Sécurisation de systèmes reposant sur Xen

Benoit Poulot-Cazajous, Laurent Corbin, and Andrei Semenov

Bertin IT, Montigny-le-Bretonneux, France
{prenom.nom}@bertin.fr
https://www.bertin-it.com

Abstract. Bertin IT developpe des systèmes de sécurité reposant sur la virtualisation. Pour ces systèmes, la maitrise de l'hyperviseur est fondamentale, ce que les hyperviseurs open-source rendent possible. Ces derniers ont des caractéristiques variées en termes de fonctionnalité et leurs approches de la sécurité ne sont pas toujours à la hauteur des attentes. Nous présentons ici des travaux que nous avons mené autour et à l'intérieur de l'hyperviseur Xen afin d'en améliorer la sécurité, et montrons ainsi qu'il est possible d'atteindre de hauts niveaux de confiance à l'aide de techniques de virtualisation open-sources.

Keywords: Virtualisation · Sécurité · Xen.

1 Introduction

Bertin IT[1] développe des systèmes de sécurité reposant sur la virtualisation.

Le premier, CrossinG[2], est une passerelle de sécurisation des échanges inter-réseaux. Elle permet de faire passer des fichiers entre deux réseaux par ailleurs déconnectés, en utilisant des mécanismes de vérification d'innocuité et de filtrage. Cette passerelle est une appliance matérielle utilisant un serveur de type x86.

Le second, SHES[3] (pour Secure Hypervisor for Embedded Systems), vise à faire cohabiter sur une même plate-forme embarquée, de type ARMv8, plusieurs environnements riches (de type Linux ou Android) avec un environnement temps réel (développé par KronoSafe[4]), en respectant de fortes contraintes en termes de sécurité et de sûreté de fonctionnement.

Dans les deux cas, nous utilisons les capacités du matériel pour assurer la séparation des environnements d'exécution : les Security Extensions (TrustZone) et Virtualisation Extensions (mode HYP) sur ARMv8, et VT-x/VT-d sur x86.

Ces deux systèmes reposent sur le même hyperviseur "bare metal" open-source : Xen[5]. Dans cette communication, nous expliquerons pourquoi la virtualisation nous semble indispensable dans tout système un peu riche qui vise un haut niveau de sécurité, pourquoi nous avons choisi l'hyperviseur Xen, puis nous présenterons quelques directions que nous avons prises pour renforcer la sécurité de cet hyperviseur, avant de conclure par une analyse a posteriori de l'expérience acquise.

2 Avantages de la virtualisation

Dans les systèmes que construit Bertin IT, la virtualisation offre trois classes d'avantages.

2.1 Ajout de fonctionnalité

La virtualisation permet d'offrir aux systèmes d'exploitation fonctionants dans les VM des fonctionnalités qu'ils n'ont pas nécessairement, ou avec un meilleur niveau de confiance. Sans nous étendre sur le sujet, on peut citer :

- Backup, mise à jour
- Chiffrement disque, effacement d'urgence
- IPsec, 802.1q en coupure
- Introspection, détection de malwares[6]
- etc...

[1] `https://www.bertin-it.com/`
[2] `https://www.bertin-it.com/cybersecurite/crossing-passerelle-securisation-interconnexion-reseaux-ser`
[3] SHES est en partie financé par DGA/MI.
[4] `http://www.krono-safe.com/`
[5] `https://xenproject.org/help/documentation/`
[6] `https://hal.inria.fr/hal-01762803v2`

2.2 Réduction de la surface d'attaque

La virtualisation permet de construire des systèmes dans lesquels chacune des VM aura une fonction précise. Par exemple fournir un service de stockage de fichier, piloter un contrôleur Ethernet, réaliser des analyses de fichiers, etc...

Cela permet d'utiliser des configurations spécialisées et minimalistes des systèmes d'exploitation hébergés par chacune des VM. Par exemple, une VM de pilotage d'un contrôleur Ethernet n'aura a priori pas besoin d'avoir accès à d'autres périphériques, ni de disposer d'une pile TCP/IP, d'un système de fichiers persistant, d'un interpréteur de commandes, etc... Cela permet de supprimer beaucoup de code, utilisateur et noyau, et ainsi de diminuer la surface de code en contact avec un utilisateur malicieux.

Bien entendu, utiliser dans les VM des briques durcies, telles que Clip OS[7] ou des unikernels[8], améliore encore les choses.

2.3 Sécurisation de l'architecture

Un autre intérêt de la virtualisation est de permettre de réduire très fortement la TCB logicielle, c'est à dire l'ensemble des logiciels en qui on doit avoir confiance. Les systèmes d'exploitation modernes offrent des fonctionnalités riches, mais ils sont trop complexes pour que l'on puisse avoir une confiance vraiment élevée en leur sécurité. En revanche, un hyperviseur est beaucoup plus simple, offre donc une surface d'attaque moindre, est plus facilement auditable et est donc une meilleure base sur laquelle construire des architectures complexes mais solides. C'est le cas même à partir de briques que l'on ne pourrait pas juger de confiance, comme des OS non maîtrisés.

Ainsi, un hyperviseur doit pouvoir permettre d'avoir la maîtrise "totale" des interactions/communications des VM, entre elles et avec le monde extérieur. Dans l'hypothèse d'une attaque réussie, cela permet de bloquer sa propagation au reste du système. Plus généralement, cela permet d'énoncer des propriétés de sécurité sur le système complet.

3 Présentation de Xen

Xen est un hyperviseur open-source qui existe depuis 2003, est maintenu par une communauté active et importante, pour laquelle la sécurité est importante. Xen fonctionne sur de nombreuses plates-formes.

En termes d'architecture, Xen se place au-dessus de la concurrence par la possibilité de laisser contrôler certains périphériques par des VM[9] (grâce au support des technologies d'accélération matérielle de la virtualisation disponibles sur les processeurs récents), mais aussi, et c'est un point fondamental, par le modèle des split-drivers, qui permet à d'autres VM d'interagir de manière sûre

[7] `https://www.ssi.gouv.fr/administration/services-securises/clip/`
[8] `https://wiki.xenproject.org/wiki/Unikernels`
[9] VM: Machines virtuelles

(grâce au mécanisme des grant-tables) avec ces VM pilotes. Cette combinaison est unique à Xen, au moins pour ce qui est des hyperviseurs open-source.

Xen prend la sécurité très au sérieux. Ainsi, Xen tente de limiter l'impact qu'aurait une VM malveillante/hostile sur le reste du système. Ainsi, toute faille qui permettraient à une VM d'accéder de manière non légitime à des données d'autres VM, ou de l'hyperviseur, reçoit toute l'attention des développeurs et est corrigée en urgence. Cela a été démontré lors de la publication de failles Spectre et Meltdown, où des solutions ont été proposées dès la publications des failles.

En revanche, la communauté Xen, un peu comme la communauté Linux dont elle est proche, attache moins d'importance aux notions de défense en profondeur et d'auditabilité. Cela signifie en particulier que le code source de Xen ne s'encombre pas de commentaires inutiles[10], et que, toutes choses étant égales par ailleurs, la performance prime sur la lisibilité.

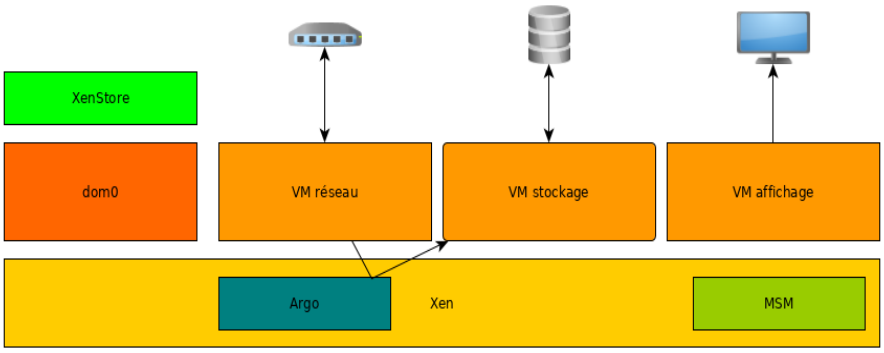
Xen étant de plus en plus utilisé dans des projets, en particulier dans l'embarqué et l'automobile, où il ne suffit pas d'annoncer la sécurité mais où il faut aussi la démontrer, l'importance de l'auditabilité est prise avec de plus en plus de sérieux.

3.1 Fonctionnement schématique

Voici, en quelques lignes, ce qu'il est important de savoir sur le fonctionnement de Xen afin de bien comprendre la suite de ce document.

Xen est un hyperviseur de type 1, "bare-metal", c'est à dire qu'il s'exécute directement sur la plate-forme matérielle et en prend le contrôle.

Au-dessus de cet hyperviseur tournent des machines virtuelles (VM) qui, selon la configuration, peuvent avoir accès à différentes ressources matérielles (CPU, RAM, périphériques).



Le diagramme ci-dessus montre une configuration typique d'un système Xen, avec plusieurs VM connectées directement à des périphériques. Les boites en vert ou bleu sont celles sur lesquelles portent les travaux abordés dans ce document.

Les différentes VM interagissent avec l'hyperviseur par le biais d'hypercalls, quand la VM sait qu'elle fonctionne au dessus d'un hyperviseur, ou d'exceptions,

[10] Les informations pertinentes se trouvent plutôt dans l'historique 'git' des fichiers.

quand la VM tente d'accéder à des ressources auxquelles l'hyperviseur a restreint l'accès. Dans ce dernier cas, l'hyperviseur va déterminer l'intention de la VM et réaliser l'action correspondante. On parle de para-virtualisation dans le premier cas, et d'émulation dans le second.

Un troisième cas se présente quand le matériel peut être configuré, par l'hyperviseur, pour fonctionner correctement dans une machine virtuelle, pratiquement sans intervention de l'hyperviseur en cours de fonctionnement. C'est le cas lorsqu'on utilise les technologies d'accélération matérielle de la virtualisation disponibles sur les processeurs récents.

Une même VM peut utiliser ces trois approches de la virtualisation sur des ressources de types différents. La combinaison la plus efficace consiste à utiliser l'accélération matérielle de la virtualisation pour les ressources de type CPU et RAM, ou plus généralement dans les cas d'accès direct au matériel (par exemple pour les VM pilotant les périphériques physiques), et la para-virtualisation pour les pilotes de périphériques virtuels.

dom0 Une VM particulière, appelé dom0, a des privilèges supérieurs aux autres VM. Cette VM a typiquement pour responsabilité de :

- construire, démarrer et arrêter les autres VM,
- héberger les services de Xen, tel que la Xenstore,
- piloter les périphériques.

Consciente qu'un dom0 volumineux fragilise la sécurité des systèmes Xen, la communauté Xen travaille à la "désagrégation du dom0", afin de permettre de repartir ses fonctionnalités sur d'autres VM plus petites et moins privilégiées, et de redescendre dans l'hyperviseur ce qui doit l'être.

Mécanismes de base

Grant tables Les grants-tables permettent à une VM de mettre à disposition des pages mémoires, identifiées par des références, pour que d'autres VM puissent les utiliser, de préférence pour des échanges des données.

Ainsi, par définition, les pages d'une VM utilisées par une autre VM sont uniquement les pages dont la première VM aura préalablement autorisé le partage[11].

Event channels Les event-channels sont le mécanisme de notification d'évènements de Xen. C'est par ce biais que les VM sont prévenues que 'quelque chose' s'est produit et réclame leur attention.

[11] Cette règle a une exception : le dom0 peut utiliser les pages d'une VM lors de la création de celle-ci, avant son démarrage.

Xen Security Modules XSM (Xen Security Modules) est un mécanisme permettant de configurer finement ce que peuvent faire les VM en termes de communications et d'accès aux ressources, matérielles ou logicielles.

On peut considérer que la configuration de XSM est à Xen ce que la configuration de SELinux est à Linux : à la fois très précise et de bas niveau. Elle est par conséquent délicate à mettre en oeuvre, et il est difficile d'avoir une confiance absolue dans ce mécanisme et dans sa configuration.

Mécanismes de plus haut niveau

Xenstore La Xenstore est une base de données centralisée que les VM utilisent principalement pour s'échanger des informations de configuration pour communiquer, tels que des numéros d'event-channels ou des références de pages dans les grant-tables.

Les échanges entre VM et la Xenstore utilisent aussi les grant-tables et les event-channels.

Nous aurons l'occasion de reparler de la Xenstore plus tard.

Argo Argo est un mécanisme de communication inter-VM. Il ne repose pas sur les grant-tables. Nous rentrerons dans les détails d'Argo dans ce qui suit.

Split-drivers La plupart des pilotes, par exemple disque et réseau, d'un système Xen utilisent le modèle des split-drivers, où un pilote peut être vu comme découpé en deux parties fonctionnant chacune dans une VM différente :

- un front-end, fonctionnant dans le noyau de la VM cliente qui a besoin d'accéder à un périphérique virtualisé.
- un back-end, fonctionnant dans une VM serveur ayant le plus souvent accès au périphérique physique, contrôlant et réalisant les d'entrée/sorties pour le compte du front-end.

Dans la plupart des cas, un même back-end peut être connecté à plusieurs front-end. Dans ce cas, le back-end réalise un multiplexage des accès. Les front-ends/back-ends peuvent s'enchaîner : une VM peut par exemple proposer un back-end exportant un disque en clair, en utilisant un front-end connecté à un back-end exportant un disque chiffré et en réalisant les opérations intermédiaires de chiffrement et déchiffrement.

Les front-ends et back-ends communiquent typiquement en utilisant de la mémoire partagée obtenue grâce au mécanisme des grant-tables, et se servent des event-channels pour se synchroniser. Le plus souvent, le back-end publie dans la Xenstore les informations pertinentes pour que les front-end puissent les découvrir.

Les back-ends sont le plus souvent dans le dom0, mais cela n'est pas obligatoire. Sur les plates-formes supportant les extensions de virtualisation, par exemple ARMv8 et x86, il est en théorie possible de dédier une VM à un périphérique

donné[12]. Le mécanisme des grant-tables permet d'assurer que ces "VM pilotes" n'ont pas besoin d'avoir accès à l'ensemble de la mémoire de leurs VM clientes, mais uniquement aux pages que ces dernières veulent partager. C'est un avantage en termes de sécurité sur d'autres hyperviseurs, tels que kvm par exemple.

4 Objectifs de sécurité visés

Nous nous sommes donnés deux objectifs de sécurité : le contrôle des communications inter-VM et la vérification du confinement mémoire.

4.1 Objectif 1 : contrôle des canaux inter-VM

Le premier objectif à trait aux communications inter-VM : nous pouvons vouloir exprimer quelle VM peut communiquer avec quelle VM. Nous voulons également vouloir contrôler dans quel sens circule l'information. L'idée est double : premièrement rendre plus difficile, voire impossible, la propagation d'une attaque réussie à l'intérieur du système. Et dans un second temps, empêcher les communications illégitimes entre deux VM qui auraient malgré tout été corrompues.

Ce second sous-objectif est d'une importance cruciale. Nous partons du principe que si deux VM "hostiles" sont connectées par un canal de communication légitime, elles pourront s'en servir pour faire fuiter de l'information sans que l'on puisse le détecter. On veut donc pouvoir assurer l'absence de canal, plutôt que d'espérer en contrôler l'utilisation. C'est à cette condition que l'on peut créer des architectures garantissant la sécurité de données, même en l'absence de garanties sur les utilisateurs de ces données.

4.2 Objectif 2 : confinement mémoire

Le second objectif a trait au confinement mémoire. Xen fait en sorte de bien gérer quelles sont les pages mémoires affectées à chaque VM et d'empêcher qu'une VM puisse accéder indûment à la mémoire d'une autre VM, ou de l'hyperviseur. La gestion des pages mémoire est un sujet compliqué, et toute faille dans ce code aurait des conséquences catastrophiques sur la sécurité du système. L'objectif ici est d'augmenter la résistance en cas de faille dans Xen ou dans sa configuration.

4.3 Hypothèse

Pour tenter d'atteindre ces deux objectifs, nous allons faire l'hypothèse que les VM, à l'exception du dom0 et de Xenstore, sont hostiles, que l'attaquant connaît parfaitement Xen et les contre-mesures mises en place.

[12] En pratique, il arrive parfois que les circuits d'IOMMU (sur x86) ou de SMMU (sur ARM) ne permettent pas d'avoir des contextes séparés pour certains périphériques. Il est alors nécessaire de regrouper les pilotes au sein d'une seule VM.

5 Base de données centralisée (Xenstore)

Xenstore est une base de donnée centralisée accessible par toutes les VM. Elle est utilisée pour stocker des informations de configuration et d'état. En particulier, dans le cas d'un split-driver, le back-end va y stocker le moyen de le contacter et des informations sur le périphérique "exporté", ce qui permettra aux front-ends intéressés (et autorisés) de s'y connecter. Xenstore peut être hébergé par le dom0 ou une VM dédiée. Bien entendu, pour réduire sa surface d'attaque, il convient de faire fonctionner Xenstore dans sa propre VM.

Le contrôle d'accès est de type DAC[13], c'est le propriétaire d'une information qui détermine qui peut y accéder. Il est assez facile pour une VM, éventuellement hostile, d'y créer un noeud et d'autoriser une autre VM, également hostile, à y accéder, ce qui permet de créer un canal de communication.

Dans l'exemple suivant, on a deux VM, vm2 et vm3, qui vont communiquer via la Xenstore.

```
vm3# xenstore-chmod /local/domain/3/attr b3
vm3# xenstore-write /local/domain/3/attr/coucou "unsecret"

vm2# xenstore-read /local/domain/3/attr/coucou
unsecret
vm2# xenstore-write /local/domain/3/attr/coucou "coucou"

vm3# xenstore-read /local/domain/3/attr/coucou
coucou
```

D'abord *vm*3 va donner à toutes les VM les droits en lecture et écriture à un répertoire qu'elle possède dans la Xenstore[14]. Elle peut le faire, le contrôle d'accès étant de type DAC. Puis, les deux VM peuvent interagir très simplement comme montré dans l'exemple.

Ce comportement est contraire au premier objectif que nous nous sommes fixés : celui d'empêcher la communication de VM qui n'ont pas à communiquer.

Pour le résoudre, nous nous reposons sur le fait que deux VM ayant un besoin légitime de communication le feront par le biais d'un mécanisme utilisant les event-channels. Les échanges par event-channels sont contrôlables par le biais de la politique de sécurité XSM. Nous avons donc simplement modifié le code de Xenstore pour qu'il interroge la politique de sécurité XSM, au sujet des event-channels, quand un domaine donné veut accéder à une information possédée par un autre domaine.

Cette solution, dont l'élégance repose surtout sur la simplicité, ajoute une petite couche de MAC[15] au-dessus du DAC : c'est la configuration de l'hyperviseur et non les VM elles-mêmes, qui décide de qui communique avec qui.

[13] DAC: Discretionary Access Control
[14] La documentation de xenstore-chmod explique pourquoi "b3" fait cela.
[15] MAC: Mandatory Access Control

6 Communications unidirectionnelles et fiables (Argo)

La solution la plus simple lorsque l'on souhaite faire communiquer de manière fiable deux VM est d'utiliser un réseau Ethernet virtuel. Les performances sont médiocres, à cause des deux piles TCP/IP traversées, mais souvent acceptables.

Lorsque l'on souhaite une communication unidirectionnelle, l'utilisation d'un réseau Ethernet virtuel n'est pas possible avec Xen, car le protocole entre les front-ends et back-ends réseau utilise des pages qui sont, au moins de manière transitoire, mappées en écriture dans les deux VM communiquant, ce qui permet très facilement de faire circuler de l'information dans les deux sens. D'autres solutions, comme les vchans d'InvisibleThings, proposent des communications unidirectionnelles, mais elles aussi imposent des partages de pages qui invalident la promesse d'unidirectionnalité face à des utilisations hostiles.

Argo[16] est un module apparu dans la version 4.12 de Xen. Il est censé permettre des communications unidirectionnelles. Il n'y pas de mémoire partagée et les copies sont faites par l'hyperviseur, et non par les VM elles-mêmes. De plus, le flux de données n'ont plus à traverser les piles TCP/IP, ce qui promet des gains de performances substantiels. Nous avons donc décidé d'utiliser Argo pour les communications unidirectionnelles entre VM.

Plusieurs problèmes subsistent :

- l'API originelle d'Argo permet l'établissement de canaux cachés descendants (du récepteur vers l'émetteur), de type "storage channels",
- l'exigence de fiabilité introduit un canal caché descendant, de type "timing channel".

6.1 Storage channels

Argo permet d'établir des canaux 1to1 ou Nto1 entre VM. Nous avons supprimé les canaux Nto1, permettant à plusieurs VM d'envoyer des données à une VM destination, car la même fonctionnalité est réalisable avec N canaux 1to1 et il est plus facile de raisonner avec des canaux 1to1.

L'API d'Argo se compose des appels suivants :

- XEN_ARGO_OP_register_ring
- XEN_ARGO_OP_unregister_ring
- XEN_ARGO_OP_sendv
- XEN_ARGO_OP_notify

XEN_ARGO_OP_register_ring et XEN_ARGO_OP_unregister_ring permettent d'allouer/libérer les ressources permettant à deux VM de communiquer via Argo. Une VM pourrait faire des boucles d'allocations/libérations pour émettre un signal, nous avons donc supprimé XEN_ARGO_OP_unregister_ring de l'API et effectuons les allocations directement à l'initialisation des VM.

[16] `https://wiki.xenproject.org/wiki/Argo:_Hypervisor-Mediated_Exchange_(HMX)_for_Xen`

XEN_ARGO_OP_notify permet à une VM d'obtenir, ou d'envoyer, des informations sur les canaux Argo du système. Une VM pourrait s'en servir pour émettre de l'information dans le mauvais sens en jouant sur les caractéristiques de son canal de réception. Nous avons supprimé cet appel.

XEN_ARGO_OP_sendv permet d'envoyer des données. Si le récepteur n'est pas en capacité de recevoir les données, par exemple parce que son buffer est plein, le sendv échoue. L'émetteur est prévenu quand le récepteur redevient disponible. Nous avons rendu cet appel bloquant : il réussit toujours mais peut prendre un temps éventuellement long, le temps que les données arrivent à bon port. La seule information que reçoit l'émetteur est donc la durée qu'a pris l'envoi.

Il n'y a pas d'appel pour recevoir les données : leur disponibilité dans la mémoire de la VM réceptrice lui est signalée par le biais d'un event-channel. En revanche, le récepteur peut modifier son pointeur de réception [17] pour réduire arbitrairement la place disponible dans son canal. Nous avons ajouté des contrôles dans Argo pour s'assurer que le récepteur n'essaye pas "d'oublier" d'avoir lu des données dans le but de bloquer le canal de réception.

6.2 Timing channel

Le caractère unidirectionnel combiné au besoin de fiabilité fait inévitablement apparaître le problème classique des canaux cachés temporels liés aux acquittements "descendants" : la VM "haute" maintient le canal de communication dans un état tel que l'envoi de messages par la VM "basse" peut être bloquant ou non, ce qui permet à cette dernière d'obtenir de l'information sur l'état de la VM "haute". Si l'on n'y prend garde, un tel canal peut avoir une bande passante considérable. Grâce à une modélisation mathématique de la capacité du canal caché "descendant", nous définissons une stratégie de comportement d'Argo vis-à-vis de la VM "basse", permettant de limiter le bande passante "descendante" au-dessous d'une limite fixée, tout en garantissant des transferts fiables et rapides.

Ainsi, il est difficile en pratique pour un canal de communication d'être à la fois unidirectionnel et fiable. Dans le cas des diodes physiques, le caractère unidirectionnel est supposé absolu par l'absence du disposition physique permettant ce canal retour. En cas de perte de message, il n'y a aucun moyen de prévenir l'émetteur. On compense cela par l'ajout de "redondance" aux messages montants, afin de pouvoir reconstituer les éventuels messages corrompus ou perdus. Tout cela à un impact sur les performances et sur la complexité des architectures mettant en oeuvre des diodes physiques.

[17] Argo utilise des "rings", qui sont des zones de mémoire avec un pointer de transmission, modifié par Argo quand de nouvelles données arrivent, et un pointer de réception, modifié par la VM quand elle a lu des données. Normalement, ces deux pointeurs "avancent" et le pointeur de transmission devance celui de réception, car on ne peut pas lire ce qui n'est pas encore arrivé.

Dans les produits que développe Bertin IT, les performances sont fondamentales. C'est pour cela que nous avons préféré nous orienter vers l'utilisation de diodes logicielles et en améliorer les caractéristiques en termes de sécurité.

Ce qui suit décrit le chemin que nous avons commencé à suivre. A l"heure où nous écrivons ces lignes, il est encore trop tôt pour afficher des mesures crédibles de débits ou de performance.

Moyen d'action Comme dit précédemment, nous avons rendu bloquant l'appel d'envoi de données. Cela signifie qu'Argo peut temporiser autant que nécessaire sa réponse à la VM émettrice.

Il s'agit maintenant de préciser les détails de ce calcul d'attente.

Source de l'information descendante Reprenons notre canal retour descendant, du récepteur R vers l'émetteur E, et supposons que ces derniers sont tous les deux malicieux et ont envie de collaborer pour s'échanger de l'information, via ce canal retour. Il y a deux moyens pour R de générer de l'information : en provoquant le blocage du canal légitime montant, et en provoquant son déblocage. Il est important de noter que l'information générée n'est pas l'état du canal mais le moment où le changement d'état se produit.

On peut donc voir le canal de retour comme produisant une liste de nombres croissants, correspondants aux instants de blocages/déblocages :

$$t_{i+1} = t_i + dt_i, dt_i \in \mathbb{R}, dt_i > 0 \tag{1}$$

Nous allons voir dans ce qui suit comment modifier cette liste afin de diminuer la quantité d'information véhiculée.

Dans le cas qui nous intéresse, les communications de E vers R passent par le sous-système Argo, lequel réside dans l'hyperviseur. Il est alors possible de ralentir les hypercalls liés à Argo faits par E afin que celui-ci perçoive une liste modifiée de nombres ayant de meilleures caractéristiques.

Quantification du temps D'abord, nous allons quantifier le temps. En effet, si l'horloge mesurant quand le canal change d'état a une précision infinie, et tout les instants mesurables sont "possibles", alors R pourra envoyer une quantité arbitraire d'information en provoquant un blocage/déblocage exactement au bon moment. Dit autrement, les dt_i peuvent théoriquement[18] chacun potentiellement transporter une quantité infinie d'information.

[18] En pratique, avec la précision des horloges actuelles (de l'ordre de quelques Giga-Hertz), cela fait de l'ordre d'une trentaine de bits d'information par évènement.

Nous allons donc limiter l'ensemble des valeurs possibles visibles à des puissances entières d'un quantum de temps. Plus formellement, on va donc transformer la suite t_i en u_i, avec les propriétés suivantes :

$$\begin{aligned} u_0 &= t_0 \\ u_{i+1} &= u_i + qt * q^{du_i}, du_i \in \mathbb{N}, q > 1 \\ u_i + qt * q^{du_i - 1} &< t_{i+1} <= u_{i+1} \end{aligned} \tag{2}$$

q et qt sont des paramètres à choisir. Le choix de qt n'a pas vraiment d'importante, nous l'avons fixé arbitrairement à 1ms. Le choix de q est plus critique, plus sa valeur est proche de 1, plus les valeurs des u_i seront proches des t_i originels. En l'absence de recul suffisant, nous l'avons fixé arbitrairement à 1.1.

Intuitivement, on peut considérer que cette opération revient à ne garder qu'un nombre fixe de bits significatifs dans l'écriture des instants de blocage/déblocage.

Pour déterminer combien d'information contiennent les du_i, un détour rapide par la théorie de l'information de Shannon s'impose.

Application de l'entropie de Shannon La théorie de l'information de Shannon, dit que la capacité, ou entropie, d'une source d'information produisant des symboles S_i avec la probabilité p_i est :

$$H = -\sum_{i=1}^{N} p_i \log_2(p_i) \tag{3}$$

Cette équation se comprend bien en réalisant qu'un symbole ayant une probabilité p porte $-log_2(p)$ bits d'information. Par exemple, un évènement qui a une chance sur 2 de se produire porte 1 bit d'information : $-log_2(1/2) = -(-1) = 1$. Autre exemple, un octet, dans un fichier compressé/aléatoire, a une probabilité de 1/256, ce qui correspond à une information de $-log_2(1/256) = 8$ bits, ce qui est assez rassurant. En revanche, un caractère qui se répète avec certitude n'apporte aucune information, car $-log_2(1) = 0$.

Si l'on considère l'ensemble des symboles possibles, on obtient l'équation 3, qui nous donne un débit maximal en bits par symbole envoyé.

Dans le cas qui nous intéresse, les symboles sont les du_i, les probabilités p_i étant calculées à l'exécution. On remarque alors que la vitesse moyenne d'émission d'un symbole est :

$$V = \sum_{i=1}^{N} p_i * qt * q^{du_i} \tag{4}$$

Si l'on souhaite limiter la bande passante du canal retour à une valeur L, exprimée en bits/secondes, alors il faut que :

$$H <= V * L \tag{5}$$

Ce qui signifie :

$$-\sum_{i=1}^{N} p_i \log_2(p_i) <= L * \sum_{i=1}^{N} p_i * qt * q^{du_i} \quad (6)$$

En pratique Les envois de message par Argo sont devenus bloquants : si l'état du canal change (passe de bloqué à débloqué, ou l'inverse), alors l'envoi sera retenté jusqu'à ce qu'on ait attendu suffisamment pour qu'on puisse décider d'envoyer un du_i permettant à l'équation 6 d'être vérifiée.

Les statistiques utilisées dans ces calculs sont fait sur des fenêtres "glissantes" de 60 secondes, pour éviter qu'une longue sous-utilisation de la bande passante du canal de retour ne puisse donner lieu à une sur-utilisation.

7 Module de Sécurisation Mémoire (MSM)

Les configurations de Xen que nous utilisons reposent sur les extensions matérielles pour la virtualisation proposées par les CPU récents. En particulier le mécanisme de double translation mémoire (ou 'Second Level Address Translation' (SLAT)) permet à l'hyperviseur d'implémenter le confinement entre VM tout en offrant la possibilité à des VM de piloter directement des périphériques matériels. Ce mécanisme est en coupure des accès mémoire des VM et des périphériques qu'elles gèrent : si les arbres SLAT sont bien configurés et bien gérés, le confinement mémoire est garanti par le matériel.

Il n'est pas exagéré de dire qu'avoir confiance en Xen impose d'avoir confiance dans la bonne gestion des arbres SLAT. C'est pour cela que nous avons décidé d'utiliser les méthodes formelles afin d'obtenir ce niveau de confiance.

L'approche est la suivante : un nouveau module de Xen, nommé le MSM, intercepte toutes les modifications faites aux arbres SLAT. A l'intérieur du MSM se trouve une description formelle de ce qu'est un état sûr des arbres SLAT, en termes de confinement mémoire. A l'aide de l'outil Frama-C et de son plugin WP, on prouve que le MSM garantit que les arbres SLAT restent dans un état sûr.

7.1 Second Level Address Translation

En général, un système d'exploitation voit et peut utiliser toute la mémoire "physique" de la machine sur laquelle il s'exécute. Afin de pouvoir permettre à plusieurs processus/programmes de fonctionner, un système d'exploitation va utiliser la notion de mémoire "virtuelle", qui permet de limiter chaque programme à son espace mémoire privé. Une table de translation, gérée par le système d'exploitation pour chaque processus et utilisée par le processeur, va traduire les adresses "virtuelles" en adresses "physiques". Cette translation permet de cacher à un programme quelles adresses "physiques" il utilise et de l'empêcher d'accéder aux adresses "physiques" utilisées par les autres programmes et le système d'exploitation.

Pour des raisons d'efficacité, la mémoire est organisée en pages, en général de 4096 octets, et les translations s'entendent en termes de pages.

Lorsqu'un hyperviseur fait fonctionner plusieurs systèmes d'exploitation dans des machines virtuelles, il lui appartient de partitionner la mémoire entre ces systèmes et de contrôler les accès à celle-ci. Les processeurs récents permettent de configurer le processeur pour qu'après la translation d'une adresses virtuelle en adresse physique se produise une translation de cette adresse physique en adresse "machine", l'adresse "machine" étant l'adresse réelle de la mémoire sur l'ordinateur. Dans ce cas, la deuxième phase de translation, de "physique" en "machine", est gérée par l'hyperviseur. Cette translation permet de cacher à chaque machine virtuelle quelles pages "machines" elle utilise, tout en lui permettant de gérer elle-même les translations d'adresses virtuelles vers physiques, comme dans le cas sans virtualisation.

Lors des communications entre un pilote de périphérique et son périphérique, ce dernier utilise des adresses "bus". Quand la virtualisation n'est pas utilisée, les adresses "bus" vues par un périphérique correspondent aux adresses "physiques" utilisées par le pilote dans le système d'exploitation. Il est important que ces adresses correspondent pour que le pilote et le périphérique désignent la mémoire de la même façon.

Quand le pilote de périphérique fonctionne dans une machine virtuelle, il faut toujours faire correspondre les adresses "bus" et "physiques", mais il faut également que les adresses "bus" soient converties en adresses "machines" (réelles) de la même façon que les adresses "physiques" sont converties en adresses "machines". Il y a donc pour les périphériques un système similaire à celui qu'utilise le CPU pour transformer les adresses physiques en adresses machines. On parle d'IOMMU ou de SMMU, selon les architectures.

Assez naturellement, les systèmes d'Intel, d'AMD et d'ARM permettent d'utiliser les mêmes structures de données pour représenter les translations "physiques" vers "machines" pour les CPU d'une part, et "bus" vers "machines" pour les périphériques d'autres part.

Dans la suite, nous parlerons de configuration **p2m**[19] pour désigner l'ensemble des arbres SLAT des VM et de leurs périphériques.

7.2 Propriétés attendues

Par confinement mémoire, nous entendons la propriété selon laquelle une VM ne peut en général pas accéder à une page mémoire d'une autre VM, ou de l'hyperviseur, que ce soit directement, ou par le biais des périphériques qu'elle contrôle. En revanche, elle peut le faire si et seulement si la seconde VM l'a autorisé par le biais du mécanisme des grant-tables.

En termes d'arbre p2m, cela se traduit schématiquement par :

- les arbres p2m contiennent des noeuds (des tables de pages) et des feuilles (des pages *normales*),

[19] p2m: Physical to Machine

- les tables de pages ont un seul propriétaire,
- les pages normales peuvent avoir jusqu'à deux propriétaires,
- au moment de l'ajout d'un deuxième propriétaire à une page, ce dernier est autorisé, pour cette page, par le premier propriétaire,
- dans ce dernier cas, la première VM doit avoir été autorisée à partager des pages avec la seconde par configuration (cf MUS),
- les arbres p2m des VM et de leurs périphériques sont identiques.

Il est important de noter que ces propriétés ne sont valides que dans les cas qui nous intéressent. Il est par exemple tout à fait possible d'imaginer des cas où une même page est mappée par trois VM. En pratique, cela n'arrive pas dans les systèmes que nous produisons.

7.3 MSM

Le MSM est un module additionnel à Xen. Son but est d'intercepter les modifications à l'arbre p2m de chaque VM, base du confinement mémoire, et de vérifier les propriétés de confinement associées.

Le code p2m de Xen a été modifié pour qu'il invoque le MSM lorsque des changements dans les tables de translation p2m sont en passe d'être effectuées. Le MSM vérifie alors si l'opération est autorisée, auquel cas Xen peut la réaliser. Sinon, l'opération n'est pas effectuée et le domaine est arrêté.

Le code du MSM, ainsi que les propriétés de sécurité qu'il garantit, sont prouvés à l'aide de l'outil Frama-C.

Cette approche permet d'apporter des garanties sur le code qui est réellement exécuté, plutôt que sur un modèle purement abstrait et faiblement lié au code réel. Xen est un projet vivant qui évolue rapidement. Il est donc important de ne pas trop s'éloigner de la version officielle open-source afin de pouvoir bénéficier des améliorations de sécurité, des corrections d'anomalies logicielles et des contournements de problèmes matériels[20].

Cet argument incite à modifier Xen de la façon la plus modulaire et la plus portable possible dans notre démarche de démonstration du confinement mémoire. Il est donc souhaitable que les preuves faites soient au plus près du code existant. Dans ce cadre, l'utilisation d'outils tels que Frama-C apparaît comme intéressante.

7.4 Frama-C

Frama-C (pour FRAmework for Modular Analysis of C code) est une plate-forme dédiée à l'analyse de programmes C créée par le CEA List et Inria. Elle est basée sur une architecture modulaire permettant l'utilisation de divers plugins avec ou sans collaborations. Les plugins fournis par défaut comprennent diverses analyses statiques (sans exécution du code analysé), dynamiques (avec exécution du code), ou combinant les deux.

[20] Comme Spectre et Meltdown.

Frama-C nous fournit un langage de spécification appelé ACSL (pour ANSI C Specification Language) et qui va nous permettre d'exprimer les propriétés que nous souhaitons vérifier sur nos programmes. Ces propriétés seront écrites sous forme d'annotations dans les commentaires, un peu comme les commentaires javadoc/doxygen.

Nous utilisons un plugin appelé WP pour Weakest Precondition. À partir des annotations ACSL et du code source, le plugin génère des obligations de preuves, qui sont des formules logiques dont la satisfiabilité doit être vérifiée. Cette vérification peut être faite de manière manuelle ou automatique. Ici nous utilisons le prouveur automatique Alt-Ergo.

7.5 Appels du MSM

Le code de Xen a été modifié pour invoquer le MSM lors de la modification des tables p2m. Le MSM est alors prévenu des adresses machines qui vont être mappées/démappées, avant qu'elles le soient, et peut alors agir en conséquence. A ce niveau-là, le code de Xen informe le MSM si la page mappée/démappée est une page machine ou si elle fait partie de l'arbre de translation. Il est à noter que le MSM n'est pas intéressé par les adresses physiques en entrée des translations p2m. Seules les informations sur les adresses machines lui sont utiles.

Le MSM est également invoqué lors de la mise en place ou du retrait des racines des tables p2m. Il peut alors vérifier qu'une seule et même table est utilisée pour les CPU et les périphériques de la VM. Lors du retrait des tables p2m, le MSM considère que le domaine est en cours de destruction et démarque les pages utilisées par ce domaine.

Dans le code de gestion des tables p2m de Xen, à chaque fois qu'une table de page du p2m est mappée, le MSM est également invoqué pour vérifier que la page est connue comme faisant partie des tables de pages p2m.

Le MSM maintient une "modélisation" des arbres p2m, en s'affranchissant de la notion d'arbre et en ne conservant pour chacun page machine que les informations dont il a besoin : les propriétaires, des compteurs de références et le fait que la table soit une page normale ou faisant partie d'un arbre p2m. Les propriétés garantissant le confinement sont exprimées sur cette modélisation des arbres p2m.

7.6 MUS

Lorsqu'une VM tente de mapper une page appartenant à une autre VM, le MSM va dans un premier temps vérifier que cette dernière a bien autorisé cette opération par le mécanisme des grant-tables. Puis il va interroger un petit module, appelé le MUS (pour Médiateur Unique de Sécurité), qui va lui indiquer, indépendamment des grant-tables et de la politique de sécurité XSM, quels sont les mappings croisés autorisés sur le système. Le MUS est généré automatiquement à partir des fichiers de configurations des VM, extrêmement faciles à vérifier[21].

[21] Ce qui n'est pas le cas d'un fichier de configuration XSM.

7.7 Illustration

Des axiomes sont insérés dans le code du MSM. Ils sont vus :

- par Xen comme des assertions portant sur l'état des données manipulées par le MSM. Si ces assertions sont violées, le système s'arrête de fonctionner et ne peut pas se mettre dans un état qui ne soit pas conforme.
- par Frama-C comme des propriétés vraies à un instant donné et sur lesquelles il est possible de raisonner.

Les invariants sont la transcription en ACSL des propriétés de sécurité sur le confinement mémoire exprimée plus tôt. Ils sont vérifiés en entrée et en sortie de chaque fonction du MSM modifiant son état, ainsi que dans l'état initial, ce qui, par récurrence, montre qu'ils sont vérifiés à tout instant.

```
#define prop3(p) (p->pt == 0 || (p->own1 == 0 && p->own2 == 0))
#define Prop3    (\forall integer i; 0 <= i < psz ==> prop3((&pmsm[i])))
/*@
  ...
  requires Prop3;
  ...
  ensures Prop3;
  ...
*/
static void pmsm_add_pg(unsigned int mfn, ...)
{
    Assert(pmsm[mfn].pt == 0, mfn);
    ...
}
```

Dans cet extrait du code du MSM, la propriété qu'une table de page ne peut être partagée est exprimée par *prop3*. L'invariant *Prop3* étend cette propriété à l'ensemble des pages du système. La fonction *pms_add_pg* est appelée lorsqu'une page machine est ajoutée à un arbre p2m. En entrée, la proposition *Prop3* est supposée vérifiée et sera prouvée comme étant vérifiée en sortie par Frama-C après analyse du code C de *pmsm_add_pg*.

7.8 Retour d'expérience

Cela montre qu'il est possible de prouver formellement, grâce à un outil open-source, des propriétés de sécurités d'un hyperviseur également open-source.

En revanche, cela n'a pas été aussi facile que nous l'avions espéré, l'expérience relevant parfois du parcours initiatique. Par exemple, quand un prouveur automatique de Frama-C ne vient pas à bout d'une obligation de preuve quasiment instantanément, il n'est pas facile de décider si c'est parce qu'il a besoin de plus de temps, ou parce qu'on a demandé de prouver quelque chose de faux ou plus généralement d'impossible à prouver par ce prouveur là. Ajouter du temps, ajouter un autre prouveur automatique sont autant de stratégies envisageables, partiellement incompatibles et pas toujours gagnantes.

8 Conclusion

Notre utilisation de Xen nous a démontré qu'il est possible de construire des systèmes présentant de hautes propriétés de sécurité à base de composants et d'outils open-source. Xen est un bon choix pour cela.

Nous pensons que le monde des hyperviseurs est en train de vivre le même type de "banalisation" que les systèmes d'exploitation. Pour nous, Xen est aux hyperviseurs ce que Linux est aux systèmes d'exploitation.