HAL
open science

Sécurité des systèmes d'exploitation répartis : architecture décentralisée de méta-politique pour l'administration du contrôle d'accès obligatoire.

Mathieu Blanc

To cite this version:

Mathieu Blanc. Sécurité des systèmes d'exploitation répartis : architecture décentralisée de méta-politique pour l'administration du contrôle d'accès obligatoire.. Réseaux et télécommunications [cs.NI]. Université d'Orléans, 2006. Français. NNT : . tel-00460610

HAL Id: tel-00460610

https://theses.hal.science/tel-00460610v1

Submitted on 1 Mar 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

THÈSE

PRÉSENTÉE

A L'UNIVERSITÉ D'ORLÉANS

POUR OBTENIR LE GRADE DE

DOCTEUR DE L'UNIVERSITÉ D'ORLÉANS

Discipline : **Informatique**

PAR

Mathieu BLANC

Sécurité des systèmes d'exploitation répartis : architecture décentralisée de méta-politique pour l'administration du contrôle d'accès obligatoire.

Soutenue le : **19 décembre 2006**

MEMBRES DU JURY

Gaétan Hains	Professeur	Université Paris 12	**Président du jury**
Laurent Oudot	Ingénieur Chercheur	CEA	**Examinateurs**
Emmanuel Bouillon	Ingénieur Chercheur	CEA	
Jean-Michel Couvreur	Professeur	Université d'Orléans	
Michel Cukier	Professeur Assistant	University of Maryland	**Rapporteur**
Christian Toinard	Professeur	ENSI de Bourges	**Directeur**

RAPPORTEURS

Michel Cukier	Professeur Assistant	University of Maryland
Michaël Rusinowitch	Directeur de Recherche	INRIA-LORIA

Remerciements

Cette thèse a fait l'objet d'un cofinancement entre la Région Centre / Université d'Orléans / Laboratoire d'Informatique Fondamentale d'Orléans (LIFO) et le Commissariat à l'Energie Atomique (CEA).

Les travaux se sont déroulés, d'un part, au CEA dans la Cellule Technique de Sécurité Informatique (DAM/DIF/DSSI/CTSI), et d'autre part, dans le projet Sécurité et Distribution des Systèmes (SDS) du LIFO situé à l'ENSI de Bourges.

Je tiens à remercier tous les gens qui ont participé, de près ou de loin, aux 3 ans et quelques mois durant lesquels j'ai mené mes travaux de thèse, et rédigé ce mémoire. Tout d'abord, je tiens à remercier Christian Toinard et Laurent Oudot, qui ont été les instigateurs de cette thèse.

Ensuite, je remercie le CEA, et plus particulièrement l'ensemble des membres de l'équipe de la CTSI, au sein de laquelle j'ai effectué ma thèse. J'ai eu droit à un cadre de travail excellent, entouré d'une équipe extrêmement compétente en sécurité informatiques, et ayant accès à des moyens dont je n'aurai pu disposer ailleurs.

Je remercie aussi l'ensemble du LIFO, membres actuels et anciens, qui ont accompli l'encadrement scientifique de ma thèse. Notamment, l'équipe de Bourges a été très impliquée dans cette thèse puisque mon directeur, Christian Toinard, en fait partie. Avec lui, je souhaite témoigner ma gratitude envers Patrice Clémente, Jean-François Lalande et Sébastien Chabrier, dont la contribution à mes travaux a été essentielle, en particulier pendant la rédaction de ce mémoire. Je remercie également Christel Vrain, l'actuelle directrice du LIFO, pour son soutien.

Par ailleurs, je souhaite remercier Michael Rusinowitch et Michel Cukier, mes rapporteurs, Gaétan Hains, président de mon jury de thèse, qui a également contribué à l'avancement de la partie Vérification de ma thèse avec Pierre Courtieu, Jean-Michel Couvreur, membre du LIFO et de mon jury de thèse. Je remercie mes copains de la promo 2002 de l'ENSEIRB, qui m'ont aussi offert leur soutien pendant ces années de thèse, n'hésitant pas à me rappeler qu'il fallait envisager de la terminer ! Merci à tous :)

Enfin je remercie toute ma famille, mes parents et mon frère, ils ont vécu cette thèse presque autant que moi !

Et merci à tous ceux qui s'attaqueront à ce mémoire ! Je vous souhaite une bonne lecture.

Mathieu

Table des matières

1 Introduction **15**
1.1 Contexte de l'étude 15
1.2 Historique du contrôle d'accès 16
1.3 Objectifs de la thèse 18
1.4 Apports de la thèse 19
1.5 Plan du mémoire 20

2 Etat de l'art **23**
2.1 Modèles théoriques de contrôle d'accès 23
2.1.1 DAC 25
2.1.2 MAC 26
2.1.3 RBAC 32
2.2 Implantations système du contrôle d'accès 34
2.2.1 Les différentes implantations pour Linux 34
2.2.2 SELinux 36
2.2.3 grsecurity 38
2.3 Langages d'expression de politiques de sécurité 40
2.3.1 Modèles multi-domaines 41
2.3.2 Modèles multi-politiques 43
2.3.3 Modèles méta-politiques 44
2.4 Conclusion 45

3 Discussion **47**
3.1 Discussion des modèles de contrôle d'accès 47
3.1.1 DAC 48
3.1.2 MAC 48
3.1.3 RBAC 49
3.2 Implantations système 50
3.2.1 Medusa 50
3.2.2 LIDS 50
3.2.3 RSBAC 51
3.2.4 SELinux 51
3.2.5 grsecurity 52
3.3 Bilan sur le contrôle d'accès 52
3.4 Discussion des modèles d'expression de politiques 53
3.4.1 Modèles multi-domaines 53

3.4.2 Modèles multi-politiques 54
3.4.3 Modèles méta-politiques 55
3.4.4 Bilan sur les modèles d'expression de politique 55
3.5 Synthèse et orientations de notre architecture 56
3.6 Conclusion 57

4 Spécification 59
4.1 Principe de la Méta-Politique 59
4.1.1 Architecture distribuée 60
4.1.2 Deux niveaux de contrôle 61
4.1.3 Agent de mise à jour 61
4.1.4 Politique de protection 62
4.1.5 Politique de modification 63
4.1.6 Traducteur de politiques 63
4.2 Propriétés de l'architecture 63
4.2.1 Méta-politique 64
4.2.2 Evolution locale des politiques de sécurité 70
4.2.3 Intégration du MAC 70
4.2.4 Performance 71
4.3 Principe algorithmique de l'architecture Méta-Politique 72
4.3.1 Vue globale 72
4.3.2 Démarrage 72
4.3.3 Requête de mise à jour 73
4.4 Conclusion 73

5 Formalisation 75
5.1 Définition générale du modèle de Méta-Politique 76
5.1.1 Contextes de Sécurité 76
5.1.2 Vecteurs d'Interaction 77
5.1.3 Langage Neutre de Politique 77
5.1.4 Langage Neutre de Contraintes 78
5.1.5 Projection du langage neutre de politique 82
5.2 Définition de la Méta-Politique de contrôle d'accès 83
5.2.1 Modèle de contrôle d'accès 83
5.2.2 Langage neutre de contrôle d'accès 87
5.2.3 Projection du langage neutre de contrôle d'accès 90
5.2.4 Exemple d'application à l'administration du contrôle d'accès 90
5.3 Application de la Méta-Politique à la détection d'intrusion 93
5.3.1 Architecture multi-niveaux et multi-agents pour le contrôle d'accès et la détection d'intrusion 94
5.3.2 Méta-Politique de détection d'intrusion 95
5.3.3 Coopération du contrôle d'accès et de la détection d'intrusion 95
5.4 Conclusion 96

6 Vérification **99**
6.1 Objectifs . . . 100
6.2 Détection de flux d'information dans les politiques statiques . . . 101
6.3 Influence de la Méta-Politique sur la vérification . . . 101
6.3.1 Cas extrême 0 . . . 102
6.3.2 Cas extrême 1 . . . 102
6.3.3 Canevas générique . . . 103
6.4 Méthode générale de vérification d'une Méta-Politique . . . 108
6.4.1 Problème d'intersection des expressions régulières . . . 109
6.4.2 Deuxième version de l'algorithme de contraction . . . 110
6.5 Evaluation de la complexité de l'algorithme . . . 110
6.6 Conclusion . . . 112

7 Implantation **113**
7.1 Implantation de l'agent de mise à jour . . . 113
7.1.1 Chargement de la Méta-Politique . . . 114
7.1.2 Réception des requêtes de mise à jour . . . 115
7.1.3 Traitement des requêtes de mise à jour . . . 115
7.1.4 Traduction vers le mécanisme de contrôle d'accès . . . 116
7.2 Implantation des langages de politique . . . 117
7.2.1 Format XML pour le langage neutre de politique . . . 117
7.2.2 Format XML pour le fichier de Méta-Politique . . . 118
7.2.3 Format XML pour les requêtes de mise à jour . . . 121
7.3 Implantation des modules de traduction . . . 122
7.3.1 Traduction de la politique neutre pour SELinux . . . 123
7.3.2 Traduction de la politique neutre pour grsecurity . . . 124
7.4 Conclusion . . . 125

8 Conclusion **127**

9 Bibliographie **131**

A Formats XML **137**
A.1 Format de méta-politique . . . 137
A.2 Format de politique neutre . . . 138
A.3 Format de politique neutre . . . 138

B Exemple de Méta-Politique en XML **141**

Table des figures

2.1 Règle d'accès. 24
2.2 Cas particulier. 24
2.3 No Read Up et No Write Down. 28
2.4 No Read Down et No Write Up. 29
2.5 Le modèle RBAC_0. 32
2.6 Les hiérarchies de rôles dans RBAC_1. 33

3.1 Coopération des modèles RBAC, MAC et DAC. 52

4.1 Schéma de l'architecture Méta-Politique. 60
4.2 Répartition des agents sur les nœuds. 62
4.3 Garantie de propriétés globales de sécurité sur un cluster. 66
4.4 Garantie de propriétés globales de sécurité pour un fournisseur. 67
4.5 Evolution locale des politiques de sécurité. 68
4.6 Evolution locale de la politique lors d'une panne de réseau. 69
4.7 Schéma interne de l'agent de mise à jour. 72

5.1 Evolution locale des politiques de sécurité 76
5.2 Moniteur de Référence. 84
5.3 Moniteur de Référence et Méta-politique 85
5.4 Graphe d'interaction de la politique. 90
5.5 Architecture multi-niveaux et multi-agents 94

6.1 Exemple de graphe représentant une politique. 101
6.2 Politique statique ne comportant pas de flux d'information entre A et B. 102
6.3 Exemples de flux d'informations entre A et B. 103
6.4 Graphe pour une politique initiale $\mathcal{LR}_{\mathcal{AC}}$. 104
6.5 Regroupement des nœuds correspondant aux règles de la politique de modification. . 105
6.6 Graphe résultant de l'algorithme de contraction. 105
6.7 Exemple de contraction de graphe pour la Méta-Politique 2. 107
6.8 Exemple de contraction de graphe pour la Méta-Politique 6.4. 108
6.9 Exemple de deux graphes de politiques utilisant deux jeux d'expressions régulières. . 109
6.10 Contraction du premier jeu d'expressions régulières. 110
6.11 Contraction du second jeu d'expressions régulières. 110
6.12 Graphe contracté global. 111
6.13 Graphe contracté global utilisant le deuxième algorithme. 111

7.1 Structure interne de l'agent. 114
7.2 Interactions avec la classe MetaPolicyLoader. 115
7.3 Interactions avec la classe QueryReceiver. 115
7.4 Interactions de la classe QueryProcessor. 116
7.5 Interactions avec la classe PolicyTranslator. 117
7.6 DTD pour la politique locale $\mathcal{LR}_{\mathcal{AC}}$ 118
7.7 DTD pour la méta-politique $\mathcal{MP}_{\mathcal{AC}}$. 120
7.8 Requête de mise à jour en XML. 121

Glossaire des sigles

ACL *Access Control List* : Liste de contrôle d'accès, page 24.

ADF *Access Control Decision Facility* : composant de prise de décision dans l'architecture GFAC, page 35.

AEF *Access Control Enforcement Facility* : composant d'application du contrôle d'accès dans GFAC, page 35.

AVC *Access Vector Cache* : Composant de SELinux, page 37.

BIBA Modèle d'intégrité conçu par K. J. Biba, page 29.

BLP Modèle de confidentialité Bell-LaPadula, nommé d'après ses auteurs D. E. Bell et L. J. La Padula, page 27.

DAC *Discretionary Access Control* : Contrôle d'accès discrétionnaire, page 25.

DOM *Document Object Model* : Modèle de représentation de documents sous forme d'objet., page 114.

DTD *Document Type Definition* : Définition de type de document, utilisé dans XML pour la création de formats de fichiers, page 113.

DTE *Domain and Type Enforcement* : Modèle de protection associant des domaines aux sujets et des types aux objets, page 31.

DTOS *Distributed Trusted Operating System* : système d'exploitation sécurisé développé par la NSA, page 36.

DTP *Denial Takes Precedence* : Modèle de résolution de conflits où l'autorisation négative est prioritaire, page 44.

GFAC *Generalized Framework for Access Control* : Architecture générique de contrôle d'accès, page 35.

HPC *High Performance Computing* : Calcul hautes performances, page 18.

HRU Modèle théorique de DAC établi par Harrison, Ruzzo et Ullman, page 25.

IDS *Intrusion Detection System* : Système de détection d'intrusions, page 100.

IPS *Intrusion Prevention System* : Système de prévention des intrusions, page 100.

IPSEC *Internet Protocol Security* : Sécurité du protocole Internet (IP), page 49.

ITSEC *Information Technology Security Evaluation Criteria* : Critères d'évaluation de la sécurité des systèmes d'information, définis par un comité de pays européens, page 23.

LGI *Law Governed Interaction* : Modèle d'administration de politiques de sécurité, page 42.

LIDS *Linux Intrusion Detection System* : Mécanisme de contrôle d'accès pour le noyau Linux, page 36.

LSM *Linux Security Modules* : Ensemble de fonction pour l'implantation de modules de sécurité dans le noyau Linux, page 36.

MAC *Mandatory Access Control* ou contrôle d'accès obligatoire, page 27.

MLS *MultiLevel Security* : Modèle de sécurité multi-niveaux, page 28.

MTAM Modèle *Monotonic Typed Access Matrix* obtenu en ôtant les opérations de suppressions du modèle TAM, page 26.

NSA *National Security Agency* : Agence de sécurité nationale américaine, à l'origine du projet SELinux, page 37.

OR-BAC *Organization-Based Access Control* : Contrôle d'accès basé sur les organisations, page 43.

PBM *Policy-Based Management* : principes d'administration de systèmes d'information par politiques, page 40.

PIGA *Policy Interaction Graph Analysis* : Analyse du graphe d'interaction de la politique, page 101.

PTP *Permission Takes Precedence* : Modèle de résolution de conflits où l'autorisation positive est prioritaire, page 44.

RBAC *Role-Based Access Control* : Contrôle d'accès à base de rôles, page 32.

RSBAC *Rule Set-Based Access Control* : Mécanisme de contrôle d'accès pour le noyau Linux, page 35.

SELINUX *Security-Enhanced Linux* : Mécanisme de contrôle d'accès pour le noyau Linux, page 36.

SI Système d'information, page 23.

SLA *Service-Level Agreement* : Accord sur la fourniture d'un service entre domaines d'administration, page 45.

TAM Modèle *Typed Access Matrix* conçu par Ravi S. Sandhu, page 26.

TCP *Transmission Control Protocol* : protocole de transmission réseau en mode connecté, page 39.

TCSEC *Trusted Computer System Evaluation Criteria* : Critères d'évaluation de la sécurité des systèmes informatiques, édités par le Ministère de la Défense américain. Aussi appelé *Orange Book*, page 23.

TE *Type Enforcement* : Version allégée de DTE, par exclusion de la notion de *domaine*, page 50.

TOS *Trusted Operating Systems* : système d'exploitation de confiance, page 70.

TPE *Trusted Path Execution* : mécanisme de sécurité intégré à grsecurity, page 38.

UDP *User Datagram Protocol* : protocole de transmission réseau en mode non connecté, page 39.

UID *User IDentifier* : identifiant numérique unique à chaque utilisateur dans les systèmes d'exploitation de type Unix, page 38.

XACML *eXtensible Access Control Markup Language* : Langage à balise extensible pour la configuration du contrôle d'accès, page 128.

XML *eXtended Markup Language* : Langage de balises étendu, page 113.

Chapitre 1

Introduction

1.1 Contexte de l'étude

Dans la société actuelle, nous avons de plus en plus affaire à l'informatique. L'administration, les entreprises, les hôpitaux, les citoyens, tous les acteurs de la société utilisent aujourd'hui massivement les systèmes informatiques pour gérer leurs activités, leurs données, leurs employés, leurs clients, et même leur argent. Le vingtième siècle a vu la naissance et la prolifération des ordinateurs, et le vingt-et-unième siècle voit l'incursion de l'informatique dans les téléphones, les appareils photos, et même dans les maisons. En parallèle, la multiplication des terminaux informatiques a participé à l'explosion des besoins de communications. Preuve en est faite, l'informatique est aujourd'hui présente dans la plupart des actions de notre vie.

Qu'il s'agisse de commerce électronique, de déclaration d'impôts en ligne, d'échanges d'information entre entreprise, ou de calcul hautes performances (HPC), les architectures réparties de systèmes informatiques sont très largement répandues. Elles supportent des centaines, voire des milliers d'utilisateurs. Certaines sont capables d'effectuer des millions de transactions par jour, d'autres de modéliser en peu de temps des systèmes physiques dont le calcul était impensable il y a seulement quelques années. Malheureusement, ces systèmes répartis intéressent une autre sorte de public, les pirates informatiques. Ils y voient de grandes sources de puissance de calcul et de trafic réseau. De plus, ces architectures réparties impliquent une concentration des moyens informatiques, donnant aux pirates les moyens de s'approprier de grandes quantités de ressources informatiques avec peu d'efforts.

C'est pourquoi la sécurité des systèmes d'exploitation répartis est cruciale. Par exemple, une attaque réussie contre des systèmes répartis à grande échelle, comme les grilles de calcul partagées entre plusieurs universités, fournit à un pirate l'accès à plusieurs milliers de machines en un seul coup. Plus grave encore, les systèmes répartis ont souvent des accès privilégiés entre eux, et après avoir réussi à s'introduire dans l'un d'eux, un pirate sera en mesure de rebondir vers d'autres réseaux d'universités ou d'entreprises.

Actuellement, la sécurité de ces grands systèmes répartis est l'objet de multiples études. Le contrôle d'accès obligatoire est un des remparts les plus efficaces à déployer dans les systèmes d'exploitation répartis. Toutefois, l'administration de ces mécanismes de sécurité à l'échelle de grands systèmes répartis est très délicate. De plus, le sujet particulier de l'administration de contrôle d'accès obligatoire en environnement réparti reste en grande partie à défricher. C'est pourquoi nous proposons, dans le cadre de cette étude, de définir, concevoir, modéliser et implanter une nouvelle architecture d'administration de politiques de contrôle d'accès pour des systèmes répartis.

1.2 Historique du contrôle d'accès

La sécurité des systèmes d'information comprend trois étapes essentielles : la prévention, la protection et la détection. Dans le cadre de la protection [TCSEC 1985], le **contrôle d'accès** est un composant central. En effet, celui-ci est directement responsable des actions accordées ou refusées aux utilisateurs, et donc de la manipulation des données présentes dans le système d'information. Par conséquent, dans le cadre de la politique de sécurité de ce système d'information, les mécanismes de contrôle d'accès doivent non seulement être capables d'autoriser les actions légitimes prévues, mais également de refuser et bloquer celles qui sont considérées illégitimes.

En matière de politique de sécurité, il est bon de préciser la terminologie. Le système d'information représente l'ensemble des utilisateurs, machines et logiciels amenés à interagir. Le contrôle d'accès est un ensemble de règles régissant les permissions des utilisateurs sur les machines et les logiciels. Les règles considérées sont des lois d'accès de la part de **sujets** (entités actives du système d'information) sur des **objets** (entités passives). Chacun de ces accès est contrôlé par le **Moniteur de Références** [Anderson 1972], responsable de l'application de la politique de sécurité. Trois composants entrent en jeu dans le fonctionnement de ce moniteur : le modèle de spécification de la politique de sécurité, le modèle de contrôle d'accès, et enfin le mécanisme d'application des décisions d'accès. Ces composants sont relativement indépendants, ainsi une même politique de sécurité peut être appliquée avec différents modèles de contrôle d'accès. Cependant, les garanties offertes par les différents composants peuvent varier d'un modèle a un autre.

Ainsi, différents modèles de contrôle d'accès ont été développés. Dans un premier temps, le modèle discrétionnaire (**DAC** ou *Discretionary Access Control*) [Lampson 1971] a été employé sur l'ensemble des systèmes d'exploitation multi-utilisateurs. Ce modèle délègue aux utilisateurs l'attribution des permissions d'accès sur les ressources qui leur appartiennent, notamment les fichiers. De plus, il existe un utilisateur qui a tout pouvoir sur le système. La sécurité de ce modèle a semblé longtemps satisfaisante, tant que l'accès au super-utilisateur était contrôlable. Or la présence d'erreurs de programmation et de vulnérabilités dans les applications, ainsi que l'apparition de techniques d'exploitation de ces failles, ont induit de nouveaux moyens d'accéder aux privilèges du super-utilisateur. Dès lors, l'existence de la moindre faille dans un service fourni par un ordinateur signifie que tout utilisateur peut outrepasser le DAC.

Dans un second temps, de nouveaux modèles ont été conçus dans le cadre militaire, principalement à cause du fait que le DAC ne répond pas à leurs exigences. En effet, dans ce contexte, ce n'est pas le propriétaire d'une ressource qui décide des permissions d'accès qui y sont associées, mais plutôt sa classification, et le besoin d'en connaître. Il s'agit là d'un des principes fondamentaux du contrôle d'accès obligatoire ou *Mandatory Access Control* (**MAC**) [Bell et La Padula 1973], qui a pour objectif de déléguer le contrôle d'accès à une entité tierce. Ce type de contrôle d'accès repose sur des labels, c'est à dire des identifiants désignant un **contexte de sécurité** sujet ou objet. Ils peuvent contenir des informations concernant à la fois le niveau de confidentialité (habilitation d'un sujet, classification d'un objet), et également des regroupement non-hiérarchiques comme les différents départements d'une organisation. Le moniteur de références utilise les contextes de sécurité pour constituer une matrice de protection indicée par ces identifiants et donnant les permissions entre le sujet et l'objet considérés. Il s'agit d'une extension des listes de contrôle d'accès, présentes dans les systèmes DAC. Cependant avec le MAC, la granularité des contextes peut être plus fine et surtout cette matrice n'est pas modifiable par les utilisateurs normaux. Cette matrice constitue une politique de sécurité appliquée de façon obligatoire indépendamment des utilisateurs.

Dès lors, l'intégration du MAC dans un système d'information protège des risques liés aux DAC, c'est-à-dire d'une part les erreurs dans l'attribution des permissions par les utilisateurs, et d'autre part

la possibilité pour le super-utilisateur d'outrepasser le DAC. En effet, les règles d'accès définies par rapport aux labels sont absolues et ne peuvent en aucun cas être outrepassées. On obtient ainsi un contrôle d'accès beaucoup plus robuste. Toutefois, deux problèmes subsistent : 1) comment intégrer un tel type de contrôle d'accès dans un système d'exploitation, et 2) quelles précautions prendre pour administrer un tel modèle de contrôle d'accès sans remettre en cause les garanties de sécurité qu'il offre ?

Concernant le premier point, des modèles de contrôle d'accès de plus haut niveau ont été conçus pour répondre aux besoins des systèmes d'exploitation. D'une part, il existe des modèles purement militaires. Le modèle BLP, et son extension MLS, exploitent les labels pour la gestion de niveaux de confidentialité, et possèdent des lois qui garantissent le respect de ces niveaux. Le modèle BIBA est conçu pour des systèmes à forte intégrité. D'autre part, on trouve des modèles issus de la sécurité des bases de données. Ainsi le modèle Role-Based Access Control (**RBAC**) [Sandhu *et al.* 1996] fait le lien entre les utilisateurs et l'ensemble des opérations possibles sur une base de données via la gestion des rôles. Un tel rôle correspond à une activité particulière de l'utilisateur, et donc un certain ensemble d'actions. Enfin, des modèles de contrôle d'accès spécifiques aux systèmes d'exploitation ont également été conçus, notamment Domain and Type Enforcement (**DTE**) [Boebert et Kain 1985]. Il définit des classes d'équivalence sur les sujets, appelées domaines, et sur les objets, appelées types, et des lois d'interaction entre domaines et types.

Différentes implémentations de MAC ont été réalisées dans la plupart des systèmes d'exploitation disponibles aujourd'hui. Il est tout particulièrement intéressant d'observer ce qui a été développé dans le cadre de l'Open Source, en effet l'accès au code source facilite la compréhension de l'implémentation. Au sujet du noyau Linux, plusieurs implémentations de MAC existent, avec différents modèles de politique de sécurité possibles. Parmi les plus intéressantes sont celles qui intègrent RBAC, il s'agit de RSBAC [Ott 2001], grsecurity [Spengler 2002] et SELINUX [Loscocco et Smalley 2001].

Concernant l'administration du contrôle d'accès, le point crucial est que l'existence d'un mécanisme d'administration ne remettre pas en cause les garanties de sécurité. En effet, un tel mécanisme autoriserait théoriquement un administrateur à modifier tout ou partie de la configuration du contrôle d'accès, lui offrant ainsi le moyen d'outrepasser totalement le MAC. On retrouverait alors le défaut du DAC. Au contraire, le mécanisme d'administration du MAC doit permettre de définir précisément pour chaque élément de la politique de sécurité, qui a le droit d'ajouter, de modifier ou de supprimer des lois. Il est donc nécessaire de définir des modèles d'administration de politiques de sécurité, afin de s'assurer qu'il n'existe pas de faiblesse à ce niveau.

La conception des modèles d'administration par politiques de sécurité a commencé bien après celle de BLP ou BIBA, avec l'article [Moffett *et al.* 1990]. Leurs objectifs sont de deux types : 1) définir des politiques d'autorisation, qui spécifient les droits des administrateurs, et 2) des politiques d'obligations, qui indiquent les devoirs des administrateurs. Les premiers modèles ont pris en compte l'aspect réparti des systèmes d'information, en intégrant dans ces modèles les notions de domaines d'applicabilité de politiques. Il s'agit des modèles **multi-domaines**. Ainsi sont apparues les possibilités de conflit lorsque les champs d'application de différentes politiques se recoupent. Dès lors, les modèles d'administration destinés à gérer différents entités organisationnelles ont intégré la notion de gestion de conflit. Les études se sont orientées vers des aspects **multi-politiques** [Hosmer 1993], c'est-à-dire la possibilité de faire cohabiter différentes politiques de sécurité, et d'en calculer la résultante pour un élément donné du système d'information.

En parallèle à ces études, le raffinement de la notion d'environnements multi-domaines et multi-politiques a mené à la conception de **méta-politiques** [Hosmer 1992b]. Il s'agit de "politiques de politiques", en d'autres termes de langages de politiques de sécurité de haut niveau, conçus pour spécifier comment et quand les politiques doivent être appliqués, composés ou raffinées.

1.3 Objectifs de la thèse

Ce travail de thèse a pour but de définir une nouvelle architecture de contrôle d'accès, répondant à des objectifs précis.

Tout d'abord, notre architecture vise la *sécurisation des systèmes répartis* à large échelle. En particulier, les problématiques de sécurité dans les environnements HPC (clusters de calcul), dans les grilles, dans les applications distribuées et dans les réseaux des fournisseurs d'accès et de services ne disposent pas aujourd'hui de solutions de contrôle d'accès satisfaisantes.

L'*intégration de modèles et mécanismes existants* constituera un second objectif. Pour obtenir un système de protection sûr, il est nécessaire de disposer d'un système d'exploitation de confiance (*Trusted Operating System*) tel que défini dans [TCSEC 1985]. Ceci implique de disposer de mécanismes de sécurité obligatoire (MAC). Concernant le noyau Linux, on dispose de nombreuses solutions comme SELINUX, grsecurity et RSBAC. Nous souhaitons donc réutiliser ces solutions, et éviter de développer un nouveau mécanisme de contrôle d'accès pour le noyau Linux. En outre, l'intégration de mécanismes existants assure une meilleure évolutivité, et un plus grand champ d'application. De plus, nous intégrerons le concept de méta-politique, et nous définirons un langage neutre, autorisant ainsi le *support de systèmes hétérogènes*.

Ensuite, notre architecture aura pour objectif de fournir un *système d'administration décentralisé*, suivant quatre axes :

- Assurer que l'administration puisse être menée sur des systèmes à large échelle en minimisant le besoin de communication et de connexion. On peut ainsi imaginer que pour un fournisseur d'accès et de services, les politiques puissent évoluer sans connexion réseau (parce que justement modifier une politique doit pouvoir se faire dans un contexte sécurisé, sans aucune ouverture sur le réseau durant le temps de mise de la politique).
- Empêcher un administrateur d'introduire des règles violant les objectifs de sécurité. En effet, sur les systèmes à base de MAC, l'administrateur peut modifier la politique, et donc introduire des règles abusives ou mettant en danger le système.
- Faire que sur chaque noeud du système réparti, l'administration puisse être locale. Cet objectif provient de la nécessité d'avoir plusieurs administrateurs pour les différents noeuds, sur les systèmes répartis à large échelle. En effet, vu le nombre de noeuds et la variété des besoins applicatifs sur les différents noeuds, il est difficile, voire inutile, de toujours effectuer des modifications globales.
- Autoriser une administration automatisée du système réparti. En effet, il est difficile pour l'administrateur de définir des modifications sûres de la politique. C'est pourquoi il est nécessaire de disposer d'outils automatisant le processus de calcul de nouvelles règles.

La *tolérance aux pannes* est un objectif très important. Deux points sont pris en compte :

- La garantie de la sécurité du système réparti ne doit pas être remise en cause parce qu'un noeud est isolé, il faut donc que le noeud possède les moyens de mettre à jour sa configuration de façon locale ;
- La présence d'un composant central est souvent un point faible dans les architecture de sécurité, comme par exemple dans le système d'authentification Kerberos où une attaque sur le KDC peut en défaire toute la sécurité [Linux Documentation Project 2004].

Les *performances de l'architecture* constituent un autre objectif. En effet, si notre architecture est déployée dans des environnements HPC, l'aspect performance est crucial. Nous devons donc, d'une part, disposer d'une infrastructure légère, à base d'agents, et d'autre part, dépendre le moins possible des communications réseau, qui induisent des latences non négligeables. En effet, tout délai dans l'application de mise à jour des politiques de sécurité signifie que le système reste plus longtemps

vulnérable, ou inutilisable. De plus, nous ne souhaitons pas implanter de nouveaux mécanismes de contrôle d'accès. Les mécanismes réutilisés doivent donc être optimisés, dans le but d'avoir un impact faible sur les performances.

La *garantie des objectifs de sécurité* est un élément important. Nous le mettons en évidence puisque l'écriture de politique de sécurité est difficile. Pour le cas d'un système réparti où les administrateurs doivent ajouter dynamiquement des applications, les politiques sont dynamiques et le problème de la vérification de politiques ce problème est plus ardu dans ce cas. La difficulté est mise en évidence par l'article fondateur [Harrison *et al.* 1976]. Les auteurs montrent que pour des politiques dynamiques où l'on peut créer de nouveaux contextes (pour les nouvelles applications ou les nouveaux objets) et modifier les droits, c'est-à-dire modifier la matrice de protection, le problème de sûreté de la protection est indécidable. Ce résultat est établi dans le cas du DAC et pour une propriété de sécurité qui vise à assurer que l'ajout d'une règle particulière n'est pas possible avec une matrice de protection initiale donnée. Hors des propriétés de sécurité plus opérationnelles sont nécessaires notamment garantir l'absence de flux d'information illégaux dans les politiques.

1.4 Apports de la thèse

Ce travail de thèse propose une approche système, alors que l'essentiel des autres solutions concernent plutôt des aspects organisationnels ou de plus haut niveau, c'est-à-dire associées à des besoins spécifiques, par exemple militaires. Ici nous définissons une nouvelle solution d'administration système dans laquelle on veut garantir des propriétés de sécurité globales à tout un système réparti.

Cela écarte de fait les approches méta-politiques ou multi-politiques traditionnelles qui ne permettent pas d'empêcher les conflits pouvant apparaître entre les politiques des différents noeuds. Nous proposons donc une nouvelle approche de méta-politique qui, contrairement aux solutions classiques, prévient les conflits. Cette propriété est obtenue grâce à des contraintes d'évolutions de la politique.

La méta-politique est donc constituée de deux parties : une politique de protection initiale, plus une politique de modification. La politique de modification contient les objectifs de sécurité. La politique initiale sert à établir la politique de protection locale au démarrage du système. Ensuite, des demandes de modification de cette politique peuvent être traitées. Ces requêtes permettent de modifier la politique locale lorsqu'elles sont conformes à la politique de modification.

Cela signifie que la méta-politique change peu souvent. Dans la pratique, elle est associée à une installation de système et permet des évolutions des politiques locales du noeud. C'est propice à un système à large échelle puisque le déploiement de la méta-politique n'a pas besoin d'être renouvelé fréquemment et donc les problèmes de distribution de cette méta-politique sont moindres. De plus, les modifications de la politique peuvent être effectuées entièrement localement sans nécessiter aucune connexion réseau. Ces éléments répondent au premier besoin d'administration à large échelle que nous avons énoncé précédemment.

Contrairement à un système MAC en mode autonome (standalone) comme SELinux, notre méta-politique empêche qu'un administrateur viole les objectifs de sécurité. En effet, ceux-ci sont matérialisés par la politique de modification. Comme un agent contrôle que les requêtes respectent la politique de modification, toutes les évolutions de la politique locale sont obligatoirement conformes aux objectifs de sécurité. Nous répondons ainsi au second besoin d'administration.

La méta-politique est déployée sur l'ensemble du système réparti. Sur chaque noeud, l'agent de mise à jour accepte des requêtes de modification de la politique de sécurité locale. Ainsi, la politique d'un noeud peut évoluer de façon spécifique en fonction des applications installées. En effet, les

requêtes de modification sont produites localement, par l'administrateur du noeud. L'administration est donc effectuée de façon locale. Ceci répond au troisième besoin d'administration.

L'agent de mise à jour de notre architecture est capable d'interagir avec des agents de détection d'intrusion ou des agents de vérification de l'application de la méta-politique. Les agents de détection peuvent proposer des modifications de la politique de protection locale lorsqu'une attaque est détectée. Les agents de vérification testent 1) si la méta-politique présente sur la machine est bien celle qui a été déployée, 2) si les évolutions de la politique sont bien conformes à la politique de modification (une évolution non conforme indiquerait une compromission de l'agent de mise à jour), et enfin 3) si la configuration produite pour le mécanisme de contrôle d'accès local (SELINUX, grsecurity) est bien conforme à la politique de protection locale. Ainsi, l'administration peut être automatisée, et des modifications pertinentes de la politique locale peuvent être proposées à l'administrateur, améliorant effectivement la qualité du processus d'évolution de la politique. Ceci répond au quatrième besoin d'administration.

La possibilité d'administrer localement chaque noeud du système réparti est assurée par la présence de l'agent de mise à jour. Ainsi, il n'est pas nécessaire de mettre en oeuvre des connexions réseau sur les noeuds pour modifier leur politique de sécurité. En effet, les requêtes de modification peuvent être produites localement, et sont traitées localement. C'est pourquoi notre architecture offre une meilleure tolérance aux pannes que des mécanismes reposant sur un composant central du réseau pour l'administration d'un système réparti. Ceci répond au besoin de tolérance aux pannes.

Dans les approches traditionnelles multi-domaines et multi-politiques, des conflits peuvent apparaître en temps réel, ce qui peut pénaliser les performances. Notre architecture ne définit qu'une seule méta-politique, commune à l'ensemble du système réparti. Ensuite les modifications de politique sont effectuées localement. Ainsi, l'éventualité d'un conflit d'application de politique est écartée, ce qui évite tout impact sur les performances de l'architecture. De plus, pour effectuer le contrôle d'accès sur les systèmes d'exploitation, nous réutilisons des solutions comme SELINUX et grsecurity, qui ont été optimisé pour être performants. Ceci répond au besoin de performances.

Les approches traditionnelles multi-domaines et multi-politiques résolvent les conflits par des priorités dans l'application des règles de politiques concurrentes. Cela ne permet donc pas en général de vérifier que pour des évolutions réparties des politiques, les objectifs de sécurité sont respectés. Par ailleurs, il n'existe pas d'outils permettant de garantir l'absence de flux d'information pour des politiques dont à la fois les contextes et les permissions sont dynamiques. Seuls quelques cas particuliers sont traités comme la séparation de privilèges dans les modèles à base de rôle. Mais clairement l'absence de flux d'informations est un cadre plus général que par exemple la simple séparation de privilèges. C'est pourquoi, notre modèle méta-politique intègre une méthode pour la vérification statique qui valide les règles de modifications. Cette méthode permet de garantir l'absence de flux d'information illégaux dans les politiques locales. C'est une avancée en soi puisque le problème est classiquement connu comme indécidable lorsque l'on considère des politiques dynamiques. Ici, les contraintes d'évolutions garantissent de fait que l'ajout d'une règle particulière n'est pas possible. Donc nous pouvons nous consacrer à des garanties plus intéressantes pour l'administrateur et qui permettent de valider une méta-politique et donc de garantir l'absence de flux illégaux sur tout le système réparti.

1.5 Plan du mémoire

Dans le chapitre 2, un état de l'art sera établi sur les modèles de contrôle d'accès (les familles DAC, MAC et RBAC), les implantations système (Medusa DS9, LIDS, RSBAC, SELINUX, grsecurity),

et sur les modèles d'expression de politiques de sécurité (multi-domaines, multi-politiques et méta-politiques). Le chapitre 3 mènera une discussion sur l'état de l'art dont le but sera d'évaluer les implantations système les plus intéressantes, et l'intérêt des modèles d'expression de politiques de sécurité vis-à-vis de nos objectifs (décentralisation de l'administration, intégration de systèmes d'exploitation de confiance (TOS), forte tolérance aux pannes, bonnes performances et support de méthodes de vérification d'absence de flux d'information illégaux). Ce chapitre conclut sur la nécessité de définir un nouveau modèle de Méta-Politique. Le chapitre 4 présentera le principe de ce nouveau modèle et en quoi ce modèle répond aux objectifs. Nous illustrons les intérêts pratiques avec des exemples concrets. Le chapitre 5 propose un modèle formel définissant d'une part un cadre général pour notre méta-politique et une application pour en déduire un modèle pour le contrôle d'accès. Le modèle de contrôle d'accès obtenu établit l'idée maîtresse de notre approche. Il repose sur deux langages : un langage neutre pour le contrôle d'accès, et un langage de contraintes sur l'évolution des politiques de contrôle d'accès. Ce chapitre élargit le modèle général pour l'appliquer à la détection d'intrusion et présente des mécanismes de coopération entre contrôle d'accès et détection d'intrusion. Le chapitre 6 développe une méthode pour garantir le respect des objectifs de sécurité. Cette méthode consiste à vérifier l'absence de flux d'information illégaux dans la méta-politique. Nous présentons l'algorithme général et en évaluons la complexité. Le chapitre 7 est consacré à l'implantation de l'agent de de mise à jour, aux formats XML de représentation des deux langages et au mécanisme de traduction du langage neutre de contrôle d'accès vers SELINUX et grsecurity. Enfin, le chapitre 8 résume l'apport de la thèse et donne les perspectives de cette étude.

Chapitre 2

Etat de l'art

Le besoin de politiques de sécurité pour les systèmes d'information est apparu dès les premières études sur les problématiques de confidentialité et d'intégrité. Elles font partie intégrante des critères d'évaluation de la sécurité, tant au niveau international dans le TCSEC [TCSEC 1985], qu'au niveau européen dans ITSEC [ITSEC 1991]. Le travail mené dans le cadre de cette thèse s'attache à résoudre des problématiques de contrôle d'accès dans des systèmes d'exploitation via un modèle de politique de sécurité adapté à la répartition.

Afin de comprendre ce que sont les politiques de sécurité, ce chapitre étudie les travaux de référence, dans les domaines du contrôle d'accès et de l'administration de politiques de sécurité. Nous commencerons par la problématique du contrôle d'accès, d'une part les différents modèles théoriques qui ont été développés pour y répondre, et d'autre part les implantations existant aujourd'hui. Ensuite nous étudierons les approches concernant l'expression des politiques de sécurité : modèles multi-domaines, modèles multi-politiques et méta-politiques.

2.1 Modèles théoriques de contrôle d'accès

Avec le contrôle d'accès, on s'intéresse à garantir deux propriétés fondamentales : la *confidentialité* et l'*intégrité* des informations contenues dans un SI.

Définition 2.1.1 (Confidentialité) *Prévention des accès non autorisés à une information.*

Définition 2.1.2 (Intégrité) *Prévention des modifications non autorisées d'une information.*

Afin de permettre l'implantation de politiques de confidentialité et d'intégrité en leur sein, les systèmes d'exploitation disposent de mécanismes de contrôle d'accès. Typiquement, ceux-ci fonctionnent sur le modèle suivant :

- Un *sujet* est une entité active du système (essentiellement les processus) ;
- Un *objet* est une entité passive, un conteneur d'information, sur lequel un sujet peut effectuer une action (les fichiers, sockets de communication, périphériques matériels...) ;
- Une *permission* est une certaine action sur un objet. Une permission peut être accordée ou refusée (par exemple, lecture, écriture, exécution...).

Le contrôle d'accès est configuré par un ensemble de règles spécifiant un sujet, un objet et des droits d'accès (cf. figure 2.1). Un cas particulier de règles (cf. figure 2.2) est celui spécifiant une action d'un sujet vers un autre sujet (envoi de signal ou de message inter-processus).

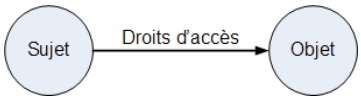

FIG. 2.1 – Règle d'accès.

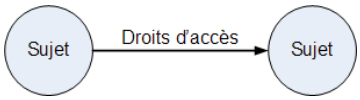

FIG. 2.2 – Cas particulier.

La manière naturelle de représenter les règles d'accès est sous forme matricielle, par une *matrice d'accès*. L'ensemble des sujets correspond aux lignes, et l'ensemble des objets aux colonnes. Dans chaque case de la matrice, on trouve un sous-ensemble des permissions.

Ainsi, une contrainte sur la confidentialité se traduira par le refus de la permission de lecture : par exemple, pour empêcher que le serveur Web `apache` n'accède au fichier contenant les mots de passe des utilisateurs, `/etc/shadow`, on ne lui autorise pas la lecture de celui-ci.

La notion de matrice d'accès a été introduite par [Lampson 1971]. Il propose de placer suivant les lignes l'ensemble D des domaines de protection (qui représentent des contextes d'exécution pour les programmes, i.e. les sujets), et en colonnes l'ensemble X des objets (qui inclut les domaines), dans la matrice A. Il pose ensuite deux définitions :

Définition 2.1.3 (Capabilities Lists) *Étant donné un domaine $d \in D$, la liste des capacités (capabilities) pour le domaine d est l'ensemble des couples $(o, A[d, o]), \forall o \in X$.*

Définition 2.1.4 (Access Control Lists) *Étant donné un objet $o \in X$, la liste de contrôle d'accès (ACL) pour l'objet o est l'ensemble des couples $(d, A[d, o]), \forall d \in D$.*

La définition 2.1.3 lie un domaine d avec les permissions $A[d, o]$, de la matrice d'accès A, qu'il possède sur chaque objet o. La définition 2.1.4 lie un objet o avec les permissions $A[d, o]$ que la matrice A accorde à chaque domaine d.

A noter que les notions de *capability* et d'ACL avaient déjà été définis de façon générale dans [Lampson 1969]. Dans [Lampson 1971], l'auteur raffine ces définitions en les liant a la notion de matrice d'accès.

Ensuite, il est nécessaire de pouvoir faire évoluer ces politiques de sécurité, afin par exemple de prendre en compte l'intégration de nouveaux utilisateurs et applications dans le système d'exploitation. Cette évolution doit être contrôlée pour prévenir les abus, ainsi que les configurations indésirables pour la politique. C'est pourquoi chaque mécanisme de contrôle d'accès définit des procédures de modification des droits d'accès sur les objets du système. Elles sont plus ou moins restrictives suivant le but recherché, et requièrent éventuellement un accès de niveau administrateur.

Typiquement, sous Linux, ces procédures sont accomplies par les commandes `chown` et `chmod`. La première modifie le propriétaire d'un fichier ou dossier, et ne peut-être utilisées que par *root*. La seconde modifie les droits d'accès sur un fichier ou dossier, et requiert que l'utilisateur en soit propriétaire. Par conséquent, pour modifier les droits d'accès d'un fichier appartenant à *root*, il est nécessaire d'utiliser le compte *root*.

Enfin, la croissance des vulnérabilités liées aux interactions entre des systèmes d'informations distants a poussé de nouvelles études sur l'administration de politiques à grande échelle. Ainsi, la prise en compte de multiples sites logiques d'application (différents réseaux par exemple) ou de différentes organisations amenées à coopérer sont les apports des recherches en matière de *politiques multi-domaines*. Également, le support des équipements relevant de langages de politiques de sécurité différents est au cœur des études sur les *modèles multi-politiques*. Enfin, la complexité d'écriture de telles politiques de sécurité a conduit à la conception de *méta-politiques*. Nous aborderons ces points dans la partie 2.3.

Les parties suivantes présentent les différentes familles de modèles théoriques. On en distingue principalement trois :

Contrôle d'accès discrétionnaire La caractéristique principale du DAC ou *Discretionary Access Control* est le fait que ce sont les utilisateurs qui attribuent les permissions sur les ressources qu'ils possèdent. C'est le type de mécanisme utilisé principalement dans les systèmes d'exploitation modernes. En effet, il est très léger en termes d'administration, étant donné que l'attribution des droits est faite par les utilisateurs et non pas par les administrateurs.

Contrôle d'accès obligatoire Contrairement au DAC, le MAC ou *Mandatory Access Control* délègue l'attribution des permissions à une entité tierce, typiquement un administrateur externe de la politique de sécurité. Ainsi, les utilisateurs du système ne peuvent pas intervenir dans l'attribution des permissions d'accès, même s'ils disposent de droits d'administration dans le système d'exploitation.

Contrôle d'accès basé sur les rôles Le modèle RBAC pour *Role-Based Access Control* a pour but de simplifier l'administration des droits d'accès des utilisateurs individuels en fournissant un niveau d'indirection supplémentaire. Plutôt que de donner directement des permissions aux utilisateurs, on définira différents rôles possibles pour l'utilisation du système d'exploitation, avec des droits d'accès associés. Ensuite chaque utilisateur a accès à une liste de rôle, suivant son activité. Un seul rôle peut être actif à un moment donné.

2.1.1 DAC

Le contrôle d'accès discrétionnaire (DAC) est le modèle implanté dans la majorité des systèmes d'exploitation actuels (Microsoft Windows, Solaris, Linux, FreeBSD). Il doit son nom au fait que l'attribution des permissions d'accès se fait "à la discrétion" du propriétaire d'une ressource. Typiquement, les droits d'accès sur un fichier sont positionnés par l'utilisateur déclaré comme propriétaire de ce fichier.

Les modèles théoriques de DAC que nous allons étudier maintenant sont HRU et TAM.

2.1.1.1 Modèle HRU

Établi par [Harrison *et al.* 1976], le modèle HRU décrit les politiques de contrôle d'accès discrétionnaires traditionnelles. Une matrice P représente l'ensemble des droits d'accès des sujets vers les objets. De plus, les sujets peuvent créer de nouveaux sujets ou objets, et ajouter des droits.

HRU propose de modéliser la protection dans les systèmes d'exploitation comme suit : l'ensemble des sujets S et l'ensemble des objets O sont tous deux l'ensemble des entiers de 1 à k. R est l'ensemble des droits génériques (tels que possession, lecture, écriture, exécution). Le système d'exploitation contient un ensemble fini C de commandes $\alpha_1, \ldots, \alpha_n$, qui représente toutes les opérations fournies par le système d'exploitation (création de fichier, modification des droits...). Du point de vue de la protection, les commandes de l'ensemble C ont toutes la même forme : elles prennent en paramètre des sujets et objets, et suivant la présence de certains droits dans la matrice P, elles effectuent un ensemble d'actions élémentaires sur le système. Ces actions élémentaires sont : `enter` et `delete` pour l'ajout et la suppression de droits, `create subject` et `create object` pour la création de nouveaux sujets et objets et enfin `destroy subject` et `destroy object` pour la destruction de sujets et objets. La configuration du système de protection est le triplet (S, O, P).

Pour étudier le problème de la sûreté d'un système de protection, HRU s'intéresse au transfert de droit, qui se produit lorsqu'une commande insère un droit particulier r dans la matrice P, dans une

case ou il était précédemment absent. Par conséquent, étant donné une configuration initiale de la politique de sécurité, un système est considéré sûr (*safe*) pour un droit r si aucune des commandes ne provoque le transfert du droit r.

Les auteurs du modèle HRU ont prouvé que dans le cas d'un système de protection *mono-opérationnel*, i.e. dans lequel toutes les commandes ne contiennent qu'une seule action élémentaire, le problème de savoir si ce système est sûr *est décidable*, toutefois l'algorithme de vérification est *NP-complet*. Or la seule commande de création de fichier d'un système d'exploitation de type UNIX se compose déjà de deux actions : création d'un nouvel objet, et positionnement des droits sur celui-ci. Les auteurs s'intéressent donc au problème dans le cas général, et prouvent que le problème de la sûreté d'un système de protection est *indécidable* dans le cas général.

En outre, les auteurs mentionnent le fait que la taille du problème est réduite à une taille polynomiale dès lors que les actions de création de sujet ou d'objet sont retirées du système de protection.

2.1.1.2 Modèle TAM

Le modèle Typed Access Matrix (TAM), introduit par [Sandhu 1992], propose une extension du modèle HRU, en intégrant la notion de typage fort. Cette notion dérive de travaux plus anciens sur SPM (Sandhu's Schematic Protection Model) par [Sandhu 1988], et se traduit par l'attachement de "types de sécurité" immuables à tous les sujets et objets du système d'exploitation.

Par rapport à la modélisation de HRU vue précédemment, il existe maintenant l'ensemble T des types de sécurité. Le point important est que cet ensemble est fini : la création de nouveaux types n'est pas possible. La gestion des types est prise en compte dans les opérations élémentaires décrites plus haut pour HRU.

Ensuite, Sandhu s'intéresse à la version monotone de TAM, soit MTAM (Monotonic Typed Access Matrix), obtenue en ôtant les opérations de suppression (droits, sujets ou objets). Il démontre que le problème de la sûreté est décidable dans le cas d'un modèle de protection MTAM où le graphe de création des sujets et objets est acyclique. Toutefois la complexité de ce problème reste NP. C'est pourquoi Sandhu définit le modèle MTAM ternaire, dans lequel toutes les commandes ont au maximum trois arguments. Au prix d'une perte d'expressivité, le problème de sureté voit sa complexité ramenée à un degré polynomial.

2.1.2 MAC

L'un des grands avantages du DAC est que l'attribution des permissions d'accès est gérée directement par les utilisateurs. Malheureusement, c'est aussi l'une de ses défaillances majeures. En effet, comme le résume [Anderson 1972], s'il est possible de se protéger contre des attaques externes (provenant de réseaux informatiques distants) en mettant en place des barrières de protection supplémentaires, il est par contre difficile de se prémunir contre les actions des *utilisateurs malicieux*. En particulier, les failles logicielles menant à l'obtention d'un accès de niveau administrateur fournissent à un pirate le moyen d'outrepasser complètement le DAC.

Pour intégrer un mécanisme de validation des accès entre sujets et objets, soit dans le système d'exploitation, soit directement dans les composants matériels, Anderson propose le modèle du *Moniteur de Référence* (Reference Monitor). Celui-ci est configuré à partir d'une matrice d'accès fixe, définie par une entité tierce externe au système, et spécifiant exhaustivement quels accès sont autorisés ou refusés entre les sujets et objets du système d'exploitation. Ainsi il devient possible de garantir que des droits d'accès considérés dangereux ne seront jamais donnés par erreur, puisque la matrice d'accès est fixe.

Le concept de Moniteur de Référence est au cœur de la définition du Contrôle d'accès obligatoire (Mandatory Access Control ou MAC). Ce terme, qui désignait initialement le modèle défini par Bell et LaPadula, a vu sa signification évoluer et représente aujourd'hui tout mécanisme qui place la gestion de l'attribution des permissions d'accès hors d'atteinte des utilisateurs concernés par ces permissions.

Les différents modèles présentés par la suite se positionnent à l'intérieur du Moniteur de Référence, comme mécanisme de validation. Ils ont tous pour objectif de spécifier une certaine politique de sécurité en matière de contrôle d'accès.

2.1.2.1 Modèle Bell-LaPadula

Le modèle Bell-LaPadula, établi par [Bell et La Padula 1973], plus couramment appelé BLP, est issu des besoins en confidentialité des données du monde militaire. En plus des notions de sujet, d'objet et de matrice d'accès, le modèle BLP introduit la notion de **label**. Un label est associé à chaque sujet et objet du système, et contient un *niveau de sécurité*. Ceux-ci sont composés de deux types d'identifiants de sécurité :

1. un identifiant hiérarchique : le **niveau de classification** pour les objets, ou niveau de sensibilité, et le **niveau d'habilitation** pour les sujets, qui sont typiquement non classifié, confidentiel, secret, top secret ;
2. des identifiants de catégories : indépendamment de la hiérarchie de confidentialité, différentes catégories d'informations existent, et correspondent aux différentes organisations manipulant ces données.

Un niveau de sécurité s'écrit donc : (classification, ensemble de catégories). De plus, il existe une relation de dominance entre les niveaux de classification. Sur cette base, il est maintenant possible de définir de nouvelles règles qui s'appliqueront en plus de la matrice d'accès classique. Ces nouvelles lois sont :

ss-property Appelée *simple security property* ou **No Read Up**, elle garantit que lorsqu'un sujet demande un accès en lecture sur un objet, son niveau est supérieur ou égal à celui de l'objet (cf. flèche de gauche sur la figure 2.3).

***-property** Appelée *star-property* ou **No Write Down**, cette loi garantit que lorsqu'un sujet demande un accès en écriture sur un objet, son niveau est inférieur ou égal à celui de l'objet (cf. flèche de droite sur la figure 2.3).

Le système est donc modélisé de la façon suivante : un ensemble de sujets S, un ensemble d'objets O, une matrice d'accès M et une fonction donnant le niveau f. On dispose également d'un ensemble de permissions d'accès $A = \{e, r, a, w\}$. Elles sont classées suivant leur capacité d'observation (lecture) et d'altération (écriture) de l'information :

e Ni observation ni altération (*execute*) ;

r Observation sans altération (*read*) ;

a Altération sans observation (*append*) ;

w Observation et altération (*write*).

Les deux lois sur les niveaux, illustrées par la figure 2.3, s'écrivent ainsi :

$$r \in M[s, o] \Rightarrow f(s) \geq f(o)$$
$$a \in M[s, o] \Rightarrow f(s) \leq f(o)$$
$$w \in M[s, o] \Rightarrow f(s) = f(o)$$

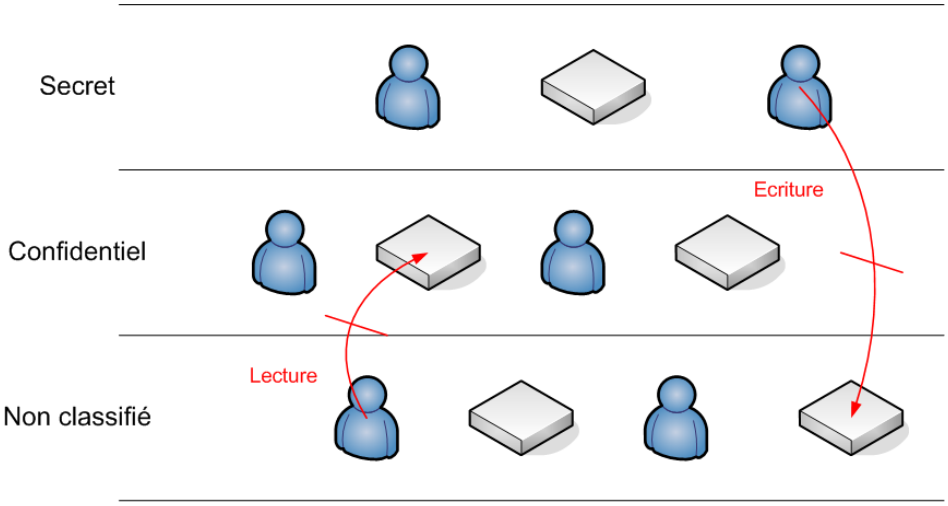

FIG. 2.3 – No Read Up et No Write Down.

Ces lois signifient que 1) pour qu'un sujet s ait accès en lecture à un objet o, son niveau d'habilitation $f(s)$ doit être supérieur ou égal au niveau de classification $f(o)$ de l'objet, 2) pour qu'un sujet s ait accès en écriture à un objet o, son niveau d'habilitation doit être inférieur ou égal au niveau de classification de l'objet, et 3) pour qu'un sujet ait accès en lecture / écriture à un objet o, son niveau d'habilitation doit être égal au niveau de classification de l'objet.

Ce modèle a historiquement été implanté dans de nombreux systèmes d'exploitation. L'article de Bell et LaPadula décrit son intégration dans MULTICS, on le trouve également dans certaines versions de Solaris, HPUX ou autres systèmes UNIX. Il n'est pas désigné sous le nom BLP, mais plutôt MLS (Multi-Level Security) ou modèle de sécurité multi-niveaux. Par exemple, le système d'exploitation Unix System V/MLS intègre la gestion des niveaux de sécurité décrits dans le modèle BLP.

Toutefois ce modèle est difficile à appliquer tel quel à l'ensemble d'un système d'exploitation. Il est difficile d'attribuer des labels à certains sujets ou objets, par exemple le cas du dossier `/tmp` où tous les processus doivent pouvoir créer des fichiers. Des aménagements ont été prévus, le fait de migrer un objet d'un niveau de classification à un niveau supérieur lorsqu'il est accédé en écriture par un sujet de niveau supérieur (car normalement il est interdit de modifier un fichier de niveau inférieur). L'objet se trouve après l'accès au même niveau que le sujet. Mais l'effet néfaste lié à cet aménagement est que les objets ont tendance à être "tirés vers le haut", et donc à tous se trouver sur le niveau le plus haut après un certain temps.

Par exemple, dans l'implantation MLS pour FreeBSD, on trouve des niveaux de sécurité spéciaux, *low*, *equal* et *high*, qui respectivement signifient un niveau toujours inférieur, toujours égal ou toujours supérieur lorsqu'ils sont appliqués. Par exemple, on peut affecter le niveau *equal* à un objet auquel on ne souhaite pas appliquer de politique particulière, *low* à un objet pour lequel on veut interdire l'écriture.

Un article ultérieur de [McLean 1987] restitue le modèle BLP de façon plus claire, et accompagnée de critiques concernant des failles apparaissant dans le modèle BLP lorsqu'il est appliqué à un système réel. Bell y a répondu par un autre article [Bell 1988], dans lequel il précise que son étude concernait un modèle abstrait de système d'exploitation, et que les critiques de McLean sont infondées dans cette optique.

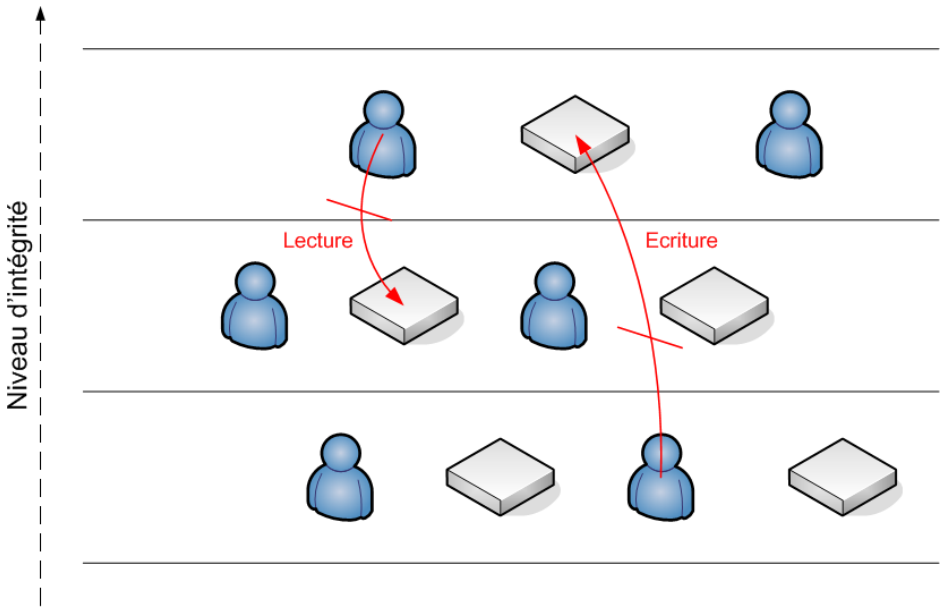

FIG. 2.4 – No Read Down et No Write Up.

2.1.2.2 Modèle Biba

Le modèle BLP vu précédemment ne répond qu'à des besoins de confidentialité. C'est pour cette raison qu'un modèle complémentaire a été établi, qu'on peut qualifier de modèle *dual* à BLP. Il s'agit du modèle Biba, d'après le nom de son auteur [Biba 1975]. Il a pour objectif, de façon similaire à BLP, de traiter les problèmes de maintien de l'intégrité de l'information au sein du système d'exploitation.

A partir d'un modèle de protection similaire à celui vu pour BLP, il introduit une nouvelle fonction qui définit le niveau d'intégrité d'un sujet ou d'un objet. Deux lois permettent d'assurer l'intégrité :

No Read Down Un sujet n'est pas autorisé à accéder en lecture à un objet d'intégrité strictement inférieure, car cela pourrait corrompre sa propre intégrité (cf. flèche de gauche sur la figure 2.4) ;

No Write Up Un sujet n'est pas autorisé à altérer le contenu d'un objet d'intégrité strictement supérieure (cf. flèche de gauche sur la figure 2.4).

Les deux lois s'écrivent (cf. figure 2.4) :

$$r \in M[s,o] \Rightarrow f(s) \leq f(o)$$
$$a \in M[s,o] \Rightarrow f(s) \geq f(o)$$
$$w \in M[s,o] \Rightarrow f(s) = f(o)$$

Ces lois signifient que 1) pour qu'un sujet s ait accès en lecture à un objet o, son niveau d'intégrité $f(s)$ doit être inférieur ou égal au niveau d'intégrité $f(o)$ de l'objet, 2) pour qu'un sujet s ait accès en écriture à un objet o, son niveau d'intégrité doit être supérieur ou égal au niveau d'intégrité de l'objet, et 3) pour qu'un sujet ait accès en lecture / écriture à un objet o, son niveau d'intégrité doit être égal au niveau d'intégrité de l'objet. Ces règles évitent à tout moment que se produise un transfert d'information d'un niveau d'intégrité bas vers un niveau d'intégrité haut, ce qui signifierait une compromission de l'intégrité du niveau haut.

A l'instar du modèle BLP, le modèle Biba est difficile à utiliser tel quel dans un système d'exploitation. Ici l'idée est de migrer un sujet vers un niveau d'intégrité inférieur lorsqu'il accède à un objet de niveau inférieur. L'effet néfaste est que l'ensemble des sujets va rapidement se trouver en bas de l'échelle des niveaux d'intégrité. Dés lors, l'intérêt du modèle Biba est fortement remis en cause, puisqu'il n'y a plus de différence entre les sujets.

2.1.2.3 Modèle Clark-Wilson

Le travail de [Clark et Wilson 1987] est fondé sur la constatation de la différence entre les besoins en sécurité du monde militaire et ceux du milieu commercial. En particulier, dans le monde du commerce, si la confidentialité des données est un point important, l'intégrité l'est encore davantage.

Dans ce modèle, on considère l'espace des données D, composé de deux parties disjointes : les données intègres (*Constrained Data Items* ou CDI) et les données non intègres (*Unconstrained Data Items* ou UDI). On a :

$$D = CDI \cup UDI$$
$$CDI \cap UDI = \emptyset$$

Deux classes de procédures sont définies : les IVP (Integrity Validation Procedure ou procédures de validations d'intégrité) et les TP (Transformation Procedure ou procédures de transformation). Les IVP sont utilisées pour trier les éléments de D, qui sont placés dans CDI si leur intégrité est validée, et dans UDI dans le cas contraire. Les TP sont les transactions valides du système, c'est-à-dire qu'elles ne perturbent pas l'intégrité des données auxquelles elles sont appliquées. L'intégrité des données est ensuite assurée par un ensemble de règles parmi lesquelles :

- Lorsqu'une TP est appliquée à un élément de CDI, le résultat reste dans CDI ;
- Certaines TP spécifiques peuvent prendre un élément de UDI en entrée, et produire un élément dans CDI.

Ce modèle repose sur des pratiques courantes concernant les données non informatisées du monde du commerce. Cependant, il est assez complexe à mettre en œuvre dans un système d'exploitation. En effet, il faudrait créer des IVP et des TP pour l'ensemble des programmes que l'on y trouve, ce qui s'avèrerait complexe, en particulier en ce qui concerne la vérification du bon fonctionnement d'un programme.

2.1.2.4 Modèle de la Muraille de Chine

Le modèle de politique de sécurité de la Muraille de Chine [Brewer et Nash 1989] est issu, comme le modèle Clark-Wilson, des besoins du monde commercial. Son objectif est d'empêcher que des sujets aient accès simultanément à des données issues d'entreprise en concurrence. Les objets sont regroupés en **"classes de conflit d'intérêt"**, et la règle est qu'un sujet peut accéder au maximum à un seul objet de chacune de ces classes.

Le modèle reprend les concepts de sujets, objets et labels de sécurité introduits par BLP. Concernant les objets, ils sont classés suivant trois niveaux hiérarchiques :

1. Au niveau le plus bas, on trouve les données simples, typiquement les fichiers, qui sont les objets du système ;
2. Au niveau intermédiaire, les objets sont groupés suivant la compagnie à laquelle ils appartiennent, formant des ensembles de données ;

3. Au niveau le plus haut, les compagnies sont groupées suivant le fait qu'elles soient en concurrence ou non. Ceci forme les classes de conflit d'intérêt mentionnées précédemment.

Dans le but de définir des lois d'accès similaires à BLP, les auteurs introduisent une matrice N qui contient l'historique des accès précédemment effectués. Il s'agit en fait de placer les sujets en ligne, et les objets en colonne, et dans chaque case un booléen indique si un accès a déjà eu lieu pour le couple (sujet, objet) correspondant. Ainsi deux règles régissent les accès entre sujets et objets :

- *Simple security* : l'accès en lecture à un objet est permis :
  - s'il appartient à la même compagnie qu'un objet déjà accédé ;
  - *ou* s'il appartient à une autre classe de conflit d'intérêt que les objets déjà en accès.
- **-property* : l'accès en écriture à un objet est permis :
  - si l'accès est autorisé par la règle de sécurité simple ;
  - *et* si le sujet ne peut lire aucun objet d'une autre compagnie qui contiendrait de l'information non assainie.

Le terme *"information assainie"* désigne une information déguisée de manière à ce qu'il soit impossible de découvrir l'identité de la compagnie à laquelle cette information appartient. La règle **-property* évite principalement qu'un utilisateur lise des données d'une compagnie et les écrive dans un fichier d'une autre compagnie, ce qui occasionnerait une fuite d'information.

2.1.2.5 Modèle DTE

Le modèle "Domain and Type Enforcement" (DTE) [Boebert et Kain 1985] est assez différent des précédents dans le sens où il ne cherche pas à garantir la sûreté de la protection qu'il fournit. Il s'agit plutôt d'un mécanisme générique à partir duquel on peut implanter diverses politiques de sécurité dans un système d'exploitation. A partir d'implémentations disponibles pour les systèmes d'exploitation commerciaux [Badger *et al.* 1995], il est particulièrement efficace pour les problématiques de *confinement* de processus, comme illustré par [Walker *et al.* 1996]. Le confinement diffère des propriétés de confidentialité ou d'intégrité, en effet, son objectif est de spécifier, pour chaque sujet du système, un ensemble d'actions autorisées. Il se rapproche du modèle TAM sur l'utilisation du typage fort.

Concrètement, dans un système d'exploitation, les politiques de sécurité définies via le modèle DTE visent à :

- Restreindre les ressources accessibles par un programme, notamment les programmes privilégiés (s'exécutant sous le compte *root*), suivant le principe de "moindre privilège" décrit dans [Department of Defense 1991] ;
- Contrôler quels programmes ont accès aux ressources "sensibles", et empêcher l'accès par tout autre programme.

Par exemple, DTE peut être utilisé pour contrôler un programme tel que le serveur Web *apache*, afin de garantir par exemple que même s'il s'exécute sous le compte *root*, il n'a accès qu'à ses fichiers de configuration, ses librairies et aux pages web. Il pourra également garantir que seul apache a accès à son fichier de configuration.

Le modèle reprend les principes de sujet, d'objet et de matrice d'accès. Toutefois, les lois d'accès n'opèrent pas sur les sujets, mais les **domaines**, et de même les objets sont englobés dans des **types**. Ainsi chaque objet du système possède un type, et chaque processus s'exécute dans un domaine précis, dont découlent des droits d'accès.

DTE définit trois tables :

1. La **table de typage**, qui assigne les types aux objets du système ;

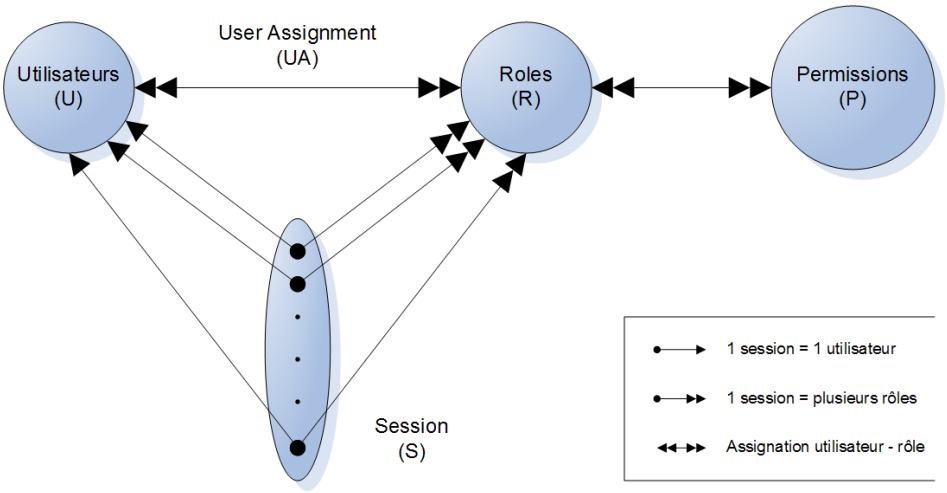

FIG. 2.5 – Le modèle RBAC_0.

2. La **table de définition des domaines** (DDT) qui spécifie les droits d'accès (lecture, écriture, exécution, ajout, suppression) de chaque domaine sur les différents types ;
3. La **table d'interaction des domaines** (DIT) qui définit les droits d'accès entre domaines (création, destruction, envoi de signal).

La DDT définit également les "points d'entrée" des domaines, c'est à dire quels sont les fichiers binaires qui, une fois exécutés, vont créer un processus dans tel ou tel domaine. La définition des trois tables est effectué par un administrateur, et les utilisateurs ne peuvent pas les modifier. C'est pourquoi DTE rentre dans la catégorie MAC.

2.1.3 RBAC

Étant donné, d'une part, la difficulté d'application des modèles MAC dans les systèmes d'exploitation non-militaires, et d'autre part la faiblesse des modèles DAC, les études sur le contrôle d'accès se portèrent vers une nouvelle famille de modèle. Un premier article par [Ferraiolo et Kuhn 1992] a mis en évidence le fait que, la plupart du temps, les permissions d'accès ne sont pas attribuées en fonction de l'identité des utilisateurs. C'est plutôt le type d'activité qui les détermine, c'est-à-dire le *rôle* qu'occupe un utilisateur dans une entreprise ou une organisation. C'est ainsi qu'est conçu le modèle RBAC (Role-Based Access Control) ou contrôle d'accès déterminé par le rôle.

Par la suite, un article plus complet de [Sandhu *et al.* 1996] a posé de façon formelle les différents modèles qui forment le cœur de la famille RBAC. Les *utilisateurs* ont un certain *rôle*, qui caractérise leur activité. Chaque rôle est associé à un ensemble de *permissions*, qui représentent des possibilités d'accès aux objets du système. Enfin, l'utilisateur exerce son activité dans le cadre de *sessions*, durant lesquelles il peut activer un sous-ensemble des rôles auxquels il appartient. Ce modèle simple est appelé RBAC_0 (cf. figure2.5).

Dans un autre article, [Ferraiolo *et al.* 1995] fournissent davantage de détails sur la conception et l'implantation de RBAC. Ils raffinent la notion de permissions comme étant une liste d'*opérations* sur les objets du système.

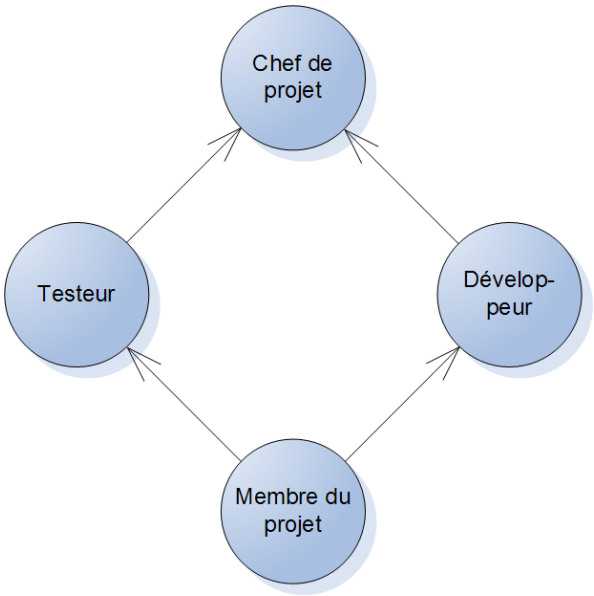

FIG. 2.6 – Les hiérarchies de rôles dans RBAC_1.

2.1.3.1 Hiérarchies de rôles

En plus du modèle simple RBAC_0, [Sandhu *et al.* 1996] introduit la notions de **hiérarchie** dans les rôles, dans le modèle RBAC_1. L'objectif de la hiérarchie des rôles est l'introduction de la notion d'héritage des permissions entre les rôles, de façon analogue à l'organisation des métiers en entreprise, où l'on trouve une hiérarchie organisée suivant le degré d'autorité et de responsabilité. Le modèle RBAC_1 apporte les définitions de rôles *junior*, avec des degrés de privilèges inférieurs, et des rôles *senior* possédant des privilèges supérieurs. Dans les diagrammes de représentation des hiérarchies, ces derniers apparaissent typiquement vers le sommet, tandis que les rôles moins privilégiés sont placés plus bas.

Comme dans l'exemple de la figure 2.6, on peut envisager une hiérarchie appliquée au développement de projets informatiques, où le rôle le plus bas est celui de "membre du projet", et définit des permissions de base. Ensuite ce rôlé est hérité et raffiné par deux autres, "Développeur" et "Testeur". Enfin le rôle de "chef de projet" hérite de toutes les permissions.

2.1.3.2 Contraintes

Le modèle RBAC_2, proposé également par [Sandhu *et al.* 1996], introduit le support des **contraintes**. Il ne s'agit toutefois pas d'un ajout au modèle RBAC_1, car les hiérarchies de rôles ne sont pas supportées dans ce modèle. L'objectif des contraintes est de modéliser le fait que des rôles puissent être totalement disjoints, comme par exemple, au sein d'une entreprise, le directeur des achats et le directeur financier. Un même utilisateur ne peut en aucun cas appartenir à ces deux rôles simultanément, car il pourrait alors commettre des fraudes.

Les types de contraintes supportés dans RBAC_2 sont généralement des considérations simples, c'est-à-dire dont la vérification et l'application se fait directement. Elles sont appliquées soit au niveau de l'attribution des rôles aux utilisateurs, soit au niveau de l'attribution des permissions aux rôles. Les principaux types de contraintes sont les suivants :

– **Rôles mutuellement exclusifs** : un même utilisateur ne peut être assigné qu'à un seul rôle dans

un ensemble de rôle mutuellement exclusifs. Cette contrainte assure le principe de *séparation des privilèges* décrit dans [Clark et Wilson 1987] ;
- **Cardinalité** : contraintes sur le nombre maximal d'utilisateur assignés à un certain rôle ;
- **Rôles pré-requis** : dans certains cas, il est intéressant d'imposer des pré-requis en matière d'assignation de rôle ou de permissions. Par exemple, un utilisateur ne peut être assigné au rôle *A* que s'il appartient déjà au rôle *B*, et très souvent *A* sera un rôle *senior* par rapport à *B*.

Il est important de noter que les contraintes, quelles qu'elles soient, ne sont valables que si un contrôle strict est appliqué en ce qui concerne l'assignation des comptes d'utilisateurs aux êtres humains. Si un employé possède deux comptes, eux-mêmes appartenant à deux rôles sensés être mutuellement exclusifs, les contraintes n'ont plus aucune efficacité.

Enfin, les hiérarchies de rôles peuvent aussi être définies comme des contraintes, et alors RBAC_1 apparait comme un sous-ensemble de RBAC_2. Cependant, le support direct des hiérarchies dans les implantations de RBAC reste préférable.

2.1.3.3 Modèle consolidé

Le dernier modèle présenté par [Sandhu *et al.* 1996] est RBAC_3, la combinaison de RBAC_1 et RBAC_2. Il supporte donc à la fois les hiérarchies de rôles et les contraintes sur l'assignation des rôles et des permissions. L'objectif de RBAC_3 est essentiellement de résoudre les problèmes soulevés par la cohabitation de ces deux concepts : support des contraintes sur la hiérarchie de rôles, résolution de conflits entre l'héritage et les contraintes (par exemple quand un rôle hérite de deux rôles mutuellement exclusifs).

2.2 Implantations système du contrôle d'accès

Dans la partie précédente nous avons abordé les modèles théoriques de contrôle d'accès. Au-delà de la définition de ces modèles, il est essentiel de disposer d'implantations fiables de ceux-ci sur les systèmes d'exploitation courants pour pouvoir réaliser pleinement leurs objectifs.

Nous nous intéresserons ici spécialement aux implantations disponibles pour le système Linux. Pour commencer, nous établirons une liste des implantations relevant des modèles théoriques présentés précédemment. Ensuite, nous étudierons plus spécifiquement deux implantations : SELINUX et grsecurity.

2.2.1 Les différentes implantations pour Linux

Le noyau Linux, au cœur des systèmes d'exploitations dits GNU/Linux, est relativement jeune comparé aux modèles théoriques que nous avons étudiés dans la section précédente. Toutefois, les implantations de modèles de contrôle d'accès type MAC et RBAC n'ont commencé que tard dans l'évolution de celui-ci. En effet, le développement de Linux a débuté en 1991, et les premières implantations de mécanismes avancés de contrôlé d'accès (c'est-à-dire autres que le DAC déjà intégré au noyau) ne sont apparues que vers 1997.

Les différentes implantations présentées par la suite sont :
- Medusa DS9 ;
- RSBAC ;
- LIDS ;
- SELINUX (présenté à part dans la partie 2.2.2) ;
- grsecurity (présenté à part dans la partie 2.2.3).

2.2.1.1 Medusa DS9

Le projet [Medusa DS9 2006] est le premier à avoir implémenté une forme de MAC pour le noyau Linux, dès 1997. En fait, il reprend le modèle DTE, avec la notion de *Virtual Space* (VS) qui s'apparente aux types.

Dans Medusa, chaque objet est associé à un ou plusieurs VS. Ensuite les processus ont un ensemble de *capacités*, qui sont des associations entre un ensemble d'opération et un ensemble de VS. Ainsi, on associe des permissions d'accès entre sujets (processus) et objets via les VS.

2.2.1.2 RSBAC

Issu du travail de [Ott 1997], RSBAC (Rule Set-Based Access Control pour Linux) est une architecture générique d'implantation de modèles de sécurité s'appuyant sur GFAC [Abrams *et al.* 1990].

GFAC est un modèle d'implantation de mécanismes de contrôle d'accès dans les systèmes d'exploitation, qui à pour objectif de supporter la plupart des modèles de contrôle d'accès existants. Pour cela, les auteurs ont, dans un premier temps, distingué les composants d'une politique de contrôle d'accès : l'**autorité**, qui écrit la politique de sécurité, identifie les informations pertinentes pour la sécurité, et assigne les valeur aux attributs des ressources contrôlées ; les **attributs** décrivent les caractéristiques des sujets et objets, par exemple le type des objets, le domaine des processus, la classification, le propriétaire ; enfin les **règles** sont des expressions formelles décrivant les autorisations d'accès entre les attributs, en accord avec la politique de sécurité. Dans un second temps, les auteurs ont identifié les deux composants nécessaire à la réalisation des mécanismes de contrôle d'accès : Access Control Decision Facility (ADF), la partie responsable des décisions de contrôle d'accès, qui connait la politique de sécurité ; et Access Control Enforcement Facility (AEF), la partie qui applique les décisions prises par ADF.

Grâce à la généricité de l'architecture, héritée du modèle GFAC, de nombreux modules sont disponibles, et chacun d'eux implante une politique particulière [Ott 2001]. Dans la suite, on trouvera une liste des modules les plus intéressants par rapport aux modèles vus précédemment.

AUTH : Authenticated User Le module AUTH est le module principal de RSBAC. Il gère les permissions de changement d'UID. En fait, chaque processus a une liste d'UID possibles (capabilities).

UM : User Management Le module UM effectue une gestion des utilisateurs et groupes standard de Linux directement dans l'espace noyau. Ainsi les procédures d'authentification et de changement de mot de passe sont implantées via des appels système.

RC : Role Compatibility Le module RC effectue la gestion des rôles, suivant le modèle RBAC. Chaque rôle a des listes de rôles et types avec lesquels il peut interagir.

ACL : Access Control Lists Le module ACL implémente les listes de contrôle d'accès telles que définies par [Lampson 1971]. Ce sont des listes de triplets :

- sujet (utilisateur, rôle RC ou groupe spécial ACL)
- objets (par type : FD, DEV, PROCESS, IPC, SCD)
- types de requêtes

MAC : Mandatory Access Control Le module MAC est une implantation directe du modèle MLS. Il autorise donc l'usage de niveaux de classification et de sensibilité comme décrit dans le modèle BLP.

2.2.1.3 LIDS

Le projet LIDS [LIDS Project 2006], pour Linux Intrusion Detection System, lancé en 1999, avait pour objectif initial l'implantation d'un système de détection d'intrusion dans le noyau Linux. Finalement le travail a débouché sur un mécanisme de contrôle d'accès mandataire, dont un des intérêts est la commande de configuration, qui a une syntaxe proche de celle de `iptables` (la commande de configuration du pare-feu de Linux).

Les possibilités en matière de contrôle d'accès sont les suivantes :
- les sujets sont les fichiers binaires exécutables ;
- les objets sont tous les fichiers du système ;
- les ACL définissent quels types d'accès sont permis entre sujets et objets ;
- l'ajout ou le retrait de *capabilities POSIX* est également contrôlable.

Par exemple, pour n'autoriser l'accès en lecture à `/etc/shadow` qu'au serveur SSH, on exécutera :

```
lfs# lidsadm -A -o /etc/shadow                    -j DENY
lfs# lidsadm -A -s /usr/sbin/sshd -o /etc/shadow -j READ
```

Autre exemple, pour empêcher que le serveur apache ne tente d'utiliser un port TCP autre que le port standard 80 :

```
lfs# lidsadm -A -s /usr/sbin/httpd \
             -o CAP_BIND_NET_SERVICE 80-80 -j GRANT
```

2.2.2 SELinux

Le projet SELINUX, commencé dans les année 2000, est issu des travaux sur la sécurité du système d'exploitation DTOS [Secure Computing Corporation 1997] (à partir de 1997). Un des intérêts majeurs est que ce projet est intégré au développement du noyau Linux, et est à l'origine de Linux Security Modules.

La section sur SELINUX est organisée en deux grandes parties :

1. la structure de Linux Security Modules ;
2. les fonctionnalités de SELINUX.

2.2.2.1 Linux Security Modules

Le projet Linux Security Modules ou LSM [Wright *et al.* 2002] est un support pour l'écriture de modules de sécurité pour le noyau Linux. Il intègre dans le noyau des points d'ancrage au niveau des appels système, et via un système de *callbacks*, il rend possible l'implantation de nouveaux modèles de contrôle d'accès de façon modulaire.

LSM dérive de l'architecture **Flask** (FLux Advanced Security Kernel) qui est à l'origine de SELINUX (cf. partie "Architecture Flask").

Dans le but de fournir des mécanismes de contrôle d'accès génériques pour le noyau Linux, le projet LSM relève de deux objectifs précis : l'intégration des nouveaux points d'ancrage dans le code source du noyau (*Security Hooks*), et le support de l'empilement des modules de sécurité.

Security Hooks L'objectif premier de LSM est l'intégration de points de contrôle dans les parties critiques du noyau, c'est-à-dire les fonctions qui manipulent les entités nommées du système (processus, fichiers, sockets, mémoire partagée, IPC). Dans les structures noyau associées à ces entités, LSM ajoute un champ dédié à la sécurité, utilisé pour l'enregistrement des informations et identifiants spécifiques, notamment les labels de sécurité. Ceux-ci sont essentiels dans le procédé de déploiement d'une politique de sécurité.

Des points de contrôle (*hooks*) sont ajoutés dans les fonctions associées aux appels système, qui constituent les points d'entrée du noyau pour la manipulation des entités nommées du système. Certains sont utilisés pour créer et modifier les champs concernant la sécurité, et le reste (la majorité) sont responsables du contrôle d'accès. Normalement, ces points de contrôle interviennent après les vérifications liées au contrôle d'accès standard du noyau Linux, ainsi les permissions normales s'appliquent toujours. Une des conséquences est que les modules de sécurité implantés via LSM sont des modules *restrictifs*, ils ne peuvent qu'interdire des actions qui seraient autorisées par Linux.

Enfin, certaines fonctions de LSM sont utilisées lors de l'initialisation du noyau. Ainsi un module de sécurité peut être chargé au démarrage du système d'exploitation, et gérer proprement les activités de démarrage du noyau, par exemple pour créer les structures de sécurité associées aux processus, vérifier si le module de sécurité doit être activé ou ne pas être activé.

LSM fournit également des moyens de communication avec l'espace utilisateur, via un pseudo système de fichiers. En particulier, celui-ci contient des statistiques sur le fonctionnement du module, et l'accès aux différents paramètres de configuration existant pour chaque module de sécurité.

Empilement des modules Les modules LSM peuvent être "empilés", i.e. il est possible de coordonner les fonctions de sécurité de plusieurs modules. Cependant, ce mécanisme comporte une restriction majeure. Le premier module chargé sera considéré comme "primaire", et responsable du chargement d'un module secondaire, et ainsi de suite, jusqu'à former une chaîne de modules. Hors chacun des modules est responsable de la prise en compte des décisions d'accès produites par le suivant. Du coup, aucune garantie n'est fournie quant à l'application de la politique de sécurité décrite par un module qui ne serait pas le premier.

2.2.2.2 Security Enhanced Linux

SELINUX est un projet lancé par la NSA, et aujourd'hui totalement intégré au noyau Linux. L'objectif est l'implantation d'un contrôle d'accès de type MAC, robuste et finement configurable, au moyen de l'architecture LSM [Smalley *et al.* 2001, Loscocco et Smalley 2001].

Architecture Flask SELINUX provient de l'architecture Flask [Spencer *et al.* 1999] dont la caractéristique principale est la séparation des parties de prise de décision (*decision-making*) et d'application (*policy enforcement*). La partie décision est contenue dans un composant appelé **Security Server**, ou serveur de sécurité, qui est l'équivalent du *Reference Monitor* d'[Anderson 1972], et donc contient la politique de sécurité du système. La partie d'application est constituée de l'ensemble des points de contrôle fournis par LSM. A l'origine, la conception de Flask est issue du modèle GFAC [Abrams *et al.* 1990] (cf. 2.2.1.2), mais par la suite elle a donné lieu à l'implémentation de Linux Security Modules.

SELINUX fournit également un système de cache entre ces deux parties, appelé Access Vector Cache (AVC). Ainsi un vecteur de permissions d'accès entre un sujet et un objet donnés n'est calculé par le Security Server qu'une seule fois, et ensuite fourni directement par le cache. De plus, dans le

cas d'un changement de la politique de sécurité, le cache est bien évidemment invalidé. L'AVC a pour objectif d'améliorer les performances de SELINUX.

Security Server Le Security Server contient la configuration de SELINUX, soit la politique de sécurité. Il est responsable des décisions d'accès.

Les informations utilisées pour calculer les permissions d'accès sont les contextes de sécurité du sujet et de l'objet, et la classe de l'objet. Le résultat est un vecteur de permissions, contenant des booléens pour tous les types d'accès associés à la classe de l'objet.

Les décisions sont de deux types : décisions d'accès et décisions de transition. Les décisions d'accès concernent les tentatives d'accès d'un sujet vers un objet, pour accorder ou refuser l'accès. Les décisions de transition ont lieu lorsqu'une nouvelle entité est créée dans le système, pour calculer son contexte de sécurité (cf. paragraphe suivant). Par exemple, ces calculs ont lieu lorsqu'un nouveau processus est exécuté, ou qu'un nouveau fichier est créé.

Contextes de sécurité Afin de garantir la neutralité du mécanisme de sécurité vis-à-vis de la politique de sécurité à appliquer, Flask utilise des contextes de sécurité (aussi appelés **labels** dans le modèle BLP). Ces labels ne sont compréhensibles que par le Security Server. Du point de vue de la partie "enforcement", les tentatives d'accès d'un sujet vers un objet ne sont que des interactions entre deux contextes de sécurité.

Le format des contextes est le suivant : `<ID>:<role>:<type>`

La partie ID est liée aux UID standards de Linux, mais diffère dans le sens où il s'agit d'identifiants spécifiques à SELINUX. L'intérêt est le suivant : l'UID peut changer au cours des actions de l'utilisateur sur le système (notamment via les programmes ayant le bit *setuid* activé), mais l'ID SELINUX ne change pas, et pointe toujours vers l'utilisateur d'origine (qui a ouvert une session sur le système d'exploitation).

La partie $role$ est liée au modèle RBAC. Dans la définition de la politique de sécurité SELINUX, les utilisateurs ont accès à un ensemble de rôles, et chaque rôle représente un ensemble de $types$ manipulables. Lorsqu'un utilisateur se connecte au système d'exploitation, après s'être authentifié, il choisit un rôle parmi l'ensemble de ceux qui lui sont autorisés.

La partie $type$ prend son sens dans le modèle DTE, qui est au cœur de la définition de la politique de sécurité SELINUX.

2.2.3 grsecurity

grsecurity [Spengler 2002] est un projet démarré en 2001, qui a pour objectif global de renforcer la sécurité du noyau Linux. Son développement n'est pas intégré au noyau, et grsecurity se présente donc sous forme d'un patch pour le code source de Linux. Il fournit trois types de protection :

- confinement : pour appliquer le principe de moindre privilège aux programmes du système ;
- prévention : pour prémunir le système contre les techniques génériques d'exploitation de vulnérabilités (comme par exemple les *buffer overflow*) ;
- détection : pour surveiller l'activité du système, et généralement auditer les activités suspectes pour lesquelles il n'existe pas de moyen de prévention.

2.2.3.1 Confinement

grsecurity fournit trois mécanismes de confinement :

- Trusted Path Execution (TPE) ;

– RBAC ;
– un système d'ACL.

Le but de TPE est d'éviter que les utilisateurs exécutent des programmes auxquels l'administrateur ne fait pas confiance. C'est pourquoi le principe de fonctionnement de TPE est de n'autoriser l'exécution que pour des fichiers se trouvant dans des dossiers appartenant à *root*, et non modifiables par les utilisateurs (c'est par exemple le cas des dossiers `/bin` et `/usr/bin`).

Le système d'ACL de grsecurity ressemble à celui de LIDS. Les sujets sont identifiés par le chemin d'un fichier exécutable, et pour chaque sujet on peut définir des règles. Celles-ci spécifient les droits d'accès sur le système de fichiers, sur les connexions réseau (TCP et UDP), sur les *capabilities POSIX* et sur la quantité de ressources système utilisable (mémoire, nombre de fichiers ouverts...). Ainsi il est possible de définir des ACL soit sur des fichiers précis, soit sur des dossiers.

Le comportement par défaut est qu'un processus fils hérite des droits d'accès de son père. Par exemple, en mettant une ACL sur un dossier, celle-ci sera héritée par tous les fichiers qu'il contient, et par les sous-dossiers. Néanmoins, il existe une directive pour briser l'héritage. Si elle est spécifiée, seuls les droits définis pour ce sujet particulier seront appliquées.

Les fonctionnalités RBAC de grsecurity ont pour objectif d'autoriser la définition de différents ensembles de règles, et de les associer aux utilisateurs suivant leur rôle. Sans cela, tous les utilisateurs ont les mêmes droits, ce qui est difficilement exploitable en environnement multi-utilisateur. De plus, un utilisateur peut, s'il en a le droit, changer de rôle pendant une même session, moyennant une authentification.

Voici un exemple de règles pour le serveur apache, qui restreint l'accès en lecture aux pages web et à la configuration, l'accès en écriture au fichier de log, et qui ne donne que la *capability* `CAP_SETUID` pour qu'Apache puisse changer d'utilisateur lors de son exécution :

```
/usr/sbin/apache
    /var/www        r
    /var/log/apache w
    /etc/apache     r

    -CAP_ALL
    +CAP_SETUID
```

2.2.3.2 Prévention

grsecurity introduit dans le noyau Linux plusieurs mécanismes de prévention, dont le but est très différent de celui du confinement vu dans le paragraphe précédent. En effet, le confinement réduit l'impact d'une attaque réussie, alors que la prévention rend plus difficile l'exploitation des vulnérabilités connues, habituellement utilisées par les pirates informatiques pour attaquer un système.

– PaX a pour objectif de rendre certaines pages mémoire non exécutables, notamment la pile et le tas des processus, empêchant ainsi les attaques par dépassement de tampon (buffer overflow) ;
– ASLR (Address Space Layout Randomization) introduit de l'aléa dans le choix des adresses mémoire des processus, notamment au niveau de la pile et des librairies ;
– Enfin, grsecurity peut rendre aléatoire le choix de certains identifiants du système, notamment les identifiants de paquets IP, de processus,...

En outre, grsecurity apporte des renforcements à l'appel système `chroot()`, normalement utilisé pour changer la racine du système de fichiers pour un processus particulier. Il existe plusieurs

possibilités pour "s'évader" de la nouvelle racine, et grsecurity essaie de prévenir celles qui sont connues.

2.2.3.3 Détection

grsecurity a de nombreuses possibilités d'audit d'évènements du système, autorisant la surveillance des actions effectuées sur le système d'exploitation, et la détection d'activité suspectes.

D'une part, le système d'ACL peut définir, en parallèle des règles de *contrôle d'accès*, des règles d'*audit*, avec la même précision. Ainsi, pour tout droit d'accès pris en considération par grsecurity (lecture, écriture...), il est possible de demander l'audit de ce type d'accès, sur l'objet de l'ACL, et donc d'obtenir le résultat des demandes d'accès qui ont lieu.

D'autre part, il est possible d'auditer :
- les appels système effectués par les processus ;
- les évènements liés à PaX, notamment les tentatives d'exécution dans la pile d'un processus, souvent symptomatiques d'une attaque par *buffer overflow* ;
- et beaucoup d'autres évènements spécifiques à la sécurité dand le noyau Linux.

2.3 Langages d'expression de politiques de sécurité

Les travaux présentés dans la partie précédente décrivent des modèles de contrôle d'accès, reposant chacun sur un modèle particulier de politique de sécurité. On parle désormais de **Policy-Based Management** (PBM) [Sloman 1994, Lupu et Sloman 1999], ou administration par politique. Or dans un système d'information réparti, les administrateurs de sécurité sont amenés à utiliser ces différents modèles simultanément. Dès lors, deux questions se posent :
- Comment administrer aisément un système réparti, qui peut contenir un grand nombre d'équipements, répondant chacun à un modèle de politique de sécurité différents ?
- Comment évaluer la sécurité globale d'un système réparti où chaque politique est administrée indépendamment ?

Ainsi de nombreuses études se sont portées sur des modèles d'expression des politiques de sécurité, avec trois principales directions de travail :
- *Interopérabilité des politiques* : la sécurité globale résulte des propriétés de sécurité de chaque politique [Sloman 1994, Alpers et Plansky 1994] ;
- *Composition des politiques* : la politique globale est une combinaison de l'ensemble des politiques [Bidan et Issarny 1998, Lupu et Sloman 1999] ;
- *Autonomic management* : tous les équipements sont capables de gérer independamment leur propre politique de sécurité, en fonction d'informations disponibles dans leur environnement [Kühnhauser 1995].

Trois catégories de modèles répondent à ces problématiques d'administration de la sécurité d'un système réparti :
- **Multi-domaines** : on considère que le système réparti est découpé en différents domaines (organisation, entreprise, ensemble arbitraire d'équipements), et chaque domaine possède une politique de sécurité, mais toutes les politiques de sécurité relèvent du même modèle. Le découpage en domaine relève typiquement d'une hiérarchie, et un principe d'héritage s'applique pour la propagation des politiques.
- **Multi-politiques** : ici on considère que chaque équipement ou système possède sa propre politique de sécurité, et le modèle définit un niveau d'abstraction pour la définition d'une politique

de sécurité qui couvre l'ensemble des équipements. De plus, il gère les conflits qui apparaissent lors d'interactions entre systèmes.

– **Méta-politiques** : ces modèles considèrent que différentes politiques de sécurité sont disponibles pour des éléments d'un système réparti. Ils définissent des contraintes ou des opérations pour choisir quelles politique ou combinaison de politiques sera appliquée en fonction de l'environnement. C'est pourquoi on parle de "politiques de politiques", suivant l'expression donnée par [Hosmer 1992b].

Les parties suivantes vont détailler ces différentes catégories. Enfin, une dernière partie s'intéressera aux différents standards qui ont été définis dans le cadre de l'expression de politiques de sécurité.

2.3.1 Modèles multi-domaines

Les modèles multi-domaines apportent une solution au problème de l'administration des réseaux et systèmes distribués. Pour cela, ils introduisent de nouveaux concepts, qui vont chercher à découper le problème de façon à simplifier chaque étape, selon le principe classique utilisé en programmation, "diviser pour régner".

La notion de **domaine** d'application de politique a été formalisée pour la première fois par [Moffett *et al.* 1990]. Elle a par la suite été affinée par plusieurs articles [Moffett et Sloman 1991, Moffett et Sloman 1993b, Sloman 1994]. D'autres auteurs ont également proposé leur modèle pour l'expression de politique d'administration (*Management Policy*) [Marriott 1993, Wies 1994].

Selon [Sloman 1994], on distingue deux types de politiques :

1. Les **politiques d'autorisation** expriment ce qu'un administrateur a ou non la permission de faire. Elles posent des contraintes, d'une part, sur les informations disponibles, et d'autre part, sur les opérations qu'il peut effectuer.
2. Les **politiques d'obligation** définissent ce qu'un administrateur doit ou ne doit pas faire : elles guident ces décisions.

La suite de cette partie abordera en premier lieu la présentation des architectures multi-domaines, puis les besoins en termes d'expression des politiques de sécurité. Enfin, les implantations existantes seront présentées.

2.3.1.1 Architecture

La gestion d'un système réparti ne peut pas être complètement centralisée, mais doit au contraire être partitionnée, de façon à refléter les sous-ensembles gérés par des administrateurs différents. C'est ici qu'apparaît la notion de **domaine**. Il s'agit de regroupements d'objets répondant à des besoins communs en terme d'administration. Formellement, un domaine se définit donc comme un ensemble de référence à des objets. Un objet est dit **membre direct** d'un domaine si celui-ci contient sa référence. Les domaines peuvent également contenir des **sous-domaines**. Enfin, un objet peut appartenir à plusieurs domaines, auquel cas les domaines en question se **chevauchent**.

En complément du découpage en domaines, une architecture d'administration (**Management Architecture**) est déployée. Plutôt que d'avoir une seule application centrale, on dispose d'un ensemble de programmes d'administration qui réalisent chacun une tâche spécifique. Ainsi l'implantation de ces petits composants est plus aisée et plus rapide, du fait du nombre réduit de fonctionnalités qu'ils ont à réaliser. De plus, lorsque l'on ajoute au système distribué un nouvel équipement, il suffit de développer un composant d'administration approprié qui s'intègrera aisément dans l'architecture. L'article de Sloman donne une idée générale de cette architecture, et d'autres publications

[Sloman *et al.* 1993, Alpers et Plansky 1994] abordent le problème plus spécifique de son implantation.

2.3.1.2 Expression des politiques d'administration

Maintenant que l'on dispose de la notion de domaine, il faut l'intégrer dans l'expression des politiques de sécurité. Une première notion en ce sens est la **classification des politiques**. Toute politique est un ensemble d'informations régissant les interactions entre un *sujet* et une *cible*. Cependant, comme précisé précédemment, elles peuvent avoir différents objectifs : l'autorisation ou l'obligation. Dans ces deux cas, les politiques sont définies, soit par rapport à l'activité, elles concernent alors les autorisations d'accès des utilisateurs (lecture, écriture, exécution...), et les obligations normales (respect du quota d'espace disque, interdiction d'utiliser des logiciels non approuvés) ; soit par rapport à l'état du système, et donc il s'agit de règles d'autorisation suivant certaines variables, par exemple le fait qu'un utilisateur puisse accéder à des fichiers classifiés s'il possède le niveau d'habilitation requis, ou de règles d'obligation répondant à des évènements précis, des limites d'utilisation de certaines ressources.

Par exemple, une politique d'autorisation liée à l'activité est la règle qui définit qu'un utilisateur a le droit de lire les fichiers qui se trouve dans son compte, mais pas dans le compte de l'administrateur. Un exemple de politique d'obligation lié à l'état du système est la règle suivant laquelle les connexions au serveur Web sont refusées si cent connexions sont déjà établies.

Ensuite, les politiques contiennent des **contraintes**. Il peut s'agir de :

- *contraintes temporelles* : par exemple, ne pas autoriser un utilisateur à se connecter au système après une certaine heure ;
- *contraintes paramétriques* : par exemple, ne pas autoriser que plus de 10 utilisateurs lisent un fichier en même temps ;
- *pré-conditions* sur les actions : par exemple, obliger l'utilisateur à fermer un fichier avant d'en ouvrir un autre.

Pour simplifier la manipulation des politiques, celles-ci sont encapsulées dans des **objets** (suivant le sens donné en programmation). Il est ainsi possible de définir des *relations* entre politiques, et leur *champ d'application*, en fonction du découpage en domaines. Ensuite, lorsqu'une politique est attachée à un domaine, l'administrateur peut spécifier si les sous-domaines doivent en hériter ou pas. De plus, toute politique contient des règles indiquant à quel type d'objet elle s'applique à l'intérieur du domaine.

2.3.1.3 Implantations

Une première implantation, issue directement des travaux initiés par Sloman, est le langage **Ponder** [Damianou *et al.* 2000]. Il s'agit d'un langage déclaratif, orienté-objet, pour la définition de politiques de sécurité et d'administration. En particulier, il spécifie des règles pour l'écriture de politiques d'autorisation, politiques d'obligation événementielles, politiques de restriction, et politiques de délégation.

Law Governed Interaction [Minsky et Ungureanu 2000] (LGI) est une implantation qui vise à respecter trois principes pour la coordination entre agents dans les systèmes répartis à large échelle : 1) les politiques doivent être négociées explicitement, et non pas implicitement codées dans les agents eux-mêmes, 2) les politiques doivent être appliquées et 3) le mécanisme d'application doit être décentralisé, pour faciliter le déploiement à grande échelle.

Enfin, le modèle **Or-BAC** [Abou El Kalam *et al.* 2003, Cuppens et Miège 2003] a pour objectif de simplifier la spécification de politiques de sécurité, tout en raffinant leur expressivité. En particulier, ce modèle étend la notion de domaine à celle de l'*organisation*. Les sujets peuvent appartenir à différents organisations, et obtenir certains rôles dans chacune. Enfin les rôles sont associés à des activités, qui représentent un ensemble d'actions sur des objets.

2.3.2 Modèles multi-politiques

L'approche des modèles multi-politiques est complémentaire à l'approche multi-domaines. En effet, les modèles multi-domaines sont plutôt conçus de haut en bas, c'est à dire qu'ils vont définir une politique globale qui va ensuite se propager vers les domaines, puis vers les objets des domaines, soit les équipements et applications. En revanche, les modèles multi-politiques s'intéressent d'abord aux *sous-politiques* qui régissent les équipements et applications, et vont calculer une politique résultante globale par **composition**.

Le problème principal dans la combinaison de politiques provient de l'apparition de conflits, lorsque par exemple une première politique permet une certaine action, et pas la seconde politique avec laquelle on veut la combiner. Dès lors, il est nécessaire de définir une stratégie de résolution des conflits, qui sera la clé de la composition des différentes politiques.

2.3.2.1 Conflits dans les politiques multi-domaines

Moffett et Sloman ont proposé une approche pour la résolution des conflits [Moffett et Sloman 1993a, Lupu et Sloman 1999], lors de la définition de leur modèle d'administration multi-domaines. Ils constatent tout d'abord que la question de la résolution des conflits est habituellement laissé aux soins de l'administrateur. En effet, s'il semble facile (bien que fastidieux) pour un administrateur de résoudre d'éventuels conflits de politiques, une approche automatisée est bien plus complexe. Ils identifient deux types de conflits : les *conflits de modalités*, et les *conflits d'objectifs*.

Les conflits de modalités sont facilement détectables. Ils sont dûs à la possibilité de définir des autorisations *positives* et des autorisations *négatives* dans les politiques d'autorisation. Par exemple, un tel conflit de produit lorsque deux politiques sont fusionnées, où d'un côté une permission est accordée, et refusée de l'autre côté. Ces conflits sont simples à résoudre (cf. partie 2.3.2.2)

En revanche, les conflits d'objectifs peuvent être plus difficiles à repérer, dans le sens où ils relèvent de langages d'expression d'objectifs qui ne sont pas de simples autorisations ou interdictions. Il s'agit plutôt d'objectifs portant sur des résultats à obtenir. Du coup, ce type de conflits ne peut être résolu que par l'intervention d'un administrateur, et implique de reprendre le processus d'écrire de la politique de sécurité. En effet, un conflit d'objectif traduit souvent un conflit de plus haut niveau, par exemple un manque de coordination entre différents départements d'une entreprise, et ne peut être résolu automatiquement.

Enfin des conflits peuvent apparaître lors de la fusion de *politiques impératives* et de *politiques d'autorisation*. Par exemple, on a le conflit entre "le sujet A est obligé d'effectuer l'action X", et "le sujet A est interdit d'effectuer l'action X". Ici le conflit est facile à repérer, mais également simple à résoudre. En effet, même si la politique impérative dit que le sujet est obligé d'effectuer l'action, la politique d'autorisation empêche que l'action soit effectuée. Il est cependant préférable que de tels conflits n'apparaissent pas, par souci de lisibilité de la politique résultant de la composition.

Il existe également des conflits liés à la *séparation des privilèges*, par exemple si un même sujet est autorisé à écrire dans un fichier particulier, puis à exécuter ce fichier. Enfin on peut identifier des

conflits :
- d'intérêts, lorsqu'un administrateur est responsable de différentes organisations ;
- lorsqu'une même ressource est administrée par plusieurs personnes ;
- sur l'auto-administration, lorsqu'un utilisateur administre lui-même ses propres droits.

2.3.2.2 Modèles de résolution des conflits

Les différentes possibilités pour résoudre automatiquement les conflits sont les suivantes :
- Utiliser un langage de politique de haut-niveau qui empêche l'apparition de conflits ;
- Détecter les conflits dans les politiques par une analyse off-line ;
- Détecter et corriger les conflits lors de la propagation des politiques vers les équipements, en adaptant les politiques suivant certaines règles ;
- Détecter l'occurrence d'un conflit au moment où il se produit, avec remontée d'une question à l'opérateur pour la résolution du conflit.

Le cas des conflits de modalités est habituellement résolu par des stratégies simples : DTP (Denial Takes Precedence) et PTP (Permission Takes Precedence). Ces stratégies ont été introduites par les articles de [Jajodia *et al.* 1997] et [Bertino *et al.* 1999], puis développées plus largement dans [Jajodia *et al.* 2001]. Le but de ces stratégies est de définir une priorité entre autorisations positives (accord d'un droit d'accès) et autorisations négatives (refus d'un droit d'accès). Ainsi, si par exemple on trouve pour un même sujet et un même objet, une autorisation positive dans un première politique, et une autorisation négative dans une seconde, la stratégie de composition DTP choisira l'autorisation négative, et PTP choisira l'autorisation positive.

L'article de [Bertino *et al.* 1996] définit les notions d'autorisations "fortes" et "faibles". Dans la politique d'autorisation, toutes les règles sont associée à l'un ou l'autre. Ainsi, on identifie trois types de conflits :
- Conflits entre deux autorisations faibles : on applique DTP ou PTP ;
- Conflits entre autorisation faible et autorisation forte : la deuxième est prioritaire ;
- Conflits entre deux autorisations fortes : le langage de définition des politiques doit éviter cela, autrement la politique est déclarée inconsistante.

Les articles de [Bidan et Issarny 1997, Bidan et Issarny 1998] proposent une approche différente : ils définissent tout d'abord un langage pour décrire les différentes politiques à implanter, puis définissent des opérateurs de composition sur ce langage. Lors de la combinaison de règles en conflits, ou utilise des opérateurs DTP ou PTP. Enfin ils explicitent la notion de *politique sûre*, qui est une politique à la fois *complète* et *cohérente*. Complète signifie que la politique contient une règle pour tout couple (sujet, objet), ou si elle est le résultat d'une combinaison, toutes ses sous-politiques sont complètes. Cohérente signifie que la politique ne contient pas à la fois des accord et des refus pour une même permission d'accès.

2.3.3 Modèles méta-politiques

En complément des modèles multi-domaines ou multi-politiques, le but des approches méta-politiques et de fournir des "politiques de politiques", c'est-à-dire des politiques décrivant comment les différentes politiques disponibles doivent être utilisées ou combinées.

Deux articles de Hosmer ont introduit cette notion de méta-politique [Hosmer 1992a, Hosmer 1992b]. Selon Hosmer, les méta-politiques peuvent être utilisées, d'une part, pour spécifier explicitement les conditions d'utilisation des politiques (par exemple, informations générales,

contraintes temporelles, organisationnelles...), et d'autre part pour définir des règles de composition (relations entre politiques, coordination multi-politique...).

Un autre travail par [Avitabile 1998] s'est intéressé à l'utilisation de méta-politiques pour l'administration des politiques de sécurité, et plus spécifiquement la description du cycle de vie d'une politique. En pratique, chaque étape du cycle est accomplie par un procédé relativement complexe, qui doit gérer différentes politiques et types d'information, et implique des interventions humaines, alors que les politiques qui en résultent sont beaucoup plus simples et appliquées de façon automatique. La définition donnée pour les méta-politiques est la suivante : "Sont définies comme méta-politiques, les politiques gouvernant les phases du cycle de vie des politiques, et les politiques ayant pour cible d'autres politiques ou des procédés de gestion de politiques. Leur écriture est guidée par les besoins en terme d'administration." L'utilisation de méta-politiques contribue donc à clarifier le cycle de vie, à configurer les procédés de définition des politiques, et à cadrer les interventions des administrateurs. Ainsi, les méta-politiques peuvent être classées suivant les étapes du cycle de vie d'une politique : raffinement, délégation, intégration, conflits, exception, application. Ensuite, l'article décrit comment ces différents principes s'appliquent dans un système de PBM existant : NoCScontrol (Nomadic Computer System Control), un système de gestion de configuration d'ordinateur portable à base de politiques.

Enfin, [Belokosztolszki et Moody 2002] examine l'application des méta-politique à la description générique d'une politique, permettant de décrire une politique sans donner explicitement son contenu, et à la collaboration inter-domaine. Cet article s'intéresse à des environnements multi-domaines, dans lesquels chaque domaine est administré de façon autonome. Afin de négocier des accords d'accès aux services inter-domaines (accès depuis un domaine à un service d'un autre domaine), également appelés SLA (*Service-Level Agreement*), il est nécessaire d'échanger des informations sur les politiques de sécurité des domaines en interaction. C'est dans ce but que [Belokosztolszki et Moody 2002] propose l'utilisation de méta-politiques. Dans un premier temps, elles sont employées pour diffuser des informations sur les politiques des domaines, mais sans diffuser les détails qui doivent rester privés. C'est pourquoi ces méta-politiques contiennent les types de données, les types génériques d'objets du domaine, les fonctions, les rôles, les règles explicites et les règles invariantes. Elles ne contiennent donc pas les règles effectives de la politiques, mais sont suffisamment complètes pour être évaluées. Dans un second temps, des méta-politiques de conformité et d'interface sont établies, et sont utilisées pour évaluer les méta-politiques diffusées par les domaines. Ces méta-politiques de conformité contiennent par exemple la description des objets et des types d'accès associés, la classification des accès (lecture / modification), la description abstraite des utilisateurs...

2.4 Conclusion

Dans ce chapitre, nous avons présenté les travaux de référence des domaines du contrôle d'accès et de l'administration des politiques.

Dans le domaine du contrôle d'accès, nous avons présenté les différents modèles, suivant trois axes : DAC, MAC, RBAC. Ensuite, nous avons introduit les différentes implantations système de contrôle d'accès obligatoire pour le noyau Linux.

Dans le domaine de l'administration de politiques, nous avons vu les différents modèles existants, suivant trois familles : les modèles multi-domaines, multi-politiques et méta-politiques.

A présent, nous allons procéder à une discussion de ces différents modèles, et les étudier par rapport aux objectifs que nous avons énoncés en introduction.

Chapitre 3

Discussion des modèles existants

Le chapitre 2 a décrit, dans un premier temps, les différents modèles de contrôle d'accès et les implantations existantes, puis dans un second temps les modèles d'expression de politiques de sécurité. Afin de définir une architecture qui corresponde aux objectifs que nous avons exprimés en introduction, nous allons maintenant revenir sur ces différents modèles. Notre but est de dégager les caractéristiques essentielles des modèles de contrôle d'accès, des implantations système, et des modèles d'administration de politique, pour déterminer leur intérêt dans le cadre de notre étude.

Nous allons tout d'abord effectuer une critique des modèles de contrôle d'accès. Nous verrons que la faiblesse du DAC est communément établie. Puis nous verrons que les modèles MAC tels que BLP et Biba sont bien moins intéressants que le modèle DTE pour le cadre de notre étude. Enfin, nous reviendrons sur le modèle RBAC, qui est également indispensable pour notre architecture.

Ensuite, nous discuterons les implantations systèmes. Dans l'introduction, nous avons identifié l'objectif de réutiliser les implantations système existantes. Nous verrons que les plus appropriées sont RSBAC, grsecurity, et plus particulièrement SELINUX. Également, nous discuterons les modèles d'administration de politiques. Dans l'optique du *Policy-Based Management*, différents niveaux ont été définis : modèles multi-domaines, multi-politiques et méta-politiques.

Enfin, la synthèse discutera l'intérêt d'une nouvelle architecture méta-politique qui, contrairement aux autres, offre une approche système de la répartition des politiques : administration décentralisée par contrôle local des évolutions (pas de communications réseau pour les mises à jour), support du MAC, forte tolérance aux pannes, garanties de propriétés sur les flux d'information malgré le caractère décentralisé des évolutions (méthode de certification hors-ligne).

Ce chapitre sera organisé en quatre grandes partie : premièrement, nous critiquerons les modèles de contrôles d'accès, discrétionnaires, obligatoires et à base de rôles, ensuite nous poursuivrons sur les implantations disponibles pour le système d'exploitation Linux, puis nous discuterons les modèles de politiques de sécurité, multi-domaines, multi-politiques et méta-politiques. Enfin, une synthèse présentera nos conclusions et l'orientation de notre architecture.

3.1 Discussion des modèles de contrôle d'accès

Comme dans le chapitre précédent (2), nous allons discuter des modèles de contrôle d'accès suivant les trois grandes familles existantes. D'abord nous reviendrons sur le contrôle discrétionnaire (DAC), et donc les modèles HRU et TAM. Ensuite nous aborderons la discussion sur le contrôle d'accès obligatoire, en particulier les modèles BLP, BIBA et DTE. Enfin nous terminerons par la famille de modèle à base de rôles RBAC.

3.1.1 DAC

Le modèle de contrôle d'accès couramment utilisé sur les systèmes d'exploitation actuels est le *Discretionary Access Control*. Le DAC délègue l'accord des permissions d'accès à la discrétion des propriétaires des ressources du système. Dans le cas des fichiers, il s'agit des utilisateur et groupe propriétaire, pour toutes les autres ressources, c'est un *super-utilisateur* (par exemple, root) qui est propriétaire.

En pratique, ce modèle de contrôle d'accès a clairement montré ses limites. En effet, les attaques possibles contre les systèmes d'exploitation visent à obtenir un accès de niveau *super-utilisateur*. Lorsqu'une telle attaque est réussie, l'attaquant obtient des pouvoirs qui outrepassent le DAC et donnent un accès complet à l'ensemble des ressources du système d'information. De fait, la faiblesse de ce contrôle est que la politique de sécurité peut être à tout moment modifiée par le super-utilisateur du système d'exploitation.

De plus, diverses études[TCSEC 1985, Ferraiolo et Kuhn 1992, Loscocco *et al.* 1998] ont établi la faiblesse des modèles DAC. En effet, le contrôle d'accès discrétionnaire repose sur la capacité des utilisateurs à définir correctement les permissions sur les fichiers dont ils sont propriétaires. Toute erreur peut mener à une défaillance de sécurité, comme par exemple si le fichier `/etc/shadow` venait à être autorisé en lecture pour tous les utilisateurs.

Le modèle HRU [Harrison *et al.* 1976] établit que le problème de déterminer si le modèle de protection est *sûr*, c'est-à-dire si on a l'assurance qu'un certain droit d'accès ne sera jamais accordé à un utilisateur donné, est un problème *indécidable*. Dans le cas du modèle MTAM [Sandhu 1992], qui introduit la notion de type de sécurité, le problème est décidable. L'auteur fournit une preuve avec une complexité NP dans le cas général, polynomiale dans le cas ternaire. Cependant le modèle MTAM ôte les opérations de suppression de droits, sujets et objets (propriété de monotonie) pour obtenir ce résultat. Or, cela n'est pas envisageable dans le cadre de l'utilisation normale d'un système d'exploitation, où la suppression d'utilisateurs, d'applications, est courante.

De même, le premier article introduisant la notion de *rôle*, [Ferraiolo et Kuhn 1992], commence par justifier le fait que les modèles DAC ne répondent pas aux exigences de sécurité, des systèmes d'exploitation, même non-militaires. Enfin, les critères d'évaluation de la sécurité des systèmes [TCSEC 1985, ITSEC 1991] établissent clairement une différence de niveau entre un système implantant seulement DAC, et un système implantant également MAC.

Il ressort donc de l'analyse des modèles DAC que ceux-ci n'offrent pas de réelle garantie quant à la protection de la confidentialité et de l'intégrité des informations manipulées. Pour que la protection puisse être prouvée, il est nécessaire d'utiliser un modèle MAC. Grâce à ceux-ci, on pourra notamment restreindre les droits accordés au *super-utilisateur* (`root`), et éviter que les utilisateurs modifient les permissions de leurs fichiers de façon erronée.

3.1.2 MAC

Les modèles *Mandatory Access Control* (MAC) ont donc été envisagés pour répondre à la faiblesse des modèles DAC. Ils reposent sur la délégation du contrôle d'accès à une entité indépendante, et évitent donc que les permissions soient définies de façon incorrecte comme cela peut arriver avec le DAC. L'existence de cette entité indépendante garantit que la politique de sécurité ne soit pas modifiable directement par les utilisateurs du système d'information. On distingue deux orientations dans les modèles de type MAC.

Une première orientation est constituée par les modèles visant à assurer la confidentialité et l'intégrité des données dans les environnement militaires et commerciaux. Ces mo-

dèles [Bell et La Padula 1973, Biba 1975, Clark et Wilson 1987, Brewer et Nash 1989] répondent à des problématiques très précises, par exemple la nécessité de disposer d'une habilitation adéquate pour la lecture de documents classifiés dans BLP. Cependant, l'usage courant des systèmes d'exploitation moderne ne peut souvent pas être décrit par une seule de ces problématiques, mais relèvent plutôt de problématiques plus complexes comme le principe de moindre privilège, la confidentialité des données entre utilisateurs, la nécessité de prendre en compte les activités des utilisateurs sur le système.

Une seconde orientation est le modèle DTE [Boebert et Kain 1985]. Plutôt que de considérer une problématique particulière, celui-ci fournit un mécanisme générique, et autorise la spécification de politiques adaptées à tout environnement. S'il est possible de l'utiliser pour implanter une politique de type BLP ou BIBA, une autre utilisation envisageable est celle du confinement [Walker *et al.* 1996], qui répond notamment aux problèmes posés par les vulnérabilités logicielles, souvent présentes dans les logiciels utilisés couramment. En revanche, si le langage fourni par DTE est simple, il est difficile d'écrire une politique accordée à ses besoins. Par exemple, dans le cas où l'on souhaite confiner une application, il faudra déterminer un ensemble de droits à accorder à celle-ci ni trop restreint (pour ne pas empêcher le fonctionnement), ni trop peu restreint (car le confinement ne serait plus assuré).

Un article [Loscocco *et al.* 1998], publié par l'équipe de recherche de la NSA à l'origine de SELINUX, résume le problème posé par la sécurité des systèmes d'exploitation modernes. Il récapitule les raisons pour lesquelles il est indispensable que les systèmes d'exploitation dispose de mécanismes de sécurité obligatoire (*Mandatory Security*), divisé en politiques de sécurité de niveau contrôle d'accès, authentification et usage de la cryptographie. Il mentionne IPSec, DTE, et diverses tentatives de systèmes d'exploitation de confiance (DTOS [Secure Computing Corporation 1997], fluke [Spencer *et al.* 1999]...). Il donne ensuite des exemples concrets d'utilisation de mécanismes de sécurité, notamment concernant le contrôle d'accès. La conclusion de cet article est que la sécurité des applications ne peut être obtenue que si le système d'exploitation lui-même est sûr.

En outre, les modèles MAC fournissent une sécurité prouvable, car ils reposent sur des modèles mathématiques, souvent appuyés sur des théorèmes et des preuves. Par exemple, BLP repose sur deux lois qui empêchent la diffusion non autorisée d'information [Bell et La Padula 1973], le modèle de la Muraille de Chine [Brewer et Nash 1989] définit des classes d'objets, avec une loi régissant l'accès entre ces classes. Enfin, ces modèles ont une politique de sécurité fixée, et non modifiable par les utilisateurs, ce qui exclut les problèmes typiques d'indécidabilité du DAC. C'est essentiellement le fait de ne pas autoriser les utilisateurs à interférer avec la politique de contrôle d'accès, qui autorise le développement de preuves de fonctionnement de ces modèles, et ce genre de preuve est requis pour des niveaux de sécurité élevés [TCSEC 1985].

En revanche, l'utilisation des modèles MAC est complexe. Soit ils fournissent une politique trop restreinte, trop peu générale (BLP, BIBA, Clark-Wilson), et sont alors difficile à déployer en pratique. Soit ils fournissent des mécanismes génériques (DTE), mais alors le travail à fournir pour définir la politique de sécurité est bien plus exigeant, et n'inclut pas de garantie contre les erreurs de l'administrateur qui la définit. En définitive, le juste milieu est une combinaison des différents modèles [Loscocco et Smalley 2001].

3.1.3 RBAC

La notion de **rôle** [Ferraiolo et Kuhn 1992] simplifie grandement la définition des politiques de sécurité. En effet, plutôt que d'attribuer directement des droits à chaque utilisateur du système (un système peut en compter plusieurs milliers), on crée un petit nombre de rôles, et chaque utilisateur est assigné à un ensemble de rôles. Ensuite les droits d'accès sont associés aux rôles plutôt qu'au

utilisateurs.

Cependant, en pratique, RBAC ne se suffit pas à lui-même pour le contrôle d'accès dans un système d'exploitation. Il faut le conjuguer avec un mécanisme de contrôle d'accès de type MAC ou DAC, pour pouvoir l'utiliser. En effet, définir les attributions de droits d'accès entre sujets et objets ne fait pas partie des objectifs des modèles RBAC. Comme décrit dans l'article [Sandhu *et al.* 1996], RBAC est un composant indépendant du contrôle d'accès, complémentaire à MAC et DAC.

Prenons par exemple le choix fait dans SELINUX [Smalley *et al.* 2001] : le modèle de contrôle d'accès est en fait une combinaison de *Type Enforcement* (TE, un allègement de DTE), RBAC et MLS. Ainsi, les utilisateurs du système d'exploitation sur lequel SELINUX est déployé ont accès à un ensemble de rôles (modèle RBAC). Ensuite chaque rôle est associé à un ensemble de types qu'il a le droit de manipuler (lien entre RBAC et TE). La politique de contrôle d'accès comprend un ensemble de règles d'interaction entre types (modèle TE). Enfin MLS est utilisé pour avoir des niveaux de classification des utilisateurs et des données dans le système d'exploitation.

C'est bien la combinaison des modèles RBAC et TE qui est fondamentale dans SELINUX. Le modèle MLS est aussi intégré, pour le support des niveaux de confidentialité, mais ne constitue pas l'intérêt premier de SELINUX. En effet, la combinaison TE / RBAC autorise la définition de politiques de protection de niveau système, par une configuration très fine de l'accès aux différents appels système du noyau Linux. Par la suite, nous ne nous intéresserons donc pas à MLS puisque c'est la protection système au sens général qui nous intéresse, et pas la définition d'une politique multi-niveaux.

3.2 Implantations système

Parmi les implantations système existantes de MAC, la partie 2.2 a présenté celles qui existent pour le noyau Linux. Nous allons ici étudier les caractéristiques de ces implantations, soit Medusa DS9, LIDS et plus particulièrement RSBAC, SELINUX et grsecurity.

3.2.1 Medusa

Medusa est le plus ancien projet de MAC pour Linux. Il a démontré qu'on pouvait utiliser un modèle dérivé de DTE pour renforcer la sécurité du contrôle d'accès du système Linux. Cependant, son développement semble être arrêté aujourd'hui.

3.2.2 LIDS

LIDS est une implantation très intéressante du point de vue du confort d'utilisation. Sa commande d'administration a une syntaxe très intuitive et très puissante. LIDS s'appuie sur un modèle dérivé de DTE pour la définition des droits d'accès par processus. De plus il intègre deux fonctionnalités pour améliorer la sécurité : et .

- La première, *Trusted Path Execution*, est décrite dans [TCSEC 1985, Loscocco *et al.* 1998]. Elle pose deux contraintes sur les fichiers binaires que les utilisateurs normaux peuvent exécuter : ils doivent se trouver dans des dossiers dont le propriétaire est *root*, et ne doivent pouvoir être modifiés que par leur propriétaire (seul le propriétaire.de l'exécutable a le droit d'écriture sur le fichier binaire).
- La seconde, *Trusted Domain Execution*, peut placer dans un environnement d'exécution restreint (une sandbox) une application qui a acquis des données par des chemins non sûrs (clavier, fichier pouvant être modifiés par tout les utilisateurs). En effet, à partir du moment où une application reçoit des données par un chemin non sûr, il est possible que ces données aient été

formatées de façon à exploiter une vulnérabilité présente dans celle-ci. Dès lors, afin de prévenir tout risque de contamination globale du système par une attaque réussie, il est intéressant de placer l'application dans un environnement restreint.

Le fichier de configuration de LIDS est difficilement exploitable directement. En effet, il s'agit d'un format binaire non documenté, et il n'existe pas de librairie de fonctions de manipulations. La configuration se fait plutôt via des scripts invoquant la commande `lidsadm` (une fois pour chaque droit d'accès à accorder). D'un point de vue système réparti, il serait possible de distribuer des scripts sur l'ensemble du réseau.

3.2.3 RSBAC

RSBAC constitue une architecture très flexible. Il autorise le déploiement de modèles de contrôle d'accès très variés. Son architecture reprend les principes de GFAC [Abrams *et al.* 1990], comme SELINUX, et on retrouve donc d'une part, le composant AEF dans les appels système de Linux, et d'autre part le composant ADF qui contient une partie fixe, et les différents modèles de contrôle d'accès choisis par l'administrateur sous forme modulaire. Les modules fournis par RSBAC implantent notamment les modèles MLS et RBAC. De plus, le module ACL implante un modèle proche de DTE.

Par contre, la configuration de RSBAC se fait principalement via des outils en ligne de commande. On ne trouve pas de documentation sur les formats des fichiers de configuration. De fait, il semble difficile de configurer RSBAC simplement par déploiement d'un fichier de règles. Il est nécessaire de disposer des commandes d'administration sur tout ordinateur où l'on souhaite l'utiliser. Ceci complique un déploiement à grande échelle, car il est plus simple de répliquer un fichier sur un ensemble de noeuds, que d'exécuter un ensemble de commandes.

3.2.4 SELinux

SELINUX dispose d'une architecture claire et flexible, héritée de GFAC [Abrams *et al.* 1990] comme pour RSBAC. Mais plutôt que de fournir différents modules, SELINUX dispose d'un langage de configuration [Smalley et Fraser 2000] à la fois simple et puissant. Simple car le nombre de mots-clés est réduit, mais puissant car il implante les modèles RBAC et DTE. Ainsi il est possible de déployer différents types de politique de sécurité [Loscocco et Smalley 2001], même si SELINUX est utilisé avant tout pour faire du confinement d'application.

Il faut noter que la configuration de SELINUX repose sur l'attribution de labels, ou contextes de sécurité. L'ensemble des ressources ayant reçu le même contexte de sécurité forme une classe d'équivalence. Par rapport à des implantations comme LIDS ou grsecurity, cela signifie que la définition d'une loi d'accès se fait en deux temps : 1) attribution des labels au sujet et à l'objet correspondants, puis 2) écriture d'une règle d'accès entre les labels. Ceci implique que l'utilisation de SELINUX est moins intuitive, par rapport à la spécification directe de règles d'accès entre fichiers du système d'exploitation.

Enfin, cette simplicité du langage implique un effort plus grand pour l'administrateur. En effet, l'écriture d'une configuration de SELINUX correspondant à une politique de sécurité correcte, implique une très bonne connaissance de l'ensemble des fichiers et programmes présents sur le système d'exploitation. De fait, pour réaliser le confinement d'une application, il faut savoir précisément quels accès celle-ci va demander sur les ressources du système, et parmi ces accès, savoir dire lesquels sont légitimes, interdits ou superflus.

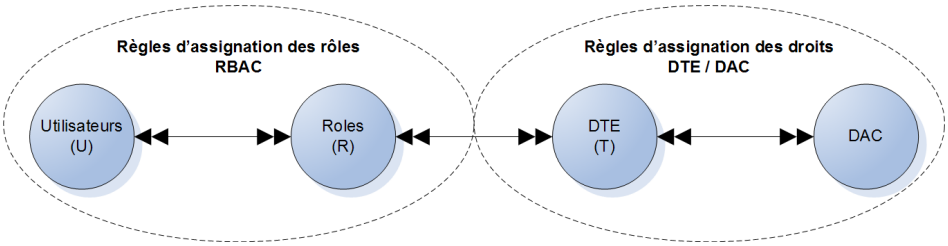

FIG. 3.1 – Coopération des modèles RBAC, MAC et DAC.

3.2.5 grsecurity

grsecurity offre des mécanismes de protection variés [Spengler 2002], certains n'ayant rien à voir avec le contrôle d'accès. Cependant la partie responsable du confinement est aussi intéressante que SELINUX, car elle implante une combinaison de RBAC et DTE. Les règles sont définies par des fichiers de configuration, dont la syntaxe est documentée. Celle-ci utilise directement les chemins des fichiers pour définir les sujets et objets, simplifiant ainsi l'écriture d'une configuration, comparativement à l'étape d'attribution des contextes de sécurité dans SELINUX.

Par contre, il est difficile de comparer l'expressivité des langages de SELINUX et grsecurity. Le premier distingue davantage de types d'accès, quasiment chaque appel système peut être accordé ou refusé, alors que le second distingue seulement une dizaine de droits d'accès, dont certains n'ont pas d'équivalent dans SELINUX. Ensuite, les contextes de sécurité de SELINUX forment des classes d'équivalence sur des objets qui peuvent être très variés, alors que, concernant les fichiers, grsecurity fonctionne uniquement par héritage des permissions des dossiers parents vers les sous-dossiers. Enfin la désignation des sujets dans les règles d'accès se fait à l'aide de contextes de sécurité dans SELINUX, et par des chemins de fichiers dans grsecurity. Du coup, SELINUX peut définir des règles pour plusieurs programmes d'un coup, tandis que dans grsecurity il faut écrire une configuration pour chaque programme.

3.3 Bilan sur le contrôle d'accès

Après avoir examiné les différents modèles de contrôle d'accès existants, il est clair qu'il est nécessaire de disposer d'une combinaison DAC / MAC / RBAC pour obtenir à la fois un contrôle d'accès robuste, et un langage de configuration clair. D'abord, DAC doit être conservé car certaines parties du système ne nécessitent pas de contrôle d'accès obligatoire (les fichiers des utilisateurs par exemple). Ensuite, MAC est nécessaire pour assurer que certains droits d'accès ne seront jamais accordés, notamment sur les parties cruciales pour le fonctionnement du système d'exploitation comme les fichiers d'authentification. Plus particulièrement, c'est le modèle DTE qui sera utilisée pour cette partie. Enfin, RBAC fournit un intermédiaire entre utilisateurs et droits d'accès, via les rôles, et il serait difficile de s'en priver.

La coopération des différents modèles est illustrée par la figure 3.1. Les utilisateurs sont associés à un ensemble de rôles. Par exemple, l'utilisateur *root* sera associé au rôle `admin_r`. Ensuite, les rôles ont le droit de manipuler un ensemble de types, ainsi le lien est fait entre les rôles et les permissions d'accès. Par exemple le rôle `admin_r` aura l'autorisation de manipuler le type `admin_t`, qui est associé aux applications d'administration. Enfin, au sein des classes d'équivalence que définissent les types, le DAC est utilisé pour toutes les parties du système ou la protection du MAC n'est par requise.

Par exemple, c'est le DAC qui s'applique lorsque l'utilisateur *root* accède aux fichiers qui se trouvent dans le dossier `/root`.

Les implantations système disponibles sont variées. Suivant les objectifs définis en 1, nous souhaitons réutiliser, au sein de notre architecture de contrôle d'accès, les mécanismes existants. Les plus intéressants sont SELINUX, grsecurity et RSBAC, car ils implémentent les modèles que nous recherchons, soit MAC/RBAC. Cette étude des implantations système a par ailleurs fait l'objet d'une publication [Blanc 2004].

L'impact de ces mécanismes de sécurité sur les performances du noyau Linux a été étudié. Chacun de ces trois projets considèrent les performances comme un point essentiel, et des travaux sont menés dans le but de les améliorer [Morris 2004, Spengler 2005, Ott 2006]. On peut donc considérer que le support de ces systèmes est un gage de performances, puisque leur implantation évolue en incluant des optimisations, notamment au point de vue du temps de calcul utilisé pour résoudre les droits d'accès.

3.4 Discussion des modèles d'expression de politiques

Les modèles d'expression de politiques étudient le problème de la spécification de politiques de haut niveau pour la description du fonctionnement d'un système d'information. Cette problématique s'inscrit dans celle du PBM (*Policy-Based Management*) ou administration par politiques. Il s'agit d'utiliser des politiques de différents types pour configurer et administrer l'ensemble d'un système d'information, comprenant donc les systèmes d'exploitation, les utilisateurs, les équipements réseau, et leur environnement. Les types de politiques sont très variés :

- autorisation / obligation
- contrôle d'accès / authentification / cryptographie
- basées sur l'activité de l'utilisateur / basées sur l'état du système

Les modèles d'expression de politiques ont pour objectif de résoudre les problématiques apparaissant dans le cadre du PBM, notamment la gestion des domaines d'administration multiples, la cohabitation de politiques incompatibles, et plus généralement la gestion du cycle de vie des politiques : déploiement, application, évolution. Nous avons identifié trois types de modèles, qui s'intéressent chacun à une partie de ces problématiques : multi-domaines, multi-politiques et méta-politiques.

3.4.1 Modèles multi-domaines

Les modèles *multi-domaines* [Moffett *et al.* 1990, Moffett et Sloman 1993b, Marriott 1993, Sloman 1994, Wies 1994, Alpers et Plansky 1994, Damianou *et al.* 2000, Minsky et Ungureanu 2000, Abou El Kalam *et al.* 2003] fournissent des solutions aux problèmes d'interactions entre domaines au sein des systèmes répartis. La notion de domaine est utilisée pour regrouper les ressources ou acteurs qui répondent à des critères d'administration similaires. Par exemple on peut envisager qu'une entreprise ou qu'une organisation constitue un domaine séparé, de même que les stations de travail, ou des ensembles de serveurs, représentent des zones d'administration distinctes.

Ainsi les modèles autorisent le découpage d'un système d'information en domaines, puis la définition de politiques avec un certain champ d'application, c'est-à-dire le ou les domaines sur lesquelles la politique doit être déployée. A cet effet, l'encapsulation des politiques dans des objets permet aisément d'y associer des informations telles que relations entre politiques, champ d'application, contraintes temporelles ou événementielles.

Nous avons ainsi identifié trois implémentations : Ponder, LGI et Or-BAC.

3.4.1.1 Ponder

Ponder [Damianou *et al.* 2000] est conçu pour définir des politiques de sécurité à l'échelle de grands systèmes répartis, dans un langage abstrait. Ces politiques sont ensuite converties en configuration pour les équipements au moyen d'agents de traduction. Toutefois, ces politiques spécifient des actions de haut niveau, et ne sont pas prévues pour s'appliquer sur des mécanismes de contrôle d'accès du type MAC ou DTE.

Le langage Ponder inclut également la définition de méta-politiques, mais celles-ci concernent la gestion de conflits entre politiques, ou des contraintes sur l'application d'ensemble de politiques sur un même élément, ou d'application d'une politique sur un ensemble d'éléments. Par exemple, ces méta-politiques peuvent contrôler que le nombre d'éléments sur lesquels une politique est appliquée ne dépasse pas un maximum. Il ne s'agit pas de contrôler la mise à jour de ces politiques.

3.4.1.2 Law Governed Interaction

LGI [Minsky et Ungureanu 2000] vise plutôt à résoudre les problématiques de négociation de politiques mutuelles lors d'interaction entre systèmes d'information. Par exemple, lorsque deux entreprises veulent partager des ressources, elles vont partager leur politique LGI et automatiquement, une politique mutuelle sera générée et appliquée de chaque côté. LGI ne concerne donc pas l'administration d'un domaine de protection, mais bien les relations entre domaines. Ceci est hors du cadre de notre étude.

3.4.1.3 Or-BAC

Or-BAC [Abou El Kalam *et al.* 2003, Cuppens et Miège 2003] a pour objectif d'ajouter une notion contextuelle à l'écriture des politiques de sécurité, notamment via l'intégration de la notion d'*organisation*. Il devient ainsi possible d'écrire des politiques prenant en compte les interactions intra-organisation, rattachant la notion de rôle à celle de l'emploi dans l'organisation, avec un contrôle fin des héritages de permissions dans la hiérarchie des emplois. Il existe également un modèle d'administration des politiques Or-BAC, il s'agit de AdOr-BAC [F. Cuppens et A. Miège 2003, F. Cuppens et A. Miège 2004].

Ici encore, il ne s'agit pas de contrôler la mise à jour des politiques. Notamment le système ne présente pas de contraintes d'évolution, ni de garanties sur les flux d'information (cf. section 3.5. Enfin, le modèle AdOr-BAC s'intéresse exclusivement à l'administration des rôles.

3.4.2 Modèles multi-politiques

La problématique *multi-politique* découle directement de la précédente. Lorsque des domaines d'administration se chevauchent, ou dès lors qu'il est nécessaire d'appliquer plusieurs politiques sur un même domaine, il apparaît des conflits dans la *composition* ou l'*interopérabilité* des politiques. Les modèles multi-politiques ont pour objectif de fournir des langages et des lois pour la résolution des conflits et l'application de politiques différentes sur un même domaine ou équipement [Moffett et Sloman 1993a, Lupu et Sloman 1999].

Les règles de résolution des conflits sont appelées des stratégies. Les plus simples d'entre elles sont PTP et DTP, dans le premier cas la stratégie est la priorité des permissions d'accès (*permission takes precedence*), et dans le second la priorité des interdictions (*denial takes precedence*). D'autres modèles plus complexes fournissent des langages pour la définition de véritables stratégies gérant les conflits au cas par cas [Bertino *et al.* 1996].

Cependant, ces modèles imposent toujours de réécrire des politiques pré-existantes dans un nouveau langage qu'ils fournissent, car les stratégies de résolution de conflits ne peuvent opérer que sur ces langages [Bidan et Issarny 1997, Bidan et Issarny 1998]. De plus, ces modèles n'apportent pas de réponse au problèmes de garantie de propriétés de sécurité, ni au support d'évolution répartie des politiques de contrôle d'accès de type MAC. En effet, ils ne s'intéressent qu'au problème de la fusion de politiques de sécurité existantes.

3.4.3 Modèles méta-politiques

Dans les premiers articles introduisant ce concept [Hosmer 1992a, Hosmer 1992b], le but des modèles *méta-politiques* est de décrire comment les différentes politiques doivent être utilisées ou composées. Il s'agit donc de politiques "de haut niveau" qui réglementent l'utilisation des politiques "de bas niveau". Toutefois, dans ces articles, l'utilisation concrète de ces méta-politiques reste assez floue. Les articles concernant les modèles multi-domaines tels que [Sloman *et al.* 1992, Marriott 1993, Wies 1994] fournissent également des outils de caractérisation et de manipulation des politiques, comparables aux méta-politiques. Le terme est même cité dans [Sloman 1994], et intégré au langage Ponder [Damianou *et al.* 2000]. Il s'agit là de cas d'applications des concepts de méta-politiques beaucoup plus concrets que les notions données par [Hosmer 1992a], bien que leur emploi se limite à la définition de contraintes sur la gestion des ensembles de politiques.

Des études ultérieures s'intéressent à la notion de méta-politique, dans le but d'en développer les usages. Selon [Avitabile 1998], les méta-politiques peuvent fournir des informations sur le cycle de vie des politiques (raffinement, délégation, conflits, exceptions, tests, application). Toutefois, il ne s'agit pas de gérer l'évolution d'une politique de sécurité au sens où nous l'entendons, c'est-à-dire avec des lois sur l'ajout ou le retrait de droits d'accès sur des sujets et objets. L'article considère plutôt des procédés d'administration de politiques de haut niveau, et la façon dont ces procédés interviennent au cours de la vie de la politique. Les méta-politiques ne décrivent donc pas directement les modifications possibles, mais plutôt les conditions d'application des procédés (raffinement, délégation, conflits, exceptions, tests, application) aux politiques existantes.

Le modèle de [Belokosztolszki et Moody 2002] s'intéresse à la gestion des interactions entre domaines d'administration indépendants. Les méta-politiques sont utilisées ici pour décrire des interfaces de politiques, et des conditions de compatibilité, de façon à donner suffisamment d'information sur les politiques d'un domaine, sans révéler le contenu exact. L'objectif est de pouvoir établir automatiquement des accords de service entre domaines (SLA). Comme dans le cas de LGI, ce modèle s'attache à résoudre des problèmes de négociation de politique.

3.4.4 Bilan sur les modèles d'expression de politique

Pour notre architecture, nous souhaitons avoir une administration complètement répartie des politiques de sécurité, tout en garantissant des propriétés globales à tout le système réparti. Ceci exclut des modèles multi-domaines du type [Sloman 1994, Wies 1994, Alpers et Plansky 1994, Damianou *et al.* 2000, Minsky et Ungureanu 2000, Abou El Kalam *et al.* 2003] qui s'intéressent davantage à des aspects organisationnels qu'à des aspects système.

Quant aux modèles multi-politiques [Moffett et Sloman 1993a, Bertino *et al.* 1996, Bidan et Issarny 1998], ils n'abordent que les problèmes de coopération / composition de politiques, et la façon de résoudre de façon automatisée des conflits simples (par exemple des conflits entre permission et interdiction). Ces conflits sont résolus par des stratégies comme DTP ou PTP.

Nous souhaitons donc réutiliser le concept de méta-politique, mais avec une signification différente de celle vue précédemment. En effet, par rapport aux modèles décrits par [Hosmer 1992a, Sloman 1994, Avitabile 1998, Damianou *et al.* 2000, Belokosztolszki et Moody 2002], notre méta-politique évite l'apparition de conflits. Cela se traduit par des contraintes sur l'évolution des politiques.

Les modèles multi-domaines, multi-politiques et méta-politiques ne traitent pas des politiques système ou de MAC, ni d'administration décentralisée, ni de tolérance aux pannes. De plus, aucun de ces modèles ne garantit l'absence de flux d'information illégaux entre contextes de sécurité.

Pour établir un parallèle avec le concept de méta-politique posé par [Hosmer 1992a], nos politiques de bas-niveau seront les configurations des mécanismes de contrôle d'accès de type SELINUX ou grsecurity. Nos politiques de haut niveau décriront les évolutions possibles de ces politiques, via un ensemble de contraintes sur leur modification. En résumé, nos méta-politiques contiendront une configuration initiale des politiques de sécurité, ensuite ces politiques évolueront librement sur tous les nœuds du système d'information, sous contrôle de la politique de modification.

3.5 Synthèse et orientations de notre architecture

Nous avons établi la nécessité pour notre architecture de supporter les modèles de contrôle d'accès DAC / MAC / RBAC. Nous disposons également d'implantations système correspondantes à ces modèles, soit SELINUX, grsecurity et RSBAC. De plus, nous avons étudié différents modèles d'expression de politiques de sécurité, répondant à la problématique d'administration des systèmes répartis. Nous avons alors proposé un nouveau modèle de méta-politique, qui conviendra à nos besoins de contrôle des modifications de la politique de protection.

En effet, la combinaison des modèles DAC / MAC / RBAC ne suffit pas pour les besoins d'une administration système décentralisée. L'administration d'un mécanisme comme SELINUX est difficilement contrôlable. En effet, tout utilisateur qui a la possibilité de modifier la politique peut intégrer de nouvelles règles à sa guise, et notamment des règles abusives ou même dangereuses pour la sécurité du système. Toutefois, il est nécessaire de pouvoir faire évoluer la configuration du contrôle d'accès. Par exemple, l'installation de nouvelles application ou la définition de nouveaux rôles d'utilisateur sont des événements courants dans la vie d'un système d'exploitation. Nous devons donc définir un modèle de contrôle de l'évolution de la politique de protection, afin de supporter une politique dynamique, tout en garantissant des propriétés de sécurité.

Hors, l'article de [Harrison *et al.* 1976], qui définit le modèle HRU, démontre que le problème de la sûreté d'un système de protection dynamique, i.e. qui autorise la création et la suppression de sujet et d'objet, et la modification de la matrice d'accès, est *indécidable* (cf. partie 2.1.1.1). Dès lors, il peut sembler vain de vouloir contrôler l'évolution d'une politique de protection dynamique.

Cependant, il est important de noter que le modèle HRU relève de l'approche DAC, et autorise donc tout sujet à créer de nouveaux sujets et objets et à modifier la matrice d'accès. Dans notre modèle, nous souhaitons intégrer l'approche MAC, et la seule conception du MAC garantit qu'il s'agit d'un modèle de protection sûr au sens donné par [Harrison *et al.* 1976]. En effet, le problème de savoir si un droit d'accès r peut être "transféré" au cours de l'évolution de la politique de protection est résolu, car le MAC n'autorise en aucun cas la modification de la politique de protection par les utilisateurs normaux du système d'exploitation. Dans un modèle commeHRU, il n'est pas envisageable de traiter des problèmes de flux d'information, étant donné qu'on ne peut déjà pas garantir l'absence d'accès direct entre deux contextes.

Nous souhaitons intégrer le support de politiques dynamiques et la possibilité de garantir l'absence

de flux illégaux, et ce malgré le caractère décentralisé des évolutions des politiques. Ceci est rendu par un mécanisme de contraintes sur les modifications de la politique de protection, qui garantit que l'évolution est contrôlée. Cela permet donc de contraindre les états possibles des politiques, et de vérifier, au moyen des contraintes, que ces états n'autorisent pas de flux illégaux. Comme nous le verrons dans le chapitre 6, dans notre modèle de protection, le problème de certification d'absence de flux illégaux est décidable. Ceci est prouvé par le fait que nous proposons un algorithme effectuant cette vérification.

Contrairement aux modèles multi-domaines, multi-politiques et méta-politiques existants, notre approche présente une politique de modification. C'est le déploiement de cette politique sur un système réparti, qui, à la différence des autres modèles :

- Autorise l'administration décentralisé, puisque les modifications de politique sont effectuées localement sur chaque noeud ;
- Offre un niveau élevé de tolérance aux pannes, puisque les mises à jour ne requièrent aucune communication réseau ;
- Intègre des systèmes MAC et DAC, via la définition d'un langage neutre de politique de protection ;
- A de bonnes performances, puisque d'une part, elle évite l'apparition de conflits, et d'autre part, elle bénéficie des optimisations réalisées dans les implantations de MAC, notamment pour le noyau Linux ;
- Garantit, par une méthode hors-ligne, l'absence de flux illégaux, et ce malgré le caractère dynamique des politiques de protection, et le caractère décentralisé des évolutions.

3.6 Conclusion

Dans ce chapitre, nous avons discuté des différents modèles de contrôle d'accès, DAC, MAC, RBAC, puis des implantations système, et enfin de modèles d'administration de politiques, qui sont soit multi-domaines, soit multi-politiques, soit méta-politiques. D'une part, nous avons établi que les approches système n'offrent pas de moyen d'administration répartie. D'autre part, nous avons montré que les approches d'administration de politiques concernent des aspects organisationnels, issus pour la plupart du monde des bases de données ou des relations entre domaines d'administration distincts. Ils ne fournissent pas les caractéristiques nécessaires à de l'administration système, à savoir l'administration décentralisée, la tolérance aux pannes, l'intégration de mécanismes système de contrôle d'accès, les performances et le support de méthodes de vérification d'absence de flux d'information illégaux. C'est pourquoi nous nous orientons vers la spécification d'une nouvelle architecture de méta-politique.

Chapitre 4

Spécification de l'architecture méta-politique

Dans le chapitre précédent, nous avons discuté les modèles de contrôle d'accès et d'administration de politiques présentés dans l'état de l'art, et nous sommes arrivés à la conclusion suivante : nous allons définir une nouvelle architecture de méta-politique, supportant l'intégration de mécanisme de contrôle d'accès obligatoire. Les propriétés recherchées pour notre modèle sont les suivantes : administration décentralisée, tolérance aux pannes, intégration de mécanismes de contrôle d'accès, bonnes performances et support de méthode de vérification de l'absence de flux d'information illégaux. Ce dernier aspect sera en fait abordé dans le chapitre 6.

Ce chapitre présente donc la spécification de notre nouvelle architecture de méta-politique. Dans un premier temps, nous présenterons les principes de fonctionnement de l'architecture. Il s'agit d'une architecture distribuée, présentant deux niveaux de contrôle : le contrôle d'accès, configuré par une politique de protection écrite dans un langage neutre, et le contrôle de l'évolution de la politique de protection, par une politique de modification écrite dans un langage de contrainte. Un mécanisme de projection est utilisé pour traduire le langage neutre vers la configuration des mécanismes de contrôle d'accès, SELINUX et grsecurity. Un agent de mise à jour, au coeur de l'architecture, réalise l'ensemble de ces fonctions sur chacun des noeuds où notre architecture est déployée.

Ensuite, nous verrons comment ces principes de fonctionnement répondent aux objectifs de définition de notre architecture. En particulier, nous verrons les propriétés du modèle de méta-politique, puis comment le contrôle réparti des modifications de politiques est utilisé pour obtenir une administration décentralisée. L'intégration des mécanismes de MAC et les performances de l'architecture seront étudiés par la suite.

Enfin, nous exposerons les principes algorithmiques de fonctionnement de l'agent de mise à jour. En particulier, les étapes de démarrage, de chargement de la méta-politique et de réception de requêtes de mise à jour seront détaillées.

La spécification de l'architecture de Méta-Politique a par ailleurs fait l'objet de plusieurs publications [Blanc *et al.* 2004b, Blanc *et al.* 2004a, Blanc *et al.* 2005a, Blanc *et al.* 2005b].

4.1 Principe de la Méta-Politique

Cette section présente le principe de fonctionnement de l'architecture Méta-Politique. Dans un premier temps, la partie *Architecture distribuée* présentera le principe global de fonctionnement. Puis

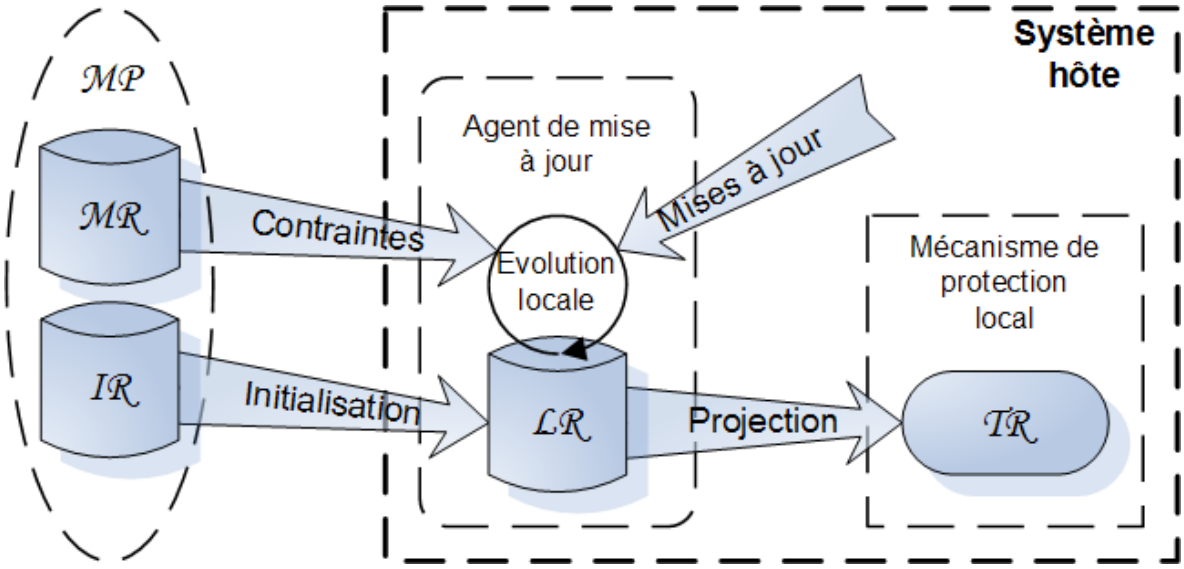

FIG. 4.1 – Schéma de l'architecture Méta-Politique.

les sections suivantes présenteront de façon plus détaillées les différents composants : l'agent de mise à jour, la politique de protection, la politique de modification et enfin le traducteur de politique.

4.1.1 Architecture distribuée

Le principe de notre architecture de contrôle est qu'elle s'applique à l'échelle d'un système réparti. Ainsi, sur chaque nœud intégré dans cette architecture, les composants logiciels nécessaires à son fonctionnement sont installés. Ces composants sont détaillées dans les parties suivantes. Leurs objectifs sont triples : assurer le déploiement de la méta-politique, garantir le respect de la politique de modification et autoriser l'évolution locale de la politique de protection.

Pour ce faire, la méta-politique $\mathcal{MP}$ est déployée sur chacun des nœuds par un moyen sûr (copie au moment de l'installation, réplication par disquette ou clé USB, communications réseau chiffrées). Tous les nœuds doivent recevoir la même méta-politique, sinon les propriétés de sécurité apportées par la méta-politique ne seront pas assurées sur l'ensemble du système réparti.

La figure 4.1 présente la Méta-Politique. Elle comprend deux parties, d'une part, la politique de protection initiale $\mathcal{IR}$, et d'autre part, la politique de modification $\mathcal{MR}$. La politique de protection initiale $\mathcal{IR}$ sert à initialiser la politique de protection locale du nœud $\mathcal{LR}$, lors de la réception de la Méta-Politique par le nœud. Ensuite, des requêtes de mise à jour de la politique sont reçues par l'agent de mise à jour (cf. partie suivante). Celles-ci sont comparées avec les contraintes de modification fournies par $\mathcal{MR}$. Si la requête respecte les contraintes de modification, alors la politique locale $\mathcal{LR}$ est mise à jour en ajoutant, supprimant ou modifiant une règle de $\mathcal{LR}$.

La politique locale $\mathcal{LR}$ est projetée vers une politique $\mathcal{TR}$, spécifique au mécanisme de contrôle d'accès disponible sur le nœud cible. Par exemple, la politique locale $\mathcal{LR}$ peut être projetée en des règles SELINUX pour un système cible qui fournit les services de contrôle d'accès de SELINUX. Dans ce cas, la politique $\mathcal{TR}$ consiste en des règles SELINUX.

Ainsi, la méta-politique ne fait que contraindre l'évolution de la politique de protection locale $\mathcal{LR}$ de chaque machine. Les nœuds peuvent donc appliquer des modifications différentes, et ainsi

les nœuds sont indépendants les uns des autres. Dès lors, l'architecture autorise des opérations du type installation d'une application particulière sur un seul nœud avec une mise à jour cohérente de la politique locale de protection $\mathcal{LR}$.

4.1.2 Deux niveaux de contrôle

La définition de deux niveaux de contrôle est étroitement liée à la définition de la méta-politique. En effet, son but est d'assurer simultanément le contrôle d'accès sur l'ensemble des nœuds d'un système réparti, et de garantir que les évolutions locales des politiques de contrôle d'accès ne violent pas les objectifs de sécurité définis globalement. Ainsi, les deux niveaux sont définis par deux politiques de sécurité distinctes.

Contrôle d'accès La *politique locale de protection* $\mathcal{LR}$ est composée de règles concernant le contrôle d'accès, et elle évolue localement sur chaque nœud du système réparti (cf. partie 4.1.4). En effet, parmi les actions courantes d'administration d'un système d'exploitation, on trouve l'installation de nouvelles applications, et la gestion des comptes des utilisateurs. L'installation de nouvelles applications implique la modification de la politique $\mathcal{LR}$, car par défaut la politique n'autorise aucun accès aux programmes inconnus. De plus, les applications installées peuvent différer d'un nœud à l'autre. Les règles présentes dans la politique $\mathcal{LR}$ sont liées aux applications installées, ainsi cette politique est différente d'une machine à l'autre.

Contrôle des modifications de la politique La *politique de modification* $\mathcal{MR}$ garantit les propriétés de sécurité globales. Elle est déployée de façon identique sur chaque nœud, et ne peut pas être modifiée par les nœuds(cf. partie 4.1.5). Elle contient des règles de contrainte sur l'évolution de la politique locale de protection $\mathcal{LR}$ de chacun des nœuds. Son champ d'application est l'ensemble du système réparti où l'architecture est déployée. Ainsi, dans le cadre de l'activité normale des nœuds, les opérations d'évolution de $\mathcal{LR}$ ne peuvent à aucun moment violer la politique de modification. En pratique, l'agent de mise à jour (cf. partie 4.1.3) assure que chaque requête de modification de $\mathcal{LR}$ sera vérifiée par rapport à ce second niveau de politique, et n'aboutira que si une règle de modification, présente dans cette politique, autorise explicitement cette mise à jour.

4.1.3 Agent de mise à jour

Le composant au cœur de l'architecture est l'agent de mise à jour. Installé sur chacun des nœuds du système réparti où s'applique notre architecture, il est responsable pour toutes les opérations qui touchent à l'administration de la politique locale de protection $\mathcal{LR}$. Cette politique étant projetée, *par l'agent*, en une politique $\mathcal{TR}$ pour configurer le mécanisme de MAC sous-jacent, il va sans dire que l'agent est directement responsable de la sécurité du nœud. Le schéma de déploiement de l'agent est représenté sur la figure 4.2.

Comme on le voit sur la figure, les différentes opérations dont l'agent est en charge sont :

- la réception de la Méta-Politique $\mathcal{MP}$ sur le nœud ;
- le chargement et le maintien en mémoire de la politique de modification $\mathcal{MR}$;
- le chargement de la politique de protection initiale $\mathcal{IR}$, sa copie vers la politique locale $\mathcal{LR}$ et son application sur le mécanisme de contrôle d'accès cible par la politique $\mathcal{TR}$;
- la réception des demandes de mise à jour de la politique locale $\mathcal{LR}$;
- la validation des demandes par rapport à la politique de modification $\mathcal{MR}$;
- la sauvegarde de ces modifications dans la politique locale $\mathcal{LR}$;

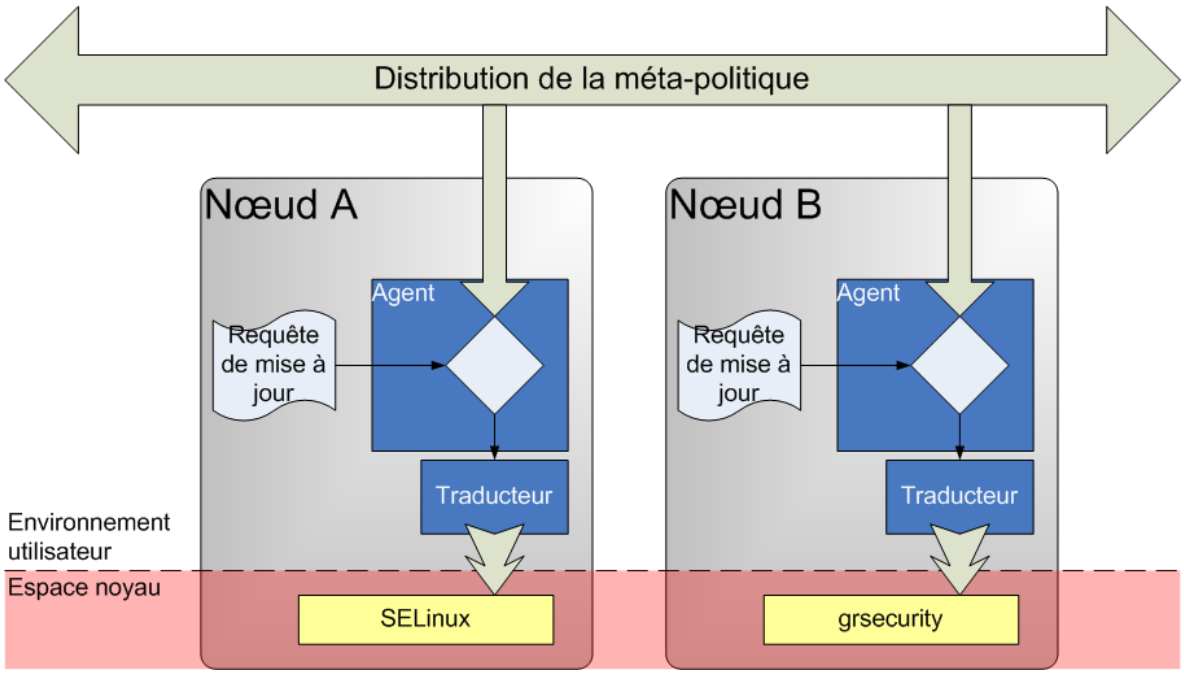

FIG. 4.2 – Répartition des agents sur les nœuds.

– enfin, l'application de la politique locale modifiée sur le nœud, via la projection en $\mathcal{TR}$.

L'agent commence son exécution au démarrage du système d'exploitation du nœud, et ne se termine qu'à l'arrêt de ce dernier. En cas de panne de l'agent, un composant standard du système d'exploitation sera chargé de le relancer. C'est donc la présence de l'agent de mise à jour sur l'ensemble des nœuds qui garantit l'application de l'architecture de contrôle.

4.1.4 Politique de protection

La politique de protection $\mathcal{LR}$ contient un ensemble de règles de contrôle d'accès, spécifiques à un nœud. Afin que notre architecture soit indépendante du mécanisme de contrôle d'accès disponible localement, les règles dans $\mathcal{LR}$ sont exprimées sous la forme d'un langage neutre, projeté sur $\mathcal{TR}$, dans le langage de configuration du MAC. Ainsi intervient notre politique de protection :

– Elle est écrite dans un langage neutre, afin de s'affranchir des spécificités des différents mécanismes de MAC.
– le langage neutre est assez riche pour être traduit vers les différents mécanismes de MAC. Il inclut la désignation des sujets, des objets, et la caractérisation des permissions d'accès.

Les politiques de protection sont associées à des représentations sous forme de matrice d'accès, comme décrit par Lampson [Lampson 1971]. Ces matrices d'accès décrivent les permissions d'accès existant entre sujets et objets d'un système d'exploitation. Les règles sont écrites sous la forme `<sujet, objet, permissions>`. Grâce à cette forme, notre politique de protection peut s'appliquer au MAC sous-jacent via un moteur de traduction dépendant du mécanisme utilisé localement sur le nœud.

4.1.5 Politique de modification

La politique de modification $\mathcal{MR}$ est l'apport de notre architecture par rapport aux modèles de contrôle d'accès existants comme MAC ou RBAC.

Cette politique contient donc des règles décrivant directement les modifications autorisées ou refusées dans la politique de protection. L'agent de mise à jour la consulte à chaque demande de modification qu'il reçoit. Il peut ainsi en déterminer la légitimité. Enfin, par défaut, toute modification qui n'est pas explicitement autorisée ou refusée est par défaut refusée, afin de respecter le principe du moindre privilège en sécurité.

Les règles de modification sont écrites par rapport au langage neutre de politique de protection, ainsi que les demandes de modification de la politique de protection qui sont faites auprès de l'agent de mise à jour. Ainsi le contrôle par rapport à la politique de modification s'effectue également dans les termes de ce langage. Enfin, les ajouts de règles validées sont effectués dans la politique locale de protection, avant d'être répercutés dans le mécanisme de MAC cible via la projection.

Les modifications possibles sont de plusieurs ordres. Tout d'abord, les demandes peuvent concerner l'ajout de nouveaux contextes et de nouvelles permissions, et donc l'ajout de vecteurs autorisant une interaction entre deux contextes. Ensuite, il peut s'agir de modifications dans des vecteurs existants. Enfin, la demande peut être le retrait de contextes ou de vecteurs.

4.1.6 Traducteur de politiques

Le dernier composant de l'architecture est le traducteur de politiques (Language Adapter). Il réalise l'étape finale dans le processus de notre architecture de contrôle, soit l'application locale des règles sur les nœuds. En fait, il est intégré à l'agent de mise à jour, et celui-ci lui transmet la politique de protection à chaque fois qu'il est nécessaire de la traduire, soit au démarrage de l'agent, puis à chaque modification de cette politique.

Le comportement de ce composant dépend essentiellement du mécanisme de MAC sous-jacent. Ce dernier est indiqué lors du chargement de l'agent, car il ne peut pas être changé dynamiquement (en effet, le changement du mécanisme de MAC requiert le remplacement du noyau du système d'exploitation).

La traduction de la politique neutre est faite par une méthode de projection, c'est-à-dire que la traduction va être faite au mieux suivant l'expressivité du langage de configuration du mécanisme de sécurité cible. En effet, le langage neutre est capable de désigner les contextes de sécurité et les droits d'accès de façon indépendante du mécanisme cible. Lors de la traduction, ces contextes et droits d'accès seront projetés en fonction des fonctions implantées dans le mécanisme cible.

Par exemple, SELINUX est capable de désigner tout appel système du noyau Linux, et de l'interdire pour un contexte sujet donné, mais ne sait pas reproduire le mode *hidden* existant dans grsecurity pour cacher un fichier ou un dossier. A l'inverse, grsecurity peut désigner des droits comme la lecture, l'écriture ou l'exécution, mais la granularité de ces droits ne va pas jusqu'à l'ensemble des appels système.

4.2 Propriétés de l'architecture

Cette partie détaille les propriétés de notre architecture, qui on été choisies en fonction des besoins et orientations définies dans en fin du chapitre 3, dans la partie 3.5 :

1. Propriétés liées à notre modèle méta-politique : objectifs globaux de sécurité, contrôle réparti, hétérogénéité des systèmes cible, réutilisation de modèles de protection ;

2. Présence de deux niveaux de contrôle ;
3. Evolution locale des politiques de sécurité ;
4. Intégration du MAC ;
5. Performance.

4.2.1 Méta-politique

L'environnement d'application de notre modèle est le système réparti en général. En particulier, les architectures réparties sur lesquels la gestion de politiques de contrôle d'accès est envisagée sont :
- les clusters et grilles de calcul, qui utilisent donc des applications distribuées ;
- les fournisseurs d'accès et de services Internet, qui souhaitent fournir des prestations de sécurité à leurs clients.

La notion de méta-politique, telle que nous la proposons dans cette étude, convient à ce type de besoin. Elle offre différentes propriétés qui couvrent bien les besoins des différents domaines d'application concernés, celles-ci sont détaillées dans les parties suivantes.

4.2.1.1 Définition d'objectifs globaux de sécurité

Description : Comme nous l'avons vu dans la section 4.1, l'architecture de méta-politique s'applique à l'ensemble d'un système réparti. La politique $\mathcal{MR}$ contient des règles de contrainte sur la mise à jour des politiques locales de protection. Ainsi, l'architecture contrôle l'évolution ce ces politiques, et décrit donc des objectifs globaux de sécurité. Ces objectifs concernent le contrôle des *flux d'information* dans le système d'exploitation.

Définition 4.2.1 (Flux d'information) *On appelle flux d'information tout transfert de données entre une entité active (sujet) et une entité passive (objet), ou entre deux sujets.*

Entre sujets et objets, l'orientation du flux dépend du type d'accès. Un accès en lecture induit un flux d'information de l'objet vers le sujet. Un accès en écriture induit un flux d'information du sujet vers l'objet. Entre sujets, l'orientation du flux dépend de l'opération.

Enfin, entre contextes de sécurité sans interaction directe, un flux d'information peut exister par transitivité des flux unitaires, c'est-à-dire qu'une suite d'interactions peut conduite à l'existence d'un flux.

Les règles de modification de la politique $\mathcal{MR}$ expriment des contraintes sur les droits d'accès qui peuvent être ajoutés dans la politique de contrôle d'accès $\mathcal{LR}$. $\mathcal{MR}$ contrôle donc la création de nouveaux flux d'information dans le système d'exploitation.

En outre, notre architecture supporte l'évolution locale des politiques de protection sur chaque nœud du système réparti. Cela signifie que chacun des nœud intègre un mécanisme de modification de sa propre politique de sécurité. C'est pourquoi il est nécessaire de définir des objectifs globaux de sécurité. Ils régissent les possibilités de modifications de la politique de protection locale. L'objectif est que, quel que soit le nœud, il existe des contraintes pour assurer que certains flux d'information ne seront jamais autorisés.

Besoin associé : La possibilité de définir des objectifs globaux de sécurité découle directement du besoin d'administration répartie, clairement identifié pour notre architecture. En effet, l'environnement d'application étant le système réparti, il est nécessaire de disposer de mécanismes pour administrer l'ensemble du système réparti de façon automatisée. Toutefois, comme on accorde à chaque nœud

le pouvoir de faire évoluer sa propre politique de contrôle d'accès, il ne faut pas que cela induise de faiblesse dans la sécurité.

Si notre modèle n'incluait pas la politique de modification $\mathcal{MR}$, et malgré la présence de MAC sur les nœuds, tout administrateur aurait le droit de modifier la politique de contrôle d'accès à sa guise. Il pourrait par exemple autoriser la lecture du fichier `/etc/shadow` à tous les utilisateurs, ce qui remettrait en cause la sécurité du mécanisme d'enregistrement des mots de passe des utilisateurs.

C'est pourquoi notre Méta-Politique inclut des objectifs globaux de sécurité (cf. exemples ci-dessous). Bien que l'administration de la politique des nœuds soit répartie, la politique $\mathcal{MR}$ assure que ces objectifs de sécurité seront respectés sur tous les nœuds.

Usages pratiques : Pour un cluster ou une grille, il peut s'agir d'autoriser à un nœud d'exécuter un nouveau code de calcul. Bien que déployé dynamiquement, ce nouveau code doit pouvoir fonctionner, c'est-à-dire accéder aux ressources systèmes dont il a besoin. Toutefois il ne doit pas violer les propriétés de sécurité visées sur tout le système, et ceci est garantit par la politique $\mathcal{MR}$.

Dans l'exemple de la figure 4.3, l'objectif de sécurité est d'empêcher certaines applications d'accéder directement à Internet. Même si un firewall est présent, le nœud doit contenir une politique garantissant que seules les applications autorisées puissent accéder à Internet tout en empêchant les autres de le faire. Un simple firewall est incapable de garantir seul qu'une application n'usurpera l'identité d'une autre, en s'attribuant par exemple l'usage des ports permis pour sortir vers Internet. Il s'agit en fait d'autoriser ou d'interdire un flux d'information entre un contexte sujet et un port. C'est pourquoi sur la figure 4.3, les applications 1 et 2 sont bloquées par la politique de contrôle d'accès. Lorsque la nouvelle application 3 est déployée, initialement elle n'a pas non plus le droit d'accéder à Internet. Les objectifs globaux de sécurité autorisent que cette nouvelle application accède à Internet. Par conséquent, à l'étape 2, lorsque de nouvelles règles de protection sont ajoutées par l'administrateur dans $\mathcal{LR}$, l'application 3 acquiert l'autorisation d'accéder à Internet, mais les applications 1 et 2 sont toujours bloquées.

Le principe est donc que le déploiement d'une nouvelle application est suivi de l'ajout de règles dans la politique locale $\mathcal{LR}$ des nœuds où celle-ci est installée. En particulier, ces règles autoriseront ou non l'application à accéder au réseau externe.

Pour un fournisseur d'accès et de services, la définition d'objectifs globaux de sécurité autorise une gestion souple de la sécurité de ses clients. En effet, même si le client souhaite effectuer l'administration de ses propres applications, le fournisseur qui héberge ces applications est tenu de garantir la sécurité de son réseau et de ses autres clients. Sur l'exemple de la figure 4.4, le fournisseur héberge des applications d'un client sur un serveur qu'il fournit. Le client souhaite administrer les applications qu'il déploie sur ce serveur. Avec la méta-politique, il est possible de déployer une politique de contrôle d'accès, qui va assurer la sécurité de base des serveurs, et le *cloisonnement* des applications du client.

Par exemple, sur la figure 4.4, le client a installé des applications $S1$ et $S2$. Celles-ci sont autorisées a accéder aux ressources $O1$, mais pas aux ressources Web. Quant au serveur Web présent sur la machine, il est autorisé à accéder aux ressources Web, mais pas à $O1$. Ainsi, la politique de protection forme un compartiment pour les applications $S1$ et $S2$, qui sont cloisonnées par rapport au serveur Web. Le client reste responsable de la sécurité de ses propres applications, car il a la possibilité de faire des requêtes de mise à jour de $\mathcal{LR}$. Mais avec le contrôle imposé par la politique $\mathcal{MR}$, en aucun cas une défaillance de la part du client ne compromettra le serveur où l'architecture de méta-politique a été déployée.

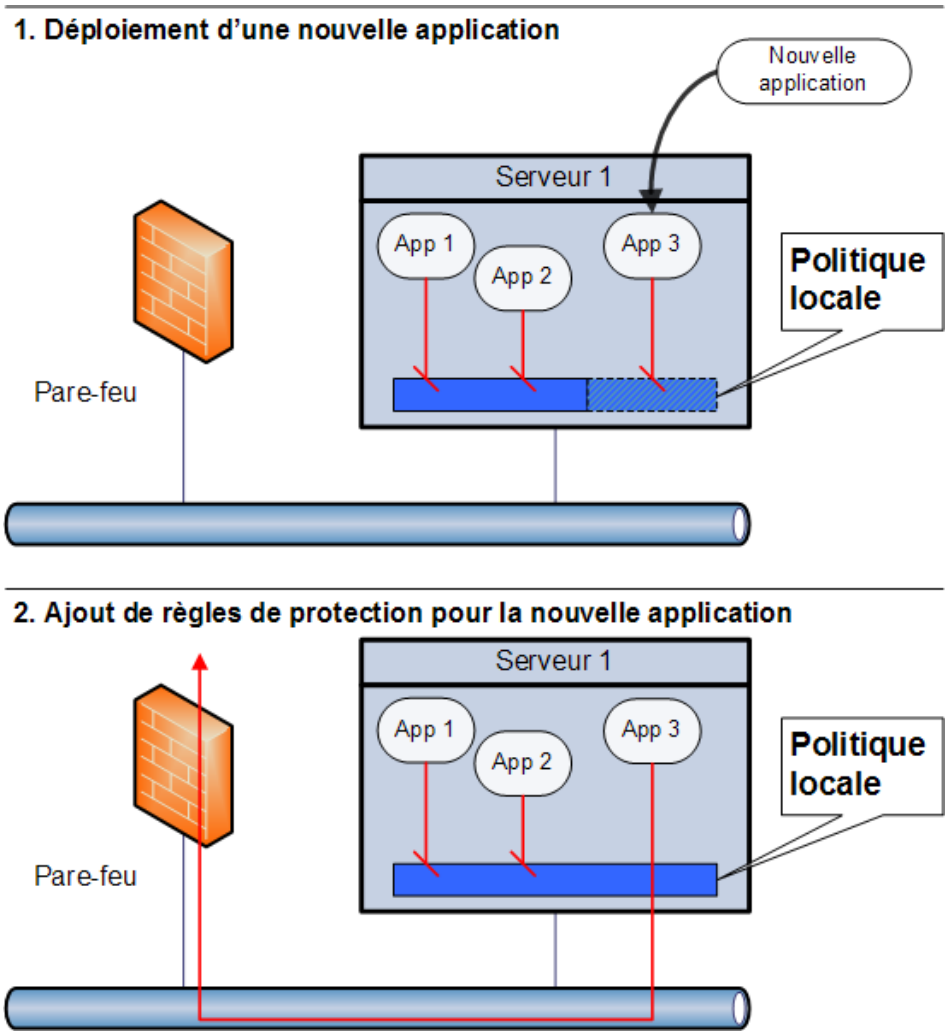

FIG. 4.3 – Garantie de propriétés globales de sécurité sur un cluster.

4.2.1.2 Contrôle réparti

Description : Chaque nœud peut demander des évolutions spécifiques de la politique locale de protection $\mathcal{LR}$ suivant ses propres besoins. La méta-politique garantit que le contrôle de l'évolution de la politique est effectif sur chaque nœud, suivant les règles contenues dans $\mathcal{MR}$. Comme l'illustre la figure 4.5, le nœud contrôle de façon décentralisée que les évolutions demandées sont légitimes, et ne requiert aucune communication par le réseau pour réaliser ce contrôle. Ainsi, la configuration du contrôle d'accès de chacun des nœuds évolue indépendamment de celle des autres nœuds, dans la limite des contraintes fournies par $\mathcal{MR}$. Cette propriété de contrôle réparti est illustrée par la figure 4.5.

Besoin associé : La propriété de contrôle réparti répond au besoin de tolérance aux pannes ou aux défaillances du système. Par exemple, la nécessité d'un serveur de contrôle pour les évolutions de la politique peut conduire à un point de fragilité du système. Un attaquant pourrait viser ce point central et le rendre inopérant pour empêcher les mises à jour nécessaires pour les politiques de pro-

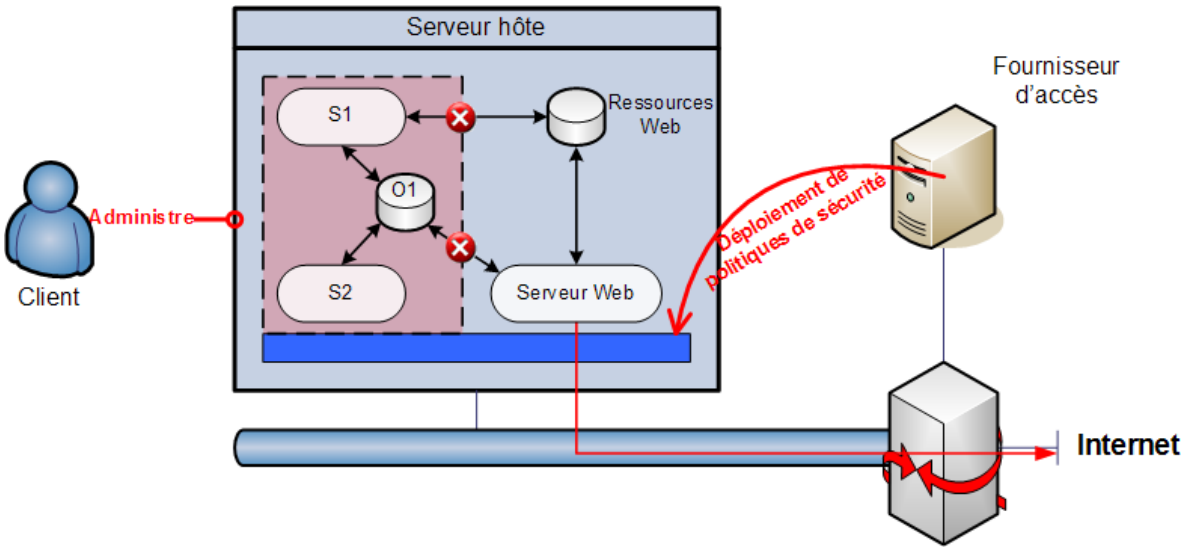

FIG. 4.4 – Garantie de propriétés globales de sécurité pour un fournisseur.

tections. Avec un contrôle réparti, il n'existe pas de composant central qui décide des modifications de la politique de sécurité. Chacun des nœuds est responsable du maintien de sa propre configuration. Toutefois, même si chaque nœud peut demander les modifications qu'il souhaite, celles-ci ne sont appliquées que dans la mesure où elles sont approuvées par les contraintes de la politique $\mathcal{MR}$. Ainsi notre architecture peut à la fois garantir la tolérance aux pannes et les objectifs globaux de sécurité.

Usages pratiques : Pour un cluster ou un grille, le contrôle réparti assure que la sécurité des nœuds reste garantie même lors de pannes importantes sur le réseau. Sur la figure 4.6, on observe le cas d'un cluster dans lequel le réseau est segmenté à la suite d'une panne. Dès lors, certains nœuds se retrouvent isolés. S'il s'agissait d'une administration centralisée, il ne serait plus possible d'administrer leur politique de sécurité. Mais comme on peut le voir sur la figure 4.6, grâce à la répartition du contrôle des modifications de la politique de sécurité, les actions d'administration des nœuds peuvent toujours être effectuées. Il n'y a pas de dépendance vis-à-vis d'un composant central du réseau.

Pour un fournisseur de services, il peut advenir que ses clients soient temporairement déconnectés du réseau. Pour continuer à assurer leur sécurité, le fournisseur installe des moyens de contre-mesure aux attaques. Ici le contrôle réparti obtenu par la méta-politique intervient à deux niveaux :

- Le contrôle réparti autorise les mécanismes de contre-mesure à faire évoluer localement la politique de sécurité en fonction des menaces repérées sur le système du client. Par exemple, supposons qu'une application malicieuse soit installée malencontreusement par le client (virus, cheval de Troie). Le mécanisme de détection repère alors cet indésirable, et déclenche une contre-mesure consistant à interdire l'exécution de cette application. Ceci est effectué par une requête de mise à jour de la politique de protection.
- Les objectifs globaux de sécurité, définis sous forme de contraintes dans la politique de modification, régissent les évolutions de la politique de sécurité, suivant les demandes de mise à jour qui sont faites par les applications d'administration. Même si le contrôle de l'évolution de la politique est décentralisé, ces contraintes assurent que le nœud ne sera jamais placé dans un état où il contiendrait des règles illégitimes. Par exemple, un mécanisme de détection d'intrusion malicieux pourrait tenter de modifier ses propres droits d'accès, pour avoir accès à Internet.

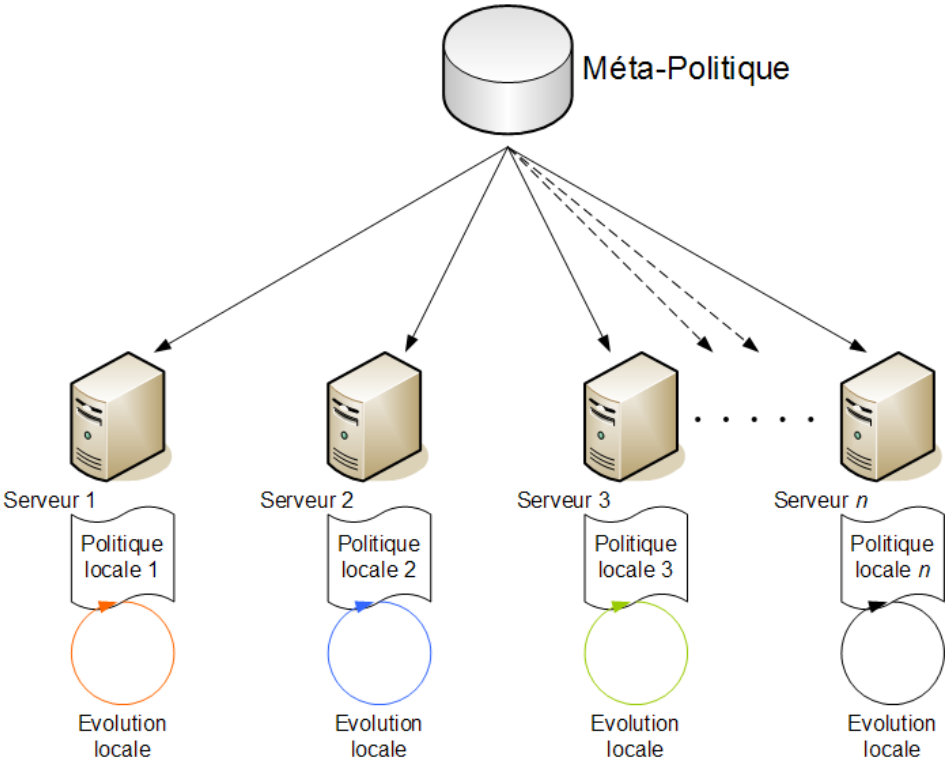

FIG. 4.5 – Evolution locale des politiques de sécurité.

Grâce à la présence de l'architecture méta-politique, ce mécanisme ne sera pas autorisé à modifier les parties de la politique qui le concernent. En effet, la politique de modification $\mathcal{MR}$ ne contient pas de règle autorisant le mécanisme de détection d'intrusion à modifier elle-même les règles de protection qui la concernent.

4.2.1.3 Hétérogénéité des systèmes cibles

Description : La méta-politique est suffisamment abstraite pour ne pas faire d'hypothèse sur le système sous-jacent, sur lequel elle est appliquée. Ainsi, le système cible peut être composé de nœuds présentant des configurations hétérogènes, notamment d'un point de vue contrôle d'accès. La méta-politique sera alors identique sur chacun de ces nœuds, et servira à calculer la politique locale. Cette dernière étant écrite dans un langage neutre, il est possible de la projeter sur des systèmes hétérogènes. De fait, l'introduction de langages neutres vis-à-vis des systèmes de contrôle d'accès sera un des points essentiels pour le support de systèmes cibles hétérogènes.

Besoin associé : Étant donné que nous ne souhaitons pas implanter un nouveau mécanisme de contrôle d'accès (cf. 1), notre architecture doit être capable de gérer des mécanismes existants. C'est pourquoi à la fois notre architecture et la méta-politique associée supportent des systèmes cibles hétérogènes. Ainsi, l'application de la méta-politique est possible via différents mécanismes de contrôle d'accès, tels que SELINUX et grsecurity, ou même vers du DAC, offrant ainsi le support de plates-formes comme Microsoft Windows. Chacun d'eux possède des caractéristiques spécifiques, mais les fonctions qui leur sont communes fournissent une base suffisante pour le déploiement de notre archi-

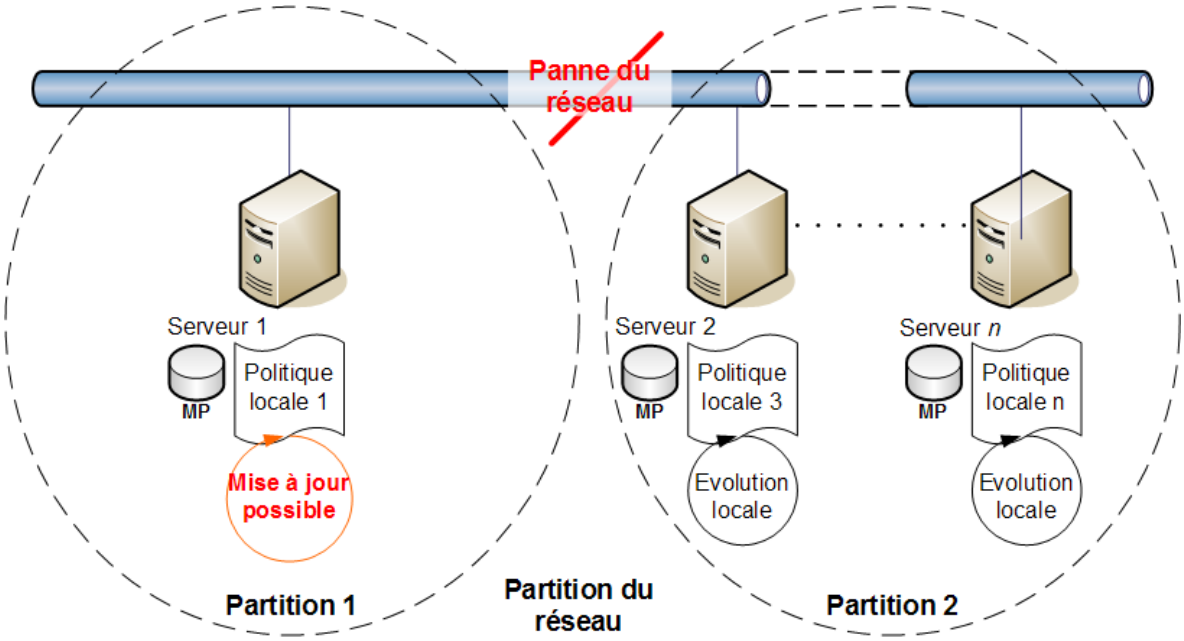

FIG. 4.6 – Evolution locale de la politique lors d'une panne de réseau.

tecture. Ces fonctions sont la désignation des sujets et des objets sur le système d'exploitation, et la définition des actions autorisées ou interdites. Toutefois, la projection de la politique neutre sera plus ou moins complète suivant les possibilités du système cible.

Usages pratiques : Pour une grille de calcul, le support de systèmes cibles hétérogènes autorise la garantie d'objectifs de sécurité sur l'ensemble des systèmes d'exploitation rattachés à la grille. En effet, les grilles de calcul comprennent souvent des machines aux environnements logiciels très différents, car il s'agit de systèmes ayant d'autres usages en temps normal, notamment lorsque les grilles incluent des stations de travail (les calculs sont par exemple effectués la nuit, lorsque les stations ne sont pas utilisées). L'architecture de méta-politique pourra être déployée de façon transparente sur l'ensemble de ces systèmes.

Pour un fournisseur d'accès, les clients peuvent avoir des mécanismes de sécurité qu'ils ont choisis, et qu'ils administrent eux-mêmes. La méta-politique peut alors être déployée quel que soit le système d'exploitation du client. Ainsi, les contraintes de sécurité seront appliquées au maximum des fonctionnalités du système d'exploitation cible, les éventuelles impossibilités de projection seront signalées à l'usager. Par exemple, dans le cas de la projection sur un système comme Windows, qui comporte un ensemble plus restreint de droits d'accès que SELINUX, certaines règles de contrôle trop spécifiques ne pourront pas être projetées.

4.2.1.4 Réutilisation de modèle de protection

Description : La possibilité d'exprimer la politique de protection $\mathcal{LR}$ sous une forme neutre permet de réutiliser les modèles de protection. Ainsi, une méta-politique qui a été écrite dans un environnement spécifique, par exemple une entreprise A, pourra être directement appliquée dans une entreprise B qui aurait les mêmes besoin en terme de sécurité. Le langage neutre est un avantage dans ce sens.

En effet, toute politique décrite dans ce langage peut potentiellement être appliquée sur n'importe quel système, via l'architecture de méta-politique.

Besoin associé : Comme expliqué en 3.1.2, l'écriture de politiques de sécurité est un procédé long et complexe. C'est pourquoi il est intéressant de pouvoir réutiliser une politique existante. Ce principe est d'ailleurs déjà utilisé pour le développement des politiques de sécurité de SELINUX. En effet, la configuration distribuée avec ce mécanisme de contrôle d'accès a été composé par un grand nombre d'utilisateur, sur un plan international. Les configurations faites pour une application spécifique sont distribuées librement, autorisant ainsi au fur et à mesure la construction d'une configuration pour un système d'exploitation entier.

Usages pratiques : Pour les grilles de calcul, le support de la réutilisation de modèles de protection rend possible, par exemple, le changement du mécanisme de sécurité présent sur les nœuds. En effet, souvent les clusters comportent des systèmes d'installation automatisée. Ainsi, lors d'un changement dans la configuration de sécurité des nœuds, le déploiement de la méta-politique est transparent.

Pour un fournisseur d'accès, l'architecture de méta-politique autorise le déploiement d'une méta-politique unique chez tous les clients, quel que soit le système d'exploitation. Ainsi, le fournisseur n'a besoin d'écrire qu'une seule méta-politique, suivant les objectifs de sécurité de son réseau, ou de son infrastructure. C'est cette méta-politique qui sera implantée dans le système d'exploitation de chaque client.

4.2.2 Evolution locale des politiques de sécurité

La faculté d'évolution locale de la politique de contrôle d'accès sur chacun des nœuds répond à deux besoins. Premièrement, elle est nécessaire pour que le contrôle d'accès soit totalement réparti. Afin de s'affranchir de la présence d'un composant central d'administration, chaque nœud du système est responsable de l'administration de sa propre politique de contrôle d'accès (figure 4.5). Cependant, comme vu dans la partie précédente, il n'est pas totalement libre, puisque l'évolution de la politique locale est toujours soumise à la validation par la méta-politique. En résumé, il doit être suffisamment libre pour ajouter les permissions d'accès nécessaires aux applications, et en même temps être contraint de façon à satisfaire les objectifs globaux de sécurité. Enfin, cette liberté du nœud l'autorise à réagir en cas de détection locale de tentative d'intrusion, et ainsi contribue à renforcer sa sécurité.

Le deuxième point qui nécessite l'évolution locale des politiques de contrôle d'accès est la tolérance aux pannes. Il peut arriver qu'un nœud soit soudainement isolé du reste du système réparti. Dès lors, s'il n'est pas capable de faire évoluer sa politique, les objectifs de sécurité ne sont plus garantis. En conséquence, l'introduction des évolutions locales de la politique garantit une tolérance aux pannes maximale, sachant que cette évolution reste dans tous les cas sous la contrainte de la méta-politique.

4.2.3 Intégration du MAC

Avec la prise en compte des systèmes répartis, le second besoin directeur de notre modèle est l'intégration de systèmes d'exploitation de confiance, ou *Trusted Operating Systems* (TOS). Il s'agit de systèmes intégrant des mécanismes de contrôle d'accès obligatoire. Comme cela a été souligné dans l'état de l'art, l'utilisation de MAC implique que les utilisateurs normaux ne peuvent pas modifier la politique de contrôle d'accès. Ceci signifie que le système d'exploitation fournit des moyens pour compartimenter les applications, et que la base d'exécution (le noyau) est considérée imperméable

aux tentatives d'attaques dont les applications font l'objet. Le noyau intégrant le mécanisme de MAC forme alors une base de confiance ou TCB (*Trusted Computing Base*).

La notion de TCB représente la capacité du système à garantir que seules les applications autorisées à interagir auront la capacité de le faire. En effet, les mécanismes de contrôle d'accès intégrés dans un TOS permettent de créer des compartiments au sein du système. Des applications placées dans des compartiments différents ne pourront jamais avoir accès l'une à l'autre.

En outre, il est crucial que notre modèle possède un rôle particulier dans ce TOS. En particulier, l'agent de mise à jour a la responsabilité de l'administration du MAC, il doit donc avoir les droits nécessaires et suffisants pour interagir avec celui-ci, et être le seul autorisé à faire le lien entre le niveau méta-politique et le niveau contrôle d'accès.

Nous avons vu que plusieurs mécanismes de MAC existent d'ores et déjà, et que leur sécurité, à défaut d'avoir été prouvée, a été largement éprouvée. C'est pourquoi notre modèle ne définit pas un tel mécanisme, mais va plutôt intégrer ceux qui existent déjà. Notre modèle est donc capable de manipuler les structures utilisées par le MAC, i.e. les sujets, objets et contextes de sécurité, au travers des langages que nous allons définir dans la partie 5..

4.2.4 Performance

Un des objectifs à ne pas négliger est le faible impact de l'architecture de contrôle d'accès sur les performances des nœuds. C'est dans ce but que le principe d'agent a été retenu, notamment par rapport à quatre points.

Premièrement, le principe d'agent de sécurité permet de réaliser une architecture légère d'un point de vue logicielle. En effet, sur chaque nœud est présent un agent autonome, responsable de la gestion de la politique de contrôle d'accès. Il sera la seule entité autorisée à accéder au mécanisme de MAC intégré au noyau. Il reçoit donc les demandes de modification de la politique de sécurité, les valide par rapport à la politique de modification, et les applique si elles sont légitimes. Parmi les autres modèles d'architecture possibles, nous aurions pu choisir des systèmes de répartition d'objets tels que CORBA, ainsi qu'il est fait dans DSI [Pourzandi *et al.* 2003], ou l'ajout de fonctionnalités dans le noyau du système d'exploitation. Cette seconde solution était délicate à mettre en œuvre, d'abord parce que les mécanismes de MAC ont d'ordinaire une interface avec des outils en espace utilisateur, et ensuite parce que l'intégration de nouveaux mécanismes dans le noyau est délicate, notamment du fait des contraintes sur la programmation et les fonctions utilisables. La première solution n'a pas été retenue en raison de la relative abondance de fonctionnalités d'un système comme CORBA, alors que les besoins de notre architecture sont assez limités en terme de communications réseau.

Deuxièmement, par rapport aux modèles multi-politiques étudiés en partie 2.3.2, notre modèle ne vise pas à résoudre les conflits apparaissant lors de la combinaison de politique. Chaque noeud possède sa propre politique. Notre architecture exclut donc l'impact sur les performances dû à la résolution de conflits.

Ensuite, notre architecture de sécurité ne nécessite pas le maintien de connexions actives entre les différents nœuds. En temps normal, l'agent est autonome et aucune communication sur le réseau n'est nécessaire à son fonctionnement. En effet, une fois que la méta-politique initiale a été obtenue, l'agent possède l'ensemble des informations requises pour administrer la politique de sécurité.

Enfin, l'agent va utiliser les interfaces des composants connus et optimisés pour de bonnes performances. Notamment, les mécanismes de MAC tels que SELINUX ou grsecurity ont été largement étudiés de ce point de vue, et garantissent un faible impact sur l'utilisation des ressources des nœuds. Notre architecture ne doit donc pas perturber ces composants, nous cherchons par le biais de notre agent à imiter les outils habituels de gestion qui leur sont associés.

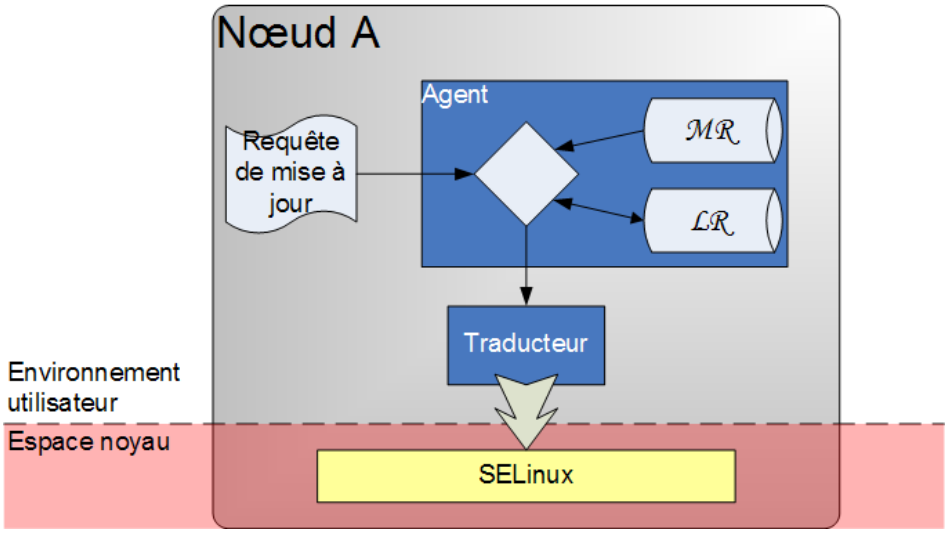

FIG. 4.7 – Schéma interne de l'agent de mise à jour.

4.3 Principe algorithmique de l'architecture Méta-Politique

Cette section aborde plus précisément les principes algorithmiques décrivant le fonctionnement de l'architecture, et notamment de l'agent de mise à jour. Une première partie rappellera rapidement la composition de l'architecture Méta-Politique. Ensuite les deux parties suivantes décriront de façon plus précise l'étape du démarrage d'un nœud sous contrôle de notre architecture, et la procédure de requête de mise à jour de la politique de protection.

4.3.1 Vue globale

L'architecture de contrôle est déployée sur l'ensemble d'un système réparti. Chacun des nœuds du système réparti contient un agent de mise à jour, avec deux politiques associées :

- la politique de modification $\mathcal{MR}$, qui est identique sur l'ensemble des nœuds ;
- la politique locale de protection $\mathcal{LR}$, qui évolue indépendamment sur chaque nœud.

Les nœuds peuvent présenter des mécanismes de MAC différents, et dans tous les cas, le traducteur de politique effectue le lien entre politique de protection et configuration du MAC.

La figure 4.7 présente la structure interne de l'agent de mise à jour. On observe la présence des politiques $\mathcal{MR}$ et $\mathcal{LR}$, conservée en mémoire par l'agent. Le nœud A inclut SELINUX, et l'agent utilise le module de traduction adapté. Enfin, lorsque l'administrateur du nœud A souhaite une modification de la politique locale, il envoie une requête de mise à jour à l'agent, comme illustré par la figure 4.7.

4.3.2 Démarrage

Lors de l'installation de chacun des nœuds, notre agent a été déployé ainsi qu'une copie de la méta-politique $\mathcal{MP}$. Celle-ci contient la politique de modification $\mathcal{MR}$, ainsi que la politique de protection initiale $\mathcal{IR}$. Cette dernière est copiée pour initialiser $\mathcal{LR}$, qui va ensuite évoluer suivant les demandes de modification qui seront faites.

Un fichier de configuration indique à l'agent où se trouve la méta-politique, et quel est le mécanisme de MAC présent localement sur chaque nœud.

Lors de son exécution, l'agent de mise à jour charge la méta-politique, et conserve en mémoire la politique de modification et la politique de protection. Ensuite il appelle le traducteur pour configurer la couche MAC. A partir de la politique en langage neutre, celui-ci produit une configuration, et l'injecte dans le mécanisme de contrôle d'accès par l'interface standard du mécanisme cible.

4.3.3 Requête de mise à jour

Dans le cadre du fonctionnement normal d'un système d'information, il est courant d'avoir besoin d'accorder de nouvelles permissions d'accès. Deux cas de figure peuvent notamment se produire :

- dans le cas de l'installation d'une nouvelle application, celle-ci va requérir l'accès à différents composants standard du système d'exploitation (librairies, réseau). De plus, les utilisateurs doivent pouvoir accéder à cette nouvelle application. Un ensemble correspondant de nouvelles permissions d'accès sera donc demandé par l'application à notre agent de mise à jour. Globalement, le but est d'autoriser les utilisateurs existants à utiliser cette nouvelle application ;
- dans le cas de l'ajout d'utilisateurs, certains peuvent nécessiter des accès particuliers, notamment si les activités sont regroupées dans des rôles distincts. Le but est alors d'ajouter des permissions pour que de nouveaux utilisateurs puissent accéder à des applications existantes.

L'application ou l'utilisateur qui est en charge de demander ces nouvelles permissions construit alors une série de requêtes de mise à jour, suivant un format dérivé de la politique de modification. Il transmet ensuite ces requêtes à l'agent de mise à jour, suivant un protocole dédié. L'agent les reçoit et déclenche alors le traitement approprié.

Dans un premier temps, il va rechercher dans la politique de modification une règle correspondant à la demande qu'il vient d'obtenir. il va commencer par rechercher des règles concernant les sujets et objets, puis ensuite va contrôler des permissions d'accès qui ont été demandées entre ceux-ci. Si aucune règle ne correspond, alors la demande est rejetée. Par contre, si une règle de modification autorise la requête considérée, alors l'agent va l'intégrer à la politique de protection locale. La recherche s'arrête dès qu'une règle correspondante est trouvée.

Afin d'intégrer les nouvelles permissions d'accès dans la politique de protection locale, l'agent va rechercher si des règles ayant des sujets et objets identiques existent déjà. S'il s'agit d'un ajout, il va rechercher un vecteur de permissions existant avec des sujets et objets existants, et ajouter les permissions d'accès demandées, ou sinon créer le vecteur adéquat. S'il s'agit d'une modification, il va rechercher un vecteur correspondant et modifier les permissions qu'il contient. Enfin, s'il s'agit d'une suppression, il va rechercher le vecteur correspondant et supprimer les permissions correspondantes, voire supprimer le vecteur s'il ne reste aucune permission.

Pour terminer l'opération, l'agent va transmettre la nouvelle politique de protection locale au Language Adapter. Ce dernier va alors traduire depuis le langage neutre vers le langage de configuration du mécanisme de MAC sous-jacent, et appliquer la nouvelle configuration ainsi produite.

4.4 Conclusion

Dans ce chapitre, nous avons exposé la spécification de notre architecture de Méta-Politique. Dans un premier temps, nous avons présenté ses principes de fonctionnement, avec notamment l'utilisation de deux politiques : une politique de protection, et une politique de modification.

Dans un second temps, nous avons montré comment l'architecture répond aux objectifs annoncés en introduction. En particulier, nous avons illustré les apports de la Méta-Politique en matière de garanties de propriétés globales de sécurité, de contrôle réparti des évolutions, de support de systèmes hétérogènes et de réutilisation de modèles de protection. Également, les propriétés d'administration décentralisée, d'intégration de mécanismes de MAC et de performance ont été détaillées.

Enfin, nous avons présenté les principes algorithmiques de fonctionnement de l'agent de mise à jour, qui réalise le chargement de la méta-politique et le contrôle des modifications de la politique de protection.

Maintenant, nous avons terminé la spécification de notre architecture. Avant de pouvoir définir des méthodes de vérification d'absence de flux d'information, nous allons procéder à la formalisation de notre modèle de méta-politique dans le chapitre suivant. Après cela, le chapitre 6 développera ces méthodes de vérification.

Chapitre 5

Formalisation du modèle de Méta-Politique

Après avoir introduit notre contribution de façon informelle, nous allons maintenant établir un modèle formel de notre modèle de méta-politique. Ce modèle formel aura notamment comme intérêt de poser les fondements mathématiques de notre proposition et permettra de lever toute ambiguïté sémantique. De plus ce modèle formel pourra être utilisé pour appliquer des techniques formelles de validation, de vérification et de test. Le chapitre 6 proposera d'ailleurs une première exploitation de ce modèle formel en s'intéressant à la recherche formelle de flux d'informations illégaux entre des contextes de sécurité.

Dans le chapitre 4, nous avons défini les spécifications de notre architecture. Nous en avons présenté les principes de fonctionnement, en particulier la présence de deux politiques spécifiques : la politique de protection, et la politique de modification.

Ce chapitre va donc formaliser le modèle issu des spécifications, en particulier préciser le rôle et la définition de la Méta-Politique, ainsi que spécifier les deux langages qui exploitent notre architecture de contrôle. Dans un premier temps, nous établirons une définition générale du modèle de Méta-Politique. Dans ce cadre, nous préciserons les formats du langage neutre de politique de protection, et le langage d'expression de contraintes de modification. Ce modèle général de Méta-Politique peut s'appliquer à différents usages tels que le contrôle d'accès, la détection d'intrusion, l'authentification.

Dans un second temps, cette définition sera appliquée à la formalisation d'une méta-politique de contrôle d'accès, et plus particulièrement nous verrons comment les différents aspects du contrôle d'accès (contextes de sécurité, vecteurs d'interaction) s'intègrent dans les deux langages définis en première section. Nous étudierons en détail un exemple d'application à l'administration du contrôle d'accès.

Enfin, nous avons envisagé d'autres applications pour notre modèle de méta-politique. Nous décrirons brièvement des travaux auxquels nous avons participé concernant la coopération entre notre méta-politique et la détection d'intrusions. Les aspects détection d'intrusions ne concernent pas directement cette thèse. Nous précisons ici seulement les idées générales de cette coopération.

La première section définit notre modèle général de Méta-Politique. Ensuite la section 5.2 détaillera plus précisément l'application de ce modèle général au contrôle d'accès, et enfin la section 5.3 précisera quelques aspects sur la coopération entre contrôle d'accès et détection d'intrusions au sein de l'architecture de méta-politique.

La formalisation de notre architecture a également été exposée dans les articles [Blanc *et al.* 2004b, Blanc *et al.* 2004a, Blanc *et al.* 2005a, Blanc *et al.* 2005b].

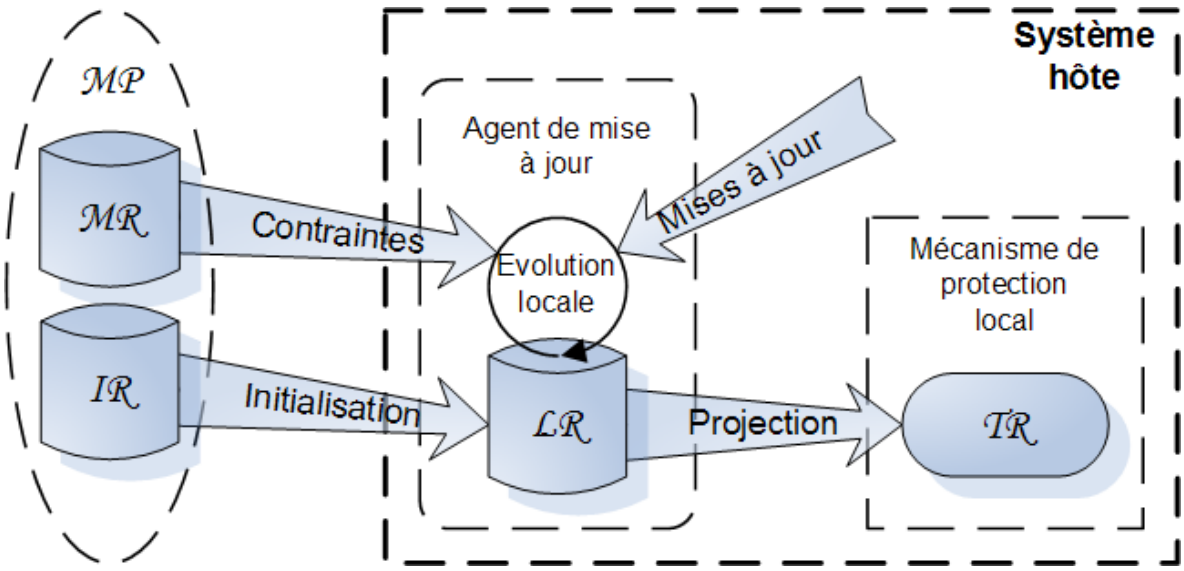

FIG. 5.1 – Evolution locale des politiques de sécurité

5.1 Définition générale du modèle de Méta-Politique

Dans cette section, on considère un modèle général de méta-politique, qui peut tout aussi bien être appliqué au contrôle d'accès qu'à la détection d'intrusion. Toutefois les exemples développés concerneront uniquement le contrôle d'accès, qui est au coeur du sujet de la thèse.

La définition générale du modèle de Méta-Politique comprend la description des différents niveaux de politiques introduits en partie 4.1.1. Comme indiqué dans la figure 5.1, trois niveaux sont gérés :

1. La Méta-Politique elle-même, $\mathcal{MP}$, inclut deux ensembles de règles : l'ensemble $\mathcal{MR}$ des règles de modification (*Modification Rules*), et l'ensemble $\mathcal{IR}$ des règles initiales (*Initial Rules*). Ces deux ensembles de règles utilisent deux langages neutres différents ;
2. La politique locale est représentée par l'ensemble de règles $\mathcal{LR}$ (*Local Rules*), en langage neutre (même langage que $\mathcal{IR}$). La politique $\mathcal{MR}$ contient les contraintes régissant la mise à jour de $\mathcal{LR}$, et $\mathcal{IR}$ donne le contenu initial de $\mathcal{LR}$ ($\mathcal{IR}$ est copiée pour initialiser $\mathcal{LR}$) ;
3. Enfin, l'ensemble $\mathcal{TR}$ contient les règles $\mathcal{LR}$ projetées vers le mécanisme présent sur le nœud auquel la Méta-Politique est appliquée, (par exemple SELINUX ou grsecurity dans le cas du contrôle d'accès obligatoire).

Nous allons maintenant étudier les différents composants du modèle général de méta-politique. En particulier, les **contextes de sécurité** concernent l'identification des entités des systèmes d'exploitation, les **vecteurs d'interaction** décrivent les droits d'accès entre contextes. Ensuite, les **règles de protection** sont écrites dans un langage neutre, qui sera explicité, ainsi que le langage décrivant les **contraintes de modification** des politiques.

5.1.1 Contextes de Sécurité

Les entités du système d'exploitation sont séparées en deux ensembles (cf. 2), celui des **sujets**, qui sont les entités actives (processus), et celui des **objets**, qui sont les entités passives (ressources,

fichiers, sockets). Les opérations effectuées par les sujets sur les objets sont représentées par des triplets *(sujet, objet, type d'accès)*. Le mécanisme de contrôle d'accès est alors la fonction qui associe à un tel triplet un booléen, dont la valeur signifie que l'accès est accordé ou refusé.

Les mécanismes de MAC utilisent des labels (cf. partie 2.1.2.1) pour caractériser ces sujets et objets. Typiquement, chaque entité du système est associée à un tel label, que nous appellerons **contexte de sécurité** ou *security context* (sc). Chaque sujet et objet du système d'exploitation est associé à un contexte de sécurité, comme expliqué en partie 3.5. Les sujets sont représentés par l'ensemble SC_S des contextes de sécurité des sujets, et les objets par l'ensemble SC_O des contextes de sécurité objet. L'équation 5.1 défini SC comme l'ensemble de tous les contextes.

$$SC = SC_S \cup SC_O \tag{5.1}$$

Les informations contenues dans le contexte de sécurité dépendent du modèle de protection associé. Par exemple, dans le cas du contrôle d'accès, les contextes contiennent des identifiants dont le but est de caractériser l'usage du processus ou de la ressource associé. Ainsi le processus du serveur Web Apache est associé au contexte de sécurité `apache_t`. La composition exacte des contextes de sécurité pour le contrôle d'accès sera détaillée en section 5.2.

5.1.2 Vecteurs d'Interaction

Avant de formaliser la notion de vecteur d'interaction, on définit IS l'ensemble des opérations élémentaires possibles (*Interaction Set*) entre sujets et objets, au niveau du système d'exploitation. $is \subset IS$ représente un ensemble d'opérations système d'un sujet vers un objet, donc un sous-ensemble de IS, comme l'indique l'équation 5.2.

$$\forall is \subset IS, is = \{eo_1, eo_2, \ldots, eo_n\} \tag{5.2}$$

Un exemple d'accès est l'appel système de lecture d'un fichier : $eo_1 = file : read$. Un sous-ensemble de droits d'accès peut être, par exemple, $is = \{file : read, write\}$.

Par abus de notation, on inclut dans l'ensemble IS une opération qui ne fait pas partie du système d'exploitation, qui est l'opération de changement de contexte de sécurité pour un sujet. Cette opération est représentée par $\{process : transition\}$.

IV (*Interaction Vector*) est l'ensemble de tous les vecteurs d'interaction. Un vecteur d'interaction iv est défini par l'équation 5.3, et représente un triplet composé d'un contexte de sécurité sujet sc_s, d'un contexte de sécurité objet sc_o et d'un ensemble de permissions is.

$$\forall iv \in IV, iv = (sc_s \in SC_S, sc_o \in SC_O, is \subset IS) \tag{5.3}$$

Par exemple, pour l'application au contrôle d'accès, on obtient des vecteurs d'interaction comme `(apache_t, var_www_t, {file : read})`. Celui-ci indique que le serveur Web Apache a l'autorisation d'accéder en lecture au dossier contenant les pages Web.

5.1.3 Langage Neutre de Politique

La partie suivante définit le langage neutre. Dans un premier temps, nous rappelons ce qui motive la définition d'un tel langage, puis nous définissons le langage lui-même.

5.1.3.1 Motivation

Comme expliqué dans la partie 4.1.4, notre architecture supporte différents mécanismes de contrôle d'accès. Nous avons donc défini une couche abstraite intermédiaire entre la méta-politique et le mécanisme de contrôle d'accès local des nœuds du système réparti : le langage neutre d'expression de règles de protection.

Grâce à ce langage neutre intermédiaire, la méta-politique ne dépend pas des mécanismes de contrôle d'accès sur lesquels elle agit. Elle ne dépend que du langage neutre. C'est ce langage qui est utilisé pour représenter la politique initiale $\mathcal{IR}$ et la politique locale de protection du nœud $\mathcal{LR}$, qui évolue sous le contrôle de la politique de modification. De plus, lorsque $\mathcal{LR}$ est modifiée, un agent de traduction configure le mécanisme de contrôle d'accès sous-jacent (par exemple, SELINUX ou grsecurity).

On distingue la modification de $\mathcal{LR}$, de la projection sur un système cible donné. La politique $\mathcal{LR}$ représente les interactions nécessaires au fonctionnement des applications installées sur le noeud, tandis que $\mathcal{TR}$ représente la capacité du système cible à contrôler ces interactions. $\mathcal{TR}$ reflète soit toutes les interactions de $\mathcal{LR}$, soit seulement un sous-ensemble, suivant les capacités du système cible. On voit donc tout l'intérêt du langage neutre, puisque la politique $\mathcal{LR}$ représente toutes les interactions nécessaires, indépendamment des capacités du système cible à les contrôler.

5.1.3.2 Définition du langage neutre de politique

La politique neutre $\mathcal{LR}$ contient un ensemble de règles, chacune décrivant un ensemble de droits d'accès is entre deux contextes de sécurité SC_S, SC_O. $\mathcal{LR}$ est donc un ensemble de vecteurs d'interaction. C'est donc un sous-ensemble de IV, comme l'expriment les équations suivantes :

$$\forall r \in \mathcal{LR}, \exists iv \in IV, r = iv \tag{5.4}$$

$$\mathcal{LR} \subset IV \tag{5.5}$$

Le listing 5.1 présente un exemple simplifié de politique neutre de contrôle d'accès concernant le serveur Web Apache. Il donne les autorisations à Apache d'exécuter des sous-processus (exécution simultanée de plusieurs copies du processus Apache), d'exécuter un processus de service (`service_u_t`), qui lui-même a l'autorisation d'accéder à des données appartenant à l'utilisateur X (`user_X_info_t`).

```
(apache_t, apache_exec_t, {file : read, execute})
(apache_t, service_u_t, {process : transition})
(service_u_t, service_u_exec_t, { file : read, execute})
(service_u_t, user_X_info_t, {file : read})
```

Listing 5.1 – Exemple de politique neutre.

5.1.4 Langage Neutre de Contraintes

Après avoir défini le langage neutre, nous disposons d'une base pour concevoir le langage qui va contraindre les modifications possibles de la politique locale. Les règles de modification seront contenues dans la politique de modification $\mathcal{MR}$, qui définira des *contraintes* d'évolution sur $\mathcal{LR}$. Il s'agit donc de définir comment les contextes de sécurité et vecteurs d'interaction peuvent être ajoutés, modifiés ou supprimés dans $\mathcal{LR}$.

Dans un premier temps, nous recenserons les opérations souhaitées pour l'évolution de la politique, et ensuite nous passerons à la définition du langage de contraintes.

5.1.4.1 Analyse des types de modification souhaités pour la politique de protection

Le modèle de modification de la politique doit prendre en compte les *opérations d'administration* couramment effectuées par les administrateurs de systèmes :

- Ajout et suppression d'utilisateurs : dans les entreprises, au fur et à mesure des embauches et des départs d'employés, la base des comptes d'utilisateurs doit être mise à jour pour refléter en temps réel l'ensemble des utilisateurs effectifs des systèmes d'exploitation ;
- Installation et retrait d'applications : dans le cadre de leurs activités, les employés ont besoin d'utiliser de nouvelles applications, ou des nouvelles versions d'applications déjà installée. L'administrateur doit donc assurer que les applications requises sont disponibles, et au besoin en installer de nouvelles. Il doit également supprimer les applications qui ne sont plus utilisés, et les anciennes versions de logiciels qui sont devenues obsolètes ;
- Gestion des permissions d'accès : l'administrateur doit assurer, à tout moment, que seuls les utilisateurs ayant le besoin d'accéder à une ressource, ou le besoin d'exécuter une application, ont effectivement les droits correspondants. Ceci implique que chaque utilisateur possède les permissions d'accès nécessaires et suffisantes à son activité. En particulier, lorsque de nouvelles applications sont déployées, il faut autoriser les utilisateurs qui en ont besoin à les utiliser.

Compte tenu de ces besoins, l'architecture de méta-politique doit gérer :

- L'ajout et la suppression de contextes de sécurité, sujets ou objets ;
- L'ajout, la modification et la suppression des droits d'accès accordés aux sujets.

Nous allons donc proposer, pour la politique de modification, des lois qui concernent, d'une part, les contextes de sécurité, et d'autre part, les interactions.

Concernant l'ajout et la suppression de contextes de sécurité, les besoins sont :

- Le contrôle de la création de nouveaux contextes. Lorsque de *nouveaux comptes d'utilisateurs* sont créés sur le système d'exploitation, ces utilisateurs doivent être pris en compte par le mécanisme de contrôle d'accès. Ceci implique d'associer des contextes de sécurité aux programmes lancés par ces utilisateurs, car les contextes sont les seules informations que le mécanisme de contrôle d'accès peut manipuler.
  La prise en compte des nouveaux utilisateurs peut également déboucher sur la création de *nouveaux rôles*. En effet, un utilisateur avec une activité spécifique (développeur, administrateur. . .) est associé à un rôle du même nom (cf. le modèle RBAC en partie 2.1.3).
  De même, l'installation de *nouvelles applications* implique la création de contextes objets. Par exemple, lors de l'installation du serveur Web Apache, le contexte `apache_t` est créé pour le processus `apache`, `apache_exec_t` pour le fichier exécutable, `apache_config_t` pour les fichiers de configuration. . . ;
- L'effacement des contextes existants : par exemple, lorsqu'un employé quitte une entreprise, ou qu'un partenaire sur l'exploitation d'un cluster de calcul décide d'arrêter son partenariat, il faut purger la configuration du contrôle d'accès de tous les contextes de sécurité associés à ces utilisateurs, puisqu'ils sont devenus inutiles.

Pour l'ajout, la modification et la suppression des droits d'accès, nous avons :

- L'ajout de nouveaux vecteurs d'interaction : lorsqu'une nouvelle application est déployée, il est nécessaire de fournir des droits d'accès pour cette application, afin qu'elle puisse utiliser les ressources du système d'exploitation. Sur l'exemple de la figure 4.3 du chapitre précédent qui

illustre l'installation d'une nouvelle application, la politique locale $\mathcal{LR}$ est modifiée pour que la nouvelle application puisse accéder à Internet.
En outre, il faut autoriser les utilisateurs à exécuter les nouvelles applications, i.e. ajouter des vecteurs d'interaction ayant pour sujet les contextes associés aux utilisateurs, et pour objet les contextes associés aux applications ;

– la modification et la suppression des vecteurs existants : lors de la suppression d'applications, il est nécessaire de supprimer de la politique $\mathcal{LR}$ les vecteurs d'interaction correspondant, afin que la politique reste cohérente avec l'état du système.
Également, lors de la détection d'attaques provenant d'un utilisateur ou d'une application présents sur le nœud, un mécanisme de contre-mesure peut décider de supprimer les vecteurs d'interaction incriminés. Par exemple, si un utilisateur installe involontairement un cheval de Troie, le mécanisme de détection va reconnaître qu'il s'agit d'un programme malicieux, et supprimer le vecteur d'interaction qui autorise l'utilisateur à exécuter ce programme.

5.1.4.2 Définition du langage d'expression des contraintes de modification

Objectifs Le but de ce langage est de *définir explicitement les modifications autorisées de la politique de sécurité*. Chacune des règles écrites dans ce langage est donc composée, d'une part, d'un *contexte de sécurité* autorisé à effectuer la modification, et d'autre part, du contexte de sécurité ou du vecteur d'interaction pouvant être modifié.

Principe Pour conserver la généricité[1] dans la Méta-Politique, le langage des contraintes ne spécifie pas directement des contextes ou vecteurs, mais des *expressions régulières*. Les expressions régulières autorisent la définition de masques (ou *pattern*). Ainsi, le langage de contrainte fournit une grande souplesse dans l'administration des contextes de sécurité et des permissions d'accès.

Concernant l'administration des contextes, l'usage d'expressions régulières autorise la désignation de familles de contexte. Par exemple, une règle de création des contextes `apache_.*` autorise l'ajout d'un ensemble de contextes pour le serveur Web Apache, comme par exemple `apache_t`, `apache_exec_t` et `apache_config_t`. Ainsi, il n'est pas nécessaire de connaître tous les contextes à l'avance, et on conserve une certaine liberté sur la stratégie de nommage des contextes, mais en même temps il est possible de contrôler que seuls des contextes concernant Apache seront ajoutés.

Concernant l'administration des permissions d'accès, on doit retrouver une correspondance avec les règles concernant les contextes. En effet, si une règle autorise l'ajout de contextes du type `apache_.*`, il est nécessaire de pouvoir écrire des règles de contraintes sur les permissions d'accès avec le même style de désignation des contextes. Sinon, il ne serait pas possible de créer des vecteurs d'interaction entre les nouveaux contextes. Par exemple, concernant le serveur Apache, l'administrateur doit avoir l'autorisation de créer des vecteurs d'interaction entre `apache_t` et `apache_config_t`. Mais comme ces noms de contextes ne sont pas connus à l'avance, la règle autorisant l'ajout de ces vecteurs portera sur les contextes sujet et objet correspondant à l'expression régulière `apache_.*`.

Le langage se compose de deux formes de règles, une pour les contextes de sécurité, et une pour les vecteurs d'interaction. Les règles concernant l'administration des contextes forment l'ensemble $\mathcal{MR}_{sc}$, et les règles concernant l'administration des vecteurs, l'ensemble $\mathcal{MR}_{iv}$. Ces deux ensembles

[1] La généricité correspond aux propriétés de support de systèmes hétérogènes et de réutilisation de modèles de protection définies dans le chapitre 4.

forment la politique de modification $\mathcal{MR}$, comme l'exprime l'équation 5.6.

$$\mathcal{MR} = \mathcal{MR}_{sc} \cup \mathcal{MR}_{iv} \tag{5.6}$$

Les modifications peuvent être des ajouts, suppressions ou modifications de vecteurs d'interaction existants. L'équation 5.7 définit l'ensemble A des actions de modification.

$$A = \{Add, Mod, Del\} \tag{5.7}$$

La présence d'expressions régulières dans nos règles nous conduit à introduire des fonctions générant un ensemble d'éléments à partir d'une expression régulière. Ainsi, la fonction g_{SC} associe à l'expression régulière exp_{sc} un ensemble de contextes de sécurité correspondants, la fonction g_{IS} associe à l'expression régulière exp_{is} un ensemble de parties de IS, qui contient les opérations élémentaires possibles. Enfin la fonction g_{IV} associe un ensemble de vecteurs d'interaction à un triplet d'expressions régulières exp_{iv}, suivant l'équation 5.8. La fonction effectue en fait le produit cartésien des ensembles calculés à partir des expressions régulières exp_{sc_s}, exp_{sc_o} et exp_{is}.

$$\begin{aligned} exp_{iv} &= (exp_{sc_s}, exp_{sc_o}, exp_{is}) \\ g_{IV}(exp_{iv}) &= g_{SC}(exp_{sc_s}) \times g_{SC}(exp_{sc_o}) \times g_{IS}(exp_{is}) \end{aligned} \tag{5.8}$$

avec

$$\begin{aligned} g_{SC}(exp_{SC}) &\subset SC \\ g_{SC}(exp_{sc_s}) &\subset SC_S \\ g_{SC}(exp_{sc_o}) &\subset SC_O \\ g_{IS}(exp_{is}) &\subset P(IS) \end{aligned}$$

Chaque règle des deux ensembles $\mathcal{MR}_{sc}$ et $\mathcal{MR}_{iv}$ est un triplet, premièrement, un contexte de sécurité $sc_{requester}$ autorisé à effectuer la modification, deuxièmement un mot-clé de l'ensemble A des actions de modification, et troisièmement la désignation des éléments sur lesquels porte la modification, autrement dit la *portée* de la contrainte de modification dans la politique locale. Les règles r_{sc} concernant l'administration des contextes sont définies par l'équation 5.9, leur troisième élément est un ensemble de contextes de sécurité généré par l'expression régulière exp_{sc}. Les règles r_{iv} concernant l'administration des vecteurs d'interaction sont définies par l'équation 5.10, leur troisième élément est un ensemble de vecteurs d'interaction généré par $g_{IV}(exp_{iv})$ (cf. équation 5.8).

$$\forall r_{sc} \in \mathcal{MR}_{sc}, r_{sc} = (sc_{requester} \in SC_S, a \in A, g_{SC}(exp_{sc}) \subset SC) \tag{5.9}$$

$$\forall r_{iv} \in \mathcal{MR}_{iv}, r_{iv} = (sc_{requester} \in SC_S, a \in A, g_{IV}(exp_{iv}) \subset IV) \tag{5.10}$$

Syntaxe Nous allons maintenant définir le langage exprimant les contraintes sur les modifications de la politique. Les règles définies par ce langage comprennent toujours les éléments suivants : le mot-clé $enable$, pour différencier ces règles du *allow* habituellement utilisé pour les règles de politique de sécurité ; puis une partie indiquant le type de modification autorisé (ajout, modification, suppression) ; ensuite un suffixe indiquant s'il s'agit d'une règle sur les contextes ou les vecteurs. Enfin, on a le contexte de sécurité autorisé à demander la modification, et la portée de la règle de contrainte (i.e. une expression régulière pour les contextes, ou un triplet d'expressions pour les vecteurs d'interaction).

La première forme, donnée par l'équation 5.11, définit les règles d'administration des contextes de sécurité, correspondant au règles r_{sc} de l'équation 5.9. Le mot-clé initial se décline en $enableAddSC$,

$enableModSC$ et $enableDelSC$, SC indiquant qu'il s'agit d'une règle sur les contextes de sécurité. Le premier paramètre, $sc_{requester}$, désigne le contexte de sécurité autorisé à effectuer la demande de modification. $pattern_{SC}$ est une expression régulière définissant de façon générique les contextes de sécurité sur lesquels cette contrainte opère.

Le paramètre $pattern_{attribut}$ est un paramètre *optionnel* dont le but est de préciser les conditions d'application de ce contexte par rapport au mécanisme de protection visé. Par exemple, dans le cas du contrôle d'accès, il sert à donner la correspondance du contexte avec les fichiers du système d'exploitation. Dans la règle `enableAddSC(`sc_{admin}`, apache_.*, /usr/.*/apache2(/.*)?)`, le paramètre `/usr/.*/apache2(/.*)?` indique les chemins valables dans le système de fichiers auxquels les contextes de la forme `apache_.*` peuvent être associés, e.g. le chemin de l'exécutable d'Apache `/usr/sbin/apache2` et le module SSL pour Apache `/usr/lib/apache2/modules/mod_ssl.so` (qui implante les fonctions de chiffrement des communications avec Apache).

$$enable***SC(sc_{requester}, pattern_{SC}) \tag{5.11}$$

$$\text{ou} \tag{5.12}$$

$$enable***SC(sc_{requester}, pattern_{SC}, pattern_{attribut}) \tag{5.13}$$

$$\text{avec } ***:= Add|Mod|Del$$

La seconde forme, donnée par l'équation 5.14, définit les règles d'administration des vecteurs d'interaction, correspondant au règles r_{iv} de l'équation 5.10. Cette forme se décline en $enableAddIV$, $enableModIV$ et $enableDelIV$. Comme précédemment, on a le premier paramètre $sc_{requester}$. Le second paramètre comprend trois expressions régulières, pour désigner un ensemble de vecteurs d'interaction possibles.

$$enable***IV(sc_{requester}, (pattern_S, pattern_O, pattern_{IS})) \tag{5.14}$$

$$\text{avec } ***:= Add|Mod|Del$$

Le listing 5.2 présente un exemple de politique $\mathcal{MR}$ appliquée au contrôle d'accès. Il s'agit là d'autoriser l'administrateur à créer les contextes de l'application Apache, et à lui donner les droits d'accès en lecture sur les pages Web.

```
enableAddSC(sc_admin, apache_.*, /usr/.*/apache2(/.*)?)
enableAddIV(sc_admin, (apache_.*, var_www_.*, {file : read}))
```

Listing 5.2 – Exemple de règles de modification.

5.1.5 Projection du langage neutre de politique

L'architecture de Méta-Politique supporte l'application d'une politique neutre sur des systèmes hétérogènes. Dans ce but, la politique de protection $\mathcal{LR}$ est exprimée dans un langage neutre, indépendant du mécanisme de sécurité cible. Il est donc nécessaire de disposer d'un mécanisme de projection des règles de $\mathcal{LR}$ vers le langage de configuration du système cible, créant un ensemble de règles $\mathcal{TR}$ (*Target Rules*).

Cette projection de $\mathcal{LR}$ vers $\mathcal{TR}$ est accomplie par un module de projection, aussi appelé *Language Adapter*. Il est dépendant du mécanisme de sécurité cible, il est donc nécessaire de disposer d'un module de projection pour chaque mécanisme devant être configuré par l'architecture de Méta-Politique.

	Méta-Politique $\mathcal{MP}$	Règles locales $\mathcal{LR}$	Règles projetées $\mathcal{TR}$
Access Control	$\mathcal{IR}_{\mathcal{AC}}$ $\mathcal{MR}_{\mathcal{AC}}$	$\mathcal{LR}_{\mathcal{AC}}$	$\mathcal{TR}_{\mathcal{AC}}$

TAB. 5.1 – Récapitulatif : Méta-Politique et Politiques

5.2 Définition de la Méta-Politique de contrôle d'accès

La section précédente a présenté notre modèle générique de Méta-Politique. Un langage neutre de politique, et un langage d'expression de contraintes de modification ont été définis. Bien que des exemples aient été donné pour le contrôle d'accès dans la section précédente, le modèle est général et peut s'appliquer à d'autres mécanismes de protection comme la détection d'intrusions. Dans cette partie, nous allons appliquer le modèle général à la problématique du contrôle d'accès.

Nous allons donc définir une Méta-Politique spécifique au contrôle d'accès, notée $\mathcal{MP}_{\mathcal{AC}}$. C'est un sous-ensemble des règles de la politique $\mathcal{MP}$ globale. La définition générale des règles donnée dans la section 5.1, pour les politiques $\mathcal{IR}$, $\mathcal{MR}$ et $\mathcal{LR}$, prend une nouvelle signification dans le cadre de l'application au contrôle d'accès. Le tableau 5.1 présente la correspondance entre la définition générale, et la définition spécifique au contrôle d'accès.

- $\mathcal{MP}_{\mathcal{AC}}$ contient deux ensembles de règles :
  - $\mathcal{IR}_{\mathcal{AC}}$ définit la configuration initiale de la politique de contrôle d'accès $\mathcal{LR}_{\mathcal{AC}}$;
  - $\mathcal{MR}_{\mathcal{AC}}$ contient les contraintes sur la modification des lois de contrôle d'accès contenues dans $\mathcal{LR}_{\mathcal{AC}}$.
- $\mathcal{LR}_{\mathcal{AC}}$ contient la politique de contrôle d'accès locale ;
- $\mathcal{TR}_{\mathcal{AC}}$ contient les règles projetées pour le mécanisme de contrôle d'accès local.

Cette section sera organisée comme suit : d'abord, nous analyserons le modèle de contrôle d'accès existant dans les systèmes d'exploitation actuels. Ensuite, nous redéfinirons le langage neutre de politique et le langage de contraintes pour le contrôle d'accès. Enfin, nous décrirons le procédé de projection du langage neutre de politique vers les implantations systèmes existantes.

5.2.1 Modèle de contrôle d'accès

Pour commencer, nous allons étudier l'apport de la méta-politique par rapport au modèle classique du contrôle d'accès, défini par [Anderson 1972].

Ensuite, nous analyserons les modèles de configuration de SELINUX et grsecurity, suivant deux points distincts :

- L'analyse des contextes de sécurité utilisés ;
- L'écriture des règles d'accès.

5.2.1.1 Modèle d'Anderson et Méta-Politique

L'article d'Anderson définit le concept de *Moniteur de Référence*. Celui-ci est en charge de valider la légitimité des références faites par un programme à d'autres programmes, à des données ou à des périphériques d'entrée/sortie, par rapport à une liste de types de référence autorisées pour ce programme. Généralement, cette liste est présente sous la forme d'une matrice d'accès.

D'un point de vue contrôle d'accès, on parle de Moniteur de Référence de la Sécurité. c'est une fonction qui prend en entrée un triplet (sujet, objet, type d'accès) et renvoie un booléen (accès refusé

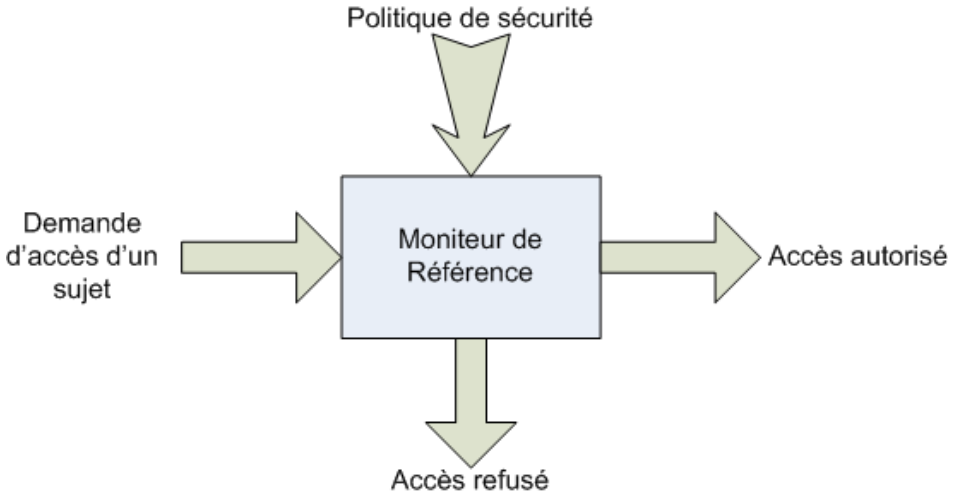

FIG. 5.2 – Moniteur de Référence.

ou accès accordé). La configuration du contrôle d'accès est faire par une politique de sécurité (cf. figure 5.2).

Avec l'introduction de la méta-politique, l'évolution de la politique de sécurité est contrôlée. En plus des demandes d'accès des sujets, il faut également traiter les demandes de modification de la politique, comme cela a été explicité en 4.1.2. La figure 5.3 expose ces deux niveaux de contrôle :

1. Le premier niveau est modélisé par le Moniteur de Référence. Il gère le contrôle d'accès, il est configuré par la politique de sécurité du système. Dans notre modèle, cela correspond à la politique $\mathcal{LR}_{\mathcal{AC}}$.
2. Le second niveau est modélisé par la Méta-Politique. Il contrôle l'évolution de la politique de contrôle d'accès, et il est configuré par la politique de modification. Dans notre modèle, cela correspond à la politique $\mathcal{MR}_{\mathcal{AC}}$.

5.2.1.2 SELinux

Description SELINUX a été présenté dans la partie 2.2.2. Il s'agit d'une implantation de contrôle d'accès obligatoire pour Linux, qui intègre notamment les modèles RBAC et DTE. Les contextes de sécurité comportent un identifiant d'utilisateur SELINUX, un rôle et un type (par exemple `root:admin_r:admin_t` est le contexte de sécurité d'un processus appartenant à *root*).

Syntaxe de configuration Le listing 5.3 procure un exemple de configuration de SELINUX. On trouve des règles d'accès sur les lignes 5 à 9 (règles commençant par `allow`). Elles décrivent des interactions autorisées entre un type sujet et un type objet. Les interactions sont composées d'opération élémentaire, par exemple `file{ read, write }`. La liste de ces opération élémentaire correspond aux appels système disponibles dans Linux.

Les lignes 1 à 3 présentent des déclaration de types (règles commençant par `type`). Les types sujets se distinguent par la présence d'un attribut `domain`. En effet, SELINUX n'implante qu'une version allégée de DTE (cf. 2.1.2.5), mais *émule* le modèle DTE complet avec les attributs.

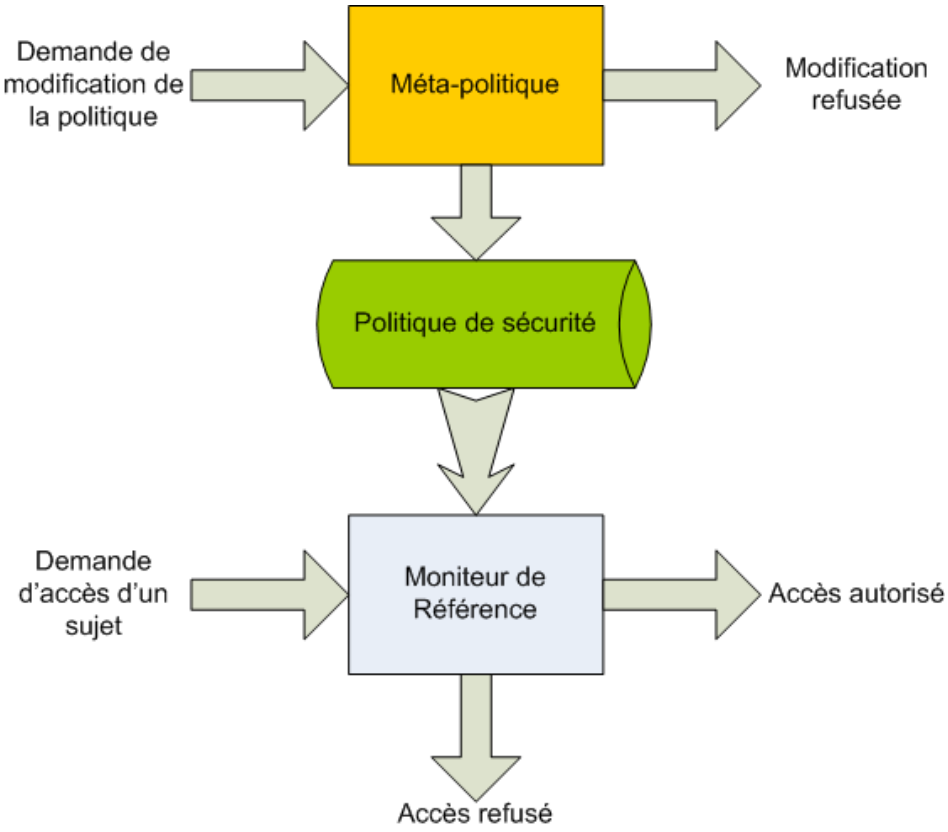

FIG. 5.3 – Moniteur de Référence et Méta-politique

Enfin les lignes 11 et 12 présentent des règles de transition (règles commençant par `type_transition`). Il s'agit de règles décrivant des évènements de changement de contexte de sécurité. Par exemple, la ligne 11 dit que, lorsqu'un processus ayant le type `httpd_t` exécute un fichier de type `httpd_php_exec_t`, il prend alors le contexte `httpd_php_t`.

Modélisation L'équation 5.15 définit l'ensemble des contextes de sécurité de SELINUX. Il s'obtient en effectuant le produit cartésien de l'ensemble des utilisateurs, l'ensemble des rôles et l'ensemble des types.

$$SC^{selinux} = \{user\} \times \{role\} \times \{type\} \tag{5.15}$$

Comme on l'a vu dans le listing 5.3, les lois dans SELINUX) sont de deux types. D'une part, il existe des règles d'accès (mot-clé `allow`, dont le but est d'autoriser un ensemble de types d'accès entre un contexte sujet et un contexte objet (équation 5.16). D'autre part, il existe des règles de transition, qui sont utilisées lors de la création de nouveaux sujets (exécution d'un nouveau processus) et de nouveaux objets (par exemple, création d'un nouveau fichier ou dossier). Ces règles ont également trois paramètres, mais ici il ne s'agit pas de droits d'accès. Il faut lire la règle de la façon suivante : lorsque le contexte sc_s crée un nouveau sujet/objet à partir du contexte sc_o, alors celui-ci se place

```
type httpd_t, domain, privlog, daemon, privmail;                                   1
type httpd_exec_t, file_type, exec_type;                                           2
type service_u_exec_t, file_type, exec_type;                                       3
 [...]                                                                             4
allow httpd_t httpd_exec_t:file { read getattr lock execute ioctl entrypoint };    5
allow httpd_t service_u_t:process transition;                                      6
allow service_u_t service_u_exec_t:file { read getattr lock execute ioctl entrypoint };  7
allow service_u_t userX_info_t:file{ read, write };                                8
allow userX_t service_u_t:process transition;                                      9
 [...]                                                                             10
type_transition httpd_t httpd_php_exec_t:process httpd_php_t;                      11
type_transition httpd_t service_u_exec_t:process service_u_t;                      12
 [...]                                                                             13
```

Listing 5.3 – Extrait de politique SELINUX.

dans le contexte sc_n (équation 5.17).

$$IV_{access}^{selinux} = (sc_s \in SC^{selinux}, sc_o \in SC^{selinux}, is \subset IS) \tag{5.16}$$

$$IV_{transition}^{selinux} = (sc_s \in SC_S, sc_o \in SC_O, sc_n \in SC) \tag{5.17}$$

5.2.1.3 grsecurity

Description grsecurity a été présenté dans la partie 2.2.3. Comme SELINUX, il s'agit d'une implantation système de contrôle d'accès obligatoire. grsecurity implante les modèles RBAC et DTE.

Syntaxe des règles Le listing 5.4 contient un exemple de configuration de grsecurity. Les lignes 1 à 3 définissent les règles d'accès associées au rôle `admin`, puis les lignes 5 à 22 définissent les règles associées au rôle `default` (il s'agit en fait du rôle par défaut, assigné à tout utilisateur pour lequel il n'existe pas de rôle à son nom). Dans chaque définition de rôle, on observe des lignes commençant par `subject`. Il s'agit de règles d'accès, qui ont pour contexte sujet le programme qui suit le mot-clé `subject`, et pour contexte objet chacune des lignes suivantes, jusqu'au prochain mot-clé `subject`. Par exemple, les lignes 17 et 18 définissent un accès en lecture et écriture (`rw`) entre le processus `/usr/X11R6/bin/XFree86` et le fichier `/dev/mem`. Les lignes commençant par `+` attribuent des *capabilities POSIX* au sujet correspondant, par exemple la ligne 22 attribue `CAP_SYS_RAWIO` au processus `/usr/X11R6/bin/XFree86` (il s'agit d'autoriser ce processus à effectuer les appels système `iopl()` et `ioperm()`).

Modélisation L'équation 5.18 définit l'ensemble des contextes dans grsecurity. Les rôles sont déclarés par la règle `role`. Les contextes de sécurité sujet (lignes commençant par `subject`) sont composés d'un rôle et d'un chemin (ou *path*) qui désigne un fichier binaire exécutable, ou un dossier et ses sous-dossier (par héritage). L'ensemble des contextes de sécurité dans grsecurity est donc le produit cartésien de l'ensemble des rôles par l'ensemble des chemins (en anglais *path*). Les contextes de sécurité objet sont simplement des dossiers ou des fichiers.

$$SC_S^{grsecurity} = \{role\} \times \{path\} \tag{5.18}$$

$$SC_O^{grsecurity} = \{path\} \tag{5.19}$$

```
role admin sA
subject / rvka
   / rwcdmlxi

role default G
role_transitions admin
subject /
   /     r
   /opt     rx
   /home    rwxcd
   /mnt     rw
   /dev
   /dev/grsec  h
   /dev/urandom   r
[...]

subject /usr/X11R6/bin/XFree86
   /dev/mem rw

   +CAP_SYS_ADMIN
   +CAP_SYS_TTY_CONFIG
   +CAP_SYS_RAWIO

[...]
```

Listing 5.4 – Extrait de configuration grsecurity.

Les règles d'accès sont de deux types. Certaines s'appliquent aux sujets, d'autres entre sujets et objets. Les accès accordés sont représentés par des identifiants d'une seule lettre, appelés *modes* (par exemple, r, w ou x). Il existe des modes qui s'appliquent seulement aux sujets, comme les capabilities POSIX ; ces modes forment l'ensemble $mode_S$. D'autres modes caractérisent des accès entre sujets et objets, ils forment l'ensemble $mode_O$. Pour comparer avec le modèle de [Lampson 1971], les règles sur les sujets sont similaires aux *capabilities*, tandis que celles sur les objets sont des ACL.

L'équation 5.20 définit l'ensemble des vecteurs d'interaction dans grsecurity, composé des deux types de règles exposés plus haut.

$$\begin{aligned} IV^{grsecurity} &= (sc_s \in SC_S^{grsecurity}, m \in mode_S) \\ &\cup (sc_s \in SC_S^{grsecurity}, sc_o \in SC_O^{grsecurity}, m \in mode_O) \end{aligned} \tag{5.20}$$

5.2.2 Langage neutre de contrôle d'accès

A partir de l'analyse faite sur les langages de configuration de SELINUX et grsecurity, nous allons maintenant définir le langage neutre de contrôle d'accès. Il sera utilisé pour l'écriture des règles contenues dans la politique $\mathcal{LR}_{\mathcal{AC}}$.

Bien que ce langage neutre soit défini pour le MAC, il peut aussi être projeté sur des systèmes à base de DAC (par exemple, Microsoft Windows). Bien évidemment, cette projection s'effectue au prix de la perte des propriétés du MAC, et de la faiblesse du DAC (cf. partie 3.1.1).

Nous définirons d'abord le format des contextes de sécurité, puis des vecteurs d'interaction. Ensuite nous étudierons la projection de ces vecteurs d'interaction vers SELINUX et grsecurity, qui sont les deux mécanismes actuellement supportés par l'architecture.

5.2.2.1 Contextes de sécurité

Concernant les contextes de sécurité, le format de SELINUX semble avoir le plus grand pouvoir d'expressivité comparé à grsecurity. En effet, les types de SELINUX sont assignés aux objets du système d'exploitation, en particulier les fichiers. grsecurity ne peux désigner que les fichiers dans ses règles d'accès. Il apparaît donc que l'ensemble des contextes de grsecurity est un sous-ensemble de celui de SELINUX.

Dans le cadre de l'application au contrôle d'accès de notre langage neutre, nous avons donc repris cette approche. Les contextes de sécurité, définis en partie 5.1, se composent donc maintenant d'un utilisateur, d'un rôle et d'un type, à la façon de SELINUX. L'équation 5.21 définit formellement ces contextes.

$$\forall sc \in SC, sc = (user, role, type) \tag{5.21}$$

L'équation 5.22 définit l'ensemble $SC_{\mathcal{AC}}$ des contextes de sécurité.

$$SC_{\mathcal{AC}} = \{user\} \times \{role\} \times \{type\} \tag{5.22}$$

Par exemple, le contexte de sécurité objet $sc_{O1} =$ `user_u, object_r, user_X_info_t` est associé à un fichier `/home/user_X/perso.info` contenant les informations personnelles de cet utilisateur. Les 3 contextes suivants représentent le processus Apache, un service d'Apache et un processus d'un utilisateur :

$sc_{S1} =$ `(system_u, system_r, apache_httpd_t)`
$sc_{S2} =$ `(system_u, system_r, service_u_t)`
$sc_{S3} =$ `(user_u, user_r, user_X_t)`

5.2.2.2 Vecteurs d'Interaction

Lorsque la Méta-Politique est appliquée au contrôle d'accès, l'ensemble $IS_{\mathcal{AC}}$ des interactions possibles entre sujets et objets, définis en section 5.1, contient les différents appels système existant sur les systèmes d'exploitation. En pratique, un appel système est associé à une certaine classe de ressource, par exemple l'appel système de lecture (*read*) peut être appliqué à des fichiers, à des sockets réseau... C'est pourquoi, dans le langage neutre de contrôle d'accès, la désignation des interactions comprend toujours deux types d'identifiants : une classe de ressource, et un ensemble d'opérations associées.

Par exemple, $is_1 =$ `{file : read, write}` représente les opérations de lecture et écriture sur un fichier. Suivant la définition donnée par l'équation 5.2, cet ensemble is se décompose en deux opérations élémentaires :

$is_1 = (eo_1, eo_2)$
$eo_1 =$ `file : read`
$eo_2 =$ `file : write`

Comme indiqué par l'équation 5.3, un vecteur d'interaction est un triplet $iv = (sc_{sujet} \in SC_{\mathcal{AC}}, sc_{objet} \in SC_{\mathcal{AC}}, is \subset IS_{\mathcal{AC}})$. Dans le cas du contrôle d'accès, ce vecteur représente un ensemble de droits d'accès is accordé au sujet sc_{sujet} sur l'objet sc_{objet}. $IV_{\mathcal{AC}}$ représente l'ensemble des vecteurs d'interaction.

En reprenant les contextes exemple donnés plus haut, on définit quatre vecteurs :

```
((system_u, system_r, apache_httpd_t), (system_u, system_r, service_u_t), {process :
    transition})
((user_u, user_r, user_X_t), (system_u, system_r, service_u_t), {process : transition})
((system_u, system_r, service_u_t), (user_u, object_r, user_X_info_t), {file : read})
((user_u, user_r, user_X_t), (user_u, object_r, user_X_info_t), {file : read, write})
```

Listing 5.5 – Exemple de politique neutre.

$iv_1 = (sc_{S1}, sc_{S2},$ `{process : transition}`$)$
$iv_2 = (sc_{S3}, sc_{S2},$ `{process : transition}`$)$
$iv_3 = (sc_{S2}, sc_{O1},$ `{file : read}`$)$
$iv_4 = (sc_{S3}, sc_{O1},$ `{file : read, write}`$)$

5.2.2.3 Langage neutre de politique de contrôle d'accès

Comme cela a été défini dans la section 5.1, le langage neutre de politique se compose d'un ensemble de vecteurs d'interaction. On a donc toujours :

$$\mathcal{LR}_{\mathcal{AC}} \subset IV_{\mathcal{AC}} \tag{5.23}$$

En reprenant les exemples des deux parties précédentes, on définit une politique de contrôle d'accès :

$$\mathcal{LR}_{\mathcal{AC}} = \{iv_1, iv_2, iv_3, iv_4\} \tag{5.24}$$

La figure 5.4 illustre cette politique par un graphe ayant pour sommets les contextes de sécurité, et pour arrêtes les interactions autorisées. Le listing 5.5 donne le contenu de la politique de sécurité. Ces règles représentent :

1. Le droit pour Apache de transiter vers le contexte du service ;
2. Le droit pour un processus de l'utilisateur de transiter vers le contexte du service ;
3. Le droit pour le service de lire le fichier d'information personnelle de l'utilisateur.
4. Le droit pour l'utilisateur de lire et écrire son fichier d'information personnelle.

L'intérêt de ces règles est de définir que le serveur Web Apache, n'a pas directement accès aux données personnelles de l'utilisateur, mais seulement par un service de traitement dédié `service_u_t`. En effet, le serveur Apache est un service réseau, il est donc particulièrement exposé aux attaques. Ainsi, si une attaque est réussie contre le serveur Apache, les données personnelles de l'utilisateur ne seront pas divulguées, car Apache n'y a pas accès directement.

5.2.2.4 Langage neutre de contraintes sur le contrôle d'accès

La définition formelle du langage de contraintes a été donnée par l'équation 5.6. Pour l'appliquer au contrôle d'accès, les contextes de sécurité sont issus de l'ensemble $SC_{\mathcal{AC}}$.

Par exemple, la règle `enableAddSC(`sc_{admin}`, apache_.*, /usr/.*/httpd(/.*)?)` autorise un administrateur local (contexte de sécurité sc_{admin}) à créer les contextes nécessaire pour le fonctionnement du serveur Web Apache (notamment les contextes `apache_httpd_t`, `apache_php_t`). Le deuxième paramètre, optionnel, indique à quels fichiers ces nouveaux contextes peuvent être associés.

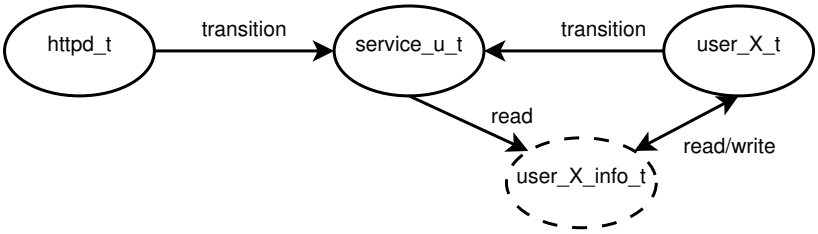

FIG. 5.4 – Graphe d'interaction de la politique.

Autre exemple, la règle `enableAddIV(`sc_{admin}`, (apache_.*, var_www_.*, {file :read}))` autorise l'administrateur local à ajouter les droits d'accès en lecture depuis les contextes d'exécution du serveur Web Apache (`apache_.*`) vers le dossier `/var/www` et les fichiers et sous-dossiers qu'il contient (normalement ce dossier contient les pages Web consultées par le biais du serveur Apache).

Ainsi, avec le jeu de règles du listing 5.6, l'administrateur peut installer l'application Apache et lui donner les droits d'accès nécessaire à son fonctionnement.

```
enableAddSC(sc_admin, apache_.*, /usr/.*/httpd(/.*)?)
enableAddIV(sc_admin, (apache_.*, var_www_.*, {file : read}))
```

Listing 5.6 – Exemple de règles de Méta-Politique.

5.2.3 Projection du langage neutre de contrôle d'accès

Pour être appliquée par l'architecture de Méta-Politique sur les nœuds où elle est déployée, la politique locale de contrôle d'accès $\mathcal{LR}_{\mathcal{AC}}$ doit être projetée vers le mécanisme de contrôle d'accès cible. Cela est effectué en projetant $\mathcal{LR}_{\mathcal{AC}}$ vers $\mathcal{TR}_{\mathcal{AC}}$, qui contient les règles de configuration pour le mécanisme local. Les mécanismes cible considérés dans cette thèse sont SELINUX et grsecurity.

C'est l'agent de mise à jour qui assure le lien entre la politique neutre $\mathcal{LR}_{\mathcal{AC}}$ et le mécanisme de contrôle d'accès local, via $\mathcal{TR}_{\mathcal{AC}}$. Il se compose donc :

- d'une partie fixe, capable d'interpréter la politique en langage neutre ;
- d'une partie modulaire, adaptée au mécanisme de sécurité local.

La politique est donc traduite en une configuration pour le mécanisme de contrôle d'accès local, puis cette configuration est transmise par l'agent via le moyen de communication spécifique au contrôle d'accès local.

5.2.4 Exemple d'application à l'administration du contrôle d'accès

L'architecture de Méta-Politique est déployée sur un système réparti. Il s'agit d'un ensemble de nœuds, pouvant présenter des systèmes d'exploitation différents, et donc des mécanismes de contrôle d'accès différents.

Sur chaque nœud, la politique locale $\mathcal{LR}_{\mathcal{AC}}$ est alors sous le contrôle de l'architecture. Comme on peut le voir sur la figure 5.1, elle est initialisée par $\mathcal{IR}_{\mathcal{AC}}$, et son évolution est contrainte par $\mathcal{MR}_{\mathcal{AC}}$, deux politiques contenues dans $\mathcal{MP}_{\mathcal{AC}}$.

A chaque fois qu'elle est modifiée, $\mathcal{LR}_{\mathcal{AC}}$ est projetée vers $\mathcal{TR}_{\mathcal{AC}}$ afin de configurer le mécanisme de contrôle d'accès local en conséquence.

Pour notre exemple d'administration, nous allons étudier le cas d'un administrateur souhaitant installer une nouvelle application sur un des nœuds où l'architecture de Méta-Politique est déployée.

5.2.4.1 Contenu de la Méta-Politique

Le listing 5.7 présente l'extrait de la politique de modification $\mathcal{MR}_{\mathcal{AC}}$ qui nous intéresse dans cet exemple (pour simplifier la lecture, seuls les types sont mentionnés dans la politique). La ligne 1 autorise l'administrateur (contexte `admin_t`) à créer des contextes pour des applications supplémentaires installées dans le dossier `/opt/apps`. La ligne 2 autorise l'administrateur à ajouter le droit d'exécuter cette nouvelle application à partir d'un shell (ligne de commande) d'utilisateur. La ligne 3 autorise l'administrateur à ajouter la permission de lire et modifier les fichiers de configuration des applications supplémentaires, depuis un shell d'administrateur. La ligne 4 autorise l'administrateur à ajouter des droits d'accès entre les nouveaux contextes définis pour les applications supplémentaires.

```
[...]                                                                                    1
enableAddSC(admin_t, opt_apps_.*, /opt/apps/.*)                                          2
enableAddIV(admin_t, (user_shell_t, opt_apps_.*, {file : execute}))                      3
enableAddIV(admin_t, (user_shell_t, opt_apps_.*, {process : transition}))                4
enableAddIV(admin_t, (admin_shell_t, opt_apps_.*_config_t, {file : read, write}))        5
enableAddIV(admin_t, (opt_apps_.*, opt_apps_.*, {file : .*}))                           6
[...]                                                                                    7
```

Listing 5.7 – Extrait de la Méta-Politique déployée.

5.2.4.2 Mise à jour de la politique

Lors de l'installation de la nouvelle application `lambda`, l'administrateur va produire une série de requêtes de mise à jour de la politique locale $\mathcal{LR}_{\mathcal{AC}}$. Celles-ci vont être transmises à l'agent de mise à jour, qui va les valider en conformité avec la politique de modification $\mathcal{MR}_{\mathcal{AC}}$ donnée précédemment.

Le listing 5.8 contient la liste des mises à jour demandées par l'administrateur lors de l'installation de l'application `lambda`.

Les règles de type `AddSC` (lignes 1 à 5) ajoutent les nouveaux contextes spécifiques à l'application : il s'agit du contexte global de l'application, `opt_apps_lambda_t`, puis des contextes pour, respectivement, les fichiers binaires exécutables, les fichiers de données de l'application, les fichiers de configuration et enfin les fichiers de journaux d'évènements.

Les règles de type `AddIV` (lignes 7 à 12) demandent l'ajout de nouvelles interactions. La première donne le droit à l'utilisateur d'exécuter l'application `lambda` à partir de son shell de connexion, la seconde autorise l'administrateur à modifier la configuration de l'application. Enfin, les autres lignes ajoutent les droits d'accès nécessaires au fonctionnement de l'application : transition des fichiers exécutables vers le contexte global de l'application, lecture et écriture des fichiers de données, ajout d'évènements dans les journaux et lecture seule pour les fichiers de configuration.

Étant donné que toutes ces requêtes sont valides suivant la politique de modification $\mathcal{MR}_{\mathcal{AC}}$, donnée par le listing 5.7, l'agent les intègre toutes dans la politique neutre de contrôle d'accès $\mathcal{LR}_{\mathcal{AC}}$. Le listing 5.9 donne l'extrait de la politique neutre résultant des requêtes de mise à jour. On y trouve les lignes produites par les commandes `AddIV`. La ligne 1 donne le droit à l'utilisateur d'exécuter

```
AddSC(opt_apps_lambda_t, /opt/apps/lambda/.*)                                          1
AddSC(opt_apps_lambda_exec_t, /opt/apps/lambda/bin/.*)                                 2
AddSC(opt_apps_lambda_data_t, /opt/apps/lambda/data/.*)                                3
AddSC(opt_apps_lambda_config_t, /opt/apps/lambda/etc/.*)                               4
AddSC(opt_apps_lambda_log_t, /opt/apps/lambda/log/.*)                                  5
                                                                                       6
AddIV(user_shell_t, opt_apps_lambda_exec_t, {file : execute})                          7
AddIV(user_shell_t, opt_apps_lambda_t, {process : transition})                         8
AddIV(admin_shell_t, opt_apps_lambda_config_t, {file : read, write})                   9
AddIV(opt_apps_lambda_t, opt_apps_lambda_data_t, {file : read, write})                 10
AddIV(opt_apps_lambda_t, opt_apps_lambda_log_t, {file : append})                       11
AddIV(opt_apps_lambda_t, opt_apps_lambda_config_t, {file : read})                      12
```

Listing 5.8 – Liste des mises à jour demandées.

```
[...]                                                                                  1
(user_shell_t, opt_apps_lambda_exec_t, {file : execute})                               2
(user_shell_t, opt_apps_lambda_t, {process : transition})                              3
(admin_shell_t, opt_apps_lambda_config_t, {file : read, write})                        4
(opt_apps_lambda_t, opt_apps_lambda_data_t, {file : read, write})                      5
(opt_apps_lambda_t, opt_apps_lambda_log_t, {file : append})                            6
(opt_apps_lambda_t, opt_apps_lambda_config_t, {file : read})                           7
[...]                                                                                  8
```

Listing 5.9 – Politique neutre résultant des mises à jour.

l'application `lambda` à partir de son shell de connexion. La ligne 2 autorise l'administrateur à modifier la configuration de l'application. Enfin, les lignes 3 à 6 donnent les droits d'accès nécessaires au fonctionnement de l'application `lambda` : transition des fichiers exécutables vers le contexte global de l'application, lecture et écriture des fichiers de données, ajout (`append`) d'évènements dans les journaux et lecture seule pour les fichiers de configuration.

5.2.4.3 Projection de la politique neutre

Une fois que les modifications ont été répercutées dans la politique neutre $\mathcal{LR}_{\mathcal{AC}}$, l'agent de mise à jour va la projeter pour configurer le mécanisme de contrôle d'accès. Il va alors produire un ensemble de règles $\mathcal{TR}_{\mathcal{AC}}$, qu'il va ensuite injecter dans SELINUX ou grsecurity suivant ce qui est installé sur le nœud.

Le listing 5.10 donne la projection dans le cas de SELINUX. Les lignes 1 à 5 sont les déclarations des types. Les lignes 7 à 12 donnent les permissions d'accès décrites dans la partie précédente. La règle de transition en ligne 14 indique qu'une transition de contexte se produit lorsqu'un utilisateur exécute l'application `lambda`.

Le listing 5.11 donne la projection dans le cas de grsecurity. Les lignes 1 à 4 concernent le rôle de d'administrateur, et autorise l'accès en lecture et écriture aux fichiers de configuration de l'application `lambda` à partir du shell. Les lignes 6 à 15 concernent le rôle d'utilisateur, et autorisent l'exécution de l'application `lambda` par les utilisateurs, ainsi que les accès de l'application à ses propres fichiers.

```
type opt_apps_lambda_t;
type opt_apps_lambda_exec_t;
type opt_apps_lambda_data_t;
type opt_apps_lambda_config_t;
type opt_apps_lambda_log_t;
[...]
allow user_shell_t opt_apps_lambda_exec_t:file { execute }
allow user_shell_t opt_apps_lambda_t:process { transition }
allow admin_shell_t opt_apps_lambda_config_t:file { read write }
allow opt_apps_lambda_t opt_apps_lambda_data_t:file { read, write }
allow opt_apps_lambda_t opt_apps_lambda_log_t:file { append }
allow opt_apps_lambda_t opt_apps_lambda_config_t:file { read }
[...]
type_transition user_shell_t opt_apps_lambda_exec_t:process opt_apps_lambda_t;
[...]
```

Listing 5.10 – Configuration générée pour SELINUX.

```
role admin
[...]
subject /bin/bash
   /opt/apps/lambda/config/      rw

role user
[...]
subject /bin/bash
   /opt/apps/lambda/bin/         x
[...]

subject /opt/apps/lambda/bin
   /opt/apps/lambda/data         rw
   /opt/apps/lambda/log          a
   /opt/apps/lambda/config       r
[...]
```

Listing 5.11 – Configuration générée pour grsecurity.

5.3 Application de la Méta-Politique à la détection d'intrusion

Dans la section 5.2, nous avons appliqué le modèle général de Méta-Politique au contrôle d'accès. Cependant, d'autres applications sont envisageables, et notamment parmi elles :

- Les mécanismes d'authentification, pour configurer le niveau d'authentification requis suivant l'utilisateur, l'origine des connexions, etc. ;
- La sécurité applicative, par exemple pour configurer des échanges de politiques de sécurité entre deux applications.
- La détection d'intrusion, pour déployer les politiques de configuration des différents outils disponibles ;

En particulier, une coopération entre la Méta-Politique et la détection d'intrusion a fait l'objet d'une étude complémentaire à laquelle nous avons participé. Les aspect purement détection d'intrusion sont l'objet d'une autre thèse [Briffaut 2007]. C'est pourquoi, dans cette section, nous expliquons simplement les principes de la coopération entre notre méta-politique et une méthode de détection

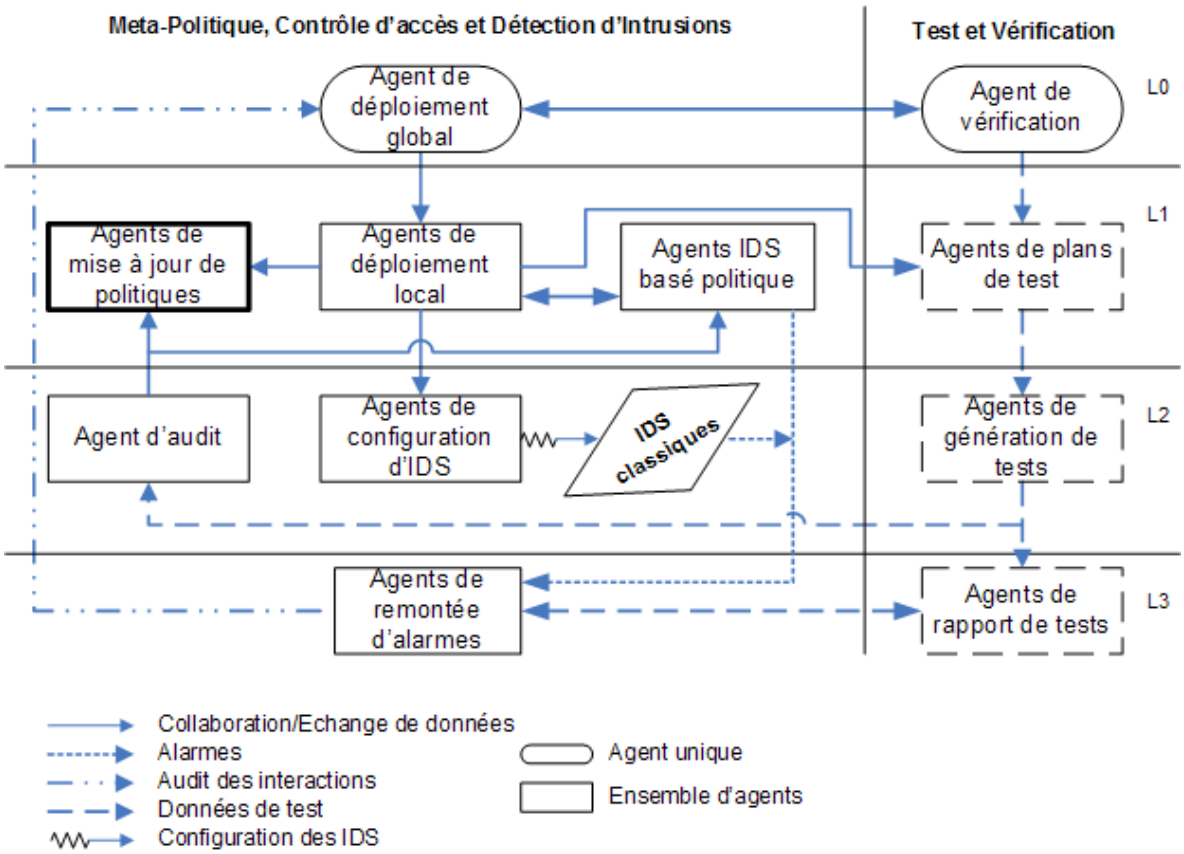

FIG. 5.5 – Architecture multi-niveaux et multi-agents

d'intrusion spécifique qui a été développée par ailleurs. Dans un premier temps, nous décrirons l'architecture multi-niveaux et multi-agents illustrant la complémentarité entre les deux études. Ensuite, nous verrons comment l'approche méta-politique peut être complétée par une solution de détection d'intrusion.

5.3.1 Architecture multi-niveaux et multi-agents pour le contrôle d'accès et la détection d'intrusion

L'objectif de cette architecture est de garantir des propriétés de sécurité globales sur un système réparti. Différents mécanismes de sécurité sont associés (politique de protection, contrôle d'accès, détection d'intrusion). L'utilisation conjointe de ces mécanismes complémentaires offre un niveau de sécurité plus élevé que lorsqu'ils sont utilisés séparément. Notamment, une nouvelle approche de détection d'intrusion basée sur la politique neutre de contrôle d'accès a été conçue, et autorise la détection d'actions illégales qui ne peuvent pas être prévenues par le contrôle d'accès seul. Cette approche a fait l'objet de plusieurs publications [Abou El Kalam *et al.* 2005b, Abou El Kalam *et al.* 2005a, Briffaut *et al.* 2006b, Briffaut *et al.* 2006a].

La figure 5.5 présente l'architecture multi-niveaux, dans laquelle chaque niveau possède un rôle spécifique. Les niveaux l_0 et l_3 sont des niveaux globaux communs à l'ensemble des noeuds, l_1 et l_2 sont des niveaux locaux déployés sur chacun des noeuds.

l_0 : Gestion de la Méta-Politique.

l_1 : Contrôle d'accès et détection d'intrusions basée politique.

l_2 : Détection d'intrusion "classique" et audit des interactions.

l_3 : Analyse et corrélation d'évènements.

Le niveau l_0 présente un agent unique pour tout le système réparti. Il s'agit de l'*agent de déploiement global*. Il prend en charge la distribution de la méta-politique, qui contient la configuration de tous les agents.

Le niveau l_1 représente le niveau de gestion de politique sur chacun des noeuds du système réparti. On retrouve l'*agent de mise à jour* déjà présenté. A côté, on a l'*agent de déploiement local* qui distribue la méta-politique aux autres agents du niveau l_1, c'est-à-dire notre agent et l'*agent de détection d'intrusions basée politique*. Les fonctions de ce dernier agent seront détaillées plus bas, dans la partie 5.3.3.

Le niveau l_2 est celui des mécanismes de sécurité intégrés au système d'exploitation de chaque noeud. Il intègre l'*agent d'audit*, qui surveille les accès effectués entre contextes de sécurité sur le noeud, et remonte les interactions marquées pour l'audit lorsqu'elles se produisent. Il contient aussi l'*agent de configuration des* IDS (Intrusion Detection System ou Système de détection d'intrusions). Enfin, on y trouve les IDS eux-mêmes.

Le niveau l_3 est celui de la génération de rapports d'alarmes. Il contient un *agent de remontée des alarmes*, qui reçoit les alarmes émises par les autres agents, et les remonte vers le niveau l_0.

En parallèle, sur la droite de la figure 5.5, on a la partie responsable de la vérification et des tests. Au niveau l_0, on trouve un *agent de vérification*. Il implante les méthodes de vérification de présence de flux d'information illégaux dans la Méta-Politique (cf. chapitre 6).

Ensuite, aux niveaux l_1, l_2 et l_3, on a différents agents concernant le test. Respectivement, il s'agit de l'*agent de plans de test* qui reçoit une politique de planification des tests, et commande les agents du niveau l_2, l'*agent de génération des test*, qui déclenche des procédures de test suivant les ordres qu'il reçoit du niveau l_1, et enfin l'*agent de rapport de tests* qui rassemble les résultats des tests et les communique à l'agent de remontée des alarmes, qui se charge de remonter ces informations au niveau l_0 [Briffaut *et al.* 2006a].

5.3.2 Méta-Politique de détection d'intrusion

La Méta-Politique appliquée à la détection d'intrusion autorise le déploiement et la mise à jour de politiques de détection d'intrusion locale sur l'ensemble des nœuds d'un système réparti.

Le tableau 5.2 donne la correspondance entre la définition générale de notre Méta-Politique, et les applications au contrôle d'accès et à la détection d'intrusion. La Méta-Politique $\mathcal{IR}_{\mathcal{IDS}}$ contient deux politiques $\mathcal{MR}_{\mathcal{IDS}}$ et $\mathcal{IR}_{\mathcal{IDS}}$. La politique de modification $\mathcal{MR}_{\mathcal{IDS}}$ contient des règles de modification de la politique de détection d'intrusion locale $\mathcal{LR}_{\mathcal{IDS}}$. La politique $\mathcal{IR}_{\mathcal{IDS}}$ contient les règles initiales de $\mathcal{LR}_{\mathcal{IDS}}$. Enfin, la politique $\mathcal{TR}_{\mathcal{IDS}}$ représente la projection de $\mathcal{LR}_{\mathcal{IDS}}$ pour un mécanisme de détection cible.

Étant donné que cette étude de la détection d'intrusion est hors du cadre de cette thèse, nous ne détaillerons pas ici l'emploi de ces politiques marquées $\mathcal{IDS}$. Pour plus d'informations, le lecteur se reportera aux articles [Abou El Kalam *et al.* 2005b, Abou El Kalam *et al.* 2005a, Briffaut *et al.* 2006b, Briffaut *et al.* 2006a].

5.3.3 Coopération du contrôle d'accès et de la détection d'intrusion

Lorsque la méta-politique est utilisée pour l'administration du contrôle d'accès, la détection d'intrusion peut être utilisée pour renforcer la sécurité de l'architecture sur deux points : pour détecter

	Méta-Politique $\mathcal{MP}$	Politique locale $\mathcal{LR}$	Politique projetée $\mathcal{TR}$
Contrôle d'accès	$\mathcal{IR}_{\mathcal{AC}}$ $\mathcal{MR}_{\mathcal{AC}}$	$\mathcal{LR}_{\mathcal{AC}}$	$\mathcal{TR}_{\mathcal{AC}}$
Détection d'intrusion	$\mathcal{IR}_{\mathcal{IDS}}$ $\mathcal{MR}_{\mathcal{IDS}}$	$\mathcal{LR}_{\mathcal{IDS}}$	$\mathcal{TR}_{\mathcal{IDS}}$

TAB. 5.2 – Méta-Politique de contrôle d'accès et de détection d'intrusion.

des évolutions de la politique $\mathcal{LR}_{\mathcal{AC}}$ non autorisées par la politique de modification $\mathcal{MR}_{\mathcal{AC}}$, et pour vérifier que la traduction de la politique $\mathcal{LR}_{\mathcal{AC}}$ vers la configuration du mécanisme cible $\mathcal{TR}_{\mathcal{AC}}$ est correcte. Ces vérifications complémentaires seront effectuées par un agent de détection d'intrusion, en utilisant une méthode basée sur l'interprétation de la Méta-Politique de contrôle d'accès.

Concernant le premier point, un IDS basé sur la Méta-Politique va être utilisé pour détecter des modification illégales de la politique $\mathcal{LR}_{\mathcal{AC}}$. Si de telles modifications se produisent, cela peut signifier deux choses. Premièrement, il peut s'agir d'une compromission de l'agent de mise à jour, dont le fonctionnement est détourné pour effectuer des modifications arbitraires de la politique de protection. Deuxièmement, il peut s'agir d'une erreur dans l'agent, ou dans le système d'exploitation.

Concernant le second point, pour détecter que la traduction de $\mathcal{LR}_{\mathcal{AC}}$ vers $\mathcal{TR}_{\mathcal{AC}}$ est correcte, un IDS spécifique obtient la politique $\mathcal{LR}_{\mathcal{AC}}$ courante auprès de l'agent de mise à jour, et surveille les accès effectués par les sujets du système. Il sera ainsi capable de relever les accès violant la politique $\mathcal{LR}_{\mathcal{AC}}$. En particulier, ce mécanisme de détection d'intrusions peut être utilisé pour pallier les déficiences du contrôle d'accès, notamment dans le cas où l'on appliquerait la Méta-Politique sur un système d'exploitation à base de DAC.

5.4 Conclusion

Dans ce chapitre, nous avons établi un modèle formel de notre méta-politique, conformément aux spécifications données dans le chapitre 4.

Nous avons commencé par définir un modèle général, avec un langage neutre de politique de protection, et un langage d'expression de contraintes de modification. En particulier, nous avons défini les notions de contexte de sécurité, de vecteur d'interaction, de politique de protection et de politique de contrôle des modifications au sein d'un modèle formel unifié. Ce modèle décrit formellement les langages neutres, et en conséquence définit formellement la Méta-Politique.

Ensuite nous avons présenté l'application de ce modèle au cas du contrôle d'accès. Après avoir explicité un modèle formel du contrôle d'accès, nous avons défini formellement les notions de contextes de sécurité et vecteurs d'interaction dans le cas du contrôle d'accès. Nous avons détaillé l'application des langages neutres, définis en première section, au contrôle d'accès. Nous avons illustré cette application par un exemple détaillé d'administration sur l'installation d'une nouvelle application et la création de règles appropriées pour son exécution.

Enfin nous avons présenté une autre application de la méta-politique, issue d'un travail conjoint sur la détection d'intrusion [Briffaut 2007]. Cette approche définit une architecture multi-agents et multi-niveaux, couplant le contrôle d'accès, la détection d'intrusions, la vérification de la Méta-Politique et le test. Nous avons détaillé des cas de coopération entre la Méta-Politique de contrôle d'accès et la détection d'intrusion.

Après avoir établi le modèle formel de notre architecture, nous pouvons l'utiliser dans le but de développer des méthodes hors-ligne de vérification de la présence de flux d'information illégaux. Le

chapitre 6 partira de cette formalisation pour établir des algorithmes de recherche de flux d'information dans la méta-politique.

Chapitre 6

Vérification des objectifs de sécurité dans la Méta-Politique

Un des objectifs présentés en section 2.1 est de garantir la confidentialité et l'intégrité des données manipulées par différents processus dans les systèmes ou les clusters de calcul. Il s'agit de garantir l'absence de flux d'informations entre des contextes de sécurité particuliers, par exemple entre un contexte sujet apache et un contexte objet correspondant au fichier `/etc/passwd`.

Dans cette partie, nous nous intéressons aux garanties d'absence de flux d'informations entre deux contextes de sécurités dans notre contexte de Méta-Politique. En effet, un système MAC ne peut pas contrôler les séquences d'interactions car les règles d'accès ne contrôlent que les interactions une à une. Il est cependant possible de développer des méthodes de vérification des flux d'information basées sur ces règles de contrôle d'accès. Dans [Guttman *et al.* 2003], sont exposés les principes de l'outil SLAT qui permet de faire une telle vérification sur les règles de contrôle d'accès SELINUX.

Dans notre cas, le problème de vérification devient plus difficile puisqu'il s'agit de prendre en compte la possibilité de modifier la politique de contrôle d'accès. En effet, dans le cadre d'un outil comme SLAT, la vérification est basée sur une politique statique (qui n'évolue pas). Dans le cadre de cette thèse, la Méta-Politique autorise à ce que les politiques soient dynamiques puisqu'il est possible de créer de nouveaux contextes et de nouvelles interactions. On souhaiterait cependant pouvoir garantir l'absence de flux d'informations pour tous les évolutions possibles des politiques.

Bien que le cœur de la thèse ne soit pas la vérification, nous allons montrer comment utiliser la Méta-Politique pour garantir l'absence de flux d'informations. Un outil comme SLAT n'est pas utilisable et il est nécessaire de développer des solutions algorithmiques spécifiques pour réaliser les vérifications.

Par ailleurs, la vérification de l'absence de flux d'information illégaux est un problème peu étudié dans les modèles de contrôle d'accès. Le modèle HRU [Harrison *et al.* 1976] s'intéresse à la sûreté des modèles de protection dans l'optique de prévenir l'apparition de nouvelles interactions dans le cadre de politiques dynamiques. Or ce problème est résolu par les modèles MAC (cf. partie 3.5), qui empêchent que la politique de contrôle d'accès soit modifiée par les utilisateurs. Dans la définition des modèles RBAC [Sandhu *et al.* 1996], le modèle RBAC_2 autorise la définition de contraintes sur l'assignation des rôles aux utilisateurs (cf. partie 2.1.3). Mais là encore, il ne s'agit pas de garanties concernant les flux d'information. Enfin, les modèles d'expression de politiques multi-domaines intègrent eux aussi des définitions de contraintes (cf. partie 2.3.1.2). Là, il s'agit uniquement de contraindre les actions des utilisateurs, suivant des paramètres de temps, de nombres ou de causalité d'une action. Il n'est pas question d'empêcher l'apparition de flux d'information illégaux.

Après avoir exposé les objectifs ainsi que les algorithmes de vérification sur une politique statique, nous présentons différents schémas de Méta-Politiques pour lesquels les techniques de vérification sont différentes. Ces différentes techniques permettent d'analyser la Méta-Politique et une approche générale en est déduite pour obtenir les garanties de sécurité, et ce malgré le caractère dynamique et entièrement réparti de la politique.

6.1 Objectifs

Garantir l'absence d'un flux d'information entre deux contextes de sécurité A et B consiste à prévenir la possibilité d'une séquence d'interactions qui véhiculerait de l'information de A vers B. En dehors de l'aspect vérification, il existe des techniques de détection ou de prévention temps réel. Le principe général de ces techniques temps réels consiste à mettre en œuvre des mécanismes de sécurité qui :

- interdisent la dernière interaction véhiculant de l'information en B (IPS) ;
- détectent *a posteriori* que de l'information a circulé de A vers B (IDS).

A l'inverse, la solution de Méta-Politique définie dans cette thèse permet de développer des techniques de vérification hors ligne pour obtenir des garanties sur ces politiques avant leur déploiement effectif. L'approche est intéressante puisqu'elle permet une garantie par validation de la Méta-Politique tout en autorisant des évolutions entièrement réparties des politiques ce qui rend plus flexible l'administration de celles-ci.

Pour une politique statique qui sera déployée sur un système et qui n'évolue donc pas, il est possible de vérifier s'il existe un chemin véhiculant de l'information de A vers B. Des outils tels que SLAT utilisant des techniques de MODEL-CHECKING ou des algorithmes de recherche de chemins permettent d'exhiber des flux indésirables [Herzog et Guttman 2002, Guttman *et al.* 2003, Guttman *et al.* 2005]. Ils sont présentés succinctement en section 6.2.

Dans d'autres travaux [Briffaut *et al.* 2006c, Blanc *et al.* 2006], nous proposons des mécanismes de détection d'intrusion permettant de surveiller ces flux d'informations interdits, basés sur des systèmes MAC. Le principe de ces outils réside dans la recherche d'un ou plusieurs chemins dans un graphe représentant la politique de contrôle d'accès : les nœuds du graphe représentent les contextes de sécurité et les arcs du graphe représentent les transitions de contextes ou une permission d'accès entre deux contextes. La figure 6.1 représente une politique de sécurité statique ou un flux d'information interdit est possible entre les contextes de sécurité A et B. On voit clairement que ce n'est pas le contrôle des interactions une à une qui empêche le flux interdit puisque ce flux est obtenu par transitivité.

Lorsque l'on autorise des modifications de la politique locale d'un système, ces modifications étant contrôlées par la Méta-Politique, il peut s'avérer difficile de garantir que les flux d'informations interdits le resteront puisqu'un administrateur peut ajouter des contextes de sécurités ou des interactions. Dans la partie suivante, nous détaillerons plusieurs exemples qui illustreront les conséquences d'une Méta-Politique particulière et nous présenterons comment il est possible de vérifier l'absence de flux d'informations interdits dans de telles situations. De plus, si la technique de vérification détecte un flux d'information interdit, l'administrateur peut alors modifier la Méta-Politique en conséquence. C'est pourquoi cette vérification doit être effectué lors de l'écriture de la Méta-Politique, et avant de la déployer sur un système réparti.

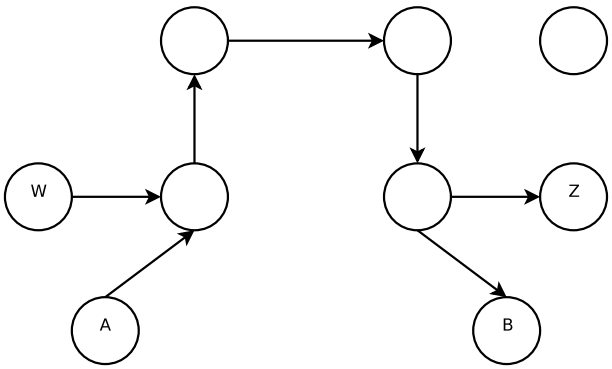

FIG. 6.1 – Exemple de graphe représentant une politique.

6.2 Détection de flux d'information dans les politiques statiques

Si l'on oublie un instant la Méta-Politique et donc la possibilité de modifier la politique locale d'un système, détecter un flux d'information entre A et B revient à déterminer l'existence d'un chemin de A vers B dans le graphe (introduit en section 6.1) de la politique initial $\mathcal{IR}_{\mathcal{AC}}$ ou de la politique projetée $\mathcal{TR}_{\mathcal{AC}}$. Différents outils algorithmiques permettent de trouver ces chemins. Nous présentons succinctement SLAT et PIGA.

SLAT (SELinux Analysis Tools) utilise une technique de MODEL-CHECKING pour rechercher des chemins dans le graphe représentant la politique de SELINUX. Le graphe est construit en plaçant les contextes de sécurité sur les sommets, et en plaçant les permissions d'accès sur les arêtes. Les chemins recherchés peuvent être arbitraires, et sont définis par une syntaxe particulière. On peut préciser un contexte de départ, un contexte d'arrivée, des contextes intermédiaires, et pour chaque arête du chemin, on peut spécifier une ou plusieurs permissions à chercher. Pour que la recherche soit pertinente, ce chemin doit correspondre à un flux d'information illégal, dont on veut vérifier l'absence dans la configuration de SELINUX. La définition du chemin à rechercher et la configuration SELINUX sont passées en entrée d'un *model checker*, et celui-ci indique si un tel chemin existe dans la configuration.

PIGA (Policy Interaction Graph Analysis) est un outil basé sur la recherche de plus courts chemins (cf. article [Briffaut *et al.* 2006c]). Il utilise un algorithme de recherche des k plus courts chemins. PIGA étant un outil de détection d'intrusion, il s'agit de chercher tous les chemins possibles permettant le flux d'information. Dans le cadre de la vérification pour le contrôle d'accès, l'objectif est uniquement de vérifier l'existence d'un chemin. Dans ce cas, un algorithme de plus court chemin simple comme Dijkstra suffit.

6.3 Influence de la Méta-Politique sur la vérification

L'objet de cette partie est de présenter plusieurs exemples de Méta-Politiques et les conséquences sur les garanties obtenues en terme de flux d'information entre deux contextes de sécurité A et B.

Nous présentons tout d'abord deux cas extrêmes de Méta-Politiques :

- le cas 0 correspond à une Méta-Politique qui ne permet aucune évolution de la politique locale.
- le cas 1 correspond à une Méta-Politique qui permet toutes les évolutions de la politique locale, c'est-à-dire qu'en pratique il n'y a pas de contraintes.

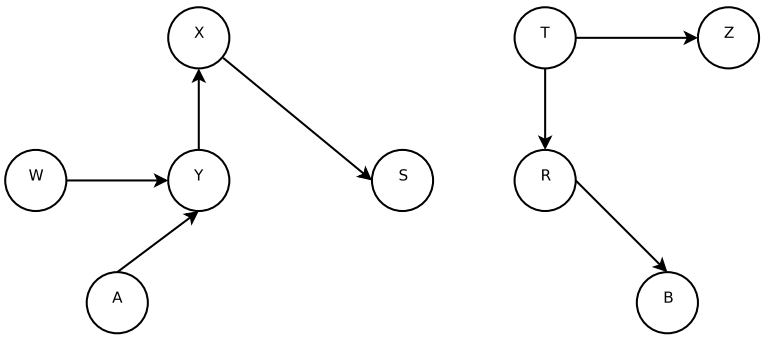

FIG. 6.2 – Politique statique ne comportant pas de flux d'information entre A et B.

Ces deux cas n'ont pas forcément d'intérêt pratique mais illustrent les situations extrêmes ou le problème de vérification devient aisé.

Nous présentons ensuite un cadre général qui utilise sur les expressions régulières pour désigner des nœuds. Ainsi, il est nous est possible de prendre en compte des interactions entre deux ensembles quelconques de nouveaux nœuds. Enfin, nous présentons les principes algorithmiques qui permettent de vérifier l'absence de flux interdits.

Nous illustrons ce principe général au moyen de deux exemples permettant d'aider à l'administration du système et nous présentons comment vérifier les propriétés de flux d'information dans ces cas précis.

6.3.1 Cas extrême 0

Nous considérons ici le cas où les règles de modification $\mathcal{MR}_{\mathcal{AC}}$ n'autorisant aucune modification de la politique locale $\mathcal{LR}_{\mathcal{AC}}$. Cela correspond à $\mathcal{MR}_{\mathcal{AC}} = \emptyset$. Dans ce cas particulier, la politique locale $\mathcal{LR}_{\mathcal{AC}}$ ne peut évoluer. Vérifier un flux d'information interdit entre un contexte de sécurité A et un contexte de sécurité B consiste donc à chercher un chemin dans le graphe de la politique local. Dans des outils comme SLAT et PIGA suffisent à vérifier l'absence de flux d'information.

Ce cas n'utilise pas toutes les possibilités de la Méta-Politique car il s'agit d'un système avec une politique statique sans possibilité d'évolution, c'est à dire $\mathcal{LR}_{\mathcal{AC}} = \mathcal{IR}_{\mathcal{AC}}$. Par exemple, la figure 6.2 présente une politique $\mathcal{IR}_{\mathcal{AC}}$ ne comportant pas de chemin entre A et B. Un outil comme PIGA est capable d'analyser cette politique écrite en langage neutre (cf. section 5.1.3). SLAT ne peut le faire mais peut par contre analyser son équivalent projeté en une politique SELinux $\mathcal{TR}_{\mathcal{AC}}$ (cf. section 5.2.1.2).

6.3.2 Cas extrême 1

Les règles du listing 6.1 sont un exemple de règles de modification $\mathcal{MR}_{\mathcal{AC}}$ autorisant la création de n'importe quels contextes de sécurité et de n'importe quelles interactions entre ces contextes dans la politique locale $\mathcal{LR}_{\mathcal{AC}}$. La règle 1 du listing 6.1 permet de créer une infinité de contextes de sécurité à l'aide de l'expression régulière `.*`. Cela peut s'écrire $\mathcal{SC} = \{.*\}$.

La règle 2 du listing 6.1 permet de créer autant d'interactions qu'il existe de couple de contextes de sécurités (paramètre 2 et 3 de la règle 2). Étant donné qu'il est possible avec la règle 1 de créer une infinité de contextes, on peut donc créer une infinité d'interactions. Une de ces nouvelles interactions peut autoriser n'importe quelle opération de l'ensemble $\mathcal{IS}_{\mathcal{AC}}$ (paramètre 4 de la règle 2).

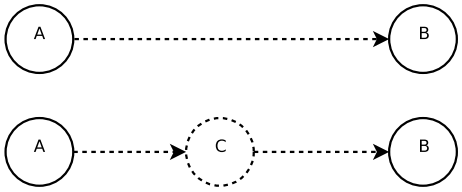

FIG. 6.3 – Exemples de flux d'informations entre A et B.

```
enableAddSC(sc_admin, .*)                                  1
enableAddIV(sc_admin, (.*, .*, \{.*\}))                    2
```

Listing 6.1 – Méta-Politique 1

A l'aide d'une telle Méta-Politique, il est possible de créer un flux d'information entre deux contextes de sécurité A et B du système. En effet, d'une part il est possible à l'administrateur de créer directement une interaction entre A et B en utilisant la règle 2. D'autre part, ll est possible de créer un contexte de sécurité intermédiaire C (règle 1) et d'ajouter deux interactions, une entre A et C et une autre entre C et A. Ces deux possibilités sont représentées en figure 6.3 ou les nœuds (resp. arcs) en pointillés sont les contextes de sécurité (resp. interactions) crées par l'administrateur.

Il est clair qu'une telle Méta-Politique ne permet plus d'avancer des garanties sur le système ni même de conserver les garanties qui seraient vérifiées sur la politique initiale $\mathcal{IR}_{\mathcal{AC}}$. Il est toutefois simple de développer un outil d'analyse des règles $\mathcal{MR}_{\mathcal{AC}}$ de la Méta-Politique pour rechercher des règles du type 1 et 2 comme présenté dans le listing 6.1 et les présenter à l'administrateur pour l'inviter à les modifier.

6.3.3 Canevas générique

Les règles du listing 6.2 utilisent deux expressions régulières qui désignent donc deux ensembles de nœuds. Grâce à une méta-politique contenant de telles règles $\mathcal{MR}_{\mathcal{AC}}$, il est possible d'autoriser des interactions entre deux ensembles quelconques de nouveaux nœuds. L'idée est de permettre la création de deux ensembles de contextes de sécurité désignés par deux expressions régulières ($expr_1$ et $expr_2$) et d'autoriser la création d'interactions entre des nœuds entre ces deux ensembles. Il s'agit d'un canevas générique où nous ne faisons aucune hypothèse sur la valeur des expressions régulières. Dans cette partie, nous donnons une solution de vérification pour ce canevas de Méta-Politique. Ce canevas offre un pouvoir d'expression libre puisque les expressions régulières sont quelconques. Nous illustrons ensuite ce canevas en section 6.3.3.2 et 6.3.3.3 au moyen de deux exemples ayant un intérêt du point de vue de l'administration du système réparti.

La règle 1 du listing 6.2 autorise la création d'un contexte de sécurité respectant l'expression régulière $expr_1$, de même pour la règle 2 et l'expression régulière $expr_2$. La règle 3 autorise la création d'interactions entre tous les contextes de l'ensemble correspondant à l'expression $expr_1$ et tous ceux de l'ensemble $expr_2$. Les opérations autorisées pour ces nouvelles interactions sont toutes celles de l'ensemble $\mathcal{IS}_{\mathcal{AC}}$.

```
enableAddSC(sc_admin, expr_1)                              1
enableAddSC(sc_admin, expr_2)                              2
enableAddIV(sc_admin, (expr_1, expr_2, {.*}))              3
```

Listing 6.2 – Cas général

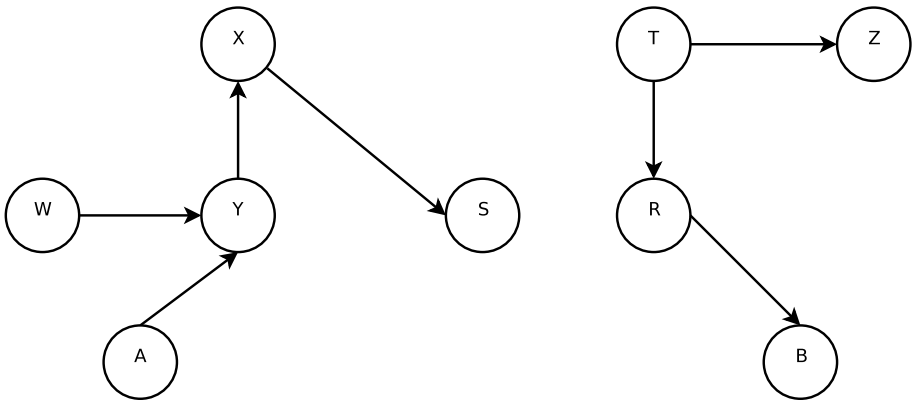

FIG. 6.4 – Graphe pour une politique initiale $\mathcal{LR}_{\mathcal{AC}}$.

Pour pouvoir vérifier un flux d'information entre deux contextes A et B, nous décrivons en section 6.3.3.1 un algorithme qui construit un nouveau graphe à partir du graphe de la politique initiale et des règles d'évolution. Nous montrons ensuite comment utiliser ce nouveau graphe pour détecter les flux d'informations entre A et B.

Nous limiterons l'étude aux règles présentées dans le listing 6.2 et nous verrons comment élargir les méthodologies présentées lorsqu'il s'agit de prendre en compte un ensemble de règles utilisant le canevas du listing 6.2. La réutilisation de ce canevas générique permettra ainsi de traiter le cas général d'une méta-politique qui permet des évolutions quelconques.

6.3.3.1 Algorithme de contraction et Vérification

Dans cette partie, nous présentons les principes algorithmiques qui permettent de construire le nouveau graphe correspondant à la Méta-Politique du listing 6.2 et comment l'utiliser lors du processus de vérification. L'idée principale de l'algorithme construisant le nouveau graphe repose sur la création de nœuds particuliers appelés méta-nœuds. Chaque méta-nœud contracte un ensemble de nœuds ce qui permet de représenter la Méta-Politique comme un graphe. Il est alors possible de faire de la vérification en analysant ce graphe.

Algorithme de contraction L'algorithme prend en entrée les deux politiques $\mathcal{MR}_{\mathcal{AC}}$ et $\mathcal{LR}_{\mathcal{AC}}$ présentes dans la Méta-Politique. Un exemple de $\mathcal{LR}_{\mathcal{AC}}$ est représenté en figure 6.4. La politique de modification de notre cas général $\mathcal{MR}_{\mathcal{AC}}$ est décrite par le listing 6.2.

Dans le graphe de la politique nous ajoutons deux méta-nœuds dont le nom correspond aux expressions régulières $expr_1$ et $expr_2$. Chaque méta-nœud correspondant à l'expression régulière $expr_i$ est un conteneur $\mathcal{C}_i$, représenté par des rectangles sur la figure 6.5, qui inclut :

- les contextes de sécurité existants dont le nom est un élément de l'ensemble des noms décrits par $expr_i$. Ils sont représentés par des cercles pleins à l'intérieur du conteneur dans la figure 6.5.
- les contextes de sécurité qui peuvent être potentiellement créés grâce à la règle 1 du listing 6.2. Ils sont représentés par des cercles en pointillés à l'intérieur du conteneur dans la figure 6.5. Ces contextes de sécurités potentiels n'ont pas besoin d'être réellement ajoutés dans le méta-nœud, ils sont contractés par le conteneur même.

Nous ajoutons un arc entre les deux méta-nœuds correspondant à la règle 3 du listing 6.2. Cet arc correspond à la possibilité de créer dynamiquement une interaction entre deux nouveaux nœuds.

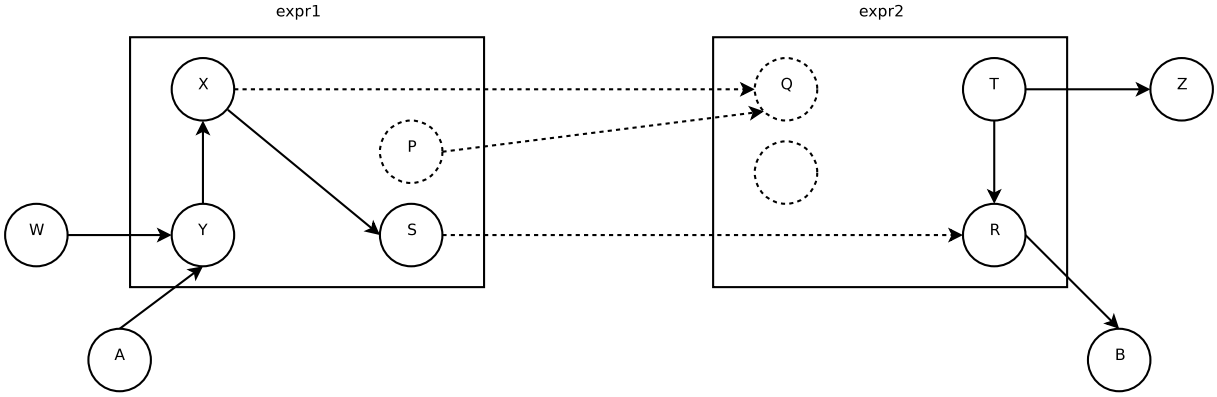

FIG. 6.5 – Regroupement des nœuds correspondant aux règles de la politique de modification. $\mathcal{MR}_{\mathcal{AC}}$ du listing 6.2

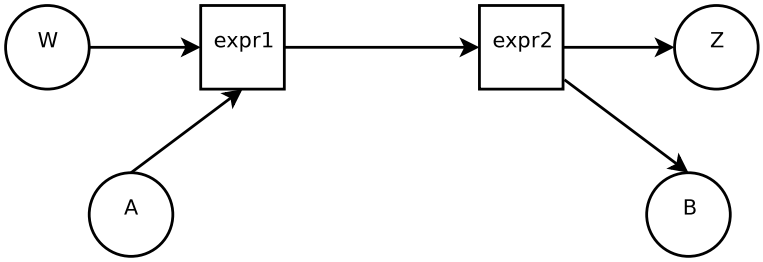

FIG. 6.6 – Graphe résultant de l'algorithme de contraction.

Ce nouvel arc contracte trois cas d'interactions pouvant être créées à partir des nœuds qui sont inclus dans le méta-nœud. Ces trois cas différents sont représentées dans la figure 6.5 par les arcs SR, XQ et PQ.

Pour chaque méta-nœud, nous ajoutons ensuite des arcs qui permettent de le connecter aux contextes de sécurité existants de la façon suivante : pour chaque nœud u existant tel que $u \in \mathcal{C}_i$ et possédant un arc avec un nœud v tel que $v \notin \mathcal{C}_i$, on créée un arc entre v et le méta-nœud $expr_i$. Cet ajout permet de conserver le fait qu'il existe une interaction entre le nœud v et un des nœuds du conteneur $\mathcal{C}_i$.

Ainsi l'algorithme construit un nouveau graphe (une sorte de contraction du graphe initial) qui fait apparaître les méta-nœuds $expr_1$ et $expr_2$ les interactions permises entre ces deux méta-nœuds ainsi que les interactions entre ces méta-nœuds et les contextes de sécurité existants dans la politique initiale. Ce nouveau graphe est représenté en figure 6.6. Ce graphe constitue bien une contraction du graphe de la figure 6.5. En effet, le méta-nœud agrège 1) les nœuds x, y, s existants dans la politique initiale et 2) tous les nouveaux nœuds tels que p et q qui peuvent être créés avec un nom conforme à l'une des deux expression régulières $expr_1$ ou $expr_2$.

Vérification A partir du graphe contracté, il est possible de faire une recherche de chemins pour vérifier l'absence de flux d'informations entre deux contextes de sécurité A et B en utilisant les méthodologies présentées en section 6.2.

La construction du graphe contracté est nécessaire car, comme le montre la figure 6.4, il n'y avait

aucun flux d'information possible entre A et B pour la politique initiale $\mathcal{IR}_{\mathcal{AC}}$ alors qu'il est possible de créer un chemin, comme montré en figure 6.5, conformément aux règles de modification $\mathcal{MR}_{\mathcal{AC}}$.

La recherche de l'existence d'un chemin sur le graphe contracté envisage le pire des scénarios puisqu'il prend en compte la possibilité de créer une interaction entre n'importe quel nœud du conteneur 1 et n'importe quel nœud du conteneur 2. Le cas présenté en figure 6.5 est volontairement général puisque le chemin utilise :

- un arc AY pour entrer dans le conteneur $expr_1$
- deux arcs YX et XS existants dans le conteneur $expr_1$
- un arc potentiel SR réalisant une interaction entre les méta-nœuds $expr_1$ et $expr_2$
- un arc RB pour sortir du conteneur $expr_2$

Il faut cependant remarquer qu'il n'est même pas nécessaire d'utiliser les interactions existantes YX et XS puisque potentiellement il est possible de créer une interaction directe YR. Ceci montre qu'il n'y a pas d'hypothèse sur la nature du graphe à l'intérieur du méta-nœud ce qui permet de prendre en compte tous les cas de figure de chemins et donc de réaliser la vérification dans le pire des cas.

6.3.3.2 Exemple 1

Les règles du listing 6.3 sont un exemple de règles de modification $\mathcal{MR}_{\mathcal{AC}}$ d'une Méta-Politique permettant de créer un TCB (cf. partie 4.2.3), c'est-à-dire une zone de confinement d'une application. Il s'agit d'un cas particulier du canevas générique donné par le listing 6.2 pour lequel $expr_1 = expr_2$.

La première règle du listing 6.3 autorise la création d'un contexte de sécurité quelconque. Une variable spéciale $\$1$ permet d'enregistrer le début du contexte de sécurité, par exemple $\$1 = "apache"$. Ainsi, la règle 2 permet d'autoriser la création d'une interaction entre deux contextes de sécurité commençant par $\$1$, par exemple entre `apache_http_t` et `apache_php_t`.

enableAddSC(sc_{admin}, $\$1.*$) 1
enableAddIV(sc_{admin}, ($\$1.*$, $\$1.*$, $\{.*\}$)) 2

Listing 6.3 – Méta-Politique 2

Une telle Méta-Politique permet de créer un ensemble de contextes de sécurité commençant par $\$1$ (on suppose que $\$1$ est non nul), puis d'autoriser toutes les interactions entre les contextes de sécurités respectant ce critère : il s'agit donc de permettre la création d'un TCB où les processus et les fichiers d'une application (par exemple apache) sont confinés.

De manière récursive, il est possible de définir un sous TCB dans le TCB correspondant à apache, en créant des contextes de sécurité ou $\$1 = "apache_php"$ par exemple. Cela permet par exemple au serveur web apache d'interagir avec un module php dont les effets sont confinés. Ce qui signifie qu'en pratique si une faille affecte php, celle-ci ne déborde pas et ne permet donc pas d'affecter les fichiers du serveur web apache.

Pour une telle Méta-Politique, on peut rechercher des flux d'information interdits pour le TCB, c'est-à-dire un flux entre un contexte du TCB et un contexte extérieur au TCB. En effet, pour un TCB il ne doit pas y avoir d'arc sortant du sous graphe associé au TCB. En pratique, même si le listing 6.3 correspond bien au canevas d'un TCB, il peut y avoir d'autres règles (par exemple dans la politique initiale $\mathcal{IR}_{\mathcal{AC}}$) qui permette de sortir du TCB. La problématique de vérification est alors de rechercher un chemin allant du méta-nœud A à tout nœud B extérieur au TCB.

La figure 6.7 schématise la problématique de vérification avec cette Méta-Politique. Le bloc correspondant aux contextes de sécurités du TCB commençant par $\$1 = "apache"$ est situé dans le rectangle : en traits plein sont représentés les contextes et interactions existantes, en pointillés est

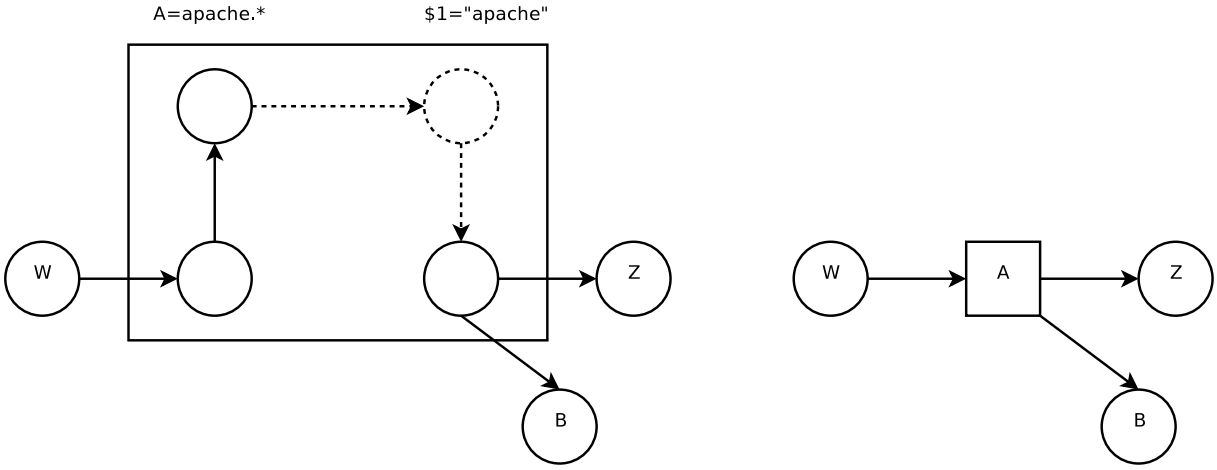

FIG. 6.7 – Exemple de contraction de graphe pour la Méta-Politique 2.

représenté la possibilité de créer des contextes et des interactions en utilisant les règles 1 et 2. Les contextes B, Z et W sont extérieurs à la zone $\$1.*$. On voit cependant clairement que la création de contextes et d'interactions permet de créer un flux d'information du méta-nœud A vers B.

En appliquant l'algorithme présenté en section 6.3.3.1 nous agrégeons la zone $\$1.*$ comme représenté dans la partie droite de la figure 6.7. Ainsi, les outils de recherche de chemin présentés en section 6.2 peuvent être utilisés pour exhiber le chemin.

On remarque ici que la vérification d'un TCB est efficace puisque on effectue une seule recherche de chemins pour l'ensemble des nœuds du TCB.

6.3.3.3 Exemple 2

Les règles $\mathcal{MR}_{\mathcal{AC}}$ du listing 6.4 permettent d'illustrer le canevas générique où les deux expressions régulières sont différentes et permettent de créer deux groupes de contextes de sécurité distincts et d'ajouter des interactions entre ces deux groupes.

Cet exemple vise à vérifier qu'un utilisateur (représenté par le contexte A) qui peut accéder au serveur apache ne peut qu'accéder aux données présentes dans le répertoire `/var/www`. La première règle du listing 6.4 autorise la création d'un contexte de sécurité pour les fichiers nécessaires à l'exécution d'apache. La règle 2 autorise la création de contextes de sécurité pour les pages web. Ensuite, la règle 3 permet à apache d'accéder au répertoire des pages web.

enableAddSC(sc_{admin}, apache.$*$) 1
enableAddSC(sc_{admin}, var_www.$*$) 2
enableAddIV(sc_{admin}, (apache.$*$, var_www.$*$, {.$*$})) 3

Listing 6.4 – Méta-Politique 3

La figure 6.8 représente, de manière similaire à la section 6.3.3.2 le graphe initial et le graphe contracté obtenu par notre algorithme. La vérification consiste à recherche un chemin entre un nœud A représentant le contexte utilisateur et pouvant intéragir avec le méta-nœud apache et un nœud B, par exemple `etc_passwd`. Le chemin ainsi exhibé donne la possibilité à l'administrateur de supprimer les interactions qu'il juge inapproprié (ici, il pourrait s'agir d'enlever l'interaction WB).

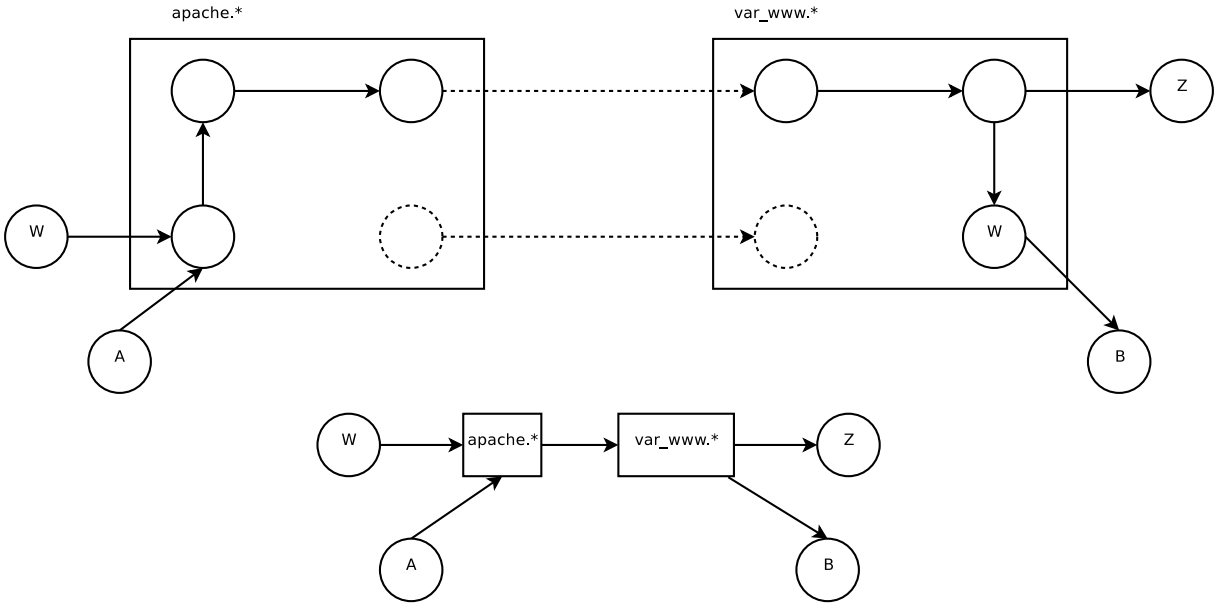

FIG. 6.8 – Exemple de contraction de graphe pour la Méta-Politique 6.4.

6.4 Méthode générale de vérification d'une Méta-Politique

Dans cette partie, nous montrons que la présence de plusieurs règles selon le canevas précédent pose un problème particulier qui nécessite l'adaptation de l'algorithme de vérification. Avec cette modification, la solution de vérification devient tout à fait générale.

La section 6.3.3 a présenté comment procéder à la vérification d'un flux d'information en considérant deux expressions régulières $expr_1$ et $expr_2$. Nous rappelons pour mémoire dans le listing 6.5 les règles du canevas générique. Nous appellerons par la suite cet ensemble de trois règles un *jeu de règles* et les deux expressions régulières considérées un *jeu d'expression régulières*.

enableAddSC(sc_{admin}, $expr_1$) 1
enableAddSC(sc_{admin}, $expr_2$) 2
enableAddIV(sc_{admin}, ($expr_1$, $expr_2$, $\{.*\}$)) 3

Listing 6.5 – Cas général

L'algorithme présenté en section 6.3.3 n'a pris en compte qu'un jeu de règles de modification. En pratique, il est prévu qu'au moyen de ce canevas général l'administrateur ajoute dans la Méta-Politique plusieurs jeux de règles. C'est pourquoi nous allons considérer deux jeux de règles utilisant un premier jeu d'expressions régulières $expr_1$ $expr_2$, puis un second utilisant un deuxième jeu d'expressions régulières $expr'_1$ $expr'_2$. Ainsi le listing 6.6 présente une telle Méta-Politique avec deux jeux de règles.

enableAddSC(sc_{admin}, $apache.*$) 1
enableAddSC(sc_{admin}, $php.*$) 2
enableAddIV(sc_{admin}, ($apache_1$, $php.*$, $\{.*\}$)) 3
4
enableAddSC(sc_{admin}, $.*\,http.*$) 5
enableAddSC(sc_{admin}, $var_www.*$) 6
enableAddIV(sc_{admin}, ($.*\,http.*$, $var_www.*$, $\{.*\}$)) 7

Listing 6.6 – Exemple d'une Méta-Politique utilisant de jeux de règles

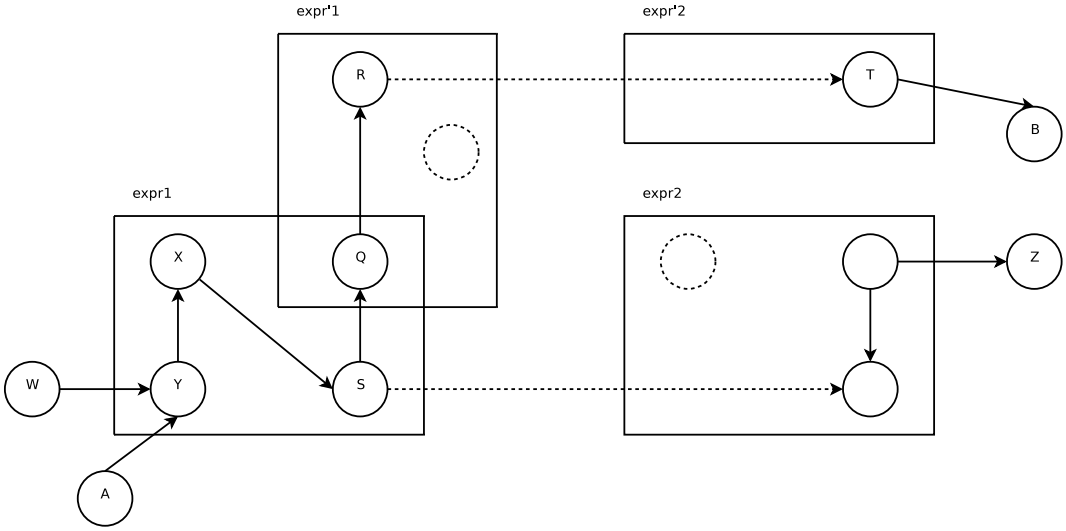

FIG. 6.9 – Exemple de deux graphes de politiques utilisant deux jeux d'expressions régulières.

Cet exemple va permettre de montrer en section 6.4.1 que l'algorithme présenté en section 6.3.3.1 ne fonctionne plus directement. La section 6.4.2 présentera alors la solution pour le cas général.

6.4.1 Problème d'intersection des expressions régulières

Nous montrons dans cette section que le problème provient de l'intersection des deux jeux d'expressions régulières. Nous considérons donc une généralisation du listing 6.5 où les expressions régulières $expr_1$ et $expr'_1$ ont une intersection. Cette généralisation correspond au listing 6.7.

enableAddSC(sc_{admin}, $expr_1$) 1
enableAddSC(sc_{admin}, $expr_2$) 2
enableAddIV(sc_{admin}, ($expr_1$, $expr_2$, {.∗})) 3
enableAddSC(sc_{admin}, $expr'_1$) 4
enableAddSC(sc_{admin}, $expr'_2$) 5
enableAddIV(sc_{admin}, ($expr'_1$, $expr'_2$, {.∗})) 6

Listing 6.7 – Cas général utilisant de jeux d'expressions régulières

Comme dans la section 6.3.3, nous représentons dans la figure 6.9 ces expressions par des rectangles englobant les contextes de sécurités existants (en traits pleins) et les contextes de sécurités qui peuvent potentiellement être créés (en pointillés).

Appliquons l'algorithme de contraction présenté en section 6.3.3. Pour le premier jeu de règle (règles 1, 2 et 3 du listing 6.7) cela conduit au graphe présenté en figure 6.10. Pour le deuxième jeu de règles (règles 4, 5 et 6 du listing 6.7) cela conduit au graphe présenté en figure 6.11. On voit clairement que le graphe 6.12 résultant de la fusion des deux graphes précédents ne comporte pas de chemin entre A et B.

On voit donc clairement que l'algorithme précédent ne fonctionne pas puisqu'il existe bien un chemin entre A et B comme représenté dans la figure 6.9. Le problème provient de la possibilité d'un nœud de type Q qui appartient à l'intersection de deux expressions régulières. On voit ainsi que considérer les deux jeux séparément ne convient pas. Dans la section 6.4.2 nous présentons une solution utilisant le calcul de l'intersection des jeux d'expressions régulières.

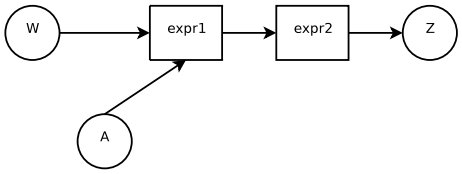

FIG. 6.10 – Contraction du premier jeu d'expressions régulières.

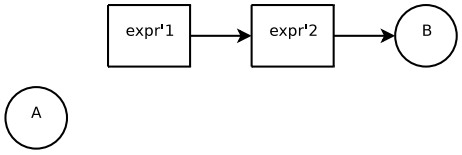

FIG. 6.11 – Contraction du second jeu d'expressions régulières.

6.4.2 Deuxième version de l'algorithme de contraction

Nous présentons succinctement une seconde version de l'algorithme de contraction permettant de contourner le problème posé par les intersections d'expression régulières. L'idée générale de notre algorithme repose sur le calcul d'intersection deux à deux des jeux d'expressions régulières et l'ajout d'arcs entre les méta-nœud présentant des intersections non vides.

Le canevas de l'algorithme est le suivant :

- Phase 1 : application de l'algorithme de la section 6.3.3 ;
- Pour tout couple de méta-nœud $expr_i$ $expr_j$, Si l'intersection est non vide, Alors ajouter un l'arc $(expr_i, expr_j)$ et l'arc $(expr_j, expr_i)$.

En utilisant cet algorithme, nous obtenons le graphe contracté représenté par la figure 6.13. On voit alors que le chemin entre A et B est ainsi exhibé.

6.5 Evaluation de la complexité de l'algorithme

Le nombre d'intersections d'expressions régulières à calculer est C_n^2 où n est la cardinalité de l'ensemble des expressions régulières contenues dans la politique $\mathcal{MR}_{\mathcal{AC}}$, que nous notons $exp_{\mathcal{MR}_{\mathcal{AC}}}$.

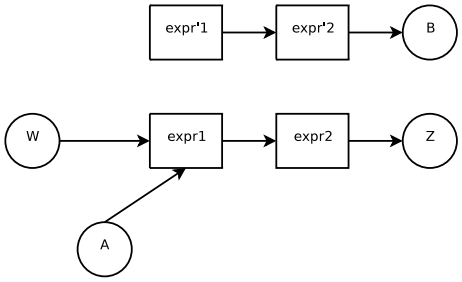

FIG. 6.12 – Graphe contracté global.

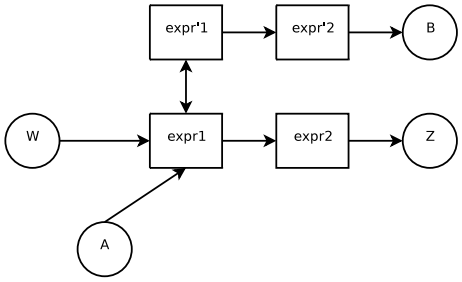

FIG. 6.13 – Graphe contracté global utilisant le deuxième algorithme.

Ceci donne une complexité pour le nombre d'intersections en :

$$O(|exp_{\mathcal{MR}_{\mathcal{AC}}}|^2)$$

La complexité de l'algorithme de recherche de plus court chemin de Dijkstra est $\theta(a + s.ln(s))$ où a est le nombre d'arêtes, et s le nombre de sommets. Dans notre cas, les arêtes correspondent aux vecteurs d'interaction présents dans la politique initiale $\mathcal{IR}_{\mathcal{AC}}$, et les sommets aux contextes de sécurité présents dans $\mathcal{IR}_{\mathcal{AC}}$ augmentés du nombre d'expressions régulières présentes dans $\mathcal{MR}_{\mathcal{AC}}$. Donc la complexité s'exprime, en tenant compte de la cardinalité de l'ensemble des vecteurs d'interaction (noté $iv_{\mathcal{IR}_{\mathcal{AC}}}$) et de l'ensemble des contextes de sécurité (noté $sc_{\mathcal{IR}_{\mathcal{AC}}}$) dans $\mathcal{IR}_{\mathcal{AC}}$, ainsi que de la cardinalité de l'ensemble des expressions régulières dans $\mathcal{MR}_{\mathcal{AC}}$, par :

$$O(|iv_{\mathcal{IR}_{\mathcal{AC}}}| + (|sc_{\mathcal{IR}_{\mathcal{AC}}}| + |exp_{\mathcal{MR}_{\mathcal{AC}}}|).ln(|sc_{\mathcal{IR}_{\mathcal{AC}}}| + |exp_{\mathcal{MR}_{\mathcal{AC}}}|))$$

L'algorithme effectue ces deux opérations de manière séquentielle, la complexité globale est donc la somme des deux complexités calculées précédemment.

$$O(|exp_{\mathcal{MR}_{\mathcal{AC}}}|^2 + |iv_{\mathcal{IR}_{\mathcal{AC}}}| + (|sc_{\mathcal{IR}_{\mathcal{AC}}}| + |exp_{\mathcal{MR}_{\mathcal{AC}}}|).ln(|sc_{\mathcal{IR}_{\mathcal{AC}}}| + |exp_{\mathcal{MR}_{\mathcal{AC}}}|))$$

Globalement, l'algorithme est en temps polynomial avec les données.

6.6 Conclusion

Dans ce chapitre, nous avons présenté une méthode de vérification de la présence de flux d'information illégaux dans la Méta-Politique. Cette méthode s'appuie sur la politique de modification pour garantir des propriétés de sécurité, et ce même si les politiques de protection sont dynamiques, et administrées de façon décentralisée.

Nous avons d'abord rappelé des résultats de vérification dans les politiques statiques. Puis nous avons exposé des cas extrêmes de politiques de modification, pour lesquels les résultats de la vérification sont simples à établir. Nous en avons déduit un algorithme général de recherche de flux d'information, s'appuyant sur la définition de méta-noeuds dans les graphes de politique, et le calcul d'intersection d'expression régulières.

Maintenant que nous disposons d'une méthode garantissant les objectifs de sécurité de la Méta-Politique, nous pouvons l'implanter, et ainsi étudier son application sur un système réparti réel.

Chapitre 7

Implantation de l'architecture Méta-Politique

Les chapitres 4 et 5 ont décrit et formalisé les principes de fonctionnement de l'architecture Méta-Politique, et ses composants. Les différentes politiques utilisées ont été décrites en section 4.1, puis leur langage a été formalisé suivant deux niveaux d'application : au niveau général en section 5.1, et au niveau du contrôle d'accès en 5.2. De plus, l'agent de mise à jour a été introduit en partie 4.1.3, ses fonctions ont été détaillées en section 4.3 et formalisées dans les parties 5.1.5 et 5.2.3.

Afin d'expérimenter le déploiement de cette architecture en conditions réelles d'utilisation, notamment l'utilisation des langage de protection et de contrainte, nous avons réalisé une implantation des langages et de l'agent de contrôle. Le langage neutre de politique (cf. 5.2.2.3) et le langage d'expression de contraintes de modification (cf. 5.2.2.4) ont été implantés avec XML, via la définition de types de documents (DTD) spécifiques. XML a été choisi pour son indépendance des plates-formes et des langages de programmation, et pour les facilités de chargement des documents au format XML. L'implantation de l'agent de mise à jour a été réalisée, dans le langage Python. Ce langage a été choisi parce qu'il autorise une implantation rapide, notamment parce qu'il intègre de nombreux types de données, et ensuite parce qu'il est orienté objet, et enfin parce qu'il dispose de plusieurs implantations de XML.

En revanche, nous n'avons pas implanté de mécanisme de distribution de la Méta-Politique, car cela ne rentrait pas dans le cadre de la thèse. La Méta-Politique est pour l'instant copiée sur les noeuds du système réparti, avec l'agent de mise à jour.

Le chapitre présentera donc dans un premier temps l'implantation de l'agent de mise à jour, et notamment l'organisation des classes d'objet qui le composent. Ensuite, il abordera l'implantation des langages de politique et de contrainte. Plus spécifiquement, nous étudierons les DTD qui ont été écrites pour représenter les politiques. Enfin, nous étudierons le mécanisme de traduction de la politique en langage neutre vers les langages de configuration de SELINUX et grsecurity.

L'implantation a également été présentée dans les articles [Briffaut *et al.* 2006c, Blanc *et al.* 2006].

7.1 Implantation de l'agent de mise à jour

L'agent de mise à jour a été présenté en partie 4.1.3, et ses principes algorithmiques de fonctionnement ont été exposés en partie 4.3. Suivant ce qui a été défini dans ces parties, nous avons implanté l'agent en Python, un langage de programmation objet. Avec ce langage, l'implantation est rapide, car

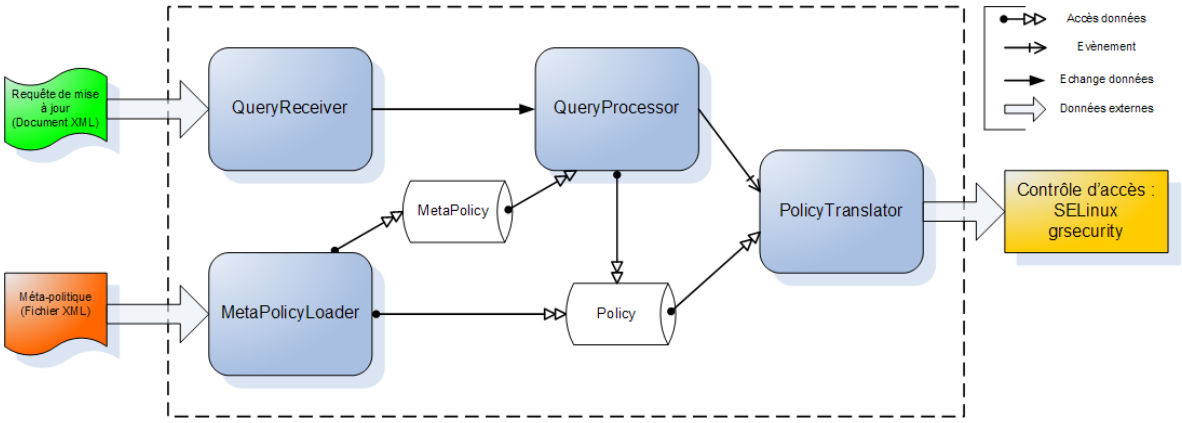

FIG. 7.1 – Structure interne de l'agent.

il s'agit d'un langage interprété (on économise l'étape de compilation des programmes), qui intègre nativement la manipulation de nombreux types de données (notamment les chaînes de caractères), et qui dispose de plusieurs classes de manipulation de documents XML. Par rapport à un langage compilé, l'intérêt majeur est que l'on a pas besoin de gérer tout ce qui concerne les allocations de mémoire.

La figure 7.1 expose la structure interne de l'agent, en particulier son découpage en classes et en structures de données. On y retrouve les fonctionnalités décrites en partie 4.3 :

MetaPolicyLoader : cette classe effectue le chargement de la Méta-Politique $\mathcal{MP}_{\mathcal{AC}}$ lors du démarrage de l'agent. Cette politique est présente sur le nœud où l'agent est exécuté, dans un fichier au format XML (ce format sera présenté en partie 7.2.2).

QueryReceiver : cette classe reçoit les requêtes de mise à jour produites par les autres programmes présents sur le nœud. Ces requêtes sont présentées sous la forme d'un message contenant un document XML (format présenté en partie 7.2.3).

QueryProcessor : classe responsable du traitement des requêtes de mise à jour reçues par le QueryReceiver. Il compare le contenu des requêtes à la politique de modification $\mathcal{MR}_{\mathcal{AC}}$, et les répercute dans la politique locale $\mathcal{LR}_{\mathcal{AC}}$, conservée en mémoire par l'agent dans un format XML présenté en partie 7.2.1.

PolicyTranslator : il s'agit du traducteur de politique neutre, qui charge la politique $\mathcal{LR}_{\mathcal{AC}}$ et la projette en une politique $\mathcal{TR}_{\mathcal{AC}}$, adaptée au mécanisme de sécurité cible (SELINUX, grsecurity). Les méthodes de traduction seront détaillées en section 7.3.

Les sous-sections suivantes détaillent les différentes classes qui composent l'agent.

7.1.1 Chargement de la Méta-Politique

Les fonctions de chargement de la Méta-Politique sont implantées dans la classe MetaPolicyLoader. La figure 7.2 détaille les interactions de cette classe au sein de l'agent.

La première fonction est le chargement de la Méta-Politique $\mathcal{MP}_{\mathcal{AC}}$. Elle est chargée par la méthode `readMetaPolicy`, à partir d'un fichier au format décrit par la DTD `meta.dtd` (cf. partie 7.2.2). Le chemin de ce fichier est enregistré dans un fichier de configuration lu au démarrage de l'agent.

A partir de ce fichier, la classe construit un arbre DOM en mémoire. DOM est une norme de représentation d'un fichier XML sous forme d'objets formant une structure d'arbre. Le langage Python

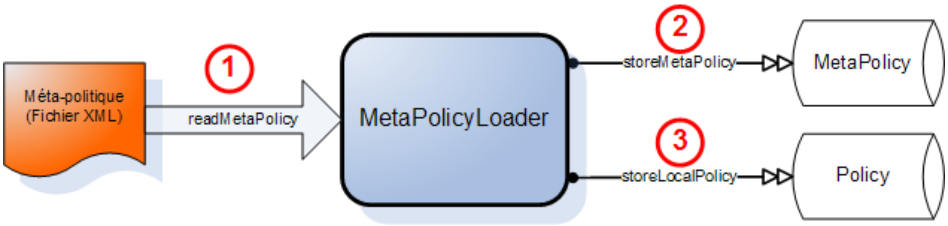

FIG. 7.2 – Interactions avec la classe MetaPolicyLoader.

étant un langage de programmation objet, la représentation DOM sera aisément exploitable. La méthode `storeMetaPolicy` parcourt le fichier chargé par la méthode précédente, et construit la représentation DOM du fichier de Méta-Politique, et enregistre cette représentation dans une variable `MetaPolicy`.

Enfin, à partir la représentation DOM de la Méta-Politique, la méthode `storeLocalPolicy` extrait la partie $\mathcal{IR}_{\mathcal{AC}}$, et la copie dans la variable `Policy`. Cette variable contient en permanence la version courante de la politique locale $\mathcal{LR}_{\mathcal{AC}}$.

7.1.2 Réception des requêtes de mise à jour

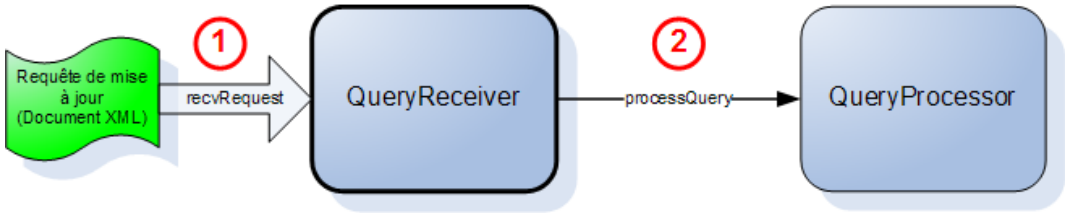

FIG. 7.3 – Interactions avec la classe QueryReceiver.

La réception des requêtes de mise à jour est effectuée par la classe QueryReceiver. Les interactions de cette classe avec les autres parties de l'agent sont détaillées dans la figure 7.3.

La méthode `recvRequest` effectue la réception d'une requête. Deux versions ont été implantées actuellement :

- Le dépôt de requête dans un fichier XML. La méthode surveille alors le contenu d'un dossier spécifique, et les programmes souhaitant faire des mises à jour de $\mathcal{LR}_{\mathcal{AC}}$ y déposent un fichier. Lorsqu'un fichier est présent, la méthode renvoie le contenu du fichier.
- La réception d'une requête par socket de type Unix. Il s'agit de sockets accessibles uniquement par les processus exécutés localement. Un tel socket est représenté sous forme d'un fichier, par exemple `/dev/agent`, et un programme souhaitant faire une demande de mise à jour se connecte à ce fichier. Lorsqu'une requête est reçue par la socket, la méthode en renvoie le contenu.

Dès qu'une requête est reçue, la classe fait appel à la méthode `processQuery` de la classe QueryProcessor, en passant en paramètre le tampon mémoire où la requête a été enregistrée.

7.1.3 Traitement des requêtes de mise à jour

La classe responsable du traitement des mises à jour est la classe QueryProcessor. Les interactions de cette classe sont données par la figure 7.4.

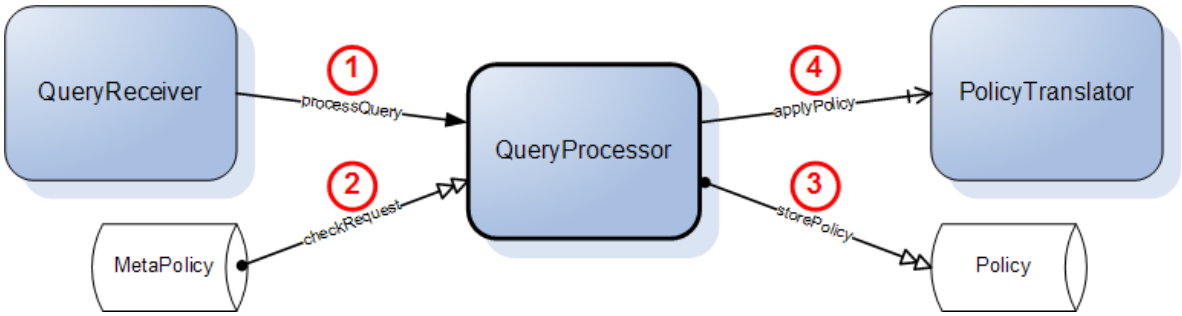

FIG. 7.4 – Interactions de la classe QueryProcessor.

La classe reçoit des requêtes de mise à jour à valider via la méthode `processQuery`, qui a en paramètre un tampon mémoire contenant une requête. L'appel de cette méthode est suivi par la vérification de la légitimité de la requête dans la politique $\mathcal{MR}_{\mathcal{AC}}$, via la méthode `checkRequest` (cf. paragraphe suivant). Elle renvoie un booléen indiquant si la requête doit être traitée, ou si elle doit être refusée.

Les requêtes sont reçues dans un format XML donné par la DTD `update.dtd`, explicité dans la partie 7.2.3. Ce format est très proche de celui de la politique de modification. Le principe de vérification de la validité des requêtes est le suivant :

- La classe QueryReceiver détermine le contexte de sécurité du processus qui a produit la requête, à partir des informations disponibles localement. Il s'agit soit du contexte du fichier, dans le cas où la requête est transmise par un fichier, soit du contexte du processus connecté au socket dans le cas d'une réception par socket Unix.
- La méthode `checkRequest` effectue une requête XPath pour charger toutes les règles de modification correspondant au contexte de sécurité calculé précédemment, et à l'action demandée (`{add, mod, del}`).
- Parmi les règles de modification trouvées, la méthode `checkRequest` recherche des expressions régulières qui correspondent à la modification demandée.
- Si une telle règle est trouvée, alors la requête de modification est valide. Dans le cas contraire, la requête est invalide et refusée.

On effectue au pire des cas deux parcours complets de la politique de modification. On a donc une complexité linéaire par rapport à la taille de la politique de modification.

Si la requête est valide, la méthode `storePolicy` effectue la modification demandée dans la politique $\mathcal{LR}_{\mathcal{AC}}$. Si la requête n'est pas valide, alors un évènement est enregistré, indiquant la réception d'une requête invalide.

Enfin, le chargement de la version mise à jour de la politique locale $\mathcal{LR}_{\mathcal{AC}}$ est effectué par la classe PolicyTranslator. Un appel à la méthode `applyPolicy` de cette classe est fait pour indiquer qu'une nouvelle version de $\mathcal{LR}_{\mathcal{AC}}$ est disponible dans la variable `Policy`.

7.1.4 Traduction vers le mécanisme de contrôle d'accès

La classe PolicyTranslator effectue la traduction de la politique locale de protection $\mathcal{LR}_{\mathcal{AC}}$ en une politique $\mathcal{TR}_{\mathcal{AC}}$, et la charge dans le mécanisme de contrôle d'accès local (SELINUX, grsecurity). La figure 7.5 détaille les interactions de cette classe dans l'agent de mise à jour.

Les appels à la méthode `applyPolicy` indiquent à la classe PolicyTranslator que la politique $\mathcal{LR}_{\mathcal{AC}}$ a été modifiée, et doit être rechargée dans le mécanisme de contrôle d'accès local. Ainsi, c'est

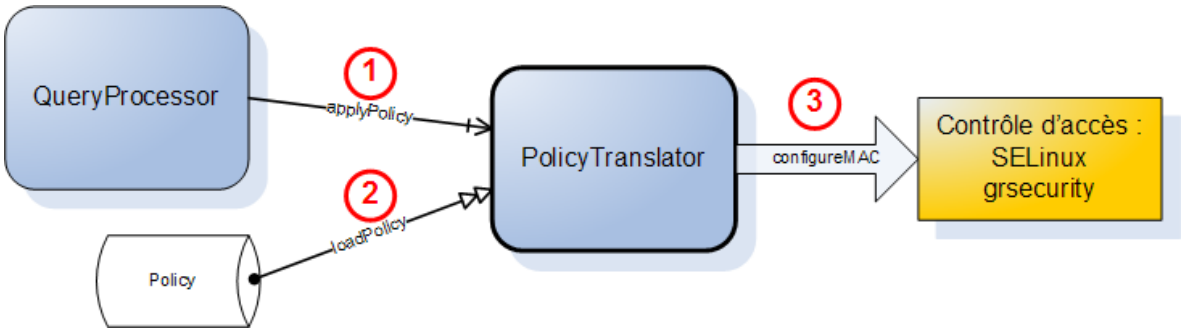

FIG. 7.5 – Interactions avec la classe PolicyTranslator.

normalement la classe QueryProcessor qui appelle cette méthode, mais elle est également appelée lors du démarrage de l'agent, lorsque la politique $\mathcal{TR}_{\mathcal{AC}}$ vient d'être copiée vers $\mathcal{LR}_{\mathcal{AC}}$.

La méthode `loadPolicy` est alors utilisée pour lire le contenu de la variable `Policy`. Lors du parcours de la représentation DOM de la politique $\mathcal{LR}_{\mathcal{AC}}$, la traduction est effectuée vers la politique $\mathcal{TR}_{\mathcal{AC}}$, en fonction du mécanisme de contrôle d'accès cible.

Enfin, la politique $\mathcal{TR}_{\mathcal{AC}}$ est transmise au mécanisme cible par la méthode `configureMAC`. Cette transmission dépend du mécanisme, par exemple pour SELINUX, la politique $\mathcal{TR}_{\mathcal{AC}}$ est écrite sur le disque dur, dans le fichier `/etc/selinux/policy.conf`, puis compilé en un fichier de politique binaire. Ce fichier binaire est ensuite chargé par la commande `load_policy`, fournie avec SELINUX.

7.2 Implantation des langages de politique

Les deux langages de politiques définis dans le cadre de cette thèse ont été formalisés dans le chapitre 5, le langage neutre de politique dans la partie 5.2.2.3 et le langage d'expression de contraintes dans la partie 5.2.2.4. Un exemple d'utilisation pratique de ces langages a été fourni en partie 5.2.4.

A partir de la définition formelle de ces langages, nous avons réalisé une implantation dans un format XML, via la création de DTD spécifiques. L'intérêt de XML est de pouvoir définir des formats de document indépendants des systèmes d'exploitation et des plates-formes où ces fichiers sont utilisés. Trois types de documents ont été définis : un pour le langage neutre de politique, un pour le langage de contraintes, et un pour les requêtes de mise à jour.

7.2.1 Format XML pour le langage neutre de politique

La DTD pour le format XML de la politique locale de protection $\mathcal{LR}_{\mathcal{AC}}$ est illustrée par la figure 7.6. Le listing complet est en annexe A.

Sur la figure 7.6, on trouve l'élément racine du format XML pour la politique neutre, `policy`. En-dessous, l'élément fils est `access_rule`, avec une relation de type `*` qui indique que dans le document XML, le nombre d'éléments `access_rule` n'est pas limité.

L'élément `access_rule` représente en fait un vecteur d'interaction. Le format de ces vecteurs a été défini en partie 5.2.2.2, par l'équation suivante (cf. équation 5.14) :

$$\forall iv \in IV, iv = (sc_s \in SC_S, sc_o \in SC_O, is \subset IS)$$

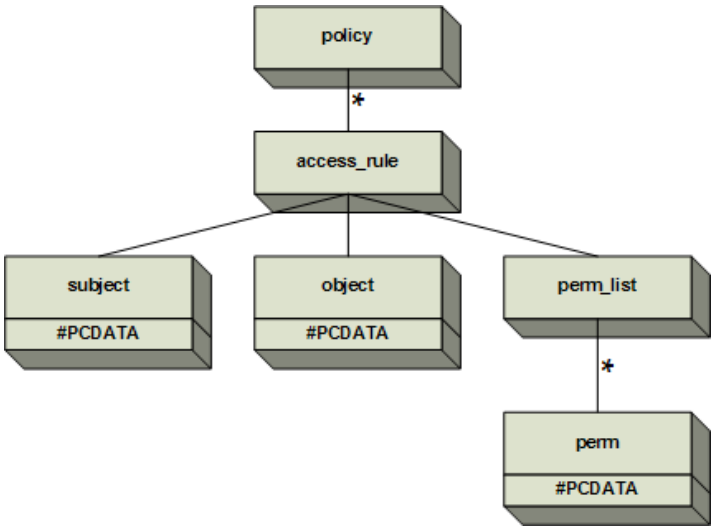

FIG. 7.6 – DTD pour la politique locale $\mathcal{LR}_{\mathcal{AC}}$

Dans la DTD, on trouve les trois éléments suivants : `subject` désigne le contexte de sécurité sujet sc_s, `object` le contexte de sécurité objet sc_o, et `perm_list` représente is. Ce dernier est un ensemble d'opérations élémentaires, ce qui se traduit dans la DTD par l'élément fils `perm`, qui est lié à `perm_list` par une relation `*`. Les éléments `perm` peuvent contenir eux-mêmes une liste de droits d'accès du type `{file : read, write}`. L'intérêt d'avoir plusieurs éléments `perm` est de pouvoir enregistrer les permissions d'accès concernant différentes classes d'objet, comme dans la première règle de l'exemple du listing 7.1.

Le listing 7.1 reprend l'exemple 5.9 de la partie 5.2.4.2, cette fois écrit dans le format XML. Les lignes 4 à 11 représentent un vecteur d'interaction, avec le contexte de sécurité sujet `(user_u, user_r, user_shell_t)` dans la balise `subject`, de même pour le contexte objet dans la balise `object`, et deux ensembles de permissions dans des balises `perm`. Les lignes 12 à 18, puis 19 à 25, représentent deux autres vecteurs d'interaction.

7.2.2 Format XML pour le fichier de Méta-Politique

La DTD pour le format XML du fichier contenant la Méta-Politique est illustrée par la figure 7.7. Le listing complet est en annexe A.

Sur la figure 7.7, on trouve la racine du document XML, l'élément nommé `metapolicy`. En-dessous, on trouve deux branches principales. Elles correspondent au deux politiques contenues dans $\mathcal{MP}_{\mathcal{AC}}$, soit 1) la politique de modification $\mathcal{MR}_{\mathcal{AC}}$, et 2) la politique de protection initiale $\mathcal{IR}_{\mathcal{AC}}$. $\mathcal{MR}_{\mathcal{AC}}$ est contenue dans le sous-arbre formé par l'élément `update_policy`, tandis que $\mathcal{IR}_{\mathcal{AC}}$ correspond au sous-arbre de l'élément `policy`.

7.2.2.1 Règles de modification

Les règles contenues dans $\mathcal{MR}_{\mathcal{AC}}$ ont été formalisées dans la partie 5.1.4. Ces règles présentent deux formes, la première concerne l'administration des contextes de sécurité, et la seconde concerne l'administration des vecteurs d'interaction. Pour rappel, voici les équations décrivant le format de ces règles (reprises des équations 5.11 et 5.14) :

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <!DOCTYPE policy SYSTEM 'meta.dtd'>
3  <policy>
4     <access_rule>
5        <subject>(user_u, user_r, user_shell_t)</subject>
6        <object>(system_u, object_r, opt_apps_lambda_t)</object>
7        <perm_list>
8           <perm>{file : execute}</perm>
9           <perm>{process : transition}</perm>
10       </perm_list>
11    </access_rule>
12    <access_rule>
13       <subject>(admin_u, admin_r, admin_shell_t)</subject>
14       <object>(system_u, object_r, opt_apps_lambda_config_t)</object>
15       <perm_list>
16          <perm>{file : read, write}</perm>
17       </perm_list>
18    </access_rule>
19    <access_rule>
20       <subject>(user_u, user_r, opt_apps_lambda_t)</subject>
21       <object>(user_u, user_r, opt_apps_lambda_data_t)</object>
22       <perm_list>
23          <perm>{file : read, write}</perm>
24       </perm_list>
25    </access_rule>
26 </policy>
```

Listing 7.1 – Politique neutre en XML.

$$enable * * * SC(sc_{requester}, pattern_{SC}, pattern_{attribut})$$
$$enable * * * IV(sc_{requester}, (pattern_S, pattern_O, pattern_{IS}))$$
$$\text{avec } * ** := Add|Mod|Del$$

Dans la DTD, on retrouve ces deux formes de règles dans deux sous-arborescences séparées. Les règles d'administration des contextes de sécurité sont situées sous l'élément `update_labels_rule_list`, et celles concernant les vecteurs d'interaction sont sous l'élément `update_perms_rule_list`. Ces deux éléments représentent en fait des listes de règles. Les règles elles-mêmes sont représentées par les éléments `update_labels_rule` et `update_perms_rule`. Les relations liant l'élément `update_labels_rule_list` à l'élément `update_labels_rule` et l'élément `update_perms_rule_list` à l'élément `update_perms_rule` sont des `*`, signifiant que l'élément suffixé par `_rule_list` peut contenir zéro ou plusieurs éléments suffixés par `_rule`.

Dans les sous-arbres des éléments `update_labels_rule` et `update_perms_rule`, on trouve deux éléments fils communs : `action` qui indique quelle est l'action de modification autorisée, et contient un des mots-clés `{add, mod, del}`; et `label`, qui contient le contexte de sécurité autorisé à effectuer l'action de modification, nommé $sc_{requester}$ dans l'équation donnée plus haut. Ceci est cohérent avec la formalisation de ces règles, qui spécifiait ces éléments communs.

Le dernier élément fils des règles de modification est spécifique à l'administration des contextes de sécurité ou des vecteurs d'interaction. Dans le cas de `update_labels_rule`, on trouve un élément

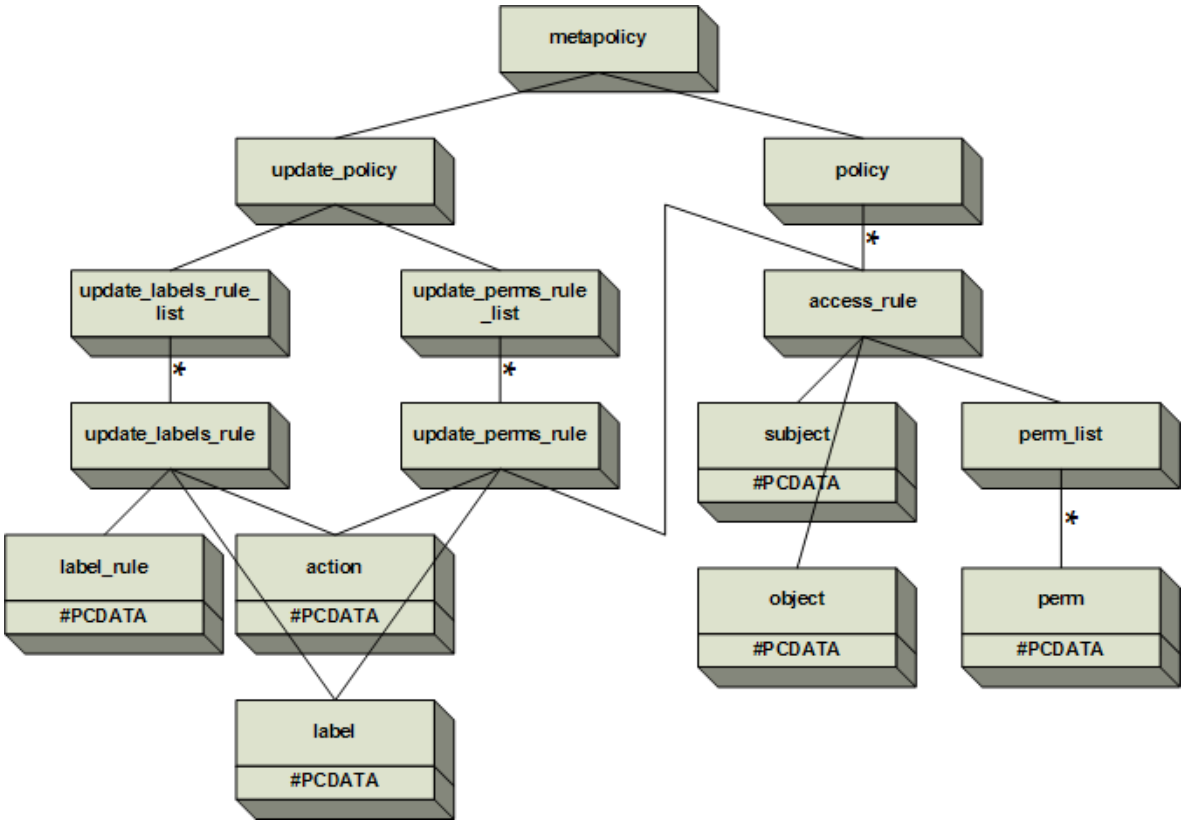

FIG. 7.7 – DTD pour la méta-politique $\mathcal{MP}_{\mathcal{AC}}$.

fils nommé `label_rule`, contenant l'expression régulière nommée $pattern_{SC}$ dans l'équation. Le paramètre optionnel $pattern_{attribut}$ n'a pas été implanté pour le moment. Dans le cas de l'élément `update_perms_rule`, on a un élément fils nommé `access_rule`, qui contient les expressions régulières pour la mise à jour des vecteurs d'interaction.

Cet élément est commun avec l'arborescence contenant la politique initiale $\mathcal{IR}_{\mathcal{AC}}$. En effet, la structure des vecteurs d'interaction est la même pour $\mathcal{MR}_{\mathcal{AC}}$ et de $\mathcal{IR}_{\mathcal{AC}}$. Toutefois, là où $\mathcal{MR}_{\mathcal{AC}}$ utilise des expressions régulières, $\mathcal{IR}_{\mathcal{AC}}$ contiendra des contextes de sécurité et listes de permissions simples (cf. partie 7.2.1).

7.2.2.2 Politique initiale de protection

La politique initiale de protection $\mathcal{IR}_{\mathcal{AC}}$ est écrite dans le langage neutre de politique. Il a été formalisé dans la partie 5.1.3. $\mathcal{IR}_{\mathcal{AC}}$ contient en fait des vecteurs d'interaction, représentant l'ensemble des accès autorisés sur le système d'exploitation.

La politique $\mathcal{IR}_{\mathcal{AC}}$ est utilisée pour initialiser la politique locale de protection $\mathcal{LR}_{\mathcal{AC}}$, par simple copie des règles. C'est pourquoi ces deux politiques ont le même format, qui est décrit dans la partie 7.2.1.

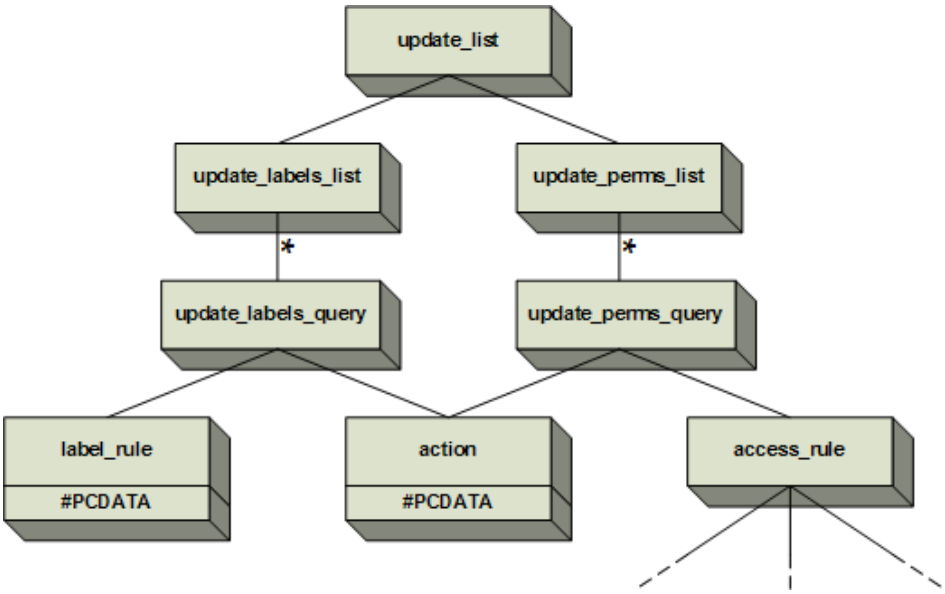

FIG. 7.8 – Requête de mise à jour en XML.

7.2.2.3 Exemple de Méta-Politique en XML

Le listing B.1 de l'annexe B présente une Méta-Politique au format XML correspondant à la DTD décrite dans les parties précédentes. Il présente une version en XML du listing 5.7.

Les lignes 4 à 52 contiennent la politique de modification $\mathcal{MR}_{\mathcal{AC}}$. On trouve d'abord les règles d'administration des contextes de sécurité, avec une règle du type $enableAddSC$ (lignes 6 à 12). Ensuite on a les règles d'administration des vecteurs d'interaction, avec une règle du type $enableAddIV$ (lignes 15 à 26). Les lignes 31 à 32 contiennent la politique $\mathcal{IR}_{\mathcal{AC}}$. Sur cet exemple, elle est vide, pour des raisons de place, mais le format est le même que l'élément `policy` du listing 7.1.

7.2.3 Format XML pour les requêtes de mise à jour

La DTD pour le format XML des requêtes de mise à jour de la politique locale $\mathcal{LR}_{\mathcal{AC}}$ est illustrée par la figure 7.8. Le listing complet est en annexe A.

Sur la figure 7.8, on trouve l'élément racine `update_list`. L'arborescence définie par cette DTD est à rapprocher de la sous-arborescence de l'élément `update_policy` de la DTD pour la Méta-Politique (cf. figure 7.7). La différence est que, dans les requêtes de mise à jour, on ne trouve que l'action (mots-clés `{add, mod, del}`), et pas de contexte autorisé à faire la mise à jour. En effet, lors de la réception d'une requête de mise à jour, c'est l'agent lui-même qui est chargé de déterminer quel contexte lui a fait cette demande, afin de pouvoir vérifier dans la politique $\mathcal{MR}_{\mathcal{AC}}$ que la requête est légitime.

En dehors du contexte effectuant la requête de mise à jour, l'arborescence de l'élément `update_list` est la même que pour l'élément `update_policy`, i.e. elle contient une liste de requêtes de modification de contextes, éventuellement vide, et une liste de requêtes de modification de vecteurs d'interaction, éventuellement vide. Une seule requête de mise à jour peut donc contenir plusieurs actions de modification, concernant à la fois les contextes et les vecteurs.

Le listing 7.2 présente un exemple de requête de mise à jour au format XML, établi à partir du listing 5.8. Les lignes 4 à 11 contiennent une liste de requêtes de modification des contextes de sécu-

rité, avec ici une demande d'ajout du contexte `(user_u, user_r, opt_apps_lambda_t)`. Ensuite, les lignes 12 à 13 contiennent la liste des requêtes de modifications de vecteurs d'interaction, avec une demande d'ajout, qui concerne la permission pour l'utilisateur d'exécuter l'application `lambda`.

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <!DOCTYPE update_list SYSTEM 'update.dtd'>
3  <update_list>
4     <update_labels_list>
5        <update_labels_query>
6           <action>add</action>
7           <label_rule>
8              <label>(user_u, user_r, opt_apps_lambda_t)</label>
9           </label_rule>
10       </update_labels_query>
11    </update_labels_list>
12    <update_perms_list>
13       <update_perms_query>
14          <action>add</action>
15          <access_rule>
16             <subject>(user_u, user_r, user_shell_t)</subject>
17             <object>(system_u, object_r, opt_apps_lambda_exec_t)</object>
18             <perm_list>
19                <perm>{file : execute}</perm>
20             </perm_list>
21          </access_rule>
22       </update_perms_query>
23    </update_perms_list>
24 </update_list>
```

Listing 7.2 – Requête de mise à jour en XML.

7.3 Implantation des modules de traduction

La section 7.1 a présenté la structure interne de l'agent. Parmi les classes décrites, PolicyTranslator est responsable de la traduction de la politique locale $\mathcal{LR}_{\mathcal{AC}}$ vers le mécanisme de contrôle d'accès cible (politique $\mathcal{TR}_{\mathcal{AC}}$). Cette politique est maintenue en mémoire par l'agent, dans un format décrit par la DTD `policy.dtd`, présentée en partie 7.2.1.

Comme cela avait été mentionné dans les parties 4.1.6 et 5.1.5, cette traduction est spécifique au mécanisme cible. C'est d'ailleurs ainsi que notre architecture supporte une application à des systèmes répartis hétérogènes. La classe PolicyTranslator possède donc une partie spécifique à chaque mécanisme de contrôle d'accès, qui est chargée en fonction du noeud sur lequel l'agent est exécuté.

La traduction spécifique à SELINUX et grsecurity est réalisée par l'implantation de deux classes héritant de PolicyTranslator : les classes SELinuxTranslator et GrsecurityTranslator. Les parties suivantes détaillent l'implantation de ces deux classes, suivant deux axes : d'abord la traduction des contextes de sécurité, puis la traduction des vecteurs d'interaction.

7.3.1 Traduction de la politique neutre pour SELinux

La classe SELinuxTranslator implante le mécanisme de traduction de la politique locale en langage neutre $\mathcal{LR}_{\mathcal{AC}}$, vers le langage de configuration de SELINUX, pour un noeud intégrant ce mécanisme. Elle est donc chargée dans l'agent si c'est le cas. Elle hérité de la classe PolicyTranslator, et présente donc les mêmes méthodes, décrites en partie 7.1.4. Les paragraphes suivants présente, premièrement, les méthodes de traduction des contextes de sécurité, et deuxièmement, les méthodes de traduction des vecteurs d'interaction.

Contextes de sécurité Le format des contextes de sécurité du langage neutre est très proche de ceux de SELINUX (cf. partie 5.2.2.1). La traduction est donc une simple copie.

Reprenons l'exemple du chapitre 5, partie 5.2.4.3. Le listing 5.9 contient la politique en langage neutre, et le listing 5.10 la configuration correspondante pour SELINUX.

Le début de la configuration de SELINUX contient la déclaration des rôles et types. Ces règles sont calculées en effectuant la liste des contextes présents dans la politique neutre, et en déterminant à quels types chaque rôle est associé, afin de produire les règles associant rôles et types. Par exemple, la règle suivante autorise le rôle `admin_r` à transiter vers le contexte sujet de type `admin_t` :

```
role admin_r types admin_t;
```

Pour générer ces règles, il suffit de récupérer toutes les associations rôle / type, à partie des contextes de sécurité présent dans la politique $\mathcal{LR}_{\mathcal{AC}}$.

Concernant l'association entre contextes de sécurité objet et chemins du système de fichier, elle repose dans l'implantation actuelle sur un fichier annexe nommé `file_contexts`, fourni par SELINUX. Le format de ce fichier consiste en deux champs par ligne, le premier champ contient une expression régulière sur les chemins du système de fichiers, et le second champ est le contexte associé, avec éventuellement un paramètre (-d) indiquant que ce contexte ne doit être associé qu'à des dossiers. Le listing 7.3 présente un extrait de ce fichier. Il est normalement utilisé par SELINUX pour associer des contextes de sécurité à tous les fichiers. Par exemple, la première ligne indique que le label par défaut pour tous les fichiers est `system_u:object_r:default_t`. Il s'agit d'un label par défaut, car si une autre ligne correspond au fichier en cours d'association, c'est cette ligne qui sera utilisée. La stratégie d'application est que la dernière expression régulière correspondant à un fichier donné est utilisée pour définir le contexte de celui-ci.

```
/.*                 system_u:object_r:default_t
/               -d system_u:object_r:root_t
/home           -d system_u:object_r:home_root_t
/home/[^/]+     -d system_u:object_r:user_home_dir_t
/home/[^/]+/.+      system_u:object_r:user_home_t
[...]
```

Listing 7.3 – Extrait du fichier `file_contexts`.

Vecteurs d'interaction Du point de vue des vecteurs d'interaction, le langage neutre est également proche de celui de SELINUX. Notamment, l'expression des droits d'accès est très proche : `{file : read, write}` dans le langage neutre donne `:file { read write }` pour SELINUX.

La traduction se fait donc simplement en précisant le mot `allow` en début de ligne, puis les contextes sujet et objet, et enfin les droits d'accès.

Concernant les règles de transition de contexte (cf. partie 5.2.1.2), la convention retenue est la suivante : lorsque qu'une règle autorise un droit d'accès `{process : transition}`, alors c'est qu'il y a une règle de transition. Le contexte objet de la règle est alors un contexte de processus. On calcule alors automatiquement le contexte de sécurité du fichier exécutable associé à ce contexte de processus, en remplaçant le suffixe`_t` par `_exec_t`, et on produit une règle `type_transition`.

Considérons par exemple la règle en langage neutre (cf. listing 5.9) :

```
(user_shell_t, opt_apps_lambda_t, {process : transition})
```

Comme expliqué dans le paragraphe précédent, `opt_apps_lambda_t`, le contexte objet de cette règle, est en fait un contexte de processus. La convention de nommage de SELINUX associe le contexte `opt_apps_lambda_exec_t` au fichier exécutable correspondant à ce processus. On génère donc la règle de transition suivante :

```
type_transition user_shell_t opt_apps_lambda_exec_t:process opt_apps_lambda_t;
```

7.3.2 Traduction de la politique neutre pour grsecurity

La traduction de la politique $\mathcal{LR}_{\mathcal{AC}}$ vers la configuration de grsecurity est effectuée par la classe GrsecurityTranslator. Sur les noeuds présentant ce mécanisme de contrôle d'accès, cette classe est chargée comme instance de la classe PolicyTranslator dont elle hérite. Nous allons détailler ci-après les méthodes de traduction des contextes de sécurité, puis des vecteurs d'interaction.

Contextes de sécurité grsecurity n'utilise pas de contextes de sécurité au même sens que SELINUX, ou que notre Méta-Politique. Plutôt que d'associer aux sujets et objets des contextes, grsecurity les désigne par leur chemin complet dans le système de fichier. Par exemple, alors que SELINUX désigne le processus du serveur Apache par le contexte `(system_u, system_r, apache_httpd_t)`, grsecurity le désigne simplement par le chemin du fichier exécutable d'Apache, soit `/usr/sbin/httpd`. L'avantage pour grsecurity est que le contrôle d'accès ne repose pas sur des informations supplémentaires. Cependant, comme les contextes de sécurité forment des classes d'équivalence dans SELINUX, il faudra davantage de règles pour réécrire des politiques de contrôle d'accès aussi complètes dans grsecurity.

Ainsi, nous avons besoin d'un mécanisme de traduction des contextes de sécurité vers les chemins correspondants, aussi bien pour les sujets que les objets. Pour cela nous allons utiliser deux fichiers annexes.

Contextes objet Le premier fichier est similaire à celui présenté dans la partie 7.3.1, il s'agit d'un fichier du type `file_contexts` (cf. listing 7.3). Ce fichier donne la correspondance entre contextes de sécurité objet et chemins. Toutefois, grsecurity ne supporte pas les expressions régulières pour les chemins, et désigne plutôt des dossiers, dont tous les fichiers et sous-dossiers héritent des permissions. Voici donc les étapes de traduction d'un contexte de sécurité objet en chemins pour grsecurity :

- On recherche toutes les lignes du fichier `file_contexts` dont le deuxième champ est égal au contexte de sécurité recherché.
- Pour chacune de ces lignes, on tronque le premier champ au premier caractère spécial rencontré (par exemple, `/var(/.*)?` devient `/var`), ce qui nous donne une liste de chemins sans expressions régulières.

– Enfin, on génère une ligne pour chaque contexte de sécurité objet, avec la traduction des permissions d'accès (e.g. `{file : read}` correspond au mode d'accès `r` de grsecurity).

Contextes sujet Concernant les sujets, le mécanisme précédent ne suffit pas. Il nous faut un mécanisme plus précis, pour traduire un contexte de sécurité sujet vers le chemin du fichier exécutable correspondant. C'est pourquoi nous avons besoin d'un second fichier complémentaire, nommé `sc_contexts`, qui fait précisément la traduction des contextes de processus (e.g. `apache_httpd_t`) vers le chemin correspondant (e.g. `/usr/sbin/httpd`). Le format de ce fichier est donné par le listing 7.4. Sur chaque ligne, on trouve un chemin d'exécutable, puis le contexte sujet correspondant.

```
/usr/sbin/httpd      apache_httpd_t
/usr/sbin/chroot     chroot_t
/usr/bin/vipw        admin_passwd_t
/usr/bin/vigr        admin_passwd_t
[...]
```

Listing 7.4 – Extrait du fichier `sc_contexts`.

Vecteurs d'interaction Le format des règles d'accès de grsecurity est simple (cf. listing 5.4) : on trouve une première ligne commençant par `subject`, puis chaque ligne suivante ne commençant pas par `subject` correspond à la désignation d'un objet avec des permissions d'accès associées. Enfin, cet ensemble de règles d'accès est rattaché à un rôle, spécifié par une ligne introduite par le mot-clé `role`. Un rôle est actif jusqu'à la prochaine ligne commençant par `role`.

Pour la traduction des vecteurs d'accès, on va donc procéder comme suit :

– On classe les vecteurs d'accès par le rôle contenu dans le contexte sujet, puis par le type du contexte sujet.
– Pour chaque rôle identifié, on génère la ligne commençant par `role`, puis pour chaque type, une ligne `subject`. Ensuite, pour chaque règle d'accès correspondant à ce contexte sujet (rôle et type), on génère une ligne avec la traduction du contexte objet, et des permissions d'accès.
– Lorsqu'une ligne a été générée pour chaque contexte objet, on passe au type sujet suivant. Lorsque tous les types sujet ont été traités, on passe au role sujet suivant. Lorsque tous les rôles ont été traités, toutes les règles d'accès ont été traduites.

Enfin, la configuration pour grsecurity est écrite dans le fichier `/etc/grsec/policy`, et chargée à l'aide de la commande `gradm`.

7.4 Conclusion

Dans ce chapitre, nous avons exposé l'implantation de notre architecture. Nous avons d'abord présenté les classes qui composent l'agent de mise à jour et les interactions entre ces classes. Nous avons ainsi reproduit les principes algorithmiques décrits en section 4.3. Ensuite nous avons décrit les formats XML de représentation des politiques de protection et de modification, et des requêtes de mise à jour. Enfin, nous avons décrit les méthodes de traduction du langage neutre de politique de protection vers les langages de configuration de SELINUX et grsecurity.

Chapitre 8

Conclusion et Perspectives

L'objet de cette thèse est la définition d'une nouvelle architecture pour l'administration du contrôle d'accès dans les systèmes répartis. La solution présentée combine, d'une part, les modèles MAC / RBAC pour un contrôle d'accès renforcé, et d'autre part, un nouveau concept de *méta-politique* autorisant la *décentralisation* de l'administration des politiques de sécurité.

Tout d'abord, dans le chapitre 2, nous avons établi une liste de modèles de contrôle d'accès suivant les familles DAC, MAC et RBAC, d'implantations système (Medusa DS9, LIDS, RSBAC, SELINUX, grsecurity), et de modèle d'expression de politiques multi-domaines, multi-politiques et méta-politiques. Dans le chapitre 3, nous avons évalué ces modèles d'expression de politiques au regard de nos *objectifs* : décentralisation de l'administration, intégration de systèmes d'exploitation de confiance (TOS), forte tolérance aux pannes, bonnes performances et support de méthodes de vérification d'absence de flux d'information illégaux. Nous avons montré que ces modèles ne répondent pas à ces objectifs. C'est pourquoi nous avons défini un nouveau modèle de Méta-Politique, contenant deux politiques distinctes : d'une part une politique de contrôle d'accès, et d'autre part une politique contrôlant les évolutions de cette première politique.

Ensuite, le chapitre 4 a présenté notre nouvelle architecture de Méta-Politique : une architecture distribuée, reposant sur un *agent de mise à jour* assurant l'administration locale de la politique de contrôle d'accès, et l'évolution de celle-ci sous la contrainte de la politique de modification. La méta-politique autorise chaque noeud du système réparti à administrer indépendamment sa propre politique, et répond ainsi à l'objectif d'*administration décentralisée*. La politique de protection est écrite dans langage neutre, qui peut être traduit pour configurer différents mécanismes de MAC, répondant à l'objectif d'*intégration de système de confiance*. Sur chaque noeud, les modifications de la politique sont contrôlées par l'agent de mise à jour, sans nécessiter de connexion réseau, ce qui répond à l'objectif de *forte tolérance aux pannes*. L'architecture de Méta-Politique évite l'apparition de conflits, et intègre des mécanismes de contrôle d'accès optimisés, en réponse à l'objectif de *bonnes performances*.

Le chapitre 5 a formalisé notre modèle de méta-politique, en deux temps. Premièrement, un modèle général a été défini, supportant l'application à différents mécanismes de protection comme le contrôle d'accès et la détection d'intrusions. Les deux langages, correspondant aux deux niveaux de contrôle, ont été définis : un *langage neutre d'expression de règles d'accès* rend l'architecture indépendante des systèmes sur lesquels elle est déployée ; un *langage d'expression de contraintes de modification* autorise une évolution configurable et contrôlée de la politique de sécurité locale. Puis ce modèle général a été appliqué au contrôle d'accès, avec une nouvelle définition des langages d'expression de politiques et de contraintes.

A partir de la formalisation, le chapitre 6 a proposé une méthode de vérification des garanties de

sécurité dans la méta-politique. Il s'agit de certifier l'absence de flux d'information illégaux dans la méta-politique. Nous avons montré que, bien qu'ayant affaire à une politique dynamique, administrée de façon décentralisée, rend plus complexe la recherche des flux d'information, les contraintes imposées par la politique de modification garantissent effectivement les objectifs. Ainsi, un algorithme général de recherche de flux a été proposé répondant à notre objectif de *support de méthodes de vérification d'absence de flux d'information illégaux*.

Enfin, le chapitre 7 a proposé une implantation de notre architecture. Notamment, les langages de politiques et de contraintes ont fait l'objet de création de DTD pour leur représentation en XML. Les méthodes de traduction de la politique neutre vers SELINUX et grsecurity ont été détaillées.

Afin d'améliorer la pertinence de notre architecture, et d'étendre son champ d'applicabilité, des perspectives s'ouvrent suivant plusieurs axes de travail : intégration de mécanismes autres que SELINUX et grsecurity, utilisation de langages de contrôle d'accès standard comme XACML, preuve de notre algorithme de vérification d'existence de flux d'information illégaux, coopération avec des méthodes de détection d'intrusion et apprentissage de Méta-Politique.

Intégration de nouveaux mécanismes de contrôle d'accès

Pour l'instant, l'architecture n'est déployable que sur des systèmes d'exploitation Linux embarquant soit SELINUX soit grsecurity. Il serait pertinent d'étudier l'applicabilité de la solution à des systèmes comme RSBAC, ou même à d'autres systèmes d'exploitation comme FreeBSD qui intègre également des mécanismes MAC / RBAC.

Utilisation de langages standards

Pour notre architecture, nous avons défini deux nouveaux langages spécifiques. Cependant, au vu du nombre croissant de langages de politiques de sécurité, les chercheurs dans le domaines des politiques se posent la question d'une éventuelle convergence. Un des langages qui pourrait tirer son épingle du jeu est XACML [OASIS 2006]. En effet, il est publié par l'organisme de standardisation OASIS, il repose entièrement sur XML, et il dispose d'une implémentation faite par Sun Microsystems.

Un travail sur la compatibilité entre le langage neutre proposé dans cette thèse, et les spécifications de XACML serait intéressant. L'objectif pourrait être de repérer d'éventuels écueils dans notre langage, voire même de proposer une alternative à notre langage en XACML.

Preuve de l'algorithme de vérification

Concernant la vérification de l'existence de flux d'information illégaux dans la Méta-Politique, notre proposition d'algorithme (cf. chapitre 6) nous incite à faire la conjecture que le problème de vérification est décidable. Cependant, nous comptons établir une preuve formelle que tous les flux illégaux sont correctement calculés par notre algorithme.

Coopération avec des méthodes de détection d'intrusions

Nous avons présenté dans la partie 5.3 une méthode générale de coopération entre notre méta-politique de contrôle d'accès, et une méthode spécifique de détection d'intrusions. Ce travail sur la détection a été effectué par ailleurs, et présente d'ores et déjà des résultats intéressants. Il montre la complémentarité entre notre approche de contrôle d'accès, et une méthode de détection d'intrusions. De nombreuses directions de travail sont encore à exploiter dans la coopération entre le contrôle d'accès et différents mécanismes de détection d'intrusion.

Apprentissage de Méta-Politique

L'écriture de politiques reste un processus complexe et coûteux en temps. Des techniques d'apprentissage sont envisagées autour de notre approche Méta-Politique. Il existe déjà des méthodes d'apprentissage de politiques de contrôle d'accès, notamment pour SELINUX et grsecurity. Nous comptons développer un apprentissage pour le calcul automatique des règles de modification, à partir de l'observation d'un système réparti.

Chapitre 9

Bibliographie

[Abou El Kalam *et al.* 2003] Abou El Kalam, A., Baida, R. E., Balbiani, P., Benferhat, S., Cuppens, F., Deswarte, Y., Miège, A., Saurel, C. et Trouessin, G. (2003). Organization based access control. *In 4th IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY 2003)*, pages 120–131. IEEE Computer Society Press.

[Abou El Kalam *et al.* 2005a] Abou El Kalam, A., Briffaut, J., Toinard, C. et Blanc, M. (2005a). Intrusion detection and security policy framework for distributed environments. *In Proceedings of the 2005 International Symposium on Collaborative Technologies and Systems (CTS'05)*, pages 100–106, Saint Louis, USA.

[Abou El Kalam *et al.* 2005b] Abou El Kalam, A., Briffaut, J., Toinard, C., Blanc, M. et Oudot, L. (2005b). Multi-level intrusion detection system (MIDS). *In The 4th Conference on Security and Network Architectures (SAR'05)*, pages 145–155, Batz sur Mer, France.

[Abrams *et al.* 1990] Abrams, M. D., LaPadula, L. J., Eggers, K. W. et Olson, I. M. (1990). A generalized framework for access control : An informal description. *In The 13th National Computer Security Conference*, pages 135–143, Washington, D.C., USA.

[Alpers et Plansky 1994] Alpers, B. et Plansky, H. (1994). Domain and policy based management : Concepts and implementation architecture. *In The Fifth IFIP/IEEE International Workshop on Distributed Systems : Operations and Management (DSOM '94)*, Toulouse, France.

[Anderson 1972] Anderson, J. P. (1972). Computer security technology planning study. Technical Report ESD-TR-73-51, Air Force Electronic Systems Division, Bedford, MA.

[Avitabile 1998] Avitabile, M. (1998). An examination of requirements for metapolicies in policy-based management. Master's thesis, Munich University of Technology.

[Badger *et al.* 1995] Badger, L., Sterne, D. F., Sherman, D. L. et Walker, K. M. (1995). A domain and type enforcement UNIX prototype. *In Proceedings of the 5th USENIX UNIX Security Symposium*, pages 127–140, Salt Lake City, Utah, USA.

[Bell 1988] Bell, D. E. (1988). Concerning 'modeling' of computer security. *In Proceedings of the IEEE Symposium on Security and Privacy*, pages 8–13, Oakland, CA, USA. IEEE.

[Bell et La Padula 1973] Bell, D. E. et La Padula, L. J. (1973). Secure computer systems : Mathematical foundations and model. Technical Report M74-244, The MITRE Corporation, Bedford, MA.

[Belokosztolszki et Moody 2002] Belokosztolszki, A. et Moody, K. (2002). Meta-policies for distributed role-based access control systems. *In 3rd IEEE International Workshop on Policies for*

Distributed Systems and Networks (POLICY 2002), pages 106–115, Monterey, California, USA. IEEE Computer Society Press.

[Bertino *et al.* 1999] Bertino, E., Jajodia, S. et Samarati, P. (1999). A flexible authorization mechanism for relational data management systems. *ACM Transactions on Information Systems (TOIS)*, 17(2):101–140.

[Bertino *et al.* 1996] Bertino, E., Jojodia, S. et Samarati, P. (1996). Supporting multiple access control policies in database systems. *In IEEE Symposium on Security and Privacy*, pages 94–109, Oakland, Canada. IEEE Computer Society Press.

[Biba 1975] Biba, K. J. (1975). Integrity considerations for secure computer systems. Technical Report MTR-3153, The MITRE Corporation.

[Bidan et Issarny 1997] Bidan, C. et Issarny, V. (1997). A configuration-based environment for dealing with multiple security policies in open distributed systems. *In 2nd European Research Seminar on Advances in Distributed Systems*, pages 240–245, Zinal, Switzerland.

[Bidan et Issarny 1998] Bidan, C. et Issarny, V. (1998). Dealing with multi-policy security in large open distributed systems. *In The 5th European Symposium on Research in Computer Security ESORICS*, volume 1485 de *Lecture Notes in Computer Science*, pages 51–66, Louvain-la-Neuve, Belgium. Springer-Verlag.

[Blanc 2004] Blanc, M. (2004). Trusted Linux systems and application to cluster architecture. *In* Smari, W. W. et McQuay, W., éditeurs : *The 2004 International Symposium on Collaborative Technologies and Systems, Special Session on Security and Collaboration (CTS'04)*, volume 36 de *Simulation Series*, pages 29–34, San Diego, CA, USA. Western Multi Conferences 2004 (WMC'04), The Society for Modeling and Simulation International - SCS.

[Blanc *et al.* 2006] Blanc, M., Briffaut, J., Lalande, J.-F. et Toinard, C. (2006). Distributed control enabling consistent MAC policies and IDS based on a meta-policy approach. *In Proceedings of the Seventh IEEE International Workshop on Policies for Distributed Systems and Networks (Policy 2006)*, pages 153–156, London, Canada. IEEE Computer Society.

[Blanc *et al.* 2004a] Blanc, M., Clemente, P., Courtieu, P., Franche, S., Oudot, L., Toinard, C. et Vessiller, L. (2004a). Hardening large-scale networks security through a meta-policy framework. *In* Wysocki, B. J. et Wysocki, T. A., éditeurs : *Third Workshop on the Internet, Telecommunications and Signal Processing (WITSP'04)*, pages 132–137, Adelaide, Australia. DSP for Communication Systems.

[Blanc *et al.* 2005a] Blanc, M., Clemente, P., Courtieu, P., Franche, S., Oudot, L., Toinard, C. et Vessiller, L. (2005a). Amélioration de la sécurité des grands réseaux par une infrastructure de méta-politique. *In Colloque Francophone sur l'Ingénierie des Protocoles (CFIP'05)*, pages 517–530, Bordeaux, France. Lavoisier.

[Blanc *et al.* 2005b] Blanc, M., Courtieu, P. et Hains, G. (2005b). Mandatory access control on distributed systems : A metapolicy framework. *In The First Colloquium on Risk and Security of the Internet and Systems (CRiSIS 2005)*, pages 133–144, Bourges, France. Ecole Nationale Supérieure d'Ingénieurs de Bourges.

[Blanc *et al.* 2004b] Blanc, M., Courtieu, P., Hains, G., Oudot, L. et Toinard, C. (2004b). A novel approach for distributed updates of MAC policies using a meta-protection framework. *In* de Rennes, U., éditeur : *The 15th IEEE International Symposium on Software Reliability Engineering (ISSRE 2004) Supplementary Proceedings*, pages 29–30, Saint Malo, France. INRIA, Rennes, France.

[Boebert et Kain 1985] Boebert, W. E. et Kain, R. Y. (1985). A practical alternative to hierarchical integrity policies. *In The 8th National Computer Security Conference*, pages 18–27, Gaithersburg, MD, USA.

[Brewer et Nash 1989] Brewer, D. F. C. et Nash, M. J. (1989). The chinese wall security policy. *In Proceedings of the IEEE Symposium on Security and Privacy*, pages 206–214, Oakland, CA, USA. IEEE.

[Briffaut 2007] Briffaut, J. (2007). *Détection d'intrusions fondée sur un modèle de méta-politique de sécurité : analyse de graphes d'interaction et architecture multi-niveaux*. Thèse de doctorat, Université d'Orléans, Orléans, France. A paraître.

[Briffaut *et al.* 2006a] Briffaut, J., Clement, P., Gad El Rab, M., Toinard, C. et Blanc, M. (2006a). A collaborative approach for access control, intrusion detection and security testing. *In* Smari, W. W. et McQuay, W., éditeurs : *Proceedings of the 2006 International Symposium on Collaborative Technologies and Systems, Special Session on Multi Agent Systems and Collaboration (MASC 2006)*, pages 270–278, Las Vegas, USA. IEEE Computer Society.

[Briffaut *et al.* 2006b] Briffaut, J., Clement, P., Gad El Rab, M., Toinard, C. et Blanc, M. (2006b). A multi-agent and multi-level architecture to secure distributed systems. *In Proceedings of the First International Workshop on Privacy and Security in Agent-based Collaborative Environments (PSACE 2006)*, Hakodate, Japan.

[Briffaut *et al.* 2006c] Briffaut, J., Lalande, J.-F., Toinard, C. et Blanc, M. (2006c). Collaboration between MAC policies and ids based on a meta-policy approach. *In* Smari, W. W. et McQuay, W., éditeurs : *Proceedings of the Workshop on Collaboration and Security (COLSEC'06)*, pages 48–55, Las Vegas, USA. IEEE Computer Society.

[Clark et Wilson 1987] Clark, D. D. et Wilson, D. R. (1987). A comparison of commercial and military computer security policies. *In Proceedings of the IEEE Symposium on Security and Privacy*, pages 184–194, Oakland, CA, USA. IEEE.

[Cuppens et Miège 2003] Cuppens, F. et Miège, A. (2003). Modelling contexts in the Or-BAC model. *In 19th Annual Computer Security Applications Conference (ACSAC 2003)*, Las Vegas, Nevada, USA.

[Damianou *et al.* 2000] Damianou, N., Dulay, N., Lupu, E. et Sloman, M. (2000). Ponder : A language for specifying security and management policies for distributed systems. Rapport technique Research Report DoC 2000/1, Imperial College.

[Department of Defense 1991] Department of Defense (1991). Integrity in Automated Information Systems. Technical Report C 79-91, National Computer Security Center.

[F. Cuppens et A. Miège 2003] F. Cuppens et A. Miège (2003). Administration Model for Or-BAC. *In Workshop on Metadata for Security, International Federated Conferences (OTM'03)*, pages 754–768, Catania, Sicily, Italy.

[F. Cuppens et A. Miège 2004] F. Cuppens et A. Miège (2004). AdOrBAC : An Administration Model for Or-BAC. *Special issue of the International Journal of Computer Systems Science and Engineering*, 19(3).

[Ferraiolo *et al.* 1995] Ferraiolo, D. F., Cugini, J. A. et Kuhn, D. R. (1995). Role-Based Access Control (RBAC) : Features and Motivations. *In 11th Annual Computer Security Applications Conference (ACSAC)*, New Orleans, Louisianna, USA.

[Ferraiolo et Kuhn 1992] Ferraiolo, D. F. et Kuhn, D. R. (1992). Role-based access controls. *In 15th National Computer Security Conference*, pages 554–563, Baltimore, MD, USA.

[Guttman *et al.* 2003] Guttman, J., Herzog, A. et Ramsdell, J. (2003). Information Flow in Operating Systems : Eager Formal Methods. *In Workshop on Issues in the Theory of Security (WITS'03)*, Warsaw, Poland.

[Guttman *et al.* 2005] Guttman, J. D., Herzog, A. L., Ramsdell, J. D. et Skorupka, C. W. (2005). Verifying information flow goals in security-enhanced Linux. *Journal of Computer Security*, 13(1): 115–134.

[Harrison *et al.* 1976] Harrison, M. A., Ruzzo, W. L. et Ullman, J. D. (1976). Protection in operating systems. *Communications of the ACM*, 19(8):461–471.

[Herzog et Guttman 2002] Herzog, A. L. et Guttman, J. D. (2002). Achieving Security Goals with Security-Enhanced Linux.

[Hosmer 1992a] Hosmer, H. H. (1992a). Metapolicies I. *ACM SIGSAC Review*, 10(2–3):18–43.

[Hosmer 1992b] Hosmer, H. H. (1992b). Metapolicies II. *In The 15th National Computer Security Conference*, pages 369–378, Baltimore, MD.

[Hosmer 1993] Hosmer, H. H. (1993). The multipolicy paradigm for trusted systems. *In New Security Paradigms Workshop*, pages 19–32, Little Compton, Rhode Island, USA. ACM Press.

[ITSEC 1991] ITSEC (1991). Information Technology Security Evaluation Criteria (ITSEC) v1.2. Technical report.

[Jajodia *et al.* 2001] Jajodia, S., Samarati, P., Sapino, M. L. et Subrahmanian, V. S. (2001). Flexible support for multiple access control policies. *ACM Transactions on Database Systems (TODS)*, 26(2):214–260.

[Jajodia *et al.* 1997] Jajodia, S., Samarati, P., Subrahmanian, V. S. et Bertino, E. (1997). A Unified Framework for Enforcing Multiple Access Control Policies. *In Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, pages 474–485, Tucson, Arizona, United States. ACM Press.

[Kühnhauser 1995] Kühnhauser, W. E. (1995). On paradigms for security policies in multipolicy environments. *In 11th IFIP International Information Security Conference*, Cape Town, South Africa.

[Lampson 1969] Lampson, B. W. (1969). Dynamic protection structures. *In AFIPS Fall Joint Computer Conference (FJCC 1969)*, volume 35, pages 27–38, Las Vegas, Nevada, USA. AFIPS Press.

[Lampson 1971] Lampson, B. W. (1971). Protection. *In The 5th Symposium on Information Sciences and Systems*, pages 437–443, Princeton University.

[LIDS Project 2006] LIDS Project (2006). LIDS Secure Linux System. `http://www.lids.org/`.

[Linux Documentation Project 2004] Linux Documentation Project (2004). An overview of a Kerberos infrastructure. `http://tldp.org/HOWTO/Kerberos-Infrastructure-HOWTO/overview.html`.

[Loscocco et Smalley 2001] Loscocco, P. et Smalley, S. (2001). Integrating flexible support for security policies into the linux operating system. *In Proceedings of the FREENIX Track : 2001 USENIX Annual Technical Conference (FREENIX '01)*. USENIX.

[Loscocco *et al.* 1998] Loscocco, P. A., Smalley, S. D., Muckelbauer, P. A., Taylor, R. C., Turner, S. J. et Farrell, J. F. (1998). The Inevitability of Failure : The Flawed Assumption of Security in Modern Computing Environments. *In Proceedings of the 21st National Information Systems Security Conference*, pages 303–314, Arlington, Virginia, USA.

[Lupu et Sloman 1999] Lupu, E. C. et Sloman, M. (1999). Conflicts in policy-based distributed systems management. *IEEE Transactions on Software Engineering*, 25(6):852–896.

[Marriott 1993] Marriott, D. A. (1993). Management policy specification. Rapport technique DoC 94/1, Imperial College, London.

[McLean 1987] McLean, J. (1987). Reasoning about security models. *In* IEEE, éditeur : *Proceedings of the IEEE Symposium on Security and Privacy*, pages 123–131, Oakland, CA, USA.

[Medusa DS9 2006] Medusa DS9 (2006). Medusa ds9 security system. `http://medusa.fornax.sk/`.

[Minsky et Ungureanu 2000] Minsky, N. H. et Ungureanu, V. (2000). Law-governed interaction : a coordination and control mechanism for heterogeneous distributed systems. *ACM Transactions on Software Engineering and Methodology*, 9(3):273–305.

[Moffett et Sloman 1991] Moffett, J. D. et Sloman, M. S. (1991). The representation of policies as system objects. *In The Conference on Organizational Computer Systems*, pages 171–184, Atlanta, Georgia, USA.

[Moffett et Sloman 1993a] Moffett, J. D. et Sloman, M. S. (1993a). Policy conflict analysis in distributed system management. *Journal of Organizational Computing*, 4(1):1–22.

[Moffett et Sloman 1993b] Moffett, J. D. et Sloman, M. S. (1993b). Policy hierarchies for distributed system management. *IEEE Journal on Selected Areas in Communications, Special Issue on Network Management and Control*, 11(9):1404–1414.

[Moffett *et al.* 1990] Moffett, J. D., Sloman, M. S. et Twidle, K. P. (1990). Specifying discretionary access control policy for distributed systems. *Computer Communications*, 13(9):571–580.

[Morris 2004] Morris, J. (2004). Recent developments in selinux kernel performance.

[OASIS 2006] OASIS (2006). eXtensible Access Control Markup Language (XACML). `http://www.oasis-open.org/committees/xacml/`.

[Ott 1997] Ott, A. (1997). Rule set based access control as proposed in the 'generalized framework for access control' approach in linux. Mémoire de D.E.A., Universität Hamburg.

[Ott 2001] Ott, A. (2001). The rule set based access control (RSBAC) linux kernel security extension. *In Proceedings of the 8th International Linux Kongress*.

[Ott 2006] Ott, A. (2006). RSBAC benchmarks. `http://rsbac.org/documentation/benchmarks`.

[Pourzandi *et al.* 2003] Pourzandi, M., Apvrille, A., Gingras, E., Medenou, A. et Gordon, D. (2003). Distributed access control for carrier class clusters. *In Proceedings of the Parallel and Distributed Processing Techniques and Applications (PDPTA '03) Conference*, Las Vegas, Nevada, USA.

[Sandhu 1988] Sandhu, R. S. (1988). The schematic protection model : Its definition and analysis for acyclic attenuating schemes. *Journal of the ACM*, 35(2):404–432.

[Sandhu 1992] Sandhu, R. S. (1992). The Typed Access Matrix Model. *In Proceedings of the IEEE Symposium on Research in Security and Privacy*, pages 122–136, Oakland, CA, USA. IEEE.

[Sandhu *et al.* 1996] Sandhu, R. S., Coyne, E. J., Feinstein, H. L. et Youman, C. E. (1996). Role-based access control models. *IEEE Computer*, 29(2):38–47.

[Secure Computing Corporation 1997] Secure Computing Corporation (1997). DTOS generalized security policy specification. Rapport technique, Secure Computing Corporation.

[Sloman 1994] Sloman, M. (1994). Policy driven management for distributed systems. *Journal of Network and Systems Management*, 2(4):333–360.

[Sloman *et al.* 1993] Sloman, M., Magee, J., Twidle, K. et Kramer, J. (1993). An architecture for managing distributed systems. *In The 4th IEEE Workshop on Future Trends of Distributed Computing Systems*, pages 40–46, Lisbon, Portugal. IEEE Computer Society.

[Sloman *et al.* 1992] Sloman, M., Moffett, J. et Twidle, K. (1992). Domino domains and policies : An introduction to the project results. Rapport technique Domino Report Arch/IC/4, Imperial College, London.

[Smalley et Fraser 2000] Smalley, S. et Fraser, T. (2000). A Security Policy Configuration for the Security-Enhanced Linux. Rapport technique, NSA.

[Smalley *et al.* 2001] Smalley, S., Vance, C. et Salamon, W. (2001). Implementing SELinux as a linux security module. Rapport technique, NSA.

[Spencer *et al.* 1999] Spencer, R., Smalley, S., Loscocco, P., Hibler, M., Andersen, D. et Lepreau, J. (1999). The Flask security architecture : System support for diverse security policies. *In Proceedings of The Eighth USENIX Security Symposium*, pages 123–129, Washington, D.C., USA.

[Spengler 2002] Spengler, B. (2002). Detection, prevention, and containment : A study of grsecurity. *In Libre Software Meeting 2002 (LSM2002)*, Bordeaux, France. `http://www.grsecurity.net/papers.php`.

[Spengler 2005] Spengler, B. (2005). Increasing performance and granularity in role-based access control systems. `http://www.grsecurity.net/researchpaper.pdf`.

[TCSEC 1985] TCSEC (1985). Trusted Computer System Evaluation Criteria. Technical Report DoD 5200.28-STD, Department of Defense.

[Walker *et al.* 1996] Walker, K. W., Sterne, D. F., Badger, M. L., Petkac, M. J., Sherman, D. L. et Oostendorp, K. A. (1996). Confining Root Programs with Domain and Type Enforcement. *In Proceedings of The Sixth USENIX Security Symposium*, pages 21–36, San Jose, California.

[Wies 1994] Wies, R. (1994). Policies in network and systems management - formal definition and architecture. *Journal of Network and Systems Management*, 2(1):63–83.

[Wright *et al.* 2002] Wright, C., Cowan, C., Smalley, S., Morris, J. et Kroah-Hartman, G. (2002). Linux security modules : General security support for the linux kernel. *In Proceedings of the 11th USENIX Security Symposium*, San Francisco, CA, USA.

Annexe A

Formats XML

Cette annexe fournit les listings correspondant aux DTD des formats XML définis en section 7.2.

A.1 Format de méta-politique

Le listing suivant contient la DTD définissant le format XML des règles contenues dans la méta-politique $\mathcal{MP}_{\mathcal{AC}}$ (cf. 7.2.2).

```
<!-- ************ -->
<!-- * ELEMENTS * -->
<!-- ************ -->

<!-- TOP Level Element -->
<!-- ***************** -->
<!ELEMENT metapolicy (update_policy, policy)>

<!-- Update policy definition -->
<!ELEMENT update_policy (update_labels_rule_list,
   update_perms_rule_list)>
<!ELEMENT update_labels_rule_list ((update_labels_rule)* )>
<!ELEMENT update_labels_rule (action, label, label_rule)>
<!ELEMENT update_perms_rule_list ((update_perms_rule)* )>
<!ELEMENT update_perms_rule (action, label, access_rule)>

<!-- Action definition -->
<!ELEMENT action (#PCDATA)>

<!-- Label rule definition -->
<!ELEMENT label_rule (label)>
<!ELEMENT label (#PCDATA)>

<!-- Protection policy definition -->
<!ELEMENT policy ((access_rule)* )>

<!-- Access rules definition -->
```

```
<!ELEMENT access_rule (subject, object, perm_list)>
<!ELEMENT subject (#PCDATA)>
<!ELEMENT object (#PCDATA)>
<!ELEMENT perm_list ((perm)* )>
<!ELEMENT perm (#PCDATA)>
```

A.2 Format de politique neutre

Le listing suivant contient la DTD définissant le format XML des règles contenues dans la politique $\mathcal{LR}_{\mathcal{AC}}$ (cf. 7.2.1).

```
<!-- ************ -->
<!-- * ELEMENTS * -->
<!-- ************ -->

<!-- TOP Level Element -->
<!-- ***************** -->
<!ELEMENT policy ((access_rule)* )>

<!-- Access rules definition -->
<!ELEMENT access_rule (subject, object, perm_list)>
<!ELEMENT subject (#PCDATA)>
<!ELEMENT object (#PCDATA)>
<!ELEMENT perm_list ((perm)* )>
<!ELEMENT perm (#PCDATA)>
```

A.3 Format de politique neutre

Le listing suivant contient la DTD définissant le format XML des règles contenues dans les requêtes de mise à jour de la politique $\mathcal{LR}_{\mathcal{AC}}$ (cf. 7.2.3).

```
<!-- ************ -->
<!-- * ELEMENTS * -->
<!-- ************ -->

<!-- TOP Level Element -->
<!-- ***************** -->
<!ELEMENT update_list (update_labels_list, update_perms_list)>

<!-- Query definition -->
<!ELEMENT update_labels_list ((update_labels_query)* )>
<!ELEMENT update_labels_query (action, label_rule)>
<!ELEMENT update_perms_list ((update_perms_query)* )>
<!ELEMENT update_perms_query (action, access_rule)>

<!-- Label rule definition -->
```

```
<!ELEMENT label_rule (label)>
<!ELEMENT label (#PCDATA)>

<!-- Action definition -->
<!ELEMENT action (#PCDATA)>

<!-- Access rules definition -->
<!ELEMENT access_rule (subject, object, class, perm)>
<!ELEMENT subject (#PCDATA)>
<!ELEMENT object (#PCDATA)>
<!ELEMENT perms ((perm)* )>
<!ELEMENT perm (#PCDATA)>
```

Annexe B

Exemple de Méta-Politique en XML

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <!DOCTYPE metapolicy SYSTEM 'meta.dtd'>
3  <metapolicy>
4     <update_policy>
5        <update_labels_rule_list>
6           <update_labels_rule>
7              <action>add</action>
8              <label>(admin_u, admin_r, admin_t)</label>
9              <label_rule>
10                <label>opt_apps_.*</label>
11             </label_rule>
12          </update_labels_rule>
13       </update_labels_rule_list>
14       <update_perms_rule_list>
15          <update_perms_rule>
16             <action>add</action>
17             <label>(admin_u, admin_r, admin_t)</label>
18             <access_rule>
19                <subject>(user_u, user_r, user_shell_t)</subject>
20                <object>(system_u, object_r, opt_apps_.*)</object>
21                <perm_list>
22                   <perm>{file : execute}</perm>
23                   <perm>{process : transition}</perm>
24                </perm_list>
25             </access_rule>
26          </update_perms_rule>
27          <update_perms_rule>
28             <action>add</action>
29             <label>(admin_u, admin_r, admin_t)</label>
30             <access_rule>
31                <subject>(admin_u, admin_r, admin_shell_t)</subject>
32                <object>(system_u, object_r, opt_apps_.*_config_t)</object>
33                <perm_list>
34                   <perm>{file : read, write}</perm>
35                </perm_list>
36             </access_rule>
37          </update_perms_rule>
38          <update_perms_rule>
39             <action>add</action>
40             <label>(admin_u, admin_r, admin_t)</label>
41             <access_rule>
```

```
42                <subject>(user_u, user_r, opt_apps_.*)</subject>
43                <object>(system_u, object_r, opt_apps_.*)</object>
44                <perm_list>
45                   <perm>{file : .*}</perm>
46                </perm_list>
47             </access_rule>
48          </update_perms_rule>
49       </update_perms_rule_list>
50       <update_labels_rule_list>
51       </update_labels_rule_list>
52    </update_policy>
53    <policy>
54    </policy>
55 </metapolicy>
```

Listing B.1 – Méta-Politique en XML.

Sécurité des systèmes d'exploitation répartis : architecture décentralisée de méta-politique pour l'administration du contrôle d'accès obligatoire.

Résumé
Dans cette thèse, nous nous intéressons au contrôle d'accès obligatoire dans les systèmes d'exploitation répartis. Nous présentons une approche novatrice fondée sur un modèle de méta-politique pour l'administration décentralisée des noeuds du système réparti. Ces travaux visent la sécurité des grands réseaux partagés, i.e. les clusters, grilles ou ensembles de noeuds coopérant par Internet. L'architecture que nous proposons garantit des propriétés de sécurité globales et une bonne tolérance aux pannes. Ces propriétés, non prises en comptes par les modèles de contrôle d'accès classiques, sont obtenues grâce à notre méta-politique. Nous en présentons une formalisation, puis nous exhibons une technique de vérification garantissant l'absence de flots d'information illégaux au sein des nœuds du réseau. Nous décrivons ensuite comment le modèle peut être étendu pour la détection d'intrusions. Enfin, nous proposons une implantation supportant différents systèmes cibles tels que SELinux et grsecurity.

Mots-clés
Sécurité, systèmes d'exploitation Open Source, politiques réparties, contrôle d'accès obligatoire, flots d'information.

Security of distributed operating systems: decentralised meta-policy architecture for the administration of mandatory access control.

Abstract
This thesis deals with mandatory access control in distributed systems. We present a novel approach based on a meta-policy model that allows the decentralised administration of distributed nodes. This approach aims at the security of large shared networks, such as computing clusters, grids and distributed stations cooperating over the Internet. The presented solution guarantees global security properties and provides good fault tolerance. These properties are not available with classical access control models, but are achieved through our meta-policy architecture. Then we expose a verification technique that checks the presence of illegal information flows in the distributed nodes. We also describe how our approach allows intrusion detection extensions that complement the access control. Finally, we provide an implementation supporting various target operating systems such as SELinux and grsecurity.

Keywords
Security, Open Source operating system, distributed policies, mandatory access control, information flows.

Discipline : Informatique

Laboratoire d'Informatique Fondamentale d'Orléans
Bâtiment IIIA
Rue Léonard de Vinci
B.P. 6759
F-45067 ORLEANS Cedex 2