Ramooflax

übervisor

Stéphane Duverger

EADS Innovation Works
Suresnes, FRANCE
stephane.duverger(@)eads.net

Résumé Nous présentons Ramooflax, un outil libre [1] de virtualisation. Cet outil prend la forme d'un noyau de système d'exploitation minimaliste faisant office d'hyperviseur et d'un client distant permettant un accès haut niveau aux fonctionnalités implémentées par l'hyperviseur.

L'architecture de cet outil facilite la mise en place de manière totalement transparente d'un environnement d'analyse de systèmes d'exploitation puissant, isolé, flexible et avant tout indépendant de toute couche logicielle.

La cible principale de l'outil concerne des systèmes d'exploitation installés et déjà fonctionnels sur machines de type x86 32 et 64 bits, disposant des extensions de virtualisation matérielle.

1 Introduction

Cet outil tente de mettre en avant l'usage de la virtualisation matérielle à des fins d'analyse distante de systèmes d'exploitation de la manière la plus transparente possible.

L'approche *on-the-fly* [2] de bon nombre d'hyperviseurs ne nous a pas paru convaincante en terme de transparence fonctionnelle. Le principal inconvénient venant du fait que l'hyperviseur, ou plus généralement le module de contrôle, fait partie intégrante de la mémoire du système cible et à ce titre dépend de fonctionnalités et de concepts implémentés par le système cible : modèle mémoire, ordonnancement, interruptibilité. Si dans le plus simple des cas cette dépendance peut se limiter à l'initialisation de l'hyperviseur, elle peut également dans le pire des cas compromettre sa pérennité. Bien que la virtualisation matérielle permette de protéger efficacement ce type d'hyperviseurs, elle nécessite néanmoins l'implémentation de mécanismes pouvant rapidement devenir complexes.

Notre approche semble originale en ce sens qu'elle se veut totalement indépendante du système cible. Pré-requis non négligeable, il est par contre nécessaire de démarrer

1. GPLv2
2. l'hyperviseur est chargé durant l'exécution du système cible sous forme de driver/module noyau.

notre couche logicielle avant le système cible[3]. Bien qu'a priori plus contraignante, cette approche a l'avantage de permettre un contrôle total de la vision du matériel donnée au système cible, comme par exemple une taille de mémoire vive effective plus petite que celle physiquement installée.

Cet outil a pour principaux objectifs :

– d'être léger, simple et performant
– de profiter des composants déjà présents (i.e. BIOS)
– d'être uniquement dépendant de l'architecture matérielle
– de déléguer la complexité fonctionnelle de l'analyse à une machine distante

L'analyse distante s'effectuera grâce à un framework python fournissant un accès de très haut niveau aux fonctionnalités de l'hyperviseur afin de pouvoir implémenter relativement aisément de nombreux plugins :

– débogueur distant
– reconstruction graphique de l'environnement mémoire d'un processus
– analyse comportementale d'un boot-loader
– ...

On pourrait également imaginer un plugin illustrant le fonctionnement de microprocesseurs modernes, dans un but totalement pédagogique, un peu à l'image de ce que Nate Robins[2] a réalisé pour appréhender l'API OpenGL.

Cet article présente la conception et l'implémentation du noyau minimaliste servant d'hyperviseur ainsi que le framework python permettant le développement d'outils d'analyse.

2 L'hyperviseur

2.1 Architecture cible

Notre hyperviseur supporte les processeurs de la famille x86 disposant des extensions matérielles de virtualisation récentes[4]. Par *récentes*, nous entendons faire principalement usage des dernières avancées concernant la virtualisation de la MMU, plus particulièrement EPT[5] et RVI[6].

L'intérêt de ces fonctionnalités réside dans le fait qu'elles permettent de largement simplifier le fonctionnement de l'hyperviseur tout en augmentant drastiquement les performances, en comparaison à une implémentation du type SPT[7]. La surface

3. Plusieurs scenarii sont envisageables, le plus simple à mettre en œuvre étant l'accès physique avec démarrage par clef USB.
4. Intel VT-x et AMD-V
5. Intel's Extended Page Tables
6. AMD's Rapid Virtualization Indexing, initialement Nested Page Tables
7. Shadow Page Tables

d'attaque de l'hyperviseur s'en trouve également réduite une fois déchargé de la complexité de l'implémentation des SPTs.

Tous les processeurs (CoreXX, Phenom, Athlon, ...) disposant des fonctionnalités de virtualisation matérielle ne disposent pas nécessairement de ces extensions récentes. Nous les retrouvons majoritairement dans des gammes de processeurs équipant des serveurs ou des stations de travail très performantes.

À l'échéance de la roadmap d'Intel[1] nous serons plus à même de les rencontrer dans des processeurs équipant de simples stations de travail.

2.2 Le concept Ramooflax

Le but est de permettre la virtualisation d'un système d'exploitation déjà installé sur une machine dédiée. La mise en place de la virtualisation a lieu au démarrage de la machine afin de lancer le système d'exploitation installé dans un environnement virtualisé.

Ceci permet donc de virtualiser, et ainsi d'analyser, des sytèmes d'exploita-tion disposant de leur environnement original et plus particulièrement de périphériques physiques rarement émulés par les solutions de virtualisation existantes.

L'idée est ici de démarrer l'hyperviseur via un stockage externe (clef USB), puis une fois l'hyperviseur initialisé de simplement demander au BIOS (à présent virtualisé) de démarrer le système d'exploitation installé.

2.3 Architecture

Ramooflax se présente sous la forme d'un bootloader et de trois noyaux minimalistes de systèmes d'exploitation : Loader, Setup et VMM.

Bootloader Notre chaîne de noyaux est démarrée grâce à GRUB, préalablement installé sur la clef USB. Le loader est considéré par GRUB comme le noyau de système d'exploitation, Setup et VMM comme des modules de celui-ci.

Loader Conforme au standard multiboot afin d'être chargé par GRUB, il s'occupe simplement de basculer le microprocesseur en mode 64 bits. En effet, le standard multiboot charge un OS 32 bits en mode protégé sans pagination. Notre hyperviseur fonctionne uniquement en mode 64 bits afin d'être en mesure de virtualiser des systèmes d'exploitation aussi bien 32 que 64 bits. Il charge finalement le setup et lui donne la main.

Setup Ce noyau 64 bits est responsable de l'initialisation du processeur et des fonctionnalités de virtualisation pour l'hyperviseur final. Tout le code d'initialisation n'étant plus nécessaire une fois exécuté, l'idée de le placer dans un noyau indépendant permettait de libérer cet espace mémoire inutile pendant le processus de virtualisation.

Le setup s'occupe également de récuperer la taille de la RAM et de reloger le VMM à la fin de celle-ci. En fournissant une taille de RAM inférieure à celle physiquement disponible, nous garantissons que le système d'exploitation virtualisé ne tentera pas d'allouer des pages de mémoire physique occupées par l'hyperviseur (hors attaque ciblée bien entendu).

Une fois l'hyperviseur initialisé, le setup installe dans la mémoire conventionnelle [8] l'instruction `int 0x19` et donne la main à l'hyperviseur. Celui-ci démarre son unique machine virtuelle sur l'instruction installée par le setup. Ceci a pour effet de demander au BIOS de charger le secteur d'amorçage de son premier périphérique bootable (i.e. disque dur dans la plupart des cas).

L'intérêt est encore une fois de profiter de services déjà présents dans un PC classique et largement éprouvés, ceux du BIOS.

VMM L'hyperviseur se présente sous la forme d'un binaire ELF 64 bits PIE [9]. Le code 64 bits nous permet comme précédemment signalé de virtualiser aussi bien des systèmes d'exploitation 32 que 64 bits et de profiter de tailles de RAM supérieures à 4GB sans avoir à jongler avec le PAE en 32 bits.

Le fait qu'il soit PIE nous permet de le reloger dynamiquement à la fin de la RAM. Chaque machine pouvant avoir une taille de RAM différente, ceci nous paraissait être la solution la plus élégante.

Le rôle de l'hyperviseur par la suite consiste à contrôler le comportement de la machine virtuelle sur le processeur et les périphériques. La section 5 présentera en détail les mécanismes utilisés par l'hyperviseur.

Le schéma 1 résume la séquence de démarrage de l'hyperviseur (VMM) et de sa machine virtuelle (VM).

2.4 Attrait de la solution

Nous souhaitions minimiser l'impact sur l'environnement d'exécution de la VM (être au plus proche de son *monde réel*) et privilégier au maximum la facilité de déploiement. En regardant de près le paysage des hyperviseurs, nous avons fait le constat suivant :

8. Les 640Ko de mémoire basse ... nostalgie.
9. Position Independent Executable

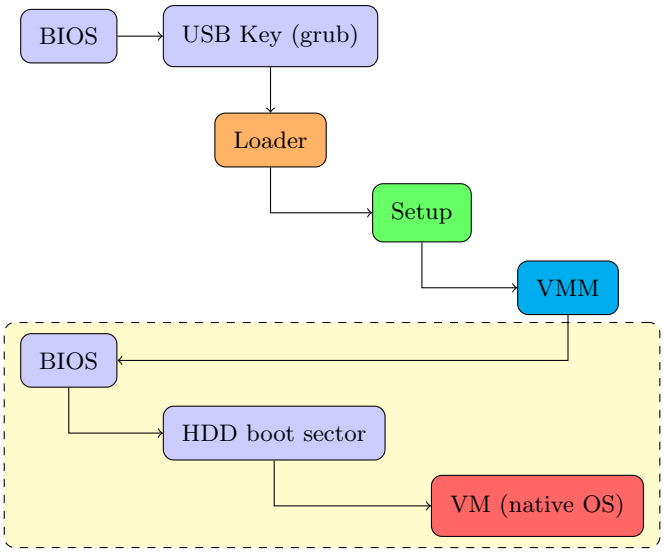

Figure 1. Chaîne de démarrage de l'OS virtualisé.

- solutions *grand public* inadaptées
  - type Xen, VirtualBox, KVM
  - trop complexes à déployer (distrib dom0, linux+userland, ...)
  - environnement trop émulé (bios minimaliste, périphériques)
- solutions trop intrusives
  - type bluepill[3], vitriol[4], virtdbg[5], hyperdbg[6], abyss[7]
  - virtualisation *in vivo* non acceptable
  - dépendance trop forte à l'OS virtualisé

Il était ainsi préférable de repartir de zéro, d'une part parce que les architectures d'hyperviseurs disponibles ne correspondaient pas à notre approche, et d'autre part parce que l'auteur n'a pas honte d'avouer qu'il se complait à réinventer la roue bien que toutefois cette architecture de virtualisation n'ait pas encore été rencontrée [10].

Notons également que tous les hyperviseurs actuels ne virtualisent que des BIOS minimalistes, très pauvres en fonctionnalités. Notre solution permet en outre d'analyser le comportement de *vrais* BIOS, modulo la limite imposée par les fonctionnalités de virtualisation matérielle. Ramooflax ambitieux !

10. À la connaissance de l'auteur

2.5 Limitations

À ce jour, Ramooflax a été testé avec succès avec Windows XP/7 Pro 32 bits et Debian GNU/Linux 5.0 32 bits. Des systèmes d'exploitation moins complexes ne devraient normalement poser aucun problème à leur virtualisation. Des noyaux Linux 64 bits ont également été virtualisés. Cependant nous préférons avancer le développement d'outils sur le framework plutôt que de fournir un hyperviseur supportant de nombreux systèmes d'exploitations tous modes confondus, mais ne disposant d'aucune fonctionnalité.

L'hyperviseur fonctionne uniquement sur des processeurs AMD pour le moment. Le portage vers les processeurs Intel est en cours de développement.

L'hyperviseur ne virtualise pour le moment qu'un seul et unique Core, ce qui n'empêche pas le système d'exploitation virtualisé d'utiliser les autres Cores disponibles. La virtualisation des Cores est relativement complexe à mettre en œuvre dans notre architecture de virtualisation de part le fait que l'initialisation des Application Processors doit être faite par l'hyperviseur afin d'y activer la virtualisation matérielle, mais également être interceptée lorsque la VM tente à son tour des les initialiser. Le but premier de l'hyperviseur étant l'analyse d'un poste physique (noyau et/ou applications), il ne nous parait pas pénalisant pour l'instant de réduire le nombre de Cores à 1 lors du démarrage de la VM, soit directement dans l'hyperviseur auquel cas la mise en œuvre peut encore une fois s'avérer complexe (interception des instructions `cpuid, rdmsr` non suffisante en raison de la configuration des tables ACPI), soit via des paramètres fournis à la VM (`/numproc`, `maxcpus`, ...).

L'hyperviseur n'implémente pas la virtualisation matérielle (*nested virtualization*). Étant donné qu'il fait usage des fonctionnalités de virtualisation, celles-ci sont masquées et ainsi rendues indisponibles aux yeux du système d'exploitation virtualisé. Nous ne prévoyons pas d'implémenter l'émulation des instructions de virtualisation matérielle.

3 La virtualisation matérielle en bref

Les extensions de virtualisation matérielle, fournies sous la forme d'un jeu d'instructions réduit et de mécanismes d'interception au sein du cpu, permettent de grandement simplifier le développement d'hyperviseurs. Bien que compatibles à l'origine, Intel et AMD ont continué à développer de manière indépendante leurs extensions de virtualisation, aboutissant à des implémentations non compatibles.

Nous présentons dans cette section, de manière non exhaustive, les approches d'Intel et d'AMD, leurs différences, leurs avantages et leurs inconvénients.

3.1 Éléments communs

Que ce soit pour l'une ou l'autre des architectures, l'exécution d'une VM sur le cpu repose principalement sur la configuration d'une structure de données (VMCS [11], VMCB [12]) permettant de préparer le contenu de registres systèmes et de mettre en place des bitmaps d'interception d'exécution d'instructions privilégiées mais également d'interception et d'injection d'évènements tels qu'exceptions, interruptions, I/O, accès aux MSRs ...

Si sur architecture AMD cette structure de données est accessible directement dans l'espace mémoire de l'hyperviseur, sur Intel il est nécessaire d'y accéder via des instructions particulières (`vmread` et `vmwrite`). Chaque champ de la structure VMCS possède un encodage spécifique permettant aux précédentes instructions d'en lire/écrire la valeur.

L'hyperviseur et la VM disposent de leur propre état des registres systèmes du cpu soit au sein de la même structure de données soit dans des structures distinctes, afin que celui-ci soit en mesure de sauvegarder/restaurer leurs états respectifs au moment où la VM démarre ou interrompt son exécution.

L'exécution/interruption d'une VM correspond respectivement à ce que l'on nomme des `vm-entry` et `vm-exit`. Notons que le cpu procède à un arsenal de vérifications de l'état de la structure de données de la VM durant ces `vm-entry` et `vm-exit`.

A titre d'exemple, la structure VMCB permet entre autres d'intercepter et configurer les éléments suivants :

- lecture/écriture des registres cr, dr, idtr, gdtr, ...
- interception de :
  - pushf, popf, cpuid, iret, int, hlt, ...
  - vmrun, vmmcall, vmload, ...
  - exception, interruptions matérielles et logicielles, smi, ...
- préparation des registres systèmes :
  - cs, ds, ..., gs (base, limite, attributs)
  - efer, cpl, rflags, cr[0-4], dr6 et dr7
  - ...

Notons qu'Intel et AMD proposent l'interception des instructions de virtualisation. Ceci permet à un VMM les interceptant d'en proposer l'émulation afin qu'une VM soit en mesure d'elle-même exécuter un hyperviseur faisant usage de ces instructions.

L'interception des instructions systèmes et des évènements s'accompagne généralement d'informations complémentaires, telles que :

11. Intel Virtual Machine Control Structure
12. AMD Virtual Machine Control Block

– le numéro de l'exception générée
– le registre cr ciblé
– le port de destination et la taille d'un `out`
– ...

Évidemment, Intel et AMD ne proposent pas les mêmes commodités lors de l'analyse d'un `vm-exit`. De plus, l'état de l'hyperviseur peut ne pas être restauré intégralement lors d'un `vm-exit` (limite de la GDT non restaurée sous Intel, LDT non restaurée sous AMD).

Malgré ces dispositifs, un hyperviseur se doit de gérer de manière logicielle les transitions de mode du cpu (réel, protégé, irréel, ...), la préparation de l'espace mémoire *physique* vu par la VM, la cohérence de l'injection des évènements (i.e. injection d'une exception durant l'interception d'une exception mène en condition normale à une double faute). De plus, la présence d'un désassembleur (certes minimaliste) afin de pouvoir effectuer une analyse plus précise des conditions d'un `vm-exit` s'avère souvent indispensable.

3.2 Intel-VT (vmx)

L'approche d'Intel s'applique parfaitement à la notion de cpu virtuel. L'hyperviseur s'exécute dans un mode dit `vmx-root` privilégié, tandis que la VM s'exécute dans un mode `vmx-nonroot`, une fois la virtualisation matérielle activée.

Au lieu de proposer des mécanismes d'interception, par exemple pour les registres de contrôle (cr0, cr4), Intel met à disposition un système de *shadowing* de ces registres. L'hyperviseur prépare à l'avance quels seront les bits qui seront modifiables par la VM, ceux qui devront être lus à telle valeur et ceux qui génèreront des `vm-exit`.

En d'autres termes :
– certains bits sont contrôlés par le VMM et génèrent des `vm-exit`
– certains bits sont lus dans une copie du registre dite *read shadow*, préparée à l'avance par le VMM
– certains bits sont à la disposition de la VM

Cette approche à l'intérêt de ne générer des `vm-exit` que sur certains bits sensibles (i.e. activation du mode protégé ou de la pagination dans cr0) et de rendre de manière transparente et sans interruption une version du registre ne reflétant pas nécessairement la réalité (i.e. la VM croit que la pagination est désactivée en lisant cr0.pg=0 alors que celle-ci est en réalité activée).

En contre-partie, la configuration de ce *shadowing* est assez sensible et dépend de nombreux éléments comme la révision du cpu ou encore les fonctionnalités de virtualisation disponibles. Intel impose par exemple que la pagination et le mode protégé soient systématiquement configurés lorsque l'on tente d'exécuter une VM.

Intel propose également des informations sur le registre général utilisé pour la lecture ou l'écriture d'un registre de contrôle. Ceci permet d'éviter le désassemblage de l'instruction.

La gestion de l'accès aux MSRs est également élégante. L'hyperviseur décide quels seront les MSRs qui reflèteront la réalité et quels seront ceux qui auront une valeur spécifique uniquement durant l'exécution de la VM. Ces MSRs spécifiques à la VM étant sauvegardé/restauré à chaque `vm-entry` et `vm-exit`.

3.3 AMD-V (svm)

La virtualisation matérielle proposée par AMD est beaucoup plus simple [13] à mettre en œuvre et malheureusement beaucoup moins subtile que sous Intel. Elle peut être vue comme une extension de mode d'exécution du cpu. Il n'existe pas de *shadowing* de registres. Toute la logique de l'hyperviseur est basée sur l'interception ou non d'accès à une ressource système.

Ainsi pour les registres de contrôle, il est seulement possible d'autoriser et/ou interdire leur lecture/écriture. Nous ne disposons pas de la granularité offerte par Intel.

Les registres systèmes chargés depuis la VMCB seront les registres systèmes utilisés par le cpu, sauf quelques exceptions notoires. Il en va de même pour les MSRs.

Si cette approche plus directe semble plus simple à mettre en œuvre, elle nécessite beaucoup plus d'analyse dans l'hyperviseur. Par exemple, les premières versions des extensions de virtualisation imposaient un désassemblage systématique des instructions interceptées de lecture et écriture des registres de contrôles afin de connaître le registre général utilisé dans l'instruction.

De même, il est assez complexe pour un hyperviseur de masquer uniquement certains bits de certains registres. Nous pensons par exemple au bit TF [14] du registre rflags. AMD met à disposition l'interception des instructions `popf` et `pushf` afin d'être en mesure de contrôler la valeur de rflags vue par la VM, mais ce mécanisme est plus complexe que le système de *shadowing* d'Intel. L'émulation d'une instruction sur architecture x86 est très sensible de par les nombreux modes de fonctionnement du cpu et ses mécanismes de protection.

3.4 Virtualisation de la MMU

Initialement, les cpu ne disposaient pas d'extensions permettant de virtualiser la MMU. Il était donc nécessaire d'implémenter ce que l'on appelle des Shadow Page

13. À titre de comparaison, la documentation ne représente qu'environ 70 pages pour AMD alors qu'elle approche les 230 pages pour Intel.
14. Trap Flag, utilisé pour le single-stepping.

Tables, mécanisme complexe permettant de simuler la présence d'une MMU entre la VM et l'hyperviseur afin de traduire les adresses physiques de la VM en adresses physiques (dites systèmes) de l'hyperviseur.

Cette approche était également très pénalisante d'un point de vue des performances, car son implémentation reposait principalement sur des fautes de pages générées par la VM du fait que l'hyperviseur maintenait de manière logicielle les tables de traduction d'adresses. Une implémentation naïve des SPTs pourrait être la suivante :

- le registre cr3 de la VM est totalement contrôlé par l'hyperviseur
- l'hyperviseur fournit le(s) répertoire(s) et les tables de pages utilisés par le cpu lors de l'exécution de la VM
- initialement (et à chaque écriture de cr3) toutes les entrées sont vierges
- à chaque faute de page, l'hyperviseur parcourt les tables de pages configurées par la VM et remplit les SPTs en conséquence
- l'hyperviseur prend soin de modifier l'adresse des pages physiques configurées par la VM par celles correspondant à l'espace physique représentant la RAM ou les *memory mapped devices* dans la mémoire du système (de l'hyperviseur).

Le schéma 2 résume ce mécanisme.

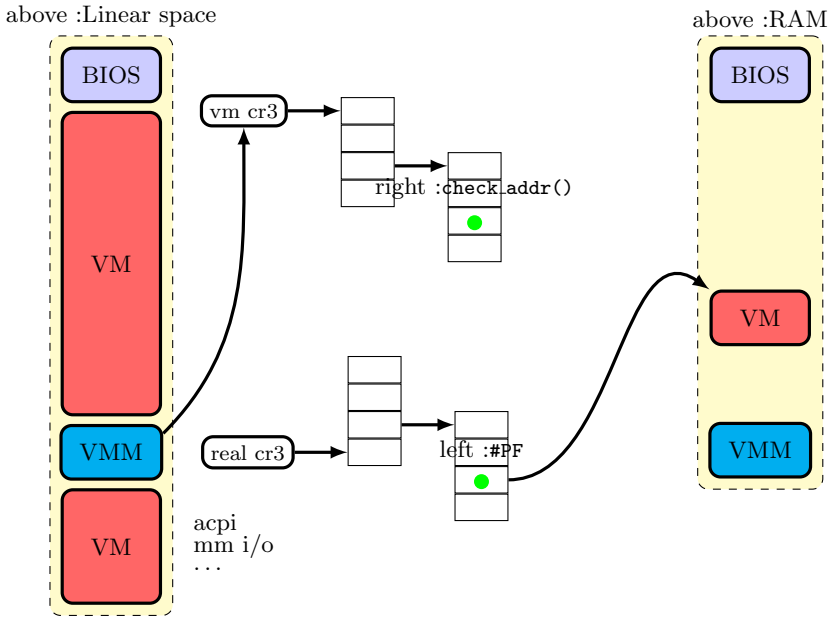

Figure 2. Shadow Page Tables.

Les cpu plus récents disposent à présent d'une MMU virtualisée. Le cpu après traduction des adresses virtuelles de la VM en adresses physiques de la VM, parcourt un autre jeu de tables de pages traduisant les adresses physiques de la VM en adresses physiques de l'hyperviseur définissant ainsi leur emplacement final.

Cette double traduction est opérée automatiquement par le cpu, d'où un gain important en terme de performances. Les secondes tables de traductions dites *nested* sont de plus configurées une seule fois par l'hyperviseur, hors transition de mode (i.e. réel/protégé).

Comparée au système des SPTs l'approche est vraiment plus simple. Elle évite à l'hyperviseur d'émuler de nombreux cas complexes de désynchroni-sation entre les TLBs du cpu, l'état des tables de pages de la VM (bits dirty, accessed) et celles maintenues par l'hyperviseur.

Le schéma 3 représente ce mécanisme.

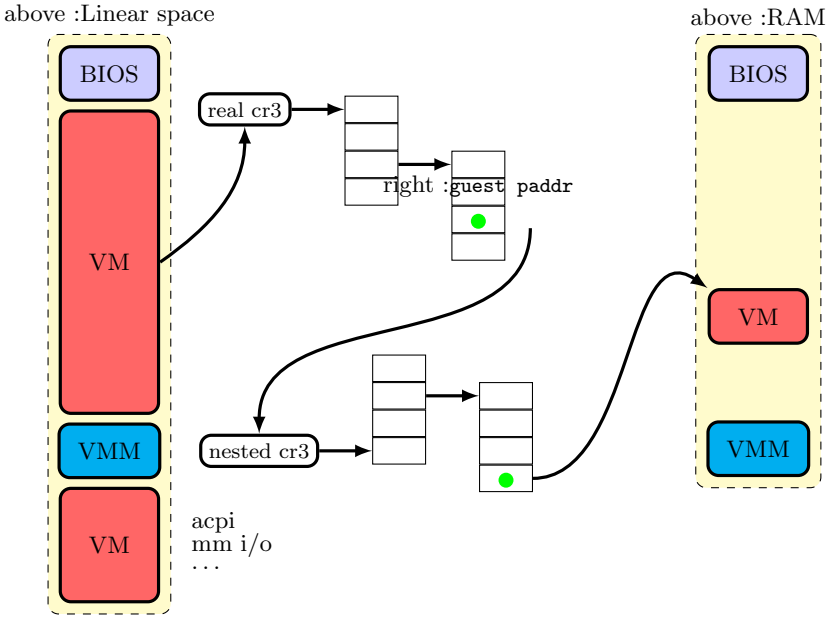

Figure 3. Nested Page Tables.

3.5 Gestion du mode (ir)réel

La virtualisation matérielle existe en réalité depuis le 80386 chez Intel. Avec l'apparition du mode protégé, Intel souhaitait tout de même offrir des commodités aux développeurs d'OS afin de permettre à des applications écrites pour s'exécuter en mode réel, de continuer à s'exécuter sans modification sous des OS s'exécutant en mode protégé. Un nouveau mode fait ainsi son apparition, le mode v8086.

Il s'agit d'un mode particulier du cpu, émulant les mécanismes du mode réel (far jumps, interruptions) pour des tâches s'exécutant en mode protégé. Ce mode est d'ailleurs assez riche en fonctionnalités, il permet notamment de facilement rediriger les interruptions et les I/O vers un handler installé dans le noyau contrôlant la tâche s'exécutant en mode v8086.

Ce n'est ni-plus ni-moins que de la virtualisation matérielle du mode réel. À ce titre, Intel préconise[15] l'utilisation du mode v8086 lorsqu'un hyperviseur souhaite virtualiser une VM tournant en mode réel. Xen lorsqu'il réalise de la *full-virtualization* sous Intel, s'appuye sur un gestionnaire de mode v8086 nommé *vmx assist*. Cette préconisation vient du fait que la virtualisation matérielle sous Intel ne permet pas l'exécution d'une VM sans que la pagination ne soit activée (et du coup le mode protégé) comme nous l'avons signalé plus haut.

Notons au passage que sous AMD, la virtualisation matérielle offre un nouveau mode de fonctionnement du cpu, appelé *paged realmode*, permettant de configurer le registre cr0 avec la pagination activée sans que le mode protégé soit activé (configuration normalement illégale). Ceci a pour conséquence de permettre à une VM de s'exécuter en mode réel mais avec une indirection sur ses accès à la mémoire physique, contrôlée par l'hyperviseur.

Ainsi, un hyperviseur souhaitant faire tourner une VM en mode réel sous Intel se doit non seulement de gérer des structures de données propres à la virtualisation matérielle (VMCS) mais doit également mettre en place toutes les structures de données permettant à une tâche en mode v8086 de s'exécuter : gdt, idt, tss et bitmaps de redirection, ... Tout ceci complexifie très largement la tâche comparée à ce que propose AMD.

Dernier point, un hyperviseur reprenant l'exécution d'une VM en mode v8086 est sujet à de nombreuses vérifications effectuées par le cpu, notamment concernant la configuration des registres de segments.

Petit rappel sur la segmentation Les registres de segments possèdent une partie visible et accessible au développeur, ainsi qu'une partie interne gérée par le cpu.

15. Intel Volume 3B Section 27.2

La partie visible correspond à un sélecteur d'une longueur de 16 bits ayant une signification différente selon le mode d'exécution du cpu. La partie cachée correspond à un ensemble de valeurs permettant de contrôler les accès mémoire effectués à partir de ce registre de segment. On retrouve une adresse de base dans l'espace linéaire, une limite, ainsi que divers attributs.

En mode protégé, un sélecteur de segment contient grossièrement un index dans une table de descripteurs de segments. Chaque descripteur de segment permet de remplir la partie cachée du registre correspondant.

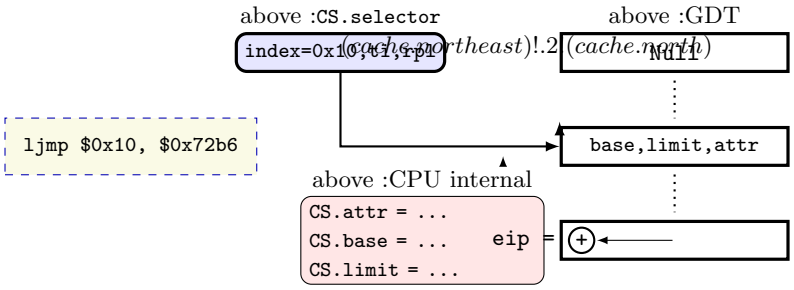

Figure 4. *Far jump* en mode protégé.

Par exemple, l'instruction `mov 0x1234, %ax` récupérera les 2 octets situés à l'adresse linéaire `ds.base_address + 0x1234`. L'adresse de base du segment référencé via le registre ds provenant du descripteur de segment dont l'index se trouve dans le sélecteur de segment de ds.

En mode réel, il n'existe pas de tables de descripteurs de segments. De plus, l'adressage mémoire s'effectuant sur 20 bits (hors GateA20), les sélecteurs de segments permettent d'ajouter les 4 bits manquants aux 16 bits d'offset présents dans les registres généraux utilisés pour les adressages mémoire.

L'instruction précédente va donc récupérer les 2 octets situés à l'adresse linéaire `ds.selector<<4 + 0x1234`. Pour des raisons évidentes de performances, chaque fois qu'un registre de segment est modifié en mode réel, le cpu met en cache dans le champ interne `base_address` du registre de segment correspondant, cette nouvelle valeur multipliée par 16.

Mode irréel La partie interne des registres de segments est donc normalement ignorée des développeurs du mode réel, car aucun mécanisme *officiel* ne permet de la configurer. À ce titre, au démarrage du cpu, l'adresse de base et la limite des registres

de segments de données sont respectivement définies à 0 et `0xffff`. Étant donné que les registres généraux sont sur 16 bits, les offsets dans les segments ne peuvent donc excéder 2^{16} octets. Ce qui justifie une limite de `0xffff`.

Néanmoins, le mode réel n'empêche pas l'usage d'un préfixe d'adresse permettant l'utilisation de registres généraux de 32 bits et donc d'offsets de 32 bits. Dans leur configuration initiale, les registres de segments ayant une limite de `0xffff`, tout accès au-delà de cette limite génèrerait une faute du type General Protection.

Nous avons précédemment vu que le mode protégé permettait de définir le contenu de la partie interne des registres de segments. Que se passe t'il lorsque l'on passe du mode protégé au mode réel ? Intel préconise de recharger les registres de segments avec des descripteurs de segments de code et de données en mode 16 bits, avant de désactiver le mode protégé dans cr0.

Excédés par les limitations d'adressage mémoire imposées par le mode réel mais ne souhaitant pas nécessairement réécrire leurs applications pour le mode protégé, certains programmeurs ont profité de ce mécanisme de *cache interne* des registres de segments. En passant en mode protégé et en chargeant les registres de segments avec comme adresse de base 0 et comme limite 4Go, puis en retournant directement en mode réel sans recharger les registres de segments, il était désormais possible via un préfixe d'adresse d'accéder à 4Go de mémoire linéaire depuis du code en mode réel. Ce mode de fonctionnement (*unreal mode*) est encore aujourd'hui très utilisé par les BIOS.

L'échec d'Intel Dans une démarche de cohérence de l'état interne des registres de segments, lorsqu'un hyperviseur reprend l'exécution d'une VM en mode v8086, le cpu vérifie [16] entre autres que la `base_address` d'un registre de segment corresponde bien à la valeur contenue dans son sélecteur multipliée par 16.

Ceci a pour conséquence de rendre impossible l'exécution d'une VM en mode irréel en utilisant le mode v8086. Le code suivant extrait d'un BIOS moderne en est la parfaite illustration :

16. Intel Volume 3B Section 23.3.1.2

```
seg000:F7284          mov     bx, 20h
seg000:F7287          cli
seg000:F7288          mov     ax, cs
seg000:F728A          cmp     ax, 0F000h
seg000:F728D          jnz     short near ptr unk_7297

seg000:F728F          lgdt    fword ptr cs:byte_8163     (1)
seg000:F7295          jmp     short near ptr unk_729D
seg000:F7297          lgdt    fword ptr cs:byte_8169

seg000:F729D          mov     eax, cr0
seg000:F72A0          or      al, 1
seg000:F72A2          mov     cr0, eax                   (2)
seg000:F72A5          mov     ax, cs
seg000:F72A7          cmp     ax, 0F000h

seg000:F72AA          jnz     short near ptr unk_72B1
seg000:F72AC          jmp     far ptr 10h:72B6h          (3)
seg000:F72B1          jmp     far ptr 28h:72B6h

seg000:F72B6          mov     ds, bx                     (4)
seg000:F72B8          mov     es, bx
seg000:F72BA          mov     eax, cr0
seg000:F72BD          and     al, 0FEh
seg000:F72BF          mov     cr0, eax                   (5)

seg000:F72C2          mov     ax, cs
seg000:F72C4          cmp     ax, 10h                    (6)
seg000:F72C7          jnz     short near ptr unk_72CE
seg000:F72C9          jmp     far ptr 0F000h:72D3h
seg000:F72CE          jmp     far ptr 0E000h:72D3h
```

En (1) et (2), le BIOS effectue son passage en mode protégé après avoir chargé une nouvelle GDT. Une fois le mode protégé activé, le BIOS initialise les composants internes du registre de segment cs en effectuant un *far jump* en (3). Il en profite également pour charger ses segments de données en (4). Puis il revient en mode réel en (5) et effectue un test sur la valeur du sélecteur de cs en (6). En situation normale, il devrait y avoir 0x10 dans cs, cependant dans le cas d'une virtualisation matérielle sous Intel, si l'adresse de base du registre de segment chargée en mode protégé par le *far jump* en (3) n'est pas 0x100, alors l'hyperviseur sera incapable de reprendre l'exécution de ce BIOS virtualisé en mode v8086 avec un sélecteur de segment de code valant 0x10.

La seule solution face à ce genre de code étant d'émuler les mécanismes du mode réel en mode protégé. La tâche est assez complexe et surtout très pénalisante d'un point de vue des performances. Chaque accès à un registre de segment doit être intercepté par l'hyperviseur afin d'émuler les *far call/jump*, *mov/pop seg*, *iret* du mode réel. Le mécanisme des interruptions, différent du mode protégé, doit également être émulé. La virtualisation matérielle ne permettant pas l'interception des accès aux registres de segments, une solution consiste par exemple à forcer la limite de la GDT

et de l'IDT à 0 afin de générer des General Protection Fault à chaque écriture dans un registre de segment ou apparition d'un interruption matérielle et/ou logicielle et d'émuler l'instruction ou l'évènement responsable de la faute.

Récemment, Intel a comblé son retard par rapport à AMD en proposant un mode *unrestricted guest* permettant à un hyperviseur de reprendre l'exécution d'une VM en mode réel. La pagination et le mode protégé étant désactivés, ce mode n'est utilisable que lorsque le cpu propose un mécanisme de virtualisation matérielle de la MMU afin de pouvoir protéger l'espace mémoire de l'hyperviseur.

3.6 Interception d'évènements

Les extensions de virtualisation matérielle permettent d'intercepter mais également d'injecter des évènements de type exception, interruptions matérielles et logicielles.

Si le mécanisme d'injection permet d'avoir la granularité la plus fine possible (le numéro de vecteur), le mécanisme d'interception est quand à lui beaucoup plus sommaire. Mise à part l'interception d'exceptions, pour lesquelles il est possible de configurer une bitmap au vecteur près, l'interception des interruptions est du type *activée/désactivée*.

Avec cette approche, l'hyperviseur est ainsi contraint d'intercepter toutes les interruptions pouvant survenir, ce qui peut induire une latence inacceptable dans le cas de l'interruption d'horloge. Des mécanismes plus complexes permettent d'intercepter plus finement les interruptions, notamment en mettant en place une *shadow* IDT, mais il est tout de même dommage qu'une bitmap n'ait pas été prévue pour chaque entrée possible de l'IDT.

Pour finir sur les interruptions matérielles, la virtualisation sous AMD conserve l'interruption matérielle *pending* lorsqu'elle est interceptée. L'hyperviseur est ainsi obligé d'activer les interruptions au sein de son propre code et d'avoir au préalable configuré une IDT complète s'il veut être en mesure de déterminer le vecteur de l'interruption survenue. Autre point conséquent, le vecteur d'entrée d'IDT d'une interruption matérielle est obtenu à partir de la ligne d'IRQ activée et d'une valeur configurée dans les (A)PICs. Si un hyperviseur ne souhaite pas nécessairement émuler un (A)PIC, il doit par ailleurs intercepter toute tentative de configuration de ce dernier afin de déterminer la valeur à ajouter au numéro d'IRQ afin d'injecter une valeur de vecteur d'interruption dans l'IDT qui soit correcte.

Intel a été plus clément, en permettant d'effectuer ou non le cycle d'*acknowledgement* (récupération de la valeur de base entre le cpu et le contrôleur d'interruptions) lors de l'interception des interruptions. Si l'interruption est déjà *acknowledged*, l'hyperviseur a uniquement à injecter le vecteur fourni par le cpu lors du `vm-exit`.

Terminons en beauté avec la gestion de l'interception des interruptions logicielles (`int xx`). Encore une fois, elle est du type *activée/désactivée* sous AMD, avec (sauf depuis peu) obligation de désassembler l'instruction pour en déterminer le vecteur.

Par contre sous Intel, il n'est pas possible [17] d'intercepter les interruptions logicielles via un paramétrage de la VMCS. Alors que le mode v8086 mettait en place, il y a de nombreuses années, des bitmaps parfaitement adaptées à l'interception de tels évènements, la virtualisation matérielle *moderne* ne fournie aucune commodité. Charge encore à l'hyperviseur de mettre en place des mécanismes détournés permettant de les intercepter.

3.7 Cas particulier des SMIs

La virtualisation matérielle permet d'intercepter les SMIs [18] que ce soit sous Intel ou sous AMD.

Sous AMD, le mécanisme d'interception est le même que pour les interruptions matérielles, celles-ci sont gardées *pending* jusqu'à ce que l'hyperviseur décide d'activer les interruptions (dans son contexte) pour les traiter, auquel cas le processeur passe immédiatement en mode SMM afin de traiter la SMI.

D'une manière générale AMD spécifie qu'il est préférable de ne pas intercepter les SMIs afin que le mode SMM puisse les traiter dans toutes les situations. Malgré cela, des *Errata* [19] dans l'implémentation de la virtualisation matérielle font qu'il peut s'avérer nécessaire d'intercepter les SMIs.

Les sources de SMIs peuvent être liées à des interruptions ou des opérations d'entrée/sortie. Dans ce dernier cas, AMD signale [20] qu'il est nécessaire de *containeriser* le SMM avant de le laisser traiter la *pending* SMI, c'est à dire exécuter le mode SMM dans une machine virtuelle dédiée.

Cependant, le fait de *containeriser* le SMM implique de devoir modifier des MSRs liés au mode SMM, qui dans la grande majorité des cas sont bloqués par le BIOS une fois que celui-ci les a configuré. La virtualisation matérielle n'apporte pas un niveau de contrôle suffisant vis-à-vis du mode SMM, du moins tant que les développeurs de BIOS protégeront *correctement* l'accès à ses MSRs.

Des BIOS buggués peuvent ainsi avoir un comportement dommageable pour l'hyperviseur. Nous avons par exemple rencontré des BIOS qui lorsque l'on interceptait les SMIs ne rendaient jamais la main à l'hyperviseur ou bien ne réactivaient pas la fonctionnalité de Last Branch Record.

17. À la connaissance de l'auteur
18. System Management Interrupts
19. Errata 342, les SMIs non interceptées peuvent désactiver les interruptions dans la VM[11]
20. AMD Manual Vol.2 section 15.22.3

4 Organisation de Ramooflax

Ramooflax est développé en langage C et en assembleur. La partie client a été intégralement développée en python.

Ramooflax dispose d'une interface de configuration pour les quelques options présentes actuellement, comme le choix de l'architecture matérielle, les périphériques utilisés pour le debugging et la prise de contrôle distante, ainsi qu'un mode *proxy*.

Ce mode permet d'intercepter certains évènements (accès aux MSRs notamment), d'en prendre note le cas échéant et de les rejouer nativement. Par défaut l'instruction `cpuid` fonctionne de cette façon du fait de la nécessité de masquer certaines fonctionnalités. Il pouvait toutefois s'avérer utile de détecter quels sont les MSRs qui entrent en compte dans l'initialisation d'un système d'exploitation moderne.

Ci-dessous quelques captures d'écran du menu de configuration de Ramooflax.

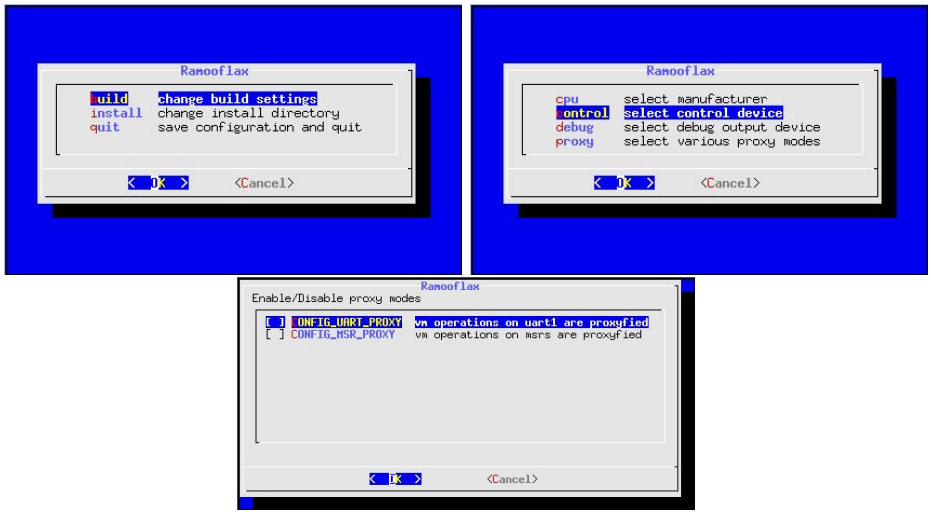

Figure 5. Menu de configuration de Ramooflax.

Le volume de code de Ramooflax est peu élevé, et se trouve résumé dans la sortie suivante :

```
riot(~) sloccount ramooflax
[...]
SLOC    Directory       SLOC-by-Language (Sorted)
11356   vmm             ansic=11093,asm=263
6767    include         ansic=6767
1898    common          ansic=1898
1443    setup           ansic=1362,asm=81
1391    client          python=1391
177     tools           sh=150,ansic=27
102     loader          ansic=92,asm=10

Totals grouped by language (dominant language first):
ansic:        21239 (91.81%)
python:        1391 (6.01%)
asm:            354 (1.53%)
sh:             150 (0.65%)
[...]
```

On y retrouve les 3 modules composant l'hyperviseur ainsi que le client et divers outils. Chaque module dispose de son propre répertoire de développement/compilation, hormis les éléments communs aux différents modules (libc, drivers). Nous avons tenté d'être le plus générique possible. Les éléments spécifiques à chaque architecture, Intel et AMD, sont localisés respectivement dans src/vmx et src/svm.

L'hyperviseur final (VMM) est organisé en sous-systèmes responsables de fonctionnalités bien précises :

```
riot(vmm) ll src/
total 32K
drwxr-xr-x 2 stf stf 4.0K Mar  3 13:37 control
drwxr-xr-x 2 stf stf 4.0K Mar  3 13:37 core
drwxr-xr-x 2 stf stf 4.0K Mar  3 13:37 devices
drwxr-xr-x 2 stf stf 4.0K Mar  3 13:37 disasm
drwxr-xr-x 2 stf stf 4.0K Mar  3 13:37 drivers
drwxr-xr-x 2 stf stf 4.0K Mar  3 13:37 libc
drwxr-xr-x 2 stf stf 4.0K Mar  3 13:37 svm
drwxr-xr-x 2 stf stf 4.0K Mar  3 13:37 vmx
```

5 Initialisation de Ramooflax

Comme nous l'avons expliqué en section 2.2, Ramooflax est architecturé autour de trois noyaux minimalistes ou modules, un pour le démarrage, un pour la configuration et un module résidant chargé de la virtualisation du système natif.

5.1 Ramooflax loader

Il s'agit du premier module chargé par GRUB. Conforme au standard multiboot[8], ce noyau 32 bits est chargé en mémoire physique à 2Mo et a pour rôle principal de préparer l'exécution du module de configuration.

Il met en place une GDT temporaire contenant à la fois des descripteurs de segments 32 et 64 bits, et un modèle de pagination de type *identity mapping* où les adresses virtuelles correspondent directement aux adresses physiques en utilisant des pages mémoire de 1Go ou 2Mo selon les fonctionnalités disponibles dans le cpu. La pagination devant être obligatoirement active pour passer en *longmode*.

Une bibliothèque C minimaliste ayant été developpée spécifiquement pour Ramooflax, elle permet en outre au loader de charger le module ELF 64 bits de configuration.

5.2 Ramooflax setup

Le module de configuration, comme son nom l'indique, est en charge de toute la configuration du poste physique avant l'activation de la virtualisation. Il s'occupe de l'initialisation de certains périphériques, notamment ceux de debugging et de contrôle de l'hyperviseur. En l'occurence, l'UART ou l'EHCI Debug port comme nous le détaillerons dans la section 10.

En dehors des fonctionnalités de virtualisation, le setup prépare l'environ-nement d'exécution de l'hyperviseur final, à commencer par son emplacement physique.

L'idée de Ramooflax était de perturber le moins possible l'environnement natif du système pré-installé et d'éviter de mettre en place des mécanismes complexes de protection de l'hyperviseur une fois la VM démarrée. Nous avons opté pour un hyperviseur PIE capable d'être ainsi relogé à n'importe quelle adresse physique, notamment à la fin de la RAM disponible.

Plus précisément, l'hyperviseur est chargé à `taille(RAM) - taille(VMM)`. La taille de la RAM est récupérée grâce à GRUB. Si la machine dispose de plus de 4Go de mémoire physique, l'hyperviseur sera toujours relogé en dessous de 4Go.

Afin de rendre une taille de RAM inférieure à ce qui est réellement disponible, le setup pépare des SMAPs, qui sont une sorte de cartographie mémoire fournie par le BIOS, modifiées spécialement pour la VM. L'hyperviseur sera en charge de l'interception de la consultation de ces SMAPs par la VM afin de fournir celles préparées par le setup et non celles du BIOS.

Chaque entrée de SMAPs est composée d'une adresse de base, d'une taille et d'un type définissant la nature de la zone mémoire concernée (ACPI, reservée, libre).

Le setup ne modifie qu'une seule entrée, celle concernant le premier *chunk* de mémoire utilisable supérieur à 1Mo et inférieur à 4Go. Généralement cette entrée est

située juste avant les zones de mémoires dédiées aux tables ACPI. La modification consiste simplement à soustraire au champ taille de l'entrée existante, la taille de la zone mémoire occupée par l'hyperviseur.

L'extrait de logs suivant, provenant d'un noyau Linux, détaille les SMAPs fournies par le BIOS. L'entrée en question à modifier aurait été la 3$^{\text{ème}}$:

```
BIOS-provided physical RAM map:
 BIOS-e820: 0000000000000000 - 000000000009bc00 (usable)
 BIOS-e820: 000000000009bc00 - 00000000000a0000 (reserved)
 BIOS-e820: 0000000000100000 - 00000000dd04d400 (usable)
 BIOS-e820: 00000000dd04d400 - 00000000dd04f400 (ACPI NVS)
 BIOS-e820: 00000000dd04f400 - 00000000e0000000 (reserved)
 BIOS-e820: 00000000f8000000 - 00000000fc000000 (reserved)
 BIOS-e820: 00000000fec00000 - 00000000fec10000 (reserved)
 BIOS-e820: 00000000fed18000 - 00000000fed1c000 (reserved)
 BIOS-e820: 00000000fed20000 - 00000000fed90000 (reserved)
 BIOS-e820: 00000000feda0000 - 00000000feda6000 (reserved)
 BIOS-e820: 00000000fee00000 - 00000000fee10000 (reserved)
 BIOS-e820: 00000000ffe60000 - 0000000100000000 (reserved)
 BIOS-e820: 0000000100000000 - 0000000120000000 (usable)
```

Nous verrons plus loin les mécanismes mis en œuvre par l'hyperviseur pour contrôler les accès de la VM aux SMAPs.

Le setup prépare également un allocateur de pages de mémoire physique très simpliste pour de futures allocations dynamiques. Notre hyperviseur ne peut en aucun cas faire usage des services de l'OS virtualisé. Ramooflax indépendant !

La GDT, l'IDT et les structures nécessaires à la mise en place de la pagination sont également relogées à la fin de la mémoire physique là où s'exécutera l'hyperviseur final. Notons que le setup prépare un ensemble de tables de pages pour le mode réel et un pour le mode protégé. Les transitions mode réel/mode protégé sont très fréquentes durant l'initialisation de la VM, il nous paraissait plus simple et plus performant de n'avoir qu'à changer un entrée de PML4. De plus le mode réel impose la mise en place de *mappings* assez complexes pour l'émulation de la Gate A20.

La mise en place des Nested Page Tables de la VM, prévoit bien entendu l'exclusion de la zone de mémoire physique où se trouve l'hyperviseur final. Les entrées de tables de pages correspondantes sont marquées comme non présentes. Les autres entrées sont configurées en *identity mapping*, car la correspondance entre les adresses physiques de la VM et celles du système réel est directe.

Finalement, le setup prépare les structures de données liées à la virtualisation matérielle. Les bitmaps d'interception des I/O et des MSRs ne concernent que le contrôleur clavier, le contrôleur système ps2 ainsi que le MSR EFER. Les accès aux registres de contrôle, les instructions sensibles comme `cpuid, hlt, intn` et toutes les instructions liées à la virtualisation sont interceptées.

Par défaut les exceptions et les interruptions matérielles ne sont pas interceptées.

Le setup termine son exécution en installant dans la mémoire conventionnelle les premières instructions qui seront exécutées par la VM : `int 0x16` et `int 0x19`.

La première est un service du BIOS attendant une interruption clavier, la seconde demande au BIOS de charger le secteur d'amorçage du premier périphérique bootable configuré (généralement un disque dur sur lequel se trouve installé le système natif).

En procédant ainsi, nous profitons des fonctionnalités présentes dans le code du BIOS (accès aux périphériques USB, SATA, ...). L'hyperviseur virtualise de manière transparente du code s'exécutant en mode réel qu'il provienne du BIOS ou non.

6 Modèle d'exécution de Ramooflax

Le schéma 6 présente l'architecture des différents chemins d'exécution de l'hyperviseur.

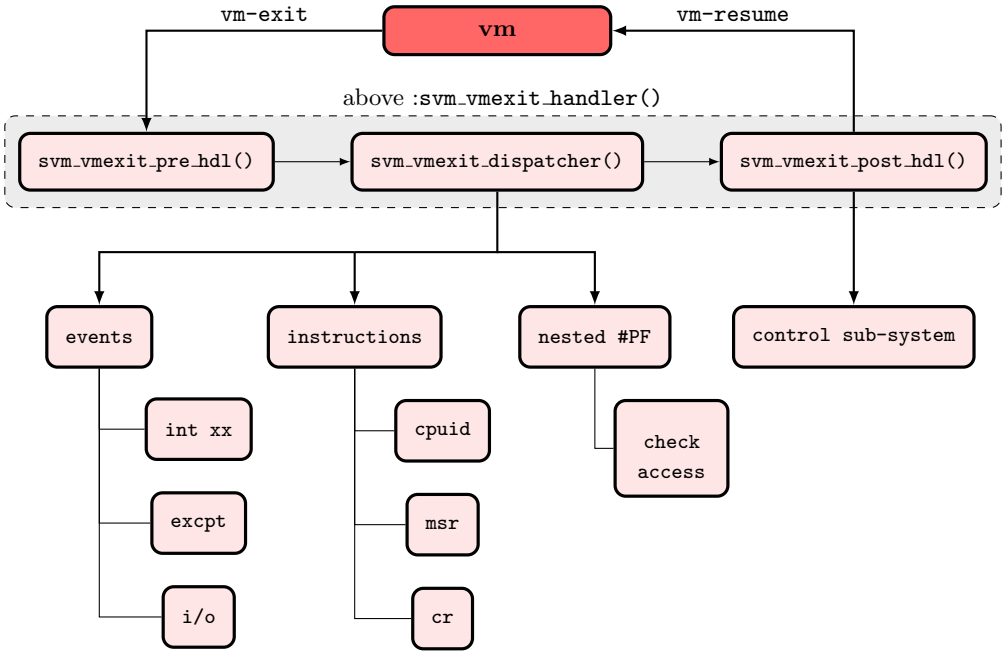

Figure 6. Chemins d'exécution de l'hyperviseur.

L'hyperviseur va passer son temps à attendre des `vm-exit` qui lui donneront une chance de s'exécuter.

```
void svm_vmexit_handler(raw64_t tsc)
{
  svm_vmexit_pre_hdl();
  svm_vmexit_dispatcher())
  svm_vmexit_post_hdl(tsc);
}

void svm_vmexit_pre_hdl()
{
  vmcb_ctrls_area_t *ctrls = &info->vm.cpu.vmc->vm_vmcb.ctrls_area;
  vmcb_state_area_t *state = &info->vm.cpu.vmc->vm_vmcb.state_area;

  svm_vmsave(&info->vm.cpu.vmc->vm_vmcb);
  info->vm.cpu.gpr->rax.raw = state->rax.raw;
  info->vm.cpu.gpr->rsp.raw = state->rsp.raw;

  if(ctrls->tlb_ctrl.tlb_control != VMCB_TLB_CTL_NONE)
     ctrls->tlb_ctrl.tlb_control = VMCB_TLB_CTL_NONE;
}

void svm_vmexit_post_hdl(raw64_t tsc)
{
  vmcb_state_area_t *state = &info->vm.cpu.vmc->vm_vmcb.state_area;

  vmm_ctrl();
  db_post_hdl();

  state->rax.raw = info->vm.cpu.gpr->rax.raw;
  state->rsp.raw = info->vm.cpu.gpr->rsp.raw;
  info->vm.cpu.gpr->rax.raw = (offset_t)&info->vm.cpu.vmc->vm_vmcb;

  info->vmm.ctrl.vmexit_cnt.raw++;
  svm_vmexit_tsc_rebase(tsc);
}
```

On retrouve un pré-traitement spécifique à l'architecture, ensuite un appel au sous-système capable de gérer le `vm-exit`, suivi d'un post-traitement s'occupant entre autre de vérifier si le client distant souhaite ou non prendre le contrôle de l'hyperviseur avant de rendre la main à la VM.

7 Filtrage des registres systèmes

7.1 Control Registers

Les accès aux registres cr0, cr3 et cr4 sont interceptés car ils contrôlent des mécanismes sensibles comme les transitions de modes d'exécution, le contrôle des TLBs et différentes options liées à la pagination et à la cohérence des caches.

En ce qui concerne la gestion des TLBs, il est nécessaire de conserver le comportement normal du cpu et de l'émuler correctement. À savoir :

– une écriture de cr3 provoque un flush des TLBs non globaux [21]
– une modification des bits PAE, PSE, PGE de cr4 provoque un flush de tous les TLBs

Depuis l'apparition des Nested Page Tables, les TLBs sont taggués à l'aide d'ASIDs [22], ce qui permet de ne flusher que les TLBs correspondant à la VM. Encore une fois, ces fonctionnalités ne sont pas présentes dans tous les cpus proposant les extensions de virtualisation matérielle.

Un autre point important est celui de la gestion du cache via le bit CD de cr0. Le noyau Linux l'active (désactivation des caches) durant son initialisation et désactive les MTRRs afin de ne plus faire aucune référence aux caches du cpu. Notre hyperviseur fonctionnant de manière la plus transparente possible, subit également la désactivation des MTRRs (plutôt que de l'émuler) et afin de rester cohérent se doit également de suivre le paramétrage de cr0 imposé par la VM en ce qui concerne la gestion des caches.

Ci-dessous un scénario classique de désactivation des caches sous Linux :

```
<0x67aaf:0xc1013d68:124>rdmsr 0x2ff | 0x0 0xc00
<0x67ab0:0xc1013d68:124>rdmsr 0xc0010010 | 0x0 0x160600
<0x67ab1:0xc1013d32:16>cache disable
<0x67ab2:0xc1013d54:137>wbinvd
<0x67ab4:0xc1013d68:124>rdmsr 0x2ff | 0x0 0xc00
<0x67ab5:0xc1013d79:124>wrmsr 0x2ff | 0x0 0x0
<0x67ab5:0xc1013d79:124>disabling mtrr
```

Les traces sont générées grâce à l'hyperviseur et son mode *proxy msr*. Linux désactive les caches via cr0, vide les lignes de cache et finalement désactive les MTRRs une fois qu'il sait que toutes les lignes de caches sont invalides. Sans entrer dans les détails de la gestion du cache mémoire sous x86, les MTRRs permettent d'associer des stratégies de cache différentes (*writeback*, *writethrough*, *uncacheable*) pour des zones mémoires fixes ou variables. Leur désactivation implique que toute la mémoire devient *uncacheable*.

Si l'hyperviseur en interceptant l'écriture de cr0, ne prend pas soin de désactiver lui aussi le cache via son propre cr0, il risque de continuer à mettre en cache de l'information, et cela après que la VM a pourtant vidé les lignes de cache. La désactivation des MTRRs s'effectuera à un point où l'hyperviseur aura encore des données en cache. Celles-ci seront perdues car les lignes de caches ne seront pas explicitement écrites en mémoire.

L'utilisation des Nested Page Tables implique également une gestion pointilleuse, depuis l'hyperviseur, de la configuration des caches par la VM notamment via les

21. Ceux dont les entrées de tables de pages n'ont pas le bit `global` positionné.
22. Address Space IDentifiers

MTRRs et le PAT. Plutôt que d'émuler ces mécanismes, l'hyperviseur suit la stratégie de cache imposée par la VM.

7.2 CPUID et MSRs

Comme nous l'avions expliqué précédemment, l'hyperviseur est capable de fonctionner en mode *passthrough*, où il laisse l'accès complet à la VM, ou bien de proxyfier un accès. Dans ce cas, il doit tout de même garantir l'exécution de l'accès à la ressource.

L'écriture d'un MSR implique :
- filtrage du msr selon l'architecture
- émulation si ce MSR est présent dans la VMCS/VMCB
- exécution native dans le cas contraire

Dans le cas d'un `cpuid` ou d'un `rdmsr`, l'interception se déroule comme suit :
- exécution native ou lecture dans la VMCS/VMCB
- post-traitement pour filtrer l'information

```
static int __resolve_cpuid()
{
  uint32_t idx = info->vm.cpu.gpr->rax.low;

  __resolve_cpuid_native();  /* native execution */
  __resolve_cpuid_arch(idx); /* amd/intel centric post-processing */

  /* generic post-processing */
  switch(idx)
  {
    case CPUID_FEATURE_INFO:
      __resolve_cpuid_feature();
      break;
    default:
      break;
  }

  return CPUID_SUCCESS;
}
```

8 Filtrage d'évènements

8.1 Interruptions logicielles

L'interception des interruptions logicielles a pour le moment lieu uniquement en mode réel. Elle permet de contrôler les requêtes faites au BIOS concernant la configuration mémoire du système (les fameuses SMAPs dont nous parlions précédemment), en l'occurence l'interruption 0x15 et les services liés à la GateA20, aux SMAPs ou

à toute autre information relative à la taille de la mémoire. Lors de l'interception, s'il s'agit d'un `int 0x15`, le mécanisme est émulé pour fournir des informations spécifiques à la VM. Dans le cas contraire, l'hyperviseur émule l'interruption en redirigeant l'exécution vers le bon handler.

8.2 Interruptions matérielles

Les interruptions matérielles ne sont pas interceptées. Mais il est possible de le faire. L'hyperviseur peut se rendre interruptible afin, dans le cas d'AMD, de détecter quel vecteur d'interruption injecter à la VM.

8.3 Exceptions

Dans le cas d'AMD, qui dispose de l'interception des interruptions logicielles, la gestion des exceptions est assez simple. L'hyperviseur vérifie simplement que l'exception qui survient n'est pas due au sous-système de contrôle (#DB et #BP pour la gestion des breakpoints et du single-step) distant. S'il s'agit d'une exception légitime pour la VM, il l'injecte.

Dans le cas d'Intel, qui ne dispose pas d'un paramétrage d'interception des interruptions logicielles, un handler spécifique d'exception devra être développé afin de traiter le cas particulier de l'émulation de l'interception d'interruptions logicielles.

8.4 Opérations d'entrées/sorties

L'interception des I/O s'effectue par le biais d'une bitmap (un bit par port). Lors d'un `vm-exit`, le cpu fournit des informations relatives à l'opération comme la direction, s'il s'agissait d'une opération de type *string*, sa taille, le port cible. À partir de ces informations, l'hyperviseur est en mesure d'émuler ou de proxyfier l'opération demandée par la VM.

Notons que l'émulation de certaines opérations (en particulier celles de type *string*) peut s'avérer très délicate et représente une cible de choix pour l'attaque d'hyperviseurs.

9 Émulation

9.1 Instructions

L'hyperviseur embarque pour le moment un désassembleur libre (udis86[9]) offrant une aide précieuse au sous-système d'émulation. Sous AMD, le besoin d'un

désassembleur est moins conséquent que sous Intel et nous aurions pu facilement nous en passer du fait de la simplicité d'encodage des instructions à émuler (`mov to/from cr, intn, clts`).

L'émulation d'instructions est assez sensible sous x86 car les mécanismes du mode protégé sont relativement complexes et induisent de nombreuses vérifications et implications, notamment en ce qui concerne des instructions *systèmes*.

Comme pour l'interception d'instructions exécutées nativement (`cpuid, rdmsr, wrmsr`), l'émulation se doit de prendre en compte le contexte complet d'apparition du `vm-exit`, en particulier en ce qui concerne le bit TF du registre rflags. Si ce dernier est positionné, l'hyperviseur se doit d'injecter, immédiatement après l'émulation, une Debug Exception. Il s'agit d'ailleurs d'un moyen très efficace de détecter des hyperviseurs, comme nous le précise [10].

Si la furtivité ne fait pas partie des spécifications de Ramooflax, il se doit néanmoins de conserver un comportement correct du système virtualisé.

9.2 Périphériques

L'hyperviseur propose l'émulation de quelques périphériques : UART, PIC, KBD et PS2 system controller.

L'émulation de l'UART permet notamment de récupérer assez simplement des logs kernel envoyés sur un port série et de les rediriger dans l'interface de debugging de l'hyperviseur. Un mode *passthrough* n'est pas suffisant de par le fait que le paramétrage de l'UART effectué par l'hyperviseur peut différer de celui imposé par la VM, notamment en ce qui concerne les bits de parités, la vitesse du lien, ... L'émulation nous permet dans ce cas un contrôle plus fin des opérations effectuées par la VM.

L'émulation du contrôleur clavier et du contrôleur système PS2 est nécessaire, du moins partiellement, afin d'empêcher la VM d'accéder à certains services historiques du monde du PC sans que l'hyperviseur ne s'en aperçoive. Ces deux périphériques permettent d'activer ou non la GateA20 qui a une influence sur la gestion de l'espace d'adressage de la VM en mode réel. Ils permettent également de redémarrer le système. Ramooflax affûté de la sensation !

10 Communication distante

L'hyperviseur laisse accès à la VM, aux périphériques présents, exceptés ceux dont il a besoin comme précisé dans le schéma 7.

L'hyperviseur distingue son interface de debug, via laquelle il sort des logs à caractère informatif, de son interface de contrôle permettant son pilotage distant.

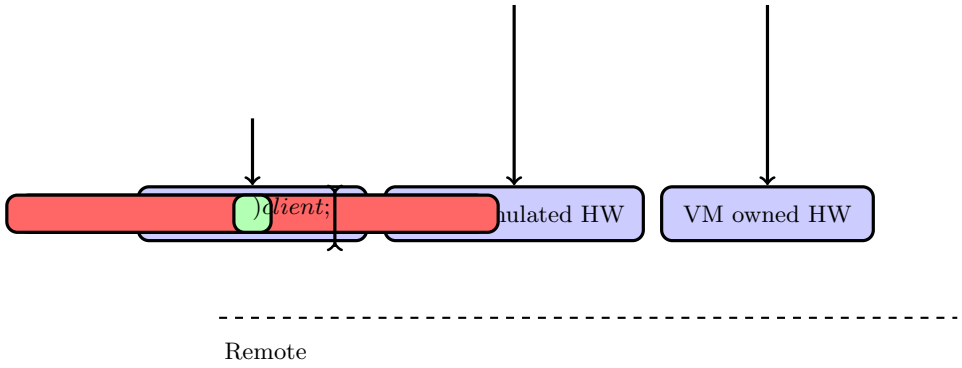

Figure 7. Interaction VMM/VM/Client lors de l'accès aux périphériques.

10.1 UART

Le port série peut être utilisé comme interface de debug et/ou de contrôle. Elle reste lente et il devient de plus en plus difficile de trouver des postes clients modernes disposant d'une interface RS232 cablée à l'arrière de la machine.

Nous proposons son implémentation uniquement comme solution de secours et de debugging.

10.2 EHCI Debug port

La spécification EHCI (USB 2.0) mentionne qu'un port USB physique peut-être utilisé comme EHCI Debug port si le contrôleur le permet. La plupart des contrôleurs EHCI que l'on retrouve dans des machines modernes, proposent la fonctionnalité de Debug port sur un des ports physiques attachés au contrôleur.

L'intérêt du Debug port est qu'il est standardisé, très simple à programmer comparé à de l'USB classique et offre des débits de l'ordre de 480Mbits/s. Ce qui en comparaison du port série est très profitable.

Côté hyperviseur L'hyperviseur implémente donc un driver de Debug port côté contrôleur EHCI. L'utilisation du Debug port impose la perte d'un port USB physique sur la machine virtualisée. L'hyperviseur garantit également que la VM ne cherchera pas à récupérer son port USB en le réinitialisant ou en détectant toute activité dessus.

Bien que la norme EHCI soit standardisée, l'implémentation du driver de Debug port côté hyperviseur ne nous a pas empêché de faire face à de petites subtilités selon les constructeurs.

Côté client Le problème majeur provient du standard USB qui ne permet pas à deux contrôleurs hôtes (tels qu'on en retrouve sur les PC) d'échanger directement des données entre eux.

Pour échanger de l'information avec un Debug port, il est nécessaire de disposer d'un debug device. On peut en trouver dans le commerce sous la forme d'un périphérique USB exposant une interface série USB côté client.

On peut également profiter des boards de développement embarqué ou des smartphones dernières générations proposant un contrôleur EHCI OTG [23] qui peut être configuré aussi bien en mode hôte que device. C'est ainsi de cette façon que l'on peut monter son smartphone comme une clef USB, car le contrôleur USB en mode device émule d'une certaine manière un périphérique USB que l'on peut brancher sur un contrôleur hôte.

Sous Linux, une API est mise à disposition des développeurs afin de permettre l'émulation de tout et n'importe quoi depuis un contrôleur device : la Gadget API.

Nous avons donc développé un Gadget USB simulant un debug device et exposant une interface série USB. Ce gadget est intégré dans la branche officielle du noyau Linux depuis sa version 2.6.36.

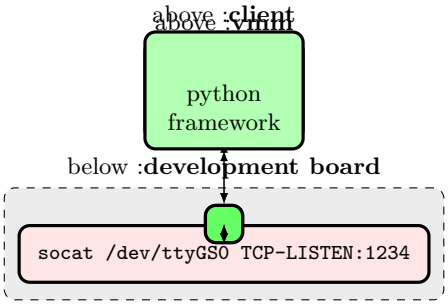

Figure 8. Contrôle de la VM via EHCI Debug port.

L'exemple d'architecture précédent, nous permet de nous connecter à l'hyperviseur via une connexion TCP sur le port 1234, relayée via `socat` dans un `tty` exposé par le Debug Device Gadget sur la board de développement. La board étant connectée en USB à la machine physique exécutant l'hyperviseur et sa VM.

23. USB On-The-Go

11 Interaction distante

L'hyperviseur comme nous l'avons déjà mentionné, passe principalement son temps à attendre des évènements provenant de la VM qu'ils soient légitimes ou liés à l'interaction avec un client distant.

11.1 Prise de contrôle

L'un des points critiques dans le contrôle distant est justement de pouvoir garantir que l'hyperviseur prendra la main sur la VM lorsque le client distant en aura besoin.

Sous Intel, un timer spécifique à la virtualisation a été ajouté récemment, le *vmx_preemption_timer*. Il permet de générer un `vm-exit` au bout d'un certains nombre de cycles cpu. Sous AMD, nous ne disposons pas d'une fonctionnalité équivalente.

Pour l'instant l'hyperviseur analyse à chaque `vm-exit` un ratio déterminant si oui ou non il doit consulter son interface de contrôle. L'idée est donc de générer le plus de `vm-exit` possible afin de garantir une prise de contrôle quasiment instantanée à distance. Par contre, plus il y a de `vm-exit` moins la VM est réactive.

L'hyperviseur se base sur une heuristique liée aux systèmes d'exploitations modernes, l'usage de cr3. En effet des systèmes d'exploitation comme Linux ou Windows ordonnent régulièrement des processus utilisateurs, ce qui implique un changement de cr3. Étant donné que l'hyperviseur intercepte les écritures dans cr3 pour des raisons expliquées précédemment, ceci lui permet de consulter à intervalle régulier son interface de contrôle attachée au client distant.

Notons que si dans le cas de l'utilisation du port série comme interface de contrôle, une interruption est générée lorsque des données sont reçues, dans le cas du Debug port les interruptions ne concernent que les communications USB classiques. De plus l'interception des interruptions matérielles étant globale (tout ou rien), nous aurions introduit une latence non acceptable dans le traitement d'interruptions n'ayant pas de rapport avec le contrôle distant de l'hyperviseur.

11.2 Stub GDB

Le sous-système de contrôle de l'hyperviseur est implémenté sous la forme d'un stub GDB serveur. Ceci permet en outre d'y connecter tout client gdb supportant le protocole gdb distant.

L'hyperviseur supporte les commandes de base du protocole GDB :
- lecture/écriture des registres généraux
- lecture/écriture mémoire
- ajout/suppression de breakpoints softwares et hardwares

– single stepping

Le protocole GDB s'applique parfaitement à des programmes du monde utilisateur. Mais lorsqu'il s'agit de débugguer un noyau de système d'exploitation ou de retrouver un processus lorsque l'on debuggue une VM, le protocole montre vite ses limites.

Le cas de la connexion à un VMware via gdb est suffisamment explicite. Dans 90% des cas on se retrouve dans du code noyau et effectuer des opérations portant uniquement sur un processus particulier devient très vite compliqué.

C'est pourquoi il a été nécessaire d'étendre le protocole afin de fournir certaines commodités.

11.3 Extensions spécifiques

Accès aux registres systèmes La première limitation concernait les registres systèmes. Avec le protocole GDB classique nous ne pouvions lire/modifier les registres de contrôle, de debug, la partie interne des registres de segments, ...

Nous proposons quelques commandes supplémentaires permettant d'accéder aux registres :

– cr0, cr2, cr3, cr4
– dr0-dr3, dr6 et dr7
– msr efer et dbgctl
– cs, ss, ds, es, fs, gs pour leur adresse de base
– gdtr, idtr, ldtr et tr

Accès mémoire La fonctionnalité de lecture/écriture mémoire de gdb n'a pas trop de sens dans le cas du debugging de VM. Du point de vue d'une application, le stub serveur n'accède qu'à l'espace virtuel du processus débuggué. Dans le cas d'un hyperviseur, lorsque le client distant prend le contrôle de la VM, les opérations de lecture/écriture en mémoire peuvent concerner aussi bien des adresses physiques (cas du mode réel) que des adresses virtuelles. Dans le cas d'adresses virtuelles, il est impératif de savoir quelles tables de pages utiliser pour effectuer la traduction en adresses physiques.

Notre extension permet de lire et écrire en mémoire physique et/ou virtuelle. Dans le cas d'adresses virtuelles, l'hyperviseur se sert du cr3 courant.

Nous fournissons néanmoins une fonctionnalité permettant de fixer une valeur de cr3 avec laquelle effectuer toutes les opérations liées à la mémoire. Ce qui s'avère très pratique pour installer des breakpoints logiciels dans l'espace virtuel d'un processus particulier.

Nous fournissons également un service de traduction d'adresses virtuelles en adresses physiques.

La section consacrée au client distant démontrera quelques cas d'usage de ces services.

Last Branch Record Parmi la foultitude de fonctionnalités proposées par l'architecture x86, celle des LBRs est vraiment très intéressante. Elle permet d'enregistrer dans des MSRs particuliers, la valeur d'eip avant et après le dernier saut effectué ainsi que lors de l'apparition d'exceptions, interruptions, ... Les MSRs suivants sont automatiquement remplis par le CPU :

- `from_eip`, valeur d'eip avant le saut
- `to_eip`, destination du saut
- `last_excp_from`, valeur d'eip avant l'apparition de l'exception
- `last_excp_to`, destination de l'exception

Nous proposons également l'activation ou non de cette fonctionnalité. Ramooflax innovant [24] !

Notons qu'AMD propose de virtualiser automatiquement ces MSRs. Ceci est très pratique lorsque l'on dispose d'un BIOS buggué ne restaurant pas l'activation des LBRs en retour de mode SMM. Auquel cas les dernières valeurs stockées dans les MSRs réels concernent des adresses du code de l'hyperviseur.

Contrôle de la virtualisation Nous proposons également quelques fonctionnalités permettant de contrôler des éléments des structures VMCS/VMCB. Elles sont incomplètes mais devraient permettre à terme de contrôler totalement les options de virtualisation.

À ce jour, il est possible de contrôler les bitmaps d'interceptions des exceptions et de l'accès aux registres de contrôle.

11.4 De l'art de single-stepper

Les différents scénarii d'usage du single-step dans le cas du debuggage d'une machine virtuelle peuvent être relativement complexes à gérer, notamment lors des transitions de niveau privilège. Nous pouvons les résumer comme suit :

- single-step global
- single-step d'un processus particulier, en ring3 uniquement
- single-step d'un processus particulier, en ring3 et en ring0 (appels systèmes, handlers d'interruptions)
- single-step d'un thread noyau particulier

24. Alexandre Gazet (Blackmore's spiritual son) propose également dans Metasm l'usage de cette fonctionnalité[13].

L'implémentation du single-step dans Ramooflax se base principalement sur la gestion du bit TF du registre rflags et l'interception de l'exception Debug (#DB). Ramooflax n'implémente pas la gestion de tous ces scenarii. Il permet pour le moment d'effectuer un single-step global ou ciblé uniquement sur un processus particulier et uniquement à son niveau de privilège (ring3).

Si le single-step d'un processus est assez simple à gérer de manière indépendante du système d'exploitation de la machine virtuelle, du fait qu'il est assez commode d'identifier un processus via son registre cr3 ou son pointeur de pile noyau (tss.esp0), il est par contre difficile d'identifier un thread noyau et d'effectuer du single-step uniquement dans son code. Matériellement parlant, seule la pile noyau du thread change lorsqu'il est ré-ordonnancé et cela sans transition de niveau de privilège. Cependant cette pile n'est représentée que par des registres (`esp/ebp`) dont l'interception à l'écriture n'est pas envisageable.

Notons également que Ramooflax ne peut garantir à tout instant un état cohérent du single-stepping. Si nous prenons le cas du single-step d'un processus utilisateur en ring3 uniquement, l'hyperviseur décide de ne pas single-stepper le code kernel lié à la gestion du processus. Ainsi lors d'une transition ring3-ring0, le single-step est désactivé par le cpu et l'hyperviseur ne s'y oppose pas. Il attendra simplement que le ré-ordonnancement du processus (et donc le rétablissement du registre rflags du processus) provoque #DB ou un `vm-exit` lié à l'interception de l'instruction `iret` afin de réactiver le single-step. Il se peut que le noyau termine le processus et ne le ré-ordonnance jamais. Il en va de même pour un thread noyau. L'hyperviseur pourrait ainsi se retrouver dans une situation où la fonctionnalité de single-step reste activée mais n'est plus exploitée.

Concernant sa furtivité mais également sa cohérence, l'hyperviseur se doit d'intercepter les instructions susceptibles de désactiver/détecter le bit TF, comme :

- `pushf` afin d'éviter à la VM d'empiler un registre rflags dont le bit TF est positionné
- `popf, iret` afin d'éviter à la VM de restaurer un registre rflags dont le bit TF serait désactivé et perdre la main
- `intN`, exceptions et interruptions matérielles, afin d'éviter au cpu d'effacer le bit TF du registre rflags courant et modifier la valeur de rflags automatiquement mise en pile

L'hyperviseur fait également usage du single-step en interne afin de rétablir les breakpoints logiciels. Ce mode de fonctionnement ne donne pas lieu à une interaction avec le client distant.

12 Client distant

Nous avons choisi de développer une API python permettant d'accéder à distance à l'hyperviseur. Nous détaillons quelques éléments de cette API ainsi que des exemples d'utilisation.

Il s'agit ici d'illustrer les fonctionnalités de l'hyperviseur et la simplicité d'accès à ses services. Des frameworks existants, comme Metasm[12], auraient très bien pu être utilisés afin de s'attacher à l'hyperviseur, moyennant un peu de développement.

12.1 Composants du framework python

L'API propose plusieurs classes assez simples d'utilisation :
- VM, fournissant les fonctionnalités de plus haut niveau
- CPU, permettant l'accès aux registres et aux filtres d'exceptions
- Breakpoints, au nom suffisament explicite
- GDB, implémentant un client GDB classique muni des extensions propres à l'hyperviseur
- Memory, contrôlant tous les accès à la mémoire
- Event, permettant au développeur de fournir son propre gestionnaire de `vm-exit`

12.2 VM

Au plus haut niveau, elle fournit les services suivants : run, interact, singlestep, resume, stop, attach, detach.

Nous avons pensé intéressant de fournir à la fois un mode interactif et un mode scriptable. Le mode interactif permet un peu (un tout petit peu) à la manière de scapy de conserver un contrôle de toute l'API dans un shell python, tandis que le mode scriptable permet d'effectuer des traitements de manière automatique.

La classe VM est instanciée comme suit :

```
vm = VM(CPUFamily.AMD, 32, "192.168.254.254:1234")
```

On retrouve le modèle de cpu ainsi que l'adresse et le port permettant d'accéder à la board de développement. L'implémentation via port série n'est pas disponible dans notre framework, qui reste encore une fois *illustratif*. Le mode interactif est appelé comme dans l'exemple suivant :

```
vm.run(dict(globals(), **locals()))
```

Tandis que le mode scriptable requiert quelques étapes supplémentaires :

```
vm.attach()  # connexion distante
vm.stop()    # arrêt de la vm

# xxxx (breakpoints, filtres, ...)

vm.resume()  # resume la vm et attend le prochain vm-exit
vm.detach()  # déconnexion, la vm reprend la main
```

L'utilisation reste relativement simple. Le mode interactif se quitte/récupère en utilisant `ctrl+d, ctrl+c`.

12.3 CPU, Memory et Breakpoints

Ces classes donnent accès aux registres généraux et systèmes, à la gestion des exceptions, aux breakpoints. Notons que la synchronisation des modifications apportées aux registres est gérée de manière transparente par l'API. Elles permettent également d'effectuer des accès en mémoire physique/virtuelle.

Ci-dessous un exemple de mise en place de breakpoints et de lecture mémoire :

```
# breakpoint en écriture
vm.cpu.breakpoints.add_data_w(vm.cpu.sr.tr+4, 4, filter, "esp0")

# lecture de la mémoire physique
xx = vm.mem.pread(0xa0000, 12)

# positionnement d'un cr3 particulier
# et lecture d'une page de mémoire virtuelle
vm.cpu.set_active_cr3(my_cr3)
pg = vm.mem.vread(0x8048000, 4096)
```

Le breakpoint concerne le champ `esp0` du TSS pointé par le registre TR. Nous verrons plus loin l'utilité de *filter*. Notons que nous pouvons donner des noms aux breakpoints (ici *esp0*). Il est également très simple de lister les breakpoints :

```
>>> vm.cpu.breakpoints
esp0 0xc1331f14 Write (4)
kernel_f1 0xc0001234 eXecute (1)
```

Tout comme les registres systèmes (mais également généraux) :

```
>>> vm.cpu.sr
cr0     = 0x000000008005003b
cr2     = 0x00000000b7681ed0
cr3     = 0x00000000371f9000
cr4     = 0x0000000000000690
dr0     = 0x0000000000000000
dr1     = 0x0000000000000000
dr2     = 0x0000000000000000
dr3     = 0x0000000000000000
dr6     = 0x00000000ffff0ff0
dr7     = 0x0000000000000400
dbgctl  = 0x0000000000000000
efer    = 0x0000000000001000
cs      = 0x0000000000000000
ss      = 0x0000000000000000
ds      = 0x0000000000000000
es      = 0x0000000000000000
fs      = 0x0000000000000000
gs      = 0x00000000c1367c00
gdtr    = 0x00000000c132e000
idtr    = 0x00000000c132d000
ldtr    = 0x0000000000000000
tr      = 0x00000000c1331f10
```

12.4 Event

Plutôt que d'implémenter la syntaxe non intuitive des breakpoints conditionnels et autres watchpoints du protocole gdb, nous avons opté pour un système de filtre ou *callback* associable à chaque `vm-exit`.

Ce système, en profitant du langage python, permet d'implémenter des breakpoints conditionnels mais également d'effectuer des opérations plus complexes, comme par exemple analyser la mémoire à chaque écriture du registre cr3.

Ces filtres permettent en outre de dissocier aisément les fonctionnalités uniquement liées à l'architecture matérielle (et fournies par le framework) de celles dépendantes de l'os tournant sur la VM (et à la charge de l'analyste).

Ainsi, la plupart des services des classes précédentes proposent de définir un filtre correspondant à une fonction développée en python.

La méthode `vm.resume()` rend la main à la VM et dès qu'un `vm-exit` survient, appelle automatiquement le filtre correspondant et renvoie sa valeur de retour.

Une exemple d'utilisation consisterait à faire renvoyer `True` à un filtre dès que l'analyste souhaite prendre la main sur la VM et passer par exemple en mode interactif.

Le code suivant illustre nos propos. Il entre en mode interactif lorsqu'un #PF est provoqué par l'instruction située à l'adresse `0x1234` :

```
def handle_excp(vm):
   if vm.cpu.gpr.eip == 0x1234:
     return True
   return False

vm.cpu.filter_exception(CPUException.page_fault, handle_excp)

while not vm.resume():
   continue

vm.interact(dict(globals(), **locals()))
```

L'annexe 14 illustre un cas d'utilisation du framework permettant de retrouver le *page directory* d'un processus sous Linux 2.6, via son nom.

13 Conclusion

Bien que facilité par la présence des extensions matérielles de virtualisation, le développement d'un hyperviseur reste suffisamment complexe et sensible. Si Ramooflax est loin d'être un produit fini, il dispose à ce jour de fonctionnalités et de commodités d'utilisation tout à fait acceptables pour tenter l'analyse de systèmes d'exploitation complexes en environnement natif. Néanmoins de nombreuses fonctionnalités restent à développer.

Concernant les limitations auxquelles Ramooflax doit faire face, notamment le mode SMM, l'apparition de BIOS libres pourrait offrir une alternative permettant de pousser encore plus loin l'analyse de systèmes *réels*. L'exploration des tables ACPI et de toutes les subtilités qu'elles cachent constituerait également un champ d'application idéal pour Ramooflax.

14 Annexe : `process_finder`

Le principe consiste à installer un filtre sur les écritures du registre cr3. À chaque écriture, l'hyperviseur donne la main au client distant qui appelle notre filtre. Le filtre inspecte la stack kernel associée au dernier processus ordonnancé (`tss.esp0`), récupère son `thread_info` puis la `task_struct` et la `mm_struct` afin d'atteindre son `pgd`. Si le champ `comm` est celui désiré, le filtre renvoit `True`.

Notons que nous préférons parcourir la liste complète des tâches dès la première écriture de cr3, car rien ne garantit que tous les processus seront ordonnancés à chaque exécution de notre filtre. Ceci provient principalement du fait que l'arrêt de la vm peut provoquer un ré-ordonnancement particulier de certaines tâches (selon la stratégie utilisée par l'ordonnanceur) et conduire à une situation de famine.

```
#!/usr/bin/env python

import sys
from vm import *

if len(sys.argv) < 2:
    print "need prog name"
    sys.exit(-1)

process_name = sys.argv[1]
process_cr3 = 0

# Some offsets for debian 2.6.32-5-486 kernel
com_off = 540
next_off = 240
mm_off  = 268
pgd_off = 36

def next_task(vm, task):
    next = vm.mem.read_dword(task+next_off)
    next -= next_off
    return next

def walk_process(vm, task):
    global process_cr3
    head = task
    while True:
        mm = vm.mem.read_dword(task+mm_off)
        if mm != 0:
            comm = task+com_off
            name = vm.mem.vread(comm, 15)
            pgd  = vm.mem.read_dword(mm+pgd_off)
            print "task",name
            if process_name in name:
                process_cr3 = pgd - 0xc0000000
                print "===> task cr3",hex(process_cr3)
                return True

        task = next_task(vm, task)
        if task == head:
            return False

def find_process(vm):
    esp0 = vm.mem.read_dword(vm.cpu.sr.tr+4)
    thread_info = esp0 & 0xffffe000
    task = vm.mem.read_dword(thread_info)
    if task == 0:
        return False
    return walk_process(vm, task)

# Main (architecture dependant only)
vm = VM(CPUFamily.AMD, 32, "192.168.254.254:1234")

vm.attach()
vm.stop()
vm.cpu.filter_write_cr(3, find_process)

while not vm.resume():
    continue

vm.cpu.release_write_cr(3)

print "success"
vm.detach()
```

Références

1. Intel virtualization roadmap
   http://software.intel.com/file/1024
   http://www.xen.org/files/xensummit_4/VT_roadmap_d_Nakajima.pdf
2. OpenGL tutors (N. Robins)
   http://www.xmission.com/~nate/tutors.html
3. bluepill (J. Rutkowska)
   http://theinvisiblethings.blogspot.com/2006/06/introducing-blue-pill.html
4. vitriol (DDZ matasano)
   http://www.theta44.org/software/HVM_Rootkits_ddz_bh-usa-06.pdf
5. virtdbg (D. Aumaitre, C. Devine)
   http://code.google.com/p/virtdbg/
6. hyperdbg (A. Fattori)
   http://code.google.com/p/hyperdbg/
7. abyss (Ivanlef0u)
   http://www.ivanlef0u.tuxfamily.org/?p=120
8. multiboot specification
   http://www.gnu.org/software/grub/manual/multiboot/multiboot.html
9. udis86, disassembler library for x86 and x86-64 (V. Thampi)
   http://udis86.sourceforge.net
10. Detecting simple hypervisors (N. Falliere)
    http://0x5a4d.blogspot.com/2009/11/detecting-simple-hypervisors.html
11. Revision Guide for AMD Family 10h Processors (AMD)
    http://support.amd.com/us/Processor_TechDocs/41322.pdf
12. The METASM assembly manipulation suite (Y. Guillot)
    http://metasm.cr0.org/
13. Metasm HowTo : bintrace (A. Gazet)
    http://esec-lab.sogeti.com/dotclear/index.php?post/2010/07/19/90-metasm-howto-bintrace