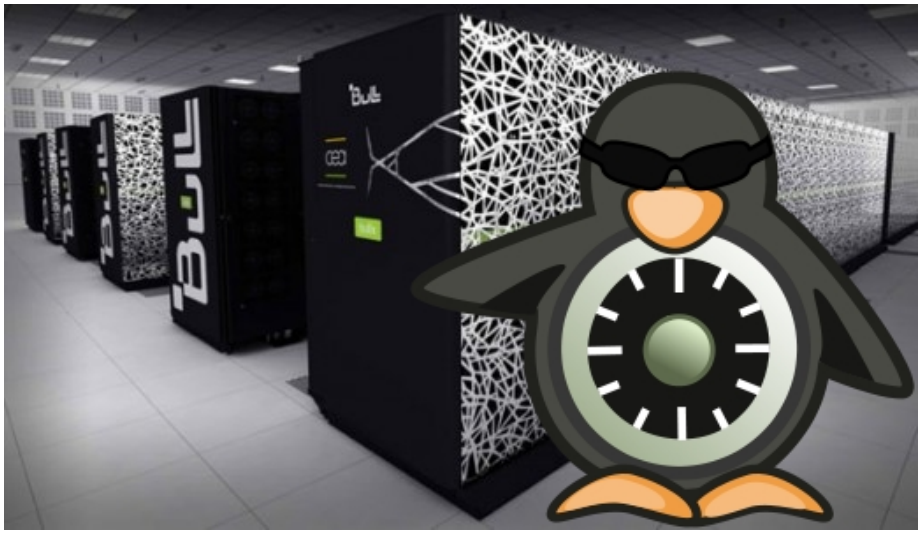
Optimisation des performances de SELinux



Rapport de stage

19 mars - 31 août 2012

Timothée Ravier

École Nationale Supérieure d'Ingénieurs de Bourges
Promotion 2012
Sécurité et Technologies Informatiques
Option : Sécurité des Systèmes Ubiquitaires

CEA, DAM, DIF, F-91297, Arpajon, France

Maître de stage

Mathieu Blanc
Ingénieur-chercheur
CEA

Tuteur académique

Christian Toinard
Professeur
ENSI de Bourges

Abstract

The current context of repeated computer break-in and data theft has made computer security one of the primary objectives for compagnies and governments. Access control models inherited from UNIX (Discretionary Access Control) were found to be unable to enforce basic security properties such as integrity or confidentiality. Mandatory Access Control was designed to overcome those limitations by enforcing a security policy on userspace programs running on a system. SELinux was created by the NSA as a comprehensive implementation of Mandatory Access Control. After being merged into the kernel in 2003, the code has been reviewed and improved by the community and enterprises such as Red Hat.

But security and performance are usually conflicting objectives, especially considering the Linux kernel. Security comes at a price as more controls means more instructions executed and less performance. In this report we will study the effects induced by the protection provided by SELinux. After a case comparison describing the state of the art in Mandatory Access Control methods, we will focus on the SELinux design and architecture. Performance measurements and improvements will be discussed in the last chapter.

On average, the performance impact due to SELinux is relatively low, but can be significant on specific workloads. This study analyze disk I/O related workflow in order to simulate cluster filesystem use by computing code. We show that opening and retrieving informations about files are the most affected operations whereas writing and reading suffer from less overhead with SELinux.

Finally, we discuss some ideas to improve performance globally by modifying a core SELinux component in the Linux kernel.

Résumé

Le noyau Linux embarque plusieurs mécanismes de contrôle d'accès dit «obligatoire» (*Mandatory Access Control* ou *MAC*). La particularité principale de ce type de contrôle d'accès est que les permissions ne sont pas contrôlées par les utilisateurs, mais par une configuration centrale appelée politique de sécurité. Un des avantages du *MAC* est la possibilité de limiter les droits d'accès sur le système même pour un programme exécuté avec des droits d'administrateur.

Toutefois, ce type de contrôle d'accès est très invasif dans le noyau Linux ; les performances peuvent potentiellement subir un impact important. Il est également assez complexe à configurer, l'objectif étant d'accorder à chaque application les droits nécessaires et suffisants à son exécution.

Nous nous sommes intéressé à SELinux, un des plus anciens MAC pour Linux (avec d'autres comme grsecurity ou RSBAC). SELinux est intégré dans la plupart des distributions classiques, notamment Fedora, Red Hat Enterprise Linux et Debian. L'objectif est d'optimiser certaines parties du code de SELinux pour diminuer l'impact sur les performances d'un système.

Nous commencerons par un état de l'art des solutions de contrôle d'accès obligatoire pour poursuivre sur le détail de l'architecture de SELinux et du fonctionnement à l'intérieur du noyau Linux. Nous présenterons alors nos résultats de tests de performance qui visent à simuler le comportement de codes de calcul puis nous évaluerons plusieurs pistes envisagées pour réduire l'impact de SELinux sur un système.

Remerciements

Je souhaite remercier tout particulièrement mon maître de stage Mathieu Blanc et Damien Gros, pour leur aide, leurs conseils et leur patience tout au long de ce stage.

Merci à l’ensemble de l’équipe qui m’a accueilli au CEA pour leur convivialité, à Christian Toinard qui a encadré et suivit mon stage et à Jérémy Briffaut pour ses conseils.

Enfin, je remercie les autres stagiaires pour leur bonne humeur et la bonne ambiance de travail de ces six mois de stage.

Table des matières

Table des figures

Table des listings

Liste des symboles

AppArmor *Application Armor* : Un mécanisme de contrôle d'accès obligatoire inclus dans le noyau Linux.

AVC *Access Vector Cache* : Cache dans le noyau Linux pour optimiser le nombre de recherches dans la politique SELinux.

DAC *Discretionary Access Control* : Contrôle d'accès discrétionnaire ou laissé à la discrétion de l'utilisateur.

HPC *High Performance Computing* : Calcul haute performance.

HRU Modèle théorique permettant de représenter les systèmes basés sur le contrôle d'accès discrétionnaire (*Discretionary Access Control*).

IDS *Intrusion Detection System* : Un élément matériel ou logiciel visant à détecter des activités sur un système ou un réseau en fonction de règles.

IPC *Inter-process communication* : Ensemble de méthodes permettant d'échanger des données entre processus.

IPS *Intrusion Prevention System* : Une extension des systèmes de détection d'intrusion (IDS) visant à prévenir et bloquer les comportements détectés.

LSM *Linux Security Modules* : Fonctions insérées (*hooks*) dans les appels système du noyau Linux permettant d'implémenter diverses méthodes de contrôle d'accès.

MAC *Mandatory Access Control* : Contrôle d'accès obligatoire, imposé par le système aux programmes.

MCS *Multi Categories Security* : Modèle de sécurité utilisant plusieurs catégories pour séparer des éléments.

MLS *Multilevel Security* : Modèle de sécurité multiniveau, généralement associé à des règles de confidentialité ou d'intégrité.

PaX Patch pour le noyau Linux, qui inclut notamment le support pour les pages mémoires non exécutables et diverses protections pour limiter la portée des attaques sur les programmes et le noyau.

PIGA *Policy Interaction Graph Analysis* : Analyse de graphes d'interactions générés à partir d'une politique de sécurité.

SELinux *Security Enhanced Linux* : Un mécanisme de contrôle d'accès obligatoire inclus dans le noyau Linux.

SMACK *Simplified Mandatory Access Control Kernel* : Un mécanisme de contrôle d'accès obligatoire inclus dans le noyau Linux.

TOMOYO Linux Un mécanisme de contrôle d'accès obligatoire inclus dans le noyau Linux.

TPE *Trusted Path Execution* : Exécution limitée aux programmes de confiance, placés dans des emplacements d'un système de fichier non modifiables par les utilisateurs non privilégiés.

Introduction

Ce rapport présente l'ensemble du travail effectué au CEA pendant mon stage de troisième année. Le sujet, *Optimisation des performances de SELinux*, est à la rencontre de deux mondes en totale opposition dans l'informatique en général et dans le noyau Linux pour notre cas d'étude : la sécurité et les performances.

En effet, introduire des contrôles de sécurité, c'est ajouter des instructions à exécuter, donc du temps d'exécution consommé sans bénéfice direct visible. C'est l'une des raisons pour lesquelles les solutions de sécurité présentes dans le noyau Linux sont sous-exploitées. Le créateur de Linux, Linus Torvalds, a par le passé rejeté de nombreux patchs de sécurité à cause de leur impact sur les performances du noyau. Les plus intéressants (par exemple PaX) ne seront probablement jamais acceptés dans la branche principale du noyau tant leur impact est important. Les performances et la sécurité sont donc bien deux objectifs distincts, et il est nécessaire de faire des compromis.

Les objectifs du CEA sont liés à ce compromis : les clusters de calcul doivent être à la fois performants et sûrs. Pour que la justesse des résultats soit assurée, il est impératif que les utilisateurs ne puissent pas modifier le système ou les fichiers d'autres utilisateurs. De plus, il est essentiel de pouvoir répartir le temps de calcul disponible entre plusieurs utilisateurs, et d'assurer qu'aucun abus ne sera possible.

La mise à disposition à la communauté scientifique européenne du super calculateur installé au Très Grand Centre de Calcul renforce ce besoin de sûreté et de sécurité. Le nombre d'utilisateur va croître et l'image du CEA serait fortement impactée si un incident de sécurité venait à impacter un super calculateur.

C'est dans ce contexte que nous allons étudier les performances d'un système avec SELinux. Nous déterminerons ensuite quels éléments peuvent être améliorés et quelles perspectives sont envisageables pour réduire l'impact de cette méthode de contrôle d'accès. Nous ne nous intéresserons donc pas à l'optimisation même des codes de calcul, qui relève d'un autre domaine.

Chapitre 1

Le CEA et le domaine du calcul haute performance

1.1 Présentation de l'entreprise

1.1.1 Le CEA

Acteur majeur de la recherche, du développement et de l'innovation, le Commissariat à l'énergie atomique et aux énergies alternatives intervient dans trois grands domaines : les énergies décarbonées, la Défense et la sécurité globale, les technologies pour l'information et la santé.

Pour être au plus haut niveau de la recherche, le CEA compte plusieurs atouts :
- une culture croisée entre ingénieurs et chercheurs, propice aux synergies entre recherche fondamentale et innovation technologique ;
- des installations exceptionnelles (supercalculateurs, réacteurs de recherche, lasers de puissance…) ;
- une forte implication dans le tissu industriel et économique.

Le CEA est structuré autour de cinq pôles opérationnels (Défense, énergie nucléaire, recherche technologique, sciences de la matière et sciences du vivant) et quatre pôles fonctionnels (maîtrise des risques, ressources humaines et formation, stratégie et relations extérieures, gestion des systèmes d'information), implantés dans 10 centres répartis dans toute la France.

Il développe de nombreux partenariats avec les autres organismes de recherche, les collectivités locales et les universités.

Reconnu comme un expert dans ses domaines de compétence, le CEA est pleinement inséré dans l'espace européen de la recherche et exerce une présence croissante au niveau international.

Le CEA compte actuellement 15 700 personnes pour un budget de 4,3 milliards d'euros.

1.1.2 La direction des applications militaires

La Direction des Applications Militaires (DAM) constitue le pôle Défense du CEA. Ce pôle a pour mission principale de concevoir, fabriquer, maintenir en condition opérationnelle puis démanteler les têtes nucléaires qui équipent les forces océaniques et aéroportées.

Apportant aux pouvoirs publics la garantie que ces têtes sont fiables et sûres, il est l'un des principaux artisans de la crédibilité de la dissuasion nucléaire française.
Aujourd'hui, l'objectif du Pôle Défense, est de continuer à assurer sur le long terme cette capacité de dissuasion sans recourir aux essais nucléaires, définitivement arrêtés depuis 1996.

Ses missions portent également sur :
- l'approvisionnement en matières nucléaires pour les besoins de la Défense ;
- la propulsion nucléaire navale ;
- la lutte contre la prolifération et le terrorisme et la sécurité globale.

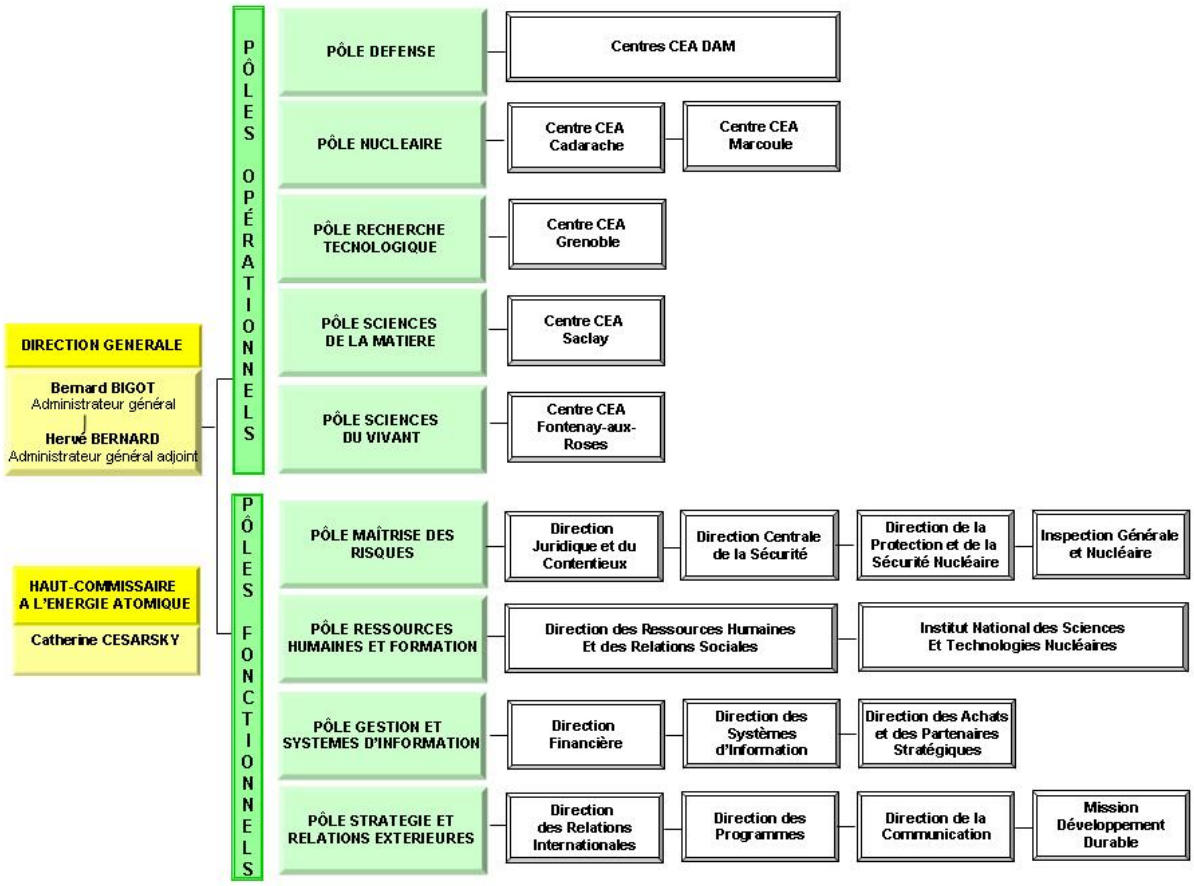


Figure 1.1 – Organigramme du CEA

Le Pôle Défense publie bon nombre de ses résultats scientifiques et développe des collaborations avec d'autres acteurs de la Recherche. Il valorise ses activités auprès des industriels par le transfert de technologies et le dépôt de brevets. Il s'est également engagé dans une importante démarche de certification qualité de l'ensemble de ses activités liées aux armes nucléaires.

La DAM compte aujourd'hui 4 750 collaborateurs, menant des activités réparties entre la recherche de base, le développement et la fabrication. Son budget est d'environ 1,8 milliards d'euros.

Elle comprend :
- Un échelon Direction ;
- Cinq directions d'objectifs (DOB), qui ont pour mission la définition et la gestion des programmes et le pilotage des projets ;
- Cinq directions opérationnelles (DOP), qui ont pour mission la réalisation des produits conformément aux directeurs d'objectifs. Les directeurs opérationnels de la DAM sont les directeurs de centre ;
- Quatre directions fonctionnelles (DF), qui ont des missions de directive, de soutien et de contrôle, pour le compte du DAM.

La DAM est implantée sur cinq centres :
- Valduc, en Bourgogne ;
- Le Ripault, en Touraine ;
- le Cesta, en Aquitaine ;
- DAM - Île de France (DIF) ;
- Gramat (CEG) en Midi-Pyrénées.

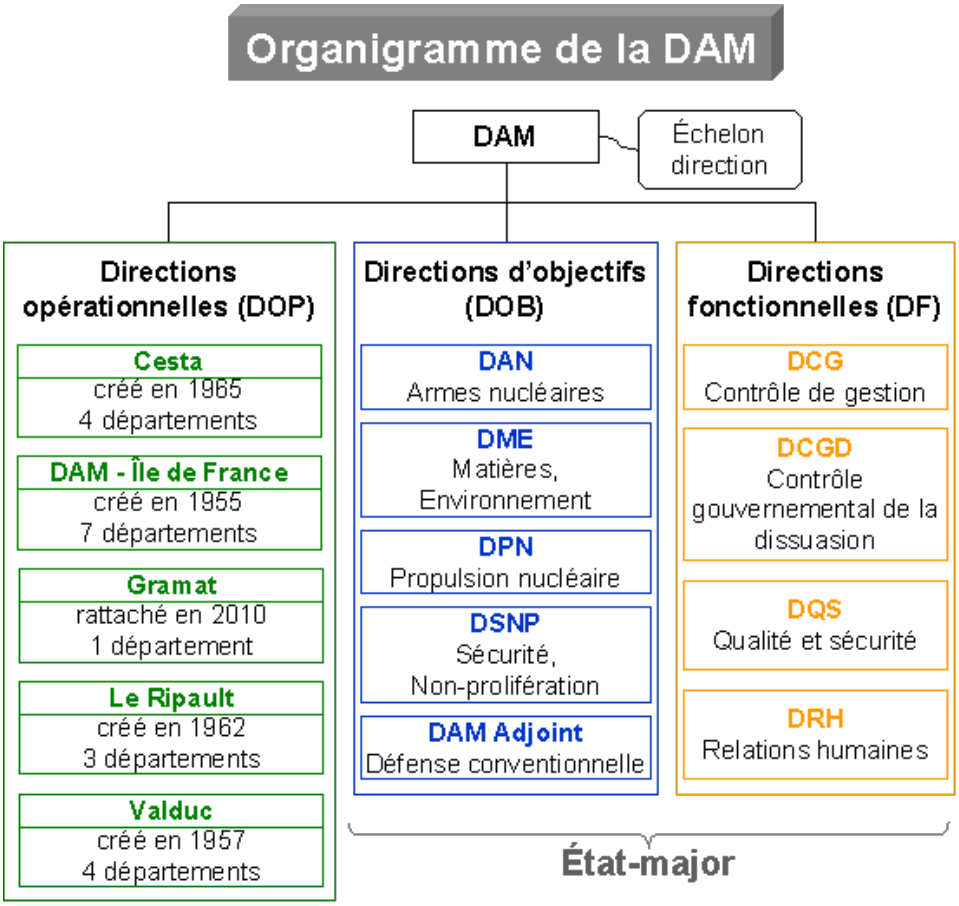


Figure 1.2 – Organigramme de la DAM

1.1.3 Le centre DAM Île de France

Le CEA/DAM - Île de France (DIF) est l'une des directions opérationnelles de la DAM.
Le site de la DIF compte environ 2000 salariés CEA et accueille quotidiennement environ 600 salariés d'entreprises extérieures.

Elle comprend deux sites :
- Le site de Bruyères le Châtel, situé à environ 40 km au sud de Paris, dans l'Essonne ;
- Le site du Polygone d'expérimentation de Moronvilliers (PEM), situé à 20 km de Reims, dans la Marne.

Les missions de la DIF comprennent :
- La conception et garantie des armes nucléaires, grâce au programme Simulation. L'enjeu consiste à reproduire par le calcul les différentes phases du fonctionnement d'une arme nucléaire et à confronter ces résultats aux mesures des tirs nucléaires passés et aux résultats expérimentaux obtenus sur les installations actuelles (machine radiographique Airix, lasers de puissance, accélérateurs de particules) ;
- La lutte contre la prolifération et le terrorisme, en contribuant notamment au programme de garantie du Traité de Non Prolifération et en assurant l'expertise technique française pour la mise en œuvre du Traité d'Interdiction Complète des Essais Nucléaires (TICE) ;
- L'expertise scientifique et technique, dans le cadre de la construction et du démantèlement d'ouvrages complexes ainsi que pour la surveillance de l'environnement et les sciences de la terre ;
- L'alerte des autorités, mission opérationnelle assurée 24h sur 24, 365 jours par an, en cas d'essai nucléaire, de séisme en France ou à l'étranger, et de tsunami dans la zone Euro-méditerranéenne. La DIF fournit aux autorités les analyses et synthèses techniques associées.

Depuis 2003, le centre DAM - Île-de-France héberge le complexe de calcul scientifique du CEA, qui regroupe l'ensemble des supercalculateurs du CEA, comprenant :

– Le supercalculateur TERA-100 pour les besoins Défense, premier calculateur européen à dépasser la barre du petaflops, c'est à dire capable d'effectuer un million de milliards d'opérations par seconde ;
– Les ordinateurs du Centre de Calculs pour la Recherche et la Technologie (CCRT), ouverts à la communauté civile de la recherche et de l'industrie, pour une puissance globale de 500 teraflops ;
– Le supercalculateur Curie, d'une puissance de 2 petaflops, deuxième élément d'un réseau de supercalculateurs de classe pétaflopique destiné aux chercheurs de la communauté scientifique européenne. Ce supercalculateur est hébergé au TGCC (Très Grand Centre de Calcul) et exploité par les équipes du CEA, qui apporte ainsi sa contribution à la participation de la France au projet PRACE (Partnership for Advanced Computing in Europe), dans le cadre de la recherche européenne.

1.2 Présentation du stage

Ce stage s'est déroulé dans l'équipe responsable de la sécurité du système d'information. Elle est notamment chargée d'assurer la sécurité et la surveillance des infrastructures de calcul haute performance (HPC) du CEA.

1.2.1 Contexte

Un super calculateur a pour but d'exécuter le plus rapidement possible des programmes souvant très gourmants en ressources. Les contraintes économiques et physiques ne permettent pas de construire un seul ordinateur avec une puissance de calcul importante. Il faut donc regrouper un ensemble d'ordinateurs de puissance «inférieure» nommés nœuds, que l'on va relier par des connexions réseau à très haut débit (par exemple en utilisant des cartes *InfiniBand*). Les tâches à exécuter seront alors réparties sur ces nœuds grâce à un ordonnanceur. La figure 1.3 détaille l'organisation d'un super calculateur.

On distingue généralement plusieurs types de nœuds :
– Les nœuds de calcul : ils constituent la majeure partie des nœuds d'un super calculateur. Ils sont responsables de l'exécution des codes de calcul. Les utilisateurs ne peuvent pas s'y connecter directement ;
– Les nœuds d'entrée/sortie : ils assurent la très haute disponibilité et la rapidité de transfert et de stockage des résultats intermédiaires et finaux des codes de calcul ;
– Les nœuds de *login* : ils permettent aux utilisateurs de se connecter au super calculateur et d'y déposer leurs données et programmes, pour pouvoir ensuite lancer une tâche de calcul.

L'exploitation de la puissance de calcul d'un super calculateur nécessite donc de nombreux services annexes. Il faut notamment s'assurer que les infrastructures réseaux sont correctement dimensionnées, que les tâches lancées par les utilisateurs sont bien réparties sur les différents nœuds, etc.

La sécurité d'une telle architecture est une tâche complexe qui nécessite impérativement des compromis. Il faut en effet assurer plusieurs objectifs de sécurité :
– La disponibilité : le maximum de nœuds doit être disponible à tout moment de façon à pouvoir optimiser la répartition des codes de calcul. Les ressources doivent être correctement réparties et les limites imposées aux utilisateurs respectées pour ne pas impacter le fonctionnement des autres utilisateurs ;
– La confidentialité : un utilisateur dispose uniquement des droits pour accéder aux informations le concernant. Il ne doit pas pouvoir lire les résultats de calcul d'un autre utilisateur par exemple ;
– L'intégrité : les résultats de calcul doivent être stockés de façon sûre. Le système doit être fiable et de confiance pour pouvoir obtenir des résultats cohérents.

Enfin, il ne faut pas oublier que la performance reste la raison d'être d'un super calculateur, et que tous les moyens sont mis en oeuvre pour la garder élevée.

1.2.2 Objectifs

Pour répondre à ces contraintes, il a été décidé de déployer une méthode de contrôle d'accès obligatoire sur l'infrastructure HPC. Le système utilisé étant basé sur la distribution Red Hat Enterprise Linux, le choix s'est porté sur SELinux. En effet, cette fonctionnalité est activée par défaut dans le noyau de base et est supportée officiellement par Red Hat.

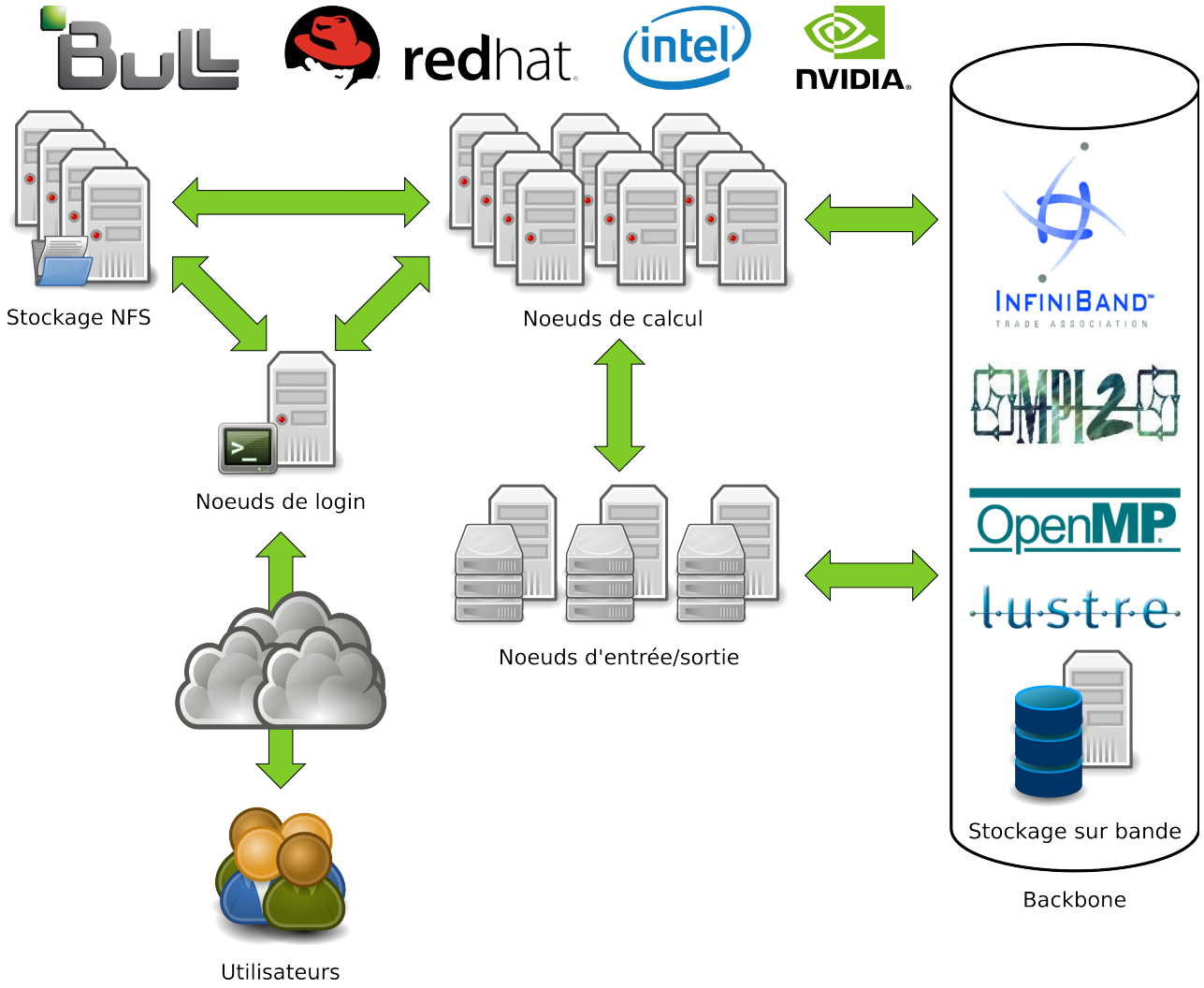


Figure 1.3 – Organisation d'un super calculateur

Cette solution est un premier pas vers un système Linux avec des services et utilisateurs confinés, qui assurerait qu'un utilisateur ne pourrait pas influencer le comportement des nœuds de calcul à son avantage ou ne pourrait pas altérer le système déployé sur ces mêmes nœuds et ainsi fausser les résultats.

1.2.3 Plan du rapport

Nous allons tout d'abord présenter l'état de l'art en terme de solutions implémentant un contrôle d'accès obligatoire sous Linux puis nous détaillerons l'organisation et l'architecture de SELinux. Les différentes méthodes permettant de tracer l'exécution d'un programme dans le noyau formeront notre troisième partie qui sera suivie par les tests de performance réalisés ainsi que les diverses solutions et idées proposées pour améliorer ces performances.

Chapitre 2

État de l'art

Cette étude vise à comparer les différentes méthodes disponibles pour assurer la sécurité d'un système exploitation. Nous allons nous concentrer et détailler plus en profondeur les solutions disponibles sous Linux. D'autres systèmes d'exploitation sont mentionnés mais ne font pas partie du cadre de cette étude.

2.1 Les différentes méthodes de contrôle d'accès

2.1.1 Réflexions générales

Les éléments constituant un système peuvent être répartis dans deux catégories suivant si ils agissent (les processus) que l'on nommera sujets ou si ils subissent (fichier, socket, etc.) que l'on nommera objets. Un sujet peut aussi agir sur un autre sujet, donc l'ensemble des sujets est compris dans celui des objets.

2.1.1.1 Propriétés ou objectifs de sécurité

Il est possible de distinguer quatre principaux objectifs de sécurité définis dans les articles [8] et les processus de validation gouvernementaux [28]. Qualifier un système d'exploitation de sécurisé ou sécurisable reviendrait ainsi à vérifier qu'il est en mesure d'imposer ces objectifs. On distingue ainsi :

La confidentialité : les informations ne doivent être disponibles qu'à ceux qui en ont besoin et qui disposent des bonnes autorisations. Exemple : un utilisateur ne doit pas avoir accès aux fichiers ou mots de passe d'un autre utilisateur. Les informations classifiées ne doivent pas être disponibles pour les utilisateurs non habilités.

L'intégrité : un système doit rester dans un état cohérent. Exemple : les informations telles que les mots de passe, les programmes installés et les fichiers de configuration ne doivent pas être modifiables par des utilisateurs non privilégiés.

La disponibilité : un système doit rester stable, utilisable et réactif. Exemple : les services attendus d'un système doivent être rendus. Les contrôles supplémentaires liés à la sécurité ne doivent pas limiter les fonctionnalités légitimes du système d'exploitation (faux positifs).

La traçabilité : Seuls les utilisateurs disposant d'une autorisation peuvent utiliser un système (authentification). Les actions qu'ils réalisent sont tracées (imputation), et ne peuvent pas être niées *a posteriori* (non-répudiation). Exemple : on doit disposer d'un journal des accès au système et son intégrité doit être assurée.

Nous allons ainsi nous intéresser principalement à la troisième propriété puisque le sujet de ce stage concerne la performance d'une solution de sécurité. Nous ne mettons pas pour autant de côté les autres propriétés qui restent l'attrait principal d'un système sécurisable.

2.1.1.2 Le principe du moindre privilège

Il peut être décomposé en deux parties :

– **La séparation des privilèges** : s'assurer qu'un programme est divisé en autant de sous programmes ou entités n'ayant besoin que d'un nombre limité de ressources et de privilèges pour fonctionner.

– **Le confinement** : restreindre au maximum les droits accordés à chacun de ces sous programmes pour ne laisser que ce qui est nécessaire à leur bon fonctionnement.

Les principaux exemples de réussite dans l'application de ce principe sont `Postfix` et `qmail`, des services de gestion de mail. Ils sont divisés en plusieurs sous-programmes qui communiquent entre eux par l'intermédiaire de fichiers ou d'*IPC SystemV*. Seuls certains d'entre eux ont accès à une *socket* réseau et leur fonctionnalités sont alors restreintes au minimum. Les programmes effectuant les tâches les plus complexes disposent du moins de droits possible.

Ce compartimentage permet d'auditer avec plus de facilité les fonctions critiques et limite l'impact d'une vulnérabilité à un programme ne disposant que de peu de droits. Dans la plupart des cas, il est nécessaire de trouver plusieurs vulnérabilités pour pouvoir obtenir suffisamment de privilèges pour affecter le système.

En règle générale, un programme ne devrait pas avoir accès aux ressources dont il n'a pas besoin, même si cette ressource n'est pas confidentielle par exemple. Il faut noter que peu de programmes sont construits et pensés de cette façon. De même, très peu de systèmes d'exploitation récents imposent ce type de contraintes, puisque les programmes lancés par l'utilisateur ont souvent accès à tous les fichiers d'un utilisateur alors qu'ils n'en ont généralement pas besoin.

Une autre possibilité consiste à utiliser des conteneurs ou *sandbox* qui séparent un ou plusieurs programmes du reste du système (par exemple : `chroot`, `vserver`, `jail BSD`, `LXC`). Les communications ne sont alors plus possibles entre les programmes confinés et le système hôte, ce qui limite les possibilités de ces outils.

2.1.1.3 A qui dois-je faire confiance ?

Quel que soit le modèle de sécurité que l'on souhaite appliquer, il est nécessaire de faire confiance à un ou plusieurs composants (matériels, logiciels ou les deux). Dans le meilleur des cas, tous les composants d'un système ont été vérifiés et sont considérés comme sûrs, comme par exemple pour le système d'exploitation QNX [33]. En revanche, les coûts induits par les nombreux tests et vérifications effectués sont peut-être prohibitifs. Le nombre de logiciel dit «de confiance» est donc très restreint. De plus, il est toujours possible qu'une erreur passe inaperçue lors des vérifications.

Comme il n'est pas envisageable de faire confiance à tous les logiciels sur un système, il est nécessaire d'imposer des contraintes par un autre moyen, qui peut être matériel. Il faudra alors s'assurer du bon fonctionnement de cet élément ce qui peut dans ce cas aussi représenter une difficulté importante.

Certains processeurs Intel disposent de la fonctionnalité *Trusted Execution Technology* (Intel TXT, figure 2.1) [25]. Elle vérifie au démarrage que les éléments constituants le cœur du système (le noyau, le *bootloader*) n'ont pas été modifiés par une personne ne disposant pas d'un secret. C'est un exemple de protection matérielle contre des altérations logicielles.

Plus généralement, le principe de la chaîne de confiance peut être appliqué à l'aide de fonctionnalités matérielles puis logicielles. Chaque élément va vérifier la signature du programme suivant dans la chaîne de démarrage avant de l'exécuter. C'est une des possibilités offertes par le remplaçant du BIOS : UEFI [40].

2.1.2 Les différents modèles de contrôle d'accès

Pour essayer d'atteindre ces objectifs de sécurité, plusieurs modèles de contrôle d'accès ont été créés. Ils sont détaillés dans cette section.

2.1.2.1 Contrôle d'accès discrétionnaire (DAC)

Le contrôle d'accès discrétionnaire laisse l'utilisateur propriétaire d'une ressource choisir les permissions liées à celle-ci. La plupart du temps, le compte d'un utilisateur ou d'un démon sera utilisé par un attaquant, qui aura alors la possibilité d'utiliser ces ressources à des fins malveillantes.

De plus, toutes les applications lancées par un utilisateur héritent des mêmes permissions et aucune séparation n'est faite entre un navigateur et un traitement de texte par exemple. `Firefox` peut aussi lire tous les fichiers présents dans le dossier d'un utilisateur, ce qui inclut par exemple les clés privées `ssh`.

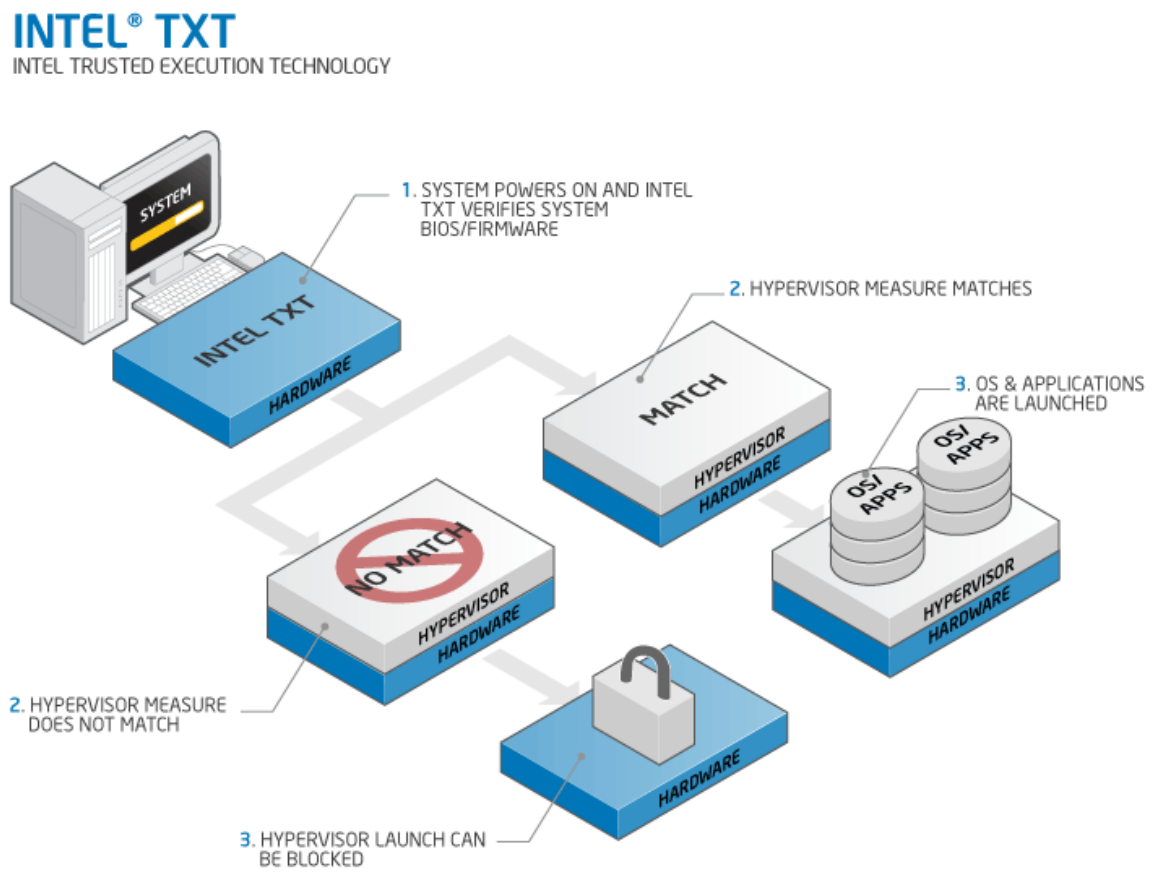


Figure 2.1 – Intel Trusted Execution Technology (Source Intel : [25])

Ce modèle inclut un compte administrateur qui dispose de tous les droits. Si une faille est trouvée dans une application exécutée avec cet utilisateur, c'est l'intégralité de la machine qui est compromise.

Le modèle HRU [24] nous permet d'affirmer qu'il n'est pas possible d'imposer des propriétés de sécurité sur un système reposant uniquement sur du contrôle d'accès discrétionnaire.

2.1.2.2 Controle d'accès obligatoire (MAC)

Les différents problèmes du contrôle d'accès discrétionnaire ont conduit à la création d'un nouveau modèle qui introduit le concept de moniteur de référence, tel que décrit dans [2]. Cette entité, logicielle ou matérielle, prend les décisions relatives au contrôle d'accès. Elle est généralement chargée d'appliquer une politique de sécurité. Il faut donc pouvoir la considérer comme fiable et s'assurer qu'elle prend bien part à l'ensemble des décisions relatives au contrôle d'accès.

Pour appliquer le principe du moindre privilège, tous les accès aux ressources d'un système sont interdits par défaut [10]. Il faut alors construire une politique de sécurité, qui déterminera quelles seront les opérations légitimes.

2.1.2.3 Sécurité basée sur les capacités

Dans ce modèle, les sujets reçoivent un ensemble de permissions autorisant l'accès à des ressources critiques. Les processus ne peuvent transmettre que les permissions qu'ils possèdent à leurs processus fils.

En pratique, ces restrictions tendraient à faire perdre à tous les processus leurs autorisations. Une nouvelle opération est ainsi ajoutée, permettant à certains processus d'échanger ou de produire certaines permissions ou capacités, souvent après vérification d'un secret. Seul un petit nombre de systèmes d'exploitation implémente un tel modèle de sécurité [41].

2.1.2.4 Sécurité multi-niveau et multi-catégories (MLS/MCS)

Ces deux modèles de sécurité sont généralement utilisés pour assurer l'intégrité ou la confidentialité. On définit préalablement un certain nombre de niveaux de sécurité et catégories, qui correspondent respectivement à des niveaux de confidentialité (public, confidentiel, secret, très secret) ou à des domaines que l'on souhaite distinguer. On peut alors répartir les éléments d'un système (sujets et objets) dans ces niveaux pour contrôler les interactions entre ces ensembles. La figure 2.2 résume le fonctionnement de ces modèles.

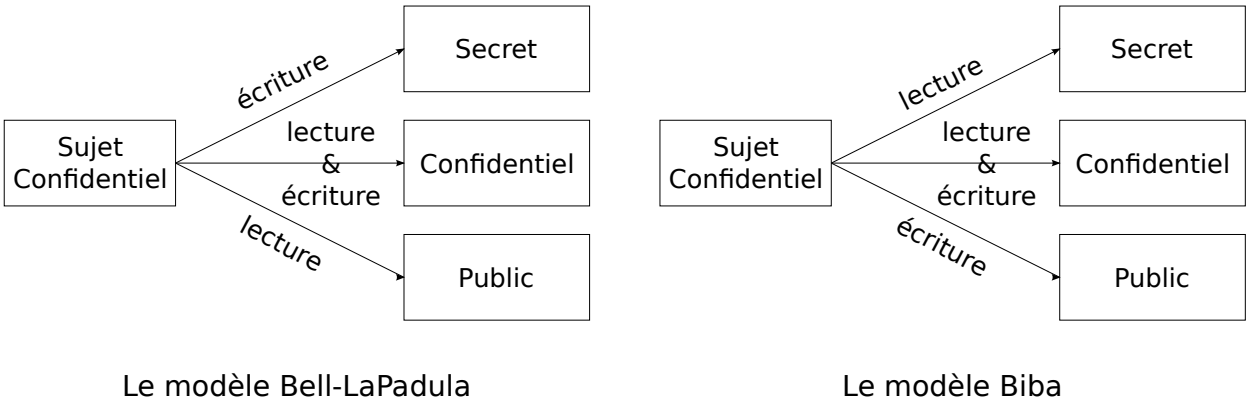


Figure 2.2 – Modèles de sécurité multi-niveau

Bell-LaPadula : [4] décrit un modèle qui préserve le niveau de confidentialité des informations. Un utilisateur ne peut écrire de l'information que dans un niveau supérieur ou égal au sien, et ne peut lire de l'information que d'un niveau inférieur ou égal au sien.

Biba : [5] détaille au contraire un modèle assurant l'intégrité de l'information. Les modifications ne peuvent être effectuées que sur des niveaux inférieurs ou égaux, et les lectures sur des niveaux supérieurs ou égaux.

Ces modèles de sécurité forment une implémentation simple qui ne peut pas imposer plus d'une contrainte de sécurité à la fois. D'autres modèles plus complets ont été établis pour gérer simultanément plusieurs propriétés de sécurité. D'autre part, il faut prendre en compte les problèmes de sur-classification de l'information pour le modèle Bell-LaPadula et de perte d'intégrité pour le modèle Biba. En effet, au fil du temps, les données ne peuvent que devenir plus classifiées (respectivement moins intègres) et il apparaît donc nécessaire de prévoir une opération de déclassification (ou de vérification), effectuée par un tiers de confiance. C'est une opération risquée puisque pouvant compromettre l'intégralité du modèle.

2.1.2.5 Contrôle d'accès à base de rôles (RBAC)

Les permissions accordées sur un système ne varient pas ou peu d'un utilisateur à un autre. Ce modèle crée une couche supplémentaire entre les sujets et les objets pour regrouper des permissions dans des rôles que l'on va pouvoir associer à des utilisateurs. Il ne sera alors plus nécessaire de préciser pour chaque utilisateur ses permissions puisqu'il suffira de lui assigner le bon rôle. Par exemple, on pourra donner le rôle "user" à l'ensemble des utilisateurs d'un système pour leur permettre d'accéder à leurs fichiers et lancer des programmes, et l'on pourra ajouter le rôle "webadmin" pour les quelques utilisateurs travaillant sur un site web. Ils pourront alors relancer le serveur web par exemple, sans avoir d'accès complet (`root`) sur le système.

Ce modèle permet aux administrateurs d'ajouter ou de supprimer de nombreuses permissions à un utilisateur en les regroupant en groupes fonctionnels. Cela simplifie donc la gestion d'un système complexe avec de nombreuses permissions.

2.1.2.6 Contrôle des interactions entre types (Type Enforcement)

Ce modèle s'appuie sur le concept de type. Un type est un jeton (chaîne de caractères, etc.) qui caractérise chaque sujet et objet d'un système. L'accès à un objet par un sujet peut alors être considéré comme une interaction entre deux types. Le contrôle peut être effectué en définissant l'intégralité des interactions autorisées entre les types.

2.1.2.7 Exécution limitée aux binaires de confiance (Trusted Path Execution : TPE)

Cette protection limite les programmes exécutables à ceux situés dans un ensemble de répertoires dont le contenu n'est modifiable que par un administrateur. Un utilisateur ne peux pas exécuter ses propres programmes ni modifier les programmes exécutables. Les seuls binaires exécutables sur ce système sont alors des fichiers que l'on peut considérer comme étant de confiance puisqu'ils sont situés aux emplacements d'un système de fichiers modifiables uniquement par un administrateur.

2.2 Implémentations de méthodes de contrôle d'accès obligatoire

Cette section détaille les différentes implémentations de contrôle d'accès obligatoire dans les systèmes d'exploitation modernes. Les cinq premières implémentations sont basées sur un concept propre à Linux : les modules de sécurité Linux [31]. Ils permettent d'ajouter des contrôles par l'intermédiaire de crochets (*hooks*) pour chaque appel système du noyau. Ces modules et l'utilisation des crochets sont critiqués par [20].

Une comparaison similaire est disponible page 86 de [7] et à l'adresse [29].

2.2.1 Simplified Mandatory Access Control Kernel (SMACK)

Le modèle SMACK (présenté dans [35, 36]) a été spécialement pensé pour être simple à configurer et à administrer. Il est basé sur des contextes de sécurité donnés à tous les sujets et objets d'un système. Le contrôle d'accès est appliqué en utilisant les contextes comme stockage explicite des règles de sécurité. Par exemple, un caractère spécial indique que seule l'écriture ou la lecture est autorisée.

Aucun support particulier dans les applications en espace utilisateur n'est requis pour que le contrôle s'effectue. L'auteur propose en revanche des scripts permettant de simplifier la mise en place et la gestion de SMACK.

Contraintes pour l'administrateur :
- La politique et les contextes de sécurité sont répartis sur l'ensemble du système et ne permettent pas de gestion centralisée ;
- Les contextes de sécurité doivent être fixés avec soin pour tous les éléments que l'on souhaite contrôler.

Avantages :
- La mise en place et l'administration sont facilités puisqu'il n'est pas nécessaire de modifier les logiciels.

2.2.2 TOMOYO Linux

TOMOYO Linux (présenté dans [22, 23]) est un projet basé sur des outils pour construire des politique de sécurité à partir de tests et d'essais successifs. Le but est de générer une politique intuitive, basée sur le contrôle d'accès à partir des chemins d'accès et de l'historique des processus. TOMOYO peut être utilisé comme outils d'audit d'applications ou encore comme outils de confinement des sessions utilisateur distantes. Un mode d'administration utilisant l'organisation en rôles (RBAC) est disponible. Une fonctionnalité permet d'appliquer des règles de pare-feu à des applications individuellement.

Malheureusement, la version courante intégrée au noyau Linux est encore incomplète [39]. De plus, les politiques TOMOYO comprennent généralement des chaînes de caractère très longues, à comparer, ce qui peut impacter les performances.

Contraintes pour l'administrateur :
- Aucune politique n'est fournie ;
- Nécessite des modifications manuelles pour être efficace.

Avantages :
- Un mode apprentissage est disponible ainsi que des outils d'audit ;
- La mise en place est progressive et la politique modifiable en cours de route.

2.2.3 Application Armor (AppArmor)

Le module de sécurité AppArmor [15] fournit une méthode de contrôle d'accès centrée sur les applications. Il applique des politiques qui définissent les accès autorisés pour chaque application individuellement en se basant sur les chemins d'accès. Conçu pour être plus simple à utiliser que SELinux, il propose un mode apprentissage. Il est ainsi à rapprocher du fonctionnement de grsecurity. John Johansen détaille en [1] les fonctionnalités de AppArmor, les différences avec SELinux, et ajoute une note sur les performances, jugées inférieures à SELinux.

Contraintes pour l'administrateur :
- Pas de politique globale, nécessite une politique pour chaque programme que l'on veut confiner. Il est tout de même possible de confiner l'ensemble du système.

Avantages :
- Certaines distributions fournissent des politiques pour les logiciels les plus utilisés ;
- Il est plus facile de confiner un ensemble restreint d'applications sans impacter globalement le système.

2.2.4 Security Enhanced Linux (SELinux)

SELinux est inclus dans le noyau Linux depuis la version 2.6 sous la forme d'un module de sécurité [42]. Développé par la *National Security Agency* (NSA), SELinux implémente un contrôle d'accès basé sur les types (TE), la gestion des rôles (RBAC), et la séparation des niveaux de sécurité et catégories (MLS/MCS). Un label doit être attribué à chaque ressource du système pour pouvoir l'intégrer dans la politique. Un moniteur de référence dans le noyau Linux se charge de l'application de la politique de sécurité. Les décisions sont prises indépendamment les unes des autres, ce qui implique que SELinux est sans état et ne peut donc pas prévenir l'utilisation de canaux cachés ou d'interactions indirectes.

Contraintes pour l'administrateur :
- L'ensemble des interactions à autoriser doit être détaillé dans le langage spécifique à la politique SELinux pour que le système fonctionne ;
- Les systèmes de fichiers doivent inclure le support des attributs étendus (support pour NFS en cours de réalisation) ;
- Difficulté d'apprentissage non négligeable liée à la granularité des accès contrôlés.

Avantages :
- Une politique de référence est proposée, et plusieurs dérivées sont disponibles ;
- Confinement à «grain-fin» des sujets s'exécutant sur un système ;
- Disponible dans le noyau Linux et testé sur les distributions «entreprise» (Red Hat Enterprise Linux, etc.).

2.2.5 Policy Interaction Graph Analysis - Mandatory Access Control (PIGA-MAC)

PIGA est un ensemble d'outils développés au sein de l'équipe SDS du LIFO (Laboratoire d'Informatique Fondamental d'Orléans) [6, 8, 9]. La partie chargée du contrôle d'accès (PIGA-MAC) s'appuie sur SELinux. Les politiques de sécurité sont exprimées dans un langage de description d'activité et un compilateur produit l'ensemble des suites d'interactions violant ces propriétés dans une politique SELinux. Cette implémentation empêche l'exploitation de canaux cachés, et peut être généralisée comme une liste noire d'interactions, qui ajoute un état au contrôle de SELinux.

Contraintes pour l'administrateur :
- Toutes les contraintes de SELinux ;
- Langage de description d'activités spécifique à PIGA ;
- La génération de politique est une opération coûteuse en temps, mais ne nécessite pas de calcul en cours d'exécution et peut être réalisée avant le passage en production.

Avantages :
- Tous les avantages de SELinux ;
- Limite l'exploitation de canaux cachés ;
- Possibilité d'audit automatisé des politiques SELinux.

2.2.6 grsecurity

grsecurity [19] est un patch pour le noyau Linux qui implémente diverses méthodes de contrôle d'accès mandataire (TPE, RBAC). Il est associé au patch PaX qui ajoute des protections au noyau Linux et aux programmes en espace utilisateur, mais ne vise pas à contrôler le comportement des programmes en espace utilisateur. Il introduit de nombreuses limitations sur l'utilisation de la mémoire et sur les interactions des utilisateurs avec le noyau.

Contraintes pour l'administrateur :
- La politique doit être générée et vérifiée avec soin pour assurer la restriction des programmes exécutés ;
- Certains programmes ne fonctionne pas avec les restrictions imposées par la partie PaX du patch.

Avantages :
- Un outil facilite la création d'une politique grâce à l'apprentissage interactif. Politique basée sur les chemins d'accès ;
- Placement réellement aléatoire des sections de données et de code pour les processus et le noyau. Pile non-exécutable (améliorations liées à PaX) ;

2.2.7 TrustedBSD

Le projet TrustedBSD [38] consiste en une série de modifications ajoutant d'autres capacités d'audit, les listes de contrôle d'accès (ACL) et plusieurs mécanismes de MAC au système d'exploitation FreeBSD. Une fonctionnalité similaire à celles implémentées dans PaX a aussi été ajoutée, W^X, pour appliquer des séparations entre les pages mémoire exécutables et celles accessibles en écriture.

2.3 Synthèse

L'impact sur les performances est peu étudié dans le cas de solutions de sécurité. Il l'est d'autant moins que beaucoup de ces solutions sont encore jeunes, ou très peu utilisées. SELinux apparaît ici comme un choix raisonnable pour plusieurs raisons :
- Intégré et supporté dans la distribution utilisé sur les super calculateurs (Red Hat Enterprise Linux) ;
- Solution testée et considérée comme fiable ;
- Impact sur les performances à priori limité.

Nous allons ainsi détailler les principaux composants de l'architecture de SELinux qui peuvent avoir un impact sur les performances.

Chapitre 3

Architecture et fonctionnement de SELinux

Pour optimiser les performances d'un outil, il est nécessaire de bien connaître son fonctionnement. Une grande partie du stage a donc été consacrée à la compréhension et à l'utilisation courante de SELinux.

Un ordinateur sans restrictions d'accès a été mis à ma disposition pour pouvoir expérimenter, modifier la politique et la configuration de SELinux. Pour étudier un environnement proche de celui utilisé sur les clusters ouverts du CEA, nous avons choisi d'utiliser la distribution CentOS dans sa version 6.2 et de comparer les problèmes rencontrés avec les versions plus récentes de SELinux dans Fedora 16 et 17.

Plusieurs éléments peu mis en avant, incomplets ou n'étant plus à jour ont été découverts, notamment le wiki officiel de SELinux [32] qui ne reflète pas les modifications introduites dans les dernières versions de Fedora.

Cette section détaille tout d'abord l'organisation globale de SELinux puis précise certains éléments peu documentés de l'architecture de SELinux.

3.1 Contextes de sécurité et contrôle basé sur les types

SELinux repose sur une politique contrôlant les interactions entre les sujets et les objets d'un système. Le contrôle d'accès effectué par SELinux s'appuie sur des contextes de sécurité, des chaînes de caractères constituées de plusieurs éléments :

- Un utilisateur : il est distinct de l'utilisateur *NIX standard. On associe à chaque utilisateur *NIX un utilisateur SELinux. Il détermine les rôles SELinux accessibles ;
- Un rôle : chaque utilisateur SELinux est associé à plusieurs rôles qui correspondent à un ensemble de permissions nécessaires pour effectuer une tâche : par exemple, administrer un système ;
- Un type : il regroupe des objets ou sujets qui disposent des mêmes permissions dans la politique SELinux ;
- Un ou plusieurs niveaux de sécurité : il fait la distinction entre les éléments publics, confidentiels ou secrets ;
- Une ou plusieurs catégories : elles ajoutent dynamiquement des cloisonnements entre sujets de même type et de même niveau de sécurité.

Table 3.1 – Exemples d'éléments de contextes de sécurité SELinux

Utilisateur	Rôle	Type	Niveau de sécurité	Categorie
`user_u`	`user_r`	`user_t`	`s0`	∅
`tim_u`	`sysadm_r`	`firefox_t`	`s42`	`c42`
`system_u`	`system_r`	`syslog_t`	`s0-100`	`c12-1023`
`guest_u`	`object_r`	`etc_t`	`s0,s4`	`c0,c4`

Par exemple, le contexte de sécurité associé au processus `Firefox`, pourra être : `tim_u:user_r:firefox_t:s0.c42`. Ceci signifie que :

- un utilisateur *NIX a été associé à l'utilisateur SELinux `tim_u` ;
- il a accès au rôle `user_r` ;
- il a pu exécuter le programme `Firefox` (contexte `firefox_t` du processus associé) ;
- il est dans le plus bas niveau de sécurité : `s0` ;

– il appartient à la catégorie `42`.

Tous les éléments du système (fichiers, processus, sockets, etc.) ont un contexte de sécurité qui leur est associé. Le noyau est chargé de prendre la décision d'autoriser ou non l'accès à un objet par un sujet ou d'une interaction entre deux sujets.

3.2 Intégration dans le noyau Linux

SELinux est intégré au noyau Linux en tant que module de sécurité (LSM) qui utilise des crochets (`hooks`) pour insérer des fonctions lors du déroulement des appels système. Ces fonctions sont regroupées dans la structure `struct security_operations selinux_ops`. Par défaut, le noyau utilise un pointeur vers une `struct security_operations` ne contenant que des fonctions vides. La valeur de ce pointeur sera remplacée par l'adresse de la structure `struct security_operations` contenant l'ensemble des fonctions de contrôle de SELinux. La figure 3.1 résume les contrôles qui seront effectués par le noyau lors de l'appel système `open`.

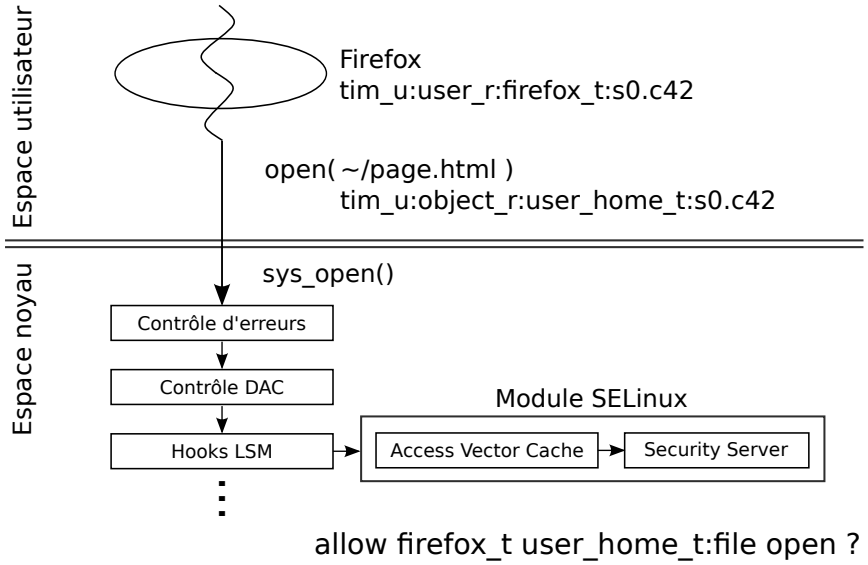


Figure 3.1 – Exemple d'exécution dans d'un appel système

Une fois les vérifications générales et le contrôle d'accès discrétionnaire effectués, le noyau va appeler la fonction correspondant à l'appel système dans la structure `security_ops`.

Chaque appel système correspond à une interaction représentée par un contexte de sécurité source, effectuant une opération sur un contexte de sécurité cible. La première étape dans le module SELinux consistera donc à chercher les contextes de sécurité associés aux éléments source et cible de l'interaction à contrôler. Ces contextes de sécurité sont stockés comme des entiers non signés de 32 bits nommés Security ID (SID), dans les attributs étendus pour les fichiers, dans la structure `task_struct->cred->security` pour les processus et dans la structure `sock->sk_security` pour les sockets.

Une fois les SID obtenus, le noyau cherche si l'interaction (ou *Access Vector*) est présente dans un cache spécifique (l'*Access Vector Cache* ou *AVC*). Ce cache est formé d'un tableau de taille fixe de 512 entrées, contenant des pointeurs vers des listes chaînées. Celles-ci sont constituées de nœuds contenant l'ensemble des informations sur une interaction et sur la décision à prendre. Une fonction de hashage propre à SELinux répartit simplement ces nœuds dans les listes chaînées. Une politique SELinux comprend généralement entre 100 000 et 300 000 interactions différentes donc ce cache n'est pas de taille suffisante pour contenir l'ensemble des cas possibles.

Si l'interaction n'est pas dans l'*AVC*, le noyau se tourne alors vers le *Security Server* qui contient l'intégralité de la politique SELinux. Il vérifie alors que chaque contexte est valide, puis cherche dans la politique la règle correspondant à l'accès réalisé. Enfin, il applique les règles liées aux niveaux de sécurité et aux catégories. Une fois la réponse obtenue, l'information est stockée dans l'*AVC*, entraînant la suppression d'un nombre fixe d'entrées si le nombre maximal de nœuds a été atteint. Le *Security Server* représente le moniteur de référence chargé de toutes les décisions de contrôle d'accès.

3.3 Premières remarques concernant les performances

L'*AVC* est initialisé au démarrage du système et se rempli progressivement au fil du temps. Il est donc très probable que lors du premier lancement d'un programme, les interactions relatives à ce programme ne soient pas présentes dans le cache. Cela induit généralement une latence supplémentaire au premier démarrage. De plus, la taille du cache est fixée à 512 entrées par défaut, et bien qu'il soit possible d'augmenter le nombre total de nœuds stockés dans les listes (`cache_threshold`), cette opération a pour effet indésirable d'augmenter la taille de ces listes et donc de réduire, à partir d'un certain seuil, les performances du cache.

3.4 Démarrage et activation de SELinux

La figure 3.2 présente schématiquement le démarrage d'un système GNU/Linux.

Figure 3.2 – Démarrage d'un système GNU/Linux

La documentation officielle de SELinux décrit le processus de démarrage avec les étapes concernant SELinux [21, 32] mais certain éléments ont évolué depuis la dernière mise à jour (13 septembre 2010). Voici donc une version plus à jour du processus pour la distribution CentOS dans sa version 6 :

1. Démarrage de la machine, initialisation du matériel par le BIOS ;
2. Le BIOS exécute le chargeur de démarrage (*bootloader*, GRUB) ;
3. GRUB récupère les informations nécessaires dans la partition de démarrage, charge et exécute le noyau Linux.

Deux cas de figure se présentent alors puisque que le démarrage peut se poursuivre avec ou sans l'utilisation d'un *initramfs*. L'*initramfs* est une image *cpio* compressée avec *gzip* contenant un ensemble de binaires et de scripts qui effectuent les opérations préliminaires au montage de la partition `root`. C'est l'utilitaire *dracut* qui réalise ces images pour la distribution CentOS. Si aucun fichier *initramfs* n'est précisé au démarrage, le noyau se charge de monter la partition `root` passée en paramètre. Dans le cas de CentOS 6, c'est un élément de l'*initramfs* qui va charger la politique SELinux :

4. Le noyau charge et extrait l'*initramfs* en mémoire ;
5. Le noyau initialise le *framework* LSM, qui appelle la fonction `security_init()` puis `selinux_init()` ;
6. Le noyau applique les contextes de sécurité en fonction du contenu du fichier `flask.h` ;
7. L'*Access Vector Cache* est initialisé (`avc_init()`) ;
8. Le pseudo système de fichiers `/selinux` est initialisé ainsi que les contextes de sécurité des sockets netlink ;
9. Le noyau exécute le script `init`, présent à la racine de l'*initramfs*, qui lance l'interpréteur de commande shell compatible POSIX `dash` ;
10. Ce script charge les modules nécessaires au démarrage des différents périphériques du système, monte la partition `root` en lecture seule puis exécute le programme `load_policy`. Ce programme vérifie qu'une politique binaire est bien présente, monte le pseudo système de fichiers `/selinux`, charge la politique SELinux en utilisant la fonction `selinux_init_load_policy()` de la `libselinux` ;
11. Une fois la politique chargée, l'initialisation de SELinux est terminée. SELinux va donc pouvoir appliquer le contrôle sur le système en fonction de la configuration définie dans `/etc/selinux/config` ;
12. Le noyau a fini de démarrer, l'*initramfs* est démonté et `/sbin/init` est lancé (ce binaire correspond en réalité à `Upstart` pour CentOS 6) ;
13. `Upstart` est un remplacement du classique SystemV init. Il fonctionne en réponse à des évènements. Le premier événement (`startup`) est émis par `Upstart` lui-même. Celui-ci entraine l'exécution de `/etc/rc.d/rc.sysinit` qui reprend la procédure de démarrage classique d'un système GNU/Linux et qui :
    (a) monte les pseudo systèmes de fichiers `/proc` et `sysfs` ;

(b) vérifie que `/selinux` est accessible et que le processus `init/upstart` est exécuté sous un contexte SELinux valide (différent de kernel_t) ;

(c) corrige, si nécessaire, les contextes SELinux des éléments dans `/dev` ;

(d) termine le processus `dash` s'il est encore en cours d'exécution ;

(e) corrige, si nécessaire, le contexte de `/dev/pts` ;

(f) monte les éléments listés dans `/etc/rwtab` et `/etc/statetab` et vérifie leur contexte SELinux ;

(g) vérifie s'il est nécessaire de relabelliser entièrement le système de fichiers (fichier `/.autorelabel`). Si `/sbin/init` a changé de contexte SELinux après cette étape, il est nécessaire de redémarrer ;

(h) corrige encore d'autres contextes de sécurité.

Il faut noter que l'introduction de `systemd` dans les dernières versions de la distribution Fedora a modifié l'organisation du démarrage du système. Cette procédure devra donc être révisée pour les versions Red Hat Enterprise Linux/CentOS 7.

3.5 Chargement et déchargement d'un module SELinux

La politique SELinux de référence est divisée en un certain nombre de «modules», qui regroupent des règles d'autorisation généralement liées à un programme ou à un utilisateur. Les modules sont ensuite regroupés pour former une politique monolithique comme décrit dans [11].

La politique est ainsi stockée sous forme monolithique dans le noyau : un seul *Security Server*, une seule politique. Il est donc nécessaire de recharger intégralement la politique pour désactiver ou activer un module SELinux, ce qui nécessite notamment l'invalidation de l'*AVC*. Il est délicat de désactiver le module SELinux qui correspond à un service en cours d'exécution. Ainsi cette modularité comporte certaines limitations et présente des difficultés en terme de stabilité et de sécurité.

Il n'est donc pas recommandé de développer ou de modifier de façon répétée la politique SELinux sur des systèmes en production. Seul le chargement au démarrage de la machine peut assurer le bon fonctionnement de la politique.

3.6 Modification d'un booléen

Certaines règles de la politique SELinux peuvent être regroupées et conditionnées à l'activation d'un ou plusieurs booléens. Les opérations agissant sur les booléens n'entraînent pas de rechargement de la politique SELinux et posent donc moins de difficultés.

3.7 Remarques sur le support des politiques

La documentation mise à disposition par Red Hat à l'adresse [34] énonce qu'il n'est pas recommandé d'utiliser la politique MLS sur un système avec une interface graphique. Plusieurs difficultés se sont en effet présentées lors de sa mise en place sur CentOS avec un environnement graphique confiné.

L'activation des booléens `xserver_object_manager` et `allow_xserver_execmem` n'est pas suffisante. Les versions récentes du navigateur `Firefox` (10.0.3 ESR) utilisent un compilateur `javascript` *just-in-time* (JIT), qui nécessite l'activation du booléen `allow_execmem`. D'autre part, j'ai créé le module `xorg_fixes.te` (Annexe A) qui corrige la politique pour l'environnement de bureau KDE et OpenOffice.org.

Chapitre 4

Travail sur les performances

Plusieurs études ont été réalisées sur l'impact de SELinux au niveau des performances d'un système. Mais la grande diversité des cas de figure à tester restreint nécessairement l'utilité de ces tests à une configuration particulière.

Lars Strand [37] détaille plusieurs tests liés à une utilisation d'un système en tant que serveur, avec la distribution RHEL 5. L'impact moyen sur des programmes tels que `postfix` ou `Apache httpd` est alors de 6%, variant entre 2% au minimum et 10% au maximum. Ces tests peuvent malheureusement déjà être considérés comme anciens, puisque les améliorations entre les différentes versions de RHEL sont conséquentes.

Michael Larabel, auteur du site Phoronix, a réalisé une série de tests [26, 27] en utilisant la *Phoronix Test Suite*. Il faut noter que la configuration par défaut de SELinux pour la distribution utilisée dans les tests (Fedora) ne confine pas les utilisateurs, qui sont dans le domaine `unconfined_t`. Aucun contrôle particulier n'est donc appliqué sur les interactions se produisant pendant ces tests (tous les accès sont autorisés). Les résultats montrent que l'impact de SELinux est négligeable pour les programmes qui font principalement des calculs et peu d'entrées/sorties (jeux vidéo, compression, encodage audio/vidéo). En revanche, les performances brutes d'`Apache httpd` sont réduites de 10%, et celle d'un serveur de mail de 5%.

Il est donc nécessaire de bien définir le cadre de notre étude pour déterminer l'influence de SELinux sur un système.

4.1 Objectifs des tests de performance

Des projets précédents réalisés au CEA ont mené au déploiement de SELinux sur les clusters de calcul. Les nœuds de *login* sont les plus exposés aux attaques car les utilisateurs disposent d'un accès direct. En revanche, les contraintes en termes de performances sont moindres, puisque ces nœuds n'ont pas pour but d'exécuter des codes de calcul de façon intensive.

Concernant les nœuds de calcul, les performances sont critiques. Le but des divers tests effectués est donc de reproduire le comportement d'un code de calcul pour déterminer quels éléments sont impactés par les contrôles de SELinux et dans quelle mesure.

Cette tâche se révèle plus difficile que prévue puisque qu'il n'existe pas de code de calcul type même s'ils exploitent globalement tous la même méthode de parallélisation et de partage de résultats de calculs : *Message Passing Interface* (*MPI*). De plus, la technologie *Infiniband* est utilisée pour les communications réseau entre les nœuds de calcul, ce qui rend le test de performances réseau plus difficile à effectuer (entre autre à cause de la communication directe avec la carte réseau, sans passer par les couches noyau). D'autre part, le stockage des données partagées de grande taille nécessite des vitesses de lecture/écriture importantes qui sont obtenues avec le système de fichiers distribué Lustre. Il faudra donc reproduire cet environnement pour obtenir des tests probants.

Il est possible de dégager un comportement global commun : la majeure partie du temps d'exécution est constituée de calculs qui ne font pas appel aux fonctions du noyau. Les calculs réalisés étant souvent longs, il est nécessaire d'écrire des résultats intermédiaires à intervalles réguliers pour permettre une récupération rapide en cas d'arrêt imprévu. Il y a donc des accès importants, sur de courtes durées, à des intervalles réguliers aux moyens de stockage. Il faut donc tout particulièrement contrôler les temps d'exécution des appels système liés aux accès au système de fichiers.

Nous avons commencé les tests en utilisant des machines virtuelles sous *VirtualBox*, mais nous nous sommes vite rendu compte que la virtualisation d'un système modifiait son comportement de façon difficilement prévisible. Un ordinateur a donc été mis à ma disposition pour effectuer les divers tests. La configuration est la suivante :

- Processeur : Intel Core i5 650 @ 3,19GHz (4 cœurs) ;
- Carte mère : Dell OXC7MM ;
- Mémoire vive : 4Go ;
- Disque dur : 320Go Western Digital WD3200AAKS-7 ;
- Système d'exploitation : CentOS 6.2 ;
- Noyau Linux : 2.6.32-220.7.1.el6.x86_64 ;
- Système de fichiers : ext4.

4.2 Tests macroscopiques

Le choix du benchmark a été influencé par les possibilités de paramétrage offertes et par la forme des résultats. Il fallait en effet pouvoir lancer un benchmark sur des temps longs, et obtenir des résultats exploitables.

Des politiques SELinux ont été écrites pour permettre le bon fonctionnement des différents benchmarks et outils utilisés. Elles sont disponibles en Annexe A.

4.2.1 Flexible FileSystem Benchmark (FFSB)

FFSB est un benchmark multiplateforme de mesure de performance de système de fichiers. Son comportement est personnalisable à l'aide de fichiers de profil et il peut utiliser plusieurs *threads* simultanément. Le profil d'exemple génère aléatoirement un ensemble de fichiers dans une arborescence, en utilisant 4 *threads*. Nous avons modifié ce comportement pour garder un aspect reproductible au benchmark en limitant le profil à un seul *thread*, sans opérations de lecture ou écriture aléatoires. Le listing 4.1 est un exemple de fichier de profil d'exécution de FFSB.

Listing 4.1– Exemple de profil d'exécution de FFSB

```
# Durée du benchmark en secondes
time    = 10

[filesystem]
  location  = /home/user/ffsbdata
  num_dirs  = 100
  [...]
[end]

[...]

[threadgroup]
  num_threads = 1

  append_weight   = 1
  append_fsync_weight = 1
  stat_weight   = 1
  create_weight   = 1
  create_fsync_weight = 1
  delete_weight   = 1
  readall_weight    = 1
  writeall_weight   = 1
  writeall_fsync_weight = 1
  open_close_weight = 1

  write_size  = 40k
  write_blocksize = 4k
  read_size = 40k
  read_blocksize  = 4k
[end]
```

Les résultats que l'on obtient avec FFSB sont sous la forme listée en 4.2.

Listing 4.2– Résultats obtenus avec FFSB

```
            Min       Avg       Max         Total Calls
            ========  ========  ==========  ===========
[   open]   0.000000  0.520422  470.427002        67359
[   read]   0.000000  0.061750  343.372986      1424291
[  write]   0.000000  0.128264  800.580994      5975229
[ unlink]   0.037000  2.941505  332.927002         8448
[  close]   0.000000  0.052100  192.964996        67359
[   stat]   0.012000  0.096160   33.810001         8501
```

FFSB va donc nous permettre de déterminer globalement l'impact de SELinux sur ces appels système.

4.2.2 Résultats obtenus

Afin d'obtenir des résultats exploitables, nous avons décidé d'utiliser la procédure suivante : FFSB est lancé au démarrage de la machine, sans interface graphique, sur une durée de 10 minutes. On répète ce schéma 20 fois par paramètre que l'on souhaite faire varier :

- l'état de SELinux : *Disabled*, *Permissive*, *Enforcing* ;
- la présence ou non de l'*AVC* ;
- la taille de l'*AVC* ;
- la taille de la politique SELinux.

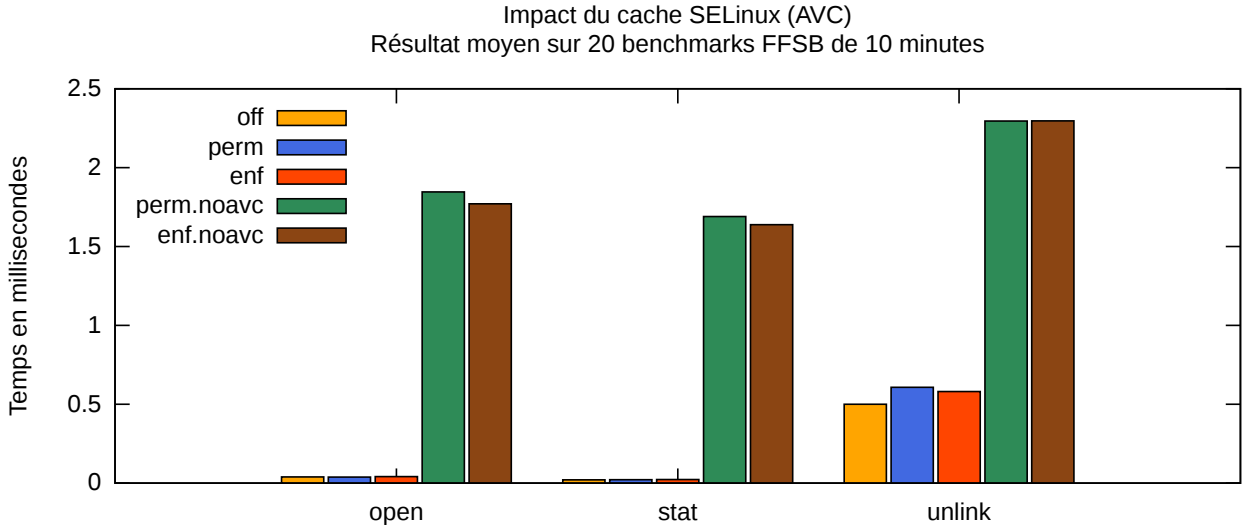


Figure 4.1 – Résultat pour les appels système `open`, `stat` et `unlink`, avec et sans l'*AVC*

La figure 4.1 détaille les résultats obtenus en comparant les temps d'exécution moyens des appels système `open`, `stat` et `unlink` en fonction du mode de SELinux et de la présence ou nom de l'*AVC*. Ceci nous montre principalement que le *Security Server* est lent et qu'il est donc clair que l'*AVC* est un composant important influençant les performances de SELinux.

La figure 4.2 détaille les résultats obtenus pour les appels système `read` et `write`. On remarque ici que ces appels systèmes sont très peu impactés par le retrait de l'*AVC*. Les performance avec et sans SELinux (avec l'*AVC*) sont aussi très proches (impact de SELinux inférieur à 5%).

4.2.3 Premières conclusions

D'autres tests ont été réalisés avec la suite de benchmarks *Phoronix Test Suite* et les résultats sont similaires. Ils montrent que l'impact de SELinux est très limité sur les opérations de type écriture/lecture sur un disque, avec le système de fichiers `ext4`. Globalement, l'impact de SELinux est faible, de l'ordre du pourcent, donc très proche de la marge d'erreur des mesures.

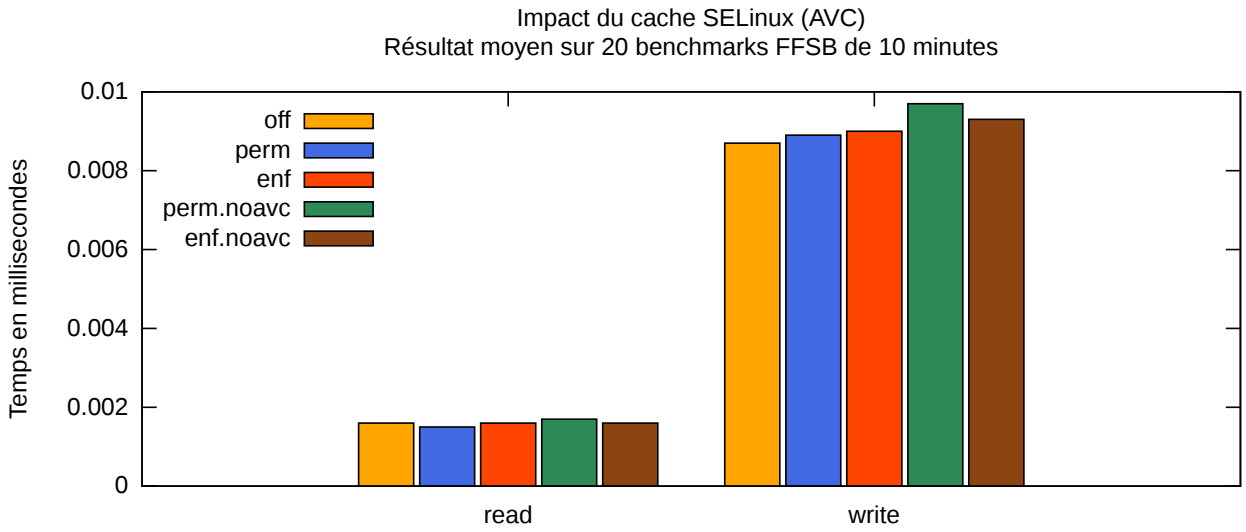


Figure 4.2 – Résultat pour les appels système `read` et `write`, avec et sans l'*AVC*

De plus, de part leur nature, les outils de benchmark en espace utilisateur ne nous permettent pas d'obtenir d'informations plus précises. Nous sommes limités à une vue macroscopique de ce qui se passe dans le noyau puisque qu'il n'est pas possible de savoir quelles fonctions y sont appelées. Nous ne pouvons que mesurer le temps global d'exécution d'un appel système.

Nous avons donc cherché à déterminer plus précisément quels éléments de SELinux dans le noyau peuvent être améliorés. Il est pour cela nécessaire d'utiliser des outils de suivi d'exécution du noyau. Plusieurs outils sont disponibles pour obtenir des informations sur l'exécution d'un programme tout en tenant compte des appels au noyau.

4.3 Outils de suivi de performance et d'exécution

Ces outils ont été conçus pour obtenir des informations sur l'exécution du noyau ou d'un programme en tenant compte des appels système qu'il effectue. Il est existe plusieurs méthodes qui permettent de recueillir des informations différentes.

4.3.1 Échantillonnage de la pile d'exécution

Cette méthode (*Stack sampling*) consiste à regarder à intervalles réguliers l'état dans lequel se trouve un programme. Pour cela, on définit une période d'échantillonnage, puis l'on note à chaque interruption quelles fonctions sont présentes dans la pile d'appel. Ce comportement a pour avantage majeur de ne pas trop impacter les performances générales du programme. En revanche, on obtient seulement un aperçu des fonctions appelées. Il n'est pas possible d'obtenir de résultat numérique précis de cette façon.

Pour les processeurs qui en possèdent, il est possible de s'aider de compteurs matériels (*performance counters*) pour obtenir d'autres informations, comme le nombre de défauts de cache (*cache-misses*), de mauvaise prédiction de branchement (*branch-misses*), le nombre d'instructions exécutées, etc.

4.3.1.1 OProfile

OProfile est constitué d'un module noyau et d'un service en espace utilisateur contrôlé par la commande `opcontrol`. Le listing 4.3 est un exemple d'utilisation complet d'OProfile pour obtenir des statistiques d'exécution incluant les fonctions du noyau.

Listing 4.3– Exemple d'utilisation de OProfile

```
1 #!/bin/bash
2
3 VERSION=‘uname -r‘
4 VMLINUX=”/usr/lib/debug/lib/modules/”‘uname -r‘”/vmlinux”
```

```
BACKTRACE=50

# Vérifie que le démon est bien éteint
opcontrol --shutdown &> /dev/null

# Désactive le watchdog nmi
‘echo 0 > /proc/sys/kernel/nmi_watchdog‘

# Initialise le démon opcontrol
opcontrol --init
opcontrol --reset

# Indique le binaire que nous souhaitons tracer
opcontrol --vmlinux=${VMLINUX}

# Précise quel binaire va être tracé
opcontrol --image=/bin/ls

# Indique la profondeur d’appel maximale pour OProfile
opcontrol --callgraph=${BACKTRACE}

opcontrol --separate=library,kernel,cpu,thread

# Démarre la capture
opcontrol --start

# On lance la commande qui doit être profilée
ls -alRZ /

# On s’assure que les logs sont sauvegardés et on stop OProfile
opcontrol --dump
opcontrol --stop
opcontrol --dump

# Obtention du rapport sous forme de graphe d’appel de fonctions
opreport -g --callgraph --symbols --merge all -o report.log || exit 1

# Utilisation de gprof2dot.py pour obtenir une vue graphique du résultat
cat report.log | ./gprof2dot.py -f oprofile | dot -Tpng -o output.png
```

Le script gprof2dot.py nous a permis d’afficher un graphe d’appel de fonctions à partir des résultats d’OProfile. La figure 4.3 est un exemple de graphe indiquant la séquence d’appel des fonctions ainsi que le temps passé dans chacune d’entre elles par rapport au temps passé dans la fonction supérieure dans la pile d’appel. La version complète de ce graphe est disponible à l’Annexe B.1.

Malheureusement, il est difficile de filtrer les rapports d’OProfile, donc l’affichage est encombré par de nombreux éléments qui ne nous intéressent pas pour notre étude.

4.3.1.2 perf

perf est un outil disponible avec le noyau Linux pour obtenir des informations similaires à celles fournies par OProfile. C’est un programme en ligne de commande qui exploite des fonctionnalités ayant été rajoutées au noyau pour échantillonner la pile d’exécution et exploiter les compteurs matériels.

Il est très clairement précisé que les résultats obtenus avec cette méthode doivent être interprétés avec beaucoup d’attention puisqu’ils dépendent de nombreux facteurs liés au matériel et difficilement prévisibles. Des exemples d’utilisation de cet outil sont détaillés dans [16].

4.3.1.3 Limites

Les outils précédemment détaillés ne sont pas assez précis pour nos contraintes. De nombreuses fonctions ne sont pas présentes dans les résultats et leur absence ne signifie pas qu’elles ne sont pas appelées. Il est de plus difficile de

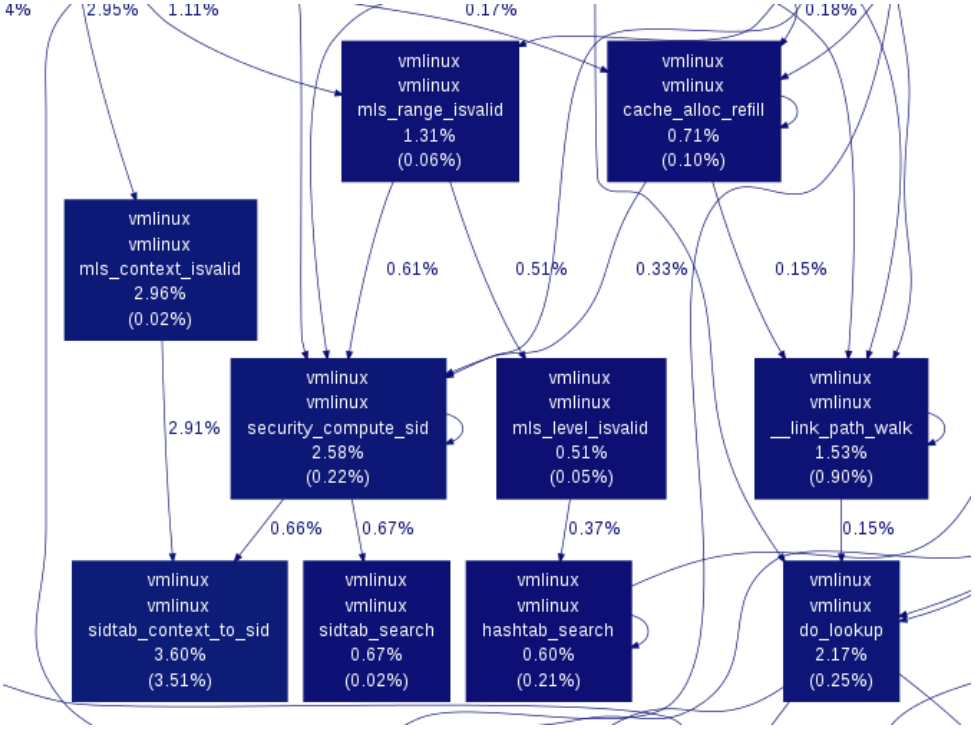


Figure 4.3 – Exemple de graphe obtenu (zoom sur une partie liée à SELinux)

déterminer l'ordre d'exécution des fonctions dont on obtient le résultat ou de séparer les résultats par appel système par exemple. Nous cherchons donc à obtenir un graphe complet des appels de fonctions, dans lequel nous pourrons distinguer les fonctions liées à SELinux des autres.

4.3.2 Sondes statiques et dynamiques

Une autre solution consiste à définir un certain nombre de sondes statiques à des endroits précis d'un logiciel. Ces sondes permettent alors d'obtenir des résultats très précis mais limitent l'étude aux fonctions qui auront été choisies préalablement. Les programmes en espace utilisateur et le noyau peuvent utiliser ce type de profilage, qui impacte relativement peu les performances d'un système puisque seules les sondes des fonctions choisies et activées sont exécutées. Malheureusement, il est nécessaire de modifier le noyau pour rajouter de nouvelles sondes.

Il est possible d'éviter ces contraintes en utilisant des sondes dynamiques. Les instructions du noyau présentes en mémoire seront modifiées pendant l'exécution pour dérouter temporairement l'exécution. Pour cela, il est nécessaire de remplacer un certain nombre d'instructions lors de l'exécution ce qui n'est pas sans risques. Si le code ajouté ne se comporte pas correctement, il est possible que le noyau ne puissent pas continuer son exécution, entrainant l'arrêt du système.

4.3.2.1 DTrace

A l'origine conçu pour le système d'exploitation Solaris, DTrace est un *framework* pour tracer dynamiquement et statiquement des programmes. Les articles [12, 13, 14] détaillent les concepts à la base du fonctionnement de DTrace qui assurent la sureté d'utilisation d'un tel outil, pourtant intrusif, dans un environnement de production. C'est le succès de DTrace sur le système d'exploitation Solaris qui a poussé la société Oracle à réaliser le portage de cet outil sur Linux. Le code source n'étant pas encore disponible publiquement, nous n'avons pas pu le tester.

4.3.2.2 SystemTap

Un mécanisme a été introduit dans le noyau Linux pour pouvoir étudier dynamiquement son exécution. Les *Kprobes*, *Jprobes* et *Return Probes* insérent des instructions respectivement à une adresse arbitraire, au début d'une fonction et à la fin d'une fonction dans le noyau.

SystemTap est un projet supporté par Red Hat et IBM pour développer un équivalent à DTrace pour Linux. Il s'appuie sur les possibilités offertes par les *Kprobes* pour instrumenter le noyau par l'intermédiaire d'un langage de script proche du C.

Le listing 4.4 est un exemple de script qui trace les appels système `open` et `close` effectués sur l'ensemble du système.

Listing 4.4– Exemple de script SystemTap

```
#!/usr/bin/stap
probe begin {
  print("Tracking sys_open & sys_close\n")
}

probe kernel.function("sys_open") {
  print("sys_open :", ppid(), " :", pid(), " :", execname(), " :", uid(), "\n")
}

probe kernel.function("sys_open").return {
  print("sys_open.return :", ppid(), " :", pid(), " :", execname(), " :", uid(), "\n")
}

probe kernel.function("sys_close") {
  print("sys_close :", ppid(), " :", pid(), " :", execname(), " :", uid(), "\n")
}

probe kernel.function("sys_close").return {
  print("sys_close.return :", ppid(), " :", pid(), " :", execname(), " :", uid(), "\n")
}

probe end {
  print("Done\n")
}
```

Malheureusement, bien que puissant, cet outil n'est pas adapté à notre besoin immédiat. Pour obtenir un graphe d'appel de fonction, il faudrait indiquer à SystemTap l'ensemble précis des fonctions du noyau que nous voulons suivre. Or, nous ne savons pas encore exactement lesquelles seront importantes par la suite. Nous cherchons donc plutôt à filtrer progressivement un graphe pour obtenir des résultats.

4.3.2.3 ftrace

ftrace est une interface de profiling noyau accessible par l'intermédiaire du pseudo système de fichiers *debugfs*. L'implémentation utilise entre autres les *Kprobes*. Pour tracer les appels de fonctions nous avons utilisé une fonctionnalité de ftrace qui fournit rapidement un graphe détaillé et très complet. Le listing 4.5 correspond à l'ensemble des fonctions qui sont appelées par l'appel système `open`, ainsi que les temps nécessaire à leur exécution.

Listing 4.5– Exemple de graphe d'appel de fonction obtenu avec ftrace

```
           |  sys_open() {
           |    do_sys_open() {
           |      getname() {
           |        kmem_cache_alloc() {
0.163 us   |          _cond_resched();
0.608 us   |        }
           |        strncpy_from_user() {
0.164 us   |          _cond_resched();
0.561 us   |        }
1.682 us   |      }
           |      alloc_fd() {
0.188 us   |        _spin_lock();
0.165 us   |        expand_files();
0.890 us   |      }
           |      do_filp_open() {
           |        get_empty_filp() {
           |          kmem_cache_alloc() {
```

```
0.164 us   |            _cond_resched();
0.631 us   |          }
           |          security_file_alloc() {
           |            selinux_file_alloc_security() {
           |              kmem_cache_alloc_notrace() {
0.163 us   |                _cond_resched();
0.542 us   |              }
0.161 us   |              slab_buffer_size();
1.202 us   |            }
1.560 us   |          }
2.705 us   |        }
           |        do_path_lookup() {
           |          path_init() {
0.186 us   |            _read_lock();
0.541 us   |          }
           |          path_walk() {
           |            __link_path_walk() {
0.164 us   |              acl_permission_check();
           |              security_inode_permission() {
           |                selinux_inode_permission() {
           |                  inode_has_perm() {
           |                    avc_has_perm() {
0.192 us   |                      avc_has_perm_noaudit();
0.171 us   |                      avc_audit();
0.856 us   |                    }
1.213 us   |                  }
1.542 us   |                }
1.872 us   |              }
           |              do_lookup() {
           |                __d_lookup() {
0.163 us   |                  _spin_lock();
0.536 us   |                }
0.164 us   |                follow_managed();
1.196 us   |              }
0.180 us   |              dput();
0.166 us   |              acl_permission_check();
           |              security_inode_permission() {
           |                selinux_inode_permission() {
           |                  inode_has_perm() {
           |                    avc_has_perm() {
0.181 us   |                      avc_has_perm_noaudit();
0.163 us   |                      avc_audit();
0.823 us   |                    }
0.530 us   |    unroll_tree_refs();
0.200 us   |    kfree();
2.604 us   |  }
```

Il est possible de filtrer progressivement la sortie de ftrace tel que détaillé dans le listing 4.6. On obtient pour commencer un rapport détaillé, long, sur une durée courte. On peut y distinguer un certain nombre de fonctions qui nous intéressent plus particulièrement. Puis on filtre la sortie de ftrace pour au final n’obtenir que les informations nécessaires.

Un outil en espace utilisateur nommé *trace-cmd* est disponible, pour simplifier la gestion des fonctions de ftrace.

Listing 4.6– Exemple d’utilisation de trace-cmd pour exploiter les résultats de ftrace

```
#!/bin/bash

# Désactive le traçage dans le noyau et vide les buffers contenant les traces.
./trace-cmd reset

# Enregistre une trace noyau  :
#       * en utilisant le plugin function_graph pour obtenir un graphe d’appel de fonctions ;
#       * en précisant le nom des fonctions dans le noyau que l’on souhaite surveiller ;
#       * en indiquant quelle commande sera exécutée.
./trace-cmd record \
```

```
  -p function_graph \
  -l "sys_*" -l "avc*" -l "avtab*" -l "mls*" -l "selinux*" -l "do_IRQ" -l "cond_compute_av" \
  -F ffsb profile_everything > resultat_ffsb.log

# Parse les résultats obtenus pour obtenir une version texte de cette trace.
./trace-cmd report > resultat_ftrace.log
```

L'impact de ftrace sur les performances est en revanche non négligeable. La figure 4.4 montre que ftrace peut faire doubler le temps d'exécution en moyenne d'un appel système. Cet outil n'est donc pas vraiment adapté pour mesurer des performances réelles, mais on obtient une idée des points à améliorer. Il faudra ainsi confirmer chaque modification par un test général.

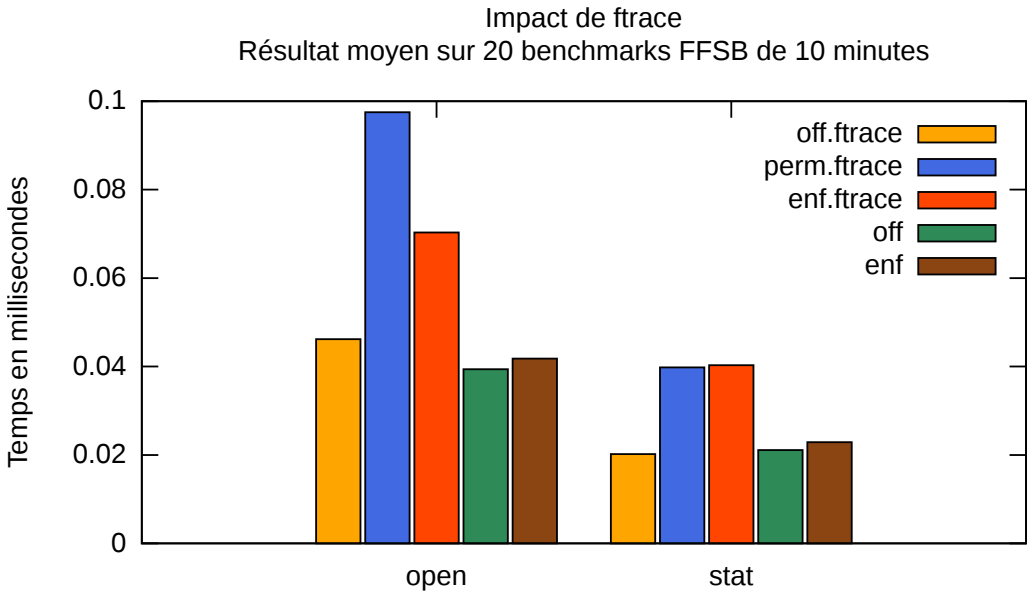


Figure 4.4 – Exemple d'impact de ftrace sur les performances

4.3.2.4 Parseur de log ftrace/trace-cmd

Nous avons réalisé plusieurs outils pour pouvoir exploiter les résultats de ftrace. Il faut en effet noter qu'une exécution de dix minutes du benchmark FFSB produit 1,3 Gio de logs, et cela après filtrage des fonctions qui nous intéressent plus particulièrement et qui sont liées à SELinux. Mes outils fournissent des résultats globaux et des moyennes que l'on peut afficher à l'aide de *gnuplot*.

L'interprétation de ces graphiques ne donnera pas lieu à des résultats chiffrés puisqu'ils ne seraient pas pertinents vu l'impact non négligeable sur les performances. Ce sont des tendances que nous recherchons, pour déterminer quelles parties de SELinux sont susceptibles d'être améliorées.

La figure 4.5 détaille le nombre d'appels système effectués par le benchmark FFSB sur une période de dix minutes.

La figure 4.6 représente la somme des temps passés dans un appel système, pour chaque appel système. De plus, cette somme est divisée en plusieurs catégories, regroupant le temps passé dans des fonctions liées à SELinux. Ainsi, la catégorie *avc* regroupe les fonctions chargée de la gestion et de la recherche dans l'*AVC*. La catégorie *avtab* correspond à la recherche dans le *Security server* et *mls* à l'ensemble des vérifications liées à l'utilisation des niveaux de sécurité.

La figure 4.7 représente les mêmes informations que la figure précédente, mais sur une échelle en pourcentage. On distingue alors plus clairement pour chaque appel système l'impact des différents éléments de SELinux. On constate ainsi que le temps d'exécution consacré à des fonctions liées à SELinux varie fortement suivant les appels système.

Tout d'abord, les appels système `read` et `write` sont peu impactés, et cela est dû au fonctionnement de SELinux : en effet, une fois un descripteur de fichier ouvert, le *SID* du processus exécutant l'appel système `open` ainsi que le *SID* de

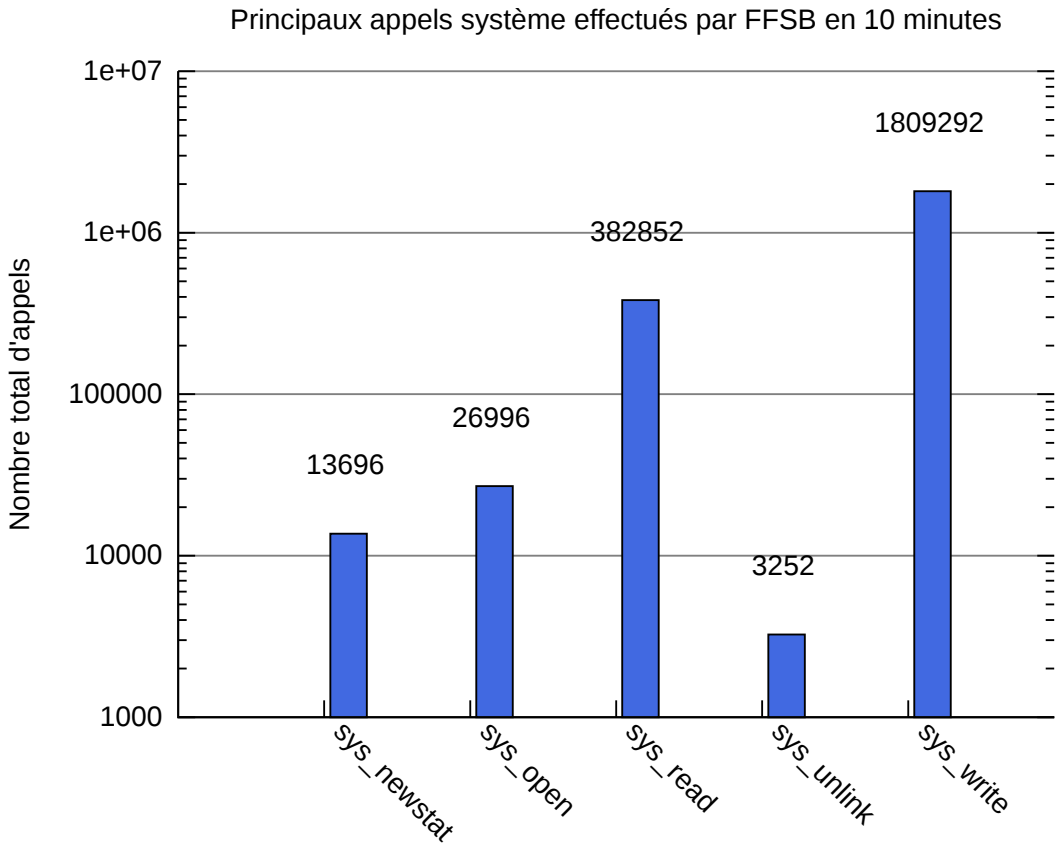


Figure 4.5 – Principaux appels système effectués par FFSB en dix minutes

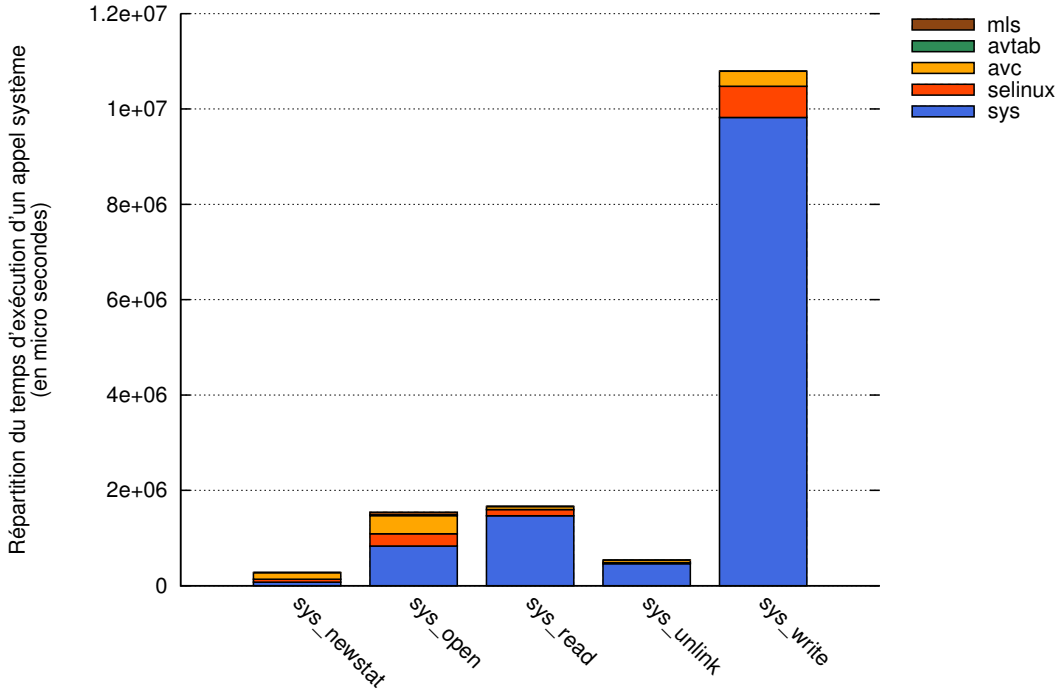


Figure 4.6 – Somme des temps d'exécution de chaque appel système

l'inode sont stockés dans la structure `struct file_security_struct` du descripteur de fichier. Les vérifications pour les appels système `read` et `write` se limiteront alors à vérifier que le contexte de sécurité du fichier ou du processus n'a pas été modifié depuis l'ouverture et que la politique SELinux n'a pas été rechargée. Le temps passé dans les fonctions liées à l'*AVC* est donc faible par rapport au temps total d'exécution pour ces appels système.

D'autre part, les appels `open` et `stat` sont fortement impactés par SELinux. Contrairement à `read` et `write` qui

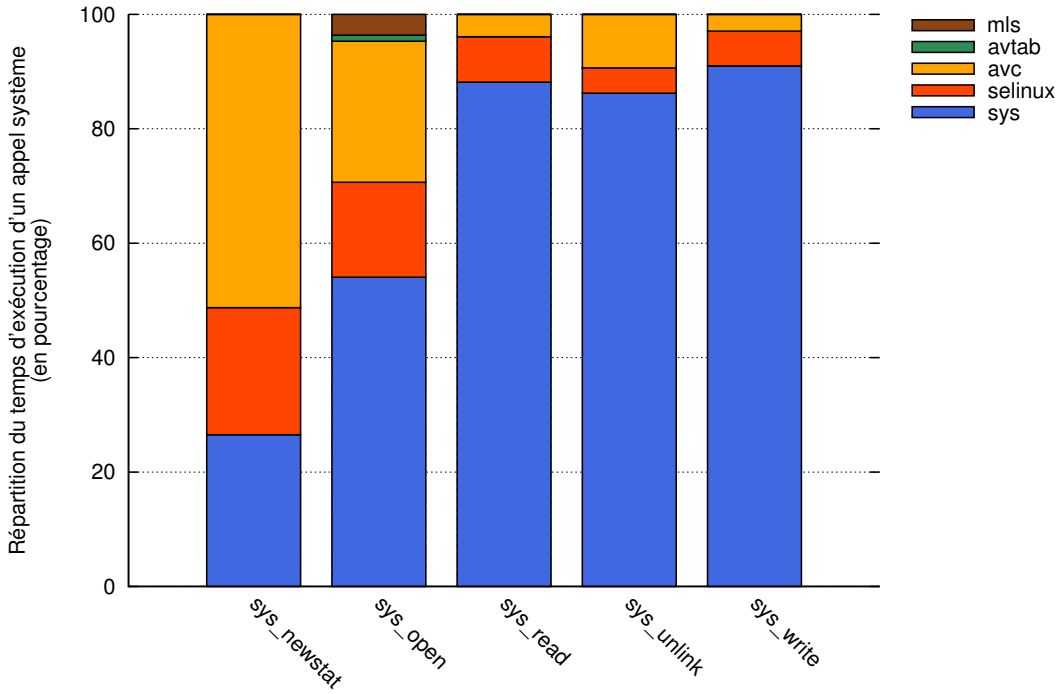


Figure 4.7 – Détail du temps passé dans chaque partie de SELinux pour différents appels système

utilisent un descripteur de fichier ouvert préalablement, ces appels agissent sur un fichier en le désignant par son chemin d'accès. Il faut donc en plus vérifier qu'un processus dispose des permissions nécessaires pour traverser chaque partie de ce chemin d'accès en partant de la racine, puis qu'il peut effectuer l'opération sur le fichier. De nombreuses requêtes sont ainsi adressées à l'*AVC* et au *Security server*. C'est aussi lors de l'ouverture d'un fichier que les vérifications concernant les niveaux de sécurité sont effectuées.

4.3.3 Synthèse

L'impact de SELinux sur un système est donc globalement faible dans notre cas de figure très ciblé. Le pire cas de figure correspond donc à l'ouverture de fichiers et aux opérations sur les attributs. Un programme effectuant beaucoup d'entrées/sorties sur un même fichier ne sera que très peu impacté par rapport à un programme écrivant des petites quantités de données dans beaucoup de fichiers différents.

Cette constatation peut être étendue à un contexte avec de nombreux utilisateurs SELinux, plusieurs niveaux de sécurité et de catégories, dans lequel l'impact sera le plus important. Chaque interaction effectuée par un utilisateur différent ou sous une catégorie différente formera une nouvelle entrée qu'il faudra ajouter à l'*AVC*, qui ne pourra plus remplir sa fonction de cache efficacement.

L'importance de l'*AVC* est confirmée par les publications [17, 18]. Nous allons tout d'abord concentrer nos efforts sur l'*AVC*, et ce pour les raisons suivantes :

- Tous les appels système utilisent l'*AVC*, donc une amélioration de cet élément sera bénéfique globalement ;
- Les modifications ont peu de risque de réduire ou compromettre la sécurité du modèle puisque l'architecture même de SELinux n'est pas modifiée ;
- L'*AVC* n'est pas un élément trop complexe et constitue donc un bon point d'entrée pour commencer à modifier le noyau.

4.4 Optimisations

Plusieurs pistes d'optimisation ont donc été envisagées à la suite des résultats obtenus. Par défaut, l'*AVC* est un tableau de taille fixe (512 entrées) qui contient des pointeurs vers des listes chaînées de tailles variables. Le nombre total d'éléments dans l'*AVC* est nommé *cache_threshold* et peut être modifié à l'exécution en écrivant la valeur choisie dans le fichier virtuel `/selinux/avc/cache_threshold`. Plus le nombre d'entrées dans le cache est grand, moins

le recours au *Security Server* est nécessaire, mais plus la taille des listes chaînées augmente et donc plus le temps de parcours de ces listes augmente.

4.4.1 Contenu de l'*AVC*

Avant de modifier l'*AVC*, nous avons souhaité étudier son contenu sur un système en cours d'exécution. Nous avons ainsi réalisé une première modification permettant de lister le contenu de l'*AVC* à un instant donné (voir patch C.1). On utilise pour cela le fichier virtuel `/selinux/avc/cache_dump`, comme dans l'exemple 4.7.

Listing 4.7– Contenu de l'*AVC*

```
cat /selinux/avc/cache_dump
[...]
# Contenu de la liste chaînée à l'indice 10  :
Line 10 :
1 : scontext=system_u system_r initrc_t s0 tcontext=system_u object_r usr_t s0 tclass=dir
2 : scontext=sysadm_u sysadm_r sysadm_t s0-s0 c0-c1023 tcontext=system_u object_r usr_t s0 tclass=file
# Contenu de la liste chaînée à l'indice 11  :
Line 11 :
1 : scontext=system_u system_r chkpwd_t s0-s0 c0-c1023 tcontext=system_u object_r shadow_t s0 tclass=file
2 : scontext=system_u system_r getty_t s0 tcontext=system_u object_r init_t s0 tclass=fd
# Les lignes 12 et 13 sont vides  :
Line 12 :
Line 13 :
[...]
```

Nous avons constaté que la répartition dans le cache n'était pas toujours optimale puisque qu'en moyenne, seulement la moitié des listes chaînées étaient utilisées. L'augmentation du *cache_threshold* n'as pas eu d'effets bénéfiques lors de nos tests : le problème de répartition est accentué et la taille des listes chaînées accrue.

4.4.2 Taille et remplissage de l'*AVC*

Le listing C.2 détaille les modifications nécessaire pour ajouter deux paramètres au noyau Linux permettant de déterminer lors du démarrage la taille souhaitée pour l'*AVC*, et le nombre d'entrée maximal qu'il doit comporter par défaut. Deux options de configuration sont définies pour pouvoir indiquer des valeurs par défaut à la compilation. Le listing 4.8 est un exemple d'entrée de configuration GRUB pour démarrer un noyau avec une taille d'*AVC* particulière. Cette première étape nous a permis d'augmenter le nombre d'éléments présent dans l'*AVC* sans risquer de perdre en performance.

Listing 4.8– Exemple d'entrée de configuration GRUB

```
[...]
title CentOS (2.6.32-220.7.1.el6.x86_64)
    root (hd0,0)
    kernel /vmlinuz-dev ro root=/dev/mapper/vg_centos-lv_root quiet selinux.AVC_CACHE_SLOTS=2048 selinux.
        AVC_DEF_CACHE_THRESHOLD=2048
    initrd /initramfs-dev.img
[...]
```

4.4.2.1 Remplacement de la fonction de répartition de l'*AVC*

En étudiant le contenu de l'*AVC*, nous nous sommes rendu compte que la répartition dans les liste chaînées n'était pas optimale pour des tailles beaucoup plus grandes (2048, 4096 ou 8192 par exemple). La répartition s'effectue en effet à l'aide d'une fonction de hashage très simple qui utilise quelques combinaisons entre les *SID* correspondant à une interaction. Pour améliorer la répartition et réduire la taille des listes chaînées, nous avons cherché un remplaçant simple, rapide et capable de se limiter à une certaine taille dynamiquement. Il ne doit pas nécessairement être sûr cryptographiquement, mais il doit bien répartir les éléments.

Nous nous sommes inspirés de la fonction utilisée dans le patch *grsecurity*, *CrapWow* [3]. Les modifications sont listées à l'Annexe C.3. La nouvelle taille maximale pour l'*AVC* est de 2^32 éléments, ce qui correspond à la taille de

la somme de contrôle retournée par la fonction *CrapWow* (32 bits). Pour des tailles de cache allant de 512 à 8192, la répartition obtenue est en moyenne supérieure à 50%, ce qui diminue la taille des listes chaînées dans l'*AVC*.

4.4.3 Benchmark spécifique pour stresser l'*AVC*

Suite aux premières modifications, il nous est apparu nécessaire de trouver un benchmark permettant de manipuler un grand nombre de contextes de sécurité pour tester le bon remplissage et les performances de l'*AVC*. Nous avons donc réalisé un programme exécutant un grand nombre de fois une même commande, en parallèle et sous des contextes de sécurité différents. Les contextes de sécurité SELinux sont associés à des *SIDs* par le *Security Server* et ce sont ces *SIDs* qui interviennent dans le calcul de la position d'une interaction dans l'*AVC*. Il est possible de faire varier l'ensemble des catégories sous lesquelles un programme s'exécute pour que le *Security Server* leurs associent un *SID* différent. Ce programme rempli donc artificiellement l'*AVC* très rapidement si la commande exécutée accède à un grand nombre de contextes de sécurité différents. Le code est disponible à l'Annexe C.4.

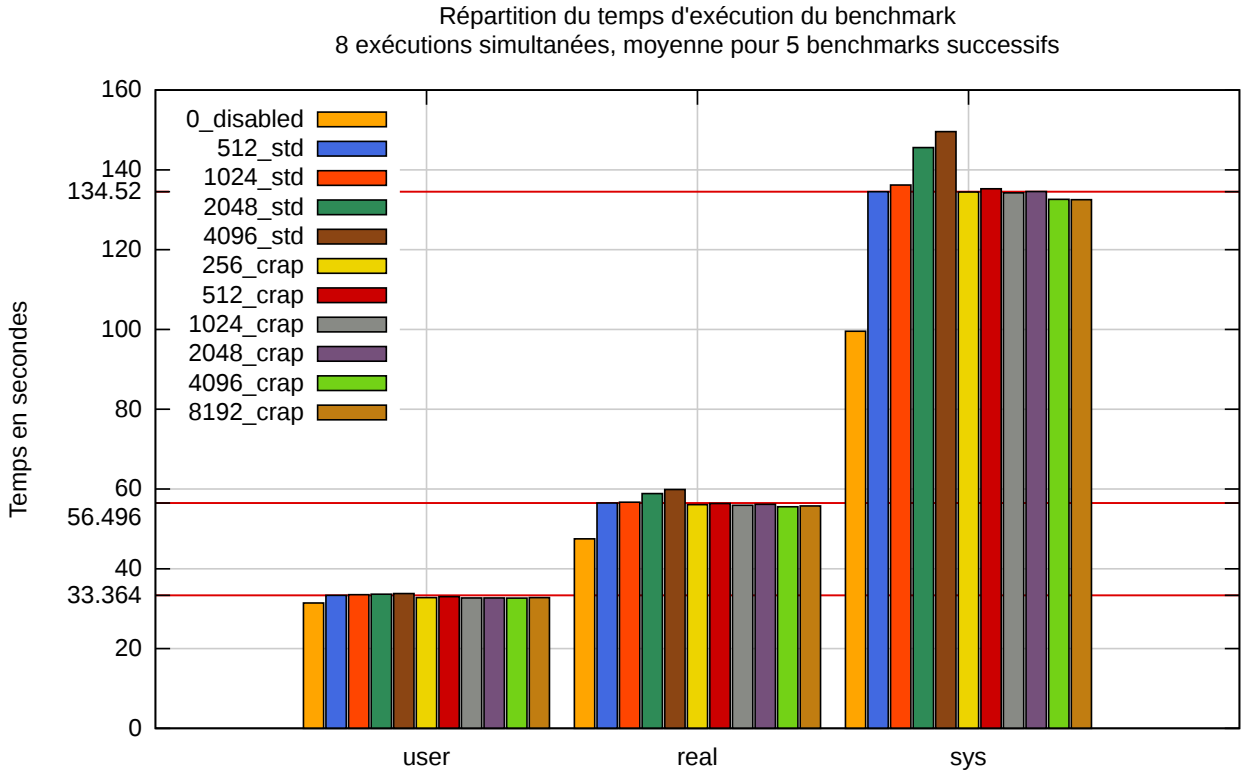


Figure 4.8 – Comparaison des temps d'exécution du benchmark avec différentes tailles pour l'*AVC*

Nous avons utilisé ce benchmark pour obtenir les résultats représentés sur la figure 4.8. Nous avons lancé simultanément huit commandes de type `ls -alR /` (détail à l'Annexe C.7) avec des catégories SELinux différentes. Pour obtenir une moyenne, cette étape a été répétée cinq fois, à la suite d'un redémarrage. Les temps ont été obtenus avec la commande `time`.

La colonne `0_disabled` correspond à une exécution sans SELinux. Les temps des colonnes `*_std` ont été obtenus en utilisant le noyau CentOS standard, en faisant varier uniquement le nombre maximal d'entrées dans l'*AVC* (*cache_threshold*). Les résultats `*_crap` correspondent à un noyau modifié pour augmenter la taille de l'*AVC* et pour utiliser l'algorithme *CrapWow*. La taille de l'*AVC* et le nombre maximal d'entrées ont été modifiés simultanément pour ces derniers résultats.

Ainsi, il est possible de constater une légère amélioration des temps d'exécution dans le noyau (*sys*) lorsque nous augmentons la taille de l'*AVC* et le nombre d'entrées. Cette amélioration est à peine discernable sur le temps global apparent d'exécution (*real*) puisque nous sommes dans un contexte multi-coeurs. Ces améliorations permettent donc bien d'utiliser des tailles de cache plus grandes, avec un nombre d'entrée plus grand.

4.5 Pistes à suivre

Plusieurs idées ont été envisagées mais non pas encore pu être implémentées par manque de temps et parfois de recul sur certain éléments.

4.5.1 Amélioration de l'*AVC*

L'algorithme chargé de la gestion de l'*AVC* est une version simplifiée de table de hashage, utilisant le principe du *Least Recently Used (LRU)* pour choisir les éléments à retirer du cache. Ainsi, des interactions utilisées régulièrement, mais de façon espacée dans le temps, ne sont pas nécessairement conservées dans le cache. D'autres algorithmes de gestion de cache sont disponibles et il serait intéressant d'utiliser un cache basé sur la fréquence d'accès ou une combinaison des deux comme décrit dans [30].

4.5.2 Amélioration du *Security Server*

Le moniteur de référence de SELinux, représenté par le *Security Server*, pourrait aussi être la cible d'améliorations.

4.5.2.1 Réduction de la taille de la politique

Tout d'abord, il est possible d'adapter la politique SELinux utilisée sur un système. Une politique plus petite, avec moins de types et d'interactions engendre un coût de recherche plus limité. Il serait alors envisageable que l'ensemble des interactions se produisant sur un système soit contenues dans l'*AVC*, si le nombre de programmes et services exécutés est restreint.

Les politiques SELinux disponibles sur Red Hat Enterprise Linux/CentOS sont volumineuses car complètes par défaut. Elles contiennent l'ensemble des modules nécessaires au fonctionnement des programmes disponibles dans les dépôts officiels ce qui ne correspond généralement pas aux programmes réellement installés sur un système.

4.5.2.2 Modularisation de la politique et du stockage

Les contraintes sur le chargement d'une politique SELinux pourraient être levées avec l'introduction d'un stockage modulaire de la politique. Cette séparation en modules pourrait aussi accélérer les recherches puisqu'un programme confiné ne peut effectuer que des opérations liées à son domaine, qui pourrait être contenu dans un module unique.

4.5.3 Répartition de l'*AVC*

Enfin, l'introduction d'un *AVC* par processus ou *thread* pourrait éviter la concurrence d'accès au cache qui est pour l'instant global. De plus, cette modification améliorerait l'utilité de l'*AVC* puisque chaque processus n'aurait alors dans son cache que les types auxquels il a accédé. La taille des caches pourrait alors être réduite ou éventuellement dynamique.

Conclusion

Dans ce rapport, nous avons détaillé les causes de l'impact de SELinux sur les performances grâce à l'étude de l'architecture et aux nombreux tests réalisés qui nous ont donné une vision globale. Les résultats ont pu être affinés à l'aide d'outils de profilage pour déterminer précisément les éléments responsables des pertes de performance. Nous avons noté la présence de plusieurs optimisations qui limitent déjà l'impact de SELinux en évitant les contrôles redondants et en mettant en cache les dernières interactions vérifiées.

En conclusion, il n'y a pas de réponse définitive sur la question des performances de SELinux. Globalement, l'impact reste faible, mais certains cas de figure peuvent mettre en évidence les limites actuelles de l'implémentation. Les appels système nécessitant la manipulation de contextes de sécurité SELinux sont les plus affectés (*open*, *stat*) alors que d'autres opérations le sont très peu (*read*, *write*). Le développement de SELinux est encore actif, ainsi de nombreuses améliorations ont été apportées aux dernières versions de la distribution Fedora par exemple, et elles seront présentes dans la prochaine version de Red Hat Enterprise Linux. Il faudra alors mettre à jour ces résultats.

Nous continuerons à effectuer des tests de performance et à travailler sur des optimisations de sorte à obtenir des résultats à proposer à la communauté pour inclusion dans les futures versions du noyau Linux. Les modifications réalisées jusque là pouvant aussi être proposées. De nouveaux tests seront réalisés notamment avec le système de fichier Lustre pour vérifier nos conclusions.

Ce stage a aussi été très enrichissant d'un point de vue personnel. Bénéficiant d'une grande autonomie dans l'organisation de mon travail, j'ai pu chercher à comprendre l'ensemble des facettes de SELinux, tant au niveau du noyau que de l'espace utilisateur. Mes encadrants ont su me guider et me conseiller tout au long du stage et je les en remercie.

Bibliographie

[1] LKML : John Johansen : Apparmor FAQ. http://lkml.org/lkml/2007/4/16/266.

[2] J.P. Anderson. Computer security technology planning study. volume 2. Technical report, DTIC Document, 1972. http://csrc.nist.gov/publications/history/ande72.pdf.

[3] Andrew. Non-cryptographic hash function zoo - crapwow, 2010. http://www.team5150.com/~andrew/noncryptohashzoo/CrapWow.html.

[4] D.E. Bell. Secure computer systems : Mathematical foundations. Technical report, DTIC Document, 1973.

[5] K.J. Biba. Integrity considerations for secure computer systems. Technical report, DTIC Document, 1977.

[6] J. Briffaut, J. Rouzaud-Cornabas, C. Toinard, and Y. Zemali. A new approach to enforce the security properties of a clustered high-interaction honeypot. In *High Performance Computing & Simulation, 2009. HPCS'09. International Conference on*, pages 184–192. IEEE, 2009.

[7] Jérémy Briffaut. *Formalisation et garantie de propriétés de sécurité système : application à la détection d'intrusions*. PhD thesis, ENSI de Bourges, 2007. Available at : http://tel.archives-ouvertes.fr/tel-00261613.

[8] Jérémy Briffaut, Jean-François. Lalande, and Christian Toinard. Formalization of security properties : enforcement for mac operating systems and verification of dynamic mac policies. *International journal on advances in security*, 2(4) :325–343, 2010.

[9] Jérémy Briffaut, Martin Peres, Jonathan Rouzaud-Cornabas, Jigar Solanki, Christian Toinard, and Benjamin Venelle. PIGA-OS : retour sur le système d'exploitation vainqueur du défi sécurité. In *RenPar'20 / SympA'14/ CFSE 8*, 2011. Available at : http://renpar.irisa.fr/cfse8/cfse8_16.pdf.

[10] Joshua Brindle. Brindle on security, Avril 2006. http://securityblog.org/brindle/2006/04/02/top-down-vs-bottom-up-policy-development/.

[11] Joshua Brindle. Brindle on security, Avril 2006. http://securityblog.org/brindle/2006/07/05/selinux-policy-module-primer/.

[12] Bryan Cantrill. Dtrace safety, Juillet 2005. http://dtrace.org/blogs/bmc/2005/07/19/dtrace-safety/.

[13] Bryan Cantrill. Hidden in plain sight. *Queue*, 4(1) :26–36, February 2006.

[14] Bryan M. Cantrill, Michael W. Shapiro, and Adam H. Leventhal. Dynamic instrumentation of production systems. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '04, pages 2–2, Berkeley, CA, USA, 2004. USENIX Association.

[15] Crispin Cowan. Securing Linux Systems with AppArmor. In *DEF CON 15*, 2007. Available at : http://www.defcon.org/images/defcon-15/dc15-presentations/dc-15-cowan.pdf.

[16] Stephane Eranian, Eric Gouriou, Tipp Moseley, and Willem de Bruijn. Perf wiki - tutorial, 2012. https://perf.wiki.kernel.org/index.php/Tutorial.

[17] L. Fiorin, A. Ferrante, K. Padarnitsas, and S. Carucci. Hardware-assisted security enhanced Linux in embedded systems : a proposal. In *Proceedings of the 5th Workshop on Embedded Systems Security*, page 3. ACM, 2010.

[18] L. Fiorin, A. Ferrante, K. Padarnitsas, and F. Regazzoni. Security enhanced linux on embedded systems : A hardware-accelerated implementation. In *Design Automation Conference (ASP-DAC), 2012 17th Asia and South Pacific*, pages 29–34. IEEE, 2012.

[19] grsecurity website. http://grsecurity.net.

[20] Why doesn't grsecurity use LSM ? http://grsecurity.net/lsm.php.

[21] Richard Haines. *The SELinux Notebook - The Foundations - 2nd edition*. Richard Haines, 2009. Available at : http://www.freetechbooks.com/the-selinux-notebook-the-foundations-t785.html.

[22] T. Harada, T. Horie, and K. Tanaka. Access policy generation system based on process execution history. *Network Security Forum 2003*, 2003. Available at : http://en.sourceforge.jp/projects/tomoyo/docs/nsf2003-en.pdf/en/2/nsf2003-en.pdf.pdf.

[23] T. Harada, T. Horie, and K. Tanaka. Towards a manageable Linux security. In *Linux Conference*, volume 2005, 2005. Available at : http://ko.sourceforge.jp/projects/tomoyo/docs/lc2005-en.pdf/en/2/lc2005-en.pdf.pdf.

[24] Michael A. Harrison, Walter L. Ruzzo, and Jeffrey D. Ullman. Protection in operating systems. *Communications of the ACM*, 19(8) :461–471, 1976. Available at : http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.106.7226&rep=rep1&type=pdf.

[25] Intel Trusted Execution Technology (TXT). http://www.intel.com/technology/malwarereduction/index.htm.

[26] Michael Larabel. The Cost of SELinux, Audit, & Kernel Debugging, 2009. http://www.phoronix.com/scan.php?page=article&item=fedora_debug_selinux.

[27] Michael Larabel. Does SELinux Slow Down Fedora 15 ?, 2011. http://www.phoronix.com/scan.php?page=article&item=fedora_15_selinux.

[28] D.C. Latham. Department of Defense Trusted Computer System Evaluation Criteria. *Department of Defense*, 1986.

[29] How is TOMOYO linux different from SELinux and AppArmor ? Secure OS comparison at a glance. http://tomoyo.sourceforge.jp/wiki-e/?WhatIs#comparison.

[30] Nimrod Megiddo and Dharmendra S. Modha. Arc : A self-tuning, low overhead replacement cache. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, pages 115–130, Berkeley, CA, USA, 2003. USENIX Association.

[31] J. Morris, S. Smalley, and G. Kroah-Hartman. Linux security modules : General security support for the linux kernel. In *USENIX Security Symposium*, 2002.

[32] SELinux Project. Selinux boot process. http://selinuxproject.org/page/NB_LSM#SELinux_Boot_Process/.

[33] QNX Realtime Operating System. http://www.qnx.com.

[34] Enabling mls in selinux. http://docs.redhat.com/docs/en-US/Red_Hat_Enterprise_Linux/6/html/Security-Enhanced_Linux/enabling-mls-in-selinux.html.

[35] Casey Schaufler. The Simplified Mandatory Access Control Kernel, 2007. Available at : http://netscale.cse.nd.edu/twiki/pub/RepositorySimplifiedMandatoryAccessControlKernel/SmackWhitePaper.pdf.

[36] Casey Schaufler. Smack is Alive and Well. In *Linux Security Summit*, 2011. Available at : http://selinuxproject.org/~jmorris/lss2011_slides/SmackIntelPlumbers2011.pdf.

[37] Lars Strand. Rhel5 selinux : A benchmark, 2007. http://blog.larsstrand.org/2007/11/rhel5-selinux-benchmark.html.

[38] The TrustedBSD project. http://www.trustedbsd.org.

[39] Akari/tomoyo functionality comparison table. http://akari.sourceforge.jp/comparison.html.en.

[40] Extensible firmware interface (efi) and unified efi (uefi). http://www.intel.com/content/www/us/en/architecture-and-technology/unified-extensible-firmware-interface/efi-homepage-general-technology.html.

[41] Robert N. M. Watson and Jonathan Anderson. Capsicum : practical capabilities for UNIX. http://www.cl.cam.ac.uk/research/security/capsicum/.

[42] C. Wright, C. Cowan, S. Smalley, J. Morris, and G. Kroah-Hartman. Linux security modules : General security support for the linux kernel. In *Proceedings of the 11th USENIX Security Symposium*, volume 2. San Francisco, CA, 2002. Available at : http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.84.6867&rep=rep1&type=pdf.

Annexe A

Politiques SELinux

Ces politiques SELinux sont des modules à ajouter à la politiques MLS proposée par Red Hat et distribuée dans le paquet `selinux-policy-mls`.

Politique pour X, Firefox, OpenOffice.org

Listing A.1– xorg_fixes.te

```
policy_module(xorg_fixes, 0.0.1)

require {
  type staff_t;
  type xserver_t;
  type xevent_t;
  type root_xdrawable_t;
  type root_xcolormap_t;
    type staff_openoffice_t;

  class x_drawable { get_property setattr show receive set_property create send add_child write read
      getattr remove_child list_child override destroy blend hide };
  class x_synthetic_event { receive send };
  class x_server grab;
  class x_keyboard { use getfocus getattr setattr };
  class x_resource { write read };
    class x_keyboard freeze;
    class x_screen setattr;
    class x_drawable { read manage };
}

allow staff_t xevent_t x_synthetic_event send;
allow staff_t xserver_t x_keyboard freeze;
allow staff_t xserver_t x_screen setattr;
allow staff_t root_xdrawable_t x_drawable { read manage };

allow staff_t staff_openoffice_t x_drawable { manage hide setattr show };
allow staff_t staff_openoffice_t x_resource { write read };
```

Politique pour FFSB

Listing A.2– ffsb.te

```
policy_module(ffsb, 0.0.1)

type ffsb_t;
type ffsb_dir_t;
type ffsb_exec_t;

```

```
require {
  type staff_t, sysadm_t, initrc_t;
  type user_home_t, user_home_dir_t;
  type setfiles_t;
  type fs_t;
  type null_device_t, user_devpts_t, device_t, urandom_device_t;
  type init_t, etc_t, root_t, usr_t, lib_t, bin_t, locale_t;
  type file_context_t;
  type ld_so_cache_t, ld_so_t;
  type home_root_t;
  type shell_exec_t;
  type devtty_t, proc_t;
  type debugfs_t;
  type newrole_t;
  type user_tty_device_t;
  type ksmtuned_t;
  class chr_file { read write ioctl };
}

role staff_r types ffsb_t;
role sysadm_r types ffsb_t;
role system_r types ffsb_t;

allow setfiles_t ffsb_exec_t file { getattr setattr open read relabelfrom relabelto };
allow setfiles_t ffsb_t file { getattr setattr open read relabelfrom relabelto };
allow setfiles_t ffsb_t dir { getattr setattr open read search relabelfrom relabelto };
allow setfiles_t ffsb_dir_t dir { getattr setattr open read search relabelfrom relabelto };

allow ffsb_t fs_t filesystem getattr;
allow ffsb_exec_t fs_t filesystem associate;
allow ffsb_dir_t fs_t filesystem associate;
allow ffsb_t fs_t filesystem associate;

domain_auto_trans(staff_t, ffsb_exec_t, ffsb_t);
domain_auto_trans(sysadm_t, ffsb_exec_t, ffsb_t);
allow ffsb_t ffsb_exec_t file entrypoint;

allow staff_t ffsb_exec_t file *;
allow staff_t ffsb_t file *;
allow staff_t ffsb_t dir *;
allow staff_t ffsb_dir_t dir *;
allow staff_t ffsb_t process signal;
allow staff_t ffsb_t lnk_file { read getattr };

allow sysadm_t ffsb_exec_t file *;
allow sysadm_t ffsb_t file *;
allow sysadm_t ffsb_t dir *;
allow sysadm_t ffsb_dir_t dir *;
allow sysadm_t ffsb_t process signal;
allow sysadm_t ffsb_t lnk_file { read getattr };

allow ffsb_t user_home_t dir { search getattr };
allow ffsb_t user_home_t file { open read getattr write };
allow ffsb_t user_home_dir_t dir search;
allow ffsb_t ffsb_t file *;
allow ffsb_t ffsb_t dir *;
allow ffsb_t ffsb_dir_t dir *;
allow ffsb_t ffsb_t lnk_file { read getattr };

type_transition ffsb_t ffsb_dir_t dir ffsb_t;
type_transition ffsb_t ffsb_dir_t file ffsb_t;

allow ffsb_t bin_t dir search;
allow ffsb_t bin_t file { open read execute getattr execute_no_trans};
allow ffsb_t bin_t lnk_file read;
allow ffsb_t debugfs_t file read;
```

```
allow ffsb_t device_t dir { search getattr read };
allow ffsb_t devtty_t chr_file { open read write };
allow ffsb_t etc_t dir search;
allow ffsb_t etc_t file { read getattr open };
allow ffsb_t home_root_t dir search;
allow ffsb_t init_t process sigchld;
allow ffsb_t ld_so_cache_t file { open read getattr };
allow ffsb_t ld_so_t file { open read getattr execute };
allow ffsb_t lib_t dir { search getattr };
allow ffsb_t lib_t file { open read getattr execute };
allow ffsb_t lib_t lnk_file read;
allow ffsb_t locale_t dir search;
allow ffsb_t locale_t file { open read getattr };
allow ffsb_t null_device_t chr_file ioctl;
allow ffsb_t null_device_t chr_file { read write open };
allow ffsb_t proc_t dir search;
allow ffsb_t proc_t file { read open getattr };
allow ffsb_t proc_t lnk_file read;
allow ffsb_t root_t dir { search getattr };
allow ffsb_t self capability dac_override;
allow ffsb_t self process { sigchld fork };
allow ffsb_t self unix_stream_socket create;
allow ffsb_t shell_exec_t file { getattr execute read open execute_no_trans };
allow ffsb_t staff_t fd use;
allow ffsb_t staff_t process sigchld;
allow ffsb_t sysadm_t fd use;
allow ffsb_t sysadm_t process sigchld;
allow ffsb_t urandom_device_t chr_file { open read };
allow ffsb_t user_devpts_t chr_file { read write getattr ioctl };
allow ffsb_t user_tty_device_t chr_file { read write ioctl };
allow ffsb_t usr_t dir search;

allow ffsb_t init_t fd use;
allow ffsb_t self unix_stream_socket connect;

allow ffsb_t var_run_t dir search;
allow ffsb_t var_t dir search;

allow ksmtuned_t ffsb_t dir getattr;

allow ffsb_t console_device_t chr_file { read write };
allow ffsb_t var_t dir search;
```

Politique pour ftrace

Listing A.3– ftrace.te

```
policy_module(ftrace, 0.0.1)

require {
    type sysadm_t, staff_sudo_t, sysadm_sudo_t, staff_t;
    type sysfs_t, debugfs_t;
    type user_home_t;
}

allow sysadm_t sysfs_t dir mounton;
allow sysadm_t debugfs_t filesystem mount;

allow staff_sudo_t user_home_t file execute;
allow staff_sudo_t user_home_t file execute_no_trans;

allow sysadm_sudo_t debugfs_t dir *;
allow sysadm_sudo_t debugfs_t file *;

```

```
18 allow sysadm_t debugfs_t dir *;
19 allow sysadm_t debugfs_t file *;
20
21 allow staff_sudo_t debugfs_t dir *;
22 allow staff_sudo_t debugfs_t file *;
23
24 allow staff_t debugfs_t dir *;
25 allow staff_t debugfs_t file *;
```

Politique pour perf

Listing A.4– perf.te

```
1 policy_module(perf, 0.0.1)
2
3 require {
4     type sysadm_t;
5     type sysfs_t;
6     type debugfs_t;
7 }
8
9 allow sysadm_t sysfs_t dir mounton;
10 allow sysadm_t debugfs_t filesystem mount;
```

Politique pour oprofile

Listing A.5– oprofile.te

```
1 policy_module(oprofile, 0.0.1)
2
3 require {
4     type sysadm_t;
5     type oprofilefs_t;
6     type sysadm_sudo_t;
7     type user_home_t;
8     type sysctl_kernel_t;
9 }
10
11 allow sysadm_t oprofilefs_t file { read open write };
12
13 allow sysadm_sudo_t sysctl_kernel_t file write;
14 allow sysadm_sudo_t user_home_t file { execute execute_no_trans };
```

Politique pour trace-cmd

Listing A.6– tracecmd.te

```
1 module tracecmd 1.0;
2
3 require {
4   type sysadm_t;
5   type staff_t;
6   type initrc_t;
7   type initrc_exec_t;
8   type setfiles_t;
9   type fs_t;
10  class file { write entrypoint getattr read open execute relabelto ioctl rename relabelfrom unlink
        setattr };
```

```
  class process { transition noatsecure siginh rlimitinh };
  class filesystem associate;
}

type script_exec_t;

role system_r types initrc_t;
role system_r types sysadm_t;
role system_r types script_exec_t;
role object_r types script_exec_t;

domain_auto_trans(initrc_t, script_exec_t, sysadm_t);
allow sysadm_t script_exec_t file entrypoint;

allow setfiles_t script_exec_t file relabelto;
allow script_exec_t fs_t filesystem associate;

allow staff_t script_exec_t file *;
allow sysadm_t script_exec_t file *;

allow initrc_t script_exec_t file ioctl;
```

Annexe B

Graphiques

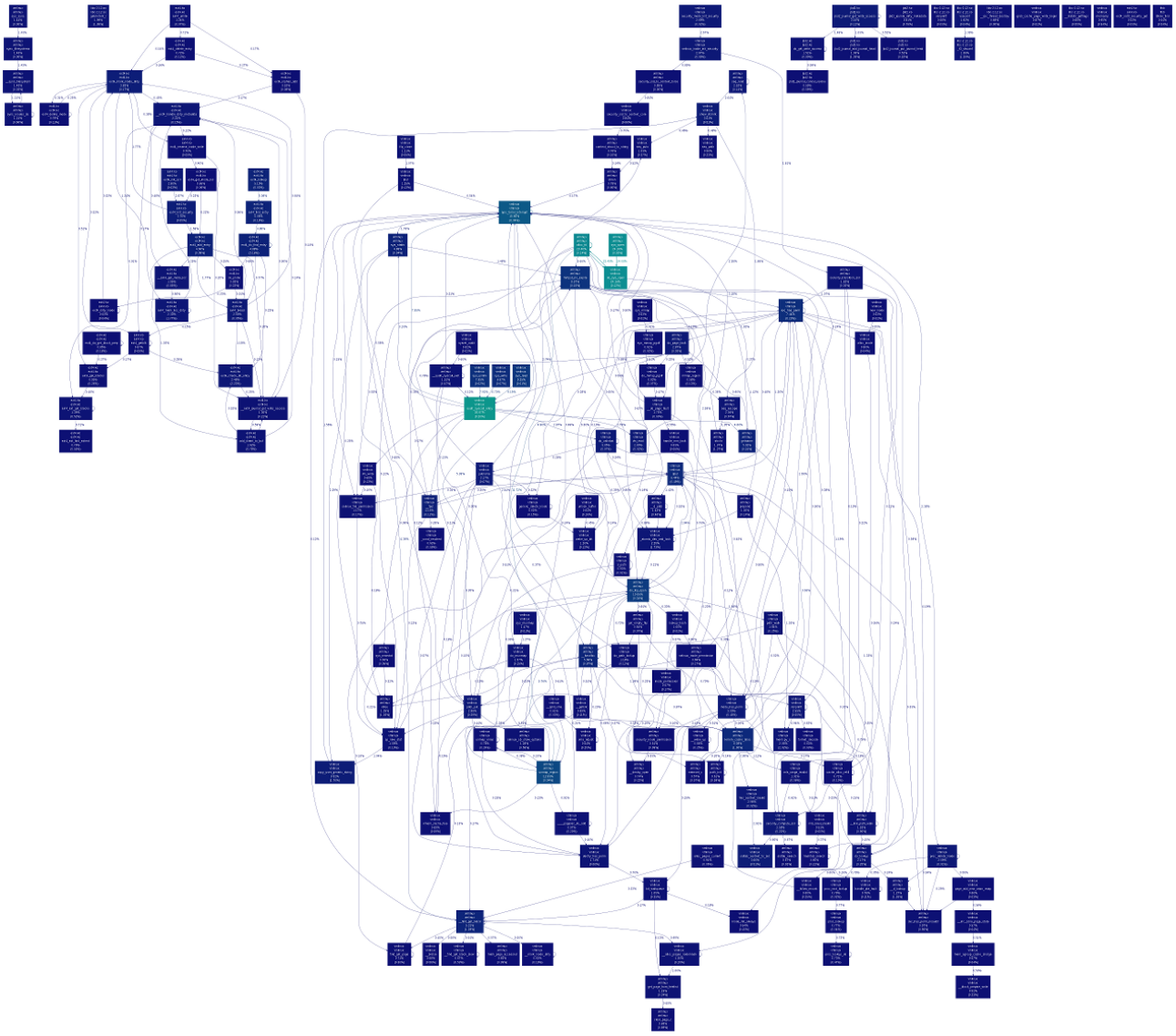


Figure B.1 – Exemple de graphe obtenu avec gprof2dot.py

Annexe C

Extraits de code

Obtention du contenu de l'*AVC*

Listing C.1– Patch pour lister le contenu de l'*AVC*

```
diff --git a/security/selinux/Kconfig b/security/selinux/Kconfig
index bca1b74..e7294f6 100644
--- a/security/selinux/Kconfig
+++ b/security/selinux/Kconfig
@@ -74,6 +74,14 @@ config SECURITY_SELINUX_AVC_STATS
    /selinux/avc/cache_stats, which may be monitored via
    tools such as avcstat.

+config SECURITY_SELINUX_AVC_DUMP
+ bool "NSA SELinux AVC dumper"
+ depends on SECURITY_SELINUX
+ default n
+ help
+   This enables users to dump the AVC when reading
+   /selinux/avc/cache_dump.
+
 config SECURITY_SELINUX_CHECKREQPROT_VALUE
   int "NSA SELinux checkreqprot default value"
   depends on SECURITY_SELINUX
diff --git a/security/selinux/avc.c b/security/selinux/avc.c
index db0fd9f..4997b53 100644
--- a/security/selinux/avc.c
+++ b/security/selinux/avc.c
@@ -328,6 +328,69 @@ static inline struct avc_node *avc_search_node(u32 ssid, u32 tsid, u16 tclass)
   return ret;
 }

+#ifdef CONFIG_SECURITY_SELINUX_AVC_DUMP
+// FIXME Move this!
+#include <linux/limits.h>
+
+/**
+ * Store an AVC dump, that has to be freed after use.
+**/
+int avc_dump(char **avc_dump_buff, ssize_t *length)
+{
+ struct avc_node *node;
+ struct hlist_head *head;
+ struct hlist_node *next;
+ int i, j, rc;
+
+ char *scontext;
+ u32 scontext_len;
+ char * tmpbuff = NULL;
+ ssize_t written = 0;
+
```

```
+ tmpbuff =  vmalloc(AVC_CACHE_SLOTS * PATH_MAX / 16 * sizeof(char));
+ if(tmpbuff == NULL){
+   return -ENOMEM;
+ }
+
+ rcu_read_lock();
+
+ for(i = 0; i < AVC_CACHE_SLOTS; ++i){
+   written += snprintf(tmpbuff + written, PATH_MAX, "Line %d :\n", i);
+   head = &avc_cache.slots[i];
+   j = 0;
+   hlist_for_each_entry_rcu(node, next, head, list) {
+     ++j;
+     written += snprintf(tmpbuff + written, 5, "%d :", j);
+     rc = security_sid_to_context(node->ae.ssid, &scontext, &scontext_len);
+     if (rc){
+       written += snprintf(tmpbuff + written, PATH_MAX, "ssid=%d", node->ae.ssid);
+     } else {
+       written += snprintf(tmpbuff + written, PATH_MAX, "scontext=%s", scontext);
+       kfree(scontext);
+     }
+
+     rc = security_sid_to_context(node->ae.tsid, &scontext, &scontext_len);
+     if (rc){
+       written += snprintf(tmpbuff + written, PATH_MAX, " tsid=%d", node->ae.tsid);
+     } else {
+       written += snprintf(tmpbuff + written, PATH_MAX, " tcontext=%s", scontext);
+       kfree(scontext);
+     }
+
+     BUG_ON(node->ae.tclass >= ARRAY_SIZE(secclass_map));
+     written += snprintf(tmpbuff + written, PATH_MAX, " tclass=%s\n", secclass_map[node->ae.tclass-1].
      name);
+   }
+ }
+
+ rcu_read_unlock();
+
+ *avc_dump_buff = tmpbuff;
+ *length = written;
+
+ return 0;
+}
+#endif
+
 /**
  * avc_lookup - Look up an AVC entry.
  * @ssid :source security identifier
diff --git a/security/selinux/include/avc.h b/security/selinux/include/avc.h
index e94e82f..63cc1b7 100644
--- a/security/selinux/include/avc.h
+++ b/security/selinux/include/avc.h
@@ -99,5 +99,9 @@ void avc_disable(void);
 DECLARE_PER_CPU(struct avc_cache_stats, avc_cache_stats);
 #endif

+#ifdef CONFIG_SECURITY_SELINUX_AVC_DUMP
+int avc_dump(char **avc_dump_buff, ssize_t *length);
+#endif
+
 #endif /* _SELINUX_AVC_H_ */

diff --git a/security/selinux/selinuxfs.c b/security/selinux/selinuxfs.c
index 264657f..2d55bdc 100644
--- a/security/selinux/selinuxfs.c
+++ b/security/selinux/selinuxfs.c
```

```
@@ -1391,6 +1391,36 @@ static const struct file_operations sel_avc_cache_stats_ops = {
 };
 #endif

+#ifdef CONFIG_SECURITY_SELINUX_AVC_DUMP
+static ssize_t sel_avc_cache_dump_read(struct file *filp, char __user *buf,
+         size_t count, loff_t *ppos)
+{
+ static char *avc_dump_buff = NULL;
+ static ssize_t length = 0;
+ ssize_t written = 0;
+
+ if(avc_dump_buff == NULL){
+   if(avc_dump(&avc_dump_buff, &length) < 0){
+     pr_warning("SELinux :avc_dump() failed!\n");
+     return 0;
+   }
+ }
+
+ written = simple_read_from_buffer(buf, count, ppos, avc_dump_buff, length);
+ if((written < count) && (avc_dump_buff != NULL)){
+   vfree(avc_dump_buff);
+   avc_dump_buff = NULL;
+   length = 0;
+ }
+
+ return written;
+}
+
+static const struct file_operations sel_avc_cache_dump_ops = {
+ .read   = sel_avc_cache_dump_read,
+};
+#endif
+
 static int sel_make_avc_files(struct dentry *dir)
 {
   int i, ret = 0;
@@ -1401,6 +1431,9 @@ static int sel_make_avc_files(struct dentry *dir)
 #ifdef CONFIG_SECURITY_SELINUX_AVC_STATS
     { "cache_stats", &sel_avc_cache_stats_ops, S_IRUGO },
 #endif
+#ifdef CONFIG_SECURITY_SELINUX_AVC_DUMP
+   { "cache_dump", &sel_avc_cache_dump_ops, S_IRUGO },
+#endif
   };

   for (i = 0; i < ARRAY_SIZE(files); i++) {
```

Paramètres pour modifier la taille de l'*AVC*

Listing C.2– Patch pour l'allocation dynamique de l'*AVC*

```
diff --git a/security/selinux/Kconfig b/security/selinux/Kconfig
index e7294f6..49e375b 100644
--- a/security/selinux/Kconfig
+++ b/security/selinux/Kconfig
@@ -82,6 +82,38 @@ config SECURITY_SELINUX_AVC_DUMP
     This enables users to dump the AVC when reading
     /selinux/avc/cache_dump.

+config SECURITY_SELINUX_BOOTPARAM_CACHE
+ bool "NSA SELinux AVC boot parameters"
+ depends on SECURITY_SELINUX
+ default n
```

```
+ help
+   This option adds two kernel parameters :selinux_avc_slots and
+   selinux_avc_threshold to choose the number of slots and the
+   threshold of the SELinux Access Vector Cache. Increasing both
+   values might improve performance for a relatively small cost in
+   memory.
+
+   If you are unsure how to answer this question, answer N.
+
+config SECURITY_SELINUX_BOOTPARAM_CACHE_SLOTS
+ int "NSA SELinux boot parameter to set default cache slots"
+ depends on SECURITY_SELINUX_BOOTPARAM_CACHE
+ range 512 32768
+ default 512
+ help
+   This option sets the default value for the number of AVC slots.
+   This should be greater than 512 which used to be the default
+   hardcoded value.
+
+config SECURITY_SELINUX_BOOTPARAM_CACHE_THRESHOLD
+ int "NSA SELinux boot parameter to set default cache threshold"
+ depends on SECURITY_SELINUX_BOOTPARAM_CACHE
+ range 512 32768
+ default 512
+ help
+   This option sets the default value for the AVC threshold. This
+   should be at least 512 which used to be the default value.
+
 config SECURITY_SELINUX_CHECKREQPROT_VALUE
   int "NSA SELinux checkreqprot default value"
   depends on SECURITY_SELINUX
diff --git a/security/selinux/avc.c b/security/selinux/avc.c
index 4997b53..60bc307 100644
--- a/security/selinux/avc.c
+++ b/security/selinux/avc.c
@@ -33,8 +33,8 @@
 #include "avc_ss.h"
 #include "classmap.h"

-#define AVC_CACHE_SLOTS      512
-#define AVC_DEF_CACHE_THRESHOLD    512
+extern int AVC_CACHE_SLOTS;
+extern int AVC_DEF_CACHE_THRESHOLD;
 #define AVC_CACHE_RECLAIM    16

 #ifdef CONFIG_SECURITY_SELINUX_AVC_STATS
@@ -61,8 +61,8 @@ struct avc_node {
 };

 struct avc_cache {
- struct hlist_head slots[AVC_CACHE_SLOTS]; /* head for avc_node->list */
- spinlock_t    slots_lock[AVC_CACHE_SLOTS]; /* lock for writes */
+ struct hlist_head *slots; /* head for avc_node->list */
+ spinlock_t    *slots_lock; /* lock for writes */
   atomic_t    lru_hint; /* LRU hint for reclaim scan */
   atomic_t    active_nodes;
   u32     latest_notif; /* latest revocation notification */
@@ -81,7 +81,7 @@ struct avc_callback_node {
 };

 /* Exported via selinufs */
-unsigned int avc_cache_threshold = AVC_DEF_CACHE_THRESHOLD;
+unsigned int avc_cache_threshold;

 #ifdef CONFIG_SECURITY_SELINUX_AVC_STATS
 DEFINE_PER_CPU(struct avc_cache_stats, avc_cache_stats) = { 0 };
```

```
@@ -172,6 +172,15 @@ void __init avc_init(void)
 {
  int i;

+ avc_cache.slots = vmalloc(sizeof(struct hlist_head) * AVC_CACHE_SLOTS);
+ avc_cache.slots_lock = vmalloc(sizeof(spinlock_t) * AVC_CACHE_SLOTS);
+
+ if((avc_cache.slots == NULL) || (avc_cache.slots_lock == NULL)){
+   panic("SELinux :Can not allocate enough memory for the AVC");
+ }
+
+ avc_cache_threshold = AVC_DEF_CACHE_THRESHOLD;
+
  for (i = 0; i < AVC_CACHE_SLOTS; i++) {
    INIT_HLIST_HEAD(&avc_cache.slots[i]);
    spin_lock_init(&avc_cache.slots_lock[i]);

diff --git a/security/selinux/hooks.c b/security/selinux/hooks.c
index ba84f93..aa024d5 100644
--- a/security/selinux/hooks.c
+++ b/security/selinux/hooks.c
@@ -125,6 +125,19 @@ __setup("selinux=", selinux_enabled_setup);
 int selinux_enabled = 1;
 #endif

+#ifdef CONFIG_SECURITY_SELINUX_BOOTPARAM_CACHE
+int AVC_CACHE_SLOTS = CONFIG_SECURITY_SELINUX_BOOTPARAM_CACHE_SLOTS;
+unsigned int AVC_DEF_CACHE_THRESHOLD = CONFIG_SECURITY_SELINUX_BOOTPARAM_CACHE_THRESHOLD;
+
+module_param(AVC_CACHE_SLOTS, int, 0444);
+MODULE_PARM_DESC(AVC_CACHE_SLOTS, "The number of slots allocated for the AVC. Must be set at boot time
    .");
+module_param(AVC_DEF_CACHE_THRESHOLD, uint, 0644);
+MODULE_PARM_DESC(AVC_DEF_CACHE_THRESHOLD, "The number of entries in the AVC. May be changed at runtime
    .");
+
+#else
+extern int AVC_CACHE_SLOTS = 512;
+extern unsigned int AVC_DEF_CACHE_THRESHOLD = 512;
+#endif

 /*
  * Minimal support for a secondary security module,
```

Modification de la fonction de hashage

Listing C.3– Patch pour utiliser la fonction *CrapWow*

```
diff --git a/security/selinux/avc.c b/security/selinux/avc.c
index 60bc307..b4e5672 100644
--- a/security/selinux/avc.c
+++ b/security/selinux/avc.c
@@ -91,9 +91,38 @@ static struct avc_cache avc_cache;
 static struct avc_callback_node *avc_callbacks;
 static struct kmem_cache *avc_node_cachep;

+
+u32 CrapWow(const u8 *key, u32 len, u32 seed)
+{
+        #define cwfold( a, b, lo, hi ) { p = (u32)(a) * (u64)(b); lo ^= (u32)p; hi ^= (u32)(p >> 32); }
+        #define cwmixa( in ) { cwfold( in, m, k, h ); }
+        #define cwmixb( in ) { cwfold( in, n, h, k ); }
+
+        const u32 m = 0x57559429, n = 0x5052acdb;
```

```
+
+        const u32 *key4 = (const u32 *)key;
+        u32 h = len, k = len + seed + n;
+        u64 p;
+
+        while ( len >= 8 ) { cwmixb(key4[0]) cwmixa(key4[1]) key4 += 2; len -= 8; }
+        if ( len >= 4 ) { cwmixb(key4[0]) key4 += 1; len -= 4; }
+        if ( len ) { cwmixa( key4[0] & ( ( (u32)1 << ( len * 8 ) ) - 1 ) ) }
+        cwmixb( h ^ (k + n) )
+        return k ^ h;
+}
+
+
 static inline int avc_hash(u32 ssid, u32 tsid, u16 tclass)
 {
- return (ssid ^ (tsid<<2) ^ (tclass<<4)) & (AVC_CACHE_SLOTS - 1);
+ u32 res = 0;
+ u8 key[10];
+
+ memcpy(key, &ssid, sizeof(u32));
+ memcpy(key+4, &tsid, sizeof(u32));
+ memcpy(key+8, &tclass, sizeof(u16));
+ res = CrapWow(key, 10, 1024);
+
+ return res & (AVC_CACHE_SLOTS - 1);
 }

 /**
```

Benchmark de l’*AVC*

Listing C.4– bench.c

```
#include <limits.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>

#include <selinux/selinux.h>

int main(int argc, char *argv[])
{
  unsigned int i;
  int res = 0;
  pid_t *pid = NULL;

  char *bench_exec;
  unsigned int iter;
  char *sec_con = calloc(1024, sizeof(char));

  char cat[8];
  size_t len = 0;

  size_t off = 0;
  size_t bench_exec_len = 0;

  security_context_t con;

  if(argc != 4){
    printf(”Usage :%s benchmark concurent_run security_context\n”, argv[0]);
    exit(EXIT_SUCCESS);
```

```
  }

  bench_exec = argv[1];
  bench_exec_len = strlen(bench_exec);
  strncpy(sec_con, argv[3], 1024);
  len = strlen(sec_con);
  if(len > 1010){
    printf("Security context too big :%zd\n", len);
    exit(EXIT_FAILURE);
  }

  iter = strtoul(argv[2], NULL, 10);
  if((iter > 999)){
    printf("Way too many concurent run!\n ");
    exit(EXIT_FAILURE);
  }
  pid = calloc(iter, sizeof(pid_t));

  if(is_selinux_mls_enabled() != 1){
    printf("No SELinux MLS on this kernel!\n");
    exit(EXIT_FAILURE);
  }

  if(getcon(&con) != 0){
    printf("Can't get current security context!");
    exit(EXIT_FAILURE);
  }
  printf("Current security context :%s\n", con);

  for(i = 0; i < iter; ++i){
    snprintf(cat, 4, "%d", i);
    sec_con[len - 1] = '\0';
    strncat(sec_con, cat, 4);
    printf("%d :launching %s, with security context :%s\n", i, bench_exec, sec_con);
    pid[i] = fork();
    if(pid[i] == 0){
      res = setexeccon(sec_con);
      if(res < 0){
        printf("Can't setexeccon :%s", sec_con);
        exit(EXIT_FAILURE);
      }
      off = 0;
      while((off < len) && (bench_exec[len - off] != '/')){
        ++off;
      }
      if(off != len){
        --off;
      }
      res = execl(bench_exec, bench_exec + (len - off), NULL);
      printf("Failed to launch benchmark :%d, with context :%s\n", res, sec_con);
      exit(EXIT_FAILURE);
    }
  }

  for(i = 0; i < iter; ++i){
    waitpid(pid[i], NULL, 0);
  }

  freecon(con);
  free(pid);

  return 0;
}
```

Listing C.5– Options de compilation pour GCC

```
$ gcc -o bench bench.c -Wall -Wextra -lselinux
```

Listing C.6– Exemples d'exécution

```
$ ./bench cmd.sh 10 staff_u staff_r staff_t s0-s0 c0
```

Listing C.7– Exemple de fichier cmd.sh

```
#!/bin/bash
ls -alR /bin /boot /dev /etc /home /lib /lib64 /lost+found /sbin /srv /usr &> /dev/null
```

Appendice

Note à propos du logo SELinux sur la première page

This image is licensed under the Creative Commons ShareAlike 2.5 license : `http://people.redhat.com/duffy/artwork/selinux-penguin.svg`