walk
0021 0 0042 B b4:b5:2f:fc:28:ce d4:be:d9:00:8f:3d MemoryRead
0022 0 0046 B b8:ca:3a:a6:76:f9 ff:ff:ff:ff:ff:ff MemoryData
0023 0 ---------------------------------------- Info : Page Walk
0024 0 ---------------------------------------- Info : Page Walk
0025 0 0042 B b4:b5:2f:fc:28:ce d4:be:d9:00:8f:3d MemoryRead
0026 0 0274 B b8:ca:3a:a6:76:f9 ff:ff:ff:ff:ff:ff MemoryData
0027 0 0042 B b4:b5:2f:fc:28:ce d4:be:d9:00:8f:3d CoreRegsRead
0028 0 0190 B b8:ca:3a:a6:76:f9 ff:ff:ff:ff:ff:ff CoreRegsData
: 28
number : 0028
core : 0
length : 0190 B
src address : b8:ca:3a:a6:76:f9
dest address : ff:ff:ff:ff:ff:ff
Core regs :

Instruction Pointer
      rip 000000009a3fcc18
GPRs  rax 0000000000000010 rbx 000000009a425258
      rcx 00000000b77d58b0 rdx 00000000b77d58b0
      r8  00000000baf802df r9  0000000000000000
      r10 000000007ad9cf08 r11 0000000307251ec
      r12 00000000b48fe020 r13 00000000b0fa8098
      r14 00000000b48f8818 r15 00000000b6ffff18
Segment
      cs 0038 ds 0030
      ss 0030 es 0030
      fs 0030 gs 0030
Pointer
      rbp 0000000000000010 rsp 000000007ad9cf00
MODE : S MTF : OFF VPT : ON   DISASS : ON
```

Figure 9. Récupération de l'état du cœur courant

```
#    N Length Source address    Dest address      Type
0007 0 0046 B b8:ca:3a:a6:76:f9 ff:ff:ff:ff:ff:ff VMExit (VMX_PREEMPTION_TIMER_EXPIRED)
0008 0 0042 B b4:b5:2f:fc:28:ce d4:be:d9:00:8f:3d CoreRegsRead
0009 0 0190 B b8:ca:3a:a6:76:f9 ff:ff:ff:ff:ff:ff CoreRegsData
0010 0 0042 B b4:b5:2f:fc:28:ce d4:be:d9:00:8f:3d VMCSRead
0011 0 0046 B b8:ca:3a:a6:76:f9 ff:ff:ff:ff:ff:ff VMCSData
0012 0 ---------------------------------------- Info : Page Walk
0013 0 0042 B b4:b5:2f:fc:28:ce d4:be:d9:00:8f:3d MemoryRead
0014 0 0046 B b8:ca:3a:a6:76:f9 ff:ff:ff:ff:ff:ff MemoryData
0015 0 ---------------------------------------- Info : Page Walk
0016 0 ---------------------------------------- Info : Page Walk
0017 0 0042 B b4:b5:2f:fc:28:ce d4:be:d9:00:8f:3d MemoryRead
0018 0 0046 B b8:ca:3a:a6:76:f9 ff:ff:ff:ff:ff:ff MemoryData
0019 0 ---------------------------------------- Info : Page Walk
0020 0 ---------------------------------------- Info : Page Walk
0021 0 0042 B b4:b5:2f:fc:28:ce d4:be:d9:00:8f:3d MemoryRead
0022 0 0046 B b8:ca:3a:a6:76:f9 ff:ff:ff:ff:ff:ff MemoryData
0023 0 ---------------------------------------- Info : Page Walk
0024 0 ---------------------------------------- Info : Page Walk
0025 0 0042 B b4:b5:2f:fc:28:ce d4:be:d9:00:8f:3d MemoryRead
0026 0 0274 B b8:ca:3a:a6:76:f9 ff:ff:ff:ff:ff:ff MemoryData
: 26

/tmp/tmpC2ayJZ:     format de fichier binary


Dassemblage de la section .data:

9a3fcc18 <.data>:
9a3fcc18:48 89 43 08          mov    %rax,0x8(%rbx)
9a3fcc1c:33 c0                xor    %eax,%eax
9a3fcc1e:48 83 c4 20          add    $0x20,%rsp
9a3fcc22:5b                   pop    %rbx
9a3fcc23:c3                   retq
9a3fcc24:e9 c7 ff ff ff       jmpq   0x9a3fcbf0
9a3fcc29:cc                   int3
9a3fcc2a:cc                   int3
9a3fcc2b:cc                   int3
9a3fcc2c:48 ff 49 10          decq   0x10(%rcx)
9a3fcc30:48 8b 49 08          mov    0x8(%rcx),%rcx
9a3fcc34:e9 3b fa ff ff       jmpq   0x9a3fc674
9a3fcc39:cc                   int3
9a3fcc3a:cc                   int3
MODE : S MTF : OFF VPT : ON   DISASS : ON
```

Figure 10. Désassemblage du code pointé par le `rip` courant