Pip, un proto-noyau fait pour renforcer la sécurité dans les objets connectés

Quentin Bergougnoux, Julien Iguchi-Cartigny, Gilles Grimaud

Compas'2017 : Parallélisme/ Architecture / Système
Sophia Antipolis, France, du 27 au 30 juin 2017

Pip, un proto-noyau fait pour renforcer la sécurité dans les objets connectés

Quentin Bergougnoux, Julien Iguchi-Cartigny, Gilles Grimaud

Université Lille 1,
Laboratoire CRIStAL
{quentin.bergougnoux,julien.iguchi-cartigny,gilles.grimaud}@univ-lille1.fr

Résumé
Avec le développement et l'évolution de plus en plus rapides des objets connectés (Internet of Things), la sécurité et la confidentialité sont devenues des propriétés désirées par chaque constructeur dans leurs appareils, moyennant un coût, en termes de performances, moindre. En tant que moyen d'assurer mathématiquement les propriétés désirées, la preuve formelle a fait son entrée dans le domaine du développement noyau. Cependant, la vérification formelle nécessite beaucoup de travail et impose plusieurs contraintes : la moindre modification du modèle ou du code provoque de nombreuses modifications sur la preuve. Comme réponse à ce problème, nous présentons Pip, un proto-noyau n'assurant que la propriété voulue, l'isolation mémoire, par le biais d'une base de confiance prouvée, tout en laissant le code utilisateur gérer les fonctionnalités restantes.

Mots-clés : isolation, modèle, exo-noyau, preuve, TCB

1. Introduction

D'ordinaire, quatre fonctionnalités semblent nécessaires pour assurer la stabilité d'un micro-noyau, qui ont été énoncées par J. Liedtke par sa définition des micro-noyaux L4 : la séparation des espaces d'adressage, la gestion des threads, l'ordonnancement et la communication inter-processus [11]. Cependant, dans le cadre des objets connectés, ces fonctionnalités ne sont parfois pas toutes nécessaires, et dans le cadre d'une vérification formelle, la seule propriété critique dans le cadre d'une assurance de sécurité est l'isolation mémoire, c'est-à-dire qu'une tâche compromise ne pourra ni compromettre une autre tâche, ni le système lui-même.
Il est assez commun, par ailleurs, de voir des logiciels système, tels que des noyaux, conçus dans des langages de bas-niveau, tels que le C et l'assembleur. Ces derniers sont idéaux lorsqu'il s'agit d'atteindre des objectifs de performance, mais se révèlent être faibles pour ce qui est des objectifs relatif à la preuve formelle, avec pour cause, par exemple, l'absence de typage fort. Nous proposons d'assurer l'isolation mémoire par le biais d'un modèle et d'une preuve réalisés par le biais de l'assistant de preuves Coq, et en exécutant directement ce modèle. Pip est une nouvelle architecture, dite de proto-noyau, adaptée au monde de l'IoT, dont la propriété d'isolation est prouvable, qui gère uniquement la configuration de la MMU,

Ce travail a été partiellement financé les contrats de projet européen CELTIC+ ODSI (2016-) et ORANGE.

traduction d'adresses ainsi que le circuit d'interruptions, laissant ainsi les fonctionnalités restantes à la charge du logiciel s'exécutant par-dessus.
Nous abordons dans cette publication l'aspect conception de Pip (l'aspect preuve étant un travail conjoint réalisé en parallèle [8]). Dans un premier temps, le modèle d'isolation mémoire, indépendant de toute architecture, sera présenté (Section 1), ainsi que la propriété d'isolation associée. Par la suite, l'implémentation de ce modèle sera abordée, ainsi que les différentes problématiques associées (Section 2). Enfin, l'impact sur les performances sera abordé, ainsi que les travaux à venir sur ce sujet.

2. Conception du modèle

L'accès à la mémoire physique par les applications en mode utilisateur est contrôlée, au cours de l'exécution, par le biais du circuit de MMU (*Memory Management Unit*), qui permet d'une part de gérer des adresses virtuelles et plusieurs environnements de mémoire, et permet d'autre part une association entre adresse virtuelle et physique en fonction de l'environnement courant. La configuration de ce circuit, au travers de l'association de pages physiques à des adresses virtuelles, est au cœur de l'isolation mémoire et représente le support de la preuve formelle à établir. Dans la suite de ce document, on appellera *page physique* une partie de la mémoire physique, identifiée par son adresse. Par ailleurs, on appellera *page virtuelle* la projection d'une page physique dans un espace d'adressage virtuel, ce dernier étant un ensemble de pages virtuelles.
La conception de Pip passe en premier lieu par l'application d'un modèle à l'état de la mémoire physique. Dans ce modèle, des *partitions* sont manipulées, représentant des espaces d'adressage, c'est-à-dire une structure d'hypervision regroupant tables de configuration de la MMU pour cette partition, structures de contrôle de Pip et pages. Ce modèle est basé sur deux concepts, qui sont *l'isolation* et *la dérivation*. La combinaison de ces deux concepts permet d'expliciter clairement la propriété que nous cherchons à assurer.

2.1. Architecture globale

Par essence, il n'est pas possible d'effectuer des opérations de bas niveau (configuration de la MMU, des interruptions...) directement depuis un code Coq. Afin de pallier à ce problème, nous avons conçu deux couches d'abstraction du matériel, en utilisant l'assembleur et le C, dont l'interface expose des opérations minimales et atomiques sur lesquelles le code du modèle peut se reposer :

— `Memory Abstraction Layer`, qui expose différentes fonctions dont le but est de manipuler, entre autres, les tables d'indirection,

— `Interrupt Abstraction Layer`, qui expose les fonctions permettant de configurer le PIC (*Programmable Interrupt Controller*), le flot de contrôle du système, ainsi que les interruptions et exceptions.

Il en va de même pour le code de démarrage du noyau (voir figure 2).

2.2. Isolation et dérivation

L'isolation mémoire est vue par Pip comme étant la séparation absolue des espaces d'adressage. Une partition (appelée partition *parent*) peut déléguer une partie de son espace d'adressage (c'est-à-dire un ensemble de pages) à une (et une seule) partition dite *enfant*, créant ainsi un modèle hiérarchique de la mémoire. En tant que telle, l'isolation n'est valable qu'entre les enfants d'une partition. L'isolation générale du système est alors déductible des différentes

dérivations effectuées depuis la première partition démarrée par Pip, comme expliqué dans la figure 1.

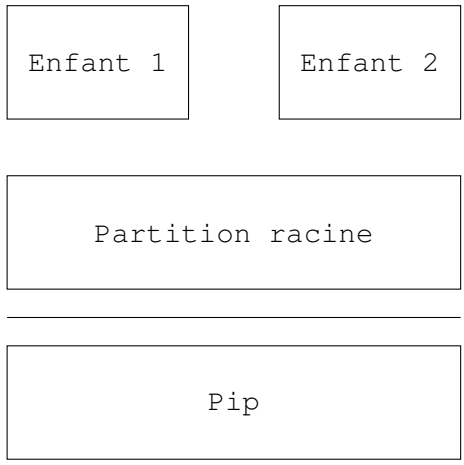

FIGURE 1 – Modèle d'isolation mémoire

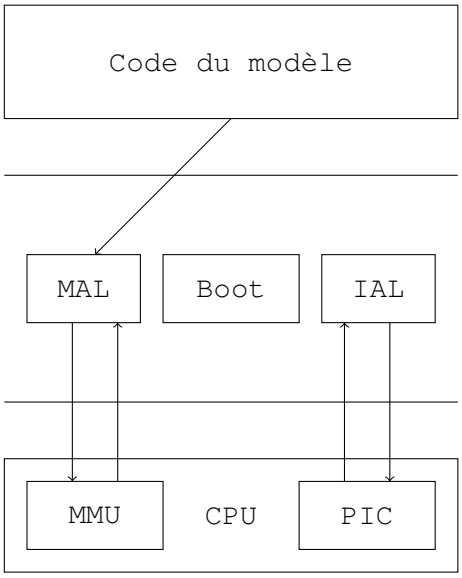

FIGURE 2 – Architecture de Pip

Définition 1. *La dérivation est définie comme suit : chaque partition peut donner un page virtuelle accessible dans son espace d'adressage à au plus une de ses partitions enfant.*

Définition 2. *L'isolation est définie comme suit : les pages virtuelles visibles dans l'espace d'adressage d'un enfant ne le sont jamais dans les espaces d'adressage des autres enfants.*

Comme dit précédemment, une partition peut dériver une partie de son espace d'adressage à un de ses enfants, et ainsi de suite. Il est possible d'imaginer de nouvelles dérivations à partir de l'exemple présenté précédemment. Ce modèle rend possible, de façon intrinsèque, l'exécution de plusieurs instances de Pip les unes au-dessus des autres, chacune exploitant les services de son parent, créant ainsi une tour de virtualisation [12].

2.3. Propriétés

Afin de raisonner sur l'état du système et de la MMU, il est nécessaire de définir plusieurs propriétés, dont la présence ou l'absence sur une adresse virtuelle donnée définiront les actions que l'exo-noyau autorisera ou refusera. Ces propriétés, appliquées à chaque page, font partie du modèle et sont au nombre de cinq :

— *present*, qui définit une entrée de page valide et correctement associée à l'espace d'adressage virtuel dans une partition donnée,
— *accessible*, qui définit une entrée de page virtuelle pour laquelle la partition a accès au contenu (équivalent des droits noyau/utilisateur),
— *is_partition_descriptor*, qui définit une entrée de page représentant une autre partition,
— *derivated*, qui définit une page d'ores et déjà dérivée à une partition enfant quelconque,
— *endpoint*, résultant de la factorisation des propriétés *accessible* et $\neg$*derivated*, et représente donc une page non-dérivée, accessible dans la partition.

Ces propriétés représentent le point central du contrôle d'accès aux méthodes de Pip : elles représentent un état du système qui, à lui seul, permet de déterminer si l'opération demandée rompt la propriété d'isolation ou non.
Par exemple, lors de la dérivation d'une page, les contrôles suivants sont effectués sur la page virtuelle :

- *present* : la page à dériver doit exister dans l'espace d'adressage de la partition appelante,
- *accessible* : la page à dériver doit être accessible depuis l'espace utilisateur afin de ne pas compromettre les pages privilégiées du noyau (cette vérification couvre le contrôle !*is_partition_descriptor*, une telle page étant déjà non accessible),
- !*derivated* : la page à dériver ne doit pas avoir d'ores et déjà été dérivée dans une autre partition.

La page sera alors dérivée dans la partition enfant, si et seulement si les contrôles précédents sont vérifiés.

3. Du modèle à l'implémentation

3.1. Travaux existants

De nombreux noyaux utilisent des fonctionnalités de l'architecture pour implémenter la virtualisation. Par exemple, sur l'architecture x86, il est possible d'utiliser un modèle étendu de tables de pages [15] afin de gérer des machines virtuelles, et de combiner ça à une gestion de processeurs virtuels afin d'abstraire totalement le matériel [1, 13]. Cependant on cherche ici à ne pas utiliser les fonctionnalités exclusives à une architecture précise pour implémenter la virtualisation : une couche d'abstraction matérielle (*HAL*) est utilisée, abstrayant les opérations de manipulation de la MMU ou du CPU requises par le modèle.
Par ailleurs, nombre de projets concernant la preuve formelle sur des logiciels système conçus dans des langages de haut-niveau ont vu le jour, incluant par exemple seL4 [3] et House [4]. Singularity, ayant notamment mis en avant le lien étroit entre langage de haut-niveau, architecture du système et preuve [7], utilise une approche différente : toutes les applications fonctionnent dans le même environnement, et l'isolation est assurée par le langage. L'hyperviseur NOVA a, en outre, également pour objectif d'assurer une fiabilité maximale du système en se basant sur un TCB (Trusted Computing Base, base élémentaire de code de confiance) minimal [14], mais n'a aucun objectif relatif à la preuve formelle. Ces projets, hormis House qui est un noyau monolithique modulaire, sont basés sur un modèle de micro-noyau, impliquant l'implémentation de fonctionnalités qui ne sont pas recherchées ici.
D'un point de vue architectural, il est indispensable de citer le modèle d'exo-noyau comme défini par le MIT dans le cadre de la conception du premier exokernel [10], qui conçoit un noyau très proche de la machine réelle, utilisant un nombre minimal voire nul d'abstractions, et centralisant dans le noyau les mécanismes de multiplexage uniquement. Des projets similaires d'exo-noyaux ont également vu le jour, tels que Barrier [6], qui ne gère cependant que l'isolation mémoire entre un noyau et les modules chargés en amont. Notre but ici est différent : il s'agit de voir les modules d'un éventuel noyau comme des partitions à part entière, et de ne pas différencier l'isolation de ceux-ci de celle appliquée aux programmes utilisateur afin de respecter la propriété recherchée à tout moment de l'exécution.
En ce qui concerne le développement d'un noyau en utilisant un langage fonctionnel, cela est effectivement possible, de la même façon que House, même si ce n'est pas l'intérêt ou le but principal du langage. seL4 était orienté sur la partie modèle [5], dans le sens où ses concepteurs se sont penchés sur le modèle et sa preuve au travers de l'assistant de preuves Isabelle, puis ont

converti ce dernier en code C. La conformité du code C généré par rapport au modèle [9] était ensuite vérifiée. À l'inverse, House est compilé directement depuis le code Haskell, utilisant une quantité très faible de code C, au travers d'une version modifiée de Glasgow Haskell Compiler, hOp.
Le modèle que nous avons choisi ici est un modèle de proto-noyau, c'est-à-dire de noyau minimaliste reportant une grande majorité des fonctionnalités du système dans l'espace utilisateur, tout en exposant dans le noyau une abstraction du matériel utilisé. Notre modèle ne gère que l'isolation mémoire par le biais des interfaces fournies par l'abstraction du matériel, exploitées par une API en Coq.

3.2. Exécuter le modèle

Afin de rendre la chaîne de développement la plus simple possible, nous avons souhaité exécuter directement le modèle Coq au-dessus du matériel. Or, exécuter directement du code fonctionnel en tant que code noyau n'est pas immédiat : il n'existe pas d'environnement Coq ni de librairie standard dans Pip. Pour pallier à ce problème, nous avons tiré avantage du fait que le code du modèle exploite les interfaces fournies par la couche d'abstraction matérielle de Pip : le code est donc, en parallèle au modèle fonctionnel, rédigé dans un style se rapprochant du modèle impératif. Nous employons donc un convertisseur de code qui va générer un code C exécutable depuis le code Coq du noyau [1].
En revanche, afin d'assurer la propriété d'isolation au sein du noyau, la conformité du code généré par rapport au code Coq doit être vérifiée, et est actuellement en cours de réalisation dans le cadre de travaux annexes.

3.3. Structures de contrôle

Pip doit garder une trace des dérivations, dans le but d'en autoriser de nouvelles en fonction de l'état actuel du système et des règles d'isolation précédemment définies (voir section 2.1). À ce but, le proto-noyau va utiliser des *shadow pages*, qui sont des clones des tables de traduction d'adresse de la MMU. Ainsi, pour chaque page de configuration de la mémoire virtuelle, deux pages supplémentaires servant à stocker des informations supplémentaires seront associées. La consommation mémoire est alors plus importante, mais ce modèle fournit un moyen efficace, extensible et indépendant de l'architecture de décrire l'état du système.
Ces tables vont stocker plusieurs informations essentielles, telles que la partition dans laquelle une page a été dérivée (si elle l'a été), de même que l'adresse virtuelle de cette page dans la partition parent. Ces *shadow pages* remplissent un double rôle : fournir les informations requises (pour la vérification des propriétés), et donner par ailleurs accès à des informations utiles à l'API afin que cette dernière puisse effectuer rapidement et efficacement les opérations demandées. De même, une liste chaînée de pages est employée en tant que table : chaque couple d'entrées contiendra l'adresse physique d'une table d'indirection, quelle qu'elle soit, ainsi que son adresse virtuelle dans l'environnement parent, afin de pouvoir aisément la retrouver en cas d'appel à l'API.
De même, chaque partition est susceptible de vouloir gérer les interruptions à sa façon, comme par exemple dans le cas où deux systèmes Linux s'exécutent côte à côte. A ce but, Pip doit connaître les fonctions à appeler en cas d'interruption, ainsi que la pile sur laquelle passer (et ce afin de se rapprocher au plus de l'architecture cible). Ces informations sont stockées par une page non privilégiée, c'est-à-dire qu'une partition supposée gérer des interruptions peut modifier son vecteur d'interruptions comme bon lui semble. Cette page est alors appelée *Virtual Interrupt Vector*.

1. Code disponible sur `https://www.github.com/2xs/coq2c`.

Par ailleurs, chaque partition est identifiée dans le système par l'adresse physique d'une page appelée *Partition Descriptor*. Cette page contient l'adresse de toutes les structures de contrôle associées à la partition (Shadow Pages, liste des tables d'indirection) ainsi que l'adresse du vecteur d'interruptions virtuel associé à la partition.

3.4. API minimale

Pip fournit une API minimale mais suffisante pour permettre à une partition s'exécutant au-dessus de gérer son espace d'adressage, et de le dériver à des partitions enfant. Pour assurer un meilleur support des architectures, il a été choisi de travailler sur des pages de 4ko, qui est une taille de page commune, au moins, aux architectures ARM et x86.
Ces services fournis sont au nombre de 8, et permettent la création ou la suppression d'une partition, la préparation, l'ajout ou la suppression d'une association d'adresse, l'échange de deux associations entre deux partitions distinctes ou encore la récupération des pages inutilisées au sein des tables de configuration d'une partition. Chaque appel à l'une de ces fonctions va être soumis à un contrôle d'accès en fonction de la présence ou non des propriétés définies précédemment.
En plus de ces appels systèmes permettant une gestion complète des partitions, Pip implémente deux appels systèmes complémentaires permettant de gérer le flot de contrôle du système, et donc de rediriger un signal vers une partition enfant ou parent, ou encore de reprendre l'exécution d'une partition interrompue. Le fonctionnement de ces appels est similaire aux instructions `INT` et `IRET` des processeurs x86, ou `SVC` et `MOVS` de certains processeurs ARM. A noter cependant que Pip ne gère que la capture d'interruption et le routage du signal, mais pas le traitement associé à l'interruption, qui sera implémenté dans une partition dédiée. Ainsi, selon le type de l'interruption reçue, le routage sera fait différemment :

— Une interruption logicielle, telle qu'une faute ou un appel système, sera relayée directement à la partition parente, ou arrêtera le système dans le cas d'une faute de la partition racine,

— Une interruption matérielle sera relayée directement à la partition racine, qui se chargera ensuite de la relayer à la partition associée à son traitement, par exemple, un pilote de carte réseau.

4. Limitations actuelles du modèle

De par notre modèle d'isolation, plusieurs facilités mises en place par les systèmes, telles que par exemple la mémoire partagée, ne sont pas immédiatement et efficacement implémentables, de même que les appels *cross-partition*, les signaux ne pouvant être redirigés qu'à un enfant ou parent immédiat. Ces limitations, actuellement bloquantes d'un point de vue performance et facilité de portage, ne sont néanmoins pas punitives dans le cadre d'un portage de système : un portage du système temps réel FreeRTOS est d'ores et déjà fonctionnel, fournissant une isolation entre plusieurs instances du système, ou entre les différentes tâches d'un même système.

5. Conclusion et perspectives

En résumé, Pip est un noyau d'hypervision à l'architecture nouvelle, dont la seule propriété qu'il doit assurer à tout moment, l'isolation mémoire, est prouvable. Son API est fiable et suffisante pour permettre à d'autres systèmes de s'exécuter au-dessus tout en assurant une isolation totale entre ces derniers. Le langage Coq étant exécutable par le biais d'un

convertisseur de code, la chaîne de développement s'en trouve grandement simplifiée. Le modèle est dès à présent exécutable sur plusieurs architectures, incluant la carte de développement Intel Galileo [2]. Plusieurs problématiques restent néanmoins à explorer, telle que le risque constitué par les contrôleurs DMA, impliquant une virtualisation de l'IOMMU également [2, 16], ou encore l'amélioration de la couche de gestion des interruptions, dans l'optique d'un éventuel portage Linux ainsi que d'une évaluation fiable des performances globales.

Bibliographie

1. Adams (K.) et Agesen (O.). – A comparison of software and hardware techniques for x86 virtualization. *SIGARCH Comput. Archit. News*, vol. 34, n5, octobre 2006, pp. 2–13.
2. Amit (N.), Ben-Yehuda (M.), Tsafrir (D.) et Schuster (A.). – viommu : Efficient iommu emulation. – In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, *USENIXATC'11*, USENIXATC'11, pp. 6–6, Berkeley, CA, USA, 2011. USENIX Association.
3. Derrin (P.), Elphinstone (K.), Klein (G.), Cock (D.) et Chakravarty (M. M. T.). – Running the manual : An approach to high-assurance microkernel development. – In *Proceedings of the 2006 ACM SIGPLAN Workshop on Haskell*, *Haskell '06*, Haskell '06, pp. 60–71, New York, NY, USA, 2006. ACM.
4. Diatchki (I. S.), Hallgren (T.), Jones (M. P.), Leslie (R.) et Tolmach (A.). – Writing systems software in a functional language : An experience report. – In *Proceedings of the 4th Workshop on Programming Languages and Operating Systems*, *PLOS '07*, PLOS '07, pp. 1 :1–1 :5, New York, NY, USA, 2007. ACM.
5. Elphinstone (K.), Klein (G.), Derrin (P.), Roscoe (T.) et Heiser (G.). – Towards a practical, verified kernel. – In *Proceedings of the 11th USENIX Workshop on Hot Topics in Operating Systems*, *HOTOS'07*, HOTOS'07, pp. 20 :1–20 :6, Berkeley, CA, USA, 2007. USENIX Association.
6. Hua (J.) et Sakurai (K.). – Barrier : A lightweight hypervisor for protecting kernel integrity via memory isolation. – In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, *SAC '12*, SAC '12, pp. 1470–1477, New York, NY, USA, 2012. ACM.
7. Hunt (G. C.) et Larus (J. R.). – Singularity : Rethinking the software stack. *SIGOPS Oper. Syst. Rev.*, vol. 41, n2, avril 2007, pp. 37–49.
8. Jomaa (N.), Nowak (D.), Grimaud (G.) et Iguchi-Cartigny (J.). – Preuve formelle d'isolation mémoire dynamique à base de MMU. – In *Vingt-sixièmes Journées Francophones des Langages Applicatifs (JFLA 2015)*, pp. 297–300, Le Val d'Ajol, France, janvier 2015.
9. Klein (G.), Derrin (P.) et Elphinstone (K.). – Experience report : Sel4 : Formally verifying a high-performance microkernel. – In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*, *ICFP '09*, ICFP '09, pp. 91–96, New York, NY, USA, 2009. ACM.
10. Leschke (T.). – Achieving speed and flexibility by separating management from protection : Embracing the exokernel operating system. *SIGOPS Oper. Syst. Rev.*, vol. 38, n4, octobre 2004, pp. 5–19.
11. Liedtke (J.). – On micro-kernel construction. *SIGOPS Oper. Syst. Rev.*, vol. 29, n5, décembre 1995, pp. 237–250.
12. Morgan (B.), Alata (E.), Nicomette (V.) et Averlant (G.). – Abyme : un voyage au coeur des

2. Code disponible sur `https://www.github.com/2xs/pipcore`.

hyperviseurs récursifs. – In *Symposium sur la sécurité des technologies de l'information et des communications 2015 (SSTIC)*, 2015.

13. Nakajima (J.), Lin (Q.), Yang (S.), Zhu (M.), Gao (S.), Xia (M.), Yu (P.), Dong (Y.), Qi (Z.), Chen (K.) et Guan (H.). – Optimizing virtual machines using hybrid virtualization. – In *Proceedings of the 2011 ACM Symposium on Applied Computing*, *SAC '11*, SAC '11, pp. 573–578, New York, NY, USA, 2011. ACM.
14. Steinberg (U.) et Kauer (B.). – Nova : A microhypervisor-based secure virtualization architecture. – In *Proceedings of the 5th European Conference on Computer Systems*, *EuroSys '10*, EuroSys '10, pp. 209–222, New York, NY, USA, 2010. ACM.
15. Wang (X.), Zang (J.), Wang (Z.), Luo (Y.) et Li (X.). – Selective hardware/software memory virtualization. *SIGPLAN Not.*, vol. 46, n7, mars 2011, pp. 217–226.
16. Willmann (P.), Rixner (S.) et Cox (A. L.). – Protection strategies for direct access to virtualized i/o devices. – In *USENIX 2008 Annual Technical Conference*, *ATC'08*, ATC'08, pp. 15–28, Berkeley, CA, USA, 2008. USENIX Association.