HAL open science

Application et assurance autonomes de propriétés de sécurité dans un environnement d'informatique en nuage

Aline Bousquet

To cite this version:

Aline Bousquet. Application et assurance autonomes de propriétés de sécurité dans un environnement d'informatique en nuage. Cryptographie et sécurité [cs.CR]. Université d'Orléans, 2015. Français. NNT : 2015ORLE2012 . tel-01280846

HAL Id: tel-01280846
https://theses.hal.science/tel-01280846

Submitted on 1 Mar 2016

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.

Centre Val de Loire UNIVERSITÉ Spirit of the Loire Valley

UNIVERSITÉ D'ORLÉANS

UNIVERSITE D'ORLEANS

ÉCOLE DOCTORALE MATHEMATIQUES, INFORMATIQUE, PHYSIQUE THEORIQUE et INGENIERIE DES SYSTEMES

LABORATOIRE D'INFORMATIQUE FONDAMENTALE D'ORLÉANS

THÈSE présentée par :

Aline BOUSQUET

soutenue le : **2 Décembre 2015**

pour obtenir le grade de : **Docteur de l'université d'Orléans**

Discipline/ Spécialité : **Informatique**

Application et assurance autonomes de propriétés de sécurité dans un environnement d'informatique en nuage

THÈSE dirigée par :

Christian TOINARD	Professeur, INSA Centre Val de Loire

RAPPORTEURS :

Hervé DEBAR	Professeur, Télécom SudParis
Christine MORIN	Directrice de Recherche, Inria Rennes

JURY :

Stéphane BETGE-BREZETZ	Ingénieur de Recherche, Alcatel-Lucent Bell Labs
Jérémy BRIFFAUT	Maître de conférences, INSA Centre Val de Loire
Jean-Michel COUVREUR	Professeur, Université d'Orléans
Hervé DEBAR	Professeur, Télécom SudParis
Christine MORIN	Directrice de Recherche, Inria Rennes
Christian TOINARD	Professeur, INSA Centre Val de Loire

Remerciements

Je remercie Hervé Debar et Christine Morin pour avoir accepté d'évaluer mes travaux en étant rapporteurs de cette thèse. Merci également à Stéphane Betge-Brezetz et Jean-Michel Couvreur pour leur participation au jury d'évaluation.

Je tiens à remercier mon directeur de thèse, Christian Toinard, pour m'avoir donné l'opportunité d'effectuer cette thèse.

Je souhaite aussi remercier Jérémy Briffaut, qui a co-encadré ma thèse, pour son soutien et pour son aide scientifique et technique tout au long de ces trois années.

Je remercie également Celtic+ pour avoir financé cette thèse, et l'ensemble des partenaires du projet Seed4C pour les discussions enrichissantes et leurs remarques pertinentes sur ces travaux, ainsi que pour la bonne ambiance au sein de ce projet. Merci tout particulièrement à Stéphane Betge-Brezetz et Bertrand Marquet pour leur investissement en tant que coordinateurs de Seed4C.

Je souhaite également remercier les membres de l'équipe SDS du LIFO pour leur accueil. Merci à Asma Guesmi pour ces trois années passées à travailler sur nos thèses respectives et à Ahmad Abdallah pour son excellent café.

Enfin, merci à mes parents, à ma soeur et à mes amis pour leur patience et leurs encouragements, et pour avoir relu ce manuscrit.

Table des matières

1 Introduction **1**
1.1 Contexte et objectifs 1
1.2 Apports de la thèse 2
1.3 Plan du mémoire 3

2 Etat de l'Art **5**
2.1 Propriétés et politiques de sécurité 5
2.1.1 Propriétés de sécurité 5
2.1.2 Politiques de sécurité 7
2.2 L'informatique en nuage 14
2.2.1 Définition 14
2.2.2 Exemples d'infrastructures en nuage 16
2.2.3 Expression des besoins 21
2.2.4 Discussion 24
2.3 Mécanismes et propriétés de sécurité 24
2.3.1 Système 24
2.3.2 Réseau 25
2.3.3 Éléments matériels 26
2.3.4 Protection de logiciels 26
2.3.5 Virtualisation et informatique en nuage 27
2.3.6 Synthèse 29
2.4 Conclusion 30

3 Langage **31**
3.1 Caractéristiques du langage 32
3.2 Vue globale 33
3.3 Contextes 34
3.3.1 Définition des contextes 34
3.3.2 Compatibilité de contextes 36
3.3.3 Association des contextes aux ressources 41
3.4 Capacités 43
3.4.1 Définition des capacités 43
3.4.2 Surcharge et priorité des capacités 44
3.4.3 Association des capacités aux mécanismes de sécurité 47
3.4.4 Définition des capacités internes 51
3.4.5 Synthèse 52
3.5 Propriétés 53
3.5.1 Définition des propriétés 53
3.5.2 Catégorisation de propriétés 55

3.5.3 Classes de propriétés . . . 58
3.5.4 Instance de propriétés . . . 60
3.5.5 Synthèse . . . 62
3.6 Politique de sécurité . . . 63
3.6.1 Définition . . . 63
3.6.2 Score d'application . . . 65
3.7 Conclusion . . . 66

4 Architecture et Projection 67
4.1 Architecture globale . . . 67
4.1.1 Modélisation et déploiement de la politique . . . 68
4.1.2 Application et assurance . . . 69
4.2 Architecture du SE^E . . . 70
4.2.1 Architectures autonomes . . . 70
4.2.2 Architecture du SE^E . . . 72
4.3 Le gestionnaire de politique . . . 74
4.4 L'application . . . 75
4.4.1 Moteur d'application . . . 76
4.4.2 Sélection des propriétés . . . 78
4.4.3 Sélection des mécanismes . . . 80
4.4.4 Application des propriétés . . . 89
4.5 L'assurance . . . 90
4.5.1 Moteur d'assurance . . . 91
4.5.2 Gestion des erreurs . . . 92
4.5.3 Assurance des mécanismes . . . 94
4.5.4 Assurance des propriétés . . . 96
4.5.5 Traitement des informations d'assurance . . . 97
4.6 Scores d'application et cycle de vie de la politique . . . 99
4.7 Conclusion . . . 101

5 Implémentation 103
5.1 Implémentation du SE^E . . . 103
5.1.1 Architecture globale . . . 103
5.1.2 Flux d'exécution . . . 104
5.2 Implémentation des modules d'extension . . . 108
5.2.1 Interface . . . 109
5.2.2 Déclaration des capacités . . . 110
5.2.3 Fonctionnement classique d'un module d'extension . . . 111
5.2.4 Modules d'extension existants . . . 113
5.3 Exemple de déploiement d'une politique de sécurité . . . 116
5.3.1 Environnement considéré . . . 116
5.3.2 Compilation des propriétés . . . 118
5.3.3 Projection des propriétés . . . 121
5.3.4 Assurance des propriétés . . . 124
5.3.5 Assurance des mécanismes . . . 125
5.3.6 Traitement des informations d'assurance . . . 126
5.4 Conclusion . . . 127

6 Expérimentation **129**
6.1 Description du cas d'usage 129
6.1.1 Environnement de test 129
6.1.2 Cas d'usage 130
6.2 Politique de sécurité 132
6.2.1 Besoins de sécurité 132
6.2.2 Politique de sécurité 133
6.2.3 Contextes 134
6.2.4 Propriétés 138
6.3 Application et assurance 141
6.3.1 IaaS 141
6.3.2 PaaS 143
6.3.3 SaaS 144
6.3.4 Synthèse 145
6.4 Résultats et évaluations 145
6.4.1 Taille de la politique 146
6.4.2 Performances 146
6.4.3 Évaluation de l'application de la politique 148
6.5 Conclusion 156

7 Conclusion **157**

Annexes **171**
A Langage 171
A.1 Capacités 171
A.2 Classes et Propriétés 173
A.3 Grammaire 180
B Expérimentation 181
B.1 Contextes 181
B.2 Politiques 183

Table des figures

2.1 Modèles de service 16
2.2 Exemple d'architecture de virtualisation 17

3.1 Éléments du langage d'expression des besoins de sécurité 33
3.2 Extrait du sous-arbre de la clef *Hardware* 35
3.3 Extrait de l'arbre des clefs de contextes 36
3.4 Exemples de types de contextes 39
3.5 Exemple d'architecture logicielle à sécuriser 63

4.1 Architecture globale basée sur l'architecture Seed4C 68
4.2 Architecture autonome : boucle MAPE-K [Jacob *et al.*, 2004] 71
4.3 Architecture globale du SE^E 72
4.4 Architecture interne du SE^E 73
4.5 Gestionnaire de politique 74
4.6 Algorithme de projection (Algo. 8) 76
4.7 Algorithme de recherche des propriétés compatibles avec P (algo. 9) 78
4.8 Algorithme de recherche des SSM compatibles avec P (algo. 12) 81
4.9 Algorithme récursif de sélection de mécanismes (algo. 14) 84
4.10 Exemple de recherche d'une clique de poids maximum 87
4.11 Principe du moteur d'assurance 91
4.12 Processus d'assurance (Algo. 20) 92
4.13 Traitement d'une erreur lors de l'application de la politique 93
4.14 Cycle de vie d'un module d'extension 95
4.15 Traitement d'une erreur au cours du fonctionnement du système 98
4.16 Cycle de vie de la politique 101

5.1 Architecture détaillée du SE^E 104
5.2 Interface moteur d'application et module d'extension 109
5.3 Proposition d'architecture de module d'extension 111
5.4 Architecture de l'exemple 116

6.1 OpenStack 130
6.2 Description du cas d'usage 131
6.3 Description du cas d'usage 133
6.4 Description du cas d'usage 134
6.5 Temps de lancement des VM 147
6.6 Temps d'application de la politique par le SE^E 148
6.7 Score d'application théorique de la politique par le SE^E 149
6.8 Score d'application réel de la politique par le SE^E 151
6.9 Taux de couverture de la politique 152

6.10 Score d'application réel pour la VM Proxy en fonction de N_T 152
6.11 Taux de couverture de la politique de la VM Proxy 153

Liste des tableaux

2.1 Systèmes d'informatique en nuage 21
2.2 Propriétés de sécurité couvertes par les mécanismes 29

3.1 Exemples de scores d'inclusion de capacités 46

4.1 Table des erreurs 90

5.1 Nombre de capacités par mécanismes 115
5.2 Description de l'environnement 116
5.3 Application de la propriété hybride P_H 120
5.4 Évolution des scores pour P_H 127

6.1 Configuration de l'environnement de test 129
6.2 Mécanismes utilisés par machine et par niveau de politique 145
6.3 Taille des politiques 146
6.4 Scores d'application des politiques par VM et par étape de test 150

Chapitre 1

Introduction

1.1 Contexte et objectifs

L'informatique en nuage est de plus en plus utilisée, à la fois par les particuliers et les entreprises. En effet, de nombreuses entreprises (Google, Dropbox, Amazon, etc.) fournissent des services reposant sur l'informatique en nuage ou permettent d'utiliser une infrastructure en nuage pour déployer des services. De plus, l'importance économique de l'informatique en nuage continue d'augmenter comme montré dans une étude publiée en 2014 [Bartels *et al.*, 2014] : en 2013, le marché de l'informatique en nuage représentait 58 milliards de dollars. L'étude prévoit que ce marché devrait atteindre 191 milliards de dollars en 2020. Cette forte augmentation témoigne donc de l'importance croissante de l'informatique en nuage.

Cependant, dans une enquête [CSA, 2015] du CSA (*Cloud Security Alliance*), l'un des freins majeurs à l'adoption de l'informatique en nuage concerne les problèmes de sécurité des données, puisque 73% des participants interrogés s'en inquiètent. Les fournisseurs de services en nuage doivent donc apporter une réponse aux questions de leurs utilisateurs concernant la sécurité : en effet, les deux principaux critères sur lesquels les entreprises évaluent les fournisseurs de services [KPMG, 2014] sont la sécurité des données (82%) et la vie privée (81%), alors que le critère de coût n'arrive qu'en troisième position (78%). La sécurité est donc un élément crucial pour les clients et par conséquent pour les fournisseurs de services. Ainsi, malgré l'importance croissante de l'informatique en nuage, les problèmes de sécurité persistent et doivent être adressés.

C'est dans ce contexte que le projet européen Celtic-Plus Seed4C [1] (*Security Embedded Element and Data privacy for the Cloud*) a été créé. Ce projet a impliqué dix-sept partenaires, à la fois industriels et académiques, en provenance de quatre pays (Corée, Espagne, Finlande et France). L'approche proposée par le projet Seed4C repose sur l'association d'éléments sécurisés matériels (*Secure Element*) aux machines physiques de l'infrastructure en nuage, et d'éléments logiciels aux machines physiques et virtuelles. Elle a pour objectif d'améliorer la sécurité de l'infrastructure en proposant une gestion complète d'une politique de sécurité depuis sa modélisation jusqu'à son application et son assurance.

Cette thèse a été effectuée dans le cadre du projet Seed4C. Elle vise à améliorer la sécurité de l'informatique en nuage en définissant, en appliquant, en assurant et en maintenant à jour une politique de sécurité dans un environnement en nuage. Cette politique est basée sur des propriétés de sécurité, telles que l'intégrité ou la confidentialité, afin de protéger à la fois les données, et les différents niveaux et applications d'une architecture en nuage.

1. `http://celticplus-seed4c.org/`

Tout d'abord, la solution proposée doit permettre d'exprimer les besoins de sécurité d'une architecture logicielle hébergée dans une infrastructure en nuage. Le nuage informatique étant un environnement hétérogène, la solution doit être indépendante du système sur lequel la politique s'applique. De plus, afin de couvrir un large spectre de systèmes et de besoins de sécurité, nous proposons de réutiliser les mécanismes de sécurité existants puisqu'il en existe actuellement un grand nombre, chacun étant spécialisé dans l'application d'un sous-ensemble de propriétés. En outre, les différentes machines à sécuriser ne disposent pas nécessairement des mêmes mécanismes de sécurité : la solution doit abstraire les mécanismes afin que les besoins de sécurité puissent être exprimés sans en dépendre explicitement.

Le second objectif de cette thèse concerne la mise à jour de la protection tout au long de la vie des machines. En effet, la solution doit être capable de détecter des changements se produisant sur le système et nuisant à la bonne application des besoins (indisponibilité d'un mécanisme, problème lors de l'application d'une propriété, etc.). Lorsqu'un tel événement est détecté, la solution proposée doit également être capable de réagir afin de continuer à répondre aux besoins de sécurité. Enfin, la solution doit proposer une méthode d'évaluation de la qualité de l'application, afin de pouvoir facilement comparer l'application d'une même politique sur différentes machines ou sur une seule machine au cours du temps.

1.2 Apports de la thèse

Au travers de l'état de l'art, nous montrons que malgré l'existence de langages et d'architectures dédiés à la sécurité, il n'existe pas actuellement de solution adaptée à des environnements hétérogènes et capable de configurer automatiquement des mécanismes de sécurité pour appliquer une politique basée sur des propriétés.

Ainsi, nous définissons un langage d'expression des besoins de sécurité. Ce langage permet à un client de l'informatique en nuage de définir une politique de sécurité qu'il souhaite voir appliquer. Cette politique est indépendante du système de nommage des ressources grâce à l'utilisation de contextes associés aux ressources : les propriétés de sécurité sont définies par rapport à ces contextes et un changement du nommage des ressources ne modifie pas la politique. De plus, le langage s'abstrait des mécanismes sous-jacents en faisant intervenir des capacités, qui correspondent aux fonctionnalités élémentaires fournies par les mécanismes. Ces capacités sont utilisées pour définir la méthode d'application des propriétés : une même capacité pouvant être offerte par plusieurs mécanismes, les propriétés peuvent être appliquées par des ensembles de mécanismes différents afin de s'adapter à ceux disponibles sur une machine donnée. Notons également que le langage présente deux vues distinctes : la première se compose uniquement des propriétés et des contextes et permet de définir les besoins de sécurité d'une architecture, tandis que la seconde correspond à la définition des propriétés à partir des capacités. Le langage offre ainsi la possibilité de définir de nouvelles propriétés ou de modifier les propriétés existantes.

Nous proposons une architecture permettant d'appliquer une politique de sécurité dans un environnement hétérogène. Nous utilisons pour cela une architecture multi-agents, où chaque agent suit le modèle de l'architecture autonome d'IBM. Ainsi, chaque agent applique la politique dédiée à une machine précise en configurant les mécanismes disponibles en accord avec la politique. Chaque agent offre également la possibilité de collaborer avec d'autres agents afin d'appliquer une propriété distribuée. Cette configuration est effectuée à l'aide de modules d'extension, orientés application, qui sont associés aux mécanismes. Cela permet à l'architecture de gérer de nombreux mécanismes et de faciliter l'ajout de nouveaux mécanismes. De plus, d'autres modules d'extension, orientés assurance, permettent

aux agents de détecter les éventuelles erreurs pouvant se produire. Les agents sont alors capables d'analyser les erreurs rencontrées afin d'y apporter une réponse appropriée. L'architecture est associée à des algorithmes d'application et d'assurance qui spécifient comment les propriétés sont appliquées (c'est-à-dire avec quels mécanismes) et comment elles sont assurées (quelles actions sont effectuées lorsqu'une erreur est détectée). Enfin, une méthode d'évaluation de l'application de la politique est proposée : celle-ci se base notamment sur des scores attribués aux mécanismes.

La thèse fournit une implémentation de cette architecture. Cette implémentation est capable d'interpréter l'intégralité du langage proposé afin d'appliquer et d'assurer une politique. Elle peut aussi effectuer les phases de reconfiguration automatique et d'évaluation. Elle a été testée dans le cadre du projet Seed4C sur plusieurs cas d'usage industriels ainsi que dans une expérimentation présentée dans ce document.

1.3 Plan du mémoire

La suite de ce document est organisée en cinq chapitres. Le chapitre 2 présente les travaux existants dans les domaines liés à cette thèse, c'est-à-dire l'informatique en nuage, ainsi que les propriétés de sécurité et les langages permettant d'exprimer des politiques, et les mécanismes existants supportant l'application des propriétés. Ce chapitre remarque l'absence d'un langage et d'une architecture permettant de sécuriser un environnement distribué et hétérogène, sur l'ensemble des niveaux impliqués (système, réseau, logiciel, matériel, etc.).

Le chapitre 3 propose un langage d'expression des besoins de sécurité adapté aux environnements en nuage. En raison de l'hétérogénéité du type d'environnement visé, le langage est indépendant du système sur lequel la politique s'applique : le langage peut s'adapter à différents systèmes de nommage, sans nécessiter de réécriture des propriétés de sécurité. De plus, l'hétérogénéité du système impliquant la présence de différents mécanismes de sécurité selon les machines, le langage est indépendant de ces mécanismes : une même propriété pourra être appliquée par différents ensembles de mécanismes en fonction de ceux disponibles sur le système considéré. Ce langage permet ainsi de définir, pour chaque propriété de sécurité, les processus d'application et d'assurance en s'abstrayant des mécanismes de sécurité sous-jacents.

Le chapitre 4 présente une architecture capable de compiler et de projeter le langage d'expression des besoins de sécurité. La solution proposée, appelée SE^E (*Security Enforcement Engine*), permet donc d'appliquer l'ensemble des propriétés de sécurité en prenant en compte les mécanismes disponibles, leurs contraintes et leurs dépendances. Le SE^E configure également des mécanismes d'assurance afin de vérifier l'état d'application de la politique et réagir à d'éventuelles erreurs ou dysfonctionnements rencontrés. Le chapitre présente donc les algorithmes permettant la recherche d'une solution pour l'application, ainsi que la configuration de l'application et de l'assurance de la politique en fonction du système.

Le chapitre 5 décrit l'implémentation du SE^E qui a été réalisée dans le cadre de ces travaux. Le SE^E est donc capable d'interpréter le langage du chapitre 3 et implémente les algorithmes définis au chapitre 4. Les modules d'extension sont également introduits : ils constituent l'interface entre le cœur du SE^E, qui compile et projette la politique, et les mécanismes de sécurité existants, sur lesquels l'architecture repose pour effectivement appliquer les propriétés. Finalement, ce chapitre détaille un exemple de déploiement d'une politique de sécurité, notamment en précisant comment les mécanismes sont sélectionnés à partir de leurs scores d'ordonnancement et d'application.

Enfin, le chapitre 6 présente une expérimentation généralisant un cas d'usage industriel. L'application de la politique et la génération des tests d'assurance sont ainsi décrites. Des exemples de dysfonctionnements des mécanismes et la réaction du SE^E sont également présentés, ce qui permet d'observer le processus d'assurance et les étapes de reconfiguration automatique. De plus, des évaluations des performances sont effectuées afin de mesurer l'impact de l'application de la politique sur le déploiement de machines virtuelles. Finalement, la qualité de l'application des politiques est évaluée à partir des scores d'application définis pour les propriétés et des taux de couverture. Cette expérimentation permet ainsi de démontrer que le langage défini est adapté aux environnements en nuage. Elle montre également que l'architecture et l'implémentation proposées s'adaptent à des systèmes différents et peuvent réagir aux problèmes rencontrés lors de l'application ou au cours de la vie du système.

Ce rapport se conclut par une synthèse des résultats obtenus et des perspectives à explorer lors de travaux futurs.

Chapitre 2

Etat de l'Art

Les problématiques de l'expression et de l'application des besoins de sécurité sont présentes sur tous les systèmes d'information. Avec l'émergence de nouveaux types d'environnements, tels que l'informatique en nuage, cette problématique se complexifie puisqu'il est nécessaire de prendre en compte leur hétérogénéité et les différents niveaux d'une telle architecture. Ce chapitre présente les travaux en lien avec cette thèse et portant sur l'expression et l'application des besoins de sécurité dans les environnements en nuage.

Nous présentons tout d'abord les propriétés de sécurité qui permettent d'exprimer les besoins de sécurité d'une architecture logicielle. Ces propriétés sont ensuite combinées pour former une politique de sécurité : différents modèles et langages d'expression de politiques de sécurité et d'assurance seront donc détaillés. Puis, nous introduisons l'informatique en nuage, qui constitue le contexte d'étude de la thèse. Nous détaillons notamment des méthodes permettant l'expression des besoins dans ces systèmes hétérogènes. Enfin, nous présentons des mécanismes déjà existants qui permettent d'appliquer des propriétés de sécurité sur certains types de ressources et de protéger une infrastructure logicielle.

2.1 Propriétés et politiques de sécurité

Dans cette section, nous décrivons des méthodes permettant de formaliser les différents besoins de sécurité. Nous présentons tout d'abord les propriétés de sécurité usuelles et celles qui peuvent en être dérivées. Puis, nous nous intéressons aux modèles existants permettant d'appliquer ces propriétés, et aux politiques de sécurité et langages permettant de les définir.

2.1.1 Propriétés de sécurité

Les propriétés de sécurité sont la base de l'expression des besoins de sécurité. L'ensemble des propriétés de sécurité est communément vu comme un ensemble dérivé de trois propriétés principales : **confidentialité**, **intégrité** et **disponibilité** (*CIA : Confidentiality, Integrity, Availability*). L'interprétation exacte de ce qu'impliquent ces trois propriétés varie selon le contexte d'utilisation, mais leur définition et leur application sont une part essentielle des critères d'évaluation de la sécurité, au niveau européen [ITSEC, 1991] et international [Tcsec, 1985]. Plusieurs définitions de ces propriétés existent dans la littérature [Tcsec, 1985, ITSEC, 1991, Bishop, 2003] : nous en présentons ici une synthèse.

2.1.1.1 Confidentialité

La confidentialité consiste à empêcher la divulgation non-autorisée d'information. Elle vise donc à interdire les accès non-autorisés à l'information. La propriété de confidentialité peut être exprimée comme suit :

Définition 2.1.1: Confidentialité
Une information est dite confidentielle si elle n'est accessible que par les entités autorisées.

La propriété de confidentialité implique donc que l'information n'est accessible que par certaines entités, c'est-à-dire que certaines entités ne doivent pas pouvoir obtenir l'information. Cette propriété est souvent utilisée dans les milieux sensibles, comme celui de la défense. Elle concerne à la fois les accès directs et les transferts d'information.

2.1.1.2 Intégrité

La propriété d'intégrité désigne le fait qu'une information ne puisse pas être modifiée ou supprimée de manière non autorisée, que ce soit lors du traitement, du stockage ou du transfert de l'information. Cette propriété concerne les modifications et suppressions volontaires, mais également les actes accidentels. On définit ainsi la propriété d'intégrité :

Définition 2.1.2: Intégrité
Une information est dite intègre si elle ne peut être modifiée ou supprimée que par les entités autorisées.

Tout comme dans le cas de la confidentialité, la propriété d'intégrité spécifie l'ensemble des entités autorisées à modifier ou supprimer une information. Par défaut, les autres entités ne peuvent donc pas altérer l'information.

2.1.1.3 Disponibilité

La propriété de disponibilité exprime la possibilité d'accéder à une information ou une ressource. Elle est liée à la fiabilité d'un système, puisqu'un système non disponible est un système défaillant.

Définition 2.1.3: Disponibilité
Une information est dite disponible si les entités autorisées peuvent y accéder à tout instant.

La propriété de disponibilité empêche donc la rétention d'information. La notion temporelle de l'accès est relative au domaine d'application : l'accès à une information ou un service sur un système critique (par exemple, dans le domaine médical) doit être fait plus rapidement que sur un système non critique (par exemple, un site internet).

2.1.1.4 Assurance

Le triplet de propriétés de confidentialité, intégrité et disponibilité est parfois étendu avec d'autres propriétés [Stoneburner, 2001], telle que la propriété d'assurance. Celle-ci a pour objectif de fournir des preuves que les autres propriétés ont été appliquées et qu'elles ont l'effet voulu. Elle est définie comme suit :

Définition 2.1.4: Assurance
L'assurance d'une propriété est la vérification que cette propriété a été correctement appliquée.

Le niveau d'assurance nécessaire dépend du système considéré et doit donc être adapté selon la criticité du système. La propriété d'assurance a donc pour objectif de vérifier que les autres propriétés de la politique ont été appliquées et que cette application a eu l'effet escompté.

2.1.1.5 Propriétés dérivées

Les propriétés de confidentialité, d'intégrité et de disponibilité sont les concepts élémentaires de la sécurité. Elles peuvent être utilisées afin de définir des propriétés dérivées, qui sont des cas particuliers, des sous-ensembles ou des combinaisons de ces propriétés de base. Dans cette section, nous décrivons quelques unes de ces propriétés dérivées.

Confinement de processus Le confinement de processus a été défini comme suit par Lampson [Lampson, 1973] :

Définition 2.1.5: Confinement de processus
Le problème du confinement concerne la prévention de la divulgation, par un service (ou un processus), d'information considérée comme confidentielle par les utilisateurs de ce service.

D'après Lampson, l'une des caractéristiques nécessaires pour qu'un processus ne puisse pas divulguer d'information est qu'il ne doit pas stocker d'information. En effet, si un processus stocke de l'information et qu'un utilisateur peut observer ce processus, alors il y a un risque que l'utilisateur puisse accéder à l'information. Si le processus ne stocke pas l'information, alors elle ne pourra pas être divulguée. Cette propriété de confinement de processus peut ainsi être vue comme l'isolation du processus du reste du système.

Authentification La propriété d'authentification est une propriété permettant d'autoriser ou d'interdire l'accès à une information ou un service à des entités. Elle peut être définie comme suit [Burr *et al.*, 2013] :

Définition 2.1.6: Authentification
L'authentification est le processus d'établissement de la confiance en l'identité d'une entité.

Cette propriété d'authentification est essentielle pour appliquer les propriétés vues précédemment : en effet, elle permet d'établir l'identité d'un utilisateur, d'un service ou d'un système, ce qui est nécessaire pour lui donner les droits appropriés.

2.1.1.6 Discussion

Les propriétés de sécurité qui ont été présentées dans cette section sont des propriétés textuelles et abstraites. Elles permettent à un utilisateur (par exemple un client de l'informatique en nuage) d'exprimer ses besoins de sécurité de manière littérale. Cependant, ces propriétés correspondent à des besoins de haut niveau qui, s'ils facilitent l'expression de la sécurité par l'utilisateur, ne peuvent pas être directement appliqués sur le système. C'est pourquoi différents modèles et langages de sécurité visent à exprimer ces propriétés de façon à pouvoir les appliquer sur un système.

2.1.2 Politiques de sécurité

Une politique de sécurité est un ensemble de propriétés permettant d'exprimer les besoins de sécurité d'un système ou d'un ensemble de systèmes. La plupart des propriétés de sécurité se basent sur un contrôle d'accès aux ressources. Par conséquent, les concepts

introduits dans les modèles de contrôle d'accès peuvent être transposés dans un langage permettant d'exprimer formellement les propriétés de la section précédente. Dans cette section, nous décrivons donc les principaux modèles de contrôle d'accès historiques (section 2.1.2.1). Puis, nous présentons des langages d'expression de politiques de sécurité (section 2.1.2.2) et de politiques d'assurance (section 2.1.2.3), qui permettent d'exprimer et de mettre en pratique des propriétés de sécurité.

2.1.2.1 Modèles historiques

Certaines propriétés de sécurité, définies précédemment, peuvent être appliquées par des mécanismes de contrôle d'accès. Dans cette section, nous détaillons donc des modèles de contrôle d'accès historiques, qui ont introduit les concepts repris par la suite par différents modèles de politiques de sécurité. Un système de contrôle d'accès est généralement modélisé à l'aide des trois éléments suivants :

- un ensemble de **sujets** qui sont les entités actives du système (par exemple, les processus) ;
- un ensemble d'**objets** qui sont les entités passives du système, sur lesquelles les sujets peuvent effectuer des actions (fichiers, sockets, etc.) ;
- un ensemble de **permissions** qui représentent les actions autorisées entre un sujet et un objet (lecture, écriture, etc.), ou entre deux sujets (envoi d'un signal).

Contrôle d'accès discrétionnaire Le contrôle d'accès discrétionnaire (DAC, *Discretionary Access Control*) est le modèle historique présent par défaut sur la majorité des systèmes d'exploitations. Dans ce modèle, la gestion des droits d'accès à une ressource est laissée à la *discrétion* du propriétaire de cette ressource. Par exemple, sous Unix, le propriétaire d'un fichier peut fixer les droits de lecture, d'écriture et d'exécution pour lui-même, pour les membres du groupe propriétaire du fichier, et pour l'ensemble des autres utilisateurs du système.

Un modèle de contrôle d'accès peut être représenté sous forme de matrice, où une ligne représente un sujet, une colonne représente un objet ou un sujet, et chaque élément de la matrice représente un ensemble de permissions du sujet sur l'objet (ou sur le second sujet). Ce modèle a été formalisé par **Lampson** [Lampson, 1969, Lampson, 1971] en utilisant les listes de capacités et les listes de contrôle d'accès (ACL, *Access Control List*). Il propose donc d'indiquer, dans une matrice A, l'ensemble D des domaines de protection (représentant des contextes d'exécution des programmes, c'est-à-dire des sujets) sur les lignes, et l'ensemble X des objets sur les colonnes. Lampson définit donc les *listes de capacités* (définition 2.1.7) qui établissent les permissions d'un domaine d sur l'ensemble des objets o du système. Il s'agit donc de l'ensemble des actions autorisées pour chaque domaine d.

Définition 2.1.7: Liste de capacités
Étant donné un domaine $d \in D$, la liste des capacités (capabilities) pour le domaine d est l'ensemble des couples $(o, A[d, o]), \forall o \in X$.

Puis, Lampson définit les *listes de contrôle d'accès* (définition 2.1.8) qui spécifient l'ensemble des permissions accordées sur un objet pour chaque domaine du système.

Définition 2.1.8: Liste de contrôle d'accès (ACL)
Étant donné un objet $o \in X$, la liste de contrôle d'accès (ACL) pour l'objet o est l'ensemble des couples $(d, A[d, o]), \forall d \in D$.

Ce modèle se révèle cependant complexe à mettre à jour. Par exemple, lors de l'ajout d'un nouvel utilisateur au système, il faut ajouter une ligne complète à la matrice A. Le modèle de Lampson a donc évolué vers le modèle HRU.

Dans le modèle **HRU** [Harrison *et al.*, 1976], le contrôle d'accès discrétionnaire est modélisé à partir d'une matrice P contenant l'ensemble des droits des sujets sur les objets. Dans ce modèle, les sujets peuvent modifier la matrice de contrôle d'accès afin de créer ou supprimer des sujets ou des objets, mais également de modifier les permissions existantes. Le modèle HRU modélise la protection en utilisant les éléments suivants :

- une matrice de contrôle d'accès P ;
- un ensemble S de sujets et un ensemble O d'objets ;
- un ensemble R de droits génériques (lecture, écriture, exécution, possession, etc.) ;
- un ensemble fini C de commandes représentant l'ensemble des opérations fournies par le système d'exploitation (création de fichiers, modification de droits, etc.) ;
- un ensemble E d'opérations élémentaires : *enter* et *delete* (ajout et suppression de droits), *create subject* et *create object* (ajout de sujets et d'objets), *destroy subject* et *destroy object* (destruction de sujets et d'objets).

Un triplet (S, O, P) représente donc la configuration de la protection du système.

Afin d'étudier le problème de la sûreté d'un système de protection, les auteurs du modèle HRU s'intéressent au transfert de privilège (droit) se produisant lorsqu'une commande insère un droit r dans la matrice P. Le problème de sûreté peut donc être défini de la manière suivante :

Définition 2.1.9: Sûreté d'un système de protection
Étant donnée une configuration initiale de la politique de sécurité, un système est considéré sûr (safe) pour un droit r si aucune des commandes de ce système ne provoque le transfert du droit r.

Les auteurs ont alors montré que, si les commandes ne contiennent qu'une seule action élémentaire, le problème de sûreté est décidable mais son algorithme de vérification est NP-complet. Cependant, dans le cas général (les commandes contiennent plusieurs actions élémentaires), le problème est indécidable. Dans le cas d'un système d'exploitation, les commandes ne sont pas mono-opérationnelles et le problème de la sûreté du système de protection est donc indécidable. Cela montre ainsi qu'il est impossible de garantir des propriétés de sécurité avec un modèle de contrôle d'accès discrétionnaire.

D'autres modèles DAC ont été définis afin d'étendre le modèle HRU, notamment TAM (*Typed Access Matrix*) [Sandhu, 1992] et DTAM (*Dynamic Typed Access Matrix*) [Soshi *et al.*, 2004]. TAM étend le modèle HRU en y intégrant une notion de *typage fort* [Sandhu, 1988] qui correspond à l'association de types de sécurité immuables à tous les sujets et objets du système. DTAM étend quant à lui le modèle TAM en y ajoutant la possibilité de modifier les types des objets de façon dynamique.

Les différents modèles de contrôle d'accès discrétionnaire sont des modèles historiques. Ils sont principalement utilisés pour la gestion des droits systèmes.

Contrôle d'accès obligatoire Le contrôle d'accès discrétionnaire laisse la gestion des permissions sur les ressources à leurs propriétaires, c'est-à-dire aux utilisateurs du système. En pratique, ce système est limité puisque de nombreuses attaques contre les systèmes visent à obtenir un accès privilégié (*root*, ou super-utilisateur). Une telle attaque, dite élévation de privilège, a pour but d'obtenir des droits plus importants que ceux qui sont possédés. Elle permet alors à l'utilisateur d'obtenir un accès complet au système et à ses ressources et de s'affranchir du contrôle discrétionnaire. De plus, des études

[Ferraiolo et Kuhn, 1992, Loscocco *et al.*, 1998] ont montré que les modèles DAC sont vulnérables, notamment en raison de la nécessité pour les utilisateurs de définir correctement l'ensemble des permissions sur les ressources. Toute erreur de définition peut engendrer une faille de sécurité qui peut être exploitée afin de gagner en privilèges.

Le contrôle d'accès obligatoire (MAC, *Mandatory Access Control*) a donc pour objectif de répondre à ce problème en imposant une politique de sécurité aux utilisateurs du système. Anderson propose l'utilisation d'un *moniteur de référence* [Anderson, 1980] afin de contrôler les interactions entre les sujets et les objets et de déterminer lesquelles sont valides (c'est-à-dire autorisées par la politique).

Cette section présente les principaux modèles de contrôle d'accès obligatoire. Ces modèles explicitent les concepts de propriétés de sécurité (intégrité, confidentialité) afin de pouvoir les appliquer.

Le modèle de **Bell-LaPadula** (BLP) [Bell et LaPadula, 1973] est basé sur les besoins de confidentialité des milieux militaires et a donc pour objectif d'empêcher les divulgations d'information. Ce modèle étend le modèle HRU en y ajoutant la notion de *label* associé à chaque sujet et objet du système. Un label correspond à un niveau de sécurité et est composé de deux identifiants de sécurité : le premier identifiant, hiérarchique, indique le niveau de classification (pour les objets) ou d'habilitation (pour les sujets), par exemple *secret* ou *top secret*. Le second identifiant, dit de catégorie, spécifie les organisations utilisant les informations, par exemple *militaire* ou *privé*.

En supplément des règles classiques (définies par une matrice de contrôle d'accès), deux nouvelles règles sont définies :

- *ss-property* (*simple security property*) : pour qu'un accès en lecture soit autorisé, le sujet le demandant doit avoir un niveau d'habilitation supérieur ou égal à celui de l'objet ;
- **-property* (*star property*) : l'information ne peut être transférée que depuis un objet de classification inférieure vers un objet de classification supérieure.

Ces deux règles permettent d'assurer la confidentialité de l'information et sa non divulgation. Cependant, l'existence de canaux cachés peut provoquer des flux d'information impossibles à contrôler. Pour cette raison, une version plus restrictive de BLP a été proposée avec les règles suivantes :

- *No Read Up* : un sujet demandant un accès en lecture à un objet doit avoir un niveau de sécurité supérieur ou égal à l'objet ;
- *No Write Down* : un sujet demandant un accès en écriture seule (ajout de données) sur un objet doit avoir un niveau de sécurité inférieur ou égal à l'objet.

Par conséquent, un sujet demandant un accès en lecture et écriture sur un objet doit avoir un niveau de sécurité identique à celui de l'objet. Ce modèle est parfois nommé MLS (*Multi-Level Security*), faisant ainsi référence au système de niveaux utilisé pour définir des règles de sécurité.

Un modèle dual à BLP, **Biba** [Biba, 1977] a été défini afin de répondre à des besoins d'intégrité.

Le modèle **DTE** (*Domain and Type Enforcement*) [Boebert et Kain, 1989] est un modèle de contrôle d'accès obligatoire de haut niveau. Contrairement à des modèles comme BLP ou Biba, il n'a pas pour objectif d'appliquer une propriété de sécurité spécifique, mais plutôt de définir les accès autorisés entre différentes entités du système. Cependant, ce modèle peut servir de base pour implémenter des modèles de propriétés de sécurité, tels que BLP, Biba ou le confinement de processus. Le modèle DTE remplace les notions de *sujets* et d'*objets* par celles de *domaines* et de *types*. Ainsi, chaque objet possède un type et chaque sujet s'exécute dans un domaine. Les droits d'accès sur les types et sur les domaines

sont alors définis pour chaque domaine. Ce modèle a donc pour objectif de restreindre les ressources accessibles par un processus (et ce même pour les processus privilégiés, en accord avec le principe de *moindre privilège*) et de contrôler quels processus ont accès aux ressources sensibles.

Les différents modèles de contrôle d'accès obligatoire peuvent être appliqués à la fois au niveau système, sur des environnements distribués (par exemple, dans le cas des infrastructures militaires), ou même être intégrés au sein d'un logiciel.

2.1.2.2 Langages d'expression de politiques de sécurité

Les modèles de contrôle d'accès qui ont été présentés sont les modèles historiques qui ont permis la définition de modèles et de langages d'expression de politiques de sécurité. Une politique de sécurité est composée d'un ensemble de règles qui peut avoir pour objectif d'appliquer des propriétés, éventuellement en suivant l'un des modèles décrits ou en réutilisant certains concepts. Dans cette section, nous décrivons donc les principaux modèles et langages d'expression de politiques.

RBAC RBAC [Ferraiolo et Kuhn, 1992] (*Role Based Access Control*) est un modèle de contrôle d'accès basé sur les rôles et ayant pour objectif de simplifier l'écriture et la gestion d'une politique de sécurité. En effet, l'administration d'une politique de contrôle d'accès obligatoire implique la gestion de multiples règles d'accès : ce processus est donc coûteux en temps et sujet à erreurs. Avec RBAC, les règles peuvent être simplifiées en les exprimant en fonction du rôle du sujet et non pas de son identité.

Ainsi, au sein d'une organisation, des rôles sont définis à partir des fonctions des différents postes et les permissions sont attribuées à ces rôles. Les utilisateurs ont des rôles qui leur permettent d'obtenir des permissions. Puisque les rôles ne sont pas directement assignés aux utilisateurs, leur gestion est facilitée (par exemple, lors de l'ajout d'un utilisateur, il est uniquement nécessaire de lui attribuer les rôles correspondants).

Différentes versions de RBAC ont été définies [Sandhu *et al.*, 1996, Sandhu *et al.*, 2000] :

- *core RBAC* comprend les concepts de base de RBAC et spécifie que les associations utilisateur-rôle et rôle-permission sont de types plusieurs-à-plusieurs ;
- *hierarchical RBAC* ajoute le support de l'héritage entre les rôles : par exemple un employé junior possède un rôle *junior* avec des permissions et le rôle *senior* hérite des permissions du rôle *junior* ;
- *constrained RBAC* introduit des contraintes afin d'appliquer le principe de séparation des privilèges.

Depuis 2004, RBAC est un standard NIST [Standard, 2004].

OrBAC OrBAC [Kalam *et al.*, 2003] (*Organization Based Access Control*) est un modèle de contrôle d'accès basé sur le concept d'organisation. OrBAC abstrait les notions de sujet, action et objet par celles de rôle (comme dans le cas de RBAC), activité et vue. Cela permet alors de regrouper des entités suivant les règles de sécurité qui les concernent, et donc de simplifier l'expression de la politique. Afin de faciliter l'expression de politiques dynamiques, OrBAC utilise des contextes. Trois types de règles sont possibles : les *permissions* (autorise un rôle à exécuter une activité sur une vue, dans un contexte donné), les *interdictions* (une activité est interdite) et les *obligations* (un rôle doit exécuter une activité sur une vue dans un contexte donné). Une règle OrBAC peut donc être exprimée de la façon suivante : un rôle peut avoir l'autorisation, l'interdiction ou l'obligation d'exécuter une activité sur une vue donnée lorsque le contexte associé est vérifié.

ABAC ABAC [Hu *et al.*, 2014] (*Attribute Based Access Control*) est un modèle de contrôle d'accès dans lequel des droits sont donnés aux utilisateurs en fonction d'attributs. Les attributs peuvent être vus comme des caractéristiques des éléments du système. Un attribut se compose d'un type (par exemple, un rôle, un projet, un niveau de sensibilité, etc.) et d'une valeur qui peut être mono-valuée ou multi-valuée (par exemple, un identifiant de rôle ou de projet, la valeur du niveau de sensibilité, etc.). Ces attributs peuvent être comparés entre eux ou à des valeurs fixées, ce qui permet donc de définir les règles d'une politique. Ainsi, une règle évalue un ou plusieurs attributs d'un sujet afin de décider d'autoriser ou d'interdire un accès sur un objet. Notons que ABAC peut être utilisé avec les modèles DAC, MAC et RBAC [Jin *et al.*, 2012].

XACML XACML [Moses *et al.*, 2005] (*eXtensible Access Control Markup Language*) est un standard du consortium OASIS [OASIS, 2015]. Il s'agit à la fois d'un langage de définition de politiques de contrôle d'accès et d'un modèle permettant d'interpréter (autoriser ou interdire) les requêtes d'accès en fonction de cette politique. XACML est basé sur le langage XML. Un ensemble de politiques XACML est constitué de politiques, elles-mêmes composées d'une *cible*, de *règles*, d'un *algorithme de combinaison des règles*, et d'*obligations*. L'élément cible indique si une politique doit être appliquée sur une requête donnée. Les règles spécifient les accès autorisés : elles se composent d'une *condition* (une expression booléenne) qui, si elle est vérifiée, entraîne un *effet* (autorisation ou interdiction) sur l'accès. L'algorithme de combinaison des règles définit comment la politique doit être interprétée lorsque plusieurs règles peuvent s'appliquer (par exemple, "l'autorisation est prioritaire sur l'interdiction"). Les obligations sont facultatives et permettent d'effectuer une action lorsqu'une règle est rencontrée (par exemple, générer une alerte). Cependant, en raison de la flexibilité et de l'expressivité d'XACML, la définition de politiques de sécurité peut se révéler complexe [Hu *et al.*, 2007]. XACML ne peut donc pas aisément permettre à l'utilisateur de l'informatique en nuage d'exprimer ses propres besoins de sécurité.

Ponder Ponder [Damianou *et al.*, 2000] est un langage de spécification de politiques de sécurité et d'administration pour les systèmes distribués. C'est un langage déclaratif et orienté objet qui supporte différents types de politiques : des politiques d'*autorisation* (spécifiant les accès autorisés ou interdits pour un sujet), d'*obligation* (les actions qu'un sujet doit effectuer en réponse à un événement), de *restriction* (les actions qu'un sujet ne doit pas effectuer), de *délégation* (les actions qu'un sujet peut déléguer à un autre sujet), de *contrainte* (pour limiter l'application d'autres politiques, par exemple en fonction du temps). Des *meta-politiques* peuvent également être définies pour gérer les interactions entre les politiques. Ponder2 [Twidle *et al.*, 2009] réutilise les concepts de Ponder en les adaptant aux systèmes autonomes. Ponder2 repose donc sur une architecture décentralisée, constituée de composants auto-gérés (SMC, *Self-Managed Cells*) capables d'interpréter et d'appliquer la politique sur les objets du système. Les objets considérés par Ponder2 sont des objets Java pouvant communiquer avec les SMC. Ces objets doivent donc être adaptés afin d'être gérés par Ponder2.

2.1.2.3 Langages d'expression de politiques d'assurance

Les politiques de sécurité peuvent être complétées par des politiques d'assurance. Une politique d'assurance doit exprimer des méthodes permettant d'évaluer le niveau de protection d'un système.

XCCDF XCCDF [Waltermire *et al.*, 2011b] (*Extensible Configuration Checklist Description Format*) est un standard, basé sur XML, permettant de spécifier des vérifications de sécurité et des tests de performances. Il s'agit donc d'un langage permettant de définir une politique d'assurance et de vérification. De plus, XCCDF peut être utilisé afin d'automatiser les vérifications de vulnérabilités et les réponses à apporter lorsque ces vulnérabilités sont détectées. XCCDF est utilisé par SCAP [Waltermire *et al.*, 2011a] (*Security Content Automation Protocol*), qui est un ensemble de spécifications définies par NIST afin de standardiser le format dans lequel sont exprimées les vulnérabilités logicielles et les configurations de sécurité. Une version distribuée de XCCDF, appelée DXCCDF [Barrère *et al.*, 2012a], permet d'exprimer des vulnérabilités distribuées.

OVAL OVAL [OVAL, 2014] (*Open Vulnerability and Assessment Language*) est un autre standard visant à unifier l'expression des politiques d'assurance. OVAL utilise XML pour évaluer l'état d'un système en effectuant une série de tests sur la machine. L'évaluation de l'état du système avec OVAL se décompose en trois étapes : la représentation des informations du système, la description des différents états et la transmission des résultats de l'évaluation. La version distribuée de OVAL est nommée DOVAL [Barrère *et al.*, 2012b] et peut gérer des vulnérabilités distribuées.

A-PPL A-PPL [Azraoui *et al.*, 2015] est un langage de politique permettant d'exprimer des obligations de responsabilité. A-PPL peut être utilisé pour des politiques de gestion de la vie privée et pour des politiques de contrôle d'accès et de contrôle d'usage. De plus, A-PPL gère des éléments spécifiques à l'assurance : il permet de définir des alertes en cas d'erreur détectée, des règles de localisation des données, mais également de spécifier quelles sont les caractéristiques voulues pour l'audit et le stockage des messages d'erreur.

2.1.2.4 Discussion

Dans cette section 2.1.2, nous avons présenté des modèles historiques pour des politiques de contrôle d'accès et des langages d'expression de politiques de sécurité et d'assurance.

Les modèles historiques de contrôle d'accès introduisent des concepts qui peuvent être généralisés à des politiques de sécurité non dédiées au contrôle d'accès. Par exemple, la modélisation sous forme de triplet sujet-permission-objet n'est pas nécessairement spécifique au contrôle d'accès. De plus, le contrôle d'accès a montré l'intérêt d'une politique de sécurité obligatoire par rapport à une politique discrétionnaire. En effet, une politique discrétionnaire ne permet pas de garantir des propriétés de sécurité 2.1.1 [Ferraiolo et Kuhn, 1992, Loscocco *et al.*, 1998].

Les modèles et langages de politiques existants permettent également d'introduire des notions essentielles à des langages d'expression des politiques de sécurité et d'assurance. Ainsi, la notion de rôle introduite dans RBAC est utilisée dans d'autres modèles de politiques de sécurité, comme OrBAC et XACML et permet de simplifier l'expression et l'administration d'une politique. Ces langages permettent donc la définition des besoins de sécurité sous forme de propriétés. Cependant, ces politiques sont souvent restreintes à l'expression d'une propriété ou ne supportent qu'un sous-ensemble de propriétés. De plus, les langages existants ne sont pour la plupart pas adaptés à la définition de propriétés de sécurité et d'assurance pour les environnements en nuage.

Ainsi, les concepts introduits dans cette section seront repris dans le langage proposé au chapitre 3. De plus, ils sont intégrés dans les multiples mécanismes de sécurité déjà existants (section 2.3). L'objectif du langage du chapitre 3 est donc de proposer un langage

intermédiaire pouvant s'adapter aux mécanismes existants, tout en réutilisant des concepts connus. Ce langage vise donc à pouvoir exprimer des propriétés de sécurité et d'assurance pour l'informatique en nuage.

2.2 L'informatique en nuage

Les propriétés de sécurité et les langages d'expression de politiques permettent de définir les besoins de sécurité d'un système. Dans cette thèse, nous cherchons à exprimer et appliquer ces propriétés dans le contexte de l'informatique en nuage.

Cette section introduit la notion d'informatique en nuage et présente des exemples d'environnements. De plus, nous décrivons des méthodes permettant d'exprimer les besoins de sécurité de l'informatique en nuage, ainsi que la notion de SLA, qui peut inclure ces besoins dans un contrat entre les différents intervenants.

2.2.1 Définition

L'informatique en nuage (*Cloud Computing*) est un modèle informatique désignant un réseau de ressources logicielles ou matérielles, accessibles depuis le réseau et fournies sous forme de service. Ainsi, l'informatique en nuage permet à des fournisseurs de proposer l'accès à des ressources (par exemple de la puissance de calcul ou de l'espace de stockage) à des utilisateurs. Par conséquent, l'utilisateur n'a plus besoin de posséder ses propres ressources mais peut bénéficier de l'infrastructure, des plateformes ou des logiciels des fournisseurs d'informatique en nuage.

De nombreuses définitions de l'informatique en nuage ont été données [Buyya *et al.*, 2008, Vaquero *et al.*, 2008, Armbrust *et al.*, 2010, Zhang *et al.*, 2010]. Nous nous basons ici sur celle donnée par le NIST (*National Institute of Standards and Technology*) qui regroupe la majorité des concepts impliqués (notamment les caractéristiques et modèles) dans les différentes définitions [Mell et Grance, 2011].

Définition 2.2.1: Informatique en nuage (NIST, [Mell et Grance, 2011])
Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction. This cloud model is composed of five essential characteristics, three service models, and four deployment models.
[L'informatique en nuage est un modèle permettant un accès réseau à la demande, pratique et ubiquitaire à un ensemble de ressources informatiques partagées et configurables (par exemple, du réseau, des serveurs, de l'espace de stockage, des applications ou des services) qui peuvent être allouées rapidement et libérées avec le minimum d'opérations de gestion et d'interaction avec le fournisseur de service. L'informatique en nuage se compose de cinq caractéristiques essentielles, trois modèles de service et quatre modèles de déploiement.]

2.2.1.1 Caractéristiques principales

Les cinq caractéristiques essentielles de l'informatique en nuage définies par le NIST sont :

- **Service à la demande** : un client peut unilatéralement allouer des ressources, telles que du temps de calcul ou de l'espace de stockage. Cette opération est faite automatiquement, sans nécessiter d'interactions humaines avec les fournisseurs.

– **Accès réseau** : les services sont disponibles sur le réseau et accessibles par des mécanismes standard permettant l'accès par des clients lourds ou légers et hétérogènes.
– **Partage des ressources** : les ressources informatiques du fournisseur sont mises en commun afin de répondre aux besoins de plusieurs clients en suivant un modèle multi-utilisateurs. Les ressources physiques et virtuelles sont allouées et ré-allouées dynamiquement en fonction des demandes des clients. Le client ne peut généralement ni contrôler, ni connaître la localisation exacte des ressources fournies, mais a parfois la possibilité de la spécifier à un niveau d'abstraction différent (pays, centre de données, etc.). Les ressources fournies sont par exemple de l'espace de stockage, de la puissance de calcul, de la mémoire ou de la bande passante.
– **Élasticité rapide** : les ressources fournies au client peuvent être ajustées automatiquement (en allouant ou libérant des ressources) afin de s'adapter rapidement à la demande. Du point de vue de l'utilisateur, les ressources disponibles apparaissent le plus souvent comme illimitées et pouvant être allouées à tout moment et en toute quantité.
– **Service mesuré** : l'utilisation des ressources est automatiquement contrôlée et optimisée en utilisant une métrique adaptée au type de service. Cette utilisation est communiquée au client et au fournisseur de manière transparente.

Ces cinq caractéristiques permettent de définir si un service fourni est, ou non, un service d'informatique en nuage. Les différents types d'informatique en nuage sont alors classifiés selon leur modèle de service et leur modèle de déploiement.

2.2.1.2 Modèles de service

L'un des principes fondamentaux de l'informatique en nuage étant la mise à disposition d'un service, les différentes infrastructures en nuage peuvent être classifiées selon le type de service fourni. Les trois principaux modèles de services considérés sont :

– **Software as a Service (SaaS)** : dans ce modèle, le fournisseur met à disposition du client des applications s'exécutant dans un nuage informatique. Les applications sont accessibles depuis divers clients via une interface Web. Le client ne contrôle pas l'infrastructure sous-jacente, notamment les serveurs, le système, le réseau ou le stockage (à l'exception possible de certains éléments de configuration de ses applications).
– **Platform as a Service (PaaS)** : le fournisseur met à disposition du client une plateforme où il peut déployer des applications utilisant les langages, librairies, services et outils du fournisseur. Le client ne contrôle pas l'infrastructure sous-jacente, notamment les serveurs, le système, le réseau ou le stockage, mais contrôle les applications déployées et éventuellement la configuration de l'environnement de l'application.
– **Infrastructure as a Service (IaaS)** : le fournisseur permet au client d'utiliser des ressources, telles que de la puissance de calcul, de l'espace de stockage, du réseau, ce qui lui permet de déployer et d'exécuter des applications (par exemple un système d'exploitation). Le client ne contrôle pas l'infrastructure sous-jacente, mais contrôle le système d'exploitation, le stockage, les applications déployées et éventuellement certains composants réseaux (pare-feux ou routeurs virtuels).

La figure 2.1 compare les trois modèles de services pour l'informatique en nuage. Elle présente également le modèle classique, sans utilisation d'une infrastructure en nuage.

Ce schéma différencie les parties de l'infrastructure gérées par le client de l'infrastructure en nuage (par exemple une entreprise) de celles gérées par le fournisseur. Dans le cas classique, l'entreprise gère l'ensemble des éléments, y compris le réseau, le stockage, le matériel et la couche de virtualisation. Dans le cas IaaS, ces quatre éléments sont gérés par le fournisseur de l'infrastructure, et le client ne s'occupe que des couches supérieures.

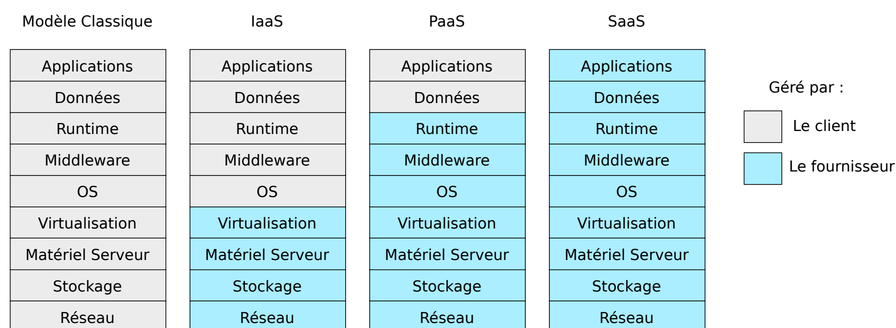

FIGURE 2.1 – Modèles de service

Dans une infrastructure de type PaaS, le fournisseur gère également l'OS, le middleware et l'environnement d'exécution. Enfin, dans le cas SaaS, le client cesse de gérer les données et les applications.

Ces trois types de services permettent donc de répondre à des besoins divers des clients, mais reposent tous sur le principe de la mutualisation des ressources.

2.2.1.3 Modèles de déploiement

Plusieurs méthodes permettent le déploiement des différents types de services en fonction des besoins de l'entreprise (par exemple, les besoins de sécurité). Quatre modèles de déploiement sont considérés dans la classification du NIST :

- **Privé** : l'infrastructure en nuage est utilisable par une seule organisation comprenant plusieurs utilisateurs. Elle peut appartenir ou être gérée par l'organisation, par une tierce partie ou par une combinaison de ces entités. Elle peut être localisée sur ou hors-site.
- **Communautaire** : l'infrastructure est utilisable par une communauté d'utilisateurs, appartenant à des organisations partageant des intérêts communs (objectifs, besoins de sécurité, etc.). Elle peut appartenir ou être gérée par une ou plusieurs organisations de la communauté, par une tierce partie ou par une combinaison de ces entités, et peut être localisée sur ou hors-site.
- **Public** : l'infrastructure est ouverte au public. Elle peut appartenir ou être gérée par une entreprise, une université, une organisation gouvernementale ou une combinaison de ces entités. Elle est localisée sur le site du fournisseur.
- **Hybride** : l'infrastructure est une combinaison d'au moins deux infrastructures en nuage distinctes (privée, communautaire, public). Elles demeurent des entités uniques mais interagissent par des technologies standard ou propriétaires, ce qui permet la portabilité des applications et des données.

2.2.2 Exemples d'infrastructures en nuage

La virtualisation est l'un des concepts techniques essentiels de l'informatique en nuage. Cette section vise donc à décrire la notion de virtualisation, présente dans tous les modèles de services, et à donner des exemples d'infrastructures en nuage.

2.2.2.1 Virtualisation

La notion de virtualisation [Goldberg, 1973, Smith et Nair, 2005a, Smith et Nair, 2005b] date des années 1960. La virtualisation correspond à la création d'une version virtuelle d'un

système ou d'un composant (processeur, mémoire, etc.). Elle permet par exemple d'exécuter plusieurs systèmes d'exploitation sur un seul serveur. Ainsi, la virtualisation peut être vue comme la division ou le partage des ressources physiques entre plusieurs composants. Cela permet alors de mieux utiliser les ressources physiques, puisqu'elles peuvent être mutualisées (augmentant ainsi leur taux d'utilisation), ce qui entraîne une réduction des coûts de l'infrastructure.

La figure 2.2 présente l'architecture simplifiée permettant la création de machines virtuelles (VM, *Virtual Machine*).

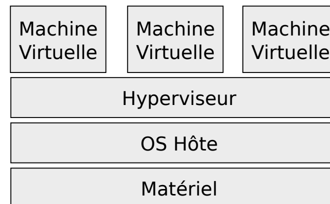

FIGURE 2.2 – Exemple d'architecture de virtualisation

Quatre couches sont représentées sur la figure 2.2 :
- Le **matériel** : il s'agit de l'ensemble des ressources physiques existantes ;
- L'**OS hôte** est responsable de la gestion des ressources physiques ;
- L'**hyperviseur** (VMM, *virtual machine monitor*) [Popek et Goldberg, 1974] gère l'accès aux ressources physiques, ainsi que leur partage entre des machines virtuelles en concurrence. Il peut être inclus dans l'OS hôte ;
- Les **machines virtuelles** sont des conteneurs, capables d'exécuter un système d'exploitation ou des applications, tout comme des machines physiques.

Plusieurs types de virtualisation existent :
- L'**émulation** permet de traduire le code assembleur invité en instruction pour l'hôte physique. Elle permet donc à un code invité de s'exécuter sur un système hôte différent. QEMU [Bellard, 2005] est un émulateur libre et open-source supportant de nombreuses architectures (ARM, x86, PowerPC, SPARC, etc.).
- La **virtualisation de niveau OS** correspond au cas où le noyau du système hôte est partagé par plusieurs espaces utilisateurs. Ces différents espaces utilisateurs, appelés conteneurs, semblent être, du point de vue de leurs utilisateurs, un système complet. Ce mécanisme de partage du noyau permet de réduire l'impact sur les performances dû à la virtualisation. Cependant, il réduit également la flexibilité (le même système est utilisé pour tous les conteneurs) et augmente les risques (une faille noyau impactera tous les conteneurs). Des exemples de virtualisation de niveau OS incluent LXC [Helsley, 2009], Docker [Merkel, 2014], OpenVZ [Kolyshkin, 2006], et les jails FreeBSD [Kamp et Watson, 2000].
- La **virtualisation complète** et la **paravirtualisation** consistent en la virtualisation d'un système complet [Sahoo *et al.*, 2010]. La paravirtualisation correspond au cas où le système invité a conscience d'être virtualisé, alors que, dans le cas d'une virtualisation complète, le système invité n'en a pas conscience et se comporte comme sur un hôte physique.

2.2.2.2 Exemples de conteneurs

LXC LXC [Helsley, 2009] est un environnement de virtualisation de niveau OS. LXC permet d'exécuter des environnements Linux dans des conteneurs isolés les uns des autres.

LXC repose notamment sur les fonctionnalités des cgroups du noyau Linux.

Docker Docker [Merkel, 2014] est une plateforme permettant de déployer automatiquement des applications dans des conteneurs. Une application peut donc être incluse dans un conteneur, ce qui améliore ainsi sa portabilité.

2.2.2.3 Exemples d'hyperviseurs

Nous présentons ici les principaux systèmes de virtualisation complète et de paravirtualisation, qui forment la base de l'informatique en nuage.

Xen Xen [Barham *et al.*, 2003] est un hyperviseur initialement développé par l'Université de Cambridge. Il s'agit actuellement d'un projet libre et open-source maintenu par la communauté. Le noyau Linux intègre par défaut le support de Xen depuis sa version 3.0. L'hyperviseur est responsable de la gestion de la mémoire et du temps processeur pour les différentes machines virtuelles (appelées *domaines*). Il exécute également un domaine privilégié (`dom0`), qui est une machine virtuelle chargée des tâches d'administration : `dom0` peut notamment créer des domaines non-privilégiés (`domU`) et contient les pilotes permettant d'interagir avec le matériel. Les domaines non-privilégiés `domU` peuvent être entièrement virtualisés ou paravirtualisés. Une version commerciale de Xen, XenServer, est également disponible [Citrix, 2015b].

VMware ESXi VMware ESXi [Chaubal, 2008] est un hyperviseur développé par VMware et utilisé par de nombreuses entreprises. L'architecture de VMware ESXi comprend la totalité du système sous-jacent qui permet d'exécuter les différents processus, notamment les applications de gestion et les machines virtuelles. VMware ESXi supporte de nombreux systèmes invités, notamment Linux, BSD, Windows et Mac OS X.

Microsoft Hyper-V Hyper-V [Velte et Velte, 2009], ou Windows Server Virtualization, est le premier hyperviseur basé sur un système Windows. Il était initialement disponible pour Windows Server 2008 et l'est maintenant pour les autres versions plus récentes de Windows Server. Les systèmes invités possibles incluent les différentes versions de Microsoft Windows, mais également des systèmes Linux. Hyper-V utilise des *partitions* pour isoler les machines virtuelles. La partition parente est celle ayant accès aux ressources matérielles et pouvant créer les partitions enfants dans lesquelles s'exécutent les systèmes invités. Les partitions enfants n'ont donc pas d'accès direct au matériel.

KVM KVM (*Kernel-based Virtual Machine*) [Kivity *et al.*, 2007, Habib, 2008] est un module noyau intégré par défaut au noyau Linux depuis la version 2.6.20. KVM permet à un programme en espace utilisateur d'utiliser les fonctions de virtualisation matérielle. Les machines virtuelles sont donc vues par le système hôte comme des processus classiques : elles peuvent bénéficier des fonctionnalités du système hôte, par exemple de ses mécanismes d'isolation entre les processus. De nombreux systèmes d'exploitation peuvent être utilisés pour les machines virtuelles, notamment différentes versions de Linux, BSD, OS X et Windows. Le projet KVM est basé sur QEMU et est une solution de virtualisation complète.

2.2.2.4 IaaS

L'infrastructure en temps que service permet à des entreprises de migrer leur infrastructure dans l'informatique en nuage tout en conservant un niveau de contrôle élevé

sur celle-ci. Un nuage informatique de type IaaS peut être déployé en interne en utilisant des logiciels spécifiques [Pepple, 2011, CloudStack, 2015, Moreno-Vozmediano *et al.*, 2012, Nurmi *et al.*, 2009] ou une infrastructure externe peut être utilisée.

Nous commençons par présenter les principaux logiciels permettant de mettre en place une infrastructure de type IaaS et ainsi de déployer à la demande des machines virtuelles.

OpenStack OpenStack [Pepple, 2011] est une plateforme libre et open-source permettant le déploiement d'une infrastructure d'informatique en nuage. OpenStack présente une architecture modulaire composée de plusieurs sous-projets, qui sont responsables de différents éléments (gestion du réseau, du stockage, de la puissance de calcul, etc.). L'utilisateur peut donc choisir quels composants installer, ce qui permet d'adapter une installation à des besoins spécifiques. Le projet OpenStack, initié par Rackspace et la NASA, est actuellement géré par la Fondation OpenStack, dont font partie de nombreuses entreprises, telles que Red Hat, IBM, VMware, Intel ou Oracle. OpenStack était initialement développé avec l'hyperviseur KVM, mais il supporte à présent Xen et, partiellement, Hyper-V et VMware ESXi. Actuellement, OpenStack est la plateforme de déploiement d'infrastructure IaaS la plus utilisée [Zenoss, 2014] et la plus mature. Il faut cependant noter que le déploiement et la configuration d'OpenStack peuvent se révéler complexes, notamment en raison de son architecture modulaire.

OpenNebula OpenNebula [Moreno-Vozmediano *et al.*, 2012] est une autre plateforme libre de mise en place d'une infrastructure IaaS, résultant d'un projet de recherche européen. OpenNebula est un projet monolithique, ce qui simplifie son administration. Cependant, OpenNebula ne dispose pas d'autant de fonctionnalités qu'OpenStack et est donc plus adapté à des déploiements d'infrastructures simples.

CloudStack Apache CloudStack [CloudStack, 2015] est un logiciel libre de mise en place d'infrastructure IaaS. CloudStack supporte les hyperviseurs les plus courants, comme KVM, XenServer, HYPER-V et VMware. Au contraire d'OpenStack, CloudStack présente une architecture monolithique, ce qui simplifie son déploiement mais diminue la flexibilité.

D'autres logiciels permettant la mise en place d'infrastructures IaaS existent, par exemple Eucalyptus [Nurmi *et al.*, 2009]. De plus, différents fournisseurs autorisent l'utilisation de leur propre infrastructure IaaS. Ces services sont généralement facturés en fonction de l'utilisation des ressources (quantité de ressources mobilisées, durée, etc.) [Armbrust *et al.*, 2010]. Nous décrivons ici les principaux fournisseurs IaaS.

Amazon EC2 Amazon EC2 (*Amazon Elastic Compute Cloud*) [Amazon, 2010] fait partie de l'ensemble de services AWS (*Amazon Web Services*) fournis par Amazon. La version de test d'Amazon EC2 a été annoncée en 2006 et sa première version stable en 2008, ce qui en fait le premier service d'infrastructure IaaS publique. L'hyperviseur utilisé par Amazon EC2 est Xen.

Microsoft Azure Virtual Machines Microsoft Azure (anciennement Windows Azure) [Microsoft, 2015] est la plateforme d'informatique en nuage de Microsoft. Microsoft Azure Virtual Machines est le service IaaS d'Azure et a été introduit en 2012. Microsoft Azure utilise son propre hyperviseur, *Microsoft Azure Hypervisor*, qui est une version modifiée d'Hyper-V.

Google Compute Engine Google Compute Engine (GCE) [Krishnan et Gonzalez, 2015] est le composant IaaS de la plateforme d'informatique en nuage *Google Cloud Platform* [Google, 2015]. GCE permet donc à des utilisateurs de créer des machines virtuelles à la demande. Compute Engine a été annoncé en 2012, tout comme Microsoft Azure. Il utilise KVM comme hyperviseur et peut donc fournir des VM Linux et Windows.

2.2.2.5 PaaS

Dans les cas des infrastructures en nuage de type Plateforme en tant que Service, le fournisseur met généralement à disposition du client un système d'exploitation, un environnement de développement, une base de données, etc. Nous présentons ici les principaux fournisseurs.

Microsoft Azure Cloud Services Microsoft Azure Cloud Services [Microsoft, 2015] est la partie PaaS de Microsoft Azure. Les langages officiellement supportés sont .NET, Java, Node.js, PHP, Python et Ruby, mais d'autres sont supportés grâce à des projets open-source. Puisque Microsoft Azure fournit à la fois des offres IaaS et PaaS, il est possible de combiner les composants des deux couches pour plus de flexibilité.

Google App Engine Google App Engine (GAE) [Zahariev, 2009] est le service PaaS de Google. GAE supporte les langages Python, Java, PHP et Go. GAE est gratuit jusqu'à un certain seuil d'utilisation des ressources. App Engine isole les différents environnements à l'aide de mécanismes de confinement.

Heroku Heroku [Salesforce, 2015b] a été créé en 2007 et était l'un des premiers services de type PaaS. Heroku ne supportait initialement que Ruby, mais intègre depuis de nombreux autres langages : Java, Node.js, Scala, Clojure, Python, PHP et Go. La plateforme fournit des environnements isolés, qui sont des conteneurs Unix virtualisés.

De nombreux autres vendeurs fournissent des solutions d'infrastructure en nuage de type PaaS [RedHat, 2015, Foundry, 2015].

2.2.2.6 SaaS

Le dernier modèle de service d'informatique en nuage est le Logiciel en tant que Service. L'objectif de SaaS est de mettre à disposition une application hébergée et gérée par son fournisseur, et accessible en utilisant un navigateur Web. De nombreuses applications sont possibles : les logiciels de messagerie et de bureautique [Conner, 2008, Murray, 2011], gestion de relation client [Salesforce, 2015a], visio-conférence [Citrix, 2015a], etc.

Google Apps Google Apps [Conner, 2008] est l'ensemble des services PaaS fournis par Google. Google Apps permet d'utiliser de manière collaborative des applications telles qu'une suite bureautique, un service de messagerie, un agenda, etc. Selon Google, plus de cinq millions d'entreprises utilisent Google Apps dans le monde.

Salesforce CRM Salesforce CRM [Salesforce, 2015a] est une solution de gestion de la relation client. Elle se sépare en plusieurs services responsables de différentes fonctionnalités, notamment Sales Cloud (gère les informations des contacts, est interfacé avec les réseaux sociaux, etc.), Service Cloud (permet des conversations par les réseaux sociaux, etc.) et Work.com (plateforme de management des ressources humaines).

Les services fournis par une infrastructure de type SaaS peuvent donc être très variés et répondent ainsi à de nombreux besoins des entreprises.

2.2.2.7 Synthèse

La table 2.1 synthétise les différents exemples d'environnements en nuage présentés dans cette section. De nombreux autres fournisseurs existent, mais ne sont pas présentés ici.

Fournisseurs	**Modèles de**		**Hyperviseurs**
	Service	**Déploiement**	
OpenStack	IaaS	Privé	KVM, Xen, Hyper-V, ESXi
OpenNebula	IaaS	Privé	KVM, Xen, ESXi
CloudStack	IaaS	Privé	KVM, Xen, Hyper-V, ESXi
Amazon	IaaS (EC2)	Public	Xen
Microsoft Azure	IaaS (Virtual Machines) PaaS (Cloud Services)	Public	Microsoft Azure Hypervisor
Google	IaaS (Compute Engine) PaaS (App Engine) SaaS (Apps)	Public	KVM
Heroku	PaaS	Public	-
Salesforce	SaaS	Public	-

Table 2.1 – Systèmes d'informatique en nuage

Dans la suite de cette thèse, nous utiliserons le logiciel OpenStack pour déployer notre infrastructure. En effet, nous choisissons un logiciel IaaS en déploiement privé afin de contrôler et d'être capable de sécuriser l'ensemble des couches (hôte, machines virtuelles, applicatif, etc.). De plus, OpenStack est plus complet que OpenNebula et CloudStack, son développement est plus actif et il est plus utilisé, ce qui en fait un choix plus approprié pour nos expérimentations.

2.2.3 Expression des besoins

L'informatique en nuage consiste à confier la gestion d'une partie de l'infrastructure à un tiers et à partager des ressources avec d'autres utilisateurs (par exemple, d'autres entreprises ou d'autres services d'une même entreprise). La sécurité de l'infrastructure et des données devient donc plus complexe à définir et à appliquer que dans le cas d'une infrastructure classique. De nombreux travaux cherchent donc à répondre à cette nouvelle difficulté.

2.2.3.1 Normes et standards

L'expression des besoins de sécurité est une tâche complexe sur un système classique, et l'est d'autant plus dans le cas d'un environnement hétérogène comme l'informatique en nuage. Plusieurs solutions permettent de simplifier cette expression pour des clients de l'informatique en nuage n'étant pas experts en sécurité [CSA, 2011].

Normes ISO/IEC L'ISO (*International Organization for Standardization*) et l'IEC (*International Electrotechnical Commission*) sont des organisations de normalisation internationales. Elles ont pour but de définir des standards dans des domaines industriels et commerciaux. La norme ISO 27001 [ISO, 2013a] est une norme portant sur les systèmes d'administration de la sécurité de l'information. Elle fournit une méthodologie pour sécuriser le système informatique d'une organisation. L'objectif est de protéger la confidentialité, l'intégrité et la disponibilité des données d'une organisation en déterminant les risques potentiels et en définissant les mesures à adopter pour faire face à ces risques. Le standard ISO 27002 [ISO, 2013b] décrit quant à lui les mesures techniques pouvant être utilisées par le responsable de la sécurité d'une organisation. Il s'agit d'un code de bonnes pratiques complémentaire de l'ISO 27001. La norme ISO 27017 [ISO, 2015] est actuellement en préparation. Elle aura pour objectif d'aider à la sécurisation d'une infrastructure en nuage, en fournissant des recommandations et des mesures spécifiques à la sécurité de ces environnements, en plus de celles fournies par l'ISO 27002.

Cloud Control Matrix CCM [CSA, 2015a] a deux objectifs principaux. Premièrement, CCM vise à proposer aux fournisseurs des services de sécurité fondamentaux. Deuxièmement, CCM veut aider les clients potentiels à évaluer le risque de sécurité global des différents fournisseurs. CCM est donc un framework détaillant les concepts de sécurité de l'informatique en nuage tels que définis par le CSA. Le CSA divise ces concepts en 14 domaines [CSA, 2011], couvrant notamment les préoccupations légales, l'administration des tâches d'audit, la sécurité des applications, la réponse aux incidents, la virtualisation, etc. Il a pour objectif d'être à la base d'un standard utilisable par les différents fournisseurs de services.

Consensus Assessments Initiative Questionnaire CAIQ [CSA, 2015b] est un ensemble de questions qu'un client ou un auditeur d'une infrastructure en nuage pourrait vouloir poser à un fournisseur. Il s'agit de questions fermées permettant d'évaluer quels sont les contrôles de sécurité offerts par le fournisseur du service. Ces questions sont basées sur les services de sécurité définis par CCM : CAIQ et CCM sont donc deux outils complémentaires, pouvant être utilisés ensemble. CAIQ peut ainsi permettre à un fournisseur de service de démontrer sa conformité avec les objectifs de sécurité du CSA. De plus, CAIQ peut servir de base à la définition d'objectifs de sécurité par le client.

Ainsi, plusieurs organisations ont proposé des standards fournissant une liste des besoins de sécurité classiques d'un environnement en nuage, mais également des méthodes pour déterminer quels besoins s'appliquent dans une situation donnée. Un utilisateur d'un service en nuage peut donc utiliser ces normes et méthodes pour déterminer l'ensemble de ses besoins de sécurité.

2.2.3.2 Service Level Agreement (SLA)

Le contrat de niveau de service (SLA, *Service Level Agreement*) est un contrat permettant au fournisseur d'un service et à un client de spécifier la qualité de service attendue [Wu et Buyya, 2012]. La qualité de service (QoS, *Quality of Service*) est le cœur d'un SLA. Il s'agit de la capacité du service à répondre aux exigences de l'utilisateur. Ces exigences varient selon les utilisateurs et peuvent concerner par exemple la disponibilité des ressources, les performances, la fiabilité ou le coût. La diversité des exigences de l'utilisateur est ce

qui nécessite l'établissement d'un SLA afin de fixer les termes du service fourni par l'opérateur de l'infrastructure en nuage. Le SLA spécifie également les limitations du service, autrement dit ce que le client ne peut pas exiger du fournisseur. Il permet donc d'établir les obligations des différents intervenants. De plus, le SLA spécifie les pénalités encourues en cas de violation de l'un des éléments de l'accord.

Plusieurs définitions des SLA ont été données [Wu et Buyya, 2012, Jin *et al.*, 2002, Wustenhoff et BluePrints, 2002, Badger *et al.*, 2012]. Nous utilisons ici la définition du NIST.

Définition 2.2.2: Service Level Agreement (NIST, [Badger *et al.*, 2012])
A document stating the technical performance promises made by the cloud provider, how disputes are to be discovered and handled, and any remedies for performance failures.
[Un document explicitant les engagements sur les performances techniques pris par le fournisseur de l'infrastructure en nuage, la manière donc les conflits sont découverts et gérés, et les réponses à apporter lorsque le fournisseur ne remplit pas ses engagements.]

Les SLA sont utilisés pour définir les engagements des deux parties impliquées. Ces engagements sont vérifiés en utilisant des métriques qui ont été définies et que le fournisseur de l'infrastructure et le client ont acceptés. Ces objectifs à atteindre sont appelés objectifs de niveau de service (SLO, *Service level objectives*). Pour pouvoir être inclus dans un SLA, un SLO doit posséder plusieurs caractéristiques [Sturm *et al.*, 2000] : il doit être atteignable, mesurable, compréhensible, significatif, contrôlable, abordable et mutuellement acceptable. Plusieurs langages et modèles de SLA existent [Ludwig *et al.*, 2003, Andrieux *et al.*, 2007, Kearney *et al.*, 2010] : ils diffèrent notamment selon les types de systèmes ou les objectifs visés.

WSLA WSLA [Ludwig *et al.*, 2003] (*Web Service Level Agreement*) est l'un des langages les plus utilisés pour les SLA visant les services Web. Le standard WSLA a été proposé par IBM et est basé sur XML. WSLA permet de spécifier les métriques associées aux applications d'un serveur Web, les objectifs désirés, et les actions à effectuer si les objectifs ne sont pas atteints.

SLA@SOI SLA@SOI [Kearney *et al.*, 2010] est un projet de recherche européen qui s'intéresse aux aspects multi-niveaux et multi-fournisseurs des SLA au sein des infrastructures en nuage. SLA@SOI propose SLA*, un langage abstrait et extensible pour SLA, inspiré de WS-Agreement [Andrieux *et al.*, 2007]. SLA* simplifie la formalisation des SLA en offrant la possibilité d'utiliser d'autres langages que le XML (BNF, Java). Dans SLA@SOI, un SLA est vu comme l'instance d'un modèle de SLA. Un modèle de SLA contient donc la liste des entités impliquées et la liste des termes de l'accord. L'extensibilité et l'adaptabilité résultant de l'utilisation de modèles de SLA simplifient donc la définition des SLA pour les environnements hétérogènes tels que les infrastructures en nuage.

Security SLA Security SLA [Bernsmed *et al.*, 2011] est un framework qui permet de répondre à l'un des manques essentiels des SLA vus précédemment : la prise en compte de la sécurité. Le framework inclut un cycle de vie pour les SLA composé des étapes de publication, négociation, engagement, allocation, surveillance et fin de vie. La phase de négociation fait intervenir trois entités : le client ayant des besoins de sécurité, un fournisseur de service avec une offre de sécurité, et une tierce partie visant à répondre aux besoins de sécurité grâce à l'offre du fournisseur.

Merkat Merkat [Costache *et al.*, 2013] est une plateforme d'informatique en nuage capable de partager les ressources entre les différentes applications. De plus, Merkat prend en compte les objectifs de performances des utilisateurs, exprimés sous forme de SLO, pour adapter les ressources disponibles pour l'application. Ces SLO sont exprimés par application, grâce à l'utilisation d'environnements virtuels autonomes qui peuvent être re-dimensionnés selon les objectifs. Le support des SLA étant directement intégré dans Merkat, il est plus étendu que sur d'autres plateformes.

Les SLA et les plateformes les utilisant permettent donc aux utilisateurs et aux fournisseurs de services en nuage de s'accorder sur les conditions de délivrance du service. Il s'agit donc d'un outil essentiel à l'adoption de services en nuage. Cependant, les SLA n'ont pas pour objectif d'appliquer les besoins de sécurité exprimés par l'utilisateur, même si certains prennent en compte la sécurité lors de leur définition.

2.2.4 Discussion

Cette section a donc décrit les éléments définissant l'informatique en nuage, c'est-à-dire ses principales caractéristiques et les différents modèles de services et de déploiement qui peuvent être utilisés pour mettre en place une telle infrastructure.

Nous avons également introduit des méthodes permettant à un utilisateur de l'informatique en nuage de déterminer l'ensemble des besoins de sécurité de son infrastructure logicielle. Ces méthodes utilisent par exemple les normes ISO ayant été établies pour améliorer la sécurité des systèmes d'information, ou des outils dédiés à la définition des besoins. Par conséquent, un utilisateur de l'informatique en nuage n'étant pas un expert en sécurité peut déterminer les besoins de sécurité de son architecture en utilisant les méthodes présentées ici. Les besoins ainsi obtenus servent de base à la définition d'une politique de sécurité (voir section 2.1).

Les SLA ont également été présentés. Ils permettent à un utilisateur de l'informatique en nuage et à un fournisseur de service de formaliser la qualité de service attendue. De plus, certains SLA peuvent inclure les besoins de sécurité de l'utilisateur que le fournisseur s'engage à appliquer. Ainsi, des besoins décrits grâce aux méthodes d'expression des besoins peuvent être intégrés dans des SLA si le langage d'expression de la sécurité est compatible avec le SLA considéré.

2.3 Mécanismes et propriétés de sécurité

Les politiques de sécurité décrites à la section 2.1 ont pour objectif d'exprimer des besoins de sécurité. De nombreux mécanismes existent et sont capables d'appliquer les propriétés de sécurité sur certaines ressources. Ainsi, des attaques visant différents niveaux, comme une application (ShellShock [NIST, 2014]) ou la couche de virtualisation (Venom [NIST, 2015]), peuvent être adressées par ces mécanismes [Moore, 2015]. Cette section décrit donc une sélection de mécanismes pouvant répondre aux besoins de sécurité aux niveaux système, réseau, matériel, logiciel et virtualisation, qui correspondent aux différentes couches d'un système d'informatique en nuage. Puis, nous donnons une association de ces mécanismes aux propriétés de sécurité.

2.3.1 Système

Droits DAC Unix Les droits d'accès DAC Unix forment le système de contrôle d'accès discrétionnaire utilisé par défaut par les systèmes de type Unix. Ils permettent au pro-

priétaire d'une ressource de fixer les droits de lecture, d'écriture et d'exécution sur cette ressource, pour lui-même, le groupe auquel appartient la ressource, et les autres utilisateurs du système. Ces permissions peuvent être modifiées par le propriétaire de la ressource. Dans le cas des droits d'accès DAC Unix, l'administrateur du système (*root*) possède tous les droits. Ainsi, si un attaquant parvient à obtenir l'identité *root* (par exemple en exploitant une faille dans une application), il obtiendra l'ensemble de ses droits, ce qui compromet la sécurité de l'ensemble du système.

SELinux SELinux [Smalley *et al.*, 2001, McCarty, 2004] est un mécanisme de contrôle d'accès obligatoire open-source, initialement développé par la NSA. Il s'agit d'un module de sécurité Linux (LSM, *Linux Security Module*) qui implémente un contrôle d'accès utilisant des types (TE), la gestion des rôles (RBAC), la sécurité multi-niveaux (MLS) et multi-catégories (MCS). SELinux identifie les ressources actives (les sujets) et passives (les objets) du système en utilisant des labels (contextes de sécurité) et repose sur l'architecture FLASK [Spencer *et al.*, 1999] (*Flux Advanced Security Kernel*). Une fois les ressources identifiées, SELinux peut contrôler (autoriser ou interdire) les accès entre les contextes de sécurité (entre deux sujets, ou entre un sujet et un objet). L'ensemble des règles contrôlant les accès entre les ressources forme la politique SELinux. La politique est formée d'un ensemble de modules, où chaque module est responsable d'une application donnée. Ces modules peuvent être ajoutés et retirés dynamiquement, permettant donc d'adapter la politique à de nouvelles applications ou de nouveaux besoins de sécurité.

PIGA PIGA [Briffaut, 2007] (*Policy Interaction Graph Analysis*) est un mécanisme de contrôle d'accès reposant sur les fonctionnalités de SELinux. PIGA permet non seulement de contrôler les accès entre deux sujets ou entre un sujet et un objet, mais également de contrôler des ensembles d'opérations, par exemple une séquence d'accès et les flux d'informations. PIGA utilise son propre langage, SPL (*Security Property Language*), qui permet de définir des propriétés systèmes de haut niveau, telle que la confidentialité ou l'intégrité.

PAM Linux PAM [Samar, 1996] (*Linux Pluggable Authentication Modules*) fournit le support de l'authentification et la gestion des paramètres d'authentification pour les applications d'un système Linux. PAM offre donc un ensemble de bibliothèques pouvant être utilisées par les applications et les services pour gérer le processus d'authentification.

De nombreux autres mécanismes de sécurité système existent, à la fois pour le contrôle d'accès [Spengler, 2002, Harada *et al.*, 2004, Bauer, 2006, Schaufler, 2008] et pour l'authentification [Neuman et Ts' O, 1994]. Nous ne les détaillons pas ici puisqu'ils ne sont pas utilisés par la suite, mais ils peuvent également répondre à des besoins de sécurité.

2.3.2 Réseau

iptables Iptables [Andreasson, 2001] est un pare-feu pour les systèmes Linux. Il s'agit d'un logiciel s'exécutant en espace utilisateur et permettant de configurer les règles du pare-feu du noyau Linux (composé de modules Netfilter [Russell et Welte, 2002]). Iptables peut donc établir, mettre à jour et inspecter les tables de filtrage de paquets dans le noyau Linux. Iptables propose un langage permettant d'exprimer des règles de contrôle des communications réseaux en fonction de différents paramètres, tels que les adresses IP, les ports, etc.

firewalld Firewalld [RedHat, 2015] est un logiciel de pare-feu pour les distributions RedHat (RHEL, CentOS, Fedora), qui remplace l'interface iptables et configure les règles du pare-feu du noyau. Une différence essentielle entre Firewalld et iptables est la gestion de la mise à jour des règles. Dans le cas d'iptables, chaque changement nécessite la mise à jour de l'ensemble des règles (les règles sont déchargées, puis rechargées). En revanche, dans le cas de Firewalld, seules les différences sont appliquées.

OpenSSH OpenSSH [OpenSSH, 2014] est un ensemble d'outils permettant de sécuriser des connexions réseaux en utilisant le protocole SSH [Ylonen et Lonvick, 2006] (*Secure Shell*). Il s'agit de l'implémentation de SSH la plus utilisée. OpenSSH inclut de nombreux outils, notamment `ssh` (connexion distante), `sshd` (seveur SSH), `scp` (copie distante), `ssh-keygen` (génération de clefs), etc. Les utilisateurs peuvent être identifiés par différentes méthodes, par exemple un mot de passe, des clefs asymétriques ou le protocole Kerberos [Neuman et Ts' O, 1994].

OpenVPN OpenVPN [Feilner, 2006] est un logiciel permettant d'établir des réseaux privés virtuels (VPN, *Virtual Private Networks*), sécurisant ainsi des connexions réseaux. OpenVPN s'appuie sur la bibliothèque OpenSSL [Viega *et al.*, 2002]. OpenVPN permet à ses utilisateurs de s'authentifier en utilisant des clefs partagées, des certificats ou des mots de passe. De plus, un serveur OpenVPN est capable de gérer plusieurs clients : il génère alors un certificat par client, signé par une autorité de certification.

2.3.3 Éléments matériels

Secure Element Un SE [GlobalPlatform, 2011] (*Secure Element*) est un composant physique et résistant aux attaques qui peut héberger des applications de manière sécurisée, stocker des données confidentielles et gérer des clefs de chiffrement. Ces différents éléments sont sécurisés en fonction de règles définies par une autorité de confiance. La norme suivie par les SE est définie par les membres de l'association GlobalPlatform qui regroupe de nombreuses entreprises (Gemalto, Oracle, etc.).

TPM Un TPM [Bajikar, 2002, Tomlinson, 2008] (*Trusted Platform Module*) est un composant matériel proposant des fonctionnalités cryptographiques. Les applications peuvent utiliser un TPM afin d'authentifier un composant physique : chaque TPM dispose d'une clef RSA unique non-modifiable, ce qui permet d'authentifier la plateforme. Les TPM peuvent être virtualisés [Perez *et al.*, 2006] afin de permettre leur utilisation sur des systèmes ne disposant pas de TPM physique.

Intel TXT Intel TXT [Greene, 2012] (*Intel Trusted Execution Technology*) est une technologie permettant de vérifier l'authenticité d'une plateforme et de son système d'exploitation, de vérifier que l'environnement d'exécution est sûr, et qui fournit à un système ayant été vérifié des fonctionnalités supplémentaires. Intel TXT utilise un TPM afin de fournir ses fonctionnalités.

2.3.4 Protection de logiciels

ModSecurity ModSecurity [Ristic, 2010] est un pare-feu applicatif open-source. Il permet de filtrer les requêtes sur un serveur Web Apache (ModSecurity peut également être

utilisé avec les serveurs Nginx et IIS). ModSecurity possède de nombreuses fonctionnalités et peut notamment conserver l'historique du trafic HTTP, sécuriser une application Web, contrôler les accès en filtrant les requêtes entrantes, etc. ModSecurity est implémenté sous la forme d'un module Apache et ses règles sont définies en utilisant des expressions régulières.

PaX PaX [PaX Team, 2003] est un patch de sécurité du noyau Linux permettant de protéger les pages mémoires. PaX empêche qu'un segment de mémoire puisse être à la fois écrit et exécuté, ce qui permet par exemple de limiter les dépassements de tampons. PaX peut également rendre aléatoire certaines adresses de l'espace mémoire (ASLR, *Address space layout randomization*). Cela permet d'éviter l'exploitation de certaines failles de sécurité.

SEJava SEJava [Venelle *et al.*, 2013] (*Security Enhanced Java*) est un mécanisme de contrôle d'accès obligatoire pour les applications Java. SEJava permet de contrôler les interactions (appel de méthode, accès aux attributs, etc.) entre les objets d'une application en utilisant des concepts similaires à ceux de SELinux, et donc sans modifier les applications. SEJava protège les applications s'exécutant dans la machine virtuelle OpenJDK. SEDalvik [Bousquet *et al.*, 2013] adapte SEJava à la machine virtuelle Dalvik d'Android, permettant ainsi de protéger les applications Android.

2.3.5 Virtualisation et informatique en nuage

sVirt sVirt [Morris, 2009] est un mécanisme de contrôle d'accès obligatoire pour les machines virtuelles. sVirt a donc pour objectif d'isoler les machines virtuelles en utilisant une politique de contrôle d'accès obligatoire. Il est ainsi possible de limiter l'exploitation des failles pouvant être présentes dans l'hyperviseur utilisé. sVirt repose sur l'utilisation de SELinux [McCarty, 2004]. Chaque machine virtuelle est associée à un contexte de sécurité et s'exécute donc dans son propre domaine, isolé des autres VM et du reste du système.

CloudSec CloudSec [Ibrahim *et al.*, 2011] est un outil permettant de sécuriser les machines virtuelles utilisées dans une infrastructure de type IaaS. CloudSec utilise l'introspection de VM afin de surveiller la mémoire physique utilisée par les machines virtuelles, sans que celles-ci soient modifiées. CloudSec est actuellement compatible avec l'hyperviseur VMWare ESXi. Il utilise l'API offerte par VMWare afin d'intercepter les accès des VM à la mémoire physique et au processeur. Cela permet à CloudSec de détecter la présence de rootkits, même s'il ne permet pas actuellement de s'en protéger.

CloudAudit CloudAudit [Hoff *et al.*, 2010] est un standard proposant une interface permettant à des fournisseurs de services en nuage d'offrir à leurs utilisateurs un processus automatique d'audit et d'assurance. De nombreuses entreprises participent à l'élaboration de ce standard, notamment Microsoft, Google et Rackspace. CloudAudit propose également un ensemble de mécanismes permettant d'automatiser la récupération des informations (de sécurité, d'assurance, etc.). CloudAudit peut être utilisé pour les trois modèles de services d'informatique en nuage (IaaS, PaaS et SaaS). Ce standard vise à rester simple et à pouvoir être étendu.

OSSEC OSSEC [Bray *et al.*, 2008] est un HIDS (*Host-based Intrusion Detection System*) open-source qui peut analyser des logs, vérifier l'intégrité des fichiers, détecter des rootkits, générer des alertes et tenter d'y répondre. OSSEC utilise une architecture centralisée, où

des agents doivent être déployés sur les machines à sécuriser. Cela permet ainsi de gérer les agents depuis un unique serveur. Des agents sont disponibles pour de nombreux systèmes d'exploitation, notamment Linux et Windows.

VESPA VESPA [Wailly *et al.*, 2012] est une architecture autonome visant à protéger les infrastructures en nuage des attaques. VESPA utilise une architecture à plusieurs plans pour les ressources, les éléments de sécurité (pare-feu, etc.), les agents et les gestionnaires autonomes. VESPA protège les ressources d'une infrastructure IaaS en utilisant plusieurs boucles de contrôle, autonomes et coordonnées. Chacune de ces boucles se concentre donc sur la protection d'une couche spécifique de l'infrastructure, ce qui permet d'obtenir un système flexible d'auto-protection des ressources. VESPA se concentre sur la réaction aux attaques : lorsqu'une attaque est détectée, VESPA tente d'y apporter une réponse en fonction de ses connaissances et du type de l'attaque. Par exemple, si un virus est détecté lors d'un téléchargement sur une VM, plusieurs réactions peuvent être choisies : le pare-feu peut bloquer les communications réseaux sur cette VM, la VM peut être migrée sur un hôte dédié afin d'être isolée, ou un message peut être envoyé à un anti-virus. Il s'agit par conséquent d'une solution complémentaire aux travaux présentés dans cette thèse, qui se focalisent sur la configuration de la sécurité afin de limiter la surface d'attaque.

MEERKATS MEERKATS [Keromytis *et al.*, 2012] est une architecture autonome dédiée à la sécurité du Cloud. MEERKATS se concentre sur l'amélioration de la résistance des éléments critiques de l'application : pour cela, MEERKATS s'enrichit à partir des attaques passées et veut ainsi anticiper les menaces futures. MEERKATS est constitué de plusieurs composants qui visent à répondre à différents types d'attaques, en utilisant des mécanismes de protection variés. Ainsi, ces composants peuvent injecter des informations pour confondre l'attaquant, tenter d'apprendre quels sont les comportements normaux et anormaux du système, migrer rapidement des processus afin de contrer des attaques, etc. Cependant, MEERKATS ne dispose pas d'un langage permettant d'exprimer les besoins de sécurité de l'infrastructure à protéger. De plus, les mécanismes de protection utilisés sont spécifiques à MEERKATS, ce qui limite les types d'attaques pouvant être adressés.

A4Cloud A4Cloud [Pearson *et al.*, 2012] est une solution se focalisant sur la responsabilité des intervenants. A4Cloud utilise le langage d'expression de politiques de responsabilité, A-PPL [Azraoui *et al.*, 2015]. En effet, la responsabilité des intervenants est un élément essentiel pour obtenir un contrôle des données personnelles et de l'entreprise. L'objectif de A4Cloud est donc d'aider à définir les responsabilités du fournisseur de service vis-à-vis de la gestion des données personnelles et confidentielles. Cela est fait en utilisant des mécanismes de prévention (limiter un risque), de détection (surveiller les violations de la politique) et de correction (réagir suite à un incident). Cependant, A4Cloud vise uniquement les problèmes liés aux données personnelles, mais n'adresse pas les autres propriétés de sécurité, telles que l'intégrité ou l'isolation.

SAaaS SAaaS [Doelitzscher *et al.*, 2012] (*Security Audit as a Service*). SAaaS est un système multi-agents qui permet d'auditer des machines virtuelles dans un environnement en nuage et qui suit une architecture autonome. SAaaS peut être décomposé en trois couches. La première est la couche d'entrée qui reçoit des informations envoyées par les différents agents lorsque ceux-ci détectent des activités anormales. Puis, la couche de traitement reçoit les informations depuis la couche d'entrée et les traite en fonction des politiques (définies dans la troisième couche). Enfin, la couche de présentation est l'unique point

d'interaction pour les utilisateurs du service : elle permet de visualiser l'état des VM et les éventuelles anomalies, mais également de définir les politiques de traitements. SAaaS n'adresse cependant pas l'application des propriétés de sécurité. Il n'y a donc pas de lien direct entre l'application et l'assurance des propriétés, ce qui est nécessaire pour assurer les besoins de l'utilisateur de l'infrastructure en nuage.

2.3.6 Synthèse

Les mécanismes présentés dans cette section permettent donc de sécuriser les différents niveaux impliqués dans une infrastructure en nuage, et ce quel que soit le modèle de service considéré (IaaS, PaaS ou SaaS). Chaque mécanisme couvre un sous-ensemble de propriétés pour certains types de ressources (système, réseaux, etc.). Ainsi, en combinant ces différents mécanismes, il est possible de couvrir un large spectre de propriétés. La table 2.2 regroupe les différents mécanismes vus précédemment et donne, pour chacun d'entre eux, les propriétés de sécurité (section 2.1.1) qu'ils peuvent couvrir. Il s'agit ici de propriétés abstraites correspondant à celles de la section 2.1.1. Par exemple, même si SELinux et un SE sont tous deux capables d'appliquer la propriété de confidentialité, celle-ci ne vise pas les mêmes ressources et n'utilise pas les mêmes méthodes. C'est pourquoi ces propriétés seront précisées, pour chaque mécanisme utilisé, au chapitre 3.

Type	**Mécanisme**	**Propriétés**						
		Conf.	**Inté.**	**Dispo.**	**Assur.**	**Confin.**	**Auth.**	**Autres**
Système	SELinux	X	X			X		
	DAC	X	X			X		
	PIGA	X	X			X		
	PAM						X	
Réseau	iptables			X				
	firewalld			X				
	OpenVPN	X					X	
	OpenSSH	X					X	
Matériel	SE	X	X			X	X	
	TPM	X	X			X	X	
	Intel TXT	X	X			X	X	
Logiciel	SEJava	X	X			X		
	ModSecurity	X	X	X				
	PaX		X					
Cloud	sVirt	X	X			X		
	CloudSec	X	X					
	CloudAudit				X			
	OSSEC		X		X			X
	VESPA	X	X	X		X		X
	MEERKATS	X	X	X				X
	A4Cloud	X			X			
	SAaaS				X			
Assurance	Oscap				X			

TABLE 2.2 – Propriétés de sécurité couvertes par les mécanismes

On peut donc observer que ces mécanismes permettent de répondre à des besoins de sécurité variés. Cependant, aucun mécanisme n'est capable d'appliquer toutes les propriétés de sécurité sur tous les niveaux d'une infrastructure en nuage. De plus, l'application d'une même propriété est différente selon le niveau de l'application : ainsi une même propriété

n'a pas le même effet si elle appliquée sur des ressources systèmes, réseaux, ou sur une application. Par conséquent, aucun de ces mécanismes de sécurité n'est capable de sécuriser à lui seul une infrastructure en nuage. En revanche, une utilisation combinée de ces mécanismes permettrait de répondre à de nombreux besoins de sécurité s'appliquant à des niveaux différents. Ces mécanismes peuvent donc être configurés individuellement afin de protéger globalement le système. Cependant, configurer les mécanismes séparément pour répondre à un besoin global est complexe. C'est pourquoi il est nécessaire de disposer d'une méthode permettant de simplifier la configuration des mécanismes à partir de besoins de sécurité globaux et de vérifier que cette application est opérationnelle : c'est donc ce qui constitue l'objectif principal de cette thèse.

2.4 Conclusion

L'informatique en nuage est un type d'environnement hétérogène et dynamique de plus en plus utilisé. Différents modèles de service et de déploiement existent et permettent de répondre à des besoins variés des utilisateurs. La combinaison d'un environnement hétérogène et des nombreuses applications utilisateurs fait de la sécurité un point essentiel mais complexe à adresser. La définition des besoins de sécurité peut en effet être une tâche difficile, d'autant plus que l'utilisateur du service ne connait pas nécessairement ces besoins. Cependant, nous avons vu qu'il existe des méthodes d'analyse des risques qui permettent à un utilisateur de déterminer ses besoins de sécurité. Ainsi, dans la suite de ce document, nous considèrerons que l'utilisateur du service est capable d'établir la liste de ses besoins, si nécessaire en utilisant l'une de ces méthodes.

Lorsque l'utilisateur du service a déterminé ses besoins de sécurité, un langage doit permettre d'exprimer ces besoins. Nous avons donc vu quelles sont les différentes propriétés de sécurité fondamentales qui peuvent correspondre aux besoins d'un utilisateur, ainsi que plusieurs modèles de politiques permettant d'exprimer ces besoins. Cependant, aucun langage ne permet actuellement d'exprimer une politique de sécurité qui soit indépendante des mécanismes sous-jacents et du système (puisqu'il s'agit d'un environnement hétérogène), qui couvre l'ensemble des couches d'une infrastructure en nuage et qui permette d'exprimer à la fois les besoins d'application de la sécurité et les besoins d'assurance.

Dans les chapitres qui suivent, nous proposons donc un langage répondant à ces différents manques, associé à une architecture capable d'interpréter ce langage. Ainsi, le langage proposé devra permettre de définir une politique de sécurité et d'assurance qui soit adaptée aux environnements en nuage : cette politique devra donc être aussi indépendante que possible du système et des mécanismes sous-jacents. L'architecture devra quant à elle être capable d'appliquer cette politique en configurant les mécanismes de sécurité présents. En effet, l'objectif de l'architecture n'est pas de proposer un nouveau mécanisme de sécurité, mais de réutiliser les fonctionnalités des nombreux mécanismes existants pour couvrir un large panel de besoins. De plus, l'architecture devra être capable de s'adapter aux mécanismes disponibles pour la projection des propriétés et de reconfigurer les mécanismes en cas de défaillance de l'un d'entre eux.

Chapitre 3

Langage

L'informatique en nuage est de plus en plus utilisée, à la fois par les particuliers et par les entreprises. Ces dernières l'utilisent dans plusieurs buts, par exemple pour stocker des données ou pour externaliser une partie de l'architecture logicielle de l'entreprise. Dans ce cas, l'administrateur de l'architecture logicielle doit pouvoir exprimer ses besoins de sécurité, c'est-à-dire définir quelles sont les ressources à protéger et quelles sont les actions autorisées ou interdites sur ces ressources. Un besoin de sécurité peut par exemple être la nécessité de garantir la confidentialité d'un fichier, autrement dit de contrôler quels utilisateurs peuvent accéder à l'information de ce fichier, ou la confidentialité d'une connexion entre deux services distants, c'est-à-dire s'assurer que l'information échangée entre ces deux services ne peut pas être lue par une entité tierce.

L'une des caractéristiques de l'informatique en nuage est l'hétérogénéité à la fois des systèmes d'exploitation utilisés, mais aussi des services ou applications déployés. L'expression des besoins de sécurité est donc complexe et doit être adressée par un langage adapté à de tels systèmes.

L'informatique en nuage est également caractérisée par son aspect dynamique, puisque des machines virtuelles peuvent être déployées, migrées ou arrêtées à tout moment. Cependant, les besoins de sécurité d'une machine virtuelle donnée évoluent peu au cours de la vie de cette machine. En effet, les machines virtuelles sont généralement dédiées à un service (par exemple, une plateforme de création de sites Web ou un serveur de partage de données). De plus, si un nouveau service doit être mis en place, la création d'une nouvelle machine virtuelle sera préférée à la modification d'une machine existante (par exemple, la modification d'une machine virtuelle servant de serveur Web en serveur de partage de fichier). Ainsi, nous choisissons de ne pas prendre en compte ce côté dynamique du nuage informatique lors de l'expression des besoins de sécurité.

L'environnement étant hétérogène, les ressources à protéger et les mécanismes utilisés diffèrent selon les systèmes. Or, il n'existe pas actuellement de langage adapté aux environnements en nuage et permettant d'exprimer des besoins de sécurité (section 2.1). L'objectif de ce chapitre est donc de définir un langage permettant d'abstraire ces éléments afin d'obtenir une politique indépendante du système et des mécanismes. Cela permet alors à une même politique d'être portable et donc d'être appliquée sur plusieurs systèmes.

L'informatique en nuage étant un environnement en constante évolution, il est indispensable que le langage soit extensible. En effet, il doit permettre de répondre à de nouveaux besoins de sécurité, par exemple des besoins spécifiques à un cas d'usage précis.

Dans le langage défini dans ce chapitre, les besoins de sécurité sont exprimés en utilisant des propriétés. Le langage doit donc être générique, afin qu'une même propriété puisse être appliquée et assurée différemment selon les ressources mises en jeu. De plus, le langage doit

être simple et offrir à l'utilisateur la possibilité de définir facilement ses besoins. Finalement, le langage doit permettre d'adresser un maximum de besoins de sécurité.

Dans ce chapitre, nous verrons tout d'abord quelles doivent être les caractéristiques d'un langage d'expression des besoins de sécurité pour l'informatique en nuage (section 3.1). Puis, nous donnerons une vue globale des différents éléments du langage et de leurs interactions (section 3.2). Les sections 3.3, 3.4 et 3.5 détailleront les éléments du langage, ce qui permettra de définir une politique de sécurité (section 3.6). Enfin, la section 3.7 conclura ce chapitre.

3.1 Caractéristiques du langage

Un langage d'expression des besoins de sécurité pouvant être utilisé dans un environnement hétérogène et dynamique doit posséder des caractéristiques spécifiques. Cette section détaille les besoins auxquels notre langage doit répondre.

En premier lieu, un environnement hétérogène implique que les différents nœuds peuvent utiliser des systèmes d'exploitation différents. Par conséquent, les ressources peuvent suivre des conventions de nommage différentes et les mécanismes de sécurité ne sont pas nécessairement les mêmes. Il est donc nécessaire que le langage puisse s'abstraire de ces deux éléments.

Sur un système, les ressources sont associées à un identifiant (leur nom ou leur chemin). Cet identifiant dépendant du système sur lequel la politique est déployée, il ne peut pas être utilisé pour identifier les ressources dans notre langage. En conséquence, le langage utilise la notion de *contextes* afin de s'abstraire du système de nommage des ressources systèmes (section 3.3), ce qui permet au langage d'être **indépendant des ressources**.

De plus, les mécanismes de sécurité utilisés pour répondre aux besoins de sécurité ne sont pas identiques selon les systèmes ou les couches du système. C'est pourquoi un langage d'expression des besoins de sécurité pour un environnement hétérogène doit être **indépendant des mécanismes**. Le langage doit donc abstraire les mécanismes de sécurité et leurs fonctionnalités en utilisant des *capacités* (section 3.4). Il sera ainsi possible d'exprimer des besoins sans préciser quels mécanismes y répondront. Grâce à cette indépendance des mécanismes, le langage peut également être **adaptatif** : il peut s'adapter aux mécanismes présents pour offrir la meilleure protection possible.

Afin de couvrir un large panel de besoins de sécurité, le langage doit également être **extensible**, c'est-à-dire qu'il doit offrir la possibilité de définir de nouvelles propriétés. En effet, bien qu'il soit nécessaire que le langage propose des propriétés générales prédéfinies permettant de protéger un système ou des applications usuelles, de nouveaux besoins peuvent émerger dans un cas d'usage spécifique. Il doit donc être possible de définir et d'ajouter des propriétés répondant à ces nouveaux besoins (section 3.5). De plus, le langage doit également permettre d'intégrer facilement de nouveaux mécanismes de sécurité en réutilisant des capacités existantes, mais également d'ajouter de nouvelles capacités, c'est-à-dire de nouvelles fonctionnalités de mécanismes.

En outre, le langage doit être **générique** : les *propriétés* de sécurité doivent être capables de s'adapter aux ressources concernées. Par exemple, une propriété de confidentialité doit pouvoir être appliquée et assurée à la fois sur des ressources systèmes et sur des ressources réseaux, en utilisant respectivement des mécanismes systèmes et réseaux. Ainsi, une même propriété doit pouvoir, selon la nature des ressources, être mise en œuvre de différentes manières, en utilisant des mécanismes appropriés et disponibles (section 3.5.3).

Enfin, le langage doit être **simple d'utilisation**. En effet, dans le cas de l'informatique en nuage, le client définissant la politique (que nous nommerons par la suite l'administrateur

de l'architecture logicielle) n'est pas nécessairement un expert en sécurité. Cependant, même s'il n'a pas de connaissances précises lui permettant de choisir comment les besoins de sécurité de son architecture doivent être appliqués et assurés, il connait ces besoins ou est capable de les définir en utilisant des méthodes d'analyse des risques (section 2.2.3). Par conséquent, une propriété doit pouvoir être définie en se basant uniquement sur les besoins en sécurité et l'architecture logicielle (section 3.5.4).

Ce chapitre détaille donc les différents éléments de notre langage : les contextes abstrayant les ressources, les capacités abstrayant les fonctionnalités des mécanismes et les propriétés exprimant les besoins de sécurité. Le chapitre montre également comment notre langage répond aux différentes exigences qui ont été définies.

3.2 Vue globale

Le langage d'expression des besoins de sécurité, présenté à la figure 3.1, met en jeu les contextes, les capacités et les propriétés, afin de définir une politique de sécurité adressant les besoins de l'administrateur de l'architecture logicielle. Le langage offre également une seconde vue à l'expert en sécurité, qui lui permet d'adapter ou de définir des propriétés.

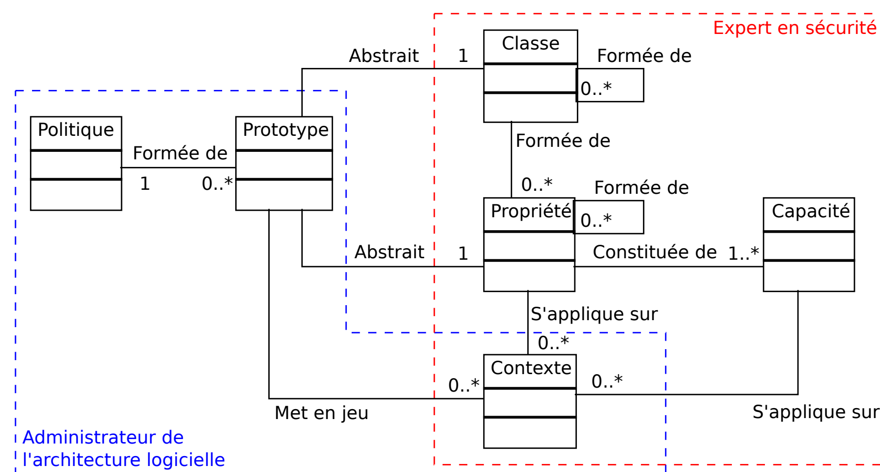

FIGURE 3.1 – Éléments du langage d'expression des besoins de sécurité

Les **contextes** sont des labels permettant d'identifier les ressources du système indépendamment des conventions de nommage. Ils permettent à une politique de s'abstraire des ressources et assurent la portabilité sur des systèmes différents. Les contextes permettent ainsi d'identifier, avec un unique système de nommage, des ressources de types différents (i.e. système, matériel, réseau, etc.)

L'abstraction des mécanismes est obtenue grâce aux **capacités**, qui correspondent aux fonctionnalités élémentaires des mécanismes de sécurité. Les capacités, qui peuvent être vues comme des fonctionnalités de sécurité de bas niveau fournies par les mécanismes, s'appliquent sur des contextes et sont donc spécifiques à certaines ressources. Un mécanisme fournit un ensemble de fonctionnalités, c'est-à-dire de capacités, et une même capacité peut être fournie par un ou plusieurs mécanismes.

Les **propriétés** correspondent à des fonctionnalités de sécurité de haut niveau, telles que la *confidentialité* ou l'*intégrité*. Elles sont obtenues par la combinaison de capacités et sont donc appliquées par un ou plusieurs mécanismes de sécurité. Les propriétés peuvent

être incluses dans des **classes**, ce qui permet de regrouper les propriétés selon le besoin de sécurité visé, indépendamment de leur implémentation ou des ressources qu'elles protègent.

L'ensemble de ces éléments permet de définir une **politique de sécurité** qui regroupe ainsi l'ensemble des ressources à protéger et les interactions autorisées entre ces ressources.

Deux vues principales sont disponibles pour utiliser le langage : la vue de l'expert en sécurité et celle de l'administrateur de l'architecture logicielle utilisant les ressources du nuage.

L'**expert en sécurité** a accès aux définitions des propriétés, c'est-à-dire à leur expression à partir des capacités. Il peut définir de nouvelles propriétés et de nouvelles capacités. Il peut également ajouter de nouveaux mécanismes de sécurité afin que ceux-ci soient accessibles par le langage.

L'**administrateur de l'architecture logicielle** correspond au client du nuage, c'est-à-dire à l'administrateur de l'application ou de la machine virtuelle déployée. Il ne connait pas nécessairement les définitions des propriétés et ne fait qu'utiliser leur prototype pour exprimer ses besoins de sécurité en définissant une politique de sécurité. Il définit les contextes associés aux ressources de son architecture logicielle, puis utilise ces contextes pour exprimer les propriétés à appliquer. Ainsi, la complexité du langage est cachée à l'administrateur de l'architecture logicielle, qui doit seulement décrire ses besoins de sécurité ou les obtenir par des méthodes d'analyse des risques.

Enfin, un troisième utilisateur est présent : le **responsable du déploiement logiciel** qui est chargé de déployer les machines.

Les éléments constitutifs du langage et les vues qui leurs sont associées sont détaillés dans la suite de ce chapitre.

3.3 Contextes

3.3.1 Définition des contextes

Dans le cadre de l'informatique en nuage, les systèmes à protéger peuvent être hétérogènes. Par conséquent, une politique de sécurité visant à protéger l'ensemble des systèmes impliqués doit être aussi indépendante du système que possible, ce qui implique que le langage doit lui aussi être indépendant du système. Pour cette raison, le langage proposé se base sur la notion de contextes associés aux ressources à protéger. En effet, en utilisant des contextes pour identifier les ressources, il est possible de s'abstraire de leur système de nommage. Cela permet à une même politique de sécurité d'être appliquée sur des nœuds ayant des systèmes d'exploitation différents ou sur des applications différentes mais offrant les même services : la politique peut ainsi être portable.

Définition 3.3.1: Contexte
Un contexte est une chaîne de caractères identifiant des ressources indépendamment des systèmes de nommage utilisés (c'est-à-dire du nom ou du chemin des ressources).

Les contextes sont donc indépendants du système. Ils peuvent être utilisés sur des systèmes hétérogènes où une même ressource est nommée de différentes manières. Un contexte est formé d'un ensemble d'attributs.

Définition 3.3.2: Attribut
Un attribut est l'association d'une clef et d'une valeur, notée `clef="valeur"`*, représentée sous la forme d'une chaîne de caractères. Un attribut fournit de l'information sur la ressource identifiée.*

La clef d'un attribut est utilisée pour fournir des informations sur la ressource identifiée. La clef peut ainsi servir à indiquer le type de la ressource (un fichier, un utilisateur, un processus, ...), la localisation de la ressource (adresse IP, le nom de l'hôte), un numéro de port (par exemple dans le cas d'une application serveur), etc. Des ressources de catégories différentes (système, réseau, matériel, etc.) peuvent ainsi être identifiées de manière uniforme. L'ensemble des clefs est représenté à l'aide d'un arbre. En effet, chaque clef peut posséder des sous-clefs qui raffinent la caractérisation des ressources et permettent de définir des propriétés de sécurité génériques. Ainsi, la clef `Hardware` possède notamment les deux sous-clefs `Smartphone` et `Computer`. Le sous-arbre de la clef `Hardware` est présenté à la figure 3.2.

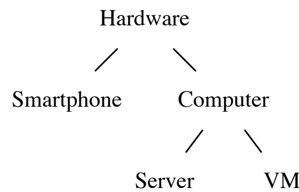

FIGURE 3.2 – Extrait du sous-arbre de la clef *Hardware*

Pour définir un contexte, les clefs sont associées à des valeurs. Alors que la clef spécifie le type de l'attribut, la valeur restreint les ressources sélectionnées. Les valeurs des attributs sont interprétées comme des expressions régulières : cela permet donc à un contexte de regrouper plusieurs autres contextes, ce qui simplifie et factorise l'expression de la politique.

Ainsi, la clef `Hardware.Computer.Server` spécifie que la ressource est un serveur ou se situe sur un serveur, tandis que les valeurs `MailServer` et `WebServer` permettent de différentier les différentes catégories de serveurs.

Un contexte peut donc être défini de la façon suivante :

Définition 3.3.3: Contexte
Soient $K_{i,1 \leq i \leq n}$ un ensemble de clefs d'attributs et $V_{i,1 \leq i \leq n}$, l'ensemble des valeurs qui leur sont associées. Soit SC *un contexte formé de ces attributs. Alors,* SC *est défini par :*

$$\begin{aligned} SC &:= (K_1 = "V_1") : (K_2 = "V_2") : [...] : (K_n = "V_n") \\ &:= (K_i, V_i)_{1 \leq i \leq n} \end{aligned}$$

Par convention, un contexte ne peut pas posséder deux attributs ayant la même clef. On définit donc la règle suivante :

Règle 3.3.4: Unicité des clefs
Les clefs des attributs d'un contexte sont uniques.
Soit $K_{i,1 \leq i \leq n}$ l'ensemble des clefs d'attributs du contexte SC. Alors,

$$\forall (i, j) \in [1, n], i \neq j \Rightarrow K_i \neq K_j$$

La figure 3.3 présente un extrait de l'actuel arbre des clefs de contextes. Cet arbre est basé sur les modèles de données utilisés dans des CMDB (*Configuration Management DataBase*) [Clark *et al.*, 2007], tel que celui de OneCMDB [OneCMDB, 2015]. En effet, les CMDB visent à vérifier le statut des éléments d'un système et ont donc besoin de classifier ces éléments : cela correspond aux clefs de contextes qui identifient les différentes ressources impliquées. De plus, les CMDB étant utilisées dans le cadre industriel, l'administrateur de

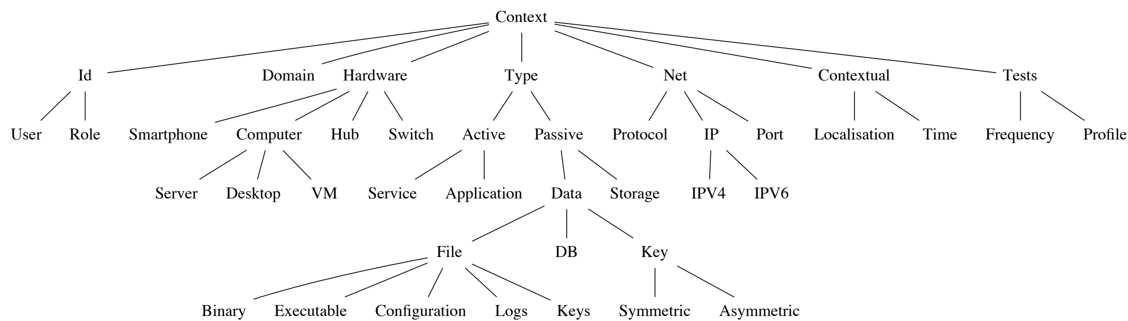

FIGURE 3.3 – Extrait de l'arbre des clefs de contextes

l'architecture logicielle peut utiliser les informations disponibles dans une CMDB pour identifier les ressources avec le langage proposé.

L'arbre des clefs de contextes couvre les ressources telles que les utilisateurs et les processus (sous-arbre `ID`), le domaine système ou réseau dont ils font partie (`Domain`), les éléments matériels (`Hardware`) ou logiciels (`Type.Active`), les données (`Type.Passive`), la configuration réseau (`Network`), les données contextuelles (`Contextual`), et les données d'assurance (`Tests`). De nouveaux attributs peuvent être ajoutés à l'arbre afin d'étendre les types de ressources pris en compte par le langage.

Chaque contexte est associé à un nom qui lui sert d'identifiant. Ce nom peut ensuite être utilisé dans les propriétés pour faire appel au contexte. Par exemple, pour identifier les fichiers de configuration d'un serveur mail, le contexte `mailConfig` peut être défini ainsi :

```
mailConfig := (Type.Passive.Data.File.Configuration="conf"):(Domain="mail"):(
    Hardware.Computer="mailServer");
```

Le premier attribut de ce contexte identifie des fichiers de configuration. Le deuxième attribut restreint ces fichiers à ceux appartenant au domaine `mail`. Enfin, le troisième attribut spécifie sur quelles machines sont localisés ces fichiers.

Considérons également le contexte `serviceConfig` :

```
serviceConfig := (Type.Passive.Data.File.Configuration="conf"):(Domain="ssh|
    mail"):(Hardware.Computer="mailServer");
```

On remarque que la valeur de l'attribut `Domain` de `serviceConfig` est une expression régulière, englobant les domaines `mail` et `ssh`. Par conséquent, une propriété concernant `serviceConfig` sera appliquée notamment sur les ressources ayant le contexte `mailConfig`.

Ainsi, les contextes nous permettent d'abstraire les ressources et de proposer un langage de politique indépendant de leur nommage.

3.3.2 Compatibilité de contextes

Comme nous le verrons dans les sections 3.4 et 3.5, une notion de compatibilité est indispensable pour manipuler les contextes de sécurité. En effet, l'expression des propriétés de sécurité reposant sur les contextes, la sélection d'une propriété se base également sur ceux-ci. De plus, les contextes étant associés aux ressources, il est nécessaire de pouvoir obtenir l'ensemble des ressources identifiées par un contexte donné.

3.3.2.1 Filtrage des types de contextes

Nous définissons tout d'abord la notion de **type de contexte** qui correspond à un contexte réduit à son ensemble de clefs d'attributs.

Définition 3.3.5: Type de Contexte
Soit un contexte SC *tel que :*

$$\texttt{SC} := (K_1 = "V_1") : (K_2 = "V_2") : [...] : (K_n = "V_n")$$
$$:= (K_i, V_i)_{1 \leq i \leq n}$$

Alors, le type du contexte SC *est défini par :*

$$\texttt{Type(SC)} := K_1 : K_2 : [...] : K_n$$
$$:= (K_i)_{1 \leq i \leq n}$$

Considérons par exemple le contexte `mailConfig` :

```
mailConfig := (Type.Passive.Data.File.Configuration="conf"):(Domain="mail"):(
    Hardware.Computer="mailServer");
```

Le type de ce contexte est l'ensemble des clefs de ses attributs, soit :

```
Type(mailConfig) := Type.Passive.Data.File.Configuration:Domain:
    Hardware.Computer;
```

Inclusion de clefs d'attribut L'inclusion de clefs d'attribut entre plusieurs types de contextes est nécessaire afin de déterminer quelles propriétés de sécurité sont applicables pour un contexte donné.

Un type de contexte étant composé de clefs d'attributs, il faut, pour déterminer si un type de contexte est présent dans un autre type, être capable de déterminer la présence ou l'absence d'une clef d'attribut dans une autre clef. L'inclusion de clef d'attribut est donc définie comme suit :

Définition 3.3.6: Inclusion de clefs d'attribut
On note $P(K_i)$ le chemin d'une clef d'attribut K_i.
Soient K_1 et K_2 deux clefs d'attributs de longueurs respectives n et m.
Soient $P(K_1) := (k_{1,i})_{1 \leq i \leq n}$ et $P(K_2) := (k_{2,j})_{1 \leq j \leq m}$ leurs chemins associés dans l'arbre des clefs de contextes, où les $k_{1,i}$ et $k_{2,j}$ sont les nœuds de l'arbre.
Alors, la clef K_1 est incluse dans K_2 si et seulement si le chemin de K_1 est inclus dans celui de K_2, soit :

$$K_1 \subset K_2 \Leftrightarrow P(K_1) \subset P(K_2)$$
$$\Leftrightarrow \forall i \in [1, n], k_{1,i} == k_{2,i}$$

Considérons par exemple les clefs K_1 et K_2 suivantes et leurs chemins associés :

$K_1 := Hardware.Computer$ $\quad P(K_1) := (Hardware, Computer)$
$K_2 := Hardware.Computer.VM$ $\quad P(K_2) := (Hardware, Computer, VM)$

Alors, le clef K_2 contient la clef K_1 puisque le chemin $P(K_1)$ est inclus dans $P(K_2)$.

Inclusion de types de contextes Un type de contexte est un ensemble d'attributs qui peut être représenté sous forme d'arbre. La détermination de l'inclusion de deux types de contextes peut ainsi être vue comme un problème simplifié d'inclusion d'arbres. On définit donc la règle suivante :

Règle 3.3.7: Inclusion de types de contextes
L'inclusion d'un type de contexte T_1 dans un type de contexte T_2 peut être ramené à un problème d'inclusion de chemins dans un arbre.

La règle précédente peut être énoncée grâce à la justification 3.3.8.

Justification 3.3.8:
Dans le cas général, un arbre T_1 est dit inclus dans un arbre T_2 s'il est possible de transformer T_2 en T_1 en effectuant une série de suppressions de nœuds sur T_2. Dans notre cas, il est nécessaire de conserver le chemin de chaque nœud, puisque la hiérarchie des nœuds est significative. Par conséquent, on ne considère que les opérations de suppression sur les feuilles de l'arbre ou sur la totalité d'un sous-arbre (si un nœud interne est supprimé, tous ses descendants le sont également). Il s'agit donc d'un calcul d'inclusion de chemins.

On peut donc déterminer l'inclusion d'un arbre T_1 dans un arbre T_2 en utilisant l'algorithme 1. Pour chaque feuille f_i de l'arbre T_1, on recherche dans l'arbre T_2 une feuille dont le chemin contient le chemin de f_i. Si toutes les feuilles de T_1 sont présentes dans T_2, alors T_1 est inclus dans T_2.

Algorithme 1 Inclusion d'un type de contexte T_1 dans un type de contexte T_2

```
 1: function InclusionDeTypesContextes (T1, T2)
 2:     F1 ← les feuilles de T1
 3:     F2 ← les feuilles de T2
 4:     for all feuille fi dans F1 do
 5:         P(fi) ← le chemin de fi
 6:         inclusionFeuille ← Faux
 7:         for all feuille fj dans F2 do
 8:             P(fj) ← le chemin de fj
 9:             if P(fi) est inclus dans P(fj) then
10:                 inclusionFeuille ← Vrai
11:             end if
12:         end for
13:         if inclusionFeuille est Faux then
14:             return Faux
15:         end if
16:     end for
17:     return Vrai
18: end function
```

Score d'inclusion Cette définition de l'inclusion des types de contextes nous permettra de sélectionner les capacités et les propriétés qui peuvent être appliquées sur des contextes spécifiques. Cependant, dans le cas où plusieurs capacités ou propriétés peuvent être appliquées à un contexte, il est nécessaire de choisir la plus adaptée. Nous définissons donc un score représentant la qualité de l'inclusion.

Définition 3.3.9: Score d'inclusion
Le score d'inclusion correspond à la distance d'édition d'arbre [Tai, 1979], définie par :

$$d(T_1, T_2) = min(cost(S))$$

où S est une série d'opérations d'édition (insertions, suppressions, substitutions) permettant de transformer T_1 en T_2.

Dans notre cas, cette distance n'est calculée que si T_1 est inclus dans T_2. Par conséquent, le calcul de la distance d'édition peut être ramené au calcul de la différence de taille entre les deux arbres, comme montré par la justification 3.3.10.

Justification 3.3.10:
Soient T_1 et T_2 deux types de contextes tels que T_1 est inclus dans T_2. En raison de cette inclusion, T_1 peut être obtenu à partir de T_2 en effectuant uniquement des opérations de

suppression de nœuds. Par conséquent, le calcul de la distance d'édition est équivalente au calcul du nombre de nœuds de T_2 à supprimer pour obtenir T_1. Or, le nombre de nœuds à supprimer est égal à la différence de nombre de nœuds entre T_2 et T_1, c'est-à-dire à la différence de taille entre les deux arbres. On peut donc en conclure que la distance d'édition entre T_1 et T_2 est égale à la différence de taille entre T_2 et T_1.

On obtient donc la définition suivante :

Définition 3.3.11: Score d'inclusion de types de contextes
Soient T_1 et T_2 deux types de contextes tels que T_1 est inclus dans T_2. Le score d'inclusion de T_1 dans T_2 est :

$$d(T_1, T_2) = size(T_2) - size(T_1)$$

où $size(T_i)$ est la taille de l'arbre, c'est-à-dire son nombre de nœuds.

Par convention, un score $d(T_1, T_2)$ strictement négatif indique que T_1 n'est pas inclus dans T_2. Dans le cas contraire, si T_1 est inclus dans T_2, le score sera positif ou nul. Un score nul indique que les deux types de contextes sont identiques, alors qu'un score de 1 indique qu'il faut supprimer un nœud pour transformer T_2 en T_1. On définit la règle suivante :

Règle 3.3.12: Comparaison des scores d'inclusion
Si le score d'inclusion de T_1 dans T_2 est positif ou nul, alors plus le score est faible, meilleure est l'inclusion. Un score d'inclusion négatif indique que T_1 n'est pas inclus dans T_2.

Le score d'inclusion peut ainsi être calculé à l'aide de l'algorithme 2.

Algorithme 2 Calcul du score d'inclusion d'un type de contexte T_1 dans un type T_2

```
1: function scoreInclusion (T1, T2)
2:     if InclusionDeTypesContextes(T1,T2) est Faux then
3:         return -1
4:     end if
5:     taille_T1 ← nombre de nœuds de T1
6:     taille_T2 ← nombre de nœuds de T2
7:     score = taille_T2 - taille_T1
8:     return score
9: end function
```

Considérons les quatre types de contextes de la figure 3.4.

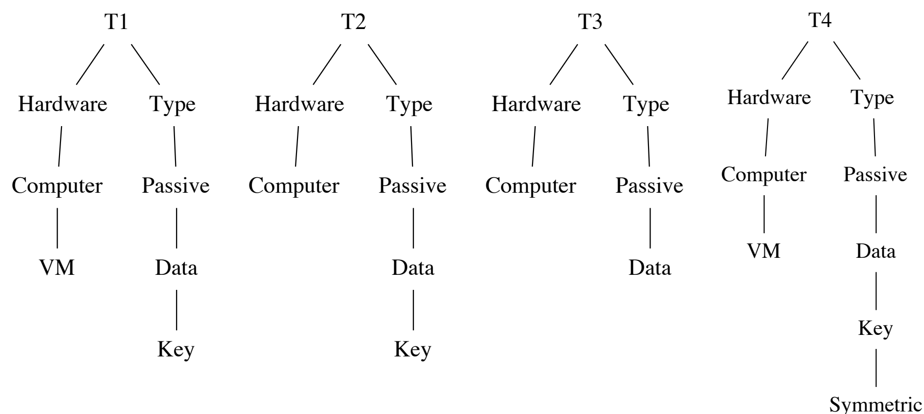

FIGURE 3.4 – Exemples de types de contextes

Les types de contextes T_2 et T_3 sont tous deux inclus dans T_1. En revanche, T_4 n'est pas inclus dans T_1 : le nœud `Symmetric` de T_4 n'est pas présent dans T_1. Les scores d'inclusion ne sont donc calculés que pour T_2 et T_3. On obtient ainsi les scores suivants :

$$d(T_2, T_1) = size(T_1) - size(T_2) = 1$$

$$d(T_3, T_1) = size(T_1) - size(T_3) = 2$$

T_2 est le type de contexte le plus proche de T_1. En effet, son score est plus faible, ce qui implique un nombre d'opérations d'éditions moins important. Par conséquent, pour le contexte T_1, une capacité attendant en argument un contexte de type T_2 sera préférée à une capacité attendant un argument de type T_3.

Ce score d'inclusion sera utilisé dans la suite de ce chapitre afin de sélectionner les capacités (section 3.4.2) et les propriétés (section 3.5.4) les plus adaptées aux contextes.

3.3.2.2 Filtrage de contextes

Afin de déterminer sur quelles ressources une propriété doit être appliquée, une recherche de motif dans les contextes complets doit être effectuée. Pour cela, nous définissons tout d'abord la notion de compatibilité entre deux valeurs d'attributs :

Définition 3.3.13: Compatibilité de valeurs d'attributs
Soient T_1 et T_2 deux attributs. La valeur de T_2 est dite compatible avec la valeur de T_1 si elle est incluse dans la valeur de T_1 en utilisant les règles usuelles de correspondance d'expressions régulières.

On peut alors définir la compatibilité entre contextes :

Définition 3.3.14: Compatibilité de contextes
Soient `A` et `B` deux contextes. `B` est compatible avec `A` si les points suivants sont vérifiés :
- *Le type de contexte de `A` est inclus dans le type de contexte de `B` ;*
- *Pour chaque valeur d'attribut de `A`, la valeur de l'attribut de `B` correspondant est compatible. Les attributs supplémentaires de `B` ne sont pas pris en compte.*

La compatibilité d'un contexte SC_1 avec un contexte SC_2 peut être déterminée en utilisant l'algorithme 3. Il est alors possible de savoir sur quelles ressources une capacité ou une propriété de sécurité doit être appliquée.

L'algorithme 3 commence par vérifier que le type de contexte de SC_2 est bien inclus dans celui de SC_1 (lignes 2 à 4). Si c'est le cas, l'algorithme parcourt ensuite les feuilles du contexte SC_2 (lignes 7 à 23). Pour chacune de ces feuilles, l'algorithme parcourt les feuilles de SC_1 (lignes 11 à 19) jusqu'à trouver une feuille ayant un chemin inclus dans celui de la feuille de SC_2 (ligne 14). Si une telle feuille est trouvée, les valeurs des deux feuilles sont comparées (ligne 15) : si la valeur de la feuille de SC_1 est compatible avec celle de SC_2, alors, la feuille de SC_1 est compatible avec SC_2 (ligne 16), et ce processus est répété pour toutes les feuilles de SC_2. Si les valeurs ne correspondent pas ou si aucune feuille n'a le bon chemin, alors le contexte SC_1 n'est pas compatible avec SC_2 (lignes 20 à 23).

Considérons les contextes `mailConfig` et `serviceConfig` suivants :

```
mailConfig := (Type.Passive.Data.File.Configuration="conf"):(Domain="mail"):(
    Hardware.Computer="mailServer");
serviceConfig := (Type.Passive.Data.File.Configuration="conf"):(Domain="ssh|
    mail");
```

Algorithme 3 Compatibilité d'un contexte SC_1 avec un contexte SC_2

```
1: function compatibilitéDeContextes (SC1, SC2)
2:     if InclusionDeTypesContextes(Type(SC2), Type(SC1)) == Faux then
3:         return Faux
4:     end if
5:     F1 ← les feuilles de SC1
6:     F2 ← les feuilles de SC2
7:     for all feuille fi dans F2 do
8:         P(fi) ← le chemin de fi
9:         V(fi) ← la valeur de fi
10:        inclusionFeuille ← Faux
11:        for all feuille fj dans F1 do
12:            P(fj) ← le chemin de fj
13:            V(fj) ← la valeur de fj
14:            if P(fi) est inclus dans P(fj) then
15:                if V(fj) est compatible avec V(fi) then
16:                    inclusionFeuille ← Vrai
17:                end if
18:            end if
19:        end for
20:        if inclusionFeuille est Faux then
21:            return Faux
22:        end if
23:    end for
24:    return Vrai
25: end function
```

Le contexte `mailConfig` est compatible avec le contexte `serviceConfig`. En effet, le type de contexte `Type(serviceConfig)` est inclus dans `Type(mailConfig)` et les valeurs des attributs de `mailConfig` sont compatibles avec celles de `serviceConfig`. Cela implique qu'une propriété mettant en jeu le contexte `serviceConfig` sera appliquée sur les ressources ayant le contexte `mailConfig`.

3.3.3 Association des contextes aux ressources

Afin de proposer une politique de sécurité indépendante du nommage des ressources, une couche d'abstraction est nécessaire. C'est l'objectif des contextes qui permettent d'identifier une ressource sans avoir recours à son nom ou à son chemin. Cependant, pour que la politique puisse être effectivement appliquée, il est nécessaire de fournir une association entre les contextes abstraits et les ressources réelles. Pour cette raison, nous utilisons une table associative entre les ressources et les contextes.

Dans le cas d'un nuage informatique, les systèmes utilisés sont hétérogènes. Cependant, les machines virtuelles sont le plus souvent instanciées en grand nombre, à partir d'une même image. La table associative peut donc être partagée par toutes ces VMs identiques. De plus, dans de nombreux cas, la table peut être pré-générée pour chaque distribution : en effet, les services, applications et fichiers usuels peuvent être associés à leur contexte respectif pour chaque système. Il n'est donc pas nécessaire de définir ces associations pour chaque élément du système, mais uniquement pour les éléments spécifiques à l'architecture logicielle du client. De plus, des applications proposant les mêmes services pourront utiliser les mêmes contextes et la même politique : seule l'association des contextes aux ressources sera différente. Enfin, il est possible d'automatiser la génération des tables, par exemple en utilisant un service de découverte des applications ou des utilisateurs. Ce service peut associer automatiquement les contextes définis dans la politique aux ressources correspon-

dantes. Ainsi, la table d'associativité lie les contextes aux ressources, sans complexifier la migration de la politique ni diminuer son caractère portable.

La table d'associativité est initialisée sur chacun des nœuds. Elle est spécifique à chaque nœud, bien que des nœuds similaires (avec le même système) puissent partager certaines entrées de cette table. Chaque entrée est un triplet $(Catégorie, Ressource, Contexte)$ où :

- *Catégorie* définit le type de la ressource considérée. Il peut s'agir d'un objet `o` passif du système (i.e. un fichier), d'un processus `p`, d'un utilisateur `u` ou d'une machine `c` (physique ou virtuelle) ;
- *Ressource* est l'identifiant de la ressource. Selon les cas, il peut s'agir du chemin absolu d'une ressource, du chemin absolu de l'exécutable ou du binaire correspondant à un processus, de l'identifiant d'un utilisateur ou d'une adresse IP. L'identifiant de la ressource peut être exprimé sous forme d'expression régulière ;
- *Contexte* est le nom du contexte associé à la ressource.

Règle 3.3.15: Unicité des contextes par paire Catégorie-Ressource
Pour chaque paire $(Catégorie, Ressource)$, le contexte associé est unique. En revanche, un même contexte peut être utilisé pour un nombre quelconque de paires $(Catégorie, Ressource)$.

Considérons les associations suivantes :

```
o /etc/postfix(/.*)?          mailFiles
o /etc/postfix/main.cf        mailConfig
o /etc/postfix/master.cf      mailConfig
o /var/log/mail.log           mailLogs
o /usr/sbin/postfix           mailBinary

u user                        unpriviledgedUser

p /usr/sbin/postfix           mailService

c 192.168.0.10                mailServer
```

Listing 3.1 – Exemple d'associations ressource/contexte sur un système de type Unix

Les cinq premières entrées définissent des contextes pour les fichiers, la configuration, les logs et le binaire d'un serveur mail. Le contexte `mailFiles` est associé à une expression régulière identifiant plusieurs fichiers. Un même contexte `mailConfig` identifie deux fichiers différents : les mêmes propriétés seront donc appliquées sur ces deux fichiers. Les fichiers de configuration sont également concernés par l'expression régulière de la première association : en raison de l'unicité du contexte pour chaque ressource, un seul contexte peut être assigné à ces deux fichiers. Les noms de fichiers sont exprimés suivant la syntaxe `FCGlob` [Miner et Athey, 2007] et l'association la plus précise est celle prise en compte : les fichiers de configuration ont donc le contexte `mailConfig`. La sixième entrée définit le contexte d'un utilisateur en spécifiant son nom. La septième entrée identifie le service mail en utilisant le fichier binaire qui est exécuté pour démarrer le service. Enfin, la dernière entrée associe une adresse IP à un contexte de serveur.

Le listing 3.1 présente donc un exemple d'associations de contextes à des ressources dans le cas du serveur mail `Postfix`. Cependant, les contextes étant indépendants de l'application, ils peuvent également être utilisés pour sécuriser un autre serveur mail. Le listing 3.2 montre les associations à ajouter afin de sécuriser un serveur `qmail` de la même manière que le serveur `Postfix`. A l'exception de ces quelques associations, la politique n'a pas besoin d'être modifiée : la politique s'adapte donc aisément aux différentes applications.

```
o /var/qmail(/.*)?            mailFiles
```

```
2 o /var/qmail/control(/.*)?   mailConfig
3 o /var/log/qmail             mailLogs
4 o /var/qmail/bin(/.*)?       mailBinary
5
6 p /var/qmail/bin(/.*)?       mailService
```

Listing 3.2 – Exemple d'associations ressource/contexte sur un système de type Unix

Une table d'association peut être prédéfinie pour les applications classiques de chaque système. L'administrateur ne doit donc préciser que les associations spécifiques à un cas d'usage donné. De plus, grâce à cette table, une même politique peut être utilisée sur des systèmes ayant des conventions de nommage distinctes. En effet, la table d'association étant la seule partie de la politique directement liée au système, c'est également la seule partie à en dépendre et à devoir être modifiée pour porter la politique sur un autre système.

On peut noter que l'association entre les contextes et les ressources ne prend pas en compte les migrations de ressources. Ce choix s'explique par l'objectif du langage proposé qui est d'appliquer et d'assurer des besoins de sécurité en utilisant des mécanismes existants. Ainsi, le langage d'expression des besoins de sécurité génère la configuration initiale des mécanismes de sécurité (et la met à jour en cas de modification de la politique) mais le suivi des évolutions est laissé aux mécanismes utilisés. Par exemple, dans le cas de SELinux, la migration des contextes associés aux ressources est réalisée par SELinux.

3.4 Capacités

3.4.1 Définition des capacités

Afin d'obtenir une définition abstraite de la politique de sécurité, il est nécessaire de s'abstraire à la fois des ressources à protéger et des mécanismes de sécurité. Les contextes permettent de répondre au problème de l'abstraction des ressources. Dans cette section, nous expliquons comment l'indépendance vis-à-vis des mécanismes de sécurité peut être obtenue grâce aux capacités de sécurité.

Les capacités sont des fonctionnalités élémentaires et de bas niveau qui permettent de lier les propriétés et les mécanismes de sécurité qui les appliquent. L'administrateur de l'architecture logicielle n'a pas conscience de ces capacités. Elles sont définies et gérées par un expert en sécurité. L'expert peut donc définir de nouvelles capacités, créer de nouvelles associations entre les capacités et les mécanismes, et définir de nouvelles propriétés à l'aide des capacités. Cependant, ces éléments sont cachés à l'administrateur de l'architecture logicielle qui peut manipuler les propriétés sans connaître leur fonctionnement interne.

Définition 3.4.1: Capacité
Une capacité est une fonctionnalité abstraite et élémentaire de sécurité fournie par un ou plusieurs mécanismes.

Une capacité peut être fournie par plusieurs mécanismes de sécurité et ceux-ci peuvent l'implémenter de manières différentes. Par conséquent, une capacité peut être utilisée pour répondre à un besoin de sécurité, sans que l'expression de ce besoin repose sur un mécanisme donné. Cela permet donc d'obtenir une expression des besoins indépendante du mécanisme qui sera chargé d'y répondre.

Une capacité est caractérisée par son nom et ses arguments. Le nom d'une capacité est une chaîne de caractère permettant de l'identifier et caractérisant la fonctionnalité de sécurité offerte. Afin de savoir sur quelles catégories de ressources les capacités s'appliquent,

chaque argument d'une capacité est un type de contexte, c'est-à-dire un ensemble de clefs d'attributs.

Les capacités peuvent être classées en deux catégories : les capacités d'application et celles d'assurance. Le langage ne différencie pas ces deux types de capacités : il s'agit uniquement d'une différence sémantique qui permet de classifier les capacités selon leurs fonctionnalités. Les capacités d'application permettent de répondre à un besoin de sécurité en interagissant avec un mécanisme de sécurité. Les capacités d'assurance ont pour objectif de contrôler l'état de la sécurité en effectuant des vérifications ou des tests.

Considérons les capacités `C1` et `C2` définies comme suit :

```
C1 := allow_read_access (Type.Passive.Data.File SCFile, ID.User SCUser);
C2 := generate_key (Type.Passive.Data.Key SCKey);
```

`C1` est une capacité permettant d'autoriser une opération de lecture sur une ressource. Le premier argument `SCFile`, de type `Type.Passive.Data.File`, spécifie la ressource protégée : il s'agit ici d'un fichier (ou d'un ensemble de fichiers). Le second argument `SCUser` indique quelles ressources (ici, quels utilisateurs) sont autorisées à lire le fichier protégé. `C1` peut être appliquée par plusieurs mécanismes de contrôle d'accès, par exemple SELinux ou les droits Unix. `C2` peut quant à elle être utilisée pour générer des clefs de chiffrement, par exemple avec OpenSSL ou une carte à puce.

Ainsi, il est possible de définir de nombreuses capacités différentes pour répondre aux besoins de sécurité en abstrayant les fonctionnalités élémentaires des mécanismes. Un mécanisme peut ainsi être vu comme un fournisseur de capacité. De plus, comme dans le cas de `C1`, plusieurs mécanismes peuvent fournir une même capacité, ce qui permet d'obtenir une indépendance vis-à-vis des mécanismes.

Les capacités s'appliquent sur des ressources identifiées par des contextes : on parle alors de capacités instanciées.

Définition 3.4.2: Instance de capacité
Une instance de capacité est une capacité mettant en jeu des contextes.

Considérons `Cinst`, l'instance de la capacité `C1` vue précédemment, qui permet d'autoriser certains accès en lecture à un fichier.

```
Cinst := allow_read_access (Type.Passive.Data.File="conf", ID.User="root");
```

`Cinst` met en jeu deux contextes : le premier identifie les fichiers concernés par la capacité et le deuxième identifie l'utilisateur autorisé à lire les fichiers.

3.4.2 Surcharge et priorité des capacités

Les capacités peuvent être surchargées : deux capacités peuvent avoir le même nom si leurs arguments diffèrent. Cela permet de définir des capacités plus ou moins précises selon le type de contexte de leurs arguments. Plus le type de contexte des arguments est explicite, plus la capacité est précise et spécialisée. On définit donc la notion de compatibilité de capacités permettant d'indiquer quelles capacités peuvent être choisies pour appliquer une capacité instanciée :

Définition 3.4.3: Compatibilité de capacités
Soient C une capacité et C_{inst} une capacité instanciée. C est dite compatible avec C_{inst} si :
- *elles ont le même nom ;*
- *elles ont le même nombre d'arguments ;*
- *pour chaque argument $Arg_1(i)$ de C et $Arg_2(i)$ de C_{inst}, le type de contexte $Type(Arg_1(i))$ est inclus dans $Type(Arg_2(i))$.*

L'algorithme 4 est utilisé pour déterminer si des capacités sont compatibles. L'algorithme commence par vérifier que les deux capacités ont le même nom et le même nombre d'arguments (lignes 2 à 4). Puis, pour chaque argument des capacités (lignes 5 à 11), l'algorithme récupère les types des arguments (lignes 6 et 7) et vérifie leur inclusion (ligne 8). Si le critère d'inclusion n'est pas respecté, les capacités ne sont pas compatibles (ligne 9). S'il est respecté pour tous les arguments, C est compatible avec C_{inst} (ligne 12).

Algorithme 4 Détermination de la compatibilité de la capacité C avec la capacité C_{inst}

```
1:  function capacitéCompatibles (C, C_inst)
2:      if C.nom ≠ C_inst.nom OU C.nbArgs ≠ C_inst.nbArgs then
3:          return Faux
4:      end if
5:      for all i dans [1, C.nbArgs] do
6:          T ← Type(C.arg[i])
7:          T_inst ← Type(C_inst.arg[i])
8:          if InclusionDeTypesContextes(T, T_inst) == Faux then
9:              return Faux
10:         end if
11:     end for
12:     return Vrai
13: end function
```

Pour un ensemble de contextes donnés, la capacité à appliquer est sélectionnée en utilisant le score d'inclusion des capacités, qui est basé sur le score d'inclusion des types de contextes défini à la section 3.3 et donné par la fonction scoreInclusion.

Définition 3.4.4: Score d'inclusion de capacités

Soit C_{inst} une instance de capacité dont les arguments sont des contextes instanciés notés $SC_{i,1 \leq i \leq n}$. Soient C une capacité compatible avec C_{inst} et $T_{i,1 \leq i \leq n}$ les types de contextes qui lui sont passés en argument.

On définit le score d'inclusion $d(C, C_{inst})$ de C dans C_{inst} comme la somme des distances des contextes, au sens de la définition 3.3.11. On a donc :

$$d(C, C_{inst}) = \sum_{i=1}^{n} d(T_i, Type(SC_i))$$

Le score d'inclusion peut être calculé à partir de l'algorithme 5. L'algorithme vérifie tout d'abord que les capacités sont compatibles (lignes 2 à 4). Si c'est le cas, alors, pour chacun des contextes passés en argument aux capacités C et C_{inst}, le score d'inclusion des types de contextes attendus est calculé. Leur somme fournit le score d'inclusion de la capacité (ligne 9). Dans le cas où la capacité C n'est pas compatible avec C_{inst}, la fonction renvoie -1 (lignes 3 et 11). Sinon, la fonction renvoie le score d'inclusion (ligne 14).

En utilisant le score d'inclusion des capacités et pour un ensemble de contextes donnés, il est possible d'ordonner les capacités surchargées en utilisant la règle suivante.

Règle 3.4.5: Sélection de capacité

Soit C_{inst} une instance de capacité. Soient $C_{i,1 \leq i \leq n}$ les capacités compatibles avec C_{inst}. Les C_i sont ordonnées selon leur score $d(C_i, C_{inst})$, en considérant le score positif le plus faible comme le meilleur.

Considérons les capacités C2 et C3 définies comme suit :

```
C2 := generate_key(Type.Passive.Data.Key);
C3 := generate_key(Type.Passive.Data.Key.Symmetric);
```

Algorithme 5 Calcul du score d'inclusion d'une capacité C dans une capacité C_{inst}

1: **function** SCOREINCLUSIONCAPACITÉ (C, C_{inst})
2: **if** CAPACITÉCOMPATIBLES(C, C_{inst}) == `Faux` **then**
3: **return** -1
4: **end if**
5: score ← 0
6: **for all** contexte[i] dans $C_{inst}.listeContextes$ **do**
7: argType ← $C.argument[i]$
8: **if** COMPATIBILITÉDECONTEXTE(argType, contexte[i]) **then**
9: score ← score+SCOREINCLUSION(argType, Type(contexte[i])) ▷ Déf. 3.3.11
10: **else**
11: **return** -1
12: **end if**
13: **end for**
14: **return** score
15: **end function**

Ces deux capacités fournissent la même fonctionnalité : elles permettent toutes deux de générer une clef de chiffrement, mais C_3 ne peut générer qu'une clef symétrique, au contraire de C_2. Par conséquent, si une clef asymétrique est nécessaire, seule C_2 peut répondre au besoin, mais si une clef symétrique est demandée (c'est-à-dire si l'argument passé est de type `Type.Passive.Data.Key.Symmetric`), C_2 et C_3 peuvent toutes deux être utilisées. Dans ce deuxième cas, C_3 étant une propriété plus précise, elle sera choisie pour répondre au besoin si un mécanisme pouvant l'appliquer est présent.

Considérons les trois contextes suivants :

```
SC1 := (Type.Passive.Data.Key="foo");
SC2 := (Type.Passive.Data.Key.Symmetric="foo");
SC3 := (Type.Passive.Data.Key.Asymmetric="foo");
```

Les scores d'inclusion, calculés comme décrit par l'algorithme 5, sont donnés dans la table 3.1.

	C_2	C_3
generate_key (SC1)	0	-1
generate_key (SC2)	1	0
generate_key (SC3)	1	-1

TABLE 3.1 – Exemples de scores d'inclusion de capacités

Un score négatif indique que la capacité ne peut pas être utilisée pour le contexte concerné. Pour chaque contexte, la capacité la plus adaptée est donc celle avec le plus faible score positif. Par conséquent, C_2 sera choisie pour générer une clef de chiffrement si les contextes `SC1` ou `SC3` sont utilisés. Dans le cas du contexte `SC2`, la capacité C_3 sera choisie. Cependant, si la capacité C_3 ne peut pas être appliquée (par exemple parce qu'aucun mécanisme disponible ne fournit C_3), alors C_2 pourra être utilisée.

Si deux capacités ont le même score pour un contexte donné, le choix se base sur le score des mécanismes associés, défini à la section 3.4.3. Si les scores des mécanismes ne permettent pas de sélectionner une capacité, alors la première capacité compatible définie est considérée comme prioritaire.

On peut donc définir des capacités s'appliquant sur des contextes plus ou moins précis, ce qui permet d'obtenir différents niveaux d'abstraction des mécanismes de sécurité. Plus la définition d'une capacité est précise (i.e. plus le type de ses arguments est explicite), plus son application sera spécifique à la catégorie des ressources considérées. La protection

sera donc plus adaptée, ce qui la rendra plus efficace : ce critère est pris en compte par le score d'inclusion des capacités.

Notons cependant l'intérêt des capacités moins précises : elles peuvent être appliquées sur un plus grand nombre de ressources, ce qui permet une généricité plus importante, bien que leur protection soit moins adaptée aux ressources. Une telle capacité ne sera utilisée que dans le cas où des capacités plus précises ne peuvent pas être appliquées, par exemple parce qu'aucun mécanisme disponible ne les fournit.

Les capacités utilisées sont choisies dynamiquement lors de l'application de la politique. Ce choix se base sur la configuration du système (c'est-à-dire sur la liste des mécanismes disponibles) et sur le score d'inclusion des capacités.

L'algorithme 6 décrit le processus de sélection et d'ordonnancement des capacités selon le nom de la capacité et la liste de contextes passée en argument. L'algorithme recherche tout d'abord les définitions de capacités ayant le même nom et le bon nombre d'arguments (ligne 4). Puis, le score d'inclusion de la capacité c recherchée dans la capacité c_i est calculé (ligne 6). Si ce score est positif ou nul, les contextes de c_i sont inclus dans ceux de c et c_i peut donc appliquer c (lignes 7 à 9). La fonction renvoie l'ensemble des capacités répondant à ces critères d'inclusion, triées par score d'inclusion.

Algorithme 6 Sélection de capacité

```
 1: function chercheCapacitésCompatibles (nom de capacité c, listeContextes)
 2:     listeCapacitésCompatibles ← ∅
 3:     for all c_i dans l'ensemble des capacités do
 4:         if CapacitéCompatible(c_i, {c, listeContextes} ) == Vrai then
 5:             compatible ← Vrai
 6:             score ← ScoreInclusionCapacité(c_i, {c, listeContextes})
 7:             if score ≥ 0 then
 8:                 listeCapacitésCompatibles ← c_i ordonnée selon le score    ▷ Règle 3.3.12
 9:             end if
10:         end if
11:     end for
12:     return listeCapacitésCompatibles
13: end function
```

3.4.3 Association des capacités aux mécanismes de sécurité

Chaque capacité est associée à un ensemble de mécanismes de sécurité qui la fournissent.

Considérons la capacité C_1 :

```
C1 := allow_read_access (Type.Passive.Data.File SCFile, ID.User SCUser);
```

C_1 peut être appliquée à l'aide de plusieurs mécanismes. Ainsi, les droits d'accès Unix peuvent être utilisés pour appliquer cette capacité, mais un mécanisme de contrôle d'accès obligatoire, comme SELinux or grsecurity, peut également être utilisé. Trois associations peuvent être définies pour C_1 : l'une avec les droits Unix, l'autre avec SELinux et la dernière avec grsecurity.

Afin de pouvoir sélectionner le mécanisme qui appliquera une capacité, plusieurs éléments sont pris en compte : les **compatibilités** et les **dépendances** entre les mécanismes, s'il s'agit de mécanismes **dominants** ou **récessifs**, et la qualité de l'application de la capacité par chacun des mécanismes évaluée par un **score d'ordonnancement**. Ces différents éléments peuvent être représentés sous forme de matrices qui sont fixées par l'expert en sécurité lors de l'ajout d'un nouveau mécanisme ou d'une nouvelle capacité. L'administrateur de l'architecture logicielle qui définit la politique n'a donc pas connaissance de ces matrices et leur gestion est pour lui entièrement transparente.

3.4.3.1 Compatibilité

Les propriétés (section 3.5) mettent en jeu un ensemble de capacités afin de répondre aux besoins de sécurité. Il faut donc savoir quels mécanismes sont capables de coopérer pour appliquer des capacités, puisque des incompatibilités peuvent exister entre certains d'entre eux.

Définition 3.4.6: Compatibilité de mécanismes
Deux mécanismes M_1 et M_2 sont dits compatibles s'ils peuvent être utilisés simultanément pour appliquer des capacités lors de l'application d'une propriété.

On définit donc une matrice $Comp$ de compatibilité :

$$Comp = \begin{array}{c} \\ M_1 \\ M_2 \\ \vdots \\ M_n \end{array} \begin{array}{c} \begin{array}{cccc} M_1 & M_2 & \cdots & M_n \end{array} \\ \begin{pmatrix} 1 & c_{1,2} & \cdots & c_{1,n} \\ c_{2,1} & 1 & \cdots & c_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{n,1} & c_{n,2} & \cdots & 1 \end{pmatrix} \end{array}$$

où :
- les M_i sont l'ensemble des mécanismes de sécurité ;
- $c_{i,j}$ indique si les mécanismes M_i et M_j sont compatibles :
  - si $c_{i,j} = O$, M_i et M_j sont incompatibles ;
  - si $c_{i,j} = 1$, M_i et M_j sont compatibles.

Cette matrice de compatibilité est donc utilisée pour sélectionner des mécanismes compatibles pour appliquer les différentes capacités formant une propriété. Cependant, dans le cas où plusieurs propriétés s'appliquent sur des ressources liées, il est nécessaire que cette compatibilité puisse être prise en compte. Pour cela, on donne un rôle spécifique à l'attribut `Domain` du langage. Cet attribut est ainsi utilisé pour identifier les différentes ressources appartenant à un domaine applicatif, c'est-à-dire les ressources qui sont liées à une application ou à un service. On utilise alors l'attribut `Domain` pour choisir des mécanismes de sécurité compatibles pour plusieurs propriétés, afin que l'application de la politique soit cohérente.

Notons que la notion de compatibilité introduite ici est une notion locale : elle indique si deux mécanismes peuvent appliquer des capacités d'une même propriété sur un même nœud sans que cela entraîne d'interférence. Une compatibilité distante entre les mécanismes est définie à la section 3.4.4.

De plus, la notion de compatibilité de mécanismes que nous proposons est indépendante des capacités. Il serait possible d'introduire également une notion de compatibilité par capacité, qui permettrait une expression plus fine des compatibilités entre mécanismes. Nous choisissons ici de ne pas prendre en compte ce niveau de précision. En effet, une compatibilité par capacité augmenterait la complexité de la recherche de solution lors de l'application d'une propriété constituée de plusieurs capacités. Cette augmentation de la complexité n'apparait pas justifiée, puisque les matrices de compatibilité par capacité et globale sont similaires : lorsque des mécanismes sont compatibles pour une capacité, ils le sont le plus souvent pour les autres. Une matrice par capacité n'apporte donc pas d'informations supplémentaires suffisantes pour compenser le gain en complexité.

3.4.3.2 Dépendance

Les mécanismes compatibles peuvent être dépendants les uns des autres, c'est-à-dire qu'il peut être nécessaire qu'un mécanisme soit disponible pour pouvoir utiliser le second.

Définition 3.4.7: Dépendance de mécanismes
Un mécanisme M_2 dépend d'un mécanisme M_1 si l'utilisation de M_2 nécessite que M_1 soit disponible sur le nœud considéré.

Considérons ainsi les mécanismes SELinux et sVirt. Ces deux mécanismes sont compatibles, c'est-à-dire qu'ils peuvent être utilisés pour appliquer ensemble une même capacité. SELinux peut être utilisé seul et ne dépend donc d'aucun autre mécanisme. En revanche, pour pouvoir appliquer une capacité, sVirt a besoin que SELinux soit activé. Par conséquent, on dit que sVirt dépend de SELinux.

On peut donc définir la matrice de dépendance Dep :

$$Dep = \begin{array}{c} \\ M_1 \\ M_2 \\ \vdots \\ M_n \end{array} \begin{array}{c} \begin{array}{cccc} M_1 & M_2 & \cdots & M_n \end{array} \\ \begin{pmatrix} 0 & d_{1,2} & \cdots & d_{1,n} \\ d_{2,1} & 0 & \cdots & d_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ d_{n,1} & d_{n,2} & \cdots & 0 \end{pmatrix} \end{array}$$

où :
- les M_i sont les mécanismes de sécurité ;
- $d_{i,j}$ indique si le mécanisme M_i dépend de M_j :
  - si $d_{i,j} = 0$, M_i et M_j sont indépendants ;
  - si $d_{i,j} = 1$, M_i dépend de M_j ;
  - si $d_{i,j} = -1$, M_j dépend de M_i.

Il est à noter que si deux mécanismes sont dépendants, alors ils doivent être compatibles. En effet, si un mécanisme M_1 dépend d'un mécanisme M_2, alors l'utilisation de M_1 implique que M_2 soit disponible, éventuellement pour que M_1 puisse l'utiliser. Par conséquent, on obtient la règle suivante :

Règle 3.4.8: Dépendance et compatibilité
Soient M_1 et M_2 deux mécanismes tels que M_1 dépend de M_2. Alors, M_1 et M_2 sont compatibles. Notons Comp et Dep les matrices de compatibilité et de dépendance. On a alors :

$$[Dep(M_1, M_2) = 1] \Rightarrow [Comp(M_1, M_2) = 1]$$

3.4.3.3 Récessivité et dominance

Certains mécanismes, lorsqu'ils sont utilisés, nécessitent de l'être pour l'ensemble des capacités qu'ils peuvent implémenter lors de l'application d'une propriété. On dit qu'ils sont dominants, au contraire des autres mécanismes qui sont dits récessifs.

Définition 3.4.9: Récessivité et dominance de mécanismes
Un mécanisme M, pouvant appliquer les capacités $C_{i,1 \leq i \leq n}$, est dit récessif si, pour toute paire $(j,k) \in [1,n]$, l'utilisation de M pour l'application de C_j n'implique pas son utilisation pour C_k. Au contraire, M est dit dominant si son utilisation pour l'application de C_j implique son utilisation pour C_k.

On définit la matrice de récessivité R suivante :

$$R = \begin{array}{c} \begin{array}{cccc} M_1 & M_2 & \cdots & M_n \end{array} \\ \begin{pmatrix} r_1 & r_2 & \cdots & r_n \end{pmatrix} \end{array}$$

où :

- les M_i sont les mécanismes de sécurité ;
- r_i indique si le mécanisme M_i est récessif :
  - si $r_i = O$, M_i est récessif ;
  - si $r_i = 1$, M_i est dominant.

Par exemple, si un mécanisme de contrôle d'accès tel que SELinux est utilisé pour appliquer une capacité, il doit l'être pour l'ensemble des capacités qu'il est capable d'appliquer. Il s'agit donc d'un mécanisme dominant. Au contraire, Oscap est un mécanisme récessif : s'il est utilisé pour appliquer une capacité, il ne l'est pas nécessairement pour les autres capacités qu'il fournit.

Tout comme pour la compatibilité des mécanismes, la récessivité et la dominance d'un mécanisme sont utilisées lors de l'application des capacités dans une propriété. Cependant, la matrice de récessivité peut être utilisée pour la sélection de mécanismes entre plusieurs propriétés si celles-ci mettent en jeu des contextes ayant le même attribut `Domain`.

3.4.3.4 Scores d'ordonnancement et d'application

Score d'ordonnancement Pour chaque association entre une capacité et un mécanisme, un score d'ordonnancement est défini. Ce score permet de favoriser un mécanisme par rapport à un autre lors de l'application d'une capacité. Le score attribué à un mécanisme pour une capacité est dépendant du système et des besoins de l'expert en sécurité. Pour cette raison, les scores peuvent être modifiés par cet expert.

Définition 3.4.10: Score d'ordonnancement des mécanismes
Le score d'ordonnancement d'un mécanisme M pour une capacité C correspond à la priorité de l'application de C par M. Ces scores sont modifiables pour pouvoir être adaptés aux conditions spécifiques de l'environnement et aux choix de l'expert en sécurité.

Ainsi, dans le cas de la capacité C_1, chacune des trois associations considérées (avec les droits Unix, avec SELinux et avec grsecurity) se voit attribuer un score reflétant, par exemple, la qualité de la sécurité offerte par le mécanisme ou ses performances.

On obtient ainsi une matrice S indiquant le niveau de protection offert par un mécanisme pour une capacité donnée :

$$S = \begin{array}{c} \\ M_1 \\ M_2 \\ \vdots \\ M_n \end{array} \begin{array}{c} \begin{array}{cccc} C_1 & C_2 & \cdots & C_n \end{array} \\ \begin{pmatrix} s_{1,1} & s_{1,2} & \cdots & s_{1,n} \\ s_{2,1} & s_{2,2} & \cdots & s_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ s_{n,1} & s_{n,2} & \cdots & s_{n,n} \end{pmatrix} \end{array}$$

où :

- les M_i sont les mécanismes de sécurité ;
- les C_i sont les capacités ;
- $s_{i,j}$ est un entier indiquant le score attribué au mécanisme M_i pour la capacité C_j.

Un score nul indique que le mécanisme ne fournit pas la capacité considérée. Plus le score est élevé, plus le mécanisme sera prioritaire pour appliquer la capacité.

Score d'application d'une capacité Chaque mécanisme capable d'appliquer une capacité possède donc un score d'ordonnancement associé à celle-ci. Ainsi, lorsqu'un mécanisme est sélectionné pour appliquer une capacité, on peut extrapoler un score d'application de la capacité basé sur ce score d'ordonnancement. On obtient donc la définition suivante :

Définition 3.4.11: Score d'application de capacité
Soit C_{inst} une capacité instanciée à appliquer.
On note c la capacité compatible avec C_{inst} et choisie pour l'appliquer, c'est-à-dire la capacité pouvant être appliquée et ayant le score d'inclusion $d(c, C_{inst})$ le plus faible (définition 3.4.4).
On note $M_{i,1 \leq i \leq n}$ l'ensemble des mécanismes choisis (et donc compatibles) pour appliquer C_{inst} en utilisant la capacité c. Soit S la matrice des scores d'ordonnancement. $S(M_i, c)$ est donc le score d'ordonnancement de M_i pour c, au sens de la définition 3.4.10.
Le score d'application $Sapp(C_{inst})$ de la capacité instanciée C_{inst} est donc défini par :

$$Sapp(C_{inst}) = \sum_{i=1}^{n} S(M_i, c) * \frac{1}{d(c, C_{inst}) + 1}$$

Grâce au score d'application de capacité, il est possible de comparer le qualité de l'application d'une capacité sur des systèmes différents. Pour cela, on utilise la règle suivante :

Règle 3.4.12: Comparaison des applications de capacité
Soit C_{inst} une capacité instanciée à appliquer. Soient $Sapp(C_{inst})_1$ et $Sapp(C_{inst})_2$ son score d'application sur deux systèmes différents. Alors, l'application ayant le score le plus élevé est la solution la plus proche de l'application idéale, telle que définie par l'expert en sécurité.

On peut donc déterminer la qualité de l'application d'une capacité en utilisant ce score et comparer l'application de capacités dans un environnement hétérogène.

3.4.4 Définition des capacités internes

La notion de capacité que nous avons vue dans les sections précédentes permet d'abstraire les fonctionnalités des mécanismes de sécurité. Cependant, des capacités indépendantes des mécanismes peuvent être utiles afin de fournir des fonctionnalités qui ne sont pas directement liées à l'application ou à l'assurance d'une propriété, mais qui permettent d'exprimer ces propriétés. Nous définissons par conséquent des capacités internes.

Définition 3.4.13: Capacité interne
Une capacité interne est une capacité non associée à un mécanisme et fournie par le logiciel interprétant le langage.

Ces capacités internes n'apportent pas de nouvelles fonctionnalités de sécurité, mais permettent, par exemple, une sélection cohérente des mécanismes ou un échange d'informations entre les différents nœuds.

Considérons une capacité réseau, c'est-à-dire une capacité s'appliquant sur plusieurs nœuds. Il est indispensable que cette capacité soit appliquée sur les différents nœuds en utilisant des mécanismes compatibles (par exemple lors de la création d'un VPN). Pour cette raison, nous définissons la capacité interne IC_1 dont l'objectif est de garantir la cohérence de l'application d'une capacité réseau. IC_1 est définie comme suit :

```
IC1 := consistent_plugin (Net.IP SC1, Net.IP SC2, CapabilityName SCcap,
    ContextList SCargs);
```

où :
- `SC1` et `SC2` sont des contextes contenant l'adresse IP des deux nœuds impliqués dans la capacité ;
- `SCcap` est le nom de la capacité que l'on cherche à appliquer ;

– SCargs est la liste des arguments sur lesquels la capacité doit être appliquée.

Ainsi, l'utilisation de la capacité IC_1 va permettre d'assurer que les nœuds SC1 et SC2 choisissent des mécanismes compatibles pour appliquer la capacité SCcap avec les arguments SCargs. Pour déterminer si des mécanismes sont compatibles pour une capacité donnée, IC_1 utilise une matrice de compatibilité réseau $CompNet$, similaire à la matrice de compatibilité $Comp$ vue précédemment. $CompNet$ est définie comme suit :

$$CompNet = \begin{array}{c} \\ M_1 \\ M_2 \\ \vdots \\ M_n \end{array} \begin{array}{c} \begin{array}{cccc} M_1 & M_2 & \cdots & M_n \end{array} \\ \begin{pmatrix} 1 & c_{1,2} & \cdots & c_{1,n} \\ c_{2,1} & 1 & \cdots & c_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{n,1} & c_{n,2} & \cdots & 1 \end{pmatrix} \end{array}$$

où :
- les M_i sont les mécanismes de sécurité ;
- $c_{i,j}$ indique si les mécanismes M_i et M_j sont compatibles au niveau réseau :
  - si $c_{i,j} = O$, M_i et M_j sont incompatibles ;
  - si $c_{i,j} = 1$, M_i et M_j sont compatibles.

IC_1 peut donc sélectionner des mécanismes compatibles pour appliquer une capacité réseau. Tout comme pour la matrice de compatibilité $Comp$, il serait possible d'étendre la matrice $CompNet$ pour définir une compatibilité réseau par capacité. Cela n'est pas fait puisque cela complexifie la recherche de solution lors de l'application d'une capacité réseau et que la plupart des mécanismes sont compatibles, ou non, pour l'ensemble de leurs capacités.

Des capacités peuvent également avoir besoin d'échanger des informations, par exemple dans le cas de la mise en place d'une architecture client-serveur. Si le serveur choisit dynamiquement le port sur lequel le client doit se connecter, il est nécessaire que le client sache quel port a été choisi. On définit donc les capacités internes IC_2 et IC_3 afin de permettre l'échange de données entre deux nœuds :

```
IC2 := publish (Context SCdata, Net.IP SCtarget);
IC3 := get_published (Net.IP SCsource, Context SCdata);
```

IC_2 permet à un nœud de publier une donnée SCdata dans un annuaire partagé, afin que le nœud cible SCtarget puisse y avoir accès. Le nœud cible peut alors récupérer la donnée en utilisant IC_3. L'implémentation de ces deux capacités doit prendre en considération le besoin de synchronisation : la donnée ne peut être lue qu'après avoir été publiée.

Les capacités internes peuvent également être utilisées pour obtenir des informations spécifiques au système. Par exemple, la capacité IC_4 permet de connaître les adresses IP de la machine locale :

```
IC4 := get_local_IP ();
```

Les propriétés internes permettent donc de répondre à plusieurs types de problématiques, comme la synchronisation des capacités ou l'obtention des caractéristiques du système.

3.4.5 Synthèse

Cette section a introduit la notion de capacité, en considérant à la fois les capacités fournies par les mécanismes et celles internes au langage. Des tables récapitulatives des capacités et des mécanismes qui leur sont associés sont disponibles à l'annexe A.1.

Lors de l'application d'une capacité, les mécanismes sont sélectionnés selon différents critères, représentés sous forme de matrices :

- la **compatibilité** : des mécanismes compatibles peuvent être utilisés ensemble pour appliquer des capacités, au contraire de mécanismes incompatibles ;
- la **dépendance** : un mécanisme dépendant d'un ensemble M non nul de mécanismes ne peut appliquer une capacité que si la totalité des éléments de M sont présents ;
- la **récessivité** : un mécanisme récessif peut être utilisé pour appliquer une capacité sans que cela influe sur le choix des mécanismes pour les autres capacités. Au contraire, un mécanisme dominant appliquera toutes les capacités qu'il fournit dès lors qu'il a été utilisé une fois ;
- le **score d'ordonnancement** : il indique la priorité d'un mécanisme pour une capacité. Ce score peut être extrapolé pour définir un score d'application de la capacité.

Ces différentes matrices sont visibles uniquement par l'expert en sécurité. Elles n'ajoutent donc pas de complexité du point de vue de l'utilisateur, tout en permettent à l'expert de paramétrer le choix des mécanismes. Des exemples de matrices sont donnés au chapitre 6.

De plus, ces matrices sont utilisées par le moteur de projection afin de déterminer le processus d'application des propriétés sur les nœuds, c'est-à-dire quels mécanismes sont utilisés pour appliquer chacune des capacités formant la propriété (chapitre 4).

3.5 Propriétés

Dans les sections précédentes, nous avons introduit les notions de contextes et de capacités, qui permettent respectivement d'abstraire les ressources réelles du système et les fonctionnalités offertes par les mécanismes de sécurité. Dans cette section, nous présentons des propriétés de sécurité qui mettent en jeu les contextes et les capacités afin de proposer des fonctionnalités de sécurité de haut niveau.

Plusieurs niveaux d'abstraction sont définis pour ces propriétés. Tout d'abord, le niveau correspondant à la définition des propriétés : il s'agit d'une définition du fonctionnement interne de la propriété mettant en jeu les capacités (sections 3.5.2 et 3.5.3). Ce niveau n'est visible que par l'expert en sécurité qui peut par exemple modifier ou définir de nouvelles propriétés. Le second niveau correspond aux prototypes des propriétés et est visible lors de la définition d'une politique de sécurité (section 3.5.4). Ce niveau est utilisé par l'administrateur de l'architecture logicielle lorsqu'il définit la politique répondant à ses besoins de sécurité.

3.5.1 Définition des propriétés

Afin d'exprimer les besoins de sécurité, nous utilisons la notion de propriété de sécurité. Une propriété de sécurité permet de répondre à un besoin de sécurité, c'est-à-dire à la fois d'appliquer une protection appropriée et de vérifier que cette protection fonctionne comme attendu. Les propriétés de sécurité doivent donc adresser ces deux éléments et proposer une méthode d'application de chaque besoin ainsi qu'une méthode permettant d'assurer que ce besoin a bien été adressé. L'application et l'assurance d'un besoin de sécurité étant intrinsèquement liées, elles sont toutes deux définies au sein de chaque propriété.

On définit alors une propriété de sécurité de la façon suivante :

Définition 3.5.1: Propriété
Une propriété est une fonctionnalité de sécurité de haut niveau, indépendante des mécanismes, et définissant à la fois son processus d'application et son processus d'assurance.

Afin de fournir des fonctionnalités de sécurité complexes, une propriété de sécurité combine plusieurs capacités élémentaires. Chaque propriété est formée de deux blocs, chacun composé de capacités : un bloc dédié à l'application de la propriété et un bloc pour l'assurance. Une propriété est donc de la forme suivante :

```
1 returnType Name (arguments) {
2  enforcement {
3   [...]    // Bloc d'application
4  }
5  assurance {
6   [...]    // Bloc d'assurance
7  }
8 }
```

Listing 3.3 – Squelette d'une propriété

Le bloc d'application a pour objectif de décrire comment la propriété peut être appliquée pour répondre à un besoin de sécurité. Il est donc composé d'un ensemble de capacités d'application qui, une fois combinées, fournissent une fonctionnalité de sécurité complexe, telles que la confidentialité ou l'intégrité d'une ressource. Le bloc d'assurance vise à définir les tâches à effectuer afin de tester la bonne application d'une propriété. Il s'agit donc de définir des tests dynamiques, mais non intrusifs, qui permettent de déterminer si une propriété est appliquée comme attendu. Les deux blocs d'une propriété mettent en jeu des capacités. Ces différentes capacités étant indépendantes des mécanismes, les propriétés le sont également. Ainsi, en utilisant une propriété, l'utilisateur ne spécifie pas quels seront les mécanismes utilisés pour la garantir, mais uniquement quel est son besoin.

Une propriété peut également faire appel à d'autres propriétés, ce qui permet ainsi de définir des propriétés plus complexes. De plus, les propriétés peuvent être surchargées : plusieurs propriétés peuvent avoir le même nom si elles ont des arguments différents. Ainsi, un même besoin en sécurité s'appliquant sur des ressources de types différents utilisera une propriété spécifique pour chacun des types.

Chaque propriété retourne une valeur de type booléen. Cette valeur permet au moteur de projection de savoir si l'application de la propriété a été correctement effectuée ou si un problème a été rencontré. Dans le cas du bloc d'application, la valeur de retour est, par défaut, définie implicitement comme un `ET` logique des valeurs de retour des capacités appliquées. Ces valeurs de retour peuvent ensuite être utilisées par le moteur d'assurance afin de modifier l'application de la propriété, par exemple en réappliquant la propriété avec des mécanismes différents. Ce processus est décrit au chapitre 4.

Les propriétés de sécurité peuvent être classées en trois catégories : les propriétés systèmes (section 3.5.2.1), les propriétés réseaux (section 3.5.2.2) et les propriétés hybrides (section 3.5.2.3). En effet, une architecture logicielle utilisant l'informatique en nuage peut être vue comme un ensemble d'applications s'exécutant sur des systèmes qui communiquent par le réseau. Par conséquent, la sécurisation de l'architecture logicielle correspond à la sécurisation des systèmes impliqués et des communications réseaux, c'est-à-dire aux propriétés systèmes et réseaux. Les propriétés hybrides associent des propriétés des deux autres catégories afin de fournir une sécurité de bout-en-bout. De plus, le langage propose des classes de propriétés qui permettent de regrouper un ensemble de propriétés partageant un même objectif de protection (section 3.5.3). Il est ainsi possible de simplifier l'utilisation des propriétés, en factorisant l'expression des besoins.

3.5.2 Catégorisation de propriétés

Cette section présente des exemples de propriétés pour illustrer les trois catégories. D'autres exemples sont disponibles à l'annexe A.2.

3.5.2.1 Propriétés système

Les propriétés systèmes sont celles qui s'appliquent sur un seul nœud et n'impliquent pas de communication réseau. Considérons la propriété P1 suivante :

```
1  P1 := boolean Confidentiality_access_control (Type.Passive.Data.File SCFile,
       Id.User SCUser) {
2   enforcement {
3    deny_all_read_accesses (SCFile);
4    allow_read_access (SCFile, SCUser);
5   }
6   assurance {
7    boolean c = true;
8    authorized_users = get_users(SCUser);
9    for (SCFileTmp IN get_files(SCFile) {
10    for (SCUserTmp IN get_all_users()) {
11     if (SCUserTmp IN authorized_users) {
12      c &= check_read (SCFileTmp, SCUserTmp);
13     }
14     else {
15      c &= (NOT check_read (SCFileTmp, SCUserTmp));
16     }
17    }
18   }
19   return c;
20  }
21 }
```

Listing 3.4 – Propriété de confidentialité d'un fichier (contrôle d'accès)

P1 permet de garantir la confidentialité d'un fichier en utilisant un mécanisme de contrôle d'accès. P1 est composée de deux blocs distincts, pour l'application (*enforcement*) et l'assurance. Le bloc d'application fait appel à deux capacités : la première permet d'empêcher les accès en lecture aux fichiers SCFile, tandis que la deuxième autorise ces accès s'ils sont effectués par un utilisateur SCUser. Comme nous l'avons vu précédemment, en fonction des mécanismes qui sont utilisés lors de l'application de la propriété et des matrices de compatibilité et de récessivité, le choix d'un mécanisme pour l'une des capacités peut influencer le choix du second mécanisme. Ceci est transparent pour l'administrateur de l'architecture logicielle : la décision est prise par le moteur de projection en fonction du système, des mécanismes disponibles et des différentes matrices (section 4.4, p. 75).

Le bloc d'assurance est, dans le cas de cette propriété, plus complexe. Pour tester l'application de la propriété, le bloc d'assurance itère sur les fichiers confidentiels et sur l'ensemble des utilisateurs du système. Pour chaque combinaison fichier/utilisateur, un test essaye de lire le fichier et vérifie si le résultat de l'accès correspond à celui attendu (i.e. accès autorisé si le contexte SCUser est inclus dans le contexte de l'utilisateur). L'application de la propriété étant dynamique et dépendant du système et des mécanismes disponibles, le bloc d'assurance permet de vérifier que la propriété a bien été appliquée.

P1 permet de s'abstraire des mécanismes de sécurité puisque chacune des capacités utilisées est indépendante des mécanismes. Par conséquent, selon les mécanismes présents sur le nœud, P1 pourra être appliquée en utilisant des mécanismes différents.

Une propriété peut être utilisée par d'autres propriétés afin de simplifier leur expression. Par exemple, considérons la propriété P2 qui garantit la confidentialité d'un fichier à l'aide de mécanismes de chiffrement. P2 fait appel à P1 afin de contrôler les accès à la clef de chiffrement.

```
1  P2 := boolean Confidentiality_encryption (Type.Passive.Data.File SCFile,
       Id.User SCUser) {
2   enforcement {
3    Context SCKey;
4    SCKey = Type.Passive.Data.Key.Symmetric;
5    SCKey += generate_key (SCKey);
6    Confidentiality_access_control (SCKey, SCUser);
7    encrypt_file (SCFile, SCKey);
8    decrypt_on_access(SCFile, SCUser);
9   }
10  assurance {
11   boolean c = true;
12   authorized_users = get_users(SCUser);
13   for (SCFileTmp IN get_files(SCFile) {
14    for (SCUserTmp IN get_all_users()) {
15     if (SCUserTmp IN authorized_users) {
16      c &= check_encrypted (SCFileTmp, SCUserTmp);
17     }
18     else {
19      c &= (NOT check_encrypted (SCFileTmp, SCUserTmp));
20     }
21    }
22   }
23   return c;
24  }
25 }
```

Listing 3.5 – Propriété de confidentialité d'un fichier (chiffrement)

Le bloc d'application de P2 crée tout d'abord un contexte de clef symétrique SCKey. Une clef est générée et stockée dans ce contexte. L'accès à cette clef est contrôlé grâce à une propriété de contrôle d'accès. Le fichier confidentiel est alors chiffré en utilisant la clef générée. Finalement, le fichier est déchiffré dynamiquement lors d'un accès en lecture par un utilisateur autorisé. Le bloc d'assurance de P2 est similaire à celui de P1 : le test itère sur les fichiers et les utilisateurs, et vérifie si le fichier lu est, ou non, déchiffré à la lecture par un utilisateur légitime.

P1 et P2 explicitent comment elles doivent être appliquées, c'est-à-dire en utilisant des mécanismes de contrôle d'accès ou de chiffrement. Le responsable de la définition de la politique doit donc choisir par quel moyen la propriété sera appliquée, même s'il ne spécifie pas les mécanismes exacts. Cela va à l'encontre du principe de notre langage qui est de définir une politique uniquement basée sur les besoins de sécurité. Les définitions de P1 et P2, deux propriétés spécialisées, peuvent donc être utilisées afin de définir une classe de propriété, plus générale, comme cela est détaillé ultérieurement. Il est ainsi possible de s'abstraire du moyen utilisé pour appliquer les besoins de sécurité.

3.5.2.2 Propriétés réseau

Les propriétés de la section précédente s'appliquent à des ressources systèmes. On définit également des propriétés réseaux qui sont appliquées sur les échanges entre plusieurs nœuds, c'est-à-dire des propriétés qui sécurisent les communications et qui sont appliquées sur l'ensemble des nœuds impliqués. Une telle application pose le problème de la synchro-

nisation des deux applications de la propriété. Ce problème doit donc être adressé par le langage, ce qui est fait en utilisant des capacités internes. On définit ainsi une propriété P3 pour garantir la confidentialité des communications réseau entre un client et un serveur, c'est-à-dire la confidentialité des échanges entre deux adresses IP.

```
1  P3 := boolean Confidentiality (Net.IP SC1, Net.IP SC2) {
2   enforcement {
3    Type.Passive.Data.Key SCKey;
4    if (SC1.Net.IP == get_local_ip()) {
5     SCKey += generate_key (Type.Passive.Data.Key.[Symmetric|Asymmetric]);
6     publish(SCKey, SC2);
7     Plugin plug = consistent_plugin (SC1, SC2, encrypt_flow, SC1, SC2, SCKey);
8     plug->encrypt_flow (SC1, SC2, SCKey);
9    }
10   else {
11    get_published(SC1, SCKey);
12    Plugin plug = consistent_plugin (SC1, SC2, encrypt_flow, SC1, SC2, SCKey);
13    plug->encrypt_flow (SC1, SC2, SCKey);
14   }
15  }
16  assurance {
17   return check_encrypted_flow (SC1, SC2);
18  }
19 }
```

Listing 3.6 – Propriété de confidentialité réseau

La propriété P3 étant une propriété réseau, elle est appliquée sur les deux nœuds SC1 et SC2. Cependant, l'application n'est pas identique sur les deux machines. Pour cette raison, le bloc d'application de P3 est séparé en deux parties, l'une dédiée au nœud client, l'autre au nœud serveur.

Le nœud serveur est celui identifié par le premier argument SC1. Par conséquent, la première partie de la propriété (lignes 4 à 9) concerne le serveur. Une clef de chiffrement est tout d'abord générée et est publiée afin que le client y ait accès : il peut s'agir d'une clef symétrique ou asymétrique, ce qui peut influencer le choix de la capacité de chiffrement. Un mécanisme commun aux deux nœuds est alors choisi en utilisant une capacité interne. Le mécanisme sélectionné est ensuite utilisé pour chiffrer les flux réseau entre les nœuds.

La partie cliente de la propriété (lignes 10 à 14) récupère la clef de chiffrement qui a été publiée : cette partie nécessite des mécanismes de synchronisation, fournis par les capacités internes utilisées. Un mécanisme compatible est alors utilisé pour configurer le chiffrement des flux réseaux sur la partie cliente.

Finalement, le bloc d'assurance (lignes 16 à 18) utilise une capacité permettant de vérifier que les communications réseaux entre les deux adresses IP sont bien chiffrées.

Les propriétés peuvent donc être utilisées pour répondre à des besoins de sécurité impliquant plusieurs nœuds.

3.5.2.3 Propriétés hybrides

Comme nous l'avons vu précédemment, les propriétés systèmes et réseaux s'appliquent respectivement à un système donné ou à des communications entre plusieurs systèmes. Cependant, il est utile de pouvoir définir des propriétés s'appliquant à la fois sur des ressources systèmes et sur des échanges réseaux. En effet, cela permet alors de définir des propriétés de bout-en-bout qui sécurisent les aspects réseaux et systèmes des applications ou services. Nous définissons donc des propriétés hybrides, plus complexes mais protégeant

des cas d'usage plus généraux. Ainsi, la propriété P4 est une propriété hybride de confidentialité. Elle utilise les propriétés P1 et P3 pour assurer la confidentialité d'un fichier sur un nœud vis-à-vis d'un utilisateur sur un nœud distant.

```
1  P4 := boolean Confidentiality (Type.Passive.Data.File SCFile, Net.IP:Net.Port
       SC1, Net.IP:Net.Port SC2, Id.User SCUser) {
2   enforcement && assurance {
3    if (SC1.Net.IP == get_local_IP()) {
4     Confidentiality_Access_Control (SCFile:SC1, SCUser:SC1);
5     Confidentiality (SC1, SC2);
6    }
7    else {
8     Confidentiality (SC1, SC2);
9     Confidentiality_Access_Control (SCFile:SC2, SCUser:SC2);
10   }
11  }
12 }
```

Listing 3.7 – Propriété de confidentialité hybride

P4 attend quatre arguments : SCFile identifie le fichier à protéger. SC1 spécifie le nœud sur lequel est situé le fichier ainsi que le port permettant d'accéder au fichier. SC2 indique le nœud sur lequel est connecté l'utilisateur, et le port par lequel il tente d'accéder au fichier. Finalement, SCUser identifie les utilisateurs autorisés à lire le fichier confidentiel. P4 utilise ensuite des propriétés de confidentialité système et réseau pour sécuriser les différentes parties de cette interaction complexe. La première propriété (appliquée sur le nœud serveur, ligne 4) garantit que le fichier, lorsqu'il est situé sur le serveur, ne peut être lu que par un utilisateur connecté sur cette même machine et ayant un contexte compatible avec SCUser. L'utilisation de la notation :SC1 permet de prendre en compte le port, et donc de sécuriser le lien entre le fichier et le port, si les mécanismes disponibles le permettent. La deuxième propriété (appliquée sur les deux nœuds, lignes 5 et 8) rend confidentielles les communications réseaux entre les machines. Enfin, la dernière propriété (sur le nœud client, ligne 9) vérifie que, lorsqu'une version du fichier est disponible sur la machine cliente, elle ne peut être lue que par un utilisateur autorisé.

Ainsi, en combinant plusieurs propriétés, il est possible de définir des propriétés de sécurité avancées. Il devient alors plus simple de protéger un scénario complet en utilisant ces propriétés hybrides.

3.5.3 Classes de propriétés

Les différentes propriétés qui ont été définies (annexe A.2) peuvent être spécialisées : elles se basent uniquement sur un type d'application, par exemple le contrôle d'accès. Les classes de propriétés permettent de remédier à ce problème.

Définition 3.5.2: Classe de propriétés
Une classe de propriété associe plusieurs propriétés ayant la même sémantique afin de simplifier l'expression des besoins.

Chaque propriété présente dans une classe est associée à un score modifiable, qui indique l'ordre de priorité des propriétés. Plus ce score est élevé, plus la propriété associée est favorisée lors de l'application de la classe de propriétés.

Définition 3.5.3: Score statique de propriété
Le score statique $Sstat(P)$ d'une propriété P est un entier strictement positif permettant de déterminer l'ordre de sélection des propriétés. Un score non précisé prend la valeur par défaut de 1.

De plus, une classe peut en inclure d'autres, formant ainsi une hiérarchie de classes de propriétés. Il s'agit ici d'un héritage simple, sans surcharge.

Une classe de propriété est définie comme décrit au listing 3.8.

```
1  CP := class @ClassName {
2     Score_1 Property_1 (Arguments_1);
3     Score_2 Property_2 (Arguments_2);
4     ...
5     Score_N Property_N (Arguments_3);
6     Class_1;
7     Class_2;
8     ...
9     Class_M;
10 }
```

Listing 3.8 – Squelette d'une classe de propriétés

Notons qu'une classe ne précise pas d'arguments, ce qui lui permet d'inclure des propriétés ayant des arguments différents. Les propriétés seront appliquées en fonctions des arguments passés à l'instance de la classe.

Considérons par exemple les propriétés `P1` et `P2` qui permettent de garantir la confidentialité d'un fichier respectivement par contrôle d'accès et par chiffrement. Ces deux propriétés peuvent être associées dans la classe **`@Confidentiality_System`** qui rassemble l'ensemble des propriétés offrant de la confidentialité au niveau système. Cette classe est décrite dans le listing 3.9 :

```
1 CP1 := class @Confidentiality_System {
2    5 Confidentiality_access_control (SCFile, SCUser);
3    3 Confidentiality_encryption (SCFile, SCUser);
4 }
```

Listing 3.9 – Classe de confidentialité système

La définition de la classe `CP1` combine deux propriétés, `P1` et `P2`, correspondant à deux manières de garantir la confidentialité d'un fichier : soit le fichier n'est accessible que par les utilisateurs autorisés, soit son contenu n'est compréhensible que par ces mêmes utilisateurs. `P1` est ici prioritaire sur `P2` puisque son score est plus élevé : ce comportement peut être modifié en adaptant le score selon les besoins d'un nœud et les préférences de l'expert en sécurité.

Les classes peuvent elles-aussi être combinées afin de former des classes de plus haut niveau. Considérons ainsi la classe `CP2` :

```
1 CP2 := class @Confidentiality {
2    @Confidentiality_System;
3    @Confidentiality_Network;
4 }
```

Listing 3.10 – Classe de confidentialité

`CP2` associe deux autres classes, l'une fournissant des propriétés de confidentialité système, et l'autre des propriétés de confidentialité réseau. La sélection de la propriété appliquée lors de l'utilisation de `CP2` se fait en se basant sur les types des arguments fournis et sur les scores des propriétés au sein des classes **`@Confidentiality_System`** et **`@Confidentiality_Network`**.

3.5.4 Instance de propriétés

3.5.4.1 Prototypes

Les définitions des propriétés sont complexes à écrire et à utiliser. C'est pourquoi elles ne sont visibles que par l'expert en sécurité chargé de définir de nouvelles propriétés ou d'intégrer de nouveaux mécanismes de sécurité. L'administrateur de l'architecture logicielle n'a pas connaissance de la description exacte des propriétés. En effet, le langage vise à ce qu'un non-expert en sécurité puisse définir une politique en se basant uniquement sur les besoins de son architecture. Par conséquent, nous introduisons la notion de prototype de propriété.

Définition 3.5.4: Prototype
Un prototype est la signature d'une propriété ou d'une classe : son nom, son type de retour et, s'il s'agit d'une propriété, ses paramètres (une liste de types de contextes).

Un prototype est donc caractérisé par son nom et la liste de ses arguments (une liste de types de contextes). De plus, chaque prototype est associé à une description textuelle permettant à l'utilisateur de savoir à quels besoins répond la propriété, sans connaître son fonctionnement interne. Par exemple, le prototype de `P1` est :

```
/**
 * Interdit l'acces a l'information de fichiers pour tous les utilisateurs sauf
      ceux specifies
 *
 * @param SCFile    les fichiers confidentiels
 * @param SCUser    les utilisateurs autorises
 */
boolean Confidentiality_Access_Control(Type.Passive.Data.File SCFile, ID.User
    SCUser);
```

Les prototypes sont utilisés pour définir une politique de sécurité, puisqu'ils permettent d'exprimer un besoin sans spécifier comment y répondre.

3.5.4.2 Prototypes de propriétés instanciés

Afin de définir une politique de sécurité qui adresse ses besoins, l'administrateur de l'architecture logicielle utilise les prototypes pour définir des propriétés instanciées prenant en arguments des contextes reflétant son architecture logicielle. La politique se base donc sur les prototypes, sur les contextes identifiant les ressources, sur l'architecture logicielle à protéger et sur ses besoins de sécurité.

Considérons par exemple le cas d'un utilisateur souhaitant garantir la confidentialité des fichiers de configuration mail d'un serveur. Cela peut être obtenu en utilisant la politique suivante :

```
1 mailConfig := (Type.Passive.Data.File.Configuration="conf"):(Domain="mail");
2 mailAdmin := (Id.Role="admin"):(Domain="mail");
3
4 @Confidentiality (mailConfig, mailAdmin);
```

Listing 3.11 – Confidentialité de fichiers de configuration

Le contexte `mailConfig` définit les fichiers de configuration qui doivent être confidentiels. Le contexte `mailAdmin` identifie les utilisateurs autorisés à lire les fichiers protégés. Enfin, la propriété de confidentialité garantit que l'information des fichiers ne peut pas être obtenue par un utilisateur non autorisé. La technique d'application n'est pas spécifiée : selon les

cas, cette propriété pourra être appliquée soit en mettant en place du contrôle d'accès, soit en utilisant du chiffrement.

Le type de mécanisme choisi dépend notamment de l'environnement et des autres propriétés de la politique. La compatibilité des mécanismes est prise en compte, et le rôle particulier de l'attribut `Domain` intervient (section 3.4.3.1, p. 48). Par exemple, si toutes les propriétés d'un domaine font intervenir des utilisateurs humains, on pourra choisir du chiffrement ou du contrôle d'accès. Cependant, si l'une des propriétés du domaine fait intervenir un service, le contrôle d'accès sera choisi afin que l'application puisse toujours accéder à ses fichiers (sauf si le chiffrement et le déchiffrement peuvent être faits de manière transparente).

Par conséquent, il est possible de définir simplement des propriétés de sécurité qui sont indépendantes à la fois des mécanismes et des méthodes d'applications.

3.5.4.3 Score d'ordonnancement et d'application

Lorsqu'une propriété instanciée doit être appliquée, le prototype de cette instance peut être compatible avec plusieurs propriétés. Il est donc nécessaire de pouvoir ordonner les propriétés compatibles avec la propriété instanciée. Pour cela, on définit un score d'ordonnancement, basé sur le score statique des propriétés (définition 3.5.3) et sur le score d'inclusion de la propriété instanciée, défini de manière équivalente au score d'inclusion des capacités (définition 3.4.4). On obtient donc la définition suivante :

Définition 3.5.5: Score d'ordonnancement de propriété
Soit P_{inst} une propriété instanciée à appliquer. Soient $P_{i,1\leq i\leq n}$ les prototypes de propriétés compatibles avec P_{inst} au sens de la définition de compatibilité des capacités (définition 3.4.3). On note $Sstat(P_i)$ le score statique de P_i, d'après la définition 3.5.3 et $d(P_i, P_{inst})$ le score d'inclusion de P_i dans P_{inst} (définition équivalente à la définition 3.4.4). Alors, le score d'ordonnancement $Sord(P_i)$ des P_i est défini par :

$$Sord(P_i) = \frac{Sstat(P_i)}{d(P_i, P_{inst}) + 1}$$

Le score d'ordonnancement est utilisé lors de la projection des propriétés (chapitre 4) afin de déterminer quelle est la meilleure propriété compatible avec la propriété instanciée donnée. Plus ce score est élevé, plus la propriété P_i est adaptée pour appliquer P.

Ce score peut être calculé grâce à l'algorithme 7. Pour chaque argument de la propriété instanciée P_{inst} (lignes 3 à 10), le score d'inclusion par rapport au type de contexte attendu par la propriété P (ligne 4) est calculé (ligne 6). En cas de non inclusion d'un des arguments, une valeur négative est retournée (ligne 8). Si tous les arguments sont inclus, le score statique de la propriété est récupéré (ligne 11) afin d'obtenir le score d'ordonnancement de la propriété P (ligne 12).

De plus, une propriété combine un ensemble de capacités et peut donc potentiellement mettre en jeu un ensemble de mécanismes durant son application. On peut donc extrapoler un score d'application d'une propriété instanciée à partir du score d'application des capacités et du score d'ordonnancement des propriétés sélectionnées.

Définition 3.5.6: Score d'application de propriété
Soit P_{inst} une propriété instanciée à appliquer. Soient $P_{i,1\leq i\leq n}$ les propriétés compatibles avec P_{inst}. On note $Sord(P_i)$ le score d'ordonnancement de P_i (définition 3.5.5). Soient $C_{i,j,1\leq j\leq m}$ les capacités composant P_i. On note $Sapp(C_{i,j})$ le score d'application de $C_{i,j}$

Algorithme 7 Calcul du score d'ordonnancement de la propriété P par rapport à P_{inst}

```
1: function scoreOrdonnancementPropriété (P, P_inst)
2:     inclus ← 0
3:     for all ctx[i] dans P_inst.listeContextes do
4:         argType ← P.argument[i]
5:         if compatibilitéDeContexte(argType, ctx[i]) then
6:             inclus ← inclus + scoreInclusion(argType, Type(ctx[i]))      ▷ Déf. 3.3.11
7:         else
8:             return -1
9:         end if
10:    end for
11:    scoreStatique ← le score statique de P définit dans la classe (par défaut, 1)
12:    return ( scoreStatique / (inclus + 1) )
13: end function
```

(définition 3.4.11). Alors, le score d'application $Sapp(P_{inst})$ de P_{inst} est défini par :

$$Sapp(P_{inst}) = \sum_{i=1}^{n}\sum_{j=1}^{m} Sord(P_i) * Sapp(C_{i,j})$$

$$\Leftrightarrow Sapp(P_{inst}) = \sum_{i=1}^{n}\sum_{j=1}^{m} \frac{Sstat(P_i)}{d(P_i, P_{inst}) + 1} * Sapp(C_{i,j})$$

$$\Leftrightarrow Sapp(P_{inst}) = \sum_{i=1}^{n}\sum_{j=1}^{m} \frac{Sstat(P_i)}{d(P_i, P_{inst}) + 1} * \sum_{l=1}^{q} S(c, M_l) * \frac{1}{d(c, C_{i,j}) + 1}$$

Les scores d'applications des propriétés permettent de comparer les manières dont une même propriété a été appliquée sur différents systèmes.

Règle 3.5.7: Comparaison des applications de propriété
Soit P_{inst} une propriété instanciée à appliquer. Soient $Sapp(P_{inst})_1$ et $Sapp(P_{inst})_2$ son score d'application sur deux systèmes différents. Alors, l'application ayant le score le plus élevé est la meilleure.

3.5.5 Synthèse

Cette section a présenté les propriétés qui permettent de répondre aux besoins de sécurité. Les propriétés sont définies par un expert en sécurité, combinent des capacités et mettent en jeu les contextes. Elles peuvent être de trois sortes : les propriétés systèmes qui sécurisent les ressources d'un système et s'appliquent indépendamment sur les machines, les propriétés réseaux qui protègent les communications entre machines, et les propriétés hybrides qui combinent les deux premiers types afin de sécuriser des interactions de bout-en-bout.

Afin de factoriser l'expression de la politique, les propriétés ayant des objectifs de sécurité similaires peuvent être regroupées en classes. Les classes et les prototypes de propriétés sont utilisés par l'administrateur de l'architecture logicielle pour exprimer les besoins de sécurité de l'architecture logicielle. Les classes et propriétés qui ont été définies sont disponibles à l'annexe A.2.

De plus, chaque propriété se voit attribuer un score d'ordonnancement et d'application indiquant la priorité et la qualité de son application. Ainsi, deux applications différentes d'une même propriété peuvent être comparées, afin de déterminer le niveau de protection effectif apporté par la propriété.

3.6 Politique de sécurité

3.6.1 Définition

Les différents éléments du langage qui ont été définis permettent d'exprimer les besoins de sécurité d'une architecture logicielle. L'ensemble de ces besoins forme la politique de sécurité. La grammaire de la politique est donnée à l'annexe A.3.

Définition 3.6.1: Politique de sécurité
Une politique de sécurité exprime les besoins de sécurité qui doivent être adressés. Elle est formée :
- *des contextes identifiant les ressources à protéger ;*
- *des propriétés mettant en jeu ces contextes afin de répondre aux besoins de sécurité exprimés.*

Considérons l'exemple décrit à la figure 3.5. Cet exemple est composé d'une unique machine utilisée comme serveur mail. Deux domaines sont définis, un pour le serveur mail et un pour le système. Le domaine `mail` regroupe l'ensemble des ressources du serveur mail : les différents fichiers, l'administrateur et le service mail. Le domaine `system` regroupe l'administrateur du système, le service de logs et les fichiers de logs système.

Les actions autorisées sont également indiquées sur la figure. Les flèches bleues correspondent aux propriétés de confidentialité, c'est-à-dire aux propriétés autorisant une opération de lecture pour un sujet, tout en les interdisant pour tous les autres. Les flèches rouges représentent les propriétés d'intégrité qui autorisent les écritures sur un objet par un sujet spécifique. Lorsque le cadre d'une ressource est rouge, une propriété d'intégrité totale s'applique sur cette ressource, empêchant toute écriture. Les flèches violettes indiquent que les lectures et les écritures sont autorisées pour le sujet considéré. La flèche verte autorise l'exécution d'un fichier par un sujet.

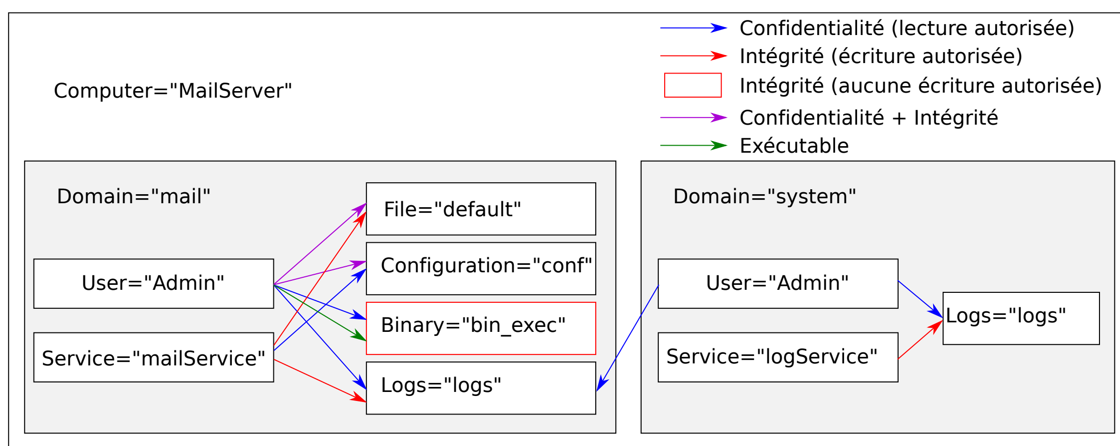

FIGURE 3.5 – Exemple d'architecture logicielle à sécuriser

A partir des besoins de sécurité exprimés dans la figure 3.5, il est possible de définir la politique de sécurité à appliquer. Cette politique reprend les éléments de la figure pour définir les contextes représentant les ressources et les propriétés régissant leurs interactions.

Le listing 3.12 présente donc la politique obtenue. Les trois premiers contextes identifient la machine sur laquelle la politique s'applique et les deux domaines que l'on considère. Le deuxième groupe de contextes (lignes 5 à 9) identifie les différents fichiers : les fichiers liés au serveur mail (ligne 5 à 8) et les logs systèmes (ligne 9). Les lignes 11 et 12 définissent

les contextes pour les services de mail et de log. Enfin, les lignes 14 et 15 définissent les administrateurs des deux domaines.

Les lignes 17 à 25 de la politique définissent les propriétés à appliquer sur les différentes ressources. Tout d'abord, la première propriété (ligne 17) exprime la confidentialité des différents fichiers du domaine mail : la lecture de ces fichiers n'est autorisée que pour l'administrateur mail et, dans le cas des fichiers de configuration, pour le service mail (ligne 18). La troisième propriété (ligne 19) permet la lecture des logs système et mail par l'administrateur système. Les logs mail pourront donc être lus à la fois par les administrateurs mail et système. Les propriétés suivantes sont des propriétés d'intégrité. La propriété de la ligne 20 autorise les écritures sur certains fichiers du domaine mail (fichiers de configuration et divers) par l'administrateur mail. A la ligne 21, la politique interdit toute écriture sur l'exécutable du serveur mail. La ligne 22 autorise l'écriture des fichiers mails par le serveur mail. La propriété à la ligne 23 (resp. ligne 24) autorise quant à elle l'écriture sur les logs mail (resp. les logs système) par le service mail (resp. le service de logs du système). Enfin, la dernière propriété (ligne 25) autorise l'administrateur mail à démarrer le serveur mail : le processus créé à l'exécution du binaire du service mail aura le contexte défini par la table d'association des contextes aux ressources, c'est-à-dire `mailService`. Notons l'existence de la propriété **`Sandbox_App`** (annexe A.2) qui permet d'isoler un domaine en donnant les droits usuels pour la lecture, l'écriture et l'exécution des fichiers. Son utilisation simplifierait l'expression de cette politique.

```
1  mailServer   := (Hardware.Computer="mailServer");
2  mailDomain   := (Domain="mail");
3  systemDomain := (Domain="system");
4
5  mailFiles    := $mailServer:$mailDomain:(Type.Passive.Data.File="default");
6  mailConfig   := $mailServer:$mailDomain:(Type.Passive.Data.File.Configuration="
       conf");
7  mailLogs     := $mailServer:$mailDomain:(Type.Passive.Data.File.Logs="logs");
8  mailBinary   := $mailServer:$mailDomain:(Type.Passive.Data.File.Binary="
       bin_exec");
9  systemLogs   := $mailServer:$systemDomain:(Type.Passive.Data.File.Logs="logs");
10
11 mailService  := $mailServer:$mailDomain:(Type.Active.Service="mailService");
12 logService   := $mailServer:$systemDomain:(Type.Active.Service="logService");
13
14 mailAdmin    := $mailDomain:(Id.Role="Admin");
15 systemAdmin  := $systemDomain:(Id.Role="Admin");
16
17 @Confidentiality (mailFiles | mailConfig | mailLogs | mailBinary, mailAdmin);
18 @Confidentiality (mailConfig, mailService);
19 @Confidentiality (mailLogs | systemLogs, systemAdmin);
20 @Integrity (mailFiles | mailConfig, mailAdmin);
21 @Integrity (mailBinary);
22 @Integrity (mailFile, mailService);
23 @Integrity (mailLogs, mailService);
24 @Integrity (systemLogs, logService);
25 Executable (mailBinary, mailAdmin);
```

Listing 3.12 – Exemple de politique sur un système de type Unix

Une fois la politique de sécurité définie, il est nécessaire de préciser l'association entre les contextes et les ressources réelles. Un exemple est fourni au listing 3.13.

```
1 o /etc/postfix(/.*)?         mailFiles
2 o /etc/postfix/main.cf       mailConfig
3 o /etc/postfix/master.cf     mailConfig
```

```
4  o /var/log(/.*)?           systemLogs
5  o /var/log/mail.log        mailLogs
6  o /usr/sbin/postfix        mailBinary
7
8  u postfixAdmin             mailAdmin
9  u root                     systemAdmin
10
11 p /usr/sbin/postfix        mailService
12 p /sbin/rsyslogd           logService
13
14 c 192.168.0.10             mailServer
```

Listing 3.13 – Exemple de politique sur un système de type Unix

Si un autre serveur mail est utilisé ou si la politique est appliquée sur un système différent, seul ce fichier a besoin d'être modifié pour prendre en compte l'ensemble des changements. Il est donc simple d'adapter la politique à un autre système ou à une autre application similaire. On a ainsi une politique de sécurité indépendante du système et de l'application utilisée, à l'exception du fichier d'associations. Dans le cas des applications usuelles, ces associations peuvent être prédéfinies par système et par application. De plus, dans un nuage informatique, de nombreuses machines virtuelles sont déployées à partir d'une même image : elles disposent donc d'une même configuration et peuvent partager ce fichier d'associations.

3.6.2 Score d'application

A partir des scores d'application des propriétés définis à la section 3.5.4, il est possible de déterminer la qualité de l'application d'une politique en fonction d'un système et d'une configuration donnés. En effet, pour une même politique, on peut comparer la qualité de l'application en se basant sur les applications de chacune de ses propriétés.

Définition 3.6.2: Score d'application de la politique
Soit P une politique de sécurité. On note $p_{i,1\leq i\leq n}$ les n propriétés instanciées dont est formée P. On définit le score d'application $Sapp(P)$ de la politique par :

$$Sapp(P) = \sum_{i=1}^{n} Sapp(p_i)$$

$$\Leftrightarrow Sapp(P) = \sum_{i=1}^{n}\sum_{j=1}^{m}\sum_{k=1}^{p} Sord(p_{i,j}) * Sapp(C_{i,j,k})$$

$$\Leftrightarrow Sapp(P) = \sum_{i=1}^{n}\sum_{j=1}^{m}\sum_{k=1}^{p} \left(\frac{Sstat(p_{i,j})}{d(p_{i,j}, p_i) + 1} * Sapp(C_{i,j,k})\right)$$

$$\Leftrightarrow Sapp(P) = \sum_{i=1}^{n}\sum_{j=1}^{m}\sum_{k=1}^{p} \left(\frac{Sstat(p_{i,j})}{d(p_{i,j}, p_i) + 1} * \sum_{l=1}^{q} S(c, M_l) * \frac{1}{d(c, C_{i,j,k}) + 1}\right)$$

Il est donc possible de comparer la qualité de l'application d'une politique sur plusieurs systèmes en utilisant son score d'application et la règle suivante :

Règle 3.6.3: Comparaison d'applications de politique
Soit P une politique de sécurité. Soient $Sapp(P)_1$ et $Sapp(P)_2$ son score d'application sur deux systèmes différents. Alors, plus le score est élevé, meilleure est l'application.

Des exemples de scores d'application seront donnés dans le chapitre 6.

3.7 Conclusion

Dans ce chapitre, nous avons présenté un langage d'expression des besoins de sécurité, visant à être utilisé dans des systèmes hétérogènes tels que ceux de l'informatique en nuage. Ce langage a pour objectif de répondre aux problèmes spécifiques liés aux environnements hétérogènes. Il est donc indépendant des ressources à protéger et des mécanismes de sécurité, simple d'utilisation, adaptatif et extensible.

Nous avons tout d'abord introduit la notion de contexte, qui permet d'identifier les ressources de manière indépendante du système de nommage utilisé sur la machine. Grâce à ces contextes, le langage est indépendant des ressources : la politique exprimant les besoins de sécurité de ces ressources est indépendante de leur nom ou des applications utilisées. La politique de sécurité est ainsi portable, puisqu'une même politique peut être utilisée sur des systèmes différents. La politique est complétée par une table d'associations des contextes aux ressources réelles. Cette table dépend du système et des applications utilisées, mais elle peut cependant être générée automatiquement et des tables prédéfinies peuvent être fournies pour les applications usuelles.

Nous avons ensuite défini les capacités qui correspondent aux fonctionnalités des mécanismes de sécurité. Chaque capacité peut être fournie par plusieurs mécanismes de sécurité, permettant ainsi au langage d'en être indépendant. Par conséquent, la politique peut être appliquée sur un système par des mécanismes différents, selon ceux qui sont disponibles. Plusieurs contraintes sont définies sur les mécanismes, permettant de paramétrer leur sélection selon leurs compatibilités, leurs dépendances et leurs statuts de dominance.

Nous avons également présenté les propriétés qui combinent des capacités afin de fournir des fonctionnalités de sécurité et d'assurance de haut niveau. Ces propriétés sont simples à utiliser, notamment en raison des classes qui peuvent les regrouper et donc simplifier la définition de la politique. En effet, l'utilisation des classes permet au langage d'être adaptatif, puisqu'un appel à une classe génèrera un appel à une propriété en fonction du type des arguments. De plus, les propriétés peuvent être surchargées, s'adaptant ainsi au type de leurs arguments. Enfin, de nouvelles capacités, propriétés et classes peuvent être définies par un expert sécurité, rendant ainsi le langage extensible.

Enfin, nous avons proposé un score d'ordonnancement permettant de sélectionner les capacités et les propriétés les mieux adaptées, et un score d'application permettant d'évaluer l'application des propriétés de sécurité. De plus, les relations entre les différents mécanismes sont gérées grâce à des matrices décrivant leurs compatibilités, dépendances et dominances.

Le langage proposé possède donc l'ensemble des caractéristiques nécessaires à son utilisation dans un système hétérogène. De plus, il propose deux vues permettant une expression simple des besoins de sécurité par l'administrateur de l'architecture logicielle et de la politique, tout en donnant la possibilité à un expert en sécurité d'étendre le langage pour adresser un plus large ensemble de problèmes de sécurité. L'administrateur n'a donc pas à se soucier des mécanismes qui sécuriseront son système, mais seulement à se focaliser sur l'expression de ses besoins, en identifiant les ressources à protéger (contextes) et leurs besoins de sécurité (propriétés). Les scores d'application permettent ensuite à l'administrateur d'avoir un retour sur la qualité de l'application de la politique. Notons qu'une politique peut être exprimée à partir des besoins définis grâce à des méthodes d'analyse des risques (section 2.2.3) et pourrait être transposée dans l'un des langages de la section 2.1.2. Une architecture pouvant appliquer une politique exprimée dans ce langage sera détaillée au chapitre 4 et son implémentation sera décrite au chapitre 5.

Chapitre 4

Architecture et Projection

Dans le chapitre précédent, nous avons défini un langage d'expression des besoins de sécurité pour une architecture en nuage. Afin de pouvoir effectivement protéger l'architecture logicielle, les propriétés de sécurité exprimées dans ce langage doivent être traduites et appliquées en utilisant les mécanismes disponibles.

L'application de la politique se faisant dans un environnement d'informatique en nuage, les systèmes concernés sont hétérogènes. De plus, les mécanismes présents pour appliquer les propriétés varient d'un système à l'autre. Par conséquent, il est nécessaire de disposer d'une architecture capable de compiler une politique de sécurité et de la projeter en s'adaptant à l'environnement. En outre, l'informatique en nuage étant un environnement dynamique, l'architecture proposée doit être capable de détecter d'éventuels dysfonctionnements et de prendre les décisions appropriées pour y répondre. La définition d'une telle architecture est l'objectif de ce chapitre, qui présentera les différentes étapes du cycle de vie d'une politique.

Ce chapitre commencera par la description de l'architecture globale dans laquelle s'inscrivent les travaux (section 4.1) avant de présenter l'architecture générale du *Security Enforcement Engine* (section 4.2) qui est chargé d'appliquer et d'assurer la politique de sécurité. Les sections suivantes détailleront les différents éléments du *Security Enforcement Engine* : le gestionnaire de politique (section 4.3) qui maintient à jour la politique de sécurité et la configuration interne, le moteur d'application (section 4.4) qui compile et projette la politique, le moteur d'assurance (section 4.5) qui traite les informations d'assurance qui lui sont transmises, et le module de scores (section 4.6) qui évalue la qualité de l'application de la politique. Le chapitre 5 présentera l'implémentation de cette architecture.

4.1 Architecture globale

L'architecture proposée dans ce chapitre a pour objectif d'implémenter et de projeter le langage d'expression des besoins de sécurité (chapitre 3) sur des mécanismes de sécurité (section 2.3) afin d'appliquer la politique de sécurité (section 3.6). L'architecture permettant de réaliser cette projection est décrite à la figure 4.1 et est basée sur l'architecture proposée dans le cadre du projet Seed4C.

Cette architecture est basée sur un modèle multi-agents [Ferber, 1999, Wooldridge, 2009]. Un agent est présent sur chaque nœud, c'est-à-dire sur chaque machine physique et sur chaque machine virtuelle impliquée dans une architecture logicielle.

L'architecture de Seed4C peut être séparée en six composants principaux :

1. Un outil de modélisation permettant de représenter graphiquement la politique de sécurité ;

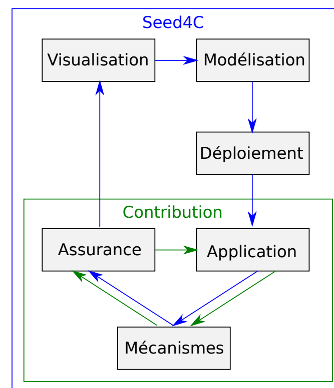

FIGURE 4.1 – Architecture globale basée sur l'architecture Seed4C

2. Un outil de déploiement qui transmet les politiques aux nœuds concernés ;
3. Un moteur d'application qui applique la politique de sécurité ;
4. Un moteur d'assurance qui vérifie que les propriétés sont correctement appliquées ;
5. Des mécanismes de sécurité, utilisés aux étapes 3 et 4, afin d'effectuer les phases d'application et d'assurance ;
6. Un tableau de bord pour visualiser les résultats de l'assurance.

L'architecture proposée dans cette thèse est fortement basée sur l'architecture définie par l'ensemble des partenaires du projet Seed4C et dont l'approche est détaillée dans [Betgé-Brezetz *et al.*, 2013]. Ainsi, l'apport de cette thèse est formé des parties d'application et d'assurance des propriétés. La principale différence entre l'architecture Seed4C et celle présentée ici est l'utilisation qui est faite des données d'assurance. Dans le cadre du projet, les informations d'assurance sont récupérées et traitées par des outils de visualisation développés par des partenaires et peuvent être renvoyées à l'outil de modélisation. Dans l'architecture que nous utilisons, les informations d'assurance sont directement interprétées par l'agent afin de mettre à jour la politique de chaque nœud en fonction des événements reçus.

De plus, les outils de modélisation et de déploiement (développés par des partenaires de Seed4C) ne sont utilisés que lors de la phase d'initialisation du système. Bien que notre architecture puisse les utiliser, elle en est indépendante et utilise uniquement la politique définie dans le langage du chapitre 3 (cette politique peut aussi être générée par l'outil de modélisation).

L'architecture générale, incluant les phases depuis la modélisation jusqu'à l'application de la politique, a été présentée dans [Bobelin *et al.*, 2014]. L'application des politiques a été détaillée dans [Bousquet *et al.*, 2014] et le processus d'assurance dans [Bobelin *et al.*, 2015].

Dans les sections suivantes, les outils de modélisation et de déploiement sont brièvement introduits, afin de décrire le processus global, mais ne sont pas détaillés car ils ne sont pas l'objet de ce document. Le reste de ce chapitre portera sur les moteurs d'application et d'assurance de la politique.

4.1.1 Modélisation et déploiement de la politique

Le langage proposé permet d'abstraire les ressources à protéger et les mécanismes utilisés afin d'appliquer une politique de sécurité. Cependant, dans le cadre de l'informatique en nuage, l'architecture logicielle déployée n'est connue que de l'administrateur, et non pas de

l'expert en sécurité. L'administrateur de l'architecture logicielle n'étant pas nécessairement expert en sécurité, définir formellement les propriétés dont il a besoin peut être complexe.

Pour cette raison, l'architecture globale de Seed4C repose sur un outil de modélisation permettant de définir graphiquement la politique de sécurité. En effet, la politique peut être vue comme un ensemble d'objets (les ressources) interagissant entre eux. Les interactions autorisées sont représentées par les propriétés. L'administrateur connaissant son architecture logicielle, il est en mesure de définir les ressources présentes. Une fois ces ressources définies, l'administrateur peut exprimer ses besoins de sécurité en définissant les propriétés nécessaires entre les ressources. De plus, des modèles peuvent être prédéfinis pour les systèmes et les applications usuels afin de simplifier la définition des ressources et l'expression des besoins de sécurité.

La modélisation de la sécurité d'une architecture logicielle est implémentée par Sam4C [1] [Lefray *et al.*, 2013a], un outil développé par A. Lefray (INRIA) dans le cadre du projet Seed4C. Cet outil permet de représenter graphiquement les différents éléments d'une architecture logicielle, par exemple les machines hôtes et les machines virtuelles, les services et les applications, les fichiers, les utilisateurs et les ressources réseau. Lorsque l'ensemble des éléments a été représenté, il est possible de définir les propriétés à appliquer entre les différents éléments afin de répondre aux besoins de sécurité.

La modélisation des propriétés de la politique est faite pour l'ensemble des machines (physiques et virtuelles) impliquées. La phase de déploiement est également effectuée par l'outil Sam4C et correspond à la distribution de sous-ensembles de propriétés aux différents nœuds. Ainsi, un nœud ne reçoit que les propriétés qu'il doit appliquer.

Ces outils de modélisation et de déploiement simplifient l'élaboration d'une politique de sécurité puisqu'ils permettent de la définir graphiquement et pour l'ensemble des nœuds impliqués. Ils ne sont cependant pas indispensables, puisqu'il est possible de définir ces politiques textuellement, même si cette opération est plus complexe.

4.1.2 Application et assurance

Le moteur d'application est responsable de l'application de la politique de sécurité qu'il reçoit de l'outil de déploiement. Par conséquent, le moteur d'application est chargé de projeter les propriétés définies dans cette politique sur les mécanismes de sécurité disponibles sur le nœud : il doit interpréter les propriétés afin de configurer les mécanismes de sécurité disponibles sur le système.

Le moteur d'assurance a pour objectif de vérifier que la politique de sécurité a été correctement appliquée. Il doit pouvoir décider d'une éventuelle mise à jour de la politique, en accord avec les résultats obtenus à la suite de l'analyse des informations d'assurance.

Dans notre cas, ces deux moteurs sont regroupés au sein d'un unique agent, appelé le *Security Enforcement Engine* (SE^E) [2].

L'architecture Seed4C prévoit que les résultats des tests d'assurance soient transmis à un outil d'analyse et de visualisation. Cela permet à l'administrateur de l'architecture logicielle d'obtenir des informations sur l'état de la sécurité de son système. Il peut ainsi mettre à jour son modèle afin d'adapter la sécurité si nécessaire. Dans notre cas, la mise à jour de la politique est faite automatiquement par le SE^E et ce processus sera décrit dans ce chapitre. Enfin, le SE^E évalue la qualité de l'application en utilisant les scores du chapitre 3, qui sont également transmis à l'administrateur de l'architecture logicielle.

1. `http://perso.ens-lyon.fr/arnaud.lefray/projects/sam4c.html`
2. Durant le projet Seed4C, le terme SE^E était utilisé pour *Secure Element Extended* en raison du rôle primordial des *Secure Element* (SE) dans le projet et du concept commun d'abstraction des fonctionnalités de sécurité. Ce nom a été modifié dans cette thèse suite à l'évolution du rôle du SE^E.

4.2 Architecture du SE^E

L'architecture du SE^E est basée sur les architectures autonomes, qui sont adaptées au côté dynamique des environnements en nuage. Cette section commence donc par décrire ce type d'architecture, avant de présenter l'architecture du SE^E.

4.2.1 Architectures autonomes

4.2.1.1 Définition

L'informatique autonome a été introduite en 2001 par IBM [Horn, 2001] afin de répondre à la difficulté croissante de l'administration des systèmes. La définition donnée par [Jacob *et al.*, 2004] est la suivante :

Définition 4.2.1: Informatique autonome [Jacob *et al.*, 2004]
Autonomic computing is the ability of an IT infrastructure to adapt to change in accordance with business policies and objectives.
[L'informatique autonome est la capacité d'une infrastructure informatique à s'adapter aux changements en fonction de politiques du client et de ses objectifs.]

Un système autonome est donc un système capable de s'administrer lui-même, c'est-à-dire de détecter des évolutions et d'y réagir en fonction d'un ensemble de connaissances.

4.2.1.2 Propriétés

Le terme d'informatique autonome vient de la biologie et notamment des fonctions autonomes du corps humain (le système nerveux autonome) non soumises au contrôle volontaire. De la même manière que pour les fonctions biologiques autonomes, l'informatique autonome doit comporter des propriétés spécifiques, définies par IBM [Kephart et Chess, 2003, Bantz *et al.*, 2003, Huebscher et McCann, 2008] : l'auto-configuration, l'auto-réparation, l'auto-optimisation et l'auto-protection.

Auto-configuration L'auto-configuration (*self-configuration*) est une caractéristique des systèmes qui se configurent automatiquement en fonction de politiques haut niveau spécifiant l'objectif voulu, et non comment l'atteindre. De plus, lorsqu'un nouveau composant est introduit, le système s'adapte à son insertion (tout comme le corps humain s'adapte à la présence d'une nouvelle cellule). L'auto-configuration adresse le problème de l'installation, la configuration et l'intégration à grande échelle de systèmes complexes. En effet, ce problème peut être difficile, demande du temps et est souvent sujet à erreur, même lorsqu'il est traité par des experts. L'automatisation de ces tâches permet donc d'y répondre.

Auto-réparation Les systèmes autonomes détectent, diagnostiquent et réparent des problèmes localisés résultants d'erreurs dans les éléments logiciels ou matériels. Un composant du système autonome chargé de diagnostiquer les problèmes utilise ses connaissances sur la configuration du système afin d'analyser les informations des fichiers de logs et d'éventuels capteurs supplémentaires. Le système peut alors soit résoudre le problème, soit alerter un intervenant humain si aucune solution n'est connue. L'auto-réparation permet donc de réagir à des erreurs se produisant dans des systèmes informatiques complexes, sans nécessiter d'intervention humaine, potentiellement coûteuse en terme de temps et de ressources.

Auto-optimisation Les systèmes autonomes cherchent continuellement à améliorer leurs performances ou leur coût. Cette optimisation se fait en fonction de la politique définie (le système doit-il favoriser les performances ou les coûts ?) et en fonction de l'évolution du système. L'auto-optimisation permet ainsi d'éviter au système des états de surcharge ou de sous-charge pouvant lui être préjudiciables.

Auto-protection L'auto-protection est la dernière des quatre propriétés des systèmes autonomes. Tout d'abord, les systèmes autonomes doivent se protéger contre les problèmes résultants des attaques malicieuses ou des erreurs qui n'ont pas été traitées par le processus d'auto-réparation. En second lieu, ils doivent anticiper les problèmes pouvant survenir afin de s'en protéger. L'auto-protection a donc pour but de permettre au système d'anticiper et de réagir aux différents types de menaces.

4.2.1.3 Architecture

IBM a proposé une architecture de référence [IBM, 2003], communément appelée la boucle MAPE-K (*Monitor, Analyse, Plan, Execute, Knowledge*), afin d'atteindre les objectifs fixés par l'informatique autonome. La boucle MAPE-K est présentée à la figure 4.2.

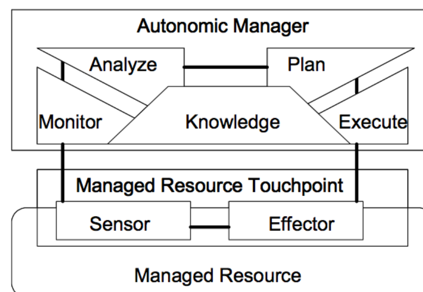

FIGURE 4.2 – Architecture autonome : boucle MAPE-K [Jacob *et al.*, 2004]

La boucle MAPE-K comprend un agent intelligent, le **gestionnaire autonome**, qui perçoit son environnement au moyen de capteurs et agit sur ce même environnement à travers des effecteurs.

Dans cette boucle, les **ressources** gérées peuvent être des éléments logiciels ou matériels. Il peut donc s'agir, par exemple, d'un serveur Web, d'une base de données, d'un composant logiciel, d'une application, d'un ensemble de machines dans une grille, d'un processeur, etc.

Les **capteurs** collectent des informations concernant les ressources qui sont gérées par le système autonomes. Selon le type des ressources, les informations récoltées diffèrent : il peut par exemple s'agir du temps de réponse d'un serveur Web, du taux d'utilisation d'un disque ou de la bande passante, de l'utilisation de la mémoire, etc. De nombreux travaux s'intéressent à l'obtention d'informations donnant l'état d'un serveur [Roblee *et al.*, 2005, Diao *et al.*, 2005, Xu *et al.*, 2005]. Les **effecteurs** sont chargés d'appliquer, sur les ressources, les décisions prises par le gestionnaire autonome. Les changements à effectuer peuvent varier depuis l'ajout ou la suppression d'un serveur dans une ferme de calcul [Schmerl et Garlan, 2002] jusqu'à des changements des paramètres de configuration d'un serveur Web [Sterritt *et al.*, 2005, Bigus *et al.*, 2002]. Les capteurs et les effecteurs forment l'unique point de contact entre le gestionnaire autonome et les ressources. Le gestionnaire n'interagit donc pas directement avec les ressources.

Le **gestionnaire autonome** [Huebscher et McCann, 2008] est donc chargé de contrôler les informations fournies par les capteurs afin d'exécuter des changements sur les ressources au travers des effecteurs. Le gestionnaire est un composant logiciel qui peut être configuré par des administrateurs humains en utilisant des objectifs haut niveau. Il utilise alors les informations des capteurs et sa connaissance du système pour décider des actions bas-niveau à effectuer pour atteindre ces objectifs haut niveau, tout en suivant la boucle MAPE-K. Cette boucle utilise une base de connaissances, qui correspond notamment aux objectifs fixés, à la description du système et aux informations des capteurs. Le gestionnaire surveille (*Monitor*) les informations reçues et les analyse (*Analyse*) afin, par exemple, de mettre à jour la base de connaissance. Il peut alors décider des actions à effectuer (*Plan*) et les exécuter (*Execute*) afin que les objectifs soient atteints.

Plusieurs éléments autonomes peuvent coopérer afin d'atteindre un objectif commun [IBM, 2003]. Par exemple, des serveurs dans une ferme de calcul peuvent chercher à minimiser le temps de réponse ou d'exécution des applications en optimisant l'allocation des ressources. On retrouve donc, dans cette notion de coopération d'éléments individuels visant un but commun, un aspect fondamental des systèmes multi-agents. Par conséquent, de nombreuses recherches s'intéressent à l'utilisation de systèmes multi-agents pour la coopération d'éléments autonomes [Kephart et Chess, 2003, Jennings, 2000, Kumar et Cohen, 2000]. Ces travaux se concentrent en particulier sur le problème rencontré par les systèmes multi-agents qui vise à garantir que le comportement résultant de la coopération des agents répond bien à l'objectif fixé.

Différents framework permettent d'implémenter des architectures autonomes et la boucle MAPE-K. On peut notamment citer les frameworks AME [Jacob *et al.*, 2004] et ABLE [Bigus *et al.*, 2002] d'IBM, ainsi que FOCALE [Strassner *et al.*, 2006, Strassner *et al.*, 2008] et JADE [Claudel *et al.*, 2006, Bouchenak *et al.*, 2011]. Cependant, la sécurité ne fait pas partie de leurs objectifs principaux, ce qui limite leur utilisation dans ce cadre.

4.2.2 Architecture du SE^E

L'architecture globale du SE^E, présentée à la figure 4.3, regroupe les moteurs d'application et d'assurance au sein d'un même agent autonome utilisant une boucle MAPE-K.

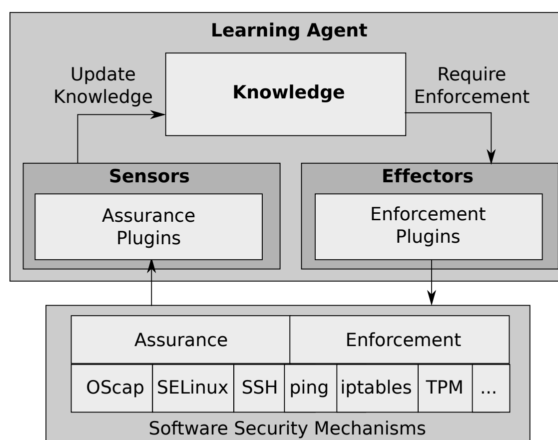

Figure 4.3 – Architecture globale du SE^E

Le SE^E est un agent et l'ensemble des SE^E forme un système multi-agents. L'agent doit déterminer comment les propriétés reçues peuvent être appliquées. Cette décision se base sur la politique elle-même, mais également sur les informations fournies par le processus

d'assurance et sur les mécanismes de sécurité disponibles. Ces différentes informations forment la base de connaissances de l'agent.

De plus, l'agent SE^E suit les concepts de l'architecture autonome proposée par IBM. La partie principale du SE^E est l'agent apprenant et collaboratif, qui correspond au gestionnaire autonome de l'architecture d'IBM. Cet agent est chargé de décider comment les propriétés de la politique (éventuellement reçues de l'outil de déploiement) sont appliquées. La décision se base sur l'ensemble des connaissances du SE^E.

Le SE^E gère un ensemble de mécanismes de sécurité, appelés SSM (*Software Security Mechanisms*). Les SSM sont donc les mécanismes fournissant les capacités d'application et d'assurance du langage. Un mécanisme peut supporter uniquement des capacités d'application ou d'assurance, mais il peut également offrir des capacités des deux types. Les mécanismes et le SE^E interagissent grâce à des modules d'extension (*plugins*), qui correspondent aux effecteurs et aux capteurs d'un agent autonome. Chaque module d'extension (ou simplement, module) pilote un mécanisme de sécurité. Les modules d'application (les effecteurs) sont chargés d'utiliser les capacités des mécanismes associés pour appliquer les propriétés (bloc d'application). Les modules d'assurance (les capteurs) remontent les résultats des tests et les preuves de l'application des propriétés au gestionnaire (bloc d'assurance).

L'utilisation de modules d'extension, c'est-à-dire de composants modulaires pouvant être ajoutés ou supprimés selon les besoins, permet à l'architecture d'être évolutive et adaptative. En effet, pour intégrer de nouveaux mécanismes, le moteur de projection n'a pas besoin d'être modifié : seul un nouveau module fournissant des capacités doit être ajouté. L'architecture peut dont évoluer pour supporter de nouveaux mécanismes, associés à des capacités existantes ou nouvelles. De plus, chaque module étant associé à un mécanisme, il est spécialisé et s'adapte aux spécificités du mécanisme considéré.

L'architecture interne du SE^E est présentée à la figure 4.4.

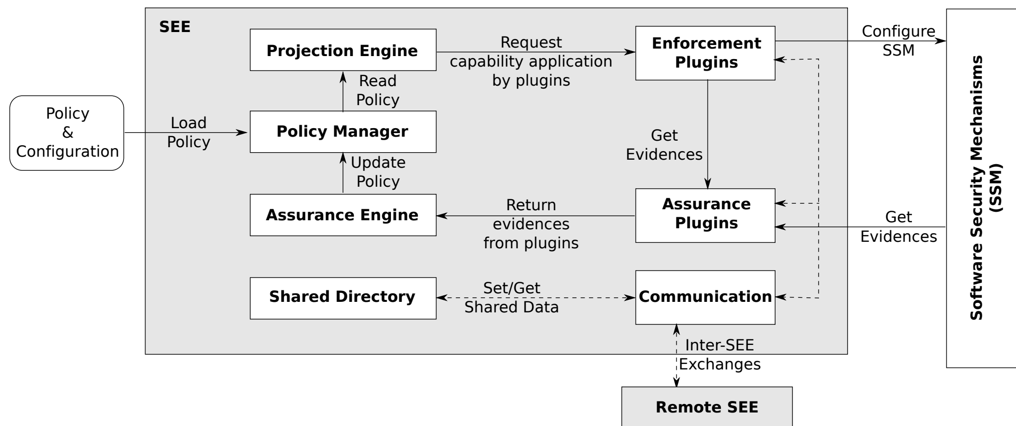

FIGURE 4.4 – Architecture interne du SE^E

Le SE^E est composé des éléments suivants :

1. Le **gestionnaire de politique** charge la politique et la met à jour si nécessaire ;
2. Le **moteur d'application** sélectionne les mécanismes pour appliquer les capacités de chaque propriété ;
3. Le **moteur d'assurance** récupère des informations concernant l'application des propriétés et met à jour la politique en conséquence ;

4. Les **modules d'application** configurent les mécanismes de sécurité en accord avec la politique établie ;
5. Les **modules d'assurance** récupèrent les informations fournies par les mécanismes concernant l'application de la politique.
6. Le **module de communication** permet les communications entre les SE^E, mais aussi entre les modules d'un même SE^E. Il utilise pour cela un **annuaire partagé**.

Ainsi, le SE^E est un agent autonome qui interprète une politique de sécurité afin de l'appliquer. Il se charge également de vérifier la bonne application de la politique en utilisant des procédés d'assurance, et met à jour sa base de connaissances en fonction des résultats obtenus. Ses différents composants sont décrits ci-après.

4.3 Le gestionnaire de politique

Le **gestionnaire de politique** est responsable du chargement et de la mise à jour de la politique. Il est décrit à la figure 4.5.

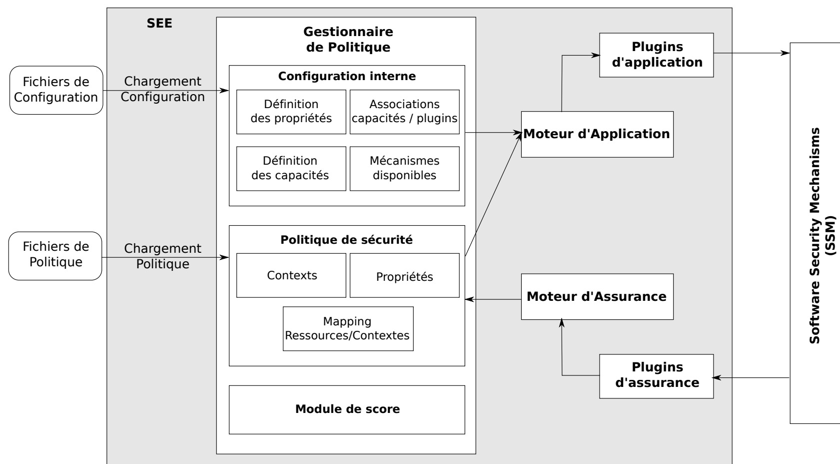

FIGURE 4.5 – Gestionnaire de politique

La base de connaissances du SE^E est composée de plusieurs éléments, regroupés en trois catégories :

- La **configuration interne** du SE^E ;
- La **politique de sécurité** à appliquer ;
- Le **module de score** évaluant la qualité de l'application de la politique.

Ces différents éléments sont utilisés par les autres composants du SE^E afin de calculer une solution pour appliquer la politique de sécurité en fonction des définitions des propriétés, des mécanismes disponibles, de l'état du système, etc.

La **configuration interne** du SE^E correspond aux définitions des éléments utilisés pour exprimer la politique de sécurité. Il s'agit donc des définitions des capacités et des propriétés exprimées dans le langage du chapitre 3. De plus, la configuration interne contient les associations entre les capacités et les modules d'extension (et donc les mécanismes de sécurité), c'est-à-dire l'ensemble des modules capables d'appliquer une capacité. Enfin, la liste des mécanismes disponibles sur le nœud considéré fait également partie de la configuration du SE^E. La configuration interne est prédéfinie et peut être adaptée par l'expert en sécurité, mais non par l'administrateur de l'architecture logicielle qui n'a pas connaissance du fonctionnement interne du SE^E. Les différents éléments de la configuration interne sont

bas niveau et directement liés à l'application effective des propriétés. Des connaissances en sécurité sur les fonctionnalités offertes par les mécanismes, sur la configuration du système et sur les différentes manières d'appliquer une propriété sont donc nécessaires. Or, l'objectif du SE^E étant d'abstraire cette complexité pour l'administrateur de l'architecture logicielle, celui-ci n'a pas accès à ces informations et ne fait que les utiliser par l'intermédiaire de la politique de sécurité.

La **politique de sécurité** indique quels sont les éléments de l'architecture logicielle que l'on cherche à protéger et quels sont les besoins de sécurité qui doivent s'appliquer sur chacun d'eux. Elle se compose de trois types d'éléments : la définition des *contextes* permettant d'identifier les ressources de manière indépendante du système de nommage, les *propriétés de sécurité* s'appliquant sur ces contextes afin de protéger les ressources correspondantes, et l'*association* des contextes aux ressources réelles spécifiant sur quelles ressources les propriétés s'appliquent effectivement. La politique de sécurité est définie par l'administrateur de l'architecture logicielle qui utilise les éléments de la configuration interne mis à sa disposition (les prototypes de propriétés).

Le **module de score** est chargé d'évaluer la qualité de l'application de la politique. Cette évaluation utilise les scores d'application des capacités, des propriétés et de la politique, définis au chapitre 3. Il est ainsi possible de comparer l'application d'une même politique sur des systèmes différents. Le module de score est également utilisé pour sélectionner la projection la plus efficace pour chacune des propriétés, en se basant sur les scores des mécanismes et les scores d'inclusion des propriétés et des capacités (voir section 3.4). Enfin, ce module permet de calculer le taux de couverture de la politique et donc d'évaluer la qualité de son application (section 4.6).

Le **gestionnaire de politique** est également chargé de vérifier et de maintenir la cohérence de la configuration interne. Ainsi, lors de son initialisation, le gestionnaire vérifie que la règle 3.4.8 est respectée : pour chaque paire de mécanismes ayant un lien de dépendance, le gestionnaire de politique vérifie que ces mécanismes sont compatibles. De plus, pour chaque mécanisme disponible, le gestionnaire vérifie que s'il a des dépendances, celles-ci sont également disponibles. Dans le cas contraire, le mécanisme auquel il manque des dépendances est désactivé. Cette cohérence doit être maintenue tout au long du cycle de vie du SE^E. Ainsi, si un mécanisme est désactivé, tous les mécanismes qui en dépendent seront également désactivés.

En conclusion, le gestionnaire de politique est responsable de la base de connaissances du SE^E, c'est-à-dire de sa configuration interne et de la politique de sécurité à appliquer. Le gestionnaire interagit avec les autres composants du SE^E afin d'appliquer la politique de la meilleure façon possible, grâce à son module de scores. Tout d'abord, le gestionnaire fournit de l'information au moteur de projection (la politique à appliquer, les définitions des propriétés, etc.) afin de déterminer le processus d'application pour chacune des propriétés. Dans un second temps, le gestionnaire modifie sa base de connaissances en fonction des résultats obtenus par le moteur d'assurance. Par exemple, si le moteur d'assurance détecte qu'un mécanisme est devenu indisponible, le gestionnaire de politique prend en compte cette information et désactive le mécanisme correspondant dans sa base de connaissances.

4.4 L'application

L'application de la politique de sécurité consiste à interpréter les propriétés afin qu'elles soient appliquées par les mécanismes disponibles sur le nœud, ce qui permet ainsi de répondre aux besoins de sécurité.

4.4.1 Moteur d'application

Le moteur d'application est l'élément central du SE^E. Pour chaque propriété de la politique, il sélectionne les mécanismes qui appliqueront les capacités composant la propriété considérée. Le moteur d'application est donc chargé de compiler le langage et de projeter les propriétés de sécurité. La sélection de la méthode d'application est faite en plusieurs étapes. Le processus complet d'application est décrit par le graphe de flot de contrôle de la figure 4.6 et sera détaillé dans les sections suivantes.

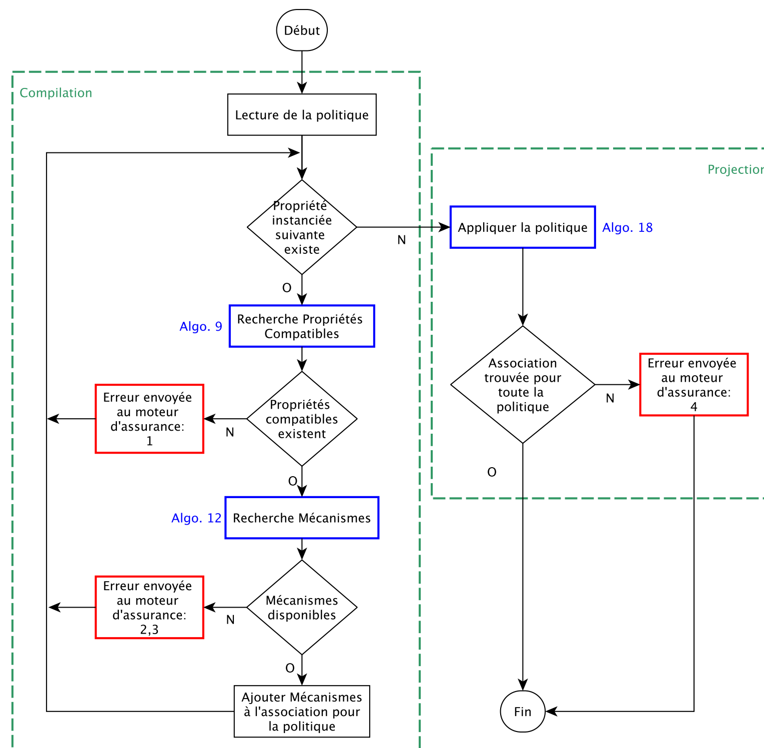

FIGURE 4.6 – Algorithme de projection (Algo. 8)

Le processus d'application de la politique se découpe en deux étapes majeures :

1. L'étape de **compilation** correspond à l'interprétation de la politique de sécurité et à la recherche d'une solution d'application (sections 4.4.2 et 4.4.3) ;
2. L'étape de **projection** est l'application de la solution trouvée lors de la phase de compilation (section 4.4.4).

Le moteur d'application commence par effectuer l'étape de compilation de la politique. Il sélectionne donc les propriétés compatibles avec la propriété instanciée. Puis, pour chaque appel à une capacité instanciée, le moteur d'application choisit une association entre une capacité et un mécanisme qui satisfait l'appel (les types de contextes doivent être compatibles). De plus, le choix des mécanismes dépend des différentes contraintes qui ont été placées sur ceux-ci (leur compatibilité et le statut récessif ou dominant des mécanismes).

Ce processus est effectué pour les deux parties des propriétés : celle définissant l'application de la propriété et celle définissant l'assurance. Les algorithmes des sections 4.4.2 et 4.4.3 permettent d'obtenir une association entre les capacités de chaque propriété et les mécanismes qui les appliqueront. Il est alors possible d'appliquer les propriétés composant la politique.

L'étape de projection de la politique se divise en trois sous-étapes : la projection elle-même, l'assurance des propriétés et l'assurance des mécanismes. La projection d'une propriété consiste en l'envoi par le moteur d'application des capacités du bloc d'application aux modules d'extension responsables des mécanismes de sécurité choisis. L'assurance d'une propriété est faite de manière similaire, mais il s'agit de l'envoi des capacités du bloc d'assurance. Enfin, l'assurance des mécanismes est réalisée par les modules appliquant les capacités : il ne s'agit pas de l'assurance liée aux propriétés mais liée aux mécanismes, permettant ainsi de vérifier le statut de chacun des mécanismes.

L'algorithme 8 décrit le processus global d'application de la politique, c'est-à-dire les étapes de compilation et de projection. L'algorithme itère sur les propriétés instanciées de la politique (lignes 4 à 13). Pour chaque propriété instanciée, l'algorithme sélectionne et ordonne les définitions de propriétés compatibles (ligne 5). Si aucune propriété compatible n'est trouvée, l'erreur 1 est envoyée au moteur d'assurance (ligne 7). Puis, la fonction recherche quels mécanismes peuvent être utilisés pour appliquer la propriété (ligne 9). Si aucun mécanisme n'est trouvé, l'erreur 2 ou 3 est envoyée au moteur d'assurance (ligne 11). La solution trouvée est alors projetée (ligne 14). Si certaines propriétés ne peuvent pas être appliquées, l'erreur 4 est envoyée au moteur d'assurance (ligne 16). Les différents types d'erreurs et leur gestion seront détaillés à la section 4.5.

Algorithme 8 Projection de la politique

```
 1: function ProjeterPolitique (Politique)
 2:     P ← Liste des propriétés
 3:     solution ← ∅
 4:     for all instance de propriété p_i dans Politique do
 5:         listePropriétés ← recherchePropriétéCompatibles(p_i)          ▷ Section 4.4.2
 6:         if listePropriétés = ∅ then
 7:             Erreur(1)
 8:         end if
 9:         solution[p_i] ← rechercheMécanismes(p_i, listePropriétés)     ▷ Section 4.4.3
10:         if solution[p_i] = ∅ then
11:             Erreur(2,3)
12:         end if
13:     end for
14:     appliquerPolitique(Politique, solution)                           ▷ Section 4.4.4
15:     if solution n'est pas complète then
16:         Erreur(4)
17:     end if
18: end function
```

Afin de sélectionner le ou les mécanismes qui entreront en jeu lors de l'application d'une propriété instanciée, trois modes de sélection sont considérés dans les fonctions utilisées par cet algorithme : le cas utilisé est fixé par l'administrateur de la politique. Les algorithmes peuvent être en mode *Best Effort*, *Défaut* ou *Maximum*.

- *Best Effort* : les mécanismes sont choisis sans prendre en compte leurs scores respectifs. Le premier mécanisme rencontré est prioritaire.
- *Défaut* : les mécanismes sont ordonnés selon leur score. Le mécanisme choisi est celui ayant le score le plus élevé et satisfaisant les contraintes (compatibilité et récessivité).
- *Maximum* : tous les mécanismes pouvant être sélectionnés le sont (en prenant en compte leur compatibilité).

D'autres modes de sélection peuvent être définis par l'expert en sécurité souhaitant configurer la manière dont les mécanismes sont choisis pour appliquer les propriétés.

4.4.2 Sélection des propriétés

Comme décrit précédemment, la première étape pour appliquer une propriété instanciée est la recherche des propriétés compatibles avec le prototype de cette instance. Le processus global de recherche est décrit à la figure 4.7 et comporte deux étapes principales : les propriétés compatibles avec le prototype de la propriété instanciée sont sélectionnées, puis elles sont triées selon leur score d'ordonnancement.

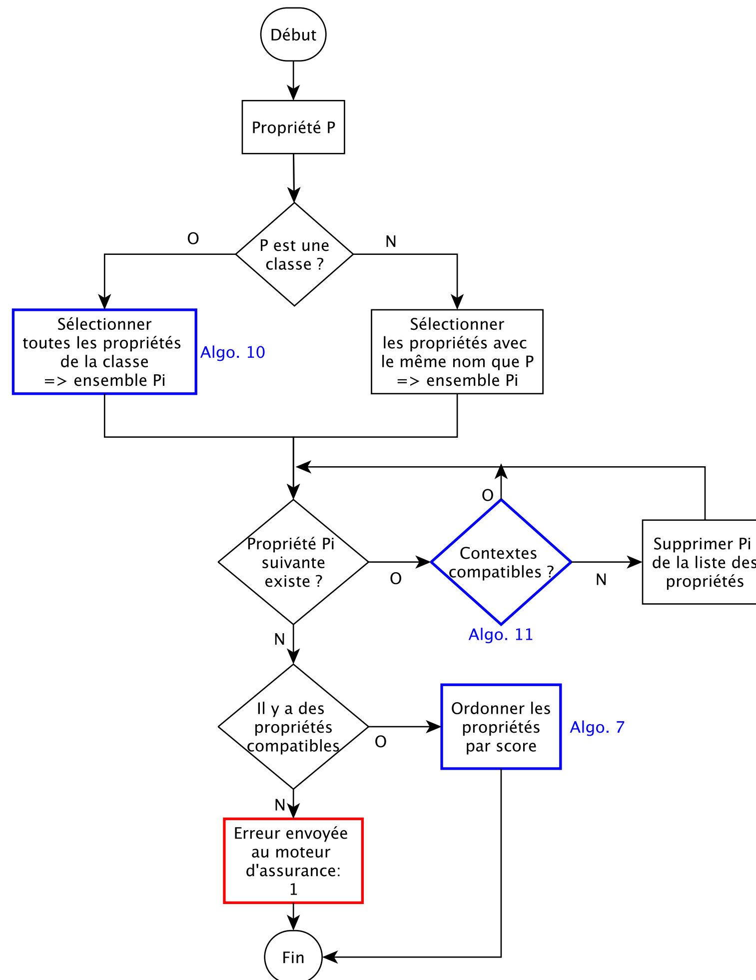

FIGURE 4.7 – Algorithme de recherche des propriétés compatibles avec P (algo. 9)

4.4.2.1 Algorithme global

L'algorithme 9, qui correspond à la figure 4.7, est utilisé pour sélectionner et ordonner l'ensemble des propriétés compatibles avec une propriété ou une classe instanciée P. Deux cas peuvent se présenter :

- Si l'instance a un nom de classe de propriétés, toutes les propriétés de cette classe sont considérées (ligne 4). Ce cas utilise la fonction de l'algorithme 10 ;
- Si l'instance a un nom de propriété, toutes les propriétés ayant le même nom sont considérées (ligne 6).

Dans les deux cas, les propriétés dont les arguments ne correspondent pas à ceux de l'instance sont ignorées (lignes 8 à 12). Pour déterminer si les arguments sont compatibles,

la fonction ArgumentsPropriétéCompatibles (algorithme 11) est utilisée.

Si aucune propriété compatible n'est trouvée, une erreur est remontée au moteur d'assurance (ligne 14). Dans le cas contraire, et si le mode de sélection choisi est `Défaut` ou `Maximum` (ligne 17), les propriétés sont ordonnées selon leur score d'ordonnancement, calculé par la fonction scoreOrdonnancementPropriété (ligne 18) définie à l'algorithme 7 (section 3.5.4). Cette fonction est donc utilisée pour classer les différentes propriétés compatibles. Finalement, la fonction renvoie une liste de propriétés (ligne 20), ordonnées ou non selon le mode de sélection choisi.

Algorithme 9 Recherche et ordonne les définitions de propriétés compatibles avec P

1: **function** recherchePropriétéCompatibles (P)
2: propriétésCompatibles ← ∅
3: **if** P est une classe de propriétés **then**
4: propriétésCompatibles ← propriétésDansClasse(C) ▷ Algo. 10
5: **else**
6: propriétésCompatibles ← propriétés nommées $P.nom$
7: **end if**
8: **for all** p_i dans propriétésCompatibles **do**
9: **if** `Not` ArgumentsPropriétéCompatibles($p_i.args$, $P.args$) **then** ▷ Algo. 11
10: Supprimer p_i de propriétésCompatibles
11: **end if**
12: **end for**
13: **if** Aucune propriété compatible n'a été trouvée **then**
14: Erreur(1)
15: **return** ∅
16: **end if**
17: **if** mode $\neq$ `BestEffort` **then**
18: Ordonner les p_i par scoreOrdonnancementPropriété(P, p_i) ▷ Déf. 3.5.5
19: **end if**
20: **return** propriétésCompatibles
21: **end function**

4.4.2.2 Gestion des classes

Dans le cas où P est une classe de propriétés, l'algorithme 9 appelle l'algorithme 10 pour rechercher récursivement les propriétés faisant partie de cette classe. Ces propriétés sont également ordonnées selon le score qui leur est assigné lors de leur inclusion dans la classe. Dans le cas d'une sous-inclusion, c'est-à-dire si une classe `C2` est incluse dans une classe `C1`, les propriétés de `C2` sont incluses par récursivité dans `C1`.

L'algorithme 10 itère ainsi sur la liste des propriétés et des classes contenues dans la classe C passée en argument (lignes 3 à 12). Si l'élément considéré est une propriété (lignes 4-5), elle est ajoutée à la liste des propriétés (ligne 5). S'il s'agit d'une classe (lignes 6 à 11), ses propriétés sont récupérées récursivement (ligne 7). Les propriétés de la sous-classe sont alors ajoutées à la liste des propriétés de C (lignes 8 à 10). La liste des propriétés incluses dans C est alors retournée par la fonction (ligne 13).

4.4.2.3 Gestion de la compatibilité

L'algorithme 11 est appelé pour exclure les propriétés dont les arguments ne sont pas inclus dans les arguments de la propriété instanciée à appliquer. Pour cela, la fonction commence par vérifier que les nombres d'arguments sont identiques (lignes 2 à 4) : si ce n'est pas le cas, la propriété considérée ne pourra pas être utilisée. Si les nombres d'arguments sont identiques, les arguments sont comparés au sens de la compatibilité des

Algorithme 10 Recherche les propriétés de la classe C

```
 1: function propriétésDansClasse (C)
 2:     listePropriétés ← ∅
 3:     for all p_i de C do
 4:         if p_i est une propriété then
 5:             Insère p_i dans listePropriétés
 6:         else
 7:             listePropSubClass ← propriétésDansClasse(p_i)
 8:             for all p_j dans listePropSubClass do
 9:                 Insère p_j dans listePropriétés
10:             end for
11:         end if
12:     end for
13:     return listePropriétés
14: end function
```

contextes (voir section 3.3.2, p. 36). Pour chacun des arguments de la propriété instanciée (lignes 5 à 10), la fonction vérifie que les types de contexte de l'argument et ceux de la propriété sont compatibles (ligne 7). Si cette compatibilité n'est pas vérifiée, la propriété considérée ne pourra pas être utilisée (ligne 8). Sinon, la propriété est compatible (ligne 11).

Algorithme 11 Vérification de l'inclusion des arguments de P dans P_{inst}

```
 1: function ArgumentsPropriétéCompatibles (P, P_inst)
 2:     if P.nbArgs ≠ P_inst.args then
 3:         return Faux
 4:     end if
 5:     for all ctx[i] dans P_inst.listeContextes do
 6:         argType ← P.argument[i]
 7:         if not compatibilitéDeContexte(argType, ctx[i]) then        ▷ Section 3.3.2
 8:             return Faux
 9:         end if
10:     end for
11:     return Vrai
12: end function
```

Suite à l'utilisation des algorithmes décrits dans cette section, on obtient une liste de propriétés, ordonnées ou non en fonction du mode de sélection choisi, qui peuvent être utilisées pour appliquer une propriété instanciée. Cette liste peut alors être utilisée dans la suite du processus d'application.

4.4.3 Sélection des mécanismes

Lorsque la liste des propriétés pouvant appliquer une propriété instanciée à été établie, la deuxième étape de la phase de compilation peut être effectuée. Cette étape a pour objectif de rechercher quelles capacités et quels mécanismes peuvent être utilisés pour appliquer la propriété. On recherche donc une association entre les capacités et les mécanismes qui puisse être utilisée pour appliquer l'une des propriétés de la liste et qui soit compatible avec les contraintes placées sur les mécanismes. Si une telle association est trouvée, la propriété peut être appliquée. Dans le cas contraire, la propriété ne peut pas être appliquée, ce qui engendre une erreur. Ce processus est décrit à la figure 4.8.

La recherche de solution pour l'application d'une propriété se décompose en trois étapes :

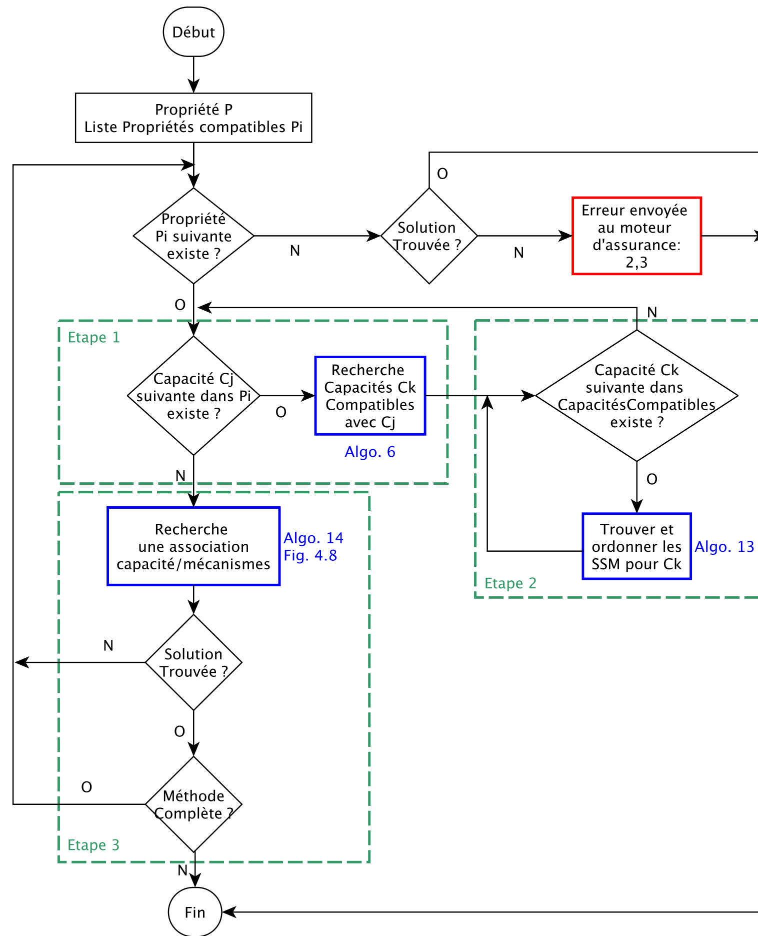

FIGURE 4.8 – Algorithme de recherche des SSM compatibles avec P (algo. 12)

1. La recherche des capacités compatibles avec les capacités instanciées de la propriété instanciée à appliquer ;
2. La recherche de mécanismes fournissant les capacités sélectionnées à l'étape 1 ;
3. La recherche d'une solution pour l'ensemble des capacités instanciées de la propriété.

Ces trois étapes correspondent à la phase de compilation d'une propriété instanciée pour un ensemble de propriétés compatibles sélectionnées.

4.4.3.1 Algorithme global

L'algorithme 12 décrit le fonctionnement global de la sélection des capacités et des mécanismes associés. Dans la suite de ce chapitre, on définit une *solution* de la manière suivante :

Définition 4.4.1: Solution
Soit P une propriété instanciée. Soit p_i une propriété compatible avec P, composée des capacités instanciées $c_{j,1 \leq j \leq n}$.
On appelle solution pour p_i une association entre des capacités $c_{j,k}$ (capacités compatibles avec les c_j) et des mécanismes $M_{j,k}$ (fournissant $c_{j,k}$), qui respecte l'ensemble des contraintes définies (compatibilité, dominance, etc.).

Cette solution est construite récursivement et un critère d'arrêt de la récursion est nécessaire. On définit donc la notion de *solution complète* :

Définition 4.4.2: Solution Complète
Une solution s pour une propriété p_i est dite complète si toutes les capacités c_j de p_i font partie de cette solution, c'est-à-dire si pour tous les c_j, s contient un couple $(c_{j,k}, M_{j,k})$.

Ainsi, pour une propriété donnée, l'algorithme 12 sélectionne et ordonne les capacités qui peuvent être utilisées (ligne 6). Puis, l'algorithme établit les différentes contraintes (compatibilité et récessivité) à respecter par les mécanismes pour une définition de capacité donnée (lignes 7 à 9). Enfin, l'algorithme recherche une solution (ligne 12). Si une telle solution est trouvée, deux cas peuvent se produire : si le mode de sélection choisi est `Maximum` (ligne 14), alors la solution trouvée est ajoutée à un tableau de solutions (ligne 15) qui seront toutes appliquées, et les autres propriétés sont évaluées. Si le mode choisi est `Défaut` ou `BestEffort` (lignes 16 à 18), la première solution trouvée est retournée à l'algorithme global et sera appliquée (ligne 17). La fin de l'algorithme (ligne 24) est atteinte dans deux cas : soit une solution a été trouvée dans le cas du mode `Maximum` et elle est retournée, soit aucune solution n'a été trouvée (ligne 22) et une solution vide est retournée après l'envoi d'une erreur au moteur d'assurance.

Algorithme 12 Association de mécanismes aux capacités d'une propriété

```
1: function rechercheMécanismes (P, listePropriétés)
2:     solutionTable ← ∅
3:     for all p_i dans listePropriétés do
4:         domaines_i ← ∅
5:         for all capacité c_j de p_i do
6:             capacitésCompatibles[c_j] ← chercheCapacitésCompatibles(c_j, P.args)  ▷ Section 3.4.2
7:             for all capacité c_j,k de capacitésCompatibles[c_j] do
8:                 mécanismes[c_j,k] ← TrouverEtOrdonnerSSM(c_j,k)                  ▷ Algo. 13
9:                 domaines_i[c_j][k] ← (c_j,k, mécanismes[c_j,k])
10:            end for
11:        end for
12:        solution ← rechercheRécursiveDeSolution(∅, domaines_i)                    ▷ Algo. 14
13:        if solution est complète then
14:            if mode = Maximum then
15:                Ajouter (p_i, solution) à solutionTable
16:            else
17:                return solution
18:            end if
19:        end if
20:    end for
21:    if solutionTable n'est pas complète then
22:        Erreur(2,3)
23:    end if
24:    return solutionTable
25: end function
```

L'algorithme 12 utilise plusieurs fonctions pour prendre en compte les différentes contraintes à respecter. Ces fonctions correspondent aux trois étapes de la sélection des mécanismes pour appliquer une propriété :

1. chercheCapacitésCompatibles (algo. 6) : sélectionne les capacités dont les arguments sont compatibles avec ceux de la propriété instanciée à appliquer. Cet algorithme a été donné à la section 3.4.2 ;
2. TrouverEtOrdonnerSSM (algo. 13) : sélectionne les mécanismes pouvant appliquer une capacité et les ordonne selon leur score de protection ;

3. RECHERCHERÉCURSIVEDESOLUTION (algo. 14) : cherche récursivement une association capacité-mécanismes, en prenant en compte la dominance éventuelle des mécanismes.

4.4.3.2 Étape 1 : Association entre capacité instanciée et capacités

La première étape a pour but de sélectionner l'ensemble des capacités compatibles avec une capacité instanciée.

La fonction CHERCHECAPACITÉSCOMPATIBLES (algo. 6) est utilisée pour effectuer cette sélection et son utilisation a été détaillée à la section 3.4.2 (p. 44).

4.4.3.3 Étape 2 : Association entre capacité et mécanismes

La seconde étape consiste en la recherche des mécanismes disponibles pouvant appliquer une capacité.

Afin de prendre en compte la contrainte du score d'ordonnancement des mécanismes pour chaque capacité, la fonction TROUVERETORDONNERSSM (algorithme 13) est utilisée. Cette fonction doit déterminer quels mécanismes sont capables d'appliquer une capacité c passée en argument. Cela est fait en se basant sur la matrice des scores d'ordonnancement (ligne 2) définie à la section 3.4.3.4 (p. 50). Si le score d'ordonnancement d'un mécanisme M_i pour c n'est pas nul, alors M_i est capable d'appliquer c (ligne 5) et il est ajouté à la liste des mécanismes à considérer (lignes 6 à 10).

L'ajout de M_i à la liste dépend du mode de sélection : si un mode `Best Effort` a été choisi, les scores d'ordonnancement ne sont pas pris en compte et les mécanismes sont ajoutés dans l'ordre où ils ont été déclarés (ligne 7). Pour les deux autres modes (`Défaut` ou `Maximum`), les mécanismes sont triés selon leur score d'ordonnancement (ligne 9).

Algorithme 13 Ordonner les mécanismes selon leur score

```
 1: function TROUVERETORDONNERSSM (c)
 2:     S ← la matrice des scores des mécanismes par capacité        ▷ Déf. 3.4.10
 3:     listeSSM ← ∅
 4:     for all M_i dans l'ensemble des mécanismes do
 5:         if S[M_i, c] ≠ 0 then                                    ▷ M_i peut appliquer c
 6:             if mode = BestEffort then
 7:                 listeSSM ← M_i
 8:             else
 9:                 listeSSM ← M_i classé selon le score S[M_i, c]
10:             end if
11:         end if
12:     end for
13:     return listeSSM
14: end function
```

L'algorithme 13 retourne alors une liste, ordonnée ou non selon le mode de sélection choisi, de mécanismes capables d'appliquer la capacité c.

4.4.3.4 Étape 3 : Recherche de solution

La troisième étape est la recherche d'une solution. Cette recherche est faite à partir des éléments, appelés *domaines*, obtenus à la suite de l'exécution des deux étapes précédentes et définis de la façon suivante :

Définition 4.4.3: Domaine
On appelle domaine une paire associant une capacité à une liste de mécanismes.

Soit c une capacité instanciée. Soient c_j une capacité compatible avec c et $listeSSM[c_j]$ la liste des mécanismes pouvant appliquer c_j. Alors la paire $(c_j, listeSSM[c_j])$ est un domaine de c.

Les algorithmes précédents permettent donc d'obtenir un ensemble de domaines vérifiant les contraintes pour chacune des capacités de la propriété considérée.

L'algorithme de recherche de mécanismes (algo. 12) génère les domaines à partir des résultats des appels à la fonction TrouverEtOrdonnerSSM. Il utilise ces domaines pour trouver une association capacité-mécanismes permettant d'appliquer une propriété. On note $domaines[c]$ l'ensemble des domaines pouvant être utilisés pour appliquer c.

Algorithme global La figure 4.9 décrit le processus de recherche récursive d'une solution.

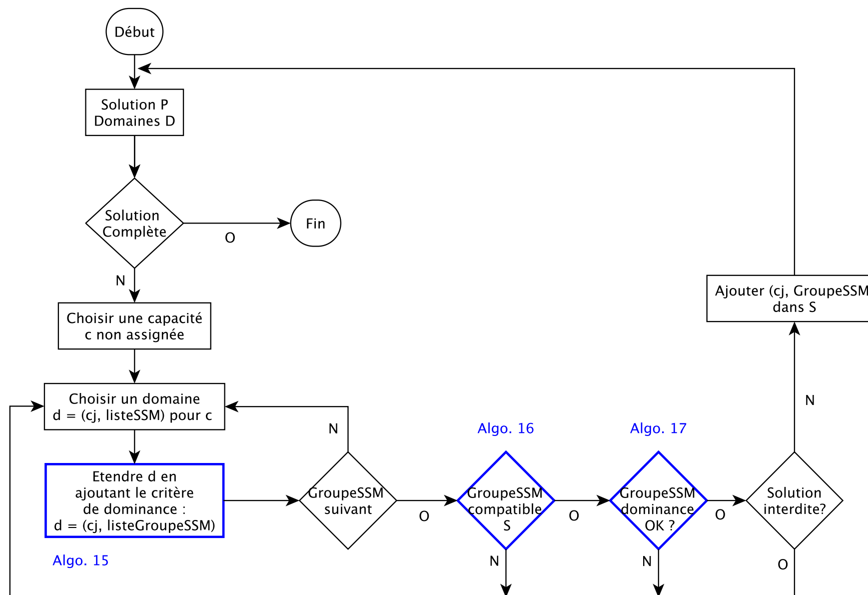

Figure 4.9 – Algorithme récursif de sélection de mécanismes (algo. 14)

La récursivité s'arrête lorsqu'une une solution complète a été trouvée. Tant que la solution n'est pas complète, les domaines qui ont été créés sont testés afin de compléter la solution. Celle-ci doit respecter plusieurs contraintes : si ce n'est pas le cas, le domaine qui a été ajouté est supprimé et le suivant est testé.

L'une des contraintes à respecter est la contrainte de dominance qui étend les domaines possibles : si des mécanismes dominants font déjà partie de la solution envisagée, alors ces mécanismes dominants doivent être ajoutés à chacun des domaines possibles.

Considérons l'exemple de la propriété **`Confidentiality_Access_Control`**, composée des deux capacités `c1 :=` **`deny_all_read_accesses`** et `c2 :=` **`allow_read_access`** (section 3.5). Considérons les deux mécanismes dominants SELinux et DAC, qui peuvent tous deux appliquer `c1` et `c2`. On a alors les domaines suivants : $d_1 = (c1, \{SELinux, DAC\})$ et $d_2 = (c2, \{SELinux, DAC\})$. Considérons que SELinux est choisi pour appliquer l'instance de `c1`. SELinux étant dominant, il devra obligatoirement être choisi pour appliquer l'instance de `c2`. On définit donc un *domaine étendu* d_2' en ajoutant SELinux à tous les choix possibles du domaine d_2. On a alors : $d_2' = (c2, \{(SELinux), (DAC, SELinux)\})$ et SELinux sera nécessairement utilisé pour appliquer `c2`.

On définit donc les domaines étendus de la manière suivante :

Définition 4.4.4: Domaine étendu
On appelle domaine étendu une paire associant une capacité à une liste de groupes de mécanismes (un groupe peut contenir un unique mécanisme).
Soit c une capacité instanciée. Soit $d = (c_j, listeSSM[c_j])$ *un domaine de c. Soient* $M_{k,1 \leq k \leq n}$*, n mécanismes dominants ayant été choisis pour appliquer d'autres capacités de la propriété. Alors, pour chaque mécanisme* M_i *de* $listeSSM[c_j]$*, on crée le groupe* $(M_i, M_{k,1 \leq k \leq n})$*. On note alors* $listeGroupeSSM[c_j]$ *la liste de ces groupes.*
Le domaine étendu d_e *de d est alors défini par :* $d_e = (c_j, listeGroupeSSM[c_j])$

L'algorithme 14 correspond à la figure 4.9 et détaille le processus de recherche de solution. Il utilise les domaines pour rechercher récursivement une association entre les capacités de la propriété et les mécanismes de sécurité disponibles. Cet algorithme génère les domaines étendus qui peuvent être choisis pour appliquer une capacité, en prenant en compte le statut de dominance des mécanismes déjà choisis (fonction GÉNÉRERDOMAINESETENDUS, ligne 7). Puis, l'algorithme recherche un domaine compatible avec la solution envisagée, en vérifiant leur compatibilité (fonction VÉRIFIERCOMPATIBILITÉ, ligne 9) et leur respect du critère de dominance (fonction VÉRIFIERDOMINANCE, ligne 9). Il vérifie également que la solution n'a pas été interdite par le processus d'assurance (ligne 9, voir section 4.5.5).

Algorithme 14 Sélection des mécanismes

```
 1: function rechercheRécursiveDeSolution (solution, domaines)
 2:     if solution est complète then
 3:         return solution
 4:     end if
 5:     c ← une capacité non encore assignée
 6:     for all (c_j, listeSSM[c_j]) dans domaines[c] do
 7:         listeGroupeSSM ← GénérerDomainesEtendus(listeSSM[c_j], solution)     ▷ Algo. 15
 8:         for all groupeSSM[k] dans listeGroupeSSM do
 9:             if VérifierCompatibilité(groupeSSM[k], solution) et VérifierDominance(groupeSSM[k], solution) et solution ∉ ListeSolutionsInterdites then     ▷ Algos. 16 et 17
10:                 Ajouter (c_j, groupeSSM[k]) dans solution
11:                 résultat ← rechercheRécursiveDeSolution(solution, domaines)
12:                 if résultat ≠ ∅ then
13:                     return résultat
14:                 end if
15:                 Supprimer (c_j, groupeSSM[k]) de solution
16:             end if
17:         end for
18:     end for
19:     return ∅
20: end function
```

Si une association complète entre les capacités et les mécanismes a été trouvée, l'algorithme 14 retourne cette solution (lignes 2 à 4). Dans le cas contraire, une capacité instanciée c non encore assignée est choisie (ligne 5). Comme indiqué précédemment, les domaines de c sont des paires composées d'une capacité c_j et de la liste des mécanismes fournissant c_j. L'algorithme itère sur ces domaines (lignes 6 à 18), et pour chacun d'entre eux, génère les domaines étendus (ligne 7) en utilisant la fonction GÉNÉRERDOMAINESETENDUS. On obtient alors une liste de groupes de mécanismes. Si un groupe de mécanismes est compatible avec la solution envisagée, il y est ajouté (ligne 10) et l'appel récursif à la

fonction est effectué (ligne 11). Si un résultat est trouvé, il est retourné par la fonction (lignes 12 à 14). Sinon, le groupe de mécanismes est supprimé de la solution (ligne 15) et un autre groupe est testé.

Génération des domaines étendus La génération des domaines étendus est faite par la fonction GÉNÉRERDOMAINESÉTENDUS de l'algorithme 15.

Algorithme 15 Générer les domaines étendus de mécanismes

```
 1: function GÉNÉRERDOMAINESÉTENDUS (c, listeSSM, solution)
 2:     R ← la matrice de récessivité                                    ▷ Déf. 3.4.9
 3:     S ← la matrice des score de priorité                             ▷ Déf. 3.4.10
 4:     Comp ← la matrice de compatibilité                               ▷ Déf. 3.4.6
 5:     listeGroupesSSM ← ∅
 6:     ssmDominants ← les M_i tels que M_i ∈ solution ET R(M_i) = 1
 7:     ssmDominantsPrésents ← les M_i tels que M_i ∈ [ ssmDominants ∩ listeSSM ]
 8:     if mode = BestEffort OU mode = Défaut then
 9:         if ssmDominantsPrésents = ∅ then
10:             return listeSSM
11:         else
12:             listeGroupesSSM[0] ← ssmDominantsPrésents
13:         end if
14:     else if mode = Maximum then
15:         G ← le graphe de SSMs tel que :
16:                 - chaque nœud est un mécanisme M_i ∈ listeSSM
17:                 - le poids d'un nœud est le score S(M_i) du mécanisme
18:                 - M_i et M_j sont adjacents si Comp(M_i, M_j) = 1
19:         if ssmDominantsPrésents ≠ ∅ then
20:             G ← réduire G au voisinage des M_i ∈ ssmDominantsPrésents
21:         end if
22:         listeGroupesSSM ← les cliques de G ordonnées par leur poids décroissant
23:     end if
24:     return listeGroupesSSM
25: end function
```

La fonction GÉNÉRERDOMAINESÉTENDUS retourne une liste de groupes de mécanismes en prenant en compte le mode de sélection et la présence ou l'absence de mécanismes dominants dans la solution partielle considérée. Quatre cas sont possibles :

1. si le mode est `Best Effort` ou `Défaut` et s'il n'y a pas de mécanisme dominant présent, la liste de mécanismes est renvoyée sans modification (ligne 10) ;
2. si le mode est `Best Effort` ou `Défaut` et s'il y a des mécanismes dominants présents, un seul groupe de mécanisme est renvoyé et il est formé des mécanismes dominants (ligne 12) ;
3. si le mode est `Maximum` et s'il n'y a pas de mécanisme dominant présent, on génère le graphe des mécanismes et chaque groupe correspond à une clique de ce graphe (lignes 15-18 et 22) ;
4. si le mode est `Maximum` et s'il y a des mécanismes dominants présents, le même graphe est généré, mais on ne considère que les mécanismes compatibles avec les mécanismes dominants (lignes 19 à 21).

Le cas du mode `Maximum` est géré séparément, puisqu'il faut alors choisir un ensemble de mécanismes et non pas un seul mécanisme (ou les mécanismes dominants) comme pour les deux autres modes.

La liste de groupes renvoyée par la fonction GÉNÉRERDOMAINESÉTENDUS est donc une liste, ordonnée ou non selon le mode de sélection, qui prend en compte le critère de

dominance des mécanismes. Chacun des groupes de mécanismes dans cette liste est ensuite testé pour vérifier s'il peut ou non être utilisé pour appliquer la capacité. Pour cela, la compatibilité des mécanismes est vérifiée avant d'effectuer la récursion.

Gestion du mode de sélection Maximum Dans les cas où le mode de sélection est `Maximum` (cas 3 et 4), un graphe G est généré. Ce graphe représente la matrice de compatibilité (définition 3.4.6) réduite aux mécanismes pouvant intervenir dans l'application de la propriété : les sommets du graphe sont les mécanismes fournissant les capacités et deux nœuds sont adjacents si, et seulement si, leurs mécanismes sont compatibles. G est un graphe pondéré : le poids d'un nœud est égal au score d'ordonnancement de son mécanisme.

La figure 4.10 présente un exemple de graphe G pour cinq mécanismes ayant des scores entre 4 et 6. Dans le cas d'une propriété ayant une capacité de contrôle d'accès système et une de contrôle d'accès réseau, il faut sélectionner le sous-ensemble de mécanismes compatibles (c'est-à-dire adjacents) ayant le poids maximal (ici, le sous-ensemble rouge, composé des mécanismes SELinux, DAC et iptables, a un poids égal à 16).

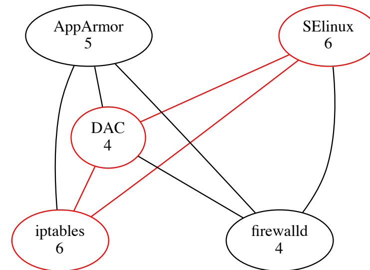

FIGURE 4.10 – Exemple de recherche d'une clique de poids maximum

Ainsi, la recherche du groupe de mécanismes ayant le score d'ordonnancement le plus élevé et étant compatibles peut être ramené à un problème de recherche de la clique de poids maximal dans un graphe. Ce problème est un problème classique de la théorie des graphes et il s'agit d'un problème NP-complet [Karp, 1972]. Cependant, dans notre cas, le nombre de nœuds de G (c'est-à-dire le nombre de mécanismes disponibles pouvant intervenir dans l'application d'une propriété) est suffisamment faible pour que la complexité de ce problème soit négligeable. Nous utilisons donc l'algorithme décrit dans [Östergård, 2001] pour obtenir les groupes de mécanismes pouvant être utilisés.

Dans le cas où des mécanismes dominants sont présents, on ne considère que le voisinage du sous-graphe composé de ces mécanismes, qui est déjà un sous-graphe complet (si ces mécanismes ont tous été sélectionnés, alors ils sont tous compatibles entre eux).

Gestion des contraintes La compatibilité des mécanismes est vérifiée par la fonction VÉRIFIERCOMPATIBILITÉ de l'algorithme 16. La fonction détermine si les mécanismes d'un groupe sont compatibles avec les mécanismes qui ont déjà été sélectionnés pour appliquer d'autres capacités. Pour cela, on utilise la matrice de compatibilité entre les mécanismes (ligne 2), définie par l'expert en sécurité. Si l'un des mécanismes du groupe est incompatible avec l'un des mécanismes déjà choisi (ligne 5), la fonction indique que le groupe de mécanismes est incompatible avec la solution envisagée (ligne 6). Sinon, la fonction retourne que les deux ensembles de mécanismes sont compatibles (ligne 10).

La dominance doit également être vérifiée. En effet, lorsque l'algorithme 15 génère les domaines étendus, il vérifie que les domaines ainsi obtenus contiennent l'ensemble des mécanismes dominants ayant été préalablement choisis pour appliquer d'autres capacités.

Algorithme 16 Vérifier les compatibilités de mécanismes

```
 1: function VérifierCompatibilité (groupeSSM, solution)
 2:     Comp ← la matrice de compatibilité                          ▷ Déf. 3.4.6
 3:     for all M_i dans groupeSSM do
 4:         for all M_j dans solution do
 5:             if Comp(M_i, M_j) = 0 then                ▷ M_i et M_j sont incompatibles
 6:                 return Faux
 7:             end if
 8:         end for
 9:     end for
10:     return Vrai
11: end function
```

Cependant, il est nécessaire de vérifier également que si un nouveau mécanisme dominant est choisi, il l'a bien été pour toutes les autres capacités qu'il fournit et qui font partie de la solution temporaire envisagée. Pour cela, on utilise la fonction VérifierDominance de l'algorithme 17.

Algorithme 17 Vérifier les dominances de mécanismes

```
 1: function VérifierDominance (groupeSSM, solution)
 2:     R ← la matrice de dominance                                  ▷ Déf. 3.4.9
 3:     S ← la matrice des score d'ordonnancement                    ▷ Déf. 3.4.10
 4:     ssmDominants ← les M_i tels que M_i ∈ solution ET R(M_i) = 1
 5:     ssmToTest ← les M_i tel que M_i ∈ groupeSSM ET M_i ∉ ssmDominants
 6:     for all M_i dans ssmToTest do
 7:         if R(M_i) = 1 then                                       ▷ M_i dominant
 8:             for all c dans solution do
 9:                 if S(M_i, c) ≠ 0 ET M_i ∉ solution[c] then
10:                     return Faux
11:                 end if
12:             end for
13:         end if
14:     end for
15:     return Vrai
16: end function
```

La fonction VérifierDominance récupère l'ensemble des mécanismes d'un groupe qui ne font pas partie de l'ensemble des mécanismes dominants déjà choisis (ligne 4). Pour chaque mécanisme M_i (lignes 5 à 13) tel que M_i est dominant (ligne 6), l'algorithme vérifie que, pour chacune des capacités c déjà appliquées (lignes 7 à 11), le critère de dominance est respecté : si M_i peut appliquer c mais n'a pas été sélectionné (ligne 8), alors le critère de dominance n'est pas respecté (M_i étant dominant, il doit être utilisé pour toutes les capacités qu'il fournit ou aucune) ce qui est donc signalé par la fonction (ligne 9).

4.4.3.5 Synthèse

A la suite de cette étape de sélection des mécanismes, on obtient une solution possible pour appliquer une propriété. La solution se compose d'associations entre des capacités et des mécanismes, et chaque association est liée à une capacité instanciée faisant partie de la propriété.

Si une solution a été trouvée, elle peut être utilisée pour appliquer la propriété, comme décrit à la section suivante. Dans le cas contraire, la propriété ne peut pas être appliquée : le traitement de cette erreur sera décrit à la section 4.5.

4.4.4 Application des propriétés

Les modules d'application sont utilisés pour appliquer les capacités en interagissant avec les mécanismes de sécurité qui les fournissent. Lorsque le moteur d'application a déterminé comment chaque propriété doit être appliquée, comme décrit dans la section précédente, il demande l'application de chacune des capacités aux modules sélectionnés. Chaque module d'extension configure alors le mécanisme qui lui est associé afin d'appliquer la capacité.

Les modules d'application sont utilisés pour appliquer les deux blocs composant une propriété : le bloc d'application et le bloc d'assurance. Les modules peuvent recevoir trois types de commandes envoyées par le moteur d'application. Tout d'abord, les modules sont utilisés pour appliquer la propriété instanciée, c'est-à-dire les capacités de son bloc d'application. En second lieu, les modules sont utilisés pour générer des tests d'assurance pour les propriétés en appliquant les capacités du bloc d'assurance. Finalement, ils génèrent les tests d'assurance liés aux mécanismes (par exemple pour vérifier qu'un mécanisme est bien fonctionnel), et ceci pour chaque mécanisme utilisé lors de l'application de la politique.

L'algorithme 18 décrit le processus de communication entre le moteur d'application et les modules lors de l'application des propriétés. Tout d'abord, les propriétés sont appliquées selon les associations capacités-modules qui ont été sélectionnées (lignes 3 à 10), c'est-à-dire que chaque capacité du bloc d'application est envoyée aux modules qui ont été sélectionnés, en utilisant la fonction AppliquerCapacité. Puis, de la même manière, l'assurance des propriétés est appliquée (lignes 11 à 18), c'est-à-dire que les capacités du bloc d'assurance sont envoyées aux modules choisis (fonction AppliquerCapacité, algo. 19). Enfin, pour chaque mécanisme qui a été utilisé, le moteur de projection signale à son module que l'application de la politique est terminé et que le module peut mettre en place l'assurance liée au mécanisme associé (lignes 19 à 24).

Algorithme 18 Demande d'exécution des décisions aux modules d'extension

```
 1: function AppliquerPolitique (Politique, solution)
 2:     PluginsActifs ← ∅
 3:     for all Propriété instanciée P dans Politique do            ▷ Application des propriétés
 4:         for all Capacité instanciée c de P, tel que c est une capacité d'application do
 5:             PluginsActifs ← AppliquerCapacité(c, P[args], solution)           ▷ Algo. 19
 6:             if un problème est rencontré lors de l'application de c then
 7:                 Erreur(4)
 8:             end if
 9:         end for
10:     end for
11:     for all Propriété instanciée P dans Politique do            ▷ Assurance des propriétés
12:         for all Capacité instanciée c de P, tel que c est une capacité d'assurance do
13:             PluginsActifs ← AppliquerCapacité(c, P[args], solution)
14:             if un problème est rencontré lors de l'application de c then
15:                 Erreur(4)
16:             end if
17:         end for
18:     end for
19:     for all Plugin p dans PluginsActifs do                      ▷ Assurance des mécanismes
20:         Envoyer "AppliquerAssurance" à p
21:         if un problème est rencontré lors de l'application de l'assurance pour p then
22:             Erreur(4)
23:         end if
24:     end for
25: end function
```

La génération de tests d'assurance pour les propriétés et pour les mécanismes permet

d'obtenir des informations sur la protection effective au cours de la vie du système. Ces éléments sont ensuite traités par le moteur d'assurance (section 4.5).

La communication effective entre le moteur d'application et les modules d'extension pour demander l'application des capacités des deux blocs d'une propriété est faite par la fonction AppliquerCapacité de l'algorithme 19.

Algorithme 19 Demande d'exécution des décisions aux modules d'extension

```
1: function AppliquerCapacité (c, arguments, solution)
2:     capacitéSélectionnée ← solution[c][capacite]
3:     listePlugins ← solution[c][plugin]
4:     for all plugin dans listePlugins do
5:         Envoyer (capacitéSélectionnée, arguments) à plugin
6:     end for
7:     return listePlugins
8: end function
```

La fonction prend en paramètres la capacité instanciée, c'est-à-dire le nom de la capacité et les contextes qui lui sont passés en arguments, et la solution qui a été établie. L'algorithme peut alors obtenir la capacité qui va être utilisée (ligne 2) et les modules d'extension qui ont été sélectionnés pour l'appliquer (ligne 3). Chacun de ces modules (lignes 4 à 6) est ensuite contacté pour demander l'application de la capacité (ligne 5, section 5.2.1). La liste des modules utilisés est retournée par la fonction afin de connaitre les mécanismes utilisés et ainsi définir l'assurance pour ces mécanismes.

4.5 L'assurance

Comme détaillé à la section 4.4, le moteur d'application configure les mécanismes afin d'appliquer les propriétés définies dans la politique.

Cependant, l'application de la politique nécessite d'être vérifiée tout au long de l'exécution du système. Pour cette raison, le moteur d'application configure des mécanismes d'assurance afin de vérifier que la politique a l'effet attendu. Le moteur d'assurance du SE^E a pour objectif de récupérer les informations d'assurance ainsi générées afin de vérifier l'état de l'application de la politique. De plus, ces informations permettent de faire de l'application de la politique un processus dynamique, qui s'adapte aux évolutions du système et aux problèmes rencontrés.

Deux principaux types d'erreurs peuvent donc être rencontrés (voir table 4.1) et correspondent aux phases d'application et d'exécution.

Phase	Étape	Erreur	Code	Algo.
Application	Compilation	Aucune propriété compatible	1	8, 9
		Aucune capacité compatible	2	8, 12
		Aucun mécanisme compatible	3	8, 12
	Projection	Problème d'application par un module	4	8, 18
Exécution		Dysfonctionnement d'un mécanisme	-	-
		Propriété non fonctionnelle	-	-

Table 4.1 – Table des erreurs

Les erreurs de la phase d'application correspondent aux erreurs générées pour les algorithmes de la section 4.4. Elles possèdent donc un code d'erreur permettant de les identifier.

Les erreurs de la phase d'exécution sont quant à elles le résultat des scripts d'assurance générés lors de l'application.

4.5.1 Moteur d'assurance

Le moteur d'assurance reçoit les informations d'assurance générées par le moteur d'application et les modules d'assurance. Les erreurs signalées au moteur d'assurance peuvent être liées à une défaillance d'un mécanisme, à une inefficacité d'une propriété (la propriété n'a pas l'effet escompté), à une erreur lors de l'application d'une capacité ou à une propriété qui n'est pas appliquée. La figure 4.11 présente les interactions entre le moteur d'assurance et les autres éléments du SE^E, c'est-à-dire le moteur d'application et les modules d'assurance.

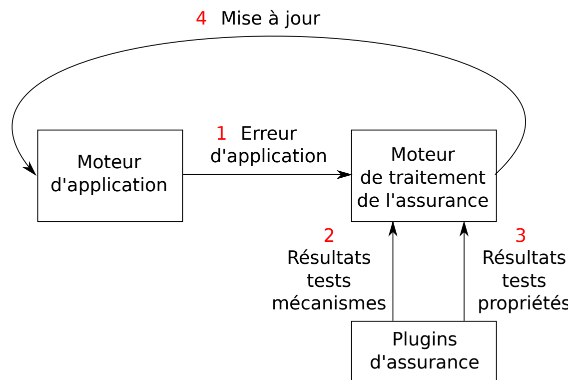

FIGURE 4.11 – Principe du moteur d'assurance

Le moteur d'assurance reçoit trois types d'entrées :

1. Les **erreurs de la phase d'application** : elles reflètent un problème lors de la compilation ou de la projection d'une propriété instanciée (section 4.5.2) ;
2. Les **erreurs de mécanismes** : elles sont le résultat des tests d'assurance pour les mécanismes, qui sont générés par le moteur d'assurance et exécutés par les modules d'assurance (section 4.5.3) ;
3. Les **erreurs de propriétés** : elles sont générées par les modules d'assurance à partir des tests générés à la phase d'application (section 4.5.4).

Le moteur d'assurance traite ces différentes informations et les utilise pour mettre à jour le gestionnaire de politique. Cela entraîne alors une mise à jour de l'application de la politique par le moteur d'application (étape 4).

Les étapes majeures du processus d'assurance sont présentées à la figure 4.12.

Selon le type d'erreur, le moteur d'assurance utilise différentes fonctions pour traiter l'information obtenue et mettre à jour la base de connaissances du SE^E. Selon les erreurs rencontrées et les traitements effectués, le moteur d'assurance peut aussi déclencher une nouvelle application d'une sous-partie de la politique.

La figure 4.12 détaille le processus effectué par le moteur d'assurance correspondant à l'algorithme 20. En cas de problème détecté par les modules d'assurance, plusieurs cas sont possibles. Si l'erreur se produit durant la phase d'application, soit une propriété ne peut pas être appliquée (problème durant l'étape de compilation, ligne 9), soit un problème a été rencontré lors de la projection d'une capacité par un mécanisme (étape de projection, ligne 7). Si l'erreur se produit durant la phase d'exécution, soit l'un des mécanismes est

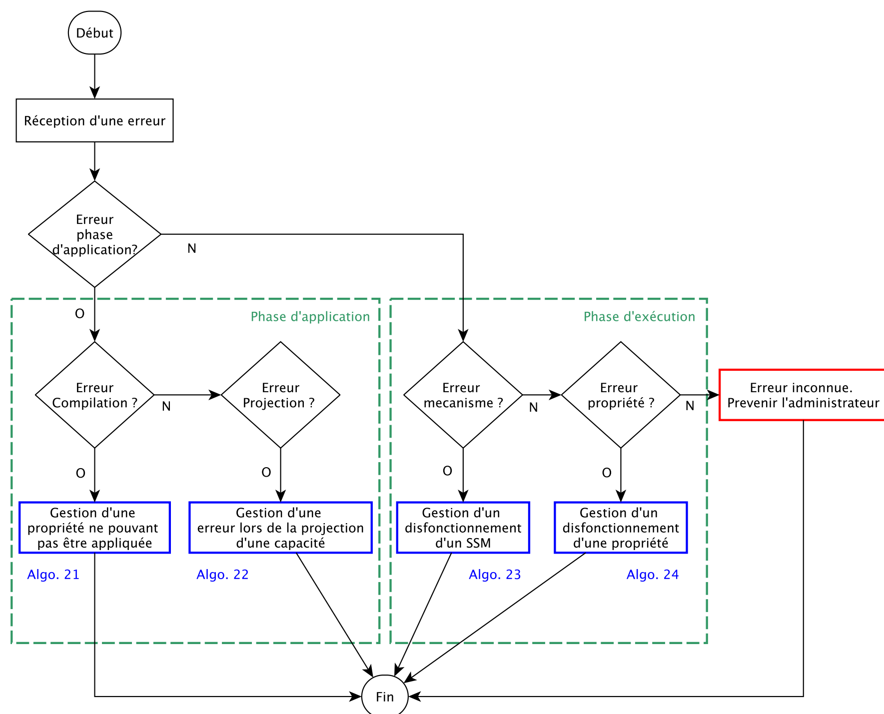

FIGURE 4.12 – Processus d'assurance (Algo. 20)

défaillant (ligne 3), soit l'une des propriétés n'a pas l'effet escompté (ligne 5). Enfin, si l'erreur rencontrée est inconnue, une alerte est remontée à l'utilisateur pour qu'une gestion plus précise soit effectuée.

Ces différentes situations sont gérées par les fonctions des algorithmes 21, 22, 23 et 24.

Algorithme 20 Processus d'assurance

```
1:  function ASSURANCE (message, Politique)
2:      if message = "impossibilité d'appliquer la propriété P" then
3:          return ERREURCOMPILATIONPROPRIÉTÉ(P)                          ▷ Algo. 21
4:      else if message = "impossibilité d'appliquer la capacité c (prop. P) SSM M" then
5:          return ERREURPROJECTION(P, c, M)                              ▷ Algo. 22
6:      else if message = "mécanisme M non fonctionnel" then
7:          return ERREURMÉCANISME(M)                                     ▷ Algo. 23
8:      else if message = "propriété P non correctement appliquée" then
9:          return ERREURPROPRIÉTÉ(P)                                     ▷ Algo. 24
10:     else
11:         Erreur inconnue : envoyer une alerte à l'administrateur
12:     end if
13:     return Faux
14: end function
```

4.5.2 Gestion des erreurs

La première catégorie d'erreurs gérées par le moteur d'assurance est celle des erreurs se produisant lors de l'application de la politique. Deux cas sont considérés, comme décrit à la figure 4.13 :

– une erreur peut s'être produite lors de la compilation de la propriété (algo. 21) ;

– une erreur peut s'être produite lors de la projection de la propriété (algo. 22).

Ces erreurs peuvent être dues à un problème de configuration du SE^E, à un problème de définition de la politique ou à une configuration inadéquate du système. Il s'agit dont le plus souvent de problèmes nécessitant une intervention externe, soit de l'administrateur de l'architecture logicielle, soit de l'expert sécurité.

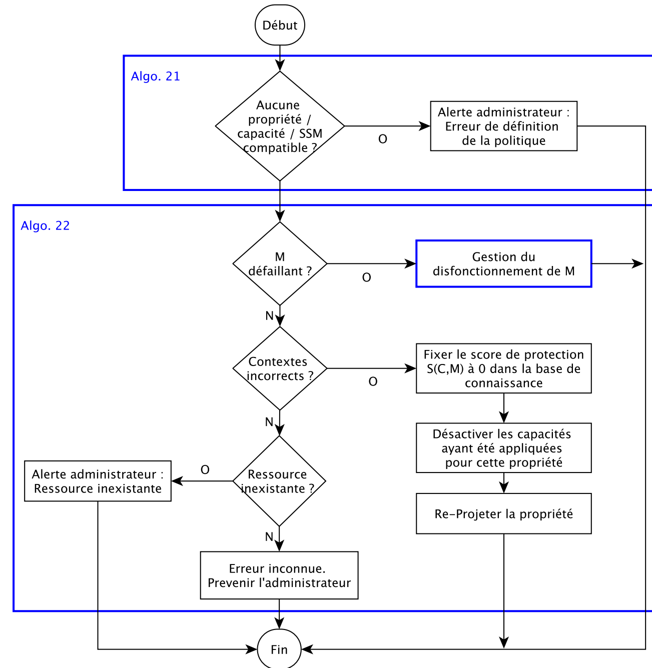

FIGURE 4.13 – Traitement d'une erreur lors de l'application de la politique

L'algorithme 21 décrit la fonction ERREURCOMPILATIONPROPRIÉTÉ qui est appelée lorsqu'aucune solution n'est trouvée pour appliquer une propriété.

Algorithme 21 Erreur lors de l'application de la propriété P

1: **function** ERREURCOMPILATIONPROPRIÉTÉ (P)
2:     **if** Aucune propriété compatible avec P **then**
3:         **return** Alerte Administrateur : propriété non définie
4:     **else if** Absence de capacité compatible **then**
5:         **return** Alerte Administrateur : capacité non définie
6:     **else if** Absence de mécanisme compatible **then**
7:         **return** Alerte Administrateur : mécanisme non disponible pour une capacité
8:     **end if**
9:     **return** Alerte Administrateur : erreur inconnue
10: **end function**

Trois situations sont gérées par l'algorithme 21 :

– Aucune propriété compatible avec P n'a été trouvée : il s'agit donc d'une erreur dans la définition de la politique (lignes 2-3) ;
– Chacune des propriétés compatibles utilise au moins une capacité indéfinie : il peut s'agir d'une erreur dans la définition des capacités ou d'une erreur dans l'appel à la capacité par la propriété (lignes 4-5) ;
– Chacune des propriétés compatibles utilise au moins une capacité pour laquelle aucun mécanisme n'est disponible : soit aucun module n'a été implémenté, soit les mécanismes ne sont pas disponibles sur le nœud considéré (lignes 6-7).

Dans ces trois cas, les erreurs sont dues à un problème de politique ou de configuration : le SE^E a donc besoin d'une décision externe, bien qu'il puisse suggérer des solutions possibles (par exemple, installer un mécanisme de sécurité fournissant une capacité ou vérifier que les propriété utilisées sont correctement appelées).

La fonction ErreurProjection (algorithme 22) est quant à elle appelée si une erreur se produit lors de la projection d'une capacité par un mécanisme.

Algorithme 22 Erreur lors de la projection de la capacité c par le mécanisme M

```
 1: function ErreurProjection (P, c, M)
 2:     if l'erreur est due à une défaillance du mécanisme M then
 3:         return ErreurMécanisme(M)                                  ▷ Algo. 23
 4:     else if c ne peut pas être appliquée sur ces arguments then
 5:         S ← la matrice des scores d'ordonnancement des mécanismes   ▷ Déf. 3.4.10
 6:         S[M, c] ← 0                                ▷ M ne peut plus appliquer c
 7:         for all c_i in P do
 8:             if c_i a été appliquée then
 9:                 Désactiver c_i
10:             end if
11:         end for
12:         return ProjeterPolitique(P)                                 ▷ Algo. 8
13:     else if Pas de ressources avec ces contextes then
14:         return Alerte Administrateur : aucune ressource trouvée
15:     end if
16:     return Alerte Administrateur : erreur inconnue
17: end function
```

Cette fonction peut être appelée dans trois cas :

- Si le module associé au mécanisme détecte que celui-ci n'est pas actif ou ne se comporte pas comme attendu (ligne 2), une défaillance de mécanisme est signalée et est traitée par la fonction ErreurMécanisme (ligne 3, algo. 23) ;
- Si le module est incapable d'appliquer la capacité sur les arguments qui lui ont été donnés (lignes 4 à 12), alors la base de connaissances est mise à jour pour indiquer que le mécanisme n'est pas capable d'appliquer la capacité : cela est fait en fixant la valeur correspondante dans la matrice des scores d'ordonnancement des mécanismes à 0 (ligne 6). Puis, si la propriété a déjà été partiellement appliquée, celle-ci est annulée (ses capacités sont désactivées, lignes 7 à 11). Enfin, la propriété est appliquée (ligne 12) : en raison de la modification de la matrice des scores, l'application sera faite différemment et le problème ne sera pas rencontré une seconde fois ;
- Si aucune ressource ne correspond aux contextes passés en arguments alors que des ressources sont attendues, l'administrateur est prévenu (lignes 13-14). Il ne s'agit pas nécessairement d'une erreur, mais cela peut être lié à un problème de définition de la politique.

Les algorithmes 21 et 22 permettent donc de traiter des erreurs rencontrées durant l'application de la politique. La plupart des erreurs peuvent être automatiquement gérées par le SE^E, limitant ainsi le nombre d'interventions humaines nécessaires. Cependant, une partie de ces problèmes sont liés à une erreur de définition de la politique ou de la configuration et nécessitent donc une intervention de l'administrateur de l'architecture logicielle ou de l'expert en sécurité.

4.5.3 Assurance des mécanismes

L'assurance des mécanismes de sécurité constitue le premier type de tests de la phase d'exécution. Il s'agit de vérifier que les mécanismes qui ont été utilisés pour appliquer les

propriétés de la politique sont tous actifs et correctement configurés. Cette section décrit la génération de ces tests et le cycle de vie des modules d'extension.

4.5.3.1 Génération

La génération des tests pour l'assurance des mécanismes est faite lors de la phase d'application. Après l'application des propriétés de sécurité, les modules d'extension actifs sont contactés afin de générer les tests d'assurance qui leur sont spécifiques (voir algorithme 18). Ces tests sont indépendants des propriétés de la politique et ont pour but de vérifier que les mécanismes utilisés ne présentent pas de dysfonctionnement et sont toujours actifs.

Ces tests étant dépendants des mécanismes, leur génération est laissée à la discrétion des modules (section 5.2), tout comme pour l'application des capacités. Ils sont ensuite exécutés par un module d'assurance si une propriété d'assurance est présente et leurs résultats peuvent être traités par le moteur d'assurance en cas de problème sur un mécanisme.

Bien que la génération des tests soit faite par le module, leur format est commun à l'ensemble des mécanismes. En effet, afin que les tests puissent être exécutés par le moteur d'assurance, ils doivent être générés sous forme de scripts. Les résultats de l'exécution des scripts ainsi obtenus seront alors traités par le moteur d'assurance.

4.5.3.2 Cycle de vie des modules d'extension

Les modules d'extension suivent tous un même cycle de vie, dont les différents états sont présentés à la figure 4.14.

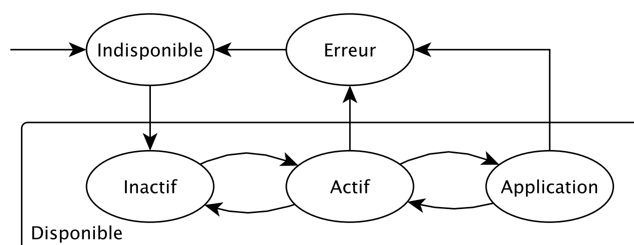

FIGURE 4.14 – Cycle de vie d'un module d'extension

Le premier état est *Indisponible.* Il s'agit de l'état par défaut d'un module et peut indiquer que le module ne peut pas être utilisé ou que son mécanisme n'est pas disponible sur le nœud considéré (le mécanisme n'est pas installé, il a rencontré une erreur, etc.).

Lors de l'initialisation du SE^E, certains modules d'extension passent de l'état *Indisponible* à l'état *Inactif* et deviennent donc *Disponibles.* Dans l'implémentation actuelle du SE^E, les modules disponibles sont spécifiés par un fichier défini par l'expert en sécurité. Un module est dit disponible quand son mécanisme associé est présent sur le nœud et peut appliquer des capacités : il s'agit d'un sur-état, englobant trois états (*Inactif*, *Actif* et *Application*).

Lorsqu'un module applique une capacité, il passe à l'état *Actif* indiquant qu'il a été utilisé par le SE^E pour au moins une propriété. Si, lors d'un changement de politique, l'ensemble des capacités appliquées par ce module sont supprimées, alors le module repasse à l'état *Inactif.*

A la réception de la demande d'application d'une capacité, le module *actif* passe temporairement à l'état *Application* : le retour à l'état *Actif* se fait automatiquement lorsque

l'application de la capacité est terminée. Si la demande d'application est faite sur un module *inactif*, alors le module passe successivement dans les états *Actif* et *Application*, avant de reprendre l'état *Actif* en attendant d'autres requêtes.

Le dernier état, *Erreur*, est atteint lorsqu'un problème est rencontré par le module : tout comme pour l'état *Application*, il s'agit d'un état temporaire, c'est-à-dire que le module ne reste dans cet état que le temps d'effectuer les actions attendues (appliquer une capacité ou traiter une erreur). L'état *Erreur* peut être atteint depuis deux autres états : à partir de l'état *Application* si une erreur se produit pendant l'application d'une capacité par le module, ou à partir de l'état *Actif* si une erreur est rencontrée au cours de la vie du module (par exemple, si un test d'assurance remonte un dysfonctionnement du module). Lorsque l'erreur a été traitée (les capacités et propriétés concernées ont été appliquées par un autre module), le module passe en mode *Indisponible*.

4.5.4 Assurance des propriétés

La deuxième partie de l'assurance concerne les propriétés. En effet, la section précédente présente les tests d'assurance liés aux mécanismes de sécurité. Cependant, il est essentiel de pouvoir vérifier si les propriétés de la politique ont l'effet attendu. Pour cela, nous définissons des tests d'assurance, directement liés aux propriétés mais indépendants des mécanismes utilisés.

4.5.4.1 Compilation

Les tests d'assurance pour les propriétés sont générés durant la phase d'application de la politique. Ils sont générés à partir du bloc d'assurance défini dans les propriétés et composé de capacités d'assurance. La génération des tests d'assurance pour les propriétés suit donc les algorithmes définis à la section 4.4.

La première étape de la génération des tests d'assurance consiste donc en la compilation du bloc d'assurance des propriétés (qui est faite durant la compilation de la politique, voir algorithme 12). On obtient alors une solution (un ensemble d'associations entre des capacités d'assurance et des mécanismes) permettant d'appliquer l'assurance des propriétés. Le même principe que pour l'application est repris, puisqu'il n'y a pas de différence intrinsèque entre les blocs d'application et ceux d'assurance.

4.5.4.2 Génération

Lorsqu'une solution a été trouvée pour chaque bloc d'assurance des propriétés, les différents modules d'extension sont contactés afin de générer les tests sous forme de scripts (qui peuvent eux-mêmes appeler des applications externes). Les modules sont appelés pour fournir les capacités d'assurance intervenant dans ces scripts. On considère donc que les appels à des capacités présents dans les blocs d'assurance correspondent à des fonctions du script. Les modules doivent donc renvoyer les fonctions correspondant aux capacités demandées.

La génération du script lui-même est réalisée par le moteur d'application et se base sur le bloc d'assurance de la propriété. Les appels aux capacités sont remplacés par les fonctions générées par les différents modules intervenant dans la propriété ou par le SE^E lui-même dans le cas de capacités internes (par exemple pour connaître les ressources ayant un contexte spécifique).

On obtient ainsi des scripts d'assurance spécifiques aux propriétés et dont les résultats peuvent être traités par le moteur d'assurance du SE^E.

4.5.5 Traitement des informations d'assurance

Les scripts d'assurance générés pour les mécanismes (section 4.5.3) et les propriétés (section 4.5.3) doivent être exécutés afin que leurs résultats puissent être traités par le moteur d'assurance. Cette section décrit donc comment les tests sont exécutés et quelles sont les conséquences des résultats d'assurance sur la politique et son application.

4.5.5.1 Exécution des scripts

L'exécution des scripts d'assurance est gérée par la propriété d'assurance qui peut être définie dans la politique de sécurité. Cette propriété prend en argument une fréquence qui détermine quand les scripts doivent être exécutés. La propriété d'assurance est donc définie comme suit :

```
1 boolean Assurance (Tests.Frequency SCFrequency) {
2  enforcement {
3   return run_assurance_tests (SCFrequency);
4  }
5 }
```

Listing 4.1 – Propriété d'assurance

Cette propriété appelle une seule capacité interne qui est chargée d'exécuter des tests d'assurance à une fréquence donnée. La propriété d'assurance est appliquée par un module spécifique qui fournit la capacité `run_assurance_tests`. Celui-ci est donc responsable de l'exécution des scripts à la bonne fréquence, mais également de la récupération de leurs résultats. De plus, le module est chargé de communiquer ces résultats au moteur d'assurance afin que celui-ci puisse interpréter les informations obtenues et prendre des décisions en conséquence.

Il s'agit ici d'une propriété d'assurance globale pour tous les tests. Une propriété plus évoluée pourrait paramétrer la fréquence des tests pour chaque propriété instanciée (annexe A.2).

4.5.5.2 Traitement et évolution de la politique

Les résultats d'assurance sont traités par le moteur d'assurance. Ils concernent donc les erreurs se produisant au cours de la vie du système, c'est-à-dire les erreurs ayant lieu après l'application de la politique de sécurité. Deux types d'erreurs peuvent être remontés :

– une défaillance est détectée sur l'un des mécanismes (algo. 23) ;
– l'application d'une propriété n'a pas l'effet attendu (algo. 24).

Le processus global de traitement des informations d'assurance est décrit à la figure 4.15.

Les deux cas traités sont donc une erreur sur un mécanisme ou une erreur sur une propriété. Une erreur sur un mécanisme entraîne la désactivation de ce mécanisme (voir algo. 23) et une erreur pour une propriété va provoquer une nouvelle application de la propriété concernée (voir algo. 24).

L'algorithme 23 présente la fonction ErreurMécanisme qui traite les messages en provenance des modules et indiquant qu'un mécanisme ne fonctionne pas comme attendu (par exemple, le mécanisme est inactif, un fichier de configuration est manquant, etc.).

Cette fonction est appelée lorsqu'un module d'assurance détecte une défaillance d'un mécanisme de sécurité utilisé par au moins une propriété. Dans ce cas, la liste des mécanismes disponibles est mise à jour dans la base de connaissances pour exclure le mécanisme considéré (ligne 3). Toutes les propriétés utilisant ce mécanisme sont alors désactivées (lignes 6 à 10) : la configuration des mécanismes utilisés est ramenée à l'état précédent

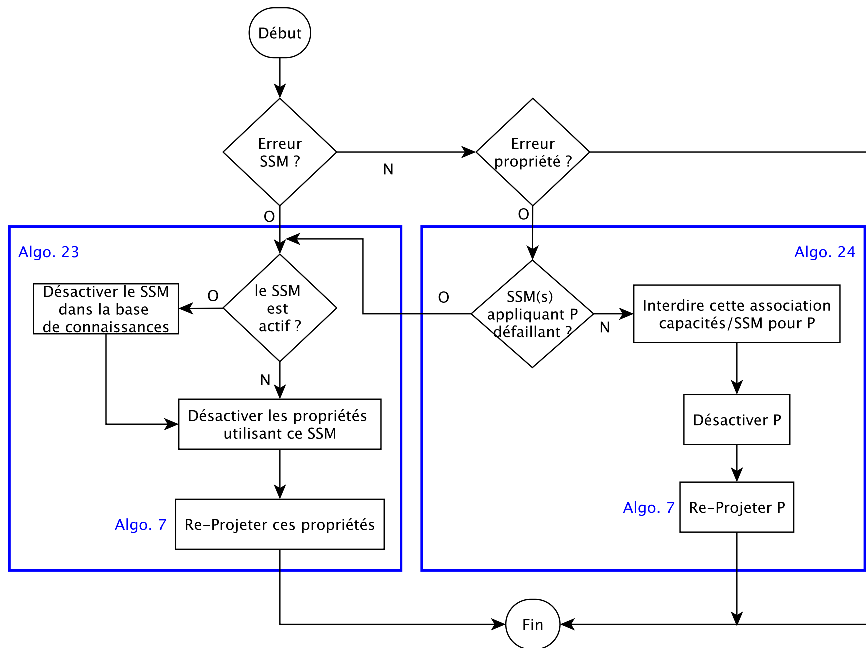

FIGURE 4.15 – Traitement d'une erreur au cours du fonctionnement du système

Algorithme 23 Dysfonctionnement d'un mécanisme M

1: **function** ERREURMÉCANISME (M)
2: **if** M est actif **then**
3: Supprimer M de la liste des SSMs disponibles
4: **end if**
5: listePropriétés$M \leftarrow \emptyset$
6: **for all** propriété P dans *Politique* **do**
7: **if** P utilise M **then**
8: Désactiver P
9: listePropriétés$M \leftarrow P$
10: **end if**
11: **end for**
12: **return** PROJETERPOLITIQUE(listePropriétésM) ▷ Algo. 8
13: **end function**

l'application des propriétés. La désactivation est gérée par chaque module d'extension (section 5.2.1) Ces propriétés sont ensuite projetées une nouvelle fois (ligne 12). Le mécanisme défaillant ayant été désactivé, l'application ne l'utilisera plus et le problème ne sera donc pas rencontré à nouveau.

L'algorithme 24 décrit la fonction ERREURPROPRIÉTÉ qui est utilisée lorsqu'une propriété a été appliquée mais n'a pas l'effet attendu. Par exemple, cette fonction est appelée si un fichier peut être lu par un utilisateur non autorisé après l'application d'une propriété de confidentialité : la propriété de confidentialité ne produit donc pas le comportement attendu et le moteur d'assurance tente d'y remédier.

La fonction ERREURPROPRIÉTÉ peut être appelée pour deux raisons :

– Si un ou plusieurs des mécanismes utilisés pour appliquer P sont défaillants, la propriété P n'a pas l'effet escompté. Cependant, cette erreur étant liée aux mécanismes, elle est traitée par la fonction ERREURMÉCANISME, qui est appelée pour chaque mécanisme défaillant (lignes 3 à 8).

Algorithme 24 Dysfonctionnement d'une propriété P

```
 1: function ErreurPropriété (P)
 2:     listeSSM ← ∅
 3:     for all M_i dans P.mécanismes do
 4:         if M_i n'est pas fonctionnel then
 5:             listeSSM ← M_i
 6:             ErreurMécanisme(M_i)                          ▷ Algo. 23
 7:         end if
 8:     end for
 9:     if listeSSM = ∅ then
10:         ListeSolutionsInterdites ← P.application
11:         Désactiver P
12:         return ProjeterPolitique(P)                        ▷ Algo. 8
13:     end if
14:     return ∅
15: end function
```

– Si tous les mécanismes sont fonctionnels (lignes 9 à 13), alors l'association capacités-mécanismes qui a été sélectionnée pour appliquer P est considérée comme invalide : elle est ajoutée à l'ensemble des associations à ne pas utiliser dans la base de connaissances (ligne 10). La propriété P est ensuite désactivée (ligne 11) puis appliquée une nouvelle fois (ligne 12) : une association capacités-mécanismes différente sera sélectionnée puisque celle-ci a été invalidée.

Ainsi, les algorithmes 23 et 24 permettent de réagir à des erreurs détectées au cours de la vie du système. Ces erreurs peuvent être dues à un dysfonctionnement d'un mécanisme ou à une propriété qui ne se comporte pas comme attendu. Dans les deux cas, le SE^E tente de corriger l'erreur détectée, ce qui se traduit par une tentative de réapplication des propriétés concernées par l'erreur. Cette tentative de correction peut être effectuée tant qu'une solution alternative est envisageable.

4.6 Scores d'application et cycle de vie de la politique

Dans la section 3.6 (p. 63), un score d'application de la politique (définition 3.6.2) a été défini en se basant sur le score d'application des propriétés (définition 3.5.6). Ce score permet de déterminer la qualité de l'application vis-à-vis de l'application idéale telle que vue par l'expert en sécurité.

Nous définissons ainsi deux scores d'application, le score d'application initial (définition 4.6.1) et le score d'application réel (définition 4.6.2) :

Définition 4.6.1: Score d'application initial
Le score d'application initial $S_{init}(P)$ est le score d'application d'une politique de sécurité P calculé à la fin de la compilation de la politique. Il s'agit donc du score d'application idéal sur un nœud donné.
Pour une politique P, ce score correspond à $Sapp(P)$ pour une configuration donnée, en ne considérant que les propriétés applicables.

Définition 4.6.2: Score d'application réel
Le score d'application réel $S_{reel}(P)$ est le score d'application effectif d'une politique de sécurité P, obtenu après l'application des propriétés. Ce score peut évoluer au cours du temps, par exemple si le moteur d'assurance provoque la réapplication de certaines propriétés après la détection d'un dysfonctionnement.

Ces deux scores permettent donc de calculer un taux de couverture de la politique :

Définition 4.6.3: Taux de couverture
Le taux de couverture $T_C(P)$ d'une politique de sécurité mesure la qualité de son application effective comparée à son application idéale sur le nœud considéré. Il est défini par :

$$T_C(P) = \frac{S_{reel}(P)}{S_{init}(P)}$$

Si $T_C(P) = 1$, alors l'application effective est identique à l'application idéale telle que définie par l'expert sécurité. De plus, plus le taux de couverture de la politique est proche de 1, meilleure est son application.

Cependant, les scores d'application étant basés sur les scores d'ordonnancement des mécanismes appliquant une propriété, T_C ne prend pas en compte les cas de non-application d'une propriété (par exemple si aucun mécanisme n'est disponible). Pour cette raison, nous définissons un taux de couverture des propriétés (définition 4.6.4)

Définition 4.6.4: Taux de couverture des propriétés
Le taux de couverture des propriétés $T_P(P)$ indique la quantité de propriétés appliquées comparée aux propriétés de la politique P.
Notons $N_P(P)$ le nombre de propriétés de la politique et $N_{Pa}(P)$ le nombre de propriétés appliquées. On a alors :

$$T_P(P) = \frac{N_{Pa}(P)}{N_P(P)}$$

Si $T_P(P) = 1$, alors toutes les propriétés ont été appliquées. Par conséquent, on peut définir l'application idéale d'une politique de la façon suivante :

Définition 4.6.5: Application idéale d'une politique
L'application d'une politique P est dite idéale si toutes les propriétés ont été appliquées et si leur application effective est identique à l'application idéale telle que définie par l'expert sécurité. On a donc :

$$\textit{L'application de P est idéale} \Leftrightarrow \left\{ \begin{array}{lcl} T_C(P) & = & 1 \\ T_P(P) & = & 1 \end{array} \right.$$

Les scores et les taux de cette section permettent d'évaluer la qualité de l'application d'une politique de sécurité. Ils seront utilisés dans le chapitre 6 qui présente une expérimentation.

Enfin, ces quatre définitions (scores d'application initial et réel, taux de couverture, et taux de couverture des propriétés) peuvent être étendues pour s'appliquer à un ensemble de machines et de SE^E. On les définit donc de la manière suivante :

Définition 4.6.6: Scores et taux d'un ensemble de politiques
Soit $P = P_{i,1 \leq i \leq n}$ un ensemble de n politiques de sécurité à appliquer sur n machines.
Les scores d'applications (initial et réel) de P et ses taux de couverture sont définis par :

$$\left\{ \begin{array}{lcl} S_{init}(P) & = & \sum_{i=1}^{n} S_{init}(P_i) \\ S_{reel}(P) & = & \sum_{i=1}^{n} S_{reel}(P_i) \end{array} \right. \quad \textit{et} \quad \left\{ \begin{array}{lcl} T_C(P) & = & \prod_{i=1}^{n} T_C(P_i) \\ T_P(P) & = & \prod_{i=1}^{n} T_P(P_i) \end{array} \right.$$

L'utilisation de ces scores et taux pour un ensemble de politique permet donc d'évaluer la qualité de l'application d'une politique de sécurité pour une architecture logicielle complète.

L'application de la politique de sécurité est un processus dynamique. En effet, cette application peut évoluer, soit en raison d'un changement de la politique de sécurité, soit en raison d'un dysfonctionnement détecté par le processus d'assurance. La figure 4.16 résume le cycle de vie complet de la politique, qui comprend l'ensemble des algorithmes qui ont été présentés dans ce chapitre.

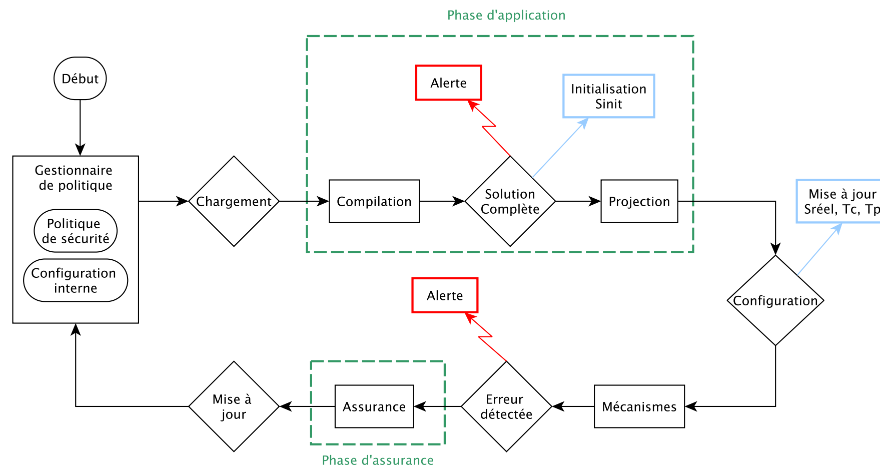

FIGURE 4.16 – Cycle de vie de la politique

Ainsi, la politique de sécurité est chargée par le moteur d'application et est traitée par les étapes de compilation et de projection (phase d'application). Si certaines propriétés ne peuvent pas être compilées, une alerte sera générée. A la fin de la phase d'application, les mécanismes de sécurité sont configurés en accord avec la politique initiale. De plus, le score d'application initial et le score d'application réel sont initialisés, ainsi que les taux de couverture.

En cas d'erreur d'un mécanisme ou d'une propriété, le moteur d'assurance intervient afin de mettre à jour la politique en fonction de l'erreur. Une mise à jour de la politique entraîne alors une nouvelle application des propriétés concernées et une mise à jour du score d'application réel et des taux de couverture.

4.7 Conclusion

Ce chapitre a présenté l'architecture permettant d'interpréter, de projeter et d'assurer une politique définie dans le langage d'expression des besoins de sécurité du chapitre 3. Il décrit les différents composants de l'architecture ainsi que leurs interactions afin de maintenir une application de la politique cohérente et efficace.

L'architecture proposée suit les principes de l'informatique autonome, ce qui lui permet de s'adapter à son environnement. Les différents algorithmes permettant de fournir les fonctionnalités de l'architecture ont été détaillés dans ce chapitre.

L'architecture est capable de compiler le langage d'expression des besoins de sécurité et de le projeter sur les mécanismes disponibles. Cette projection s'adapte donc à l'état des systèmes considérés et aux mécanismes présents sur ceux-ci. L'application des propriétés est également configurable grâce aux trois modes de sélection (`Défaut`, `Best Effort` et `Maximum`) qui permettent d'adapter la sélection des mécanismes en fonction des choix de l'expert en sécurité (par exemple, le mode `Best Effort` offre de meilleures performances,

et le mode `Maximum` une protection renforcée) De plus, l'architecture gère l'intégralité du langage, ce qui permet d'appliquer la politique et de mettre en place un système d'assurance des propriétés la composant. Par conséquent, l'architecture est capable de détecter d'éventuelles erreurs au cours de la vie des systèmes.

L'architecture repose sur l'utilisation de mécanismes de sécurité existants. Cela permet à l'architecture d'adresser de nombreux besoins de sécurité, notamment systèmes, réseaux ou applicatifs. De plus, de nouveaux mécanismes peuvent être ajoutés à l'ensemble de ceux gérés par l'architecture, ce qui lui permet d'être évolutive et de s'adapter à des systèmes différents et à de nouveaux besoins. Notons que l'architecture proposée n'a pas pour objectif d'être un nouveau mécanisme de sécurité indépendant : elle vise à réutiliser les capacités offertes par les mécanismes existants et à répondre ainsi à un large spectre de besoins de sécurité.

L'architecture propose également une partie dédiée à l'assurance des propriétés. Elle permet donc une reconfiguration automatique et dynamique des mécanismes en fonction des évènements se produisant sur le système. Ainsi, l'architecture est capable de réagir aux erreurs détectées suite à la génération de tests d'assurance, par exemple un dysfonctionnement d'un mécanisme ou une mauvaise application d'une propriété. Cette réaction peut se traduire par une nouvelle projection des propriétés concernées par l'erreur : l'indépendance du langage vis-à-vis des mécanismes est donc essentielle pour que cette reconfiguration de la politique soit possible. Par conséquent, l'association d'un langage abstrait exprimant les besoins de sécurité et d'une architecture autonome est ce qui permet de fournir un système de reconfiguration automatique.

Enfin, une méthode d'évaluation de la politique de sécurité a également été introduite. Cette évaluation se base sur des scores d'ordonnancement permettant à l'expert en sécurité de classer les mécanismes de sécurité disponibles et aux différents algorithmes de sélectionner la combinaison idéale pour une propriété sur un système donné. Les scores d'ordonnancement sont alors utilisés pour définir le score d'application des propriétés et de la politique. La qualité de l'application de la politique peut alors être déterminée en utilisant ce score d'application et en l'associant au taux de couverture et au taux de couverture des propriétés. Il est ainsi possible de déterminer si la politique a été entièrement appliquée et si son application peut être améliorée sur un nœud donné. De plus, les applications d'une même politique sur des nœuds différents peuvent être comparées, ce qui permet de comparer des environnements différents vis-à-vis d'une politique de sécurité.

Le chapitre 5 présentera une implémentation de cette architecture et une expérimentation sera détaillée au chapitre 6.

Chapitre 5

Implémentation

Cette thèse a été effectuée dans le cadre du projet européen Seed4C. L'un des objectifs du projet était de fournir des prototypes fonctionnels pour les différents éléments de l'architecture proposée.

Le *Security Enforcement Engine* (SE^E) est donc l'un des prototypes développés dans ce cadre et a pour objectif d'appliquer une politique de sécurité dans un environnement en nuage. Ainsi, le rôle du SE^E est d'interpréter une politique de sécurité exprimée avec le langage du chapitre 3 afin de la projeter sur les mécanismes de sécurité et d'assurer une mise à jour automatique de la politique en cas d'erreur. Cela est fait en suivant le processus détaillé au chapitre 4.

Ce chapitre décrit l'implémentation du SE^E (section 5.1) et celle des modules d'extension (section 5.2). Enfin, le chapitre présente des exemples de projection pour quelques propriétés (section 5.3). Le chapitre 6 présentera une expérimentation reprenant l'ensemble des éléments des différents chapitres.

5.1 Implémentation du SE^E

Cette section décrit l'implémentation du SE^E et de ses différents composants.

Le SE^E ayant pour objectif de déployer une politique de sécurité dans un environnement hétérogène, il a été développé en Java afin d'être facilement porté sur différents systèmes. Dans le cadre des expérimentations du projet et de cette thèse (chapitre 6), le SE^E a été testé sur plusieurs distributions Linux : CentOS 6 et 7, Fedora 19 et 20, Debian 6 et Ubuntu 14.04.

Le SE^E est constitué de plusieurs composants qui appliquent les algorithmes d'application et d'assurance définis au chapitre 4. Ces composants sont décrits dans les sections qui suivent.

5.1.1 Architecture globale

La figure 5.1 décrit le fonctionnement général du SE^E dont les principaux composants sont les suivants :

- Le **gestionnaire de politique** reçoit la politique de sécurité à appliquer (les propriétés instanciées et les contextes mis en jeu) et contient la configuration du SE^E (les propriétés pouvant être utilisées, les capacités existantes, les mécanismes disponibles) ;
- Le **moteur d'application** interprète la politique de sécurité et décide comment l'appliquer en fonction des mécanismes disponibles (algorithme 8, section 4.4) ;

– Le **moteur d'assurance** gère les erreurs durant la phase d'application de la politique et les informations d'assurance remontées lors de l'exécution du système. Il met aussi à jour la configuration du SE^E en fonction des erreurs rencontrées (algorithme 20, section 4.5) ;
– Le **moteur de communication** permet la communication et la coopération entre modules d'extension et entre SE^E ;
– Les **modules d'extension** gèrent les communications avec les mécanismes afin de configurer ceux-ci en accord avec la politique et ils permettent également la récupération des informations d'assurance.

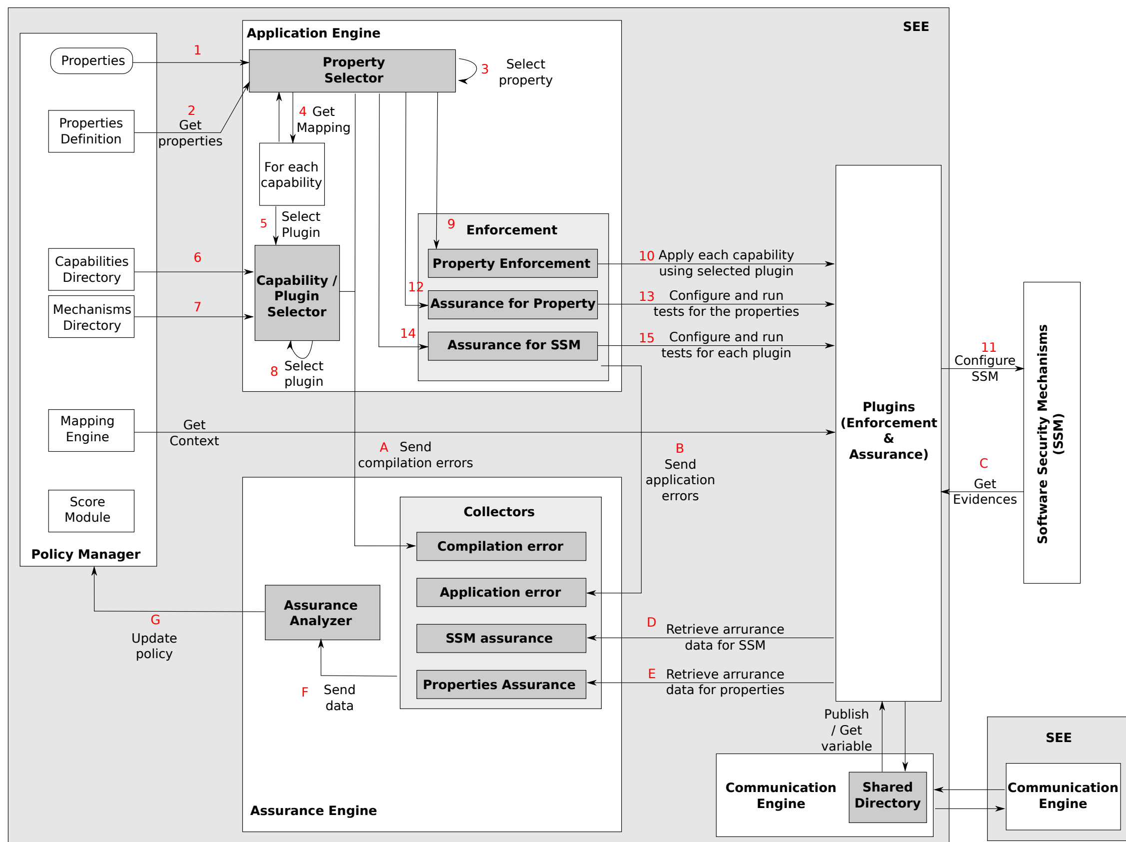

FIGURE 5.1 – Architecture détaillée du SE^E

5.1.2 Flux d'exécution

Cette section décrit le flux global d'application et d'assurance de la politique de sécurité à travers ses éléments centraux et en se basant sur la figure 5.1. Les modules d'extension, qui ne font pas partie du cœur du SE^E, sont décrits à la section 5.2.

5.1.2.1 Gestionnaire de politique

Afin de sécuriser une architecture logicielle, le SE^E a besoin de deux éléments : sa **configuration interne** et la **politique de sécurité**.

La configuration interne du SE^E est prédéfinie mais peut être modifiée par l'expert en sécurité. Elle contient les propriétés, les capacités, ainsi que les mécanismes disponibles et les capacités qui leur sont associées.

La politique de sécurité est définie par l'administrateur de l'architecture logicielle et exprime ses besoins de sécurité. Elle contient les contextes identifiant les ressources (ainsi que leur association) et les propriétés instanciées mettant en jeu ces contextes.

La configuration interne évolue en fonction des résultats d'assurance. En effet, la détection d'un dysfonctionnement d'un mécanisme ou d'un comportement inattendu d'une propriété peut entraîner la désactivation d'un module d'extension ou la réapplication d'une propriété.

Enfin, le module de score calcule et met à jour les scores d'application (initial et réel), le taux de couverture et le taux de couverture des propriétés en fonction des résultats fournis par les moteurs d'application et d'assurance.

5.1.2.2 Moteur d'application

Lorsque le gestionnaire de politique a chargé la configuration et la politique, le moteur d'application peut compiler la politique afin de la projeter sur les mécanismes (phase d'application).

Le moteur de sélection de propriétés est responsable de la première partie du processus d'application (étapes 1 à 3). Il interroge le gestionnaire de politique (étape 2) afin de déterminer quelles propriétés peuvent être utilisées pour appliquer une propriété instanciée (c'est-à-dire déterminer quelles sont les propriétés compatibles). Ce moteur implémente l'algorithme 9 (section 4.4.2, p. 78).

Dans un second temps, le moteur de sélection des mécanismes recherche, pour chaque propriété compatible avec la propriété instanciée, une solution associant des mécanismes aux capacités et maximisant le score d'application de la propriété (étapes 4 à 8). Cette opération implique également des échanges avec le gestionnaire de politique pour savoir quelles capacités peuvent être utilisées (étape 6), quels mécanismes les fournissent (étape 7) et quelles sont leurs contraintes de compatibilité et de dominance. Cette partie implémente l'algorithme 12 (section 4.4.3, p. 80).

Les propriétés instanciées peuvent alors être projetées selon la solution trouvée. Cela correspond à l'algorithme 18 (section 4.4.4, p. 89). Trois étapes se succèdent. Tout d'abord, le moteur d'application projette les propriétés : il contacte les modules d'extension afin d'appliquer les capacités du bloc d'application (étapes 9 à 11). Dans une deuxième phase, le moteur d'application contacte les modules pour appliquer les blocs d'assurance des propriétés, ce qui permet de vérifier que les propriétés appliquées ont bien l'effet attendu (étapes 12 et 13). Enfin, pour chaque module qui a été utilisé pour au moins une capacité, le moteur d'application demande la génération de tests d'assurance pour vérifier le statut du mécanisme associé (étapes 14 et 15).

Il est à noter que, tout au long du processus d'application (phases de compilation et de projection) et dès lors que des erreurs sont rencontrées, des informations sont envoyées au moteur d'assurance (étapes A et B).

5.1.2.3 Moteur d'assurance

Le processus d'assurance se décompose en deux étapes principales : le traitement des erreurs relatives à l'application de la politique de sécurité et celui des informations obtenues au cours de l'exécution du système.

Tout d'abord, le moteur d'assurance reçoit et traite les erreurs se produisant lors de l'application de la politique (étapes A et B), en suivant le processus décrit à la section 4.5.2 (p. 92). Le moteur d'assurance obtient ces informations depuis le moteur d'application, ce qui correspond à une erreur lors de la compilation ou de la projection d'une propriété.

Le second cas traité par le moteur d'assurance concerne des erreurs détectées au cours de la vie du système (étapes C). Ces erreurs se produisent alors que la politique a été correctement appliquée et peuvent avoir deux causes : un dysfonctionnement d'un mécanisme (étape D) ou une propriété n'ayant pas l'effet escompté (étape E).

L'ensemble de ces informations est alors interprété afin de mettre à jour la base de connaissances du SE^E (étapes F et G). Une telle mise à jour implique une nouvelle projection de la sous-partie de la politique de sécurité ayant rencontré un problème.

5.1.2.4 Moteur de communications

Finalement, pour qu'une application cohérente de la politique de sécurité soit possible, il est nécessaire que les modules d'extension et les SE^E puissent communiquer. En effet, les SE^E forment un système multi-agents, ce qui implique qu'ils puissent communiquer entre eux afin de se coordonner et d'échanger des informations. Deux types de communications peuvent se produire : soit la communication s'effectue entre deux modules d'un même SE^E, soit elle fait intervenir des SE^E distants. Ces communications se produisent par l'intermédiaire du répertoire partagé du SE^E.

Partage de données Des échanges d'informations peuvent se produire lorsque les SE^E ont besoin de partager des données dans le cadre d'une propriété. Par exemple, dans le cas de la propriété de confidentialité réseau, les capacités internes **`publish`** et **`get_published`** permettent à deux SE^E de partager une clef de chiffrement symétrique (voir propriété `P3`, section 3.5.2.2). De plus, des communications peuvent être nécessaires entre plusieurs modules d'un même SE^E. Ainsi, un module souhaitant utiliser un port peut demander son ouverture à un module de pare-feu : le port à ouvrir est transmis grâce au module de communication.

Ce partage utilise les mêmes capacités internes pour la communication intra et inter-SE^E. Seuls les contextes mis en jeu lors de l'appel à ces capacités indiquent de quel type de communication il s'agit.

Communications intra-SE^E Dans le cas de communications intra-SE^E, deux types de communications peuvent avoir lieu : les échanges de données et les demandes d'application de capacités supplémentaires.

Les échanges de données se produisent lorsqu'un mécanisme a besoin d'une information générée par un autre mécanisme. Par exemple, un mécanisme fournissant une capacité de chiffrement doit récupérer la localisation de la clef à utiliser. Celle-ci peut avoir été générée par un autre mécanisme et il est donc nécessaire que ces deux mécanismes puissent communiquer. Dans le prototype proposé, cet échange d'information est fait par l'intermédiaire d'un annuaire local et interne au SE^E. Chaque module d'extension peut publier des informations dans cet annuaire (section 3.4.4, p. 51). Ces informations sont à destination d'un seul module et sont associées à la propriété en cours d'application. Par conséquent, l'information ne peut être lue que par le module destinataire et uniquement pendant l'application de la propriété. Les informations ne peuvent donc pas être récupérées par d'autres modules. L'information stockée par le SE^E est de la forme suivante :

$$(Source, Destination, Propriété, Information)$$

où :

- *Source* et *Destination* désignent respectivement le module publiant l'information et celui autorisé à la récupérer ;

– *Propriété* indique quelle propriété demande le partage d'information ;
– *Information* contient les données effectivement partagées.

Ces quatre informations et les données partagées sont représentées sous forme de contexte. En effet, lorsque deux modules d'extension souhaitent partager de l'information, cela a pour but d'enrichir les contextes mis en jeu par les capacités de la propriété concernée.

Le deuxième type de communications intra-SE^E se produit lorsqu'un module demande l'application de capacités supplémentaires. Des capacités supplémentaires peuvent par exemple être demandées dans le cas de mécanismes dépendants. En effet, lorsqu'un mécanisme M_1 dépendant d'un mécanisme M_2 est appelé, cette dépendance peut entraîner l'appel à une capacité de M_2. Si un tel appel se produit, il faut donc que M_1 puisse demander au moteur de projection l'application d'une capacité par M_2.

Considérons le cas d'un module IPSec dépendant du module Racoon. Alors, lors de l'application de la capacité `encrypt_flow`, le module IPSec appelle le module Racoon pour appliquer la capacité `create_security_association`. Ce type de communications est automatiquement réalisé par le moteur d'application qui, lors de l'application d'une capacité, récupère une valeur de retour contenant deux éléments : une valeur booléenne qui indique si l'application de la capacité a rencontré ou non une erreur et une liste des capacités supplémentaires requises.

Communications inter-SE^E Des modules d'extension appartenant à différents SE^E peuvent également avoir besoin de communiquer. Cela se produit lors de l'application de propriétés réseaux impliquant plusieurs nœuds. De telles communications doivent être réalisées de manière sécurisée et elles se font donc en utilisant le réseau administrateur d'OpenStack (la plateforme d'informatique en nuage utilisée dans le cadre de nos expérimentations). Si une autre plateforme était utilisée, il faudrait de la même manière que les communications entre SE^E aient lieu sur un réseau sécurisé différent du réseau utilisateur ou que les communications entre SE^E soient chiffrées et authentifiées.

Des SE^E peuvent communiquer dans deux types de situations : lors de l'application de capacités ayant besoin de partager de l'information et lors des phases de négociation pour la projection.

Différentes solutions peuvent être envisagées pour implémenter le partage d'information entre SE^E. Dans le cadre de notre prototype, nous utilisons l'annuaire interne des SE^E, déjà utilisé pour les communications intra-SE^E. Un annuaire global pourrait également être utilisé, mais des annuaires locaux permettent de conserver un système décentralisé. De la même manière que lors des échanges intra-SE^E, les informations publiées par un SE^E ne sont visibles que par le SE^E destinataire. Ainsi, chaque information publiée est associée à sa source et à son destinataire. La source et le destinataire peuvent être soit un SE^E, soit un module d'extension spécifique d'un SE^E : si le destinataire est un module, les autres modules du SE^E destinataire n'auront pas accès à l'information publiée. Dans les deux cas, l'information publiée est associée à la propriété ayant publié l'information : en effet, cette information ne sera disponible que pour cette propriété, appliquée sur un nœud distant.

Le deuxième cas de communications inter-SE^E se produit lors de l'application de propriétés réseaux. En effet, pour appliquer une propriété réseau, une phase de négociation est nécessaire afin de sélectionner des modules compatibles pour chacune des capacités. Cette négociation se fait également sur le réseau dédié aux communications entre SE^E.

L'algorithme 25 permet de sélectionner les couples de mécanismes disponibles sur deux nœuds N_1 et N_2 pour une capacité c.

Algorithme 25 Sélection de modules d'extension compatibles

1: **function** SelectConsistentPlugin (capacité c, nœud N_1, nœud N_2)
2: $SSM_1 \leftarrow$ TrouverEtOrdonnerSSM(c) sur N_1
3: $SSM_2 \leftarrow$ TrouverEtOrdonnerSSM(c) sur N_2
4: $CompNet \leftarrow$ Matrice de compatibilité réseau ▷ Section 3.4.4
5: $listeSSM \leftarrow \emptyset$
6: **for all** mécanisme M_i dans SSM_1 **do**
7: **for all** mécanisme M_j dans SSM_2 **do**
8: **if** M_i et M_j sont compatibles (d'après $CompNet$) **then**
9: $score \leftarrow$ somme des scores d'ordonnancement de M_i et M_j
10: Ajouter (M_i, M_j) à $listeSSM$ selon $score$
11: **end if**
12: **end for**
13: **end for**
14: **return** $listeSSM$
15: **end function**

Cet algorithme utilise la fonction TrouverEtOrdonnerSSM afin d'obtenir les mécanismes pouvant appliquer c sur chacun des nœuds considérés (lignes 2 et 3). Puis, il récupère la matrice de compatibilité réseau indiquant quels sont les mécanismes compatibles pour appliquer une même capacité au niveau réseau (ligne 4). Enfin, l'algorithme parcourt les mécanismes disponibles sur N_1 (lignes 6 à 13) et ceux disponibles sur N_2 (lignes 7 à 12) afin de trouver des mécanismes compatibles (ligne 8). Les couples de mécanismes compatibles sont alors ajoutés à la liste de couples possibles (ligne 10) selon la somme de leur score d'ordonnancement (ligne 9). Finalement, cette liste de couples ordonnés est renvoyée par la fonction (ligne 13) et sera utilisée comme un domaine par les algorithmes de recherche de solution.

Grâce à cet algorithme, il est ainsi possible de sélectionner les mécanismes de sécurité de façon à ce que les applications des propriétés réseaux par plusieurs SE^E soient compatibles.

5.2 Implémentation des modules d'extension

Lorsque le moteur d'application a sélectionné les mécanismes à utiliser pour appliquer les propriétés, il contacte les modules d'extension responsables des mécanismes. Cette section détaille les éléments généraux de l'implémentation d'un module d'extension et décrit brièvement chacun des modules existants.

Un module d'extension permet au SE^E de demander l'application d'une capacité par un mécanisme : les modules font ainsi le lien entre le moteur d'application et les mécanismes. Un module est donc associé à un mécanisme spécifique et fournit des capacités correspondant aux fonctionnalités de sécurité de ce mécanisme.

L'implémentation des modules d'extension est laissée à la discrétion de leurs développeurs : le SE^E n'impose pas de méthode d'implémentation spécifique, hormis le respect d'une interface pour la prise en compte du module. En effet, l'une des caractéristiques essentielles du SE^E est la possibilité d'y intégrer de nombreux mécanismes de sécurité hétérogènes qui ne peuvent donc pas tous être gérés de manière identique. Par conséquent, l'unique contrainte imposée est que les modules doivent implémenter l'interface. Celle-ci est utilisée par le SE^E pour interagir avec le module, par exemple pour demander l'application de capacités ou pour récupérer des informations d'assurance.

Dans le cadre du projet, une documentation détaillant les étapes nécessaires à l'implémentation d'un nouveau module a été rédigée. Cette documentation a été fournie à l'ensemble des partenaires, ce qui a permis le développement de modules additionnels.

5.2.1 Interface

Chaque module d'extension doit implémenter un interface Java fournie par le SE^E. Le fonctionnement général d'un module est décrit à la figure 5.2.

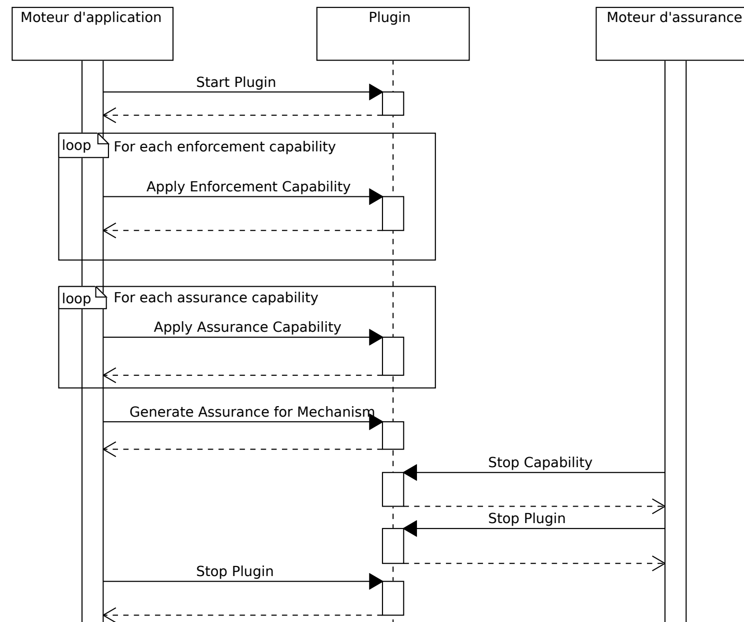

FIGURE 5.2 – Interface moteur d'application et module d'extension

5.2.1.1 Application

Les méthodes `startPlugin` et `stopPlugin` permettent d'effectuer des actions lors de l'initialisation ou de l'arrêt du module. Par exemple, le module peut déterminer la configuration du mécanisme avant l'application de la politique ou vérifier que le mécanisme est dans l'état attendu (par exemple, si le mécanisme doit être actif avant toute configuration par le SE^E). La méthode `stopPlugin` est quant à elle appelée lors de l'arrêt du SE^E.

La méthode d'application d'une capacité est utilisée par le moteur d'application pour demander au module d'interagir avec son mécanisme associé. La méthode d'application d'une capacité demande ainsi à un module de configurer son mécanisme pour répondre à la capacité. La capacité à appliquer est précisée par son identifiant et les ressources concernées sont données dans une liste de contextes passée en argument. L'application de la capacité est ensuite laissée à la discrétion du module. Cette méthode est appelée pour l'ensemble des capacités d'application d'une propriété (première boucle), mais également pour les capacités d'assurance (seconde boucle).

Une méthode permet également de demander la génération des tests d'assurance pour les mécanismes. Elle est appelée une seule fois pour chaque module actif (c'est-à-dire pour chaque module ayant appliqué au moins une capacité). Cette méthode permet de signaler aux modules la fin de l'application de la politique et de demander la génération de tests d'assurance spécifiques à chacun des mécanismes. Ces tests peuvent vérifier que le mécanisme associé au module est bien fonctionnel, mais ils peuvent aussi vérifier que la configuration attendue est bien en place (par exemple pour s'assurer que la configuration n'a pas été modifiée par un intervenant extérieur au SE^E ou que de telles modifications n'impactent pas l'application des propriétés).

5.2.1.2 Assurance

Chaque module d'extension dispose également d'une méthode permettant de désactiver des capacités. Elle reçoit les mêmes arguments que la méthode d'application et est utilisée pour annuler l'application d'une capacité. Elle peut par exemple être utilisée dans le cas où l'application d'une propriété échoue : il faut alors annuler les capacités déjà appliquées composant cette propriété. Cette méthode retourne une valeur booléenne permettant ainsi de fournir des informations au moteur d'assurance sur la réussite ou l'échec de la désactivation. De la même manière que pour l'application, la désactivation d'une capacité est entièrement gérée par le module qui doit donc être capable de ramener le système à son état précédent.

Cette méthode est donc appelée par le moteur d'assurance lorsqu'il détecte un dysfonctionnement à partir des scripts d'assurance générés précédemment. Lorsque toutes les capacités d'un module ont été désactivées, le module est stoppé. Le moteur d'application peut alors réappliquer les propriétés en contactant d'autres modules.

5.2.2 Déclaration des capacités

Pour utiliser les capacités fournies par un module d'extension, et donc par son mécanisme associé, ces capacités doivent être déclarées au SE^E par le module.

Le listing 5.1 illustre comment les capacités sont déclarées.

```
1  <?xml version="1.0"?>
2  <capabilitiesList>
3      <capability name="capability_name" id="id_X">
4          <argument id="1">
5              <type>attributeType_1</type>
6              <type>Type.Passive.Data.File</type>
7          </argument>
8          <argument id="2">
9              [...]
10         </argument>
11         <return_value>
12             <type>returnType</type>
13         </return_value>
14     </capability>
15 </capabilitiesList>
```

Listing 5.1 – Schéma de déclaration des capacités

La déclaration des capacités est indépendante des mécanismes qui les fournissent. La définition d'une capacité inclut le nom de cette capacité (attribut `name`, ligne 3) et un identifiant unique (attribut `id`). Puis, les arguments attendus par la capacité sont spécifiés (lignes 4 à 10). Chaque argument attend un contexte ayant un ou plusieurs types d'attribut spécifiques (lignes 5 et 6), par exemple `Type.Passive.Data.File` (ligne 6). Enfin, le type de retour de la capacité est précisé (lignes 11 à 13).

Une fois les capacités définies, elles doivent être associées aux modules d'extension qui peuvent les appliquer, comme décrit au listing 5.2.

```
1  <?xml version="1.0"?>
2  <pluginsList>
3      <ssm name="path.to.Plugin">
4          <capabilitiesList>
5              <capability id="id_X"/>
6              <capability id="[...]" score="n"/>
7          </capabilitiesList>
```

```
8      </ssm>
9      <ssm name="path.to.Plugin2">
10         [...]
11     </ssm>
12 </pluginsList>
```

Listing 5.2 – Schéma d'association des capacités aux modules d'extension

Chaque module possède un élément ssm dont l'attribut name a pour valeur le nom de la classe Java associée au module (ligne 3). La liste des capacités (lignes 4 à 7) contient des éléments capability (lignes 5 et 6) pour lesquels l'attribut id fait référence aux identifiants uniques des capacités (voir listing 5.1). Un attribut optionnel score peut être précisé (ligne 6) : il correspond au score d'ordonnancement du mécanisme pour la capacité. Par défaut, ce score est fixé à 1.

Ces deux fichiers permettent ainsi de définir les capacités et d'associer ces capacités aux différents modules d'extension qui les fournissent. La séparation de ces deux définitions permet de refléter l'indépendance des capacités vis-à-vis des modules : l'ajout ou la suppression d'un module ne modifie pas les définitions des capacités existantes, mais uniquement l'ensemble des modules disponibles pour une capacité donnée. De même, une capacité peut être associée à plusieurs modules.

5.2.3 Fonctionnement classique d'un module d'extension

L'implémentation de chaque module d'extension est libre, à l'exception de l'interface à implémenter qui permet les communications avec le SE^E. Cependant, les modules actuels suivent tous un modèle similaire qui est décrit à la figure 5.3. Ce modèle correspond aux mécanismes utilisant des fichiers textes pour leur configuration, ce qui le rend adapté aux systèmes de type Unix.

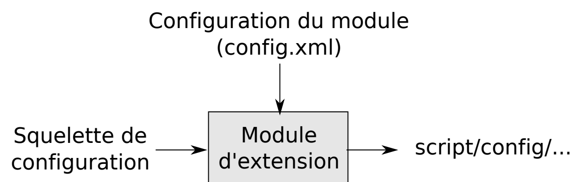

Figure 5.3 – Proposition d'architecture de module d'extension

Le modèle classique d'un module utilise un squelette de configuration et un fichier de configuration définissant des variables (commandes, noms de fichiers...) en fonction de la distribution utilisée. Ces deux fichiers sont utilisés pour générer des scripts ou des fichiers de configuration afin de configurer le mécanisme associé en fonction de la capacité à appliquer et du système considéré.

Un module qui suit ce modèle utilise un fichier de configuration (config.xml). Il s'agit d'un fichier XML définissant des variables qui peuvent être spécifiques aux systèmes utilisés. Son format est présenté au listing 5.3.

```
1 <?xml version="1.0"?>
2 <systemsConfig>
3 <system name="system1">
4     <options>
5         <key_1>value_1</key_1>
6         <key_2>value_2</key_2>
7         [...]
8     </options>
```

```
9  </system>
10 <system name="centos">
11     <options>
12         <logs>/var/log/seed_pam.log</logs>
13         <grepCmd>/bin/grep</grepCmd>
14         <notifyLoginConfig>/etc/seed/scripts/notify-login</notifyLoginConfig>
15         [...]
16     </options>
17 </system>
18 </systemsConfig>
```

Listing 5.3 – Schéma d'un fichier de configuration de module d'extension

L'élément `systemsConfig` (lignes 2 à 17) contient autant d'éléments `system` que de systèmes sur lequel le module peut être utilisé. L'élément `system` (lignes 3 à 9) possède un attribut `name` (ligne 3) permettant d'identifier le système considéré (par exemple, CentOS ou Debian). Il possède aussi des `options` (lignes 4 à 8), où chaque option est composée d'un nom et d'une valeur : il s'agit des variables pouvant être utilisées par le module lors de la configuration d'un mécanisme. Cela permet de gérer l'hétérogénéité des systèmes, par exemple les différences de localisation des binaires et des fichiers systèmes.

Considérons la configuration du module PAM pour CentOS (lignes 10 à 16). Trois options sont précisées, `logs`, `grepCmd` et `notifyLoginConfig`, et les valeurs sont spécifiques à CentOS : en effet, sous d'autres distributions, ces fichiers peuvent avoir une autre localisation (`/usr/bin/grep` pour `grepCmd` sur Fedora). Lorsque le module reçoit une demande d'application d'une capacité par le moteur de projection, il peut utiliser un squelette de configuration. Ce squelette peut correspondre à un script, à un fichier de configuration ou à une commande à effectuer. Il comprend des variables que le moteur d'application doit substituer par les valeurs définies dans le fichier de configuration ou dans les arguments de la capacité à appliquer. Le module PAM utilise le squelette du listing 5.4.

```
1 #!/bin/bash
2 PAMSTRING="session  optional  pam_exec.so  log=${logs}  ${notifyLoginConfig}"
3 ${grepCmd} -q "$PAMSTRING" /etc/pam.d/system-auth || echo "$PAMSTRING" >> /etc/
    pam.d/system-auth
4 ${grepCmd} -q "$PAMSTRING" /etc/pam.d/password-auth || echo "$PAMSTRING" >> /
    etc/pam.d/password-auth
```

Listing 5.4 – Exemple de squelette pour le module d'extension de PAM

Ce squelette est utilisé pour ajouter aux fichiers de configuration de PAM l'appel à un script `notifyLoginConfig`. Il comprend trois variables (identifiées par `${}`) : `logs`, `notifyLoginConfig` et `grepCmd`. Ces variables sont donc remplacées par les valeurs du fichier de configuration du module. Pour CentOS, on obtient ainsi le script du listing 5.5.

```
1 #!/bin/bash
2 PAMSTRING="session  optional  pam_exec.so  log=/var/log/seed_pam.log  /etc/seed
    /scripts/notify-login"
3 /bin/grep -q "$PAMSTRING" /etc/pam.d/system-auth || echo "$PAMSTRING" >> /etc/
    pam.d/system-auth
4 /bin/grep -q "$PAMSTRING" /etc/pam.d/password-auth || echo "$PAMSTRING" >> /etc
    /pam.d/password-auth
```

Listing 5.5 – Exemple de script pour le module d'extension de PAM

Le script obtenu peut alors être exécuté par le module afin de configurer PAM comme demandé par la capacité (ici, exécuter un script lors d'une authentification réussie et écrire les logs dans le fichier `seed_pam.log`).

Ainsi, en prenant en compte le fichier de configuration et les squelettes, le module peut configurer le mécanisme en fonction des capacités demandées et indépendamment du système utilisé. Le SE^E fournit une API pour automatiser la génération des fichiers de configuration et des scripts en fonction de la configuration du module et des squelettes de fichiers. Cela permet de simplifier l'implémentation d'un module pour les mécanismes reposant sur des fichiers de configuration ou sur l'utilisation de scripts.

5.2.4 Modules d'extension existants

Cette section présente les différents modules d'extension qui ont été développés. La section 5.2.4.1 présente les modules qui ont été développés dans le cadre de cette thèse et la section 5.2.4.2 introduit brièvement ceux ayant été développés par des partenaires du projet Seed4C. Les mécanismes associés à ces modules ont été décrits à la section 2.3.

5.2.4.1 Exemples de modules d'extension

iptables Un module iptables a été développé afin de disposer de ses fonctionnalités de pare-feu. Il fournit donc des capacités permettant d'autoriser ou d'interdire des communications réseaux, au niveau des ports, des interfaces, des adresses IP, de rediriger des communications sur un autre port, etc. Ainsi, le module génère des règles de pare-feu pour chacune des capacités proposées.

firewalld Un module a également été développé pour un second pare-feu, firewalld. Tout comme le module iptables, il fournit des capacités de contrôle d'accès réseau pour les ports, les interfaces, etc.

Droits d'accès DAC Unix Un module d'extension permet de gérer les droits d'accès Unix, c'est-à-dire les droits discrétionnaires de lecture, écriture et exécution sur les fichiers. Ce module fournit ainsi des capacités de contrôle d'accès discrétionnaire pour des ressources systèmes. Il permet de modifier les droits d'accès aux fichiers selon les capacités demandées.

SELinux SELinux est également associé à un module du SE^E. Tout comme dans le cas des droits Unix, le module d'extension SELinux permet d'appliquer des capacités de contrôle d'accès sur des ressources systèmes. Ce module d'extension génère un nouveau module SELinux (ou met à jour un module existant si approprié) pour un groupe de capacités s'appliquant sur des ressources d'un même domaine.

OpenVPN Un module d'extension OpenVPN est disponible pour mettre en place des tunnels VPN entre des nœuds. Il s'agit donc de chiffrer les communications réseaux entre des machines. Le module OpenVPN est utilisé dans des propriétés réseaux, autrement dit ses capacités font intervenir des phases de négociation entre plusieurs SE^E afin de configurer un tunnel compatible (en utilisant des mécanismes compatibles sur les différents nœuds). Sur chacun des nœuds impliqués, le module génère des fichiers de configuration et démarre le VPN. Ce module fait appel à des capacités d'ouverture de ports : il nécessite donc la présence d'un module pour un pare-feu.

PAM Le module PAM est utilisé pour fournir des capacités liées à l'authentification d'un utilisateur sur le système. Il s'agit donc de détecter les authentifications qui sont faites sur le système, mais également de les contrôler afin de restreindre les accès aux seuls utilisateurs

autorisés. Ce module modifie les fichiers de configuration de PAM afin de refléter les besoins énoncés dans la politique de sécurité.

JCE Un module JCE (*Java Cryptography Extension*) a été développé pour fournir des capacités de chiffrement. Ce module est capable de générer des clefs de chiffrement et de les utiliser pour chiffrer ou déchiffrer des fichiers.

Oscap Oscap est utilisé pour appliquer des capacités d'assurance. Le module associé interprète des fichiers décrivant les tests à effectuer et traite les résultats obtenus. Une capacité faisant appel à ce module sera donc appliquée en configurant Oscap afin que les tests d'assurance soient effectués.

AssuranceExec Le module AssuranceExec permet d'exécuter des scripts d'assurance générés pour les mécanismes ou les propriétés. Tout comme le module Oscap, il traite les résultats ainsi obtenus afin de pouvoir réagir à un éventuel dysfonctionnement, par exemple en réappliquant les propriétés.

nmap Un module d'extension nmap est également disponible afin de vérifier que les capacités d'ouverture de ports ont été correctement appliquées. Ce module nécessite des communications entre les SE^E : le SE^E sur lequel la propriété est appliquée demande à un SE^E distant d'effectuer les tests de connexion.

Web Access Un module d'accès Web a été implémenté pour gérer les droits d'accès aux pages Web d'un site. Il peut donc éditer les fichiers *.htaccess* afin de contrôler les accès.

su Un module utilisant la commande `su` a été implémenté. Il permet de vérifier que les connexions locales sont autorisées ou interdites pour les utilisateurs, en accord avec les propriétés d'authentification.

5.2.4.2 Autres modules d'extension

Les quatre modules d'extension présentés dans cette section sont des contributions externes, notamment de partenaires du projet européen Seed4C. Ils ont été développés à l'aide de la documentation fournie détaillant l'intégration de nouveaux modules. Ces modules ne font donc pas partie des contributions de cette thèse et ils sont présentés uniquement à titre informatif.

SSH Vicomtech [1] a proposé un module permettant d'établir des tunnels SSH entre plusieurs machines en utilisant les fonctionnalités de redirection de port fournies par SSH. Ce module est donc également utilisé dans le cadre de propriétés de confidentialité réseau afin de sécuriser les communications entre les nœuds. Le module propose deux capacités qui mettent en place un tunnel de manière soit locale, soit distante.

DPM Alcatel-Lucent Bell-Labs [2] a fourni un module pour utiliser le DPM (*Data Protection Module*), un outil développé par Alcatel-Lucent et ayant pour but de sécuriser les données personnelles. Le DPM permet à un utilisateur de définir une politique d'accès à

1. http://www.vicomtech.es/
2. http://www.alcatel-lucent.com/wps/portal/belllabs

un fichier, politique qui est ensuite attachée à ce fichier. Le fichier est donc stocké chiffré et il est déchiffré de manière transparente pour l'utilisateur lors d'un accès autorisé. Une fois le DPM activé, il protège automatiquement l'ensemble des fichiers pour lesquels une politique a été définie. Par conséquent, le module d'extension DPM ne fournit qu'une seule capacité qui permet uniquement d'activer l'utilisation du DPM.

SE Gemalto [3] a développé un module permettant d'utiliser les fonctionnalités d'un *Secure Element* (SE) tel que définit dans *Global Platform*. Un SE est donc un composant matériel permettant de générer des clefs de chiffrement, mais également de les stocker et de les gérer (plusieurs utilisateurs peuvent stocker leurs clefs au sein d'un même SE). Le SE fournit également des fonctions permettant par exemple de chiffrer et déchiffrer des fichiers en utilisant les clefs qu'il stocke. Le module associé fournit ainsi des capacités liées à des fonctions cryptographiques.

md5deep Un module d'extension md5deep a été développé par des étudiants de l'Insa Centre Val de Loire [4] dans le cadre d'un projet. Il permet de générer l'empreinte MD5 de fichiers protégés par une propriété d'intégrité. Cette intégrité peut alors être vérifiée par le module lors de la phase d'assurance de la propriété.

5.2.4.3 Synthèse

Chacun des modules d'extension développés fournit un ensemble de capacités. La table 5.1 présente le nombre de capacités que ces modules peuvent appliquer. Une table complète des modules d'extension par capacité est disponible à l'annexe A.1.

Mécanismes	Capacités			Taille du module (nb. lignes)
	Nombre	Catégorie	Classe	
iptables	7	Contrôle d'accès	Réseau	466
firewalld	7	Contrôle d'accès	Réseau	234
Droits DAC Unix	8	Contrôle d'accès	Système	316
SELinux	10	Contrôle d'accès	Système	926
OpenVPN	1	Chiffrement	Réseau	655
PAM	3	Authentification	Système	332
Web Access	1	Authentification	Système	135
JCE	3	Chiffrement	Autre	291
Oscap	2	Assurance	Autre	134
AssuranceExec	2	Assurance	Autre	168
SSH	3	Chiffrement	Réseau	378
DPM	1	Contrôle d'accès	Système	107
SE	3	Chiffrement	Autre	321
su	1	Authentification	Système	110
nmap	1	Accès	Réseau	137
md5deep	1	Intégrité	Système	168

TABLE 5.1 – Nombre de capacités par mécanismes

On peut noter que l'intégration d'un mécanisme ne demande que peu de lignes de code, notamment grâce au mode de fonctionnement proposé à la section 5.2.3 et à l'API fournie. Des exemples de compatibilité et de dépendance entre ces différents mécanismes sont présentés au chapitre 6.

3. http://www.gemalto.com/
4. http://www.insa-centrevaldeloire.fr/

5.3 Exemple de déploiement d'une politique de sécurité

Cette section synthétise le fonctionnement global du SE^E, la définition d'une politique de sécurité, l'application et l'assurance de cette politique, et les scores obtenus. Trois propriétés de confidentialité système, réseau et hybride, nommées respectivement P_S, P_R et P_H sont considérées pour illustrer cet exemple. Cette section permet ainsi d'illustrer le langage d'expression des besoins de sécurité (chapitre 3) et les algorithmes d'application utilisés par le SE^E (chapitre 4).

5.3.1 Environnement considéré

L'architecture de l'exemple développé dans cette section est présentée à la figure 5.4. Cet exemple impliquant une propriété réseau, nous considérons deux nœuds, client et serveur. Le nœud client permet à un administrateur mail de s'authentifier et d'accéder au serveur mail. Le nœud serveur contient un serveur mail : nous considérons uniquement les fichiers de configuration de ce serveur mail et l'administrateur mail. La propriété P_S empêche la lecture du fichier de configuration mail par des utilisateurs autres que l'administrateur mail, P_R correspond à la confidentialité des communications entre le client et le serveur, et P_H combine P_S et P_R en une propriété hybride.

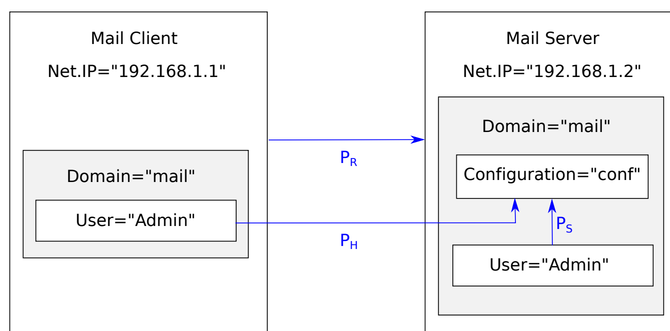

FIGURE 5.4 – Architecture de l'exemple

L'environnement considéré pour le client et le serveur est décrit dans le tableau 5.2 : il contient la liste des mécanismes disponibles sur chaque nœud et le mode de sélection choisi pour chacune des propriétés.

Cas	**Mécanismes disponibles**		**Mode de sélection**
	Client	**Serveur**	
Cas 1	SELinux DAC JCE SSH -	SELinux DAC JCE SSH OpenVPN	$P_S \rightarrow$ `Défaut` $P_R \rightarrow$ `Défaut` $P_H \rightarrow$ `Défaut`
Cas 2	SELinux DAC JCE SSH OpenVPN	SELinux DAC JCE SSH OpenVPN	$P_S \rightarrow$ `Maximum` $P_R \rightarrow$ `Défaut` $P_H \rightarrow$ `Défaut`

TABLE 5.2 – Description de l'environnement

Deux cas d'application de la politique sont considérés, *cas 1* et *cas 2*. Le *cas 1* applique toutes les propriétés en utilisant le mode `Défaut` : le SE^E cherche donc à maximiser le score d'application des propriétés en utilisant un seul mécanisme par capacité. De plus, dans le *cas 1*, le nœud client ne dispose pas d'OpenVPN, au contraire du nœud serveur. Le *cas 2* se différencie du *cas 1* sur deux points. Tout d'abord, la propriété de confidentialité système P_S est appliquée suivant le mode de sélection `Maximum` : le SE^E sélectionne donc le plus de mécanismes possibles pour appliquer chacune des capacités. De plus, le nœud client dispose d'OpenVPN qui peut donc être utilisé pour la propriété de confidentialité réseau P_R.

Le listing 5.6 déclare les noms des classes, propriétés et capacités qui sont utilisées dans cet exemple. Il s'agit donc d'un extrait de la politique de configuration, spécifique à cet exemple. La classe `Class1` regroupe l'ensemble des propriétés de confidentialité (lignes 3 à 6) et la définition complète des propriétés est spécifiée à l'annexe A.2. Puis, ce listing donne la liste des capacités disponibles pour cet exemple.

```
1  Class1 := @Confidentiality;
2
3  P1 := Confidentiality_Access_Control (Type.Passive.Data.File, Id.User);
4  P2 := Confidentiality_Encryption (Type.Passive.Data.File, Id.User);
5  P3 := Confidentiality (Net.IP, Net.IP);
6  P4 := Confidentiality (Type.Passive.Data.File, Net.IP, Net.IP, ID.User);
7
8  C1 := deny_all_read_accesses (Type.Passive.Data.File);
9  C2 := allow_read_access (Type.Passive.Data.File, Id.User);
10 C3 := generate_key (Type.Passive.Data.Key.Symmetric);
11 C4 := generate_key (Type.Passive.Data.Key.Asymmetric);
12 C5 := encrypt_flow (Net.IP, Net.IP, Type.Passive.Data.Key.Symmetric);
13 C6 := encrypt_flow (Net.IP, Net.IP, Type.Passive.Data.Key.Asymmetric);
```

Listing 5.6 – Configuration du SE^E

Le listing 5.7 contient la configuration interne du SE^E : les scores statiques des propriétés (lignes 2-3), la configuration des mécanismes de contrôle d'accès (scores d'ordonnancement des mécanismes pour les capacités, compatibilité et dépendance des mécanismes, lignes 6-8) et de chiffrement (scores d'ordonnancement des mécanismes pour les capacités et compatibilité réseau, lignes 11-14). Ces scores sont identiques sur les nœuds client et serveur, et dans les deux cas d'application considérés.

```
1  // Score statique des proprietes
2  Sstat(P1) = 2             Sstat(P2) = 1
3  Sstat(P3) = 1             Sstat(P4) = 1
4
5  // Configuration des mecanismes de controle d'acces
6  S(C1, SELinux) = 6        S(C2, SELinux) = 6       R(SELinux) = 1
7  S(C1, DAC) = 4            S(C2, DAC) = 3           R(DAC) = 1
8  Comp(SELinux, DAC) = 1    Dep(SELinux, DAC) = 0
9
10 // Configuration des mecanismes de chiffrement
11 S(C3, JCE) = 5            S(C3, GPG) = 4
12 S(C4, JCE) = 6            S(C4, GPG) = 4
13 S(C5, OpenVPN) = 6        S(C6, SSH) = 4
14 CompNet(OpenVPN, SSH) = 0
```

Listing 5.7 – Configuration du système

5.3.2 Compilation des propriétés

Dans cette section, nous détaillons l'étape de compilation de la phase d'application des trois propriétés système, réseau et hybride. Les scores d'ordonnancement et d'application permettant de déterminer quels mécanismes doivent être utilisés sont également présentés.

5.3.2.1 Propriété système

La propriété système P_S est une propriété de confidentialité sur des fichiers. Le listing 5.8 présente l'extrait de la politique de sécurité correspondant à cette propriété.

```
1 mailConfig := (Type.Passive.Data.File.Configuration="conf"):(Domain="mail");
2 mailAdmin := (Id.User="admin"):(Domain="mail");
3
4 @Confidentiality (mailConfig, mailAdmin);
```

Listing 5.8 – Propriété de confidentialité système

Le moteur d'application reçoit cette propriété instanciée et doit déterminer quelle est la meilleure application possible. La propriété fait référence à la classe **`@Confidentiality`**. En prenant en compte les types des contextes passés en paramètres, on peut remarquer que deux propriétés peuvent être utilisées : P_1 et P_2. D'après les définitions des scores statiques des propriétés (listing 5.7), P_1 a un score plus élevé que P_2. Les types de contextes attendus par les propriétés étant identiques, les scores d'inclusion de P_1 et P_2 dans P_S sont les mêmes (score d'inclusion = 3). On obtient donc les scores d'ordonnancement suivants :

$$Sord(P_1) = \frac{Sstat(P_1)}{d(P_1, P_S) + 1} = \frac{2}{3+1} = 0,5$$

$$Sord(P_2) = \frac{Sstat(P_2)}{d(P_2, P_S) + 1} = \frac{1}{3+1} = 0,25$$

P_1 a un meilleur score d'ordonnancement et sera donc appliquée (s'il y a des mécanismes disponibles pour chacune des capacités de P_1).

Le moteur de projection cherche donc à appliquer la propriété P_1 sur les contextes `mailConfig` et `mailAdmin`. Les capacités instanciées de P_1 peuvent être appliquées par les capacités : C_1 et C_2. Le moteur de projection cherche à appliquer C_1. Les deux mécanismes disponibles sur le système et capables d'appliquer C_1 sont SELinux et les droits DAC Unix. SElinux ayant un meilleur score d'ordonnancement pour C_1, il est sélectionné. La capacité C_2 peut être appliquée par les mêmes mécanismes. SELinux étant un mécanisme dominant, il est nécessairement utilisé pour C_2 puisqu'il a été sélectionné pour C_1. Le score d'application de P_S obtenu est donc :

$$Sapp(P_S) = Sord(P_1) * [Sapp(C_1) + Sapp(C_2)]$$

$$\Leftrightarrow Sapp(P_S) = Sord(P_1) * [S(C_1, SELinux) * \frac{1}{d(C_1, C) + 1} + S(C_2, SELinux) * \frac{1}{d(C_2, C) + 1}]$$

$$\Leftrightarrow Sapp(P_S) = 0,5 * [6 * \frac{1}{0+1} + 6 * \frac{1}{0+1}]$$

$$\Leftrightarrow Sapp(P_S) = 6$$

Dans le *cas 2*, c'est-à-dire si le mode `Maximum` est choisi pour cette propriété, le moteur de projection choisira le plus de mécanismes possibles. Dans ce cas, SELinux et les

droits DAC seront tous deux utilisés puisqu'ils sont compatibles. On obtient alors le score d'application suivant :

$$Sapp(P_S) = Sord(P_1) * [Sapp(C_1) + Sapp(C_2)]$$

$$\Leftrightarrow Sapp(P_S) = Sord(P_1)*[S(C_1, SELinux)+S(C_1, DAC)+S(C_2, SELinux)+S(C_1, DAC)]$$

$$\Leftrightarrow Sapp(P_S) = 0,5 * [(6+4) * \frac{1}{0+1} + (6+3) * \frac{1}{0+1}]$$

$$\Leftrightarrow Sapp(P_S) = 9,5$$

On obtient donc un score d'application plus élevé lorsque les droits DAC Unix sont utilisés en complément de SELinux, ce qui indique une meilleure application au sens de l'application idéale définie par l'expert en sécurité.

5.3.2.2 Propriété réseau

La seconde propriété P_R est une propriété de confidentialité réseau (voir listing 5.9).

```
1 mailServer := (Net.IP="192.168.1.2");
2 mailClient := (Net.IP="192.168.1.1");
3
4 @Confidentiality (mailServer, mailClient);
```

Listing 5.9 – Propriété de confidentialité réseau

La propriété est une nouvelle fois appelée grâce à la classe **@Confidentiality**. Une seule propriété de cette classe correspond aux arguments de la propriété instanciée : elle est donc sélectionnée.

Sur le nœud serveur, deux capacités instanciées doivent être appliquées : **generate_key** (C_3 ou C_4) et **encrypt_flow** (C_5 ou C_6).

Les deux mécanismes disponibles pour C_3 et C_4 sont JCE et GPG. Le score d'ordonnancement attribué à JCE étant plus élevé pour les deux capacités, c'est le mécanisme choisi. De plus, C_4 a un meilleur score que C_3 et sera donc utilisé (si ce choix est compatible avec les autres capacités de P_R).

Sur le serveur, deux capacités peuvent appliquer la capacité **encrypt_flow** : C_5 ou C_6. Pour chacune de ces capacités, un mécanisme est disponible : OpenVPN ou SSH. OpenVPN ayant un meilleur score d'ordonnancement, il sera préféré à SSH. Cependant, **encrypt_flow** est une propriété réseau qui est également appliquée sur le nœud client. Ce dernier ne dispose pas d'OpenVPN : il ne peut donc utiliser que C_6 avec SSH. Comme OpenVPN et SSH ne sont pas compatibles au sens de la matrice $CompNet$ (listing 5.7), SSH est sélectionné sur les deux nœuds pour appliquer C_6.

On obtient donc les scores d'application suivants pour la propriété P_R sur les nœuds serveur et client.

$$\begin{cases} Sapp(P_{R_server}) & = & Sord(P_3) * [S(C_4, JCE) + S(C_6, SSH)] \\ Sapp(P_{R_client}) & = & Sord(P_3) * [S(C_6, SSH)] \end{cases}$$

$$\Leftrightarrow \begin{cases} Sapp(P_{R_server}) & = & 1 * [6+4] & = & 10 \\ Sapp(P_{R_client}) & = & 4 \end{cases}$$

Dans le *cas 2*, c'est-à-dire si le nœud client dispose également de OpenVPN, les capacités utilisées seront C_3 et C_5. Les scores d'application obtenus sont les suivants :

$$\begin{cases} Sapp(P_{R_server}) & = & Sord(P_3) * [S(C_3, JCE) + S(C_5, OpenVPN)] \\ Sapp(P_{R_client}) & = & Sord(P_3) * [S(C_5, OpenVPN)] \end{cases}$$

$$\Leftrightarrow \begin{cases} Sapp(P_{R_server}) &= 1 * [5+6] = 11 \\ Sapp(P_{R_client}) &= 6 \end{cases}$$

Ainsi, nous pouvons remarquer que les compatibilités réseaux entre les mécanismes influent sur le score d'application de la politique sur l'ensemble des nœuds considérés.

5.3.2.3 Propriété hybride

Enfin, nous présentons dans cette section l'application d'une propriété de confidentialité hybride P_H, faisant donc intervenir des parties systèmes et des parties réseaux.

```
1 mailConfig := (Type.Passive.Data.File.Configuration="conf"):(Domain="mail");
2 mailAdmin  := (Id.User="admin"):(Domain="mail");
3 mailServer := (Net.IP="192.168.1.2");
4 mailClient := (Net.IP="192.168.1.1");
5
6 @Confidentiality (mailConfig, mailServer, mailClient, mailAdmin);
```

Listing 5.10 – Propriété de confidentialité hybride

La seule propriété correspondant à cette instance est P_4, qui fait appel aux propriétés P_1 et P_3 dont les applications ont été détaillées précédemment.

Le tableau 5.3 présente les configurations qui sont obtenues pour les deux cas considérés (cas 1 : mode de sélection `Défaut` et pas de VPN sur le client, cas 2 : mode de sélection `Maximum` pour la partie système et VPN présent sur les deux nœuds).

Nœud	**Propriétés**	**Capacités et Mécanismes**	
		Solution 1	**Solution 2**
Serveur	P_1	$C_1 \rightarrow$ SELinux $C_2 \rightarrow$ SELinux	$C_1 \rightarrow$ SELinux+DAC $C_2 \rightarrow$ SELinux+DAC
	P_3	$C_4 \rightarrow$ JCE $C_6 \rightarrow$ SSH	$C_3 \rightarrow$ JCE $C_5 \rightarrow$ OpenVPN
Client	P_1	$C_1 \rightarrow$ SELinux $C_2 \rightarrow$ SELinux	$C_1 \rightarrow$ SELinux+DAC $C_2 \rightarrow$ SELinux+DAC
	P_3	$C_6 \rightarrow$ SSH	$C_5 \rightarrow$ OpenVPN

TABLE 5.3 – Application de la propriété hybride P_H

Les scores d'application résultant des deux solutions envisagées sont les suivants :

$$\text{Cas 1 :} \begin{cases} Sapp(P_{H_server_1}) &= 6+10 &= 16 \\ Sapp(P_{H_client_1}) &= 6+4 &= 10 \end{cases}$$

$$\text{Cas 2 :} \begin{cases} Sapp(P_{H_server_2}) &= 9.5+11 &= 20,5 \\ Sapp(P_{H_client_2}) &= 9.5+6 &= 15,5 \end{cases}$$

Les scores d'application pour la deuxième solution sont donc plus élevés, ce qui indique que la politique a été appliquée avec des mécanismes plus adaptés (au sens de la définition par l'expert en sécurité) et plus nombreux (mode `Maximum` pour la propriété P_1 de contrôle d'accès). La qualité de l'application est donc plus élevée dans le second cas.

Cependant, le mode de sélection `Maximum` possède aussi des désavantages. Tout d'abord, le mode `Maximum` implique l'utilisation d'un plus grand nombre de mécanismes que les autres modes : le temps de déploiement de la politique est donc plus important. De plus, plus le nombre de mécanismes utilisés est élevé, plus les performances du système sont

impactées. Enfin, l'utilisation de plusieurs mécanismes peut présenter des risques (cas du surchiffrement). C'est pourquoi, dans le cas classique, la politique est appliquée en utilisant le mode `Défaut`. Cela reste cependant configurable par l'expert en sécurité.

5.3.3 Projection des propriétés

Cette section décrit la configuration des mécanismes qui est faite à la suite de l'étape de compilation. On considère uniquement l'application de la propriété système P_S sur le nœud serveur et de la propriété réseau P_R, puisque la propriété hybride est une combinaison de P_S et P_R.

5.3.3.1 Propriété Système

La propriété P_S peut être appliquée soit avec SELinux (cas 1), soit avec DAC et SELinux (cas 2). La configuration de ces deux mécanismes pour P_S est décrite ici.

DAC Afin d'appliquer P_S, le module d'extension DAC crée un groupe correspondant à l'utilisateur (ou à l'ensemble des utilisateurs) autorisés. Puis, le module autorise la lecture du fichier confidentiel pour les utilisateurs appartenant à ce groupe et enlève les droits pour les autres. Le module génère et exécute le script du listing 5.11. Ce script est généré à partir d'un squelette de configuration (section 5.2.3) : les chemins des exécutables utilisés (lignes 4, 6, 9-11) dépendent de la distribution considérée et sont déterminés par le SE^E lors de la génération du script.

```
1  #!/bin/bash
2  USERS=[...] # Liste des utilisateurs autorises, generee par le SEE
3  FILES=[...] # Liste des fichiers confidentiels, generee par le SEE
4  /usr/sbin/groupadd mailAdmin
5  for user in "${USERS[@]}" ; do
6    /usr/sbin/usermod -aG mailAdmin user
7  done
8  for file in "${FILES[@]}" ; do
9    /bin/chown :mailAdmin file
10   /bin/chmod g+r file
11   /bin/chmod o-rx file
12 done
```

Listing 5.11 – Configuration des droits d'accès DAC Unix

Ce script récupère auprès du SE^E la liste des utilisateurs et des fichiers concernés par la propriété (lignes 2 et 3). Puis, il crée un nouveau groupe (ligne 4) et y ajoute les utilisateurs (lignes 5 à 7). Enfin, les droits de lecture sont donnés aux membres du groupe (lignes 9 et 10) et supprimés pour les autres utilisateurs (ligne 11). Les utilisateurs qui ne sont pas membres de ce groupe ne pourront donc pas lire les fichiers de configuration mail protégés. Notons que les droits du propriétaire ne sont pas impactés, ce qui explique que ce module ait un score moins élevé que le module d'extension SELinux.

SELinux Pour appliquer P_S, le module d'extension SELinux génère un nouveau module de politique SELinux (dans le langage dédié à l'expression des politiques SELinux). Le listing 5.12 présente la création du module `mail` (ligne 1). Un nouveau domaine est tout d'abord créé (ligne 3), puis un type est créé pour les fichiers de configuration mails (ligne 4). Enfin, l'administrateur mail est autorisé à lire les fichiers de configuration (ligne 5).

```
policy_module(mail,1.0.0)
# Create a new domain for the mail application
see_create_service_domain(mail)
see_create_files_type(mail_conf_t)
see_files_type_read(admin_t,mail_conf_t)
```

Listing 5.12 – Création d'un nouveau module SELinux

Le module d'extension SELinux utilise ici trois fonctions. Ces fonctions ont été implémentées sous forme de macros SELinux spécifiques au SE^E afin de fournir une API facilitant la génération du module SELinux pour appliquer les propriétés. Un extrait de la fonction `see_create_service_domain` est présenté au listing 5.13 et permet la création d'un domaine applicatif pour le serveur mail.

```
template(`see_create_service_domain',`
    type $1_t;
    type $1_exec_t;

    application_domain($1_t,$1_exec_t)
    [...]
')
```

Listing 5.13 – Création d'un nouveau domaine SELinux

Le listing 5.14 détaille l'une des fonctions utilisées pour appliquer la propriété P_S : il s'agit de la fonction donnant les droits de lecture sur les fichiers de configuration mail à l'administrateur mail. Elle attend deux arguments (ligne 2) puis donne les droits de lecture (lignes 3-5) pour le premier argument sur le second.

```
interface(`see_files_type_read',`
    gen_require(` type $1; type $2; ')
    allow $1 $2:file { read_file_perms };
    allow $1 $2:dir { search read };
    allow $1 $2:lnk_file { read_lnk_file_perms };
')
```

Listing 5.14 – Ajout des droits de lecture sur un fichier

La création d'un domaine SELinux pour le serveur mail permet ainsi d'appliquer la propriété P_S en générant un module SELinux spécifique.

5.3.3.2 Propriété Réseau

Selon le cas considéré, la propriété P_R peut être appliquée par SSH ou par OpenVPN. Dans les deux cas, des clefs de chiffrement sont générées par JCE.

JCE Selon le mécanisme utilisé pour le chiffrement du flux, la clef générée par JCE sera soit une paire de clefs asymétriques (SSH), soit une clef symétrique (OpenVPN, ici en mode symétrique). Le listing 5.15 contient un exemple de génération d'une clef symétrique par JCE. Tout d'abord, l'algorithme de chiffrement (AES, ligne 1) et la taille de la clef (128 bits, ligne 2) sont choisis. Puis, la clef est générée (ligne 3).

```
KeyGenerator keyGen = KeyGenerator.getInstance("AES");
keyGen.init(128);
SecretKey secretKey = keyGen.generateKey();
```

Listing 5.15 – Génération d'une clef symétrique avec JCE

Le listing 5.16 contient un extrait de code pour générer une paire de clefs asymétriques pour l'algorithme RSA (ligne 1) de 2048 bits (ligne 2). Puis, la paire de clefs est générée (ligne 3) et les clefs privée et publique sont obtenues (lignes 4 et 5).

```
1 KeyPairGenerator kpg = KeyPairGenerator.getInstance("RSA");
2 kpg.initialize(2048, new SecureRandom());
3 KeyPair RSAkey = kpg.generateKeyPair();
4 RSAPrivateKey prvKey = (RSAPrivateKey)RSAkey.getPrivate();
5 RSAPublicKey pubKey = (RSAPublicKey)RSAkey.getPublic();
```

Listing 5.16 – Génération d'une clef asymétrique avec JCE

Les deux types de clefs peuvent donc être générés par le module d'extension JCE afin d'être utilisés par les mécanismes de chiffrement réseau. Ces clefs peuvent ensuite être exportées (par exemple dans un fichier) afin d'être transmises aux autres capacités.

SSH Pour établir un tunnel SSH, le module utilise la librairie *JSch*. Le listing 5.17 est un exemple de code Java utilisant *JSch* pour établir un tunnel SSH.

```
1 JSch jsch = new JSch();
2 jsch.addIdentity(keypath);
3 Session session = jsch.getSession(user, server, sshPort);
4 UserInfo ui = new UserInfo() {
5   [...]
6 }
7 session.setUserInfo(ui);
8 session.connect();
9 session.setPortForwardingL(lport, "0.0.0.0", rport);
```

Listing 5.17 – Configuration d'un tunnel SSH avec JSch

JSch utilise une clef (ligne 2, générée par JCE) pour établir un tunnel par redirection de ports (*port forwarding*) écoutant sur le port `sshPort` (ligne 3) sur le serveur. Cela correspond à la commande `ssh -L` d'OpenSSH. Les communications réseaux entre ce serveur et un client se connectant par SSH sont donc chiffrées, ce qui correspond à la propriété de confidentialité réseau à appliquer.

OpenVPN Dans le cas considéré, le SE^E configure un VPN symétrique. Les fichiers de configuration générés par le SE^E sur le client et le serveur sont basés sur les squelettes donnés aux listings 5.18 et 5.19.

Le listing 5.18 permet de configurer un serveur OpenVPN écoutant sur le port `port` (ouvert grâce à la capacité **`open_port`** d'un module de pare-feu) et utilisant la clef de chiffrement `keyPath` (générée par la capacité **`generate_key`** du module JCE).

```
1 #OpenVPN Config for VPN server
2 local ${localIP}
3 port ${port}
4 dev tun${id}
5 ifconfig ${serverTunIP} ${clientTunIP}
6 secret ${keyPath}
7
8 comp-lzo
9 keepalive 10 60
10 ping-timer-rem
11
12 persist-tun
13 persist-key
14
```

```
15 daemon
```

Listing 5.18 – Configuration serveur VPN

Le listing 5.19 permet de configurer un client OpenVPN pour établir un tunnel VPN vers le serveur.

```
1  #OpenVPN Config for VPN client
2  dev tun${id}
3  remote ${serverIP} ${port}
4  ifconfig ${clientTunIP} ${serverTunIP}
5  secret ${keyPath}
6
7  comp-lzo
8  keepalive 10 60
9  ping-timer-rem
10
11 persist-tun
12 persist-key
13
14 daemon
```

Listing 5.19 – Configuration client VPN

On obtient ainsi un tunnel OpenVPN entre le client et le serveur qui permet de garantir la confidentialité de leurs échanges. Cette configuration est issue à la fois de la collaboration entre deux SE^E (client et serveur) et trois modules d'extension (OpenVPN, JCE et un pare-feu).

5.3.4 Assurance des propriétés

Cette section décrit la génération des tests d'assurance pour les trois propriétés de confidentialité appliquées à la section précédente. Cette génération est faite à partir du bloc d'assurance de la propriété, en utilisant les capacités d'assurance pouvant être appliquées par des mécanismes. Tout comme pour l'application des propriétés, on ne présente ici que l'assurance pour P_S et P_R (l'assurance pour P_H peut en être déduite).

5.3.4.1 Propriété Système

Le bloc `assurance` de la propriété P_1 (annexe A.2) fait intervenir quatre capacités. Les trois capacités **`get_users`**, **`get_files`** et **`get_all_users`** sont des capacités internes et seront donc appliquées par le SE^E (il n'est pas nécessaire de sélectionner un module d'extension). La capacité **`check_read`** peut quant à elle être appliquée par une commande système. Ces différentes capacités sont utilisées pour générer des scripts d'assurance permettant de vérifier la bonne application de la propriété. Ainsi, le script du listing 5.20 permet de vérifier la bonne application de P_S.

```
1  #!/bin/bash
2  RET=$RESULT_PASS
3  check_read(){su -c "test -r "$1"" $2; return $?;}
4  FILES=[...]     # liste des fichiers confidentiels
5  USERS=[...]     # liste de tous les utilisateurs
6  OK_USERS=[...]  # liste des utilisateurs autorises a lire les fichiers
7
8  for file in "${FILES[@]}" ; do
9   for user in "${USERS[@]}" ; do
10   check_read $file $user
11   READ_OK=$?
```

```
12
13   if [[ " ${OK_USERS[@]} " =~ " $user " ]] ; then
14    if [[ $READ_OK -ne "0" ]] ; then
15     RET=$RESULT_FAIL
16     echo "Unexpected access denial: $user->$file"
17    fi
18   else
19    if  [[ $READ_OK -eq "0" ]] ; then
20     RET=$RESULT_FAIL
21     echo "Unauthorized access: $user->$file"
22    fi
23   fi
24  done
25 done
26 exit $RET
```

Listing 5.20 – Script d'assurance pour la confidentialité système

La fonction *check_read* (ligne 3) est générée par le module d'assurance et associée à la commande *test*. Les variables *FILES*, *USERS* et *OK_USERS* (lignes 4 à 6) sont définies par des capacités internes du SE^E afin de contenir les ressources associées aux fichiers et utilisateurs concernés par P_S. Le script itère alors sur les fichiers (lignes 8 à 25) et les utilisateurs (lignes 9 à 24) afin de déterminer si P_S est respectée. Si un utilisateur autorisé (ligne 13) ne parvient pas à lire le fichier (ligne 14), ou inversement (lignes 18-19), alors une erreur est remontée au moteur d'assurance (lignes 15 et 20).

5.3.4.2 Propriété Réseau

Afin de vérifier que la propriété P_R a été correctement appliquée, il est possible d'observer les communications réseaux chiffrées. Pour cela, la capacité **`check_encrypted_flow`** peut utiliser la commande `tcpdump` du listing 5.21.

```
1 tcpdump -i eth0 'tcp and (((ip[2:2] - ((ip[0]&0xf)<<2)) - ((tcp[12]&0xf0)>>2))
    != 0) and (tcp[((tcp[12:1] & 0xf0) >> 2)+5:1] = 0x01) and (tcp[((tcp[12:1]
    & 0xf0) >> 2):1] = 0x16)'
```

Listing 5.21 – Commande utilisée par le script d'assurance pour la confidentialité réseau

Ainsi, on utilise le filtrage avancé [Wains, 2013] de *tcpdump* sur l'interface utilisée pour établir le VPN (ici, `eth0`). On filtre ensuite le trafic `tcp` en utilisant les champs suivants du paquet : `ip[2:2]` est la longueur totale du paquet IP, `(ip[0]&0xf)`«2 est la longueur de l'entête IP, et `(tcp[12]&0xf0)`»2 est celle de l'entête TCP. Si la longueur obtenue en soustrayant les entêtes à la taille totale n'est pas nulle, le paquet contient des données. On cherche alors à déterminer s'il s'agit d'une connexion SSL en recherchant le paquet *SSL Handshake Hello*. Pour cela, il faut que le premier octet des données (`tcp[((tcp[12:1] & 0xf0) » 2):1]`) ait pour valeur `0x16` (`Handshake`) et que le sixième octet des données (`tcp[((tcp[12:1]&0xf0) » 2)+5:1]`) ait pour valeur `0x01` (`Client Hello`).

On peut donc ainsi récupérer les connexions SSL entre le client et le serveur et donc déterminer si les communications sont bien chiffrées.

5.3.5 Assurance des mécanismes

L'assurance des mécanismes a pour but de vérifier que les mécanismes ayant été utilisés sont bien fonctionnels, et éventuellement que leur configuration est bien celle attendue. Cette partie de l'assurance diffère de l'assurance des propriétés puisque qu'elle dépend de l'application de celles-ci. Ces deux types d'assurance sont donc complémentaires.

5.3.5.1 Mécanismes Système

Dans le cas où P_S est appliquée avec SELinux, il est nécessaire de vérifier le mode d'exécution de SELinux. Le listing 5.22 présente le script d'assurance de SELinux, généré par son module d'extension.

```
1 #!/bin/bash
2 result="$(/usr/sbin/getenforce | /bin/grep Enforcing | /usr/bin/wc -l)"
3 if [ $result = 1 ]
4  then exit $RESULT_PASS
5  else exit $RESULT_FAIL
6 fi
```

Listing 5.22 – Script d'assurance pour SELinux

Le script détecte le mode d'exécution de SELinux grâce à la commande *getenforce* (ligne 2). Si SELinux est en mode *Enforcing*, l'assurance réussit (lignes 3 et 4). Dans le cas contraire (mode *Permissif*), le script renvoie une erreur (ligne 5).

5.3.5.2 Mécanismes Réseau

Considérons maintenant le cas où P_R est appliquée en utilisant OpenVPN. Le statut d'OpenVPN peut être vérifié en utilisant le script généré par le module d'extension OpenVPN (listing 5.23)

```
1 #!/bin/bash
2 RET="$(/bin/ps aux | /bin/grep /usr/sbin/openvpn | /usr/bin/wc -l)"
3 if [ "$RET" -lt "2" ]
4  then exit $RESULT_FAIL
5  else exit $RESULT_PASS
6 fi
```

Listing 5.23 – Script d'assurance pour OpenVPN

Le script recherche les processus *openvpn* s'exécutant sur le système (ligne 2). Si aucun processus n'est présent, le script retourne une erreur (lignes 3 et 4), sinon il réussit (ligne 5). Notons que ce test permet de vérifier que le serveur OpenVPN est lancé mais pas que le tunnel, et donc la propriété P_R, fonctionne. C'est pourquoi les tests d'assurance pour les propriétés qui ont été présentés précédemment sont également nécessaires.

5.3.6 Traitement des informations d'assurance

Les résultats des scripts d'assurance des propriétés et des mécanismes sont envoyés au moteur d'assurance afin d'être traités. Par exemple, si le script d'assurance lié à SELinux détecte un dysfonctionnement de SELinux, le moteur d'assurance doit réagir en conséquence. La réaction se décompose en plusieurs étapes.

Tout d'abord, le moteur d'assurance sélectionne les propriétés ayant été appliquées en utilisant, au moins partiellement, SELinux. Ici, on obtient donc les propriétés P_S et P_H. Le moteur d'assurance demande alors l'arrêt de ces propriétés, c'est-à-dire l'arrêt de chacune de leurs capacités. Cet arrêt est fait par les modules d'extension ayant appliqué les propriétés et permet de restaurer l'état du système.

Puis, le moteur d'assurance stoppe le module et le désactive, ce qui le met dans l'état *indisponible*, le rendant donc inutilisable pour des applications de capacités. Le moteur d'assurance demande alors une nouvelle application des propriétés qui ont été arrêtés (P_S et P_H). Puisque le module d'extension ayant rencontré une erreur (SELinux) n'est pas

disponible, il ne sera pas considéré pour l'application des propriétés. Elles seront donc appliquées en utilisant le module gérant les droits d'accès Unix.

Considérons que cette défaillance de SELinux se produit lors de l'application de P_H, uniquement sur le nœud serveur et dans le cas 1. On peut alors observer l'évolution des scores d'application de la politique et du taux de couverture comme décrit à la table 5.4.

Nœud	**Prop.**	**Mécanismes & Scores**				**Score de la politique**	T_C
		Avant erreur		**Après erreur**			
Serveur	P_1	SELinux	6	DAC	3.5	16 -> 13,5	1 -> 0.8
	P_3	JCE+SSH	10	JCE+SSH	10		
Client	P_1	SELinux	6	SELinux	6	10 -> 10	1 -> 1
	P_3	SSH	4	SSH	4		

TABLE 5.4 – Évolution des scores pour P_H

Ainsi, la détection d'une erreur sur un mécanisme entraîne une mise à jour de l'application et, par conséquent, une mise à jour du score d'application et du taux de couverture. Le score initial est cependant conservé afin de connaître le score idéal pour un nœud donné : si le score initial est égal au score effectif à un instant t (comme c'est le cas sur le nœud client), alors l'application est la meilleure possible sur ce nœud. En revanche, si le score effectif est inférieur au score initial (taux de couverture ayant une valeur inférieure à 1), alors l'application pourrait être améliorée (par exemple en réactivant le mécanisme).

Notons que nous n'avons pas présenté le taux de couverture T_P des propriétés dans cet exemple. En effet, dans les deux cas considérés, l'ensemble des propriétés sont appliquées et l'on a donc toujours $T_P = 1$. Ce taux évoluerait si aucun mécanisme n'était disponible pour appliquer l'une des propriétés.

5.4 Conclusion

Dans ce chapitre, nous avons présenté l'architecture du *Security Enforcement Engine* (SE^E) qui permet d'appliquer et d'assurer une politique de sécurité, en prenant en compte l'hétérogénéité des systèmes et des mécanismes. L'implémentation proposée supporte le langage d'expression des besoins de sécurité présenté au chapitre 3 et est capable de le compiler et de le projeter sur les mécanismes en utilisant les algorithmes décrits au chapitre 4. Le SE^E a été développé en Java afin de pouvoir être utilisé sur des systèmes hétérogènes tels que ceux de l'informatique en nuage. Il a été utilisé dans le cadre du projet Seed4C pour appliquer les propriétés de sécurité de plusieurs cas d'usage.

Nous avons également décrit les modules d'extension permettant la gestion des mécanismes de sécurité. Chaque module est ainsi responsable de la gestion d'un mécanisme de sécurité, ce qui permet l'utilisation de mécanismes variés et ayant des systèmes de configuration différents (fichiers de configuration, commande système). L'utilisation d'un système de modules d'extension permet d'ajouter ou de supprimer simplement des mécanismes et de s'adapter dynamiquement aux mécanismes disponibles. Ainsi, plusieurs modules d'extension ont été développés par des partenaires du projet Seed4C afin d'intégrer des mécanismes supplémentaires : certains de ces modules ont fourni de nouvelles capacités alors que d'autres ont proposé une nouvelle implémentation de capacités existantes.

De plus, le SE^E est indépendant du système sur lequel il doit appliquer une politique de sécurité. En effet, chaque module d'extension peut configurer le mécanisme associé en fonc-

tion du système considéré. Cela est fait grâce à un système de squelettes de configuration utilisés dans le cadre du fonctionnement classique (et optionnel) d'un module. Les modules d'extension sont donc chargés d'interfacer les mécanismes afin que les moteurs d'application et d'assurance du SE^E puissent utiliser leurs fonctionnalités, tout en permettant au langage d'être indépendant des mécanismes et du système d'exploitation utilisé.

Les modules d'extensions permettent également de configurer l'assurance des mécanismes et des propriétés. Le SE^E est alors capable de détecter les dysfonctionnements des mécanismes ou les problèmes d'application des propriétés (quand l'application d'une propriété n'a pas l'effet attendu). Ces erreurs sont ensuite traitées par le SE^E afin de mettre à jour dynamiquement l'application de la politique : le traitement des informations est effectué en suivant les algorithmes du chapitre 4.

Le SE^E s'adapte au système sur lequel il doit déployer la politique, en fonction des capacités et des mécanismes disponibles. La projection obtenue n'est pas toujours optimale (la présence de mécanismes plus adaptés pourrait l'améliorer) mais elle est faite au mieux selon les mécanismes disponibles. Cette projection peut être évaluée par le SE^E grâce aux scores d'applications et aux taux de couverture.

Les SE^E sont également capables de collaborer pour appliquer et assurer des propriétés impliquant plusieurs nœuds. Cette collaboration permet de sélectionner des mécanismes compatibles pour appliquer des propriétés réseaux de façon cohérente. De plus, les SE^E peuvent échanger des informations nécessaires à la bonne application des propriétés.

Une expérimentation complète sera présentée au chapitre 6.

Chapitre 6

Expérimentation

Dans le cadre du projet européen Seed4C, plusieurs expérimentations ont été menées afin d'éprouver la validité du langage et de l'architecture proposés. Le cas d'usage principal [Bousquet *et al.*, 2015] présenté dans Seed4C a été proposé par Ikusi [1]. Il est basé sur l'architecture de diffusion de publicité dans des aéroports fournie par Ikusi.

Dans ce chapitre, nous présentons une expérimentation généralisant le cas d'usage proposé par Ikusi et nous montrons comment le langage et l'architecture définis dans cette thèse peuvent être utilisés pour sécuriser cette infrastructure. De plus, cette expérimentation permet d'illustrer les phases d'assurance et de reconfiguration lors de l'exécution des systèmes.

Ce chapitre commence par présenter l'environnement de test et le cas d'usage (section 6.1). Puis, la politique de sécurité est définie à partir des besoins exprimés par l'administrateur de l'architecture logicielle (section 6.2). La section 6.3 détaille les phases d'application et d'assurance de la politique de ce cas d'usage. Enfin, la section 6.4 présente les résultats obtenus au cours de cette expérimentation.

6.1 Description du cas d'usage

6.1.1 Environnement de test

L'expérimentation est réalisée sur une plateforme OpenStack [Pepple, 2011] qui permet de déployer des machines virtuelles et de fournir un service de type IaaS. La version d'OpenStack utilisée est Juno [OpenStack, 2015]. La configuration des machines physiques utilisées est présentée à la table 6.1.

Type	**Nb.**	**CPU**	**RAM**	**Disque**	**Nom**
Dell PowerEdge T110	6	8-cores (xeon 2,67 GHz)	8 Go	1 To RAID 0 (2*500 Go)	xeon-1001 → 1006
Dell PowerEdge T110 II	5	8-cores (xeon 3,4 GHz)	16 Go	1 To RAID 0 (2*500 Go)	xeon-1007 → 1011

Table 6.1 – Configuration de l'environnement de test

La figure 6.1 détaille la configuration utilisée pour le déploiement d'OpenStack.

Deux machines physiques sont réservées pour le nœud de contrôle (*Controller*, xeon-1001) et pour le nœud de gestion de réseau (*Neutron*, xeon-1002). Le nœud de contrôle

1. `http://www.ikusi.com`

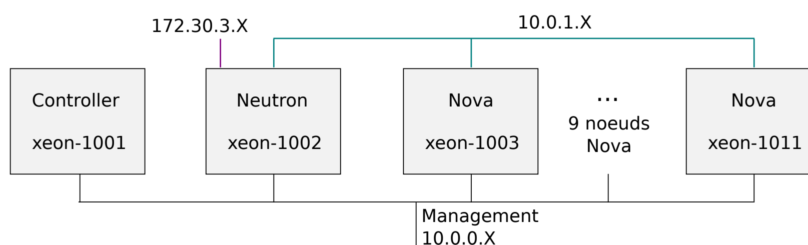

FIGURE 6.1 – OpenStack

comprend le service d'authentification (*Keystone*), le service de gestion des images (*Glance*) et la partie gestion du réseau et des nœuds de calcul. Il héberge également l'interface Web (*Horizon*) permettant de visualiser l'état du système et d'effectuer des actions (démarrer des VM...). Le nœud de gestion du réseau (*Neutron*) provisionne les réseaux des machines virtuelles et fournit les services de routage, de NAT et de DHCP. Il permet également l'accès à Internet pour les VM. Les neuf autres nœuds sont utilisés comme nœuds de calcul (*Nova*, xeon-1003 à xeon-1011) : il s'agit donc des nœuds sur lesquels les VM sont instanciées. Chacun de ces nœuds réserve 240 Go de disque aux VM : par conséquent, OpenStack dispose de 72 VCPU, 107.9 Go de RAM et 2.1 To d'espace disque pour déployer des VM.

Trois réseaux distincts sont présents :

- Un réseau d'administration (10.0.0.0/24) sur lequel toutes les machines physiques sont connectées ;
- Un réseau privé (10.0.1.0/24), incluant le nœud de gestion du réseau et les nœuds de calcul, utilisé comme réseau privé entre les VM ;
- Le réseau externe (172.30.3.0/24) permettant au nœud de gestion du réseau de fournir un accès externe à certaines VM.

Le système installé sur les nœuds physiques est CentOS 7. Les mécanismes disponibles sur ces nœuds sont iptables, DAC, SELinux, sVirt, PAM, SSH.

6.1.2 Cas d'usage

Le cas d'usage que nous présentons a pour objectif de fournir une architecture LAMP (Linux, Apache, MySQL, PHP) dans une infrastructure en nuage afin d'héberger et de gérer des sites Web. De plus, nous cherchons à fournir une administration sécurisée de LAMP à l'utilisateur.

6.1.2.1 Architecture

La figure 6.2 présente l'architecture du cas d'usage considéré.

Le cas d'usage utilise deux réseaux : un réseau externe (172.30.3.0/24) et un réseau virtuel interne (192.168.1.0/24). Cinq types de machines virtuelles utilisent ces réseaux :

1. Une VM *reverse proxy* permet à des clients d'accéder à des sites Web hébergés sur les VM Apache. Cette VM est connectée sur les réseaux externe (172.30.3.1) et interne (192.168.1.254). Cette VM fait également office de passerelle pour l'accès aux VM internes (Apache et MySQL) ;
2. Des VM *clientes* que les clients utilisent pour se connecter à leur site ou aux VM à travers le *reverse proxy*. Elles sont connectées au réseau externe ;
3. Des VM *Apache* hébergeant les sites Web. Ces machines incluent également phpMyAdmin [PHP, 2015] et ApacheGUI [ApacheGUI, 2015] afin d'administrer des sites

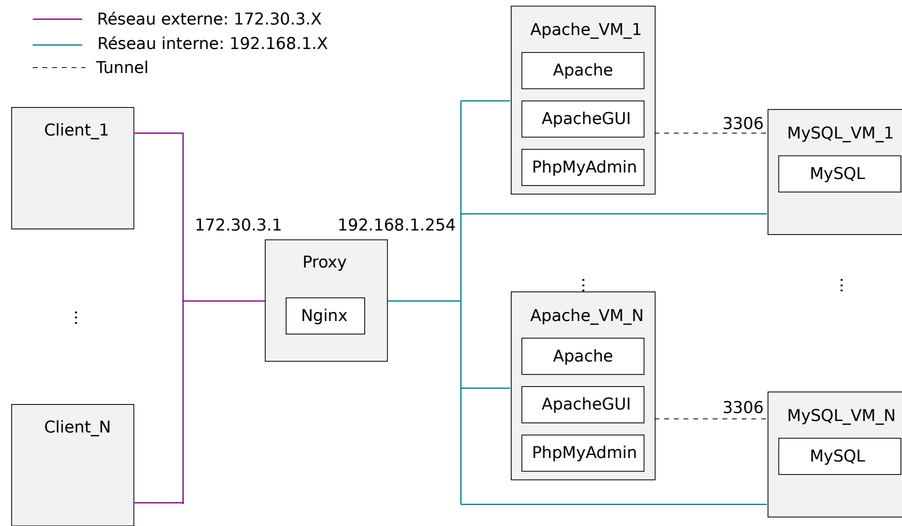

FIGURE 6.2 – Description du cas d'usage

Web. Elles ne sont accessibles que depuis le réseau interne (192.168.1.1 à 192.168.1.100) ;

4. Des VM *MySQL* fournissant une base de données. Chacune de ces machines est associée à une VM *Apache* donnée. Elles ne sont accessibles que depuis le réseau interne (192.168.1.101 à 192.168.1.200) ;

Ainsi, un client peut demander la création d'un site, ce qui entraîne le déploiement d'une VM *Apache* (192.168.1.X, $1 \leq X \leq 100$) et d'une VM *MySQL* (192.168.1.Y, $Y = X+100$). Dans le cas de cette expérimentation, le déploiement des VM est fait grâce à un script utilisant l'API d'OpenStack : la création des VM est donc effectuée automatiquement. Le nombre maximum de machines pouvant être déployées compte tenu des machines physiques disponibles est de 1 VM Proxy, 30 VM *Apache*, 30 VM *MySQL* et 30 VM Clientes.

Les VM clientes utilisent la distribution Fedora 22. Les mécanismes présents sur ces VM sont DAC, iptables, firewalld, PAM, SSH et OpenVPN. Les autres VM (Proxy, Apache et MySQL) utilisent CentOS 7 et disposent de firewalld, iptables, PAM, DAC, SELinux, sVirt, OpenVPN, JCE et SSH.

On considère quatre niveaux de protection, correspondant aux besoins de sécurité de quatre utilisateurs : le système hôte (et les VM qu'il héberge), le système des VM, la plateforme fournie au client (Apache) et les applications (PhpMyAdmin, ApacheGUI). Le cas d'usage considère ainsi quatre utilisateurs/groupes pour ces quatre niveaux :

1. *CloudProvider* : il fournit l'infrastructure utilisée. Il gère donc les hôtes physiques et fournit les VM aux clients ;
2. *TenantOperator* : il s'agit de l'administrateur du système des VM ;
3. *TenantAdmin* : il est responsable de l'administration des applications (Apache, PhpMyAdmin, ApacheGUI) fournies au client ;
4. *User* : il utilise les applications pour gérer son site Web.

6.1.2.2 Configuration du SE^E

La configuration du SE^E se compose des différentes matrices présentées à la section 3.4. En raison de la taille de matrices, on ne donne ici que les paires de mécanismes incompatibles (aux niveaux système et réseau) et dépendants. Les paires non-spécifiées sont considérées compatibles et indépendantes. Les mécanismes incompatibles au niveau système

sont iptables et firewalld et, au niveau réseau, OpenVPN et SSH. Les mécanismes de ce cas d'usage sont tous indépendants, et seuls iptables, firewalld et SELinux sont dominants (les autres mécanismes sont donc récessifs).

On note S la matrice des scores d'ordonnancement des mécanismes par capacité. Les scores non spécifiés ici sont considérés comme nuls (le mécanisme ne fournit pas la capacité). De plus, afin de simplifier la définition des scores, nous considérons que chaque mécanisme a un score identique pour l'ensemble des capacités qu'il fournit. Ce comportement est entièrement configurable.

$$\begin{array}{lclclcl} S(PAM, .*) & = & 1 & & S(OpenVPN, .*) & = & 6 \\ S(firewalld, .*) & = & 4 & & S(SSH, .*) & = & 4 \\ S(iptables, .*) & = & 6 & & S(JCE, .*) & = & 4 \\ S(sVirt, .*) & = & 1 & & S(OpenSSL, .*) & = & 6 \\ S(SELinux, .*) & = & 6 & & S(AssuranceEngine, .*) & = & 1 \\ S(DAC, .*) & = & 1 & & & & \end{array}$$

Cette configuration est identique sur toutes les VM. Elle favorise l'utilisation de certains mécanismes : iptables est préféré à firewalld (les règles peuvent être plus précises), SELinux à DAC (le contrôle d'accès obligatoire permet de garantir des propriétés de sécurité) et OpenSSL à JCE (plus de fonctionnalités).

6.2 Politique de sécurité

Cette section a pour objectif de définir la politique de sécurité à appliquer sur les différentes machines impliquées dans le cas d'usage. Pour cela, nous commençons par décrire les besoins de sécurité de l'architecture logicielle, puis nous représentons graphiquement l'architecture et ses besoins. Enfin, les ressources sont associées à des contextes, ce qui permet d'exprimer les propriétés de sécurité répondant aux besoins.

6.2.1 Besoins de sécurité

Les besoins de sécurité sont exprimés par l'administrateur de l'architecture logicielle en se basant sur la figure 6.2. Cinq besoins de sécurité généraux sont établis :

1. Isoler les services et applications systèmes : Apache, ApacheGUI, phpMyAdmin, MySQL, nginx ;
2. Isoler les services et applications réseaux : seuls les ports de ces services doivent être atteignables ;
3. Isoler les VM ;
4. Fournir un accès à la paire Apache/MySQL au client correspondant.
5. Donner à chaque utilisateur des droits spécifiques :
   - *CloudProvider* peut administrer les hôtes physiques et OpenStack ;
   - *TenantOperator* peut administrer les VM (installer les paquets, vérifier les logs systèmes), mais pas les applications administrées par *TenantAdmin* ;
   - *TenantAdmin* peut administrer les services fournis à l'utilisateur (Apache, phpMyAdmin, MySQL), modifier leur configuration, lire leurs logs et les redémarrer ;
   - *User* peut utiliser les trois applications (ApacheGUI, phpMyAdmin, MySQL).

La définition de ces besoins peut résulter de l'utilisation d'une méthode d'évaluation des risques et permet la définition de la politique.

6.2.2 Politique de sécurité

Dans cette section, nous détaillons les politiques de sécurité qui découlent des besoins et qui sont utilisées pour protéger l'architecture logicielle du cas d'usage. La figure 6.3 synthétise cette architecture logicielle. Seul un triplet de VM Client-Apache-MySQL est représenté : les politiques sur les autres VM sont identiques (à l'exception des associations de contextes aux ressources qui peuvent différer selon la distribution choisie).

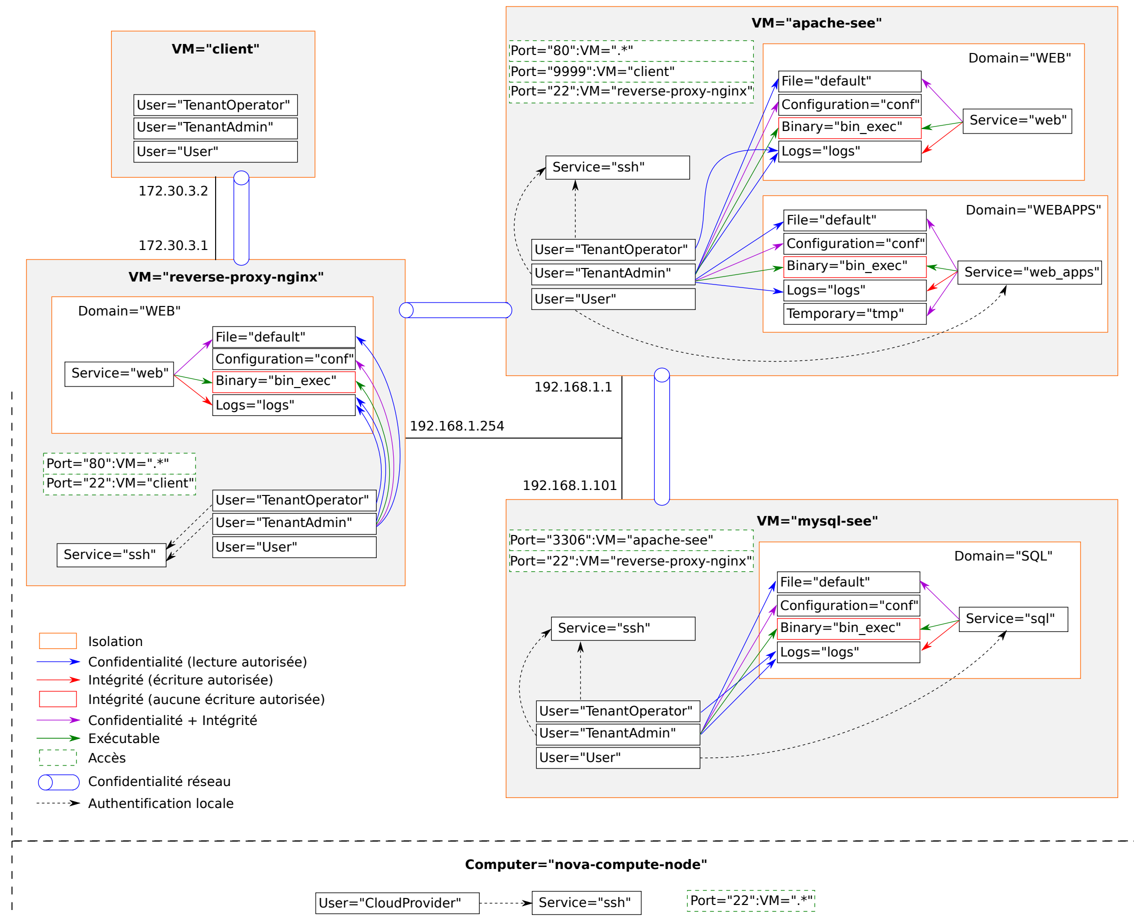

FIGURE 6.3 – Description du cas d'usage

La figure présente donc les quatre machines virtuelles du cas d'usage. La VM Apache est le serveur Web et exécute trois applications : Apache et PHPMyAdmin (domaine `Web`), et ApacheGUI (domaine `WebApps`). La VM MySQL héberge la base de données associée au serveur Web. Ces deux VM sont reliées sur le réseau interne à la VM proxy (Nginx).

Les propriétés à exprimer, et répondant aux besoins, sont les suivantes :

1. Chacune des applications est isolée dans un domaine (cadre orange) et les interactions à l'intérieur de ce domaine font partie de la propriété d'isolation. Les opérations de lecture et d'écriture (flèches bleues, rouges et violettes) vers les domaines sont précisées : les utilisateurs `TenantOperator` et `TenantAdmin` n'ont donc pas les mêmes autorisations d'interactions avec les ressources de ces applications. De plus, les exécutions de fichiers sont spécifiées (flèches vertes) pour chacun des fichiers exécutables (autorisées pour `TenantAdmin`).
2. Les cadres vert-pointillés représentent les propriétés d'accès, c'est-à-dire les demandes

d'ouverture de ports réseaux pour les VM sources spécifiées.

3. Ces VM sont déployées sur les nœuds de calcul OpenStack. Elles peuvent être sur une seule machine physique ou sur des machines différentes. Les nœuds de calculs ont tous la même politique de sécurité, qui autorise la connexion SSH du fournisseur de l'infrastructure (nécessitant ainsi l'ouverture du port SSH). De plus, chaque nœud de calcul est chargé d'isoler les différentes VM qu'il héberge (cadre orange).
4. La propriété permettant les connexions d'un client aux VM Apache et MySQL est également décrite sur le schéma.
5. Les authentifications locales autorisées (c'est-à-dire les utilisateurs autorisés à se connecter en utilisant des services) sont également visibles sur le schéma (flèches pointillées).

La figure 6.4 présente les propriétés d'authentifications distantes de la politique (flèches pointillées). Pour chaque flèche représentant une propriété d'authentification, les utilisateurs de la machine source peuvent se connecter sur un service (nœud intermédiaire) avec le nom d'utilisateur cible (fin de la flèche).

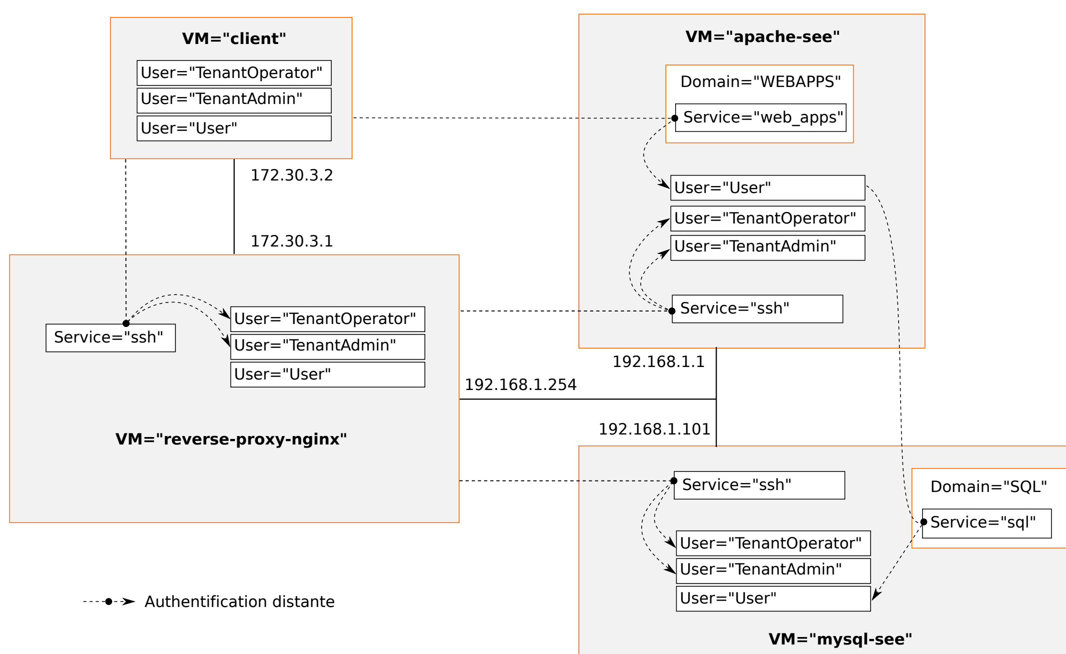

FIGURE 6.4 – Description du cas d'usage

Le reste de cette section présente un extrait des définitions des contextes, de leur association aux ressources et des propriétés à appliquer. La politique complète de ce cas d'usage est disponible à l'annexe B.

6.2.3 Contextes

En se basant sur l'architecture logicielle et sur les besoins de sécurité définis par l'administrateur de l'architecture logicielle, nous pouvons définir l'ensemble des contextes mis en jeu dans les propriétés. Nous pouvons également définir les associations entre les contextes et les ressources réelles, associations qui dépendent des systèmes déployés.

6.2.3.1 Définition

La politique de sécurité définit tout d'abord les contextes qui seront mis en jeu dans les propriétés. Cette section présente les contextes pour les politiques des différentes machines. Tous ne sont pas utilisés sur chacune des machines, mais la majorité intervient dans plusieurs politiques.

Le listing 6.1 présente les contextes associés aux machines impliquées dans ce cas d'usage. Le premier contexte identifie les machines physiques, c'est-à-dire les nœuds de calcul OpenStack. Les contextes suivants (lignes 2 à 5) identifient chacune des VM, le contexte suivant (ligne 6) identifie l'ensemble de ces VM, et les deux derniers contextes identifient toutes les machines.

```
1 HOST                    = (Hardware.Computer = "nova-compute-node");
2 HostReverseProxy        = (Hardware.Computer.VM = "reverse-proxy-nginx");
3 HostClient              = (Hardware.Computer.VM = "client");
4 HostServerWEB           = (Hardware.Computer.VM = "apache-see");
5 HostServerSQL           = (Hardware.Computer.VM = "mysql-see");
6 VMs                     = (Hardware.Computer.VM = ".*");
7 anyone                  = (Hardware.Computer = ".*");
8 anyIP                   = (Network.IP = ".*");
```

Listing 6.1 – Contextes des machines

Le listing 6.2 définit les contextes des utilisateurs intervenant dans le cas d'usage ainsi que leurs rôles. Les rôles possibles sont `StandardUser` (pas d'autorisation spécifique), `PackageAdmin` (installation de paquets), `SysnetAdmin` (gestion du réseau), `LoggingAdmin` (accès aux logs systèmes) et `WEBAdmin` (administration des applications Web). Les deux premiers utilisateurs ont un rôle d'administrateur système, `CloudProvider` pour les machines hôtes et `TenantOperator` pour les VM. `TenantAdmin` administre les applications Web exécutées sur les VM et `User` les utilise. Les propriétés spécifiques aux rôles systèmes ne seront pas détaillées, puisque seule la partie de la politique liée au cas d'usage est donnée.

```
1 CloudProvider  = (Identity.Username="idCloudProvider"):(Identity.Role="
      StandardUser|PackageAdmin|SysnetAdmin");
2 TenantOperator = (Identity.Username="idTenantOperator"):(Identity.Role="
      StandardUser|PackageAdmin|SysnetAdmin");
3 TenantAdmin    = (Identity.Username="idTenantAdmin"):(Identity.Role="
      StandardUser|LoggingAdmin|WEBAdmin");
4 User           = (Identity.Username="idUser"):(Identity.Role="StandardUser");
```

Listing 6.2 – Contextes des utilisateurs

Le listing 6.3 contient les contextes associés aux ressources des applications fournies. Les lignes 2 à 4 définissent trois domaines qui correspondent aux différentes applications présentes. `DomainWEB` identifie à la fois `nginx` (sur la VM proxy), et Apache et PHP-MyAdmin (sur la VM Apache). `DomainWEBAPPS` identifie ApacheGUI, et `DomainSQL` identifie la base de données. Les contextes des lignes 7 à 10 identifient les fichiers du domaine Web, et les deux groupes de contextes suivants (lignes 12-16 et 18-21) identifient ceux des domaines WebApps et SQL.

```
1 // Domains
2 DomainWEB     = (Domain="App_Web");
3 DomainWEBAPPS = (Domain="App_Web_Apps");
4 DomainSQL     = (Domain="App_Database");
5
6 // Resources
```

```
7  FileWEB        = DomainWEB:(Type.Passive.Data.File="default");
8  BinaryWEB      = DomainWEB:(Type.Passive.Data.File.Binary="bin_exec");
9  ConfigWEB      = DomainWEB:(Type.Passive.Data.File.Configuration="conf");
10 LogWEB         = DomainWEB:(Type.Passive.Data.File.Logs="logs");
11
12 FileWEBAPPS    = DomainWEBAPPS:(Type.Passive.Data.File="default");
13 BinaryWEBAPPS = DomainWEBAPPS:(Type.Passive.Data.File.Binary="bin_exec");
14 ConfigWEBAPPS = DomainWEBAPPS:(Type.Passive.Data.File.Configuration="conf");
15 TmpWEBAPPS     = DomainWEBAPPS:(Type.Passive.Data.File.Temporary="tmp");
16 LogWEBAPPS     = DomainWEBAPPS:(Type.Passive.Data.File.Logs="logs");
17
18 FileSQL        = DomainSQL:(Type.Passive.Data.File="default");
19 BinarySQL      = DomainSQL:(Type.Passive.Data.File.Binary="bin_exec");
20 ConfigSQL      = DomainSQL:(Type.Passive.Data.File.Configuration="conf");
21 LogSQL         = DomainSQL:(Type.Passive.Data.File.Logs="logs");
```

Listing 6.3 – Contextes des domaines et de leurs fichiers

Le listing 6.4 contient les contextes identifiant les services et applications impliqués dans le cas d'usage : SSH et les trois services appartenant aux domaines définis précédemment.

```
1 ServiceSSH     = (Type.Active.Service="SSH");
2 ServiceWEB     = HostServerWEB:DomainWEB:(Type.Active.Service="App_Web"):(
      Identity.Role="System");
3 ServiceWEBAPPS = HostServerWEB:DomainWEBAPPS:(Type.Active.Service="App_Web_Apps
      "):(Identity.Role="System");
4 ServiceSQL     = HostServerSQL:DomainSQL:(Type.Active.Service="App_Database"):(
      Identity.Role="System");
```

Listing 6.4 – Contextes des services et applications

Enfin, le listing 6.5 contient les contextes indiquant les ports intervenant dans la politique : ceux des applications (lignes 1 à 3) et celui pour SSH (ligne 4).

```
1 WEBPort     = (Network.Port="80");
2 WEBAPPSPort = (Network.Port="9999");
3 MysqlPort   = (Network.Port="3306");
4 SSHPort     = (Network.Port="22");
```

Listing 6.5 – Contextes des ports réseaux

Notons que les contextes définis sont indépendants du système de nommage des ressources du système. Ces contextes peuvent être fournis à l'ensemble des machines (physiques et virtuelles) sans distinction du rôle de la machine (client, proxy, serveur Web, base de données) ou du système utilisé (Fedora, CentOS, ...).

6.2.3.2 Association

Cette section présente l'unique partie de la politique qui est dépendante du système sur lequel la politique doit être appliquée. En effet, elle décrit l'association des contextes abstraits aux ressources réelles et est donc dépendante du système de nommage utilisé et de la localisation des ressources. Étant donné que la même distribution est utilisée sur tous les nœuds serveurs, cette association est identique pour toutes ces machines.

Comme vu à la section 3.3.3, quatre types d'associations sont possibles : `u` pour les utilisateurs, `c` pour les adresses IP, `o` pour les ressources passives et `p` pour les processus. Le listing 6.6 contient les associations des contextes utilisateurs. La première colonne indique le type d'association dont il s'agit, la deuxième les noms d'utilisateurs Unix et la troisième leur contexte associé. On a ainsi les quatre utilisateurs intervenant dans le cas d'usage.

```
u cloudprovider   CloudProvider
u tenant-admin    TenantAdmin
u tenant-operator TenantOperator
u user            User
```

Listing 6.6 – Association des contextes aux utilisateurs

Dans un second temps, les contextes des machines sont associés à leur adresse IP (voir listing 6.7). Les deux premières lignes donnent les adresses IP de la VM proxy : une seule VM proxy est déployée et ces adresses ne changent donc pas. Les trois lignes suivantes contiennent les adresses d'un groupe de trois VM : elles sont fixées lors du déploiement des VM et donc renseignées dynamiquement dans la politique avant son application. Notons que la VM client a une adresse sur le réseau externe et non le réseau interne, et que la VM proxy a une adresse sur chaque réseau.

```
c 192.168.1.254 HostReverseProxy
c 172.30.3.1    HostReverseProxy
c 192.168.1.1   HostServerWEB
c 192.168.1.101 HostServerSQL
c 172.30.3.2    HostClient
```

Listing 6.7 – Association des contextes aux machines

Le listing 6.8 contient un extrait des associations de contextes aux ressources passives et aux processus sur la VM Apache (les associations relatives au domaine `Web`). On voit ainsi que les fichiers de ce domaine (lignes 1 à 12) sont relatifs à deux applications : Apache et PHPMyAdmin. Le contexte `ServiceWEB` est quant à lui associé à un fichier : lorsque ce fichier sera exécuté, le processus résultant aura le contexte associé.

```
o /usr/sbin/httpd                        BinaryWEB
o /usr/lib/httpd(/.*)?                   BinaryWEB
o /etc/httpd(/.*)?                       ConfigWEB
o /var/www(/.*)?                         FileWEB
o /var/www/cgi-bin(/.*)?                 BinaryWEB
o /etc/phpMyAdmin(/.*)?                  ConfigWEB
o /usr/share/phpMyAdmin(/.*)?            FileWEB
o /var/log/httpd(/.*)?                   LogWEB
o /usr/lib/systemd/system/httpd\.service BinaryWEB
o /var/run/httpd(/.*)?                   FileWEB
o /etc/httpd/logs                        LogWEB
o /etc/httpd/modules                     BinaryWEB

p /usr/sbin/httpd                        ServiceWEB
```

Listing 6.8 – Association des contextes aux objets/processus sur la VM Apache

Considérons maintenant l'association de ces mêmes contextes aux ressources de la VM proxy (listing 6.9). Les contextes sont associés aux ressources de l'application Nginx (lignes 1 à 6) et à son processus (ligne 8). L'application à protéger est donc différente de celles sur la VM Apache, cependant les contextes utilisés sont identiques : les propriétés peuvent donc être les mêmes sur les deux VM (à condition que les besoins de sécurité soient identiques). Les propriétés sont donc bien indépendantes du système de nommage.

```
o /usr/sbin/nginx                        BinaryWEB
o /etc/nginx(/.*)?                       ConfigWEB
o /usr/share/nginx(/.*)?                 FileWEB
o /var/log/nginx(/.*)?                   LogWEB
o /usr/lib/systemd/system/nginx\.service BinaryWEB
```

```
6 o /var/run/nginx\.pid                    FileWEB
7
8 p /usr/sbin/nginx                        ServiceWEB
```

Listing 6.9 – Association des contextes aux objets/processus sur la VM Proxy

On voit ainsi que seules les associations de contextes sont dépendantes du système sur lequel la politique doit être appliquée. Cependant, certains éléments peuvent être générés automatiquement selon les configurations classiques des distributions ou par une découverte automatique utilisant le système de paquets, simplifiant ainsi l'écriture des associations.

6.2.4 Propriétés

Une fois les contextes définis et associés aux ressources, il est possible de définir les propriétés répondant aux besoins de sécurité. Nous séparons ces propriétés en trois niveaux qui correspondent aux niveaux IaaS, PaaS et SaaS de l'informatique en nuage. Afin de simplifier cette politique, nous ne nous intéressons ici qu'aux applications spécifiques au cas d'usage : la politique de base du système (non spécifique au cas d'usage) n'est pas présentée.

6.2.4.1 IaaS

La première partie de la politique concerne la partie IaaS du cas d'usage. Elle se divise en deux parties : la politique à appliquer sur les machines physiques (qui peut être définie par le fournisseur de l'infrastructure) et celle à appliquer sur les VM (qui peut être définie par l'administrateur des VM).

Machines physiques La politique appliquée sur les machines physiques correspond à celle demandée par le fournisseur de l'architecture et est présentée au listing 6.10.

```
1 Authentication (anyone, ServiceSSH, CloudProvider);
2 Access (SSHPort, anyIP);
3 Isolation (VMs);
```

Listing 6.10 – Politique des machines physiques

La première propriété permet au fournisseur de l'infrastructure (`CloudProvider`) de se connecter en SSH sur les machines physiques. Le premier contexte de la propriété d'authentification indique la machine d'où provient la tentative de connexion (ici, toutes les machines), le deuxième spécifie le service sur lequel l'authentification se produit (ici, SSH), et le dernier les utilisateurs autorisés à se connecter. La seconde propriété demande l'ouverture du port d'écoute de SSH afin de permettre les connexions. Enfin, la troisième propriété demande l'isolation de l'ensemble des VM déployées sur cette machine physique.

Machines virtuelles La deuxième politique de niveau IaaS est celle établie par le client IaaS et elle a pour but de protéger le système des machines virtuelles. Considérons la politique du listing 6.11 protégeant le système de la VM Apache.

```
1 Authentication (HostServerWEB, ServiceSSH, "TenantAdmin|TenantOperator");
2 Authentication (HostReverseProxy, ServiceSSH, "TenantAdmin|TenantOperator");
3
4 Access (SSHPort, HostServerWEB);
5 Access (SSHPort, HostReverseProxy);
```

Listing 6.11 – Politique système des VM Apache

Les deux premières propriétés gèrent les authentifications possibles sur la VM. Ainsi, les connexions SSH sont autorisées pour l'administrateur de la VM (*TenantOperator*) et pour l'administrateur des applications (*TenantAdmin*) en local (*HostServerWEB*) et depuis la machine proxy (*HostReverseProxy*). Les deux autres propriétés permettent d'ouvrir le port SSH (pour les connexions locales ou en provenance du proxy).

Les politiques pour les autres VM sont similaires à celle du listing 6.11 : des propriétés d'authentification par SSH (et éventuellement sur les autres services) et des propriétés d'accès pour ouvrir des ports réseaux.

Machines physiques et virtuelles Certaines propriétés de la politique sont appliquées sur l'ensemble des machines, à la fois physiques et virtuelles. Tout d'abord, une politique de base peut être utilisée pour protéger le système (par exemple, protéger les applications et les logs systèmes). Cependant, cette partie de la politique n'est pas détaillée ici et nous choisissons de concentrer cette expérimentation sur la protection de l'architecture logicielle et non sur le système.

Ainsi, dans notre cas d'usage, le politique commune est constituée d'une propriété d'assurance (listing 6.12) permettant de fixer la fréquence des tests à effectuer concernant l'état des mécanismes et l'application des propriétés. La première ligne définit un contexte contenant une fréquence (10 minutes). La deuxième est la propriété d'assurance : les scripts générés pour vérifier les mécanismes et les propriétés seront donc exécutés à la fréquence spécifiée.

```
1 Frequency = (Tests.Frequency="10m");
2 Assurance (Frequency);
```

Listing 6.12 – Propriété d'assurance

Cette propriété d'assurance est une version simple. Elle pourrait être affinée en passant en paramètre des contextes pour sélectionner les différents tests à effectuer. Les tests pourraient alors être exécutés à des fréquences différentes selon le type des tests et leur criticité.

6.2.4.2 PaaS

La politique de niveau PaaS présentée au listing 6.13 permet de protéger les applications fournies par la plateforme. Elle est donc définie par l'administrateur de l'architecture logicielle, c'est-à-dire le client du PaaS.

```
1 Authentication (HostClient, ServiceWEBAPPS, "User");
2
3 @Isolation (DomainWEB, ServiceWeb);
4 @Isolation (DomainWEBAPPS, ServiceWEBAPPS);
5
6 Confidentiality_access_control (LogWEB, TenantOperator);
7 Confidentiality_access_control (LogWEB, TenantAdmin);
8 Confidentiality_access_control (LogWEBAPPS, TenantAdmin);
9
10 Confidentiality_access_control (ConfigWEB, TenantAdmin);
11 Confidentiality_access_control (FileWEB, TenantAdmin);
12 Confidentiality_access_control (ConfigWEBAPPS, TenantAdmin);
13 Confidentiality_access_control (FileWEBAPPS, TenantAdmin);
14
15 Integrity (ConfigWEB, TenantAdmin);
16 Integrity (ConfigWEBAPPS, TenantAdmin);
17
```

```
18 Executable (BinaryWEB, TenantAdmin);
19 Executable (BinaryWEBAPPS, TenantAdmin);
20
21 Access (WEBAPPSPort, HostClient);
```

Listing 6.13 – Politique PaaS des VM Apache

La première propriété autorise la connexion de l'utilisateur `User` sur l'application ApacheGUI depuis la VM Client.

Les deux propriétés suivantes font appel à la classe **@Isolation** pour isoler les applications Apache, PHPMyAdmin et ApacheGUI. La propriété **Sandbox_App**, définie au listing 6.14, fait partie de cette classe et correspond aux contextes passés en arguments.

```
1  boolean Sandbox_App (Domain SCDom, Type.Active SCApp) {
2   enforcement && assurance {
3      Isolation_System (SCDom);
4      Confidentiality_access_control (SCDom:(Type.Passive.Data.File=".*"), SCDom:
           SCApp);
5      Integrity (SCDom:(Type.Passive.Data.File.Configuration=".*"), SCDom:SCApp);
6      Integrity (SCDom:(Type.Passive.Data.File.Logs=".*"), SCDom:SCApp);
7      Integrity (SCDom:(Type.Passive.Data.File.Key=".*"), SCDom:SCApp);
8      Integrity (SCDom:(Type.Passive.Data.File.Temporary=".*"), SCDom:SCApp);
9      Integrity (SCDom:(Type.Passive.Data.File.Binary=".*"));
10     Integrity (SCDom:(Type.Passive.Data.File.Executable=".*"));
11     Executable (SCDom:(Type.Passive.Data.File.Executable=".*"), SCDom:SCApp);
12  }
13 }
```

Listing 6.14 – Propriété d'isolation système d'une application

La propriété **Sandbox_App** isole, au niveau système, un domaine `SCDom` (ligne 3) qui est lié à une application ou un service `SCApp`, puis définit les interactions autorisées à l'intérieur de ce domaine. Ainsi, l'ensemble des fichiers sont accessibles en lecture par une application appartenant à ce domaine (ligne 4). Les fichiers de configuration, de logs, de clefs, et les fichiers temporaires peuvent être édités par cette application (lignes 5 à 8). Les fichiers binaires et exécutables ne peuvent pas être édités (lignes 9 et 10). Enfin, les applications du domaine sont autorisées à exécuter les fichiers exécutables liés au domaine (ligne 11). Cette propriété est générique et correspond aux besoins de sécurité que l'on peut appliquer sur un grand nombre d'applications ou services systèmes. Notons que si, pour un domaine, aucune ressource n'est associée à un contexte (par exemple, aucune clef de chiffrement), la propriété interne correspondante est ignorée mais la propriété **Sandbox_App** est appliquée.

Lorsque les domaines `DomainWEB` et `DomainWEBAPPS` sont isolés, des interactions supplémentaires sont définies au listing 6.13 afin de correspondre aux besoins de sécurité définis. L'administrateur du système (`TenantOperator`) est autorisé à lire les fichiers de logs du serveur Web (ligne 6), et l'administrateur des applications peut lire les logs du serveur Web et des applications (lignes 7 et 8). L'administrateur des applications est quant à lui autorisé à lire les fichiers de configuration et les fichiers divers des applications (lignes 10 à 13) et à éditer les fichiers de configuration (lignes 15 et 16). De plus, il peut exécuter les fichiers binaires (lignes 18 et 19) du domaine afin, par exemple, de redémarrer le service associé. Enfin, le port utilisé par ApacheGUI est ouvert (ligne 21) et accessible uniquement depuis le client. Des politiques similaires sont définies pour les applications des VM Proxy et MySQL. Elles sont cependant plus simples puisque ces VM n'ont qu'un seul domaine applicatif à sécuriser (respectivement `DomainWEB` et `DomainSQL`).

6.2.4.3 SaaS

La dernière partie de la politique est celle du niveau SaaS (voir listing 6.15). Elle permet à l'utilisateur des applications d'accéder à ses applications allouées (ApacheGUI, PHPMyAdmin) afin de créer des sites Web.

```
1 Access (WEBPort, AnyIP);
2 Confidentiality (HostReverseProxy, "HostClient|HostServerWEB|HostServerSQL");
```

Listing 6.15 – Politique SaaS des VM Apache

La première propriété ouvre le port Web à toutes les adresses IP, autorisant ainsi les connexions au site Web. La deuxième propriété crée un tunnel entre le client et les VM Apache et MySQL correspondantes : l'utilisateur peut ainsi se connecter sur ses VM en passant par le proxy, de manière transparente. Cette propriété de confidentialité réseau est appliquée sur chaque VM intervenant dans le quadruplet de VM Client / Reverse Proxy / Apache / MySQL, afin de créer, par exemple, un tunnel ou un VPN en fonction des mécanismes disponibles sur chaque hôte. Par conséquent, les mécanismes choisis doivent être compatibles au niveau réseau afin d'assurer une cohérence de l'application.

De plus, dans le cas de l'utilisation d'un VPN, le port sur lequel le serveur écoute est choisi par le module d'extension. Par conséquent, le module utilise les capacités internes du SE^E pour partager le numéro du port qui a été choisi avec les autres SE^E intervenant dans cette propriété. On remarque aussi que la propriété de confidentialité réseau (section 3.5.2.2) ne contient pas de capacité d'ouverture de port. Cette ouverture est donc gérée par le module lui-même qui demande donc au SE^E l'application d'une nouvelle capacité, induite par le fonctionnement spécifique du module d'extension choisi.

Cette propriété étant établie entre un triplet de VM Client / Apache / MySQL, les communications entre VM de triplets différents ne sont pas autorisées. Ainsi, un utilisateur sur la VM Client du premier triplet pourra accéder au serveur Web du premier triplet, mais pas à ceux des autres triplets.

Enfin, notons que nous ne définissons pas de propriété protégeant les informations stockées dans la base de données ou l'exécution de l'application. De telles propriétés pourraient être ajoutées, cependant nous avons choisi de concentrer ces travaux sur les mécanismes systèmes et réseaux, et non sur les mécanismes applicatifs. Les modules d'extensions correspondant aux mécanismes applicatifs qui pourraient être utilisés ne sont donc pas actuellement implémentés.

6.3 Application et assurance

Cette section décrit l'application des différents niveaux de politiques sur les machines, en fonction de la configuration et des mécanismes disponibles sur chaque machine. Nous considérons le cas de l'application idéale (tous les mécanismes sont disponibles) et le mode de sélection `Défaut`. De plus, les tests d'assurance, exécutés par le SE^E, sont ici simulés manuellement. Leurs résultats sont donc également présentés. Notons que nous ne détaillons pas les configurations générées : des exemples ont été donnés au chapitre 5.

6.3.1 IaaS

Les machines physiques ont une politique composée de trois propriétés. La propriété d'authentification est appliquée par PAM et les accès SSH (propriété d'accès) sont gérés par iptables (le seul pare-feu disponible sur les machines hôtes). La propriété d'isolation met en jeu des VM et est par conséquent appliquée par sVirt qui peut confiner des VM dans des

domaines. Chaque instance de VM reçoit un label différent, ce qui permet aux VM d'être isolées les unes des autres. Ces labels sont générés dynamiquement à la création des VM et leurs interactions sont contrôlées par SELinux. La deuxième partie de la politique IaaS est celle sécurisant le système des VM. Les propriétés d'authentification sont appliquées en utilisant PAM et celles d'accès en utilisant iptables (iptables a un score d'ordonnancement plus élevé que firewalld, ce qui le favorise). Enfin, une propriété d'assurance est chargée d'effectuer les tests d'assurance, c'est-à-dire d'exécuter les scripts générés pour vérifier l'état des mécanismes et la bonne application des propriétés. Une table récapitulative des mécanismes utilisés pour chaque niveau est disponible à la section 6.3.4.

Ainsi, pour vérifier l'application des propriétés d'authentification et d'accès, une tentative de connexion SSH peut être effectuée (listing 6.16).

```
1 [root@reverse-proxy-nginx ~]# ssh tenantOperator@apache-see-1
2 [tenantOperator@apache-see-1 ~]$
```

Listing 6.16 – Connexion sur la VM Apache depuis la VM Proxy

La connexion étant réussie, le port SSH a bien été ouvert et l'utilisateur `tenantOperator` peut se connecter. Les deux propriétés ont donc été appliquées comme attendu. D'autres tests, non détaillés ici, ont été effectués afin de vérifier que seuls les utilisateurs autorisés à se connecter le peuvent. De plus, des tests peuvent être générés pour vérifier que les mécanismes utilisés sont bien fonctionnels. Ainsi, le listing 6.17 montre comment vérifier le statut d'iptables (lignes 1 à 6) et les règles qui sont générées (lignes 8 à 13).

```
1  [root@reverse-proxy-nginx ~]# /bin/systemctl status  iptables.service
2  iptables.service - IPv4 firewall with iptables
3            Loaded: loaded (/usr/lib/systemd/system/iptables.service; enabled)
4            Active: active (exited) since Sun 2015-05-31 02:59:18 EDT; 12min ago
5            [...]
6
7  [root@reverse-proxy-nginx ~]# iptables -L -n
8  Chain INPUT (policy ACCEPT)
9  target   prot opt source        destination
10 ACCEPT   tcp  --  0.0.0.0/0     0.0.0.0/0     ctstate NEW tcp dpt:22
11 ACCEPT   all  --  0.0.0.0/0     0.0.0.0/0     ctstate RELATED,ESTABLISHED
12 REJECT   all  --  0.0.0.0/0     0.0.0.0/0     reject-with icmp-host-prohibited
13 [...]
```

Listing 6.17 – Vérification des règles iptables sur l'hôte physique

On peut donc voir que iptables est bien actif (ligne 4) et que le port SSH est ouvert comme spécifié par la propriété d'accès (ligne 10).

Enfin, les propriétés d'accès peuvent être vérifiées par des tests indépendants du mécanisme utilisé. Ainsi, le listing 6.18 montre comment la propriété d'accès sur la VM Apache peut être vérifiée en utilisant nmap depuis la VM Proxy.

```
1 [root@reverse-proxy-nginx ~]# nmap apache-see-1
2 Starting Nmap 6.40 ( http://nmap.org ) at 2015-06-09 09:05 EDT
3 Nmap scan report for apache-see-1 (192.168.1.1)
4 Host is up (0.00053s latency).
5 Not shown: 997 filtered ports
6 PORT     STATE SERVICE
7 22/tcp   open  ssh
8 [...]
```

Listing 6.18 – Vérification des règles d'accès sur la VM Apache

Le port SSH est bien ouvert (ligne 7) comme spécifié par la propriété d'accès de la politique IaaS.

Ces exemples de tests sont générés par le moteur d'application sous forme de scripts et, une fois exécutés et en cas d'erreur, permettent de déclencher une mise à jour de l'application de la politique par le moteur d'assurance.

6.3.2 PaaS

La politique PaaS sécurise les applications déployées dans les VM. Les propriétés de contrôle d'accès sont appliquées en utilisant SELinux (qui a un meilleur score d'ordonnancement que DAC). Celui-ci génère un nouveau module pour chaque domaine et autorise les actions correspondant aux différentes propriétés. La propriété d'accès est elle appliquée avec iptables ou firewalld selon les VM.

L'état de SELinux peut être vérifié grâce aux commandes présentées au listing 6.19.

```
1 [root@apache-see-1 ~]# getenforce
2 Enforcing
3
4 [root@apache-see-1 ~]# semodule -l
5 [...]
6 seeWEB          1.0.0
7 seeWEBAPPS      1.0.0
8 [...]
```

Listing 6.19 – Vérification du statut de SELinux sur la VM Apache

SELinux est donc actif et en mode *Enforcing* (actions interdites bloquées, et non pas seulement détectées, ligne 2) et les modules SELinux correspondants aux domaines applicatifs à protéger ont bien été générés et chargés (lignes 6 et 7). De plus, pour vérifier l'application des propriétés appliquées avec SELinux, des tests directs peuvent être effectués (listing 6.20).

```
1 [tenantOperator@apache-see-1 ~]$ test -r /etc/httpd/conf/httpd.conf
2 [tenantOperator@apache-see-1 ~]$ echo $?
3 1
4 [tenantAdmin@apache-see-1 ~]$ test -r /etc/httpd/conf/httpd.conf
5 [tenantAdmin@apache-see-1 ~]$ echo $?
6 0
```

Listing 6.20 – Vérification de l'application des propriétés sur la VM Apache

Le test consiste à vérifier les droits en lecture sur un fichier ayant le contexte `ConfigWEB` pour les deux utilisateurs : `tenantOperator` n'est pas autorisé à lire le fichier (lignes 1 à 3) alors que `tenantAdmin` a les bons droits (lignes 4 à 6). Cela correspond donc bien aux propriétés de confidentialité sur le contexte `ConfigWEB`.

Concernant le statut du pare-feu utilisé, il peut être vérifié comme vu à la section précédente (listing 6.17, dans le cas d'iptables) ou comme indiqué au listing 6.21 (dans le cas de firewalld)

```
1 [root@apache-see-1 ~]# sudo systemctl status firewalld
2 firewalld.service - firewalld - dynamic firewall daemon
3    Loaded: loaded (/usr/lib/systemd/system/firewalld.service; disabled)
4    Active: active (running) since Sun 2015-05-31 02:55:59 EDT; 2min 57s ago
5    [...]
6
7 [root@apache-see-1 ~]# firewall-cmd --list-all
8 public
```

```
 9   interfaces: eth0
10   services: mdns dhcpv6-client ssh
11   ports: 9999/tcp
12 [...]
```

Listing 6.21 – Vérification des règles firewalld sur la VM Apache

On voit donc que firewalld est actif (ligne 4) et que le port `PortWEBAPPS` (9999) est ouvert (ligne 11).

6.3.3 SaaS

Enfin, pour la partie SaaS de la politique, le SE^E utilise iptables pour la propriété d'accès. La propriété de confidentialité est appliquée en combinant OpenVPN et JCE. De plus, l'utilisation d'OpenVPN entraîne de nouvelles demandes de capacités à iptables pour ouvrir le port d'écoute d'OpenVPN. Le listing 6.22 donne le processus OpenVPN s'exécutant sur la VM Proxy. Dans le cas où plusieurs triplets de VM Client / Apache / MySQL sont déployés, autant de processus serveurs OpenVPN seront listés.

```
1 [root@reverse-proxy-nginx ~]# ps aux | grep openvpn
2 root 1521  0.0  0.1  44864  1132 ? Ss   09:28   0:00 /usr/sbin/openvpn --config
      /etc/openvpn/vpn-server.conf0
```

Listing 6.22 – Vérification de la présence d'un serveur OpenVPN sur la VM Proxy

Le listing 6.23 montre la nouvelle interface `tun0` créée pour le serveur OpenVPN et l'adresse de la VM Proxy pour ce VPN (10.4.0.1).

```
1 [root@reverse-proxy-nginx ~]# ip addr show tun0
2 4: tun0: <POINTOPOINT,MULTICAST,NOARP,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast
     state UNKNOWN qlen 100
3     link/none
4     inet 10.4.0.1 peer 10.4.0.2/32 scope global tun0
```

Listing 6.23 – Vérification de la création de l'interface pour OpenVPN sur la VM Proxy

Enfin, le listing 6.24 présente les règles iptables correspondant aux demandes d'ouverture de ports et d'interface par le module d'extension OpenVPN. Ainsi, le port 1194/udp est ouvert pour les chaînes `INPUT` et `OUTPUT` (lignes 5 et 16), et l'interface `tun0` est ouverte pour les chaînes `INPUT`, `FORWARD` et `OUTPUT` (lignes 4, 10 et 15). Si d'autres triplets de VM sont présents, le numéro de port et le nom de l'interface sont incrémentés et les règles de pare-feu correspondantes sont ajoutées. On peut noter que les règles ne spécifient pas d'adresse IP puisque les connexions sont authentifiées par OpenVPN. Ce comportement pourrait cependant être adapté.

```
 1 [root@reverse-proxy-nginx ~]# iptables -L -n -v
 2 Chain INPUT (policy ACCEPT 0 packets, 0 bytes)
 3  pkts bytes target     prot opt in     out   source     destination
 4     0     0 ACCEPT     all  --  tun0   *     0.0.0.0/0  0.0.0.0/0
 5    56  5064 ACCEPT     udp  --  eth0   *     0.0.0.0/0  0.0.0.0/0  udp dpt:1194
 6     0     0 REJECT     all  --  *      *     0.0.0.0/0  0.0.0.0/0
 7 [...]
 8 Chain FORWARD (policy ACCEPT 0 packets, 0 bytes)
 9  pkts bytes target     prot opt in     out   source     destination
10     0     0 ACCEPT     all  --  tun0   *     0.0.0.0/0  0.0.0.0/0
11     0     0 REJECT     all  --  *      *     0.0.0.0/0  0.0.0.0/0
12 [...]
13 Chain OUTPUT (policy ACCEPT 156 packets, 17580 bytes)
14  pkts bytes target     prot opt in     out   source     destination
```

```
15     0     0 ACCEPT     all  --  *      tun0  0.0.0.0/0  0.0.0.0/0
16    56  4928 ACCEPT     udp  --  *      eth0  0.0.0.0/0  0.0.0.0/0  udp dpt:1194
17     0     0 REJECT     all  --  *      *     0.0.0.0/0  0.0.0.0/0
18 [...]
```

Listing 6.24 – Vérification des règles iptables sur la VM Proxy

Le listing 6.25 montre la vérification du fonctionnement du tunnel : on voit donc qu'il est possible d'atteindre la VM Apache à partir de la VM Proxy en utilisant le tunnel VPN qui a été mis en place.

```
1 [root@reverse-proxy-nginx ~]# ping apache-see-1
2 PING apache-see-1 (10.4.0.6) 56(84) bytes of data.
3 64 bytes from apache-see-1 (10.4.0.6): icmp_seq=1 ttl=64 time=0.694 ms
4 64 bytes from apache-see-1 (10.4.0.6): icmp_seq=2 ttl=64 time=0.672 ms
```

Listing 6.25 – Vérification du fonctionnement du VPN que la VM Proxy

On peut donc vérifier que le VPN a bien été établi et que le module d'extension OpenVPN a bien demandé l'ouverture des ports et des interfaces nécessaires à un module de pare-feu. Notons que la politique SaaS présentée dans ce cas d'usage ne protège pas les applications au cours de leur exécution. L'utilisation de mécanismes adaptés permettrait de protéger les applications. On pourrait par exemple envisager l'utilisation de PaX [PaX Team, 2003] pour protéger des dépassements de tampons, de Modsecurity [Ristic, 2010] pour filtrer les requêtes sur Apache, ou de SEJava [Venelle *et al.*, 2013] pour contrôler les interactions entre les objets Java d'une application.

6.3.4 Synthèse

La table 6.2 présente une synthèse des mécanismes utilisés pour chacune des machines du cas d'usage, dans le cas du mode de sélection `Défaut` et d'une application idéale.

Mécanisme	**Hôte physique**	**Machines virtuelles**			
		Proxy	**Apache**	**MySQL**	**Client**
PAM	IaaS	IaaS	IaaS	IaaS/PaaS	-
Web Access	-	-	PaaS	-	-
iptables	IaaS	All	All	All	SaaS
sVirt	IaaS	-	-	-	-
SELinux	-	PaaS	PaaS	PaaS	-
OpenVPN	-	SaaS	SaaS	SaaS	SaaS
JCE	-	SaaS	-	-	-
AssuranceExec	All	All	All	All	All

TABLE 6.2 – Mécanismes utilisés par machine et par niveau de politique

La section suivante illustrera des cas de reconfiguration automatique entraînant l'utilisation d'autres mécanismes.

6.4 Résultats et évaluations

Cette section présente les résultats obtenus au cours des expérimentations. Au total, trente tests ont été effectués : pour chaque test, N_T triplets de VM ont été utilisés (où N_T est le numéro du test). Pour chaque test, le protocole d'expérimentation est le suivant :

1. Initialisation de la VM Proxy. L'initialisation comprend la création de la machine

par OpenStack, son démarrage, la mise à jour du système, l'installation du SE^E, le téléchargement de la politique et le redémarrage de la VM (suite aux mises à jour) ;
2. Initialisation de N_T triplets de VM Client / Apache / MySQL ;
3. L'application de la politique de sécurité par le SE^E ;
4. Les simulations de dysfonctionnements :
   (a) L'arrêt de SELinux : DAC remplace SELinux ;
   (b) L'arrêt d'iptables : firewalld remplace iptables ;
   (c) L'arrêt d'OpenVPN : des propriétés ne sont plus appliquées ;
   (d) L'arrêt de firewalld : des propriétés ne sont plus appliquées.

Les simulations de dysfonctionnements sont faites par un script SSH se connectant sur chacune des machines (en commençant par la VM Proxy) pour exécuter les commandes stoppant les mécanismes.

6.4.1 Taille de la politique

La taille de la politique est évaluée à partir du nombre de propriétés et de contextes qu'elle contient.

Machine	**Nombre**		
	Propriétés	**Contextes**	**Total**
Hôte physique	4	8	12
VM Proxy	14	17	31
VM Apache	22	25	47
VM MySQL	15	18	33
VM Client	3	7	10

TABLE 6.3 – Taille des politiques

On peut voir qu'il est possible de sécuriser des applications ou un cas d'usage avec des politiques de petite taille, puisque la plus grande politique comporte seulement 22 propriétés. Cela est notamment dû à l'utilisation des classes, qui permettent d'inclure des propriétés afin de correspondre à un besoin de sécurité classique.

6.4.2 Performances

Nous détaillons tout d'abord les performances obtenues pour l'initialisation des VM et pour l'application de la politique.

6.4.2.1 Initialisation des VM

La figure 6.5 présente le temps moyen nécessaire pour que les VM soient entièrement initialisées, en fonction du nombre N_T de triplets de VM. Ce temps est fourni par OpenStack pour les machines créées à partir des images fournies par la plateforme. La VM client n'étant pas créée à partir d'une image officielle, son temps d'initialisation n'est pas indiqué.

On remarque que le temps d'initialisation des VM Apache et MySQL augmente avec le nombre de triplets de VM instanciés. Ce comportement s'explique par deux raisons :
– plus le nombre de VM instanciées est important, plus les ressources physiques sont sollicitées : les VM ont donc un accès limité aux ressources, ce qui les rend nécessairement moins performantes ;

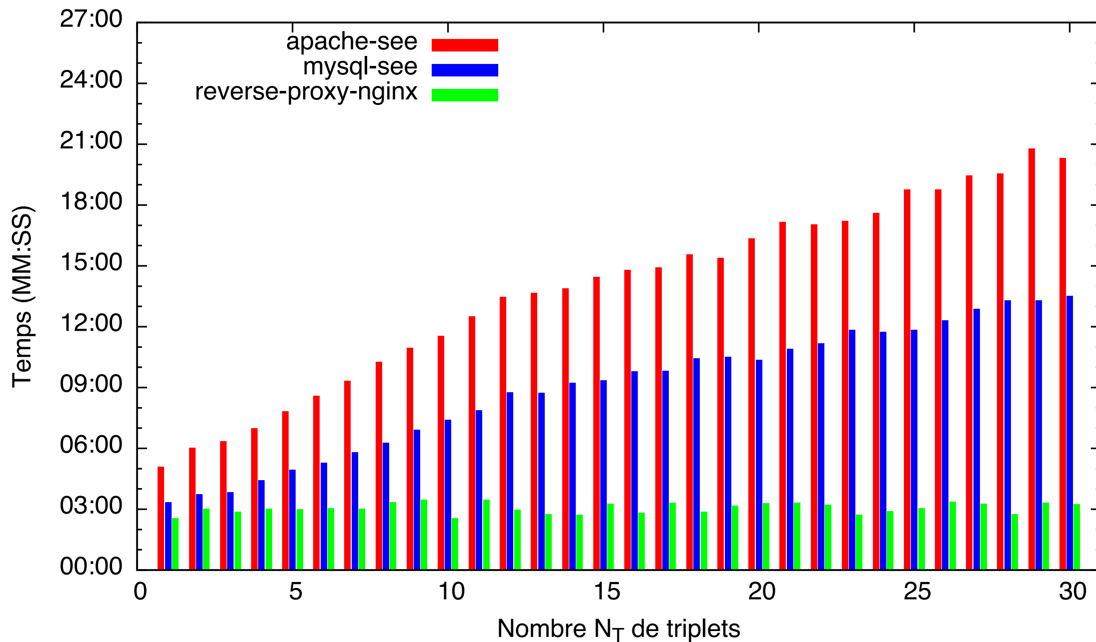

FIGURE 6.5 – Temps de lancement des VM

– plus le nombre de VM instanciées est important, plus le temps de récupération des paquets lors de la mise à jour du système est élevé (miroir local, mais réseau Gigabit sous-dimensionné).

Cependant, cette augmentation est indépendante de la présence du SE^E, puisque seule l'installation du SE^E est faite à cette étape (l'application de la politique n'est pas incluse ici). L'augmentation est donc liée aux limitations de notre environnement de test.

On note également que le temps d'initialisation de la VM Proxy varie peu. En effet, la VM Proxy est toujours instanciée avant les autres VM : elle peut ainsi être entièrement instanciée avant que les ressources physiques et la bande passante soient surchargées.

6.4.2.2 Application de la politique par le SE^E

La figure 6.6 présente le temps d'application de la politique de sécurité par le SE^E sur chacune des VM, en fonction du nombre N_T de triplets lancés. Les temps présentés ici comprennent à la fois le temps de compilation de la politique (la sélection des mécanismes) et le temps de projection (la configuration des mécanismes).

L'application de la politique sur la VM client prend très peu de temps : cela s'explique principalement par la non-utilisation de mécanismes coûteux. En effet, la politique sur la VM Client est appliquée en utilisant OpenVPN, DAC et iptables : ces mécanismes sont rapides à configurer.

Les politiques des autres VM nécessitent plus de temps afin d'être appliquées. En effet, l'ensemble de ces autres politiques utilisent SELinux afin d'appliquer les propriétés de contrôle d'accès système. Or, SELinux est un mécanisme lent à configurer (principalement en raison du temps nécessaire au chargement d'un nouveau module par SELinux), ce qui ralentit l'application de la politique, indépendamment du SE^E. Cela explique donc les temps d'application de 2 et 4 minutes pour un seul triplet de VM. De plus, on remarque que, pour $N_T = 1$, le temps d'application de la politique pour la VM Apache est le double

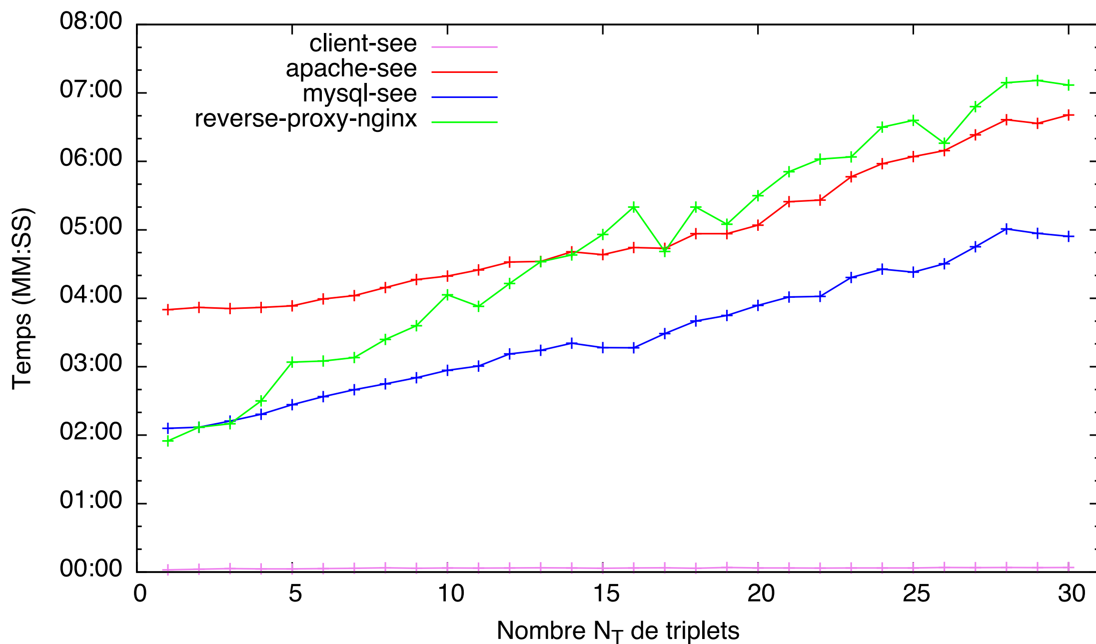

FIGURE 6.6 – Temps d'application de la politique par le SE^E

de celui pour les VM Proxy et MySQL. Or, la politique de la VM Apache contient deux domaines applicatifs à isoler, alors que les deux autres VM n'en contiennent qu'un. On peut donc facilement en déduire que le temps d'application de la politique pour une VM est directement lié au temps nécessaire à SELinux pour charger un nouveau module.

L'augmentation du temps d'application avec le nombre de triplets de VM s'explique quant à elle par la sollicitation plus importante des ressources physiques (comme cela a été vu pour l'initialisation des VM). En effet, plus le nombre de VM instanciées est important, plus les ressources pour chaque VM sont limitées.

Notons que l'augmentation du temps d'application est plus importante pour la VM Proxy que pour les VM Apache et MySQL. En effet, la politique de la VM Proxy évolue avec le nombre de triplets : pour chaque triplet supplémentaire, trois propriétés de confidentialité réseau sont ajoutées afin de garantir la confidentialité des communications entre les VM de ce triplet. Or, les propriétés réseaux nécessitent des communications entre les différents SE^E, afin de choisir le mécanisme à utiliser et le port sur lequel le serveur écoute.

6.4.3 Évaluation de l'application de la politique

Pour étudier l'évolution de l'application de la politique de sécurité, nous considérons cinq étapes : l'initialisation de l'application, suivi par la désactivation de quatre mécanismes :

1. **Init** : il s'agit de la première application de la politique de sécurité. Tous les mécanismes sont fonctionnels et la meilleure application possible est donc choisie ;
2. **Test 1** : la première erreur est détectée (normalement, SELinux) ;
3. **Test 2** : la deuxième erreur est détectée (normalement, iptables) ;
4. **Test 3** : la troisième erreur est détectée (normalement, OpenVPN) ;
5. **Test 4** : la quatrième erreur est détectée (normalement, firewalld).

Cette section présente tout d'abord les scores théoriques obtenus au cours de ces quatre étapes, puis les scores réellement obtenus lors de l'expérimentation. Nous verrons que les scores obtenus peuvent différer des scores attendus, en raison d'une désactivation des mécanismes faite dans un ordre différent de celui prévu. Les taux de couverture de la politique et des propriétés seront également présentés.

6.4.3.1 Évolution théorique du score d'application

La figure 6.7 présente les scores théoriques d'application qui sont attendus lorsque les différentes désactivations de mécanismes sont effectuées dans l'ordre prévu.

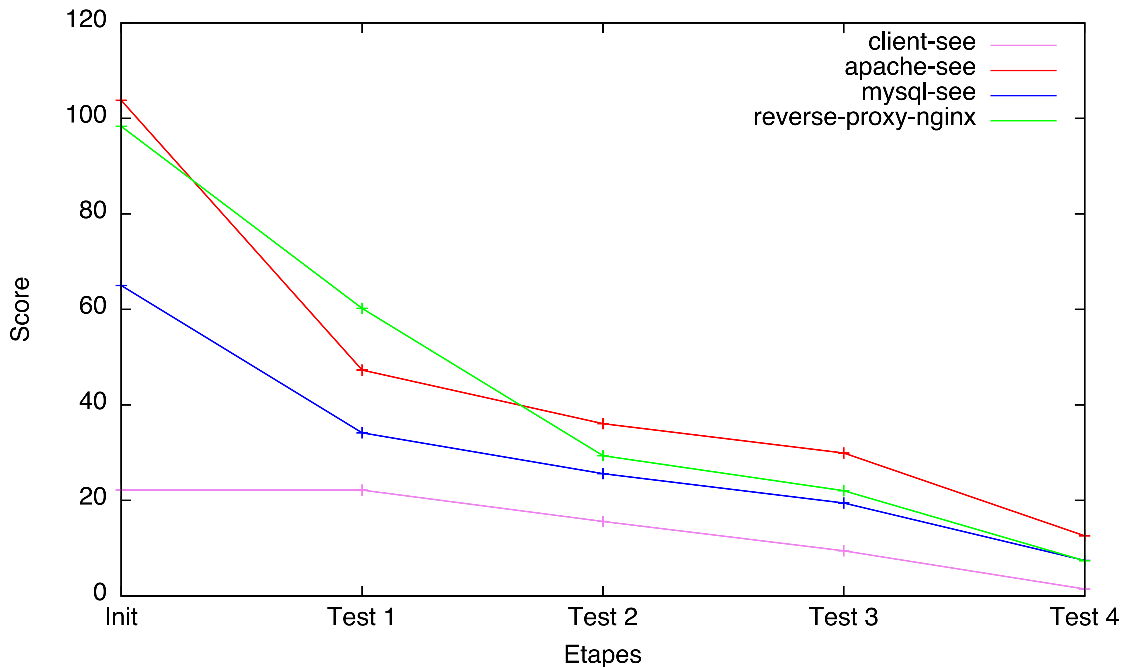

FIGURE 6.7 – Score d'application théorique de la politique par le SE^E

Le premier test correspond à la désactivation de SELinux. Les propriétés de contrôle d'accès système sont donc réappliquées en utilisant DAC, ce qui diminue le score d'application. Notons que la VM Client n'utilisant pas SELinux, le score théorique d'application de sa politique n'est pas modifié lors de ce test.

Le second test correspond à la désactivation d'iptables. Les propriétés utilisant ce mécanisme sont donc réappliquées en utilisant firewalld : cela inclut les propriétés d'accès, mais également les propriétés de confidentialité réseau (le module d'extension OpenVPN fait appel à des capacités de pare-feu).

Le troisième test est l'arrêt d'OpenVPN. Les propriétés de confidentialité réseau ne sont donc plus appliquées et une alerte est remontée par le SE^E à l'administrateur de l'architecture logicielle.

Enfin, le dernier test correspond à la désactivation de firewalld. Les propriétés d'accès ne sont donc plus appliquées. Le score d'application final obtenu correspond donc aux scores d'application des propriétés de contrôle d'accès système (appliquées par DAC), d'authentification (PAM) et d'assurance (Assurance Engine).

Notons que le score théorique présenté pour la VM Proxy correspond au cas où un seul triplet de VM est lancé : en effet, chaque triplet supplémentaire ajoute une propriété de

confidentialité réseau, ce qui modifie le score théorique pour la VM Proxy.

6.4.3.2 Évolution réelle du score d'application

La synthèse des scores d'application réels obtenus est présentée à la table 6.4. Cette table présente donc, pour chaque VM de type Client, Apache et MySQL, les scores théoriques attendus à chacune des étapes de tests et les scores réels obtenus (pour chaque score réel, le nombre de fois où il est obtenu est également donné). Les tests ayant été lancés pour tous les nombres de triplets entre 1 et 30, le nombre total de VM de chaque type créé est 465. Dans le cas de la VM Proxy, les scores obtenus sont donnés pour les tests avec 1, 10, 20 et 30 triplets de VM. Ces résultats sont détaillés dans la suite de cette section.

		Init	**Test 1**	**Test 2**	**Test 3**	**Test 4**
Apache	Théorique	103	47	36	29	12
	Réel	103 (465)	47 (214) 95 (251)	36 (169) 38 (296)	29 (465)	12 (465)
MySQL	Théorique	64	34	25	19	7
	Réel	64 (465)	34 (281) 56 (184)	25 (451) 20 (14)	19 (465)	7 (465)
Client	Théorique	22	22	15	9	1
	Réel	22 (465)	22 (465)	15 (465)	9 (465)	1 (465)
Proxy	$N_T = 1$	98	67	52	22	7
	$N_T = 10$	441	410	326	22	7
	$N_T = 20$	823	792	630	22	7
	$N_T = 30$	1204	1173	934	22	7

TABLE 6.4 – Scores d'application des politiques par VM et par étape de test

VM Client, Apache et MySQL La figure 6.8 présente les scores d'application de la politique sur les VM Client, Apache et MySQL. Pour comparaison, les scores théoriques pour ces VM sont également rappelés (en pointillé).

On peut remarquer que, pour l'ensemble de ces VM, le score initial réel correspond au score théorique attendu : tous les mécanismes sont donc bien fonctionnels lors de la première application de la politique. Cependant, dans le cas des VM Apache et MySQL, on note une différence significative entre les scores attendus et obtenus pour le premier test et, dans une moindre mesure, pour le deuxième test. Ces scores s'expliquent par le fait que les mécanismes sont désactivés dans un ordre différent de celui attendu.

Considérons le cas du premier test pour la VM Apache. En se basant sur la table 6.4 donnant les scores réels pour les VM et le nombre de fois où ces scores apparaissent, on voit que deux scores sont obtenus :

- Score de 47 : c'est le score attendu, c'est-à-dire le score après désactivation de SELinux et réapplication des propriétés de contrôle d'accès système par DAC ;
- Score de 95 : c'est le score obtenu après désactivation d'OpenVPN : le VPN a donc été désactivé avant SELinux. Cela s'explique par un effet de bord du protocole de test. En effet, pour désactiver les mécanismes, un script exécute les commandes nécessaires en se connectant par SSH sur chaque machine (en commençant par la VM Proxy). La désactivation du VPN sur la VM Proxy peut donc désactiver le VPN sur les autres VM avant SELinux, expliquant ainsi la différence entre les scores attendus et réels.

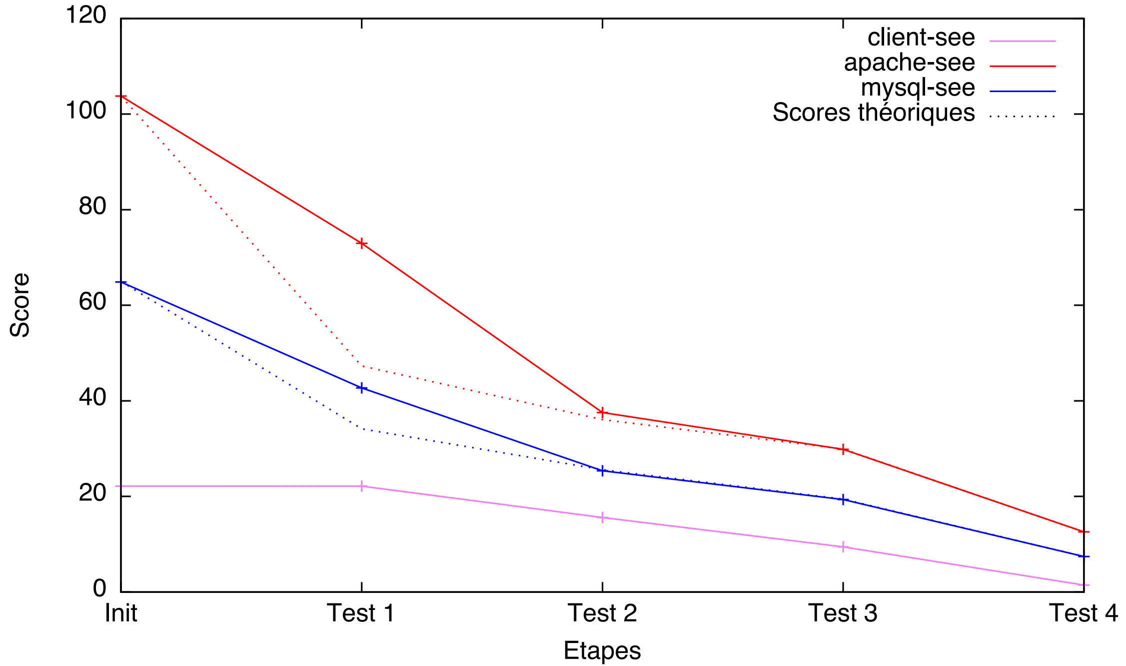

FIGURE 6.8 – Score d'application réel de la politique par le SE^E

La différence au niveau du deuxième test s'explique de manière similaire : le score de 38 est obtenu lorsque SELinux et OpenVPN ont été désactivés (mais pas iptables). Le nombre de scores différents du score attendu augmente puisqu'il comprend les VM pour lesquelles l'effet de bord lié à la désactivation du VPN sur la VM Proxy a eu lieu avant la désactivation de SELinux, et celles pour lesquelles il a eu lieu entre les désactivations de SELinux et d'iptables. Cependant, dans tous les cas, les scores d'applications réels convergent bien vers le score d'application attendu. Les scores permettent donc, comme attendu, de refléter l'état d'application de la politique.

La figure 6.9 présente l'évolution du taux de couverture T_C et du taux de couverture des propriétés T_P.

On peut ainsi voir que T_C diminue rapidement au cours des différentes étapes de test, en raison de l'utilisation de mécanismes ayant des scores plus faibles que ceux originellement sélectionnés. En revanche, le taux T_P de couverture des propriétés reste proche de 1 jusqu'au troisième test : on voit ainsi que la possibilité d'appliquer des propriétés avec des mécanismes différents (grâce au caractère abstrait des capacités) permet d'adapter l'application de la politique à l'environnement. Ainsi, même en cas de dysfonctionnement d'un ou plusieurs mécanismes, la politique continue à être appliquée tant que des mécanismes sont disponibles.

VM Proxy La figure 6.10 présente les scores d'application réels obtenus pour la VM Proxy en fonction du nombre N_T de triplets de VM qui ont été créés.

La politique de sécurité de la VM Proxy (annexe B) peut être vue comme :

- Une propriété d'isolation système, trois de confidentialité système par contrôle d'accès, une d'intégrité système, deux d'authentification, quatre d'accès et une d'assurance ;
- N_T fois trois propriétés de confidentialité réseau (avec les VM Client, Apache et MySQL).

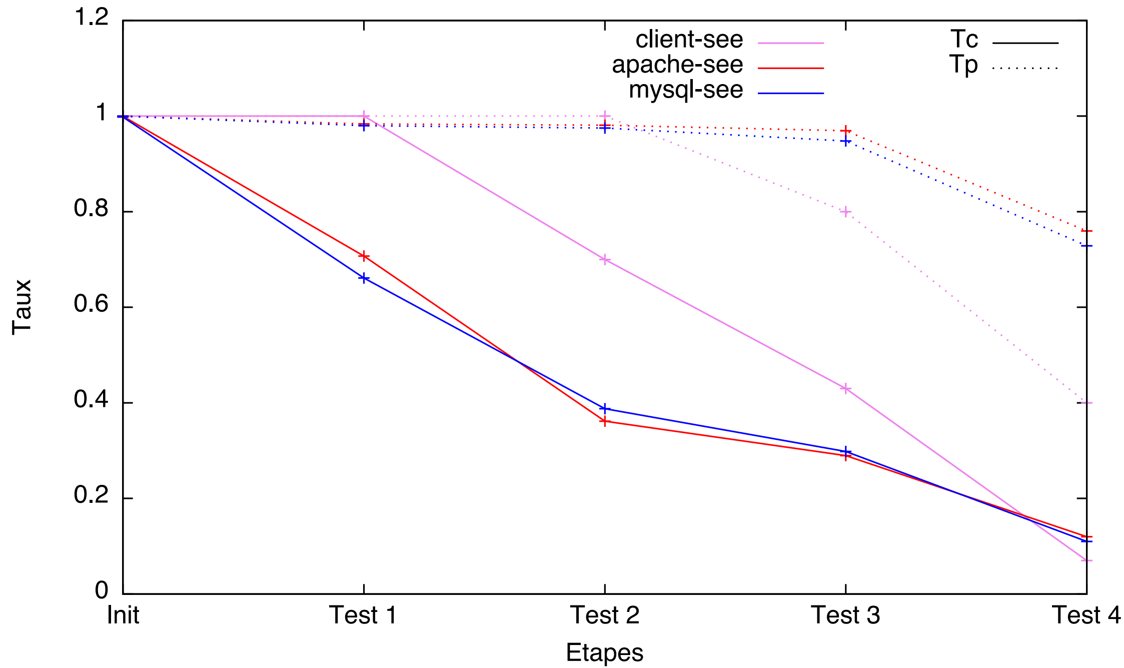

FIGURE 6.9 – Taux de couverture de la politique

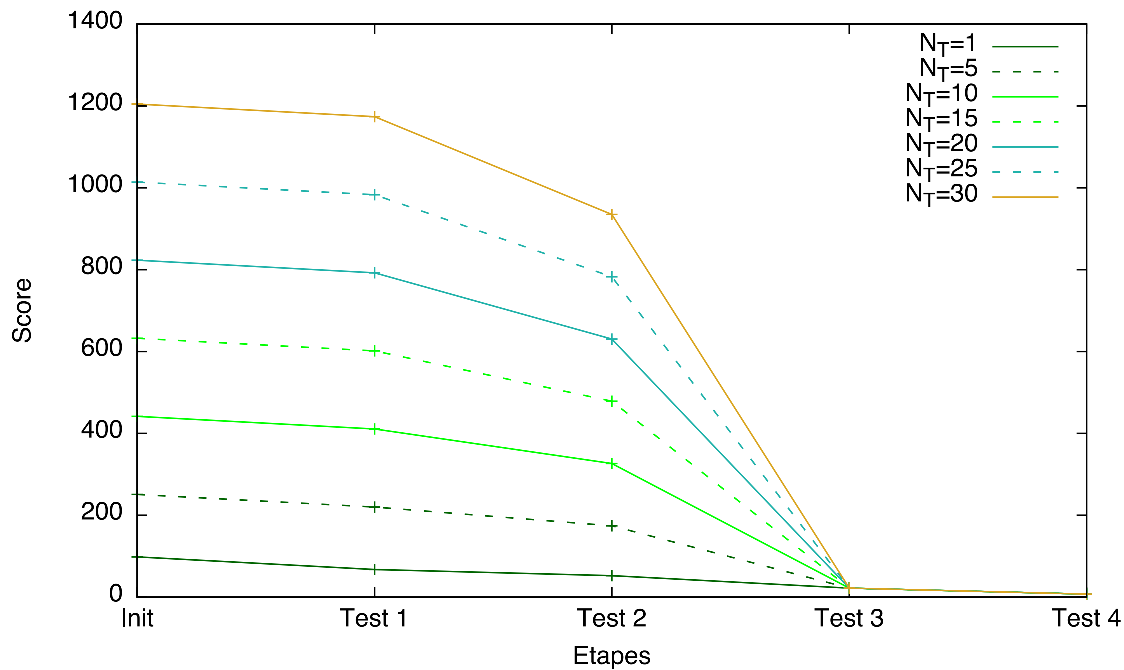

FIGURE 6.10 – Score d'application réel pour la VM Proxy en fonction de N_T

On voit donc que le score initial augmente avec le nombre de triplets, ce qui s'explique par l'augmentation du nombre de propriétés à appliquer. Lors des deux premières étapes de test, le score diminue faiblement dû au changement de mécanismes. Cependant, le score est fortement réduit lors de la troisième étape (désactivation de OpenVPN) puisque toutes les propriétés de confidentialité réseau cessent d'être appliquées. De plus, aux étapes 3 et 4, les scores d'applications sont indépendants de N_T : les propriétés réseaux ne pouvant

plus être appliquées, l'application de la politique n'est plus liée au nombre de triplets de VM. Notons que, les tests de désactivations de mécanismes étant toujours effectués en commençant par la VM Proxy, il n'y a pas d'effet de bord sur les scores comme c'est le cas pour les autres VM.

La figure 6.11 présente les taux T_C et T_P pour la VM Proxy dans les cas de 5, 10, 20 et 30 triplets de VM.

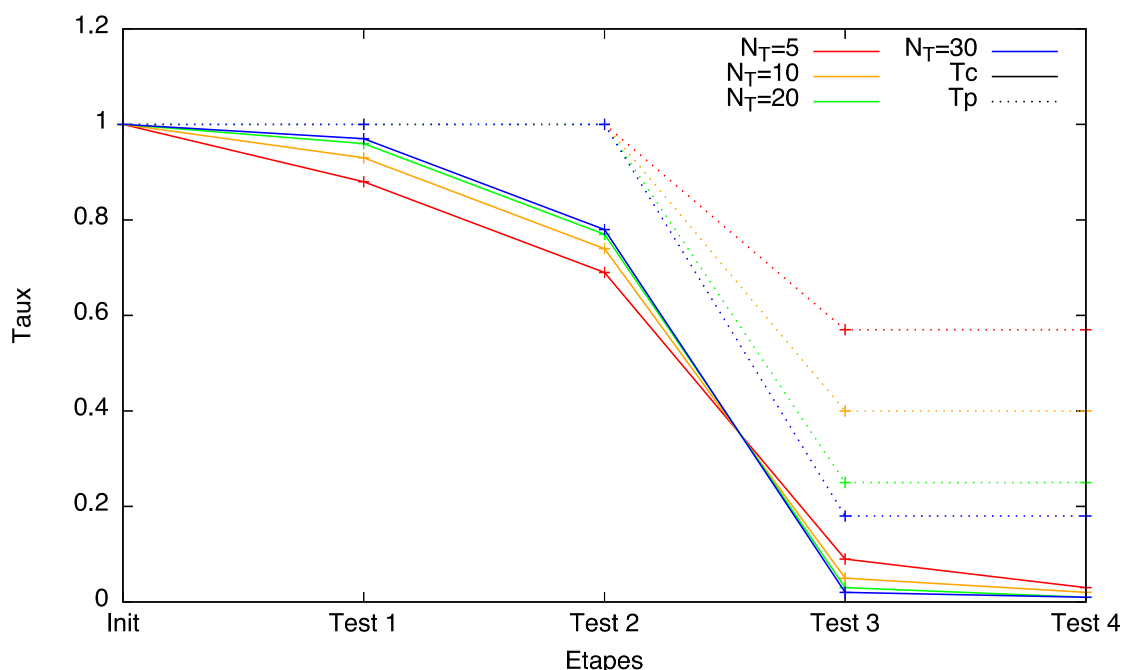

FIGURE 6.11 – Taux de couverture de la politique de la VM Proxy

On peut voir que l'évolution des taux est similaire quelque soit le nombre de triplets de VM. Cependant, plus le nombre de triplets est important, plus les taux T_C et T_P sont affectés. Cela s'explique par le fait que les propriétés les plus affectées par les désactivations de mécanismes sont celles de confidentialité réseau. Or, le nombre de ces propriétés augmente avec le nombre de triplets.

6.4.3.3 Tests

Cette section présente des tests effectués afin de vérifier l'application de la politique, notamment en fonction des mécanismes utilisés. Elle permet d'illustrer les tests réalisés par le SE^E.

Propriété de confidentialité système Considérons tout d'abord les propriétés de confidentialité système pour les fichiers de configuration du serveur Web sur la VM Apache. Ces propriétés sont tout d'abord appliquées en utilisant SELinux. Le listing 6.26 donne le résultat des tentatives de lecture d'un fichier de configuration du serveur Web par les utilisateurs `tenant-admin` (action autorisée par la politique), `tenant-operator` et `root`.

```
1 [tenant-admin@apache-see ~]# cat /etc/httpd/conf/httpd.conf
2 [...] # Le contenu du fichier est affiche
3
4 [tenant-operator@apache-see ~]# cat /etc/httpd/conf/httpd.conf
5 cat: /etc/httpd/conf/httpd.conf: Permission denied
```

```
6
7 [root@apache-see ~]# cat /etc/httpd/conf/httpd.conf
8 cat: /etc/httpd/conf/httpd.conf: Permission denied
```

Listing 6.26 – Lecture d'un fichier confidentiel protégé avec SELinux

La propriété est bien respectée puisque seul l'utilisateur `tenant-admin` peut lire le fichier (ligne 2), alors que les tentatives de `tenant-operator` et `root` échouent (lignes 5 et 8).

Le premier test effectué désactive SELinux et la propriété est alors réappliquée en utilisant les droits DAC Unix. Le listing 6.27 présente les résultats obtenus.

```
1 [tenant-admin@apache-see ~]# cat /etc/httpd/conf/httpd.conf
2 [...] # Le contenu du fichier est affiche
3
4 [tenant-operator@apache-see ~]# cat /etc/httpd/conf/httpd.conf
5 cat: /etc/httpd/conf/httpd.conf: Permission denied
6
7 [root@apache-see ~]# cat /etc/httpd/conf/httpd.conf
8 [...] # Le contenu du fichier est affiche
```

Listing 6.27 – Lecture d'un fichier confidentiel protégé avec DAC

On observe que la propriété est correctement appliquée pour les utilisateurs `tenant-admin` (ligne 2) et `tenant-operator` (ligne 5). En revanche, l'effet attendu n'est pas obtenu pour l'utilisateur `root` (ligne 8) : en effet, celui-ci passe outre les droits DAC et parvient à lire le fichier de configuration du serveur Web. Ainsi, bien que la propriété soit toujours appliquée, la qualité de l'application est moindre (ce qui explique que DAC ait un score d'ordonnancement plus faible que SELinux).

Propriété de confidentialité réseau Le listing 6.28 présente le résultat de l'application des propriétés de confidentialité réseau. Ces propriétés ont pour objectif de permettre l'accès au serveur Web et à la base de données pour les utilisateurs autorisés (c'est-à-dire les utilisateurs connectés sur la VM Client du même triplet de VM) et de l'interdire pour les autres utilisateurs.

```
1 [user@client-1 ~]$ wget apache-see-1/site1
2 --2015-09-16 10:39:49--  http://apache-see-1/site1
3 Connecting to apache-see-1:80... connected.
4 HTTP request sent, awaiting response... 200 OK
5 Length: 25 [text/html]
6 Saving to: 'site1.1'
7
8 100%[=================================================>] 25    --.-K/s   in 0s
9
10 2015-09-16 10:39:49 (2.81 MB/s) - 'site1.1' saved [25/25]
11
12 [user@client-1 ~]$ wget apache-see-2/site2
13 --2015-09-16 10:40:16--  http://apache-see-2/site2
14 Connecting to apache-see-2:80... failed: Connection timed out.
15 Retrying.
16 [...]
```

Listing 6.28 – Accès à un serveur Web depuis les VM Client

Ainsi, la première partie du listing (lignes 1 à 10) contient le résultat de l'accès à la VM `apache-see-1` (VM Apache du premier triplet) par l'utilisateur `user` depuis la VM `client-1` (VM Client du premier triplet). Cet accès est autorisé par la politique et

l'utilisateur peut donc obtenir les fichiers du serveur Web (ligne 10). La seconde partie du listing (lignes 12 à 16) donne le résultat d'une tentative d'accès depuis une VM Client vers une VM Apache d'un triplet de VM différent (`apache-see-2`). Cette opération n'est pas autorisée par la politique qui indique qu'un serveur Web n'est accessible que par la VM Client correspondante. On voit donc que l'accès échoue (ligne 14).

Le listing 6.29 présente quant à lui le résultat de l'accès à un serveur Web après le test de désactivation d'OpenVPN. Les propriétés de confidentialité réseau ne sont donc plus appliquées et le tunnel VPN ayant été mis en place entre les VM par le SE^E n'est plus présent.

```
1 [user@client-1 ~]$ wget apache-see-1/site1
2 --2015-09-16 10:49:14--  http://apache-see-1/site1
3 Connecting to apache-see-1:80... failed: Invalid argument.
```

Listing 6.29 – Accès à un serveur Web depuis les VM Client (VPN arrêté)

On peut observer que le serveur Web n'est plus accessible depuis les VM Client (ligne 3), et ce même pour celles autorisées par la politique de sécurité.

Propriété d'accès Dans ce cas d'usage, les propriétés d'accès sont utilisées pour gérer les règles de pare-feu. Le listing 6.30 présente le résultat d'un scan réseau réalisé avec `nmap` sur une VM Apache depuis les VM Proxy (lignes 1 à 10) et Client (lignes 12 à 22).

```
1  [root@reverse-proxy-nginx ~]# nmap apache-see-1
2  Starting Nmap 6.40 ( http://nmap.org ) at 2015-09-16 14:49 UTC
3  Nmap scan report for apache-see-1
4  Host is up (0.0013s latency).
5  Not shown: 998 filtered ports
6  PORT   STATE SERVICE
7  22/tcp open  ssh
8  80/tcp open  http
9
10 Nmap done: 1 IP address (1 host up) scanned in 4.57 seconds
11
12 [root@client-0 user]# nmap apache-see-1
13 Starting Nmap 6.40 ( http://nmap.org ) at 2015-09-16 10:47 EDT
14 Nmap scan report for apache-see-1
15 Host is up (0.0010s latency).
16 Not shown: 998 filtered ports
17 PORT     STATE SERVICE
18 80/tcp   open  http
19 9999/tcp open  abyss
20 MAC Address: FA:16:3E:59:B8:3D (Unknown)
21
22 Nmap done: 1 IP address (1 host up) scanned in 4.92 seconds
```

Listing 6.30 – Scan réseau d'une VM Apache

On observe donc que les ports 22 et 80 sont ouverts pour la VM Proxy (lignes 7 et 8) et les ports 80 et 9999 (lignes 18 et 19) pour la VM Client. Cela correspond aux règles d'accès établies sur la VM Apache autorisant l'accès au serveur Web depuis toutes les sources, l'accès à l'application ApacheGUI depuis une VM Client, et l'accès SSH depuis la VM Proxy. Ces résultats sont obtenus après la première application des propriétés (avec iptables) et à leur seconde application (avec firewalld).

Nous avons simulé dans cette section des tests effectués par le SE^E. Ces tests sont automatiquement générés et exécutés par le SE^E lors de la phase d'assurance. Notons que tous les tests normalement effectués par le SE^E dans ce cas d'usage n'ont pas été

présentés ici. Cette simulation d'indisponibilité des mécanismes a ainsi permis d'illustrer la réapplication automatique de la politique par le SE^E.

6.5 Conclusion

Ce chapitre a présenté une expérimentation généralisant un cas d'usage du projet Seed4C [Bousquet *et al.*, 2015]. D'autres cas d'usages industriels ont été traités dans le cadre du projet : les politiques exprimant les besoins de sécurité des différents cas ont été définies et appliquées à l'aide du SE^E et des différents modules d'extensions implémentés.

Nous avons donc tout d'abord décrit l'environnement de test dont nous disposions ainsi que le cas d'usage. Puis, nous avons montré comment définir la politique à partir de l'expression des besoins de sécurité. Nous avons alors pu voir que la politique obtenue est de taille réduite, en raison de l'utilisation de propriétés répondant à des besoins de sécurité génériques et de haut niveau.

Nous avons également présenté l'application de la politique et la génération des tests d'assurance. L'application de chaque propriété est donc faite, comme attendu, en fonction des ressources mises en jeu et des mécanismes disponibles sur la machine. Les tests d'assurance générés sont dépendants soit des mécanismes utilisés, soit des propriétés à appliquer : il est ainsi possible de détecter un dysfonctionnement de mécanisme ou une mauvaise application d'une propriété (indépendamment des mécanismes utilisés).

Finalement, nous avons détaillé les résultats des expérimentations. Nous avons tout d'abord étudié les performances obtenues pour l'application de la politique et nous avons observé que le temps d'application est directement lié aux mécanismes utilisés. Ainsi, le temps de compilation de la politique par le SE^E est négligeable par rapport au temps de configuration des mécanismes, qui est indépendant du SE^E. L'amélioration des performances du SE^E implique soit l'amélioration des performances des mécanismes (solution complexe, nécessitant des coopérations avec les communautés), soit l'utilisation de mécanismes plus rapides (en les favorisant lors de la sélection à l'aide des scores d'ordonnancement).

Des tests de réapplication ont également été effectués : des défaillances de mécanismes ont été simulées afin de provoquer la réapplication de certaines propriétés avec d'autres mécanismes disponibles. Nous avons donc vu que ces défaillances sont bien détectées par le SE^E et que celui-ci tente de réappliquer les propriétés de sécurité tant que des mécanismes sont disponibles pour fournir les capacités nécessaires. Ces reconfigurations impactent le score d'application de la politique, puisque l'application s'éloigne de celle idéalement souhaitée par l'expert en sécurité. Cependant, le taux de couverture des propriétés montre que l'indépendance des propriétés vis-à-vis des mécanismes permet de maintenir une application presque totale de la politique, même après l'arrêt de plusieurs mécanismes.

Ce chapitre et cette expérimentation ont permis d'illustrer l'utilisation du langage d'expression des besoins (chapitre 3) sur un cas d'usage réel. Cela a ainsi montré que le langage est adapté à une utilisation dans le cadre d'une architecture en nuage. De plus, l'architecture (chapitre 4) et son implémentation (chapitre 5) ont été éprouvées. Ce chapitre a donc permis de montrer l'efficacité du SE^E lors de l'application et de l'assurance d'une politique, et sa capacité à s'adapter aux mécanismes disponibles.

Chapitre 7

Conclusion

La contribution principale de cette thèse est la définition d'une solution de protection des environnements hétérogènes tels que l'informatique en nuage. La solution proposée se compose d'un langage orienté propriété permettant d'exprimer les besoins de sécurité, et d'une architecture pouvant les appliquer et mettre à jour leur application si nécessaire. Le langage proposé permet à un utilisateur non-expert en sécurité de définir une politique adressant ses différents besoins. Le langage prend en compte à la fois l'application et l'assurance des propriétés considérées. L'architecture peut quant à elle appliquer, assurer et réappliquer automatiquement la politique. De plus, l'application de la politique peut évoluer de façon autonome et transparente si cela s'avère nécessaire. Ainsi, le système proposé permet à la fois d'exprimer les besoins de sécurité et de les appliquer dynamiquement en utilisant les meilleurs mécanismes disponibles.

Dans le chapitre 2, nous avons présenté l'état de l'art. Nous avons donc introduit le concept d'informatique en nuage qui constitue le contexte d'étude de cette thèse. Nous avons également présenté les propriétés de sécurité usuelles, ainsi que les différents modèles et langages permettant d'exprimer une politique de sécurité. De plus, des modèles d'architectures et leurs applications à la sécurité ont été introduits. Ainsi, nous avons vu que de nombreux travaux permettent d'exprimer ou d'appliquer une politique de sécurité, mais ces travaux ne gèrent qu'un sous-ensemble de propriétés et ne visent que certains types de ressources. En outre, de nombreux mécanismes existants permettent de couvrir différents sous-ensembles de propriétés : il est donc possible de les utiliser pour appliquer une politique de sécurité globale. Cet état de l'art nous a permis d'établir l'absence d'une solution pouvant exprimer les besoins de sécurité indépendamment de leur méthode d'application et les appliquer en réutilisant des mécanismes existants afin d'obtenir une meilleure couverture.

Le chapitre 3 a défini un langage d'expression des besoins de sécurité répondant aux caractéristiques spécifiques de l'informatique en nuage. Deux vues sont possibles pour ce langage. La première vue est celle offerte à un expert en sécurité : elle lui permet d'administrer les définitions des propriétés à partir de capacités. Ces capacités sont la solution choisie pour abstraire les fonctionnalités des mécanismes de sécurité et obtenir des propriétés indépendantes des mécanismes sous-jacents. La seconde vue est utilisée par l'administrateur de l'architecture logicielle. Il peut ainsi définir une politique de sécurité en utilisant les prototypes des propriétés qui mettent en jeu des contextes. Les contextes sont des représentations abstraites des ressources réelles, qui permettent de s'abstraire du système d'exploitation ou des applications utilisés. La politique est donc indépendante à la fois des mécanismes et du système, puisque les liens capacités/mécanismes et contextes/ressources n'impactent pas la définition des propriétés. Les contraintes entre les différents mécanismes peuvent être

exprimées par l'expert en sécurité, grâce à trois critères : la compatibilité (locale et réseau), la dépendance et la récessivité. De plus, l'expert en sécurité peut personnaliser la sélection des mécanismes pour une capacité, grâce au score d'ordonnancement attribué à chaque paire mécanisme/capacité. Ce score permet alors d'évaluer la qualité de l'application de la politique. Finalement, le langage est extensible : de nouvelles propriétés peuvent être définies, des capacités peuvent être ajoutées, et des mécanismes additionnels peuvent être supportés.

Dans le chapitre 4, nous avons présenté une architecture permettant d'appliquer et d'assurer une politique de sécurité définie dans le langage d'expression des besoins. Cette architecture propose l'utilisation d'une architecture multi-agents pour appliquer la politique de sécurité sur une architecture logicielle. Chaque agent, appelé SE^E (*Security Enforcement Engine*), est un agent autonome chargé de compiler la politique et de la projeter sur les mécanismes disponibles sur la machine. Ce chapitre décrit les algorithmes permettant de sélectionner l'ensemble des mécanismes à utiliser pour appliquer une propriété. La sélection est configurable à travers trois modes de sélection : `Défaut` (pour chaque capacité, le mécanisme le plus adapté est choisi), `Maximum` (pour chaque capacité, on choisit l'ensemble de mécanismes compatibles le plus adapté) ou `BestEffort` (le premier mécanisme compatible est choisi). Les algorithmes correspondant au processus d'assurance sont également donnés. Ils définissent le comportement du SE^E lorsqu'une erreur ou un dysfonctionnement est détecté. Ainsi, si le SE^E détecte que l'un des mécanismes utilisé n'est pas fonctionnel ou qu'une propriété ne produit pas l'effet escompté, il tente d'appliquer à nouveau les propriétés en utilisant d'autres mécanismes. Enfin, une méthode d'évaluation de la qualité de l'application de la politique est introduite, en se basant notamment sur les scores d'ordonnancement des mécanismes.

Le chapitre 5 décrit l'implémentation du SE^E. Celle-ci peut compiler l'ensemble du langage d'expression des besoins et implémente les différents algorithmes d'application et d'assurance détaillés au chapitre 4. Le chapitre explicite également le processus de configuration des mécanismes en fonction des capacités : le SE^E utilise pour cela des modules d'extension. Un module d'extension est associé à un mécanisme spécifique et peut soit projeter une capacité sur ce mécanisme, en le configurant ou en interagissant avec lui, soit récupérer des informations d'assurance ou effectuer des tests. Le chapitre montre comment de nouveaux modules peuvent être intégrés, ce qui permet d'ajouter le support de nouveaux mécanismes. Plusieurs modules ont ainsi été développés (notamment par des partenaires du projet Seed4C) et offrent par exemple des fonctionnalités de chiffrement, de contrôle d'accès, d'authentification ou d'accès distant. Finalement, le chapitre détaille toutes les étapes de la compilation, de la projection, et de l'assurance d'une politique simple.

Enfin, l'expérimentation présentée au chapitre 6 montre comment définir une politique à partir des besoins de sécurité. Elle décrit également l'application et l'assurance de cette politique. De plus, dans le cadre de l'expérimentation, des dysfonctionnements de mécanismes sont provoqués afin de valider le processus de réaction du SE^E : les propriétés sont réappliquées tant que d'autres mécanismes sont disponibles. Ainsi, l'indépendance des propriétés vis-à-vis des mécanismes permet de maintenir une protection du système, même si celle-ci est dégradée par rapport à la protection initiale. Enfin, des tests sont effectués pour vérifier l'efficacité de la protection mise en place : ils correspondent aux tests effectués automatiquement par le SE^E afin de détecter une application n'ayant pas l'effet escompté. Ce chapitre démontre ainsi l'efficacité du langage d'expression des besoins dans le cadre de l'informatique en nuage. En effet, le langage permet de couvrir un large panel de propriétés afin de sécuriser une architecture de bout en bout. De plus, ce chapitre montre également que la définition d'une politique répondant à l'ensemble des besoins reste simple et com-

pacte. Enfin, il démontre l'efficacité de l'implémentation et des algorithmes de projection dans le cas d'une architecture réelle.

Perspectives

A la suite des travaux effectués au cours de cette thèse, plusieurs axes de perspectives peuvent être considérés.

Vérification de la politique

Actuellement, la politique doit être entièrement définie par l'administrateur de l'architecture logicielle. Cependant, ce processus de définition peut se révéler complexe et entraîner des erreurs. Il serait donc intéressant de pouvoir garantir la validité de la politique définie.

Tout d'abord, un processus de vérification formelle pourrait être mis en place, afin de s'assurer que la politique ne contient pas de règles contradictoires. Des travaux existants [Lefray *et al.*, 2013b] permettent déjà de vérifier la cohérence d'une politique de sécurité et des liens pourraient donc être envisagés avec notre langage.

En supplément de la vérification de la cohérence entre les règles, il serait également possible d'améliorer la cohérence de la politique vis-à-vis des besoins du client. Cela pourrait par exemple être fait en utilisant les méthodes existantes de détermination des besoins de sécurité (CCM, CAIQ, etc.) et en vérifiant que les besoins ainsi obtenus correspondent à ceux exprimés par la politique.

Enfin, des solutions pourraient être envisagées pour éviter les redondances au sein de la politique (des propriétés exprimées plusieurs fois ou incluses dans d'autres propriétés). En effet, de telles redondances peuvent entraîner des problèmes d'application (par exemple du sur-chiffrement) ou impacter inutilement les performances du système.

Gestion de la politique

La gestion de la politique peut être améliorée en se concentrant sur plusieurs éléments. Tout d'abord, des politiques prédéfinies peuvent être établies pour les systèmes et applications usuels. Ainsi, des politiques pourraient être définies pour protéger le système de base : les logs système, les fichiers de configuration, les applications, etc.

De plus, des politiques pourraient être fournies pour les applications fréquemment utilisées, par exemple un serveur Web ou un service de base de données. En effet, les besoins de sécurité de ces applications varient peu lors d'une utilisation classique, et des politiques communes peuvent donc être envisagées (puisque dans de nombreux cas, seuls l'association des contextes aux ressources est modifiée). Notons également que ces politiques pourraient être distribuées grâce aux systèmes de paquets des distributions Linux, ce qui faciliterait leur utilisation.

Amélioration de la dynamicité

Dans cette thèse, nous avons considéré qu'une politique ne devait pas nécessairement évoluer au cours de la vie d'une machine. En effet, pour la majorité des machines virtuelles d'un environnement en nuage, leur rôle n'évolue pas et les besoins de sécurité sont donc fixes. Cependant, la gestion de l'ajout et de la suppression de propriétés à la volée permettrait de prendre en compte les cas où ces besoins de sécurité évoluent. Pour cela, il est

nécessaire d'étudier l'impact de ces actions sur le reste de la politique et de vérifier que celle-ci reste cohérente.

La dynamicité de l'environnement est quant à elle extrêmement présente dans les environnements en nuage, par exemple en ce qui concerne la migration de machines virtuelles ou de données. Hors, cette situation n'est actuellement pas prise en compte par la solution proposée. Une solution possible est de créer un lien entre les ressources et les propriétés qui s'y appliquent, par exemple en utilisant les attributs étendus des systèmes de fichiers.

Délégations et interactions des politiques

Dans le cas d'un service hébergé sur une infrastructure en nuage, différents niveaux de politiques peuvent être considérés. En effet, le fournisseur d'un service cherche à sécuriser l'environnement qu'il fournit, alors que le client veut sécuriser ses données et applications. Il faut donc permettre aux différents intervenants de définir des politiques sur les ressources dont ils sont responsables (en fonction du modèle de service).

La définition de politiques de sécurité par des intervenants distincts implique la gestion des interactions entre les politiques. Cela peut être fait en utilisant des méta-politiques [Blanc, 2006] afin de spécifier les différents critères à prendre en compte : sur quelles ressources un intervenant peut-il définir une politique ? En cas de conflit entre deux politiques, quelle propriété est jugée prioritaire ?

Intégration avec les outils et plateformes existants

Afin de simplifier l'utilisation de la solution, son intégration aux outils existants devrait être envisagée. Cela passe tout d'abord par la simplification de l'expression de la politique. Comme cela a été vu, l'expression des besoins peut être faite à partir d'outils d'analyse des risques. Une méthode pouvant traduire les résultats d'une telle analyse en politique de sécurité utilisant notre langage faciliterait fortement la définition des politiques par l'administrateur de l'architecture logicielle.

De plus, le SE^E doit actuellement être déployé sur les machines que l'on souhaite sécuriser. Une intégration du SE^E sur les infrastructure d'informatique en nuage simplifierait également l'utilisation de la solution. Ainsi, il serait possible de proposer des images utilisant le SE^E et pouvant être déployées sur des services en nuage. Une intégration au tableau de bord (*dashboard*) de l'infrastructure en nuage utilisée pourrait également être envisagée, à la fois pour la définition de la politique et pour la visualisation des données d'assurance et des scores.

Enfin, la politique de sécurité pourrait être intégrée aux SLA définis entre les clients et les fournisseurs de services. En effet, lorsque le client définit une politique de sécurité, le fournisseur s'engage à la respecter. Cela correspond donc au modèle des SLA où les clients et les fournisseurs définissent les termes du contrat à respecter par les deux parties.

Bibliographie

[Amazon, 2010] Amazon, E. (2010). Amazon elastic compute cloud (amazon ec2). *Amazon Elastic Compute Cloud (Amazon EC2).*

[Anderson, 1980] Anderson, J. P. (1980). Computer security threat monitoring and surveillance. Rapport technique, Technical report, James P. Anderson Company, Fort Washington, Pennsylvania.

[Andreasson, 2001] Andreasson, O. (2001). Iptables tutorial 1.2. 2.

[Andrieux *et al.*, 2007] Andrieux, A., Czajkowski, K., Dan, A., Keahey, K., Ludwig, H., Nakata, T., Pruyne, J., Rofrano, J., Tuecke, S. et Xu, M. (2007). Web services agreement specification (ws-agreement). *In Open Grid Forum*, volume 128, page 216.

[ApacheGUI, 2015] ApacheGUI (2015). ApacheGUI. `http://apachegui.net/`.

[Armbrust *et al.*, 2010] Armbrust, M., Fox, A., Griffith, R., Joseph, A. D., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., Stoica, I. *et al.* (2010). A view of cloud computing. *Communications of the ACM*, 53(4):50–58.

[Azraoui *et al.*, 2015] Azraoui, M., Elkhiyaoui, K., Önen, M., Bernsmed, K., De Oliveira, A. S. et Sendor, J. (2015). A-ppl : An accountability policy language. *In Data Privacy Management, Autonomous Spontaneous Security, and Security Assurance*, pages 319–326. Springer.

[Badger *et al.*, 2012] Badger, L., Grance, T., Patt-Corner, R. et Voas, J. (2012). Nist's cloud computing synopsis and recommendations.

[Bajikar, 2002] Bajikar, S. (2002). Trusted platform module (tpm) based security on notebook pcs-white paper. *Mobile Platforms Group Intel Corporation*, pages 1–20.

[Bantz *et al.*, 2003] Bantz, D. F., Bisdikian, C., Challener, D., Karidis, J. P., Mastrianni, S., Mohindra, A., Shea, D. G. et Vanover, M. (2003). Autonomic personal computing. *IBM Systems Journal*, 42(1):165–176.

[Barham *et al.*, 2003] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I. et Warfield, A. (2003). Xen and the art of virtualization. *ACM SIGOPS Operating Systems Review*, 37(5):164–177.

[Barrère *et al.*, 2012a] Barrère, M., Badonnel, R. et Festor, O. (2012a). Collaborative remediation of configuration vulnerabilities in autonomic networks and systems. *In Proceedings of the 8th International Conference on Network and Service Management*, pages 357–363. International Federation for Information Processing.

[Barrère *et al.*, 2012b] Barrère, M., Badonnel, R. et Festor, O. (2012b). Towards the assessment of distributed vulnerabilities in autonomic networks and systems. *In Network Operations and Management Symposium (NOMS), 2012 IEEE*, pages 335–342. IEEE.

[Bartels *et al.*, 2014] Bartels, A., Rymer, J. R. et Staten, J. (2014). The Public Cloud Market Is Now In Hypergrowth - Sizing The Public Cloud Market, 2014 To 2020.

[Bauer, 2006] Bauer, M. (2006). Paranoid penguin : an introduction to novell apparmor. *Linux Journal*, 2006(148):13.

[Bell et LaPadula, 1973] Bell, D. E. et LaPadula, L. J. (1973). Secure computer systems : Mathematical foundations. Rapport technique, DTIC Document.

[Bellard, 2005] Bellard, F. (2005). Qemu, a fast and portable dynamic translator. *In USENIX Annual Technical Conference, FREENIX Track*, pages 41–46.

[Bernsmed *et al.*, 2011] Bernsmed, K., Jaatun, M. G., Meland, P. H. et Undheim, A. (2011). Security slas for federated cloud services. *In Availability, Reliability and Security (ARES), 2011 Sixth International Conference on*, pages 202–209. IEEE.

[Betgé-Brezetz *et al.*, 2013] Betgé-Brezetz, S., Bousquet, A., Briffaut, J., Caron, E., Clevy, L., Dupont, M.-P., Kamga, G.-B., Lambert, J.-M., Lefray, A., Marquet, B. *et al.* (2013). *Seeding the Cloud : An Innovative Approach to Grow Trust in Cloud Based Infrastructures*. Springer.

[Biba, 1977] Biba, K. J. (1977). Integrity considerations for secure computer systems. Rapport technique, DTIC Document.

[Bigus *et al.*, 2002] Bigus, J. P., Schlosnagle, D. A., Pilgrim, J. R., Mills III, W. N. et Diao, Y. (2002). Able : A toolkit for building multiagent autonomic systems. *IBM Systems Journal*, 41(3):350–371.

[Bishop, 2003] Bishop, M. (2003). What is computer security ? *Security & Privacy, IEEE*, 1(1):67–69.

[Blanc, 2006] Blanc, M. (2006). *Sécurité des systèmes d'exploitation répartis : architecture décentralisée de méta-politique pour l'administration du contrôle d'accès obligatoire.* Thèse de doctorat, Université d'Orléans.

[Bobelin *et al.*, 2015] Bobelin, L., Bousquet, A. et Briffaut, J. (2015). An autonomic cloud management system for enforcing security and assurance properties. *In Proceedings of the 2015 Workshop on Changing Landscapes in HPC Security*, pages 1–8. ACM.

[Bobelin *et al.*, 2014] Bobelin, L., Bousquet, A., Briffaut, J., Couturier, J.-F., Toinard, C., Caron, E., Lefray, A. et Rouzaud-Cornabas, J. (2014). An advanced security-aware cloud architecture. *In High Performance Computing & Simulation (HPCS), 2014 International Conference on*, pages 572–579. IEEE.

[Boebert et Kain, 1989] Boebert, W. E. et Kain, R. Y. (1989). A practical alternative to hierarchical integrity policies. *NIST SPECIAL PUBLICATION SP*, pages A–10.

[Bouchenak *et al.*, 2011] Bouchenak, S., Boyer, F., Claudel, B., De Palma, N., Gruber, O. et Sicard, S. (2011). From autonomic to self-self behaviors : The jade experience. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 6(4):28.

[Bousquet *et al.*, 2015] Bousquet, A., Briffaut, J., Caron, E., Dominguez, E. M., Franco, J., Lefray, A., Lopez, O., Ros, S., Rouzaud-Cornabas, J., Toinard, C. et Uriarte, M. (2015). Enforcing security and assurance properties in cloud environment. *In Utility and Cloud Computing (UCC), 2015 IEEE/ACM 8th International Conference on*. IEEE.

[Bousquet *et al.*, 2013] Bousquet, A., Briffaut, J., Clévy, L., Toinard, C. et Venelle, B. (2013). Mandatory access control for the android dalvik virtual machine. *In 2013-USENIX Federated Conferences, ESOS : Workshop on Embedded Self-Organizing Systems*.

[Bousquet *et al.*, 2014] Bousquet, A., Briffaut, J. et Toinard, C. (2014). An autonomous cloud management system for in-depth security. *In Cloud Networking (CloudNet), 2014 IEEE 3rd International Conference on*, pages 368–374. IEEE.

[Bray *et al.*, 2008] Bray, R., Cid, D. et Hay, A. (2008). *OSSEC host-based intrusion detection guide*. Syngress.

[Briffaut, 2007] Briffaut, J. (2007). *Formalization and guaranty of system security properties : application to the detection of intrusions*. Thèse de doctorat, Université d'Orléans.

[Burr *et al.*, 2013] Burr, W. E., Dodson, D. F. et Polk, W. T. (2013). Electronic authentication guideline : Recommendations of the national institute of standards and technology. *NIST Special Publication*, pages 800–63.

[Buyya *et al.*, 2008] Buyya, R., Yeo, C. S. et Venugopal, S. (2008). Market-oriented cloud computing : Vision, hype, and reality for delivering it services as computing utilities. *In High Performance Computing and Communications, 2008. HPCC'08. 10th IEEE International Conference on*, pages 5–13. Ieee.

[Chaubal, 2008] Chaubal, C. (2008). The architecture of vmware esxi. *VMware White Paper*, 1:7.

[Citrix, 2015a] Citrix (2015a). Gotomeeting. `http://www.gotomeeting.com/`.

[Citrix, 2015b] Citrix (2015b). XenServer. `http://xenserver.org/`.

[Clark *et al.*, 2007] Clark, D., Dublish, P., Johnson, M., Kowalski, V., Labrou, Y., Negritoiu, S., Vampenepe, W., Waschke, M., Wiles, V. et Wurster, K. (2007). The federated cmdb vision. *Joint White Paper from BMC, CA, Fujitsu, HP, IBM and Microsoft, Version*, 1.

[Claudel *et al.*, 2006] Claudel, B., De Palma, N., Lachaize, R. et Hagimont, D. (2006). Self-protection for distributed component-based applications. *In Stabilization, Safety, and Security of Distributed Systems*, pages 184–198. Springer.

[CloudStack, 2015] CloudStack, A. (2015). Apache cloudstack : Open source cloud computing. `https://cloudstack.apache.org/`.

[Conner, 2008] Conner, N. (2008). *Google Apps : The Missing Manual : The Missing Manual*. " O'Reilly Media, Inc.".

[Costache *et al.*, 2013] Costache, S., Parlavantzas, N., Morin, C. et Kortas, S. (2013). Merkat : A market-based slo-driven cloud platform. *In Cloud Computing Technology and Science (CloudCom), 2013 IEEE 5th International Conference on*, volume 1, pages 403–410. IEEE.

[CSA, 2011] CSA (2011). Security guidance for critical areas of focus in cloud computing v3.0. *Cloud Security Alliance*.

[CSA, 2015] CSA (2015). Cloud Adoption Practices and Priorities Survey Report. https ://cloudsecurityalliance.org/download/cloud-adoption-practices-priorities-survey-report/.

[CSA, 2015a] CSA (2015a). Cloud Controls Matrix (CCM). `https://cloudsecurityalliance.org/cm.html`.

[CSA, 2015b] CSA (2015b). Consensus Assessments Initiative Questionnaire (CAIQ) . `https://cloudsecurityalliance.org/cai.html`.

[Damianou *et al.*, 2000] Damianou, N., Dulay, N., Lupu, E. et Sloman, M. (2000). A language for specifying security and management policies for distributed systems. *London : Department of Computing, Imperial College, Tech. Rep.*

[Diao *et al.*, 2005] Diao, Y., Hellerstein, J. L., Parekh, S., Griffith, R., Kaiser, G. et Phung, D. (2005). Self-managing systems : A control theory foundation. *In Engineering of Computer-Based Systems, 2005. ECBS'05. 12th IEEE International Conference and Workshops on the*, pages 441–448. IEEE.

[Doelitzscher *et al.*, 2012] Doelitzscher, F., Reich, C., Knahl, M., Passfall, A. et Clarke, N. (2012). An agent based business aware incident detection system for cloud environments. *Journal of Cloud Computing*, 1(1):1–19.

[Feilner, 2006] Feilner, M. (2006). *OpenVPN : Building and integrating virtual private networks*. Packt Publishing Ltd.

[Ferber, 1999] Ferber, J. (1999). *Multi-agent systems : an introduction to distributed artificial intelligence*, volume 1. Addison-Wesley Reading.

[Ferraiolo et Kuhn, 1992] Ferraiolo, D. et Kuhn, R. (1992). Role-based access control. *In In 15th NIST-NCSC National Computer Security Conference*.

[Foundry, 2015] Foundry, C. (2015). Cloud Foundry. `http://cloudfoundry.org/`.

[GlobalPlatform, 2011] GlobalPlatform (2011). Card Specification v2.2.1.

[Goldberg, 1973] Goldberg, R. P. (1973). Architecture of virtual machines. *In Proceedings of the workshop on virtual computer systems*, pages 74–112. ACM.

[Google, 2015] Google (2015). Google cloud platform. `https://cloud.google.com/`.

[Greene, 2012] Greene, J. (2012). Intel trusted execution technology. *Intel Technology White Paper*.

[Habib, 2008] Habib, I. (2008). Virtualization with kvm. *Linux Journal*, 2008(166):8.

[Harada *et al.*, 2004] Harada, T., Horie, T. et Tanaka, K. (2004). Task oriented management obviates your onus on linux. *In Linux Conference*, volume 3.

[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.

[Helsley, 2009] Helsley, M. (2009). Lxc : Linux container tools. *IBM devloperWorks Technical Library*.

[Hoff *et al.*, 2010] Hoff, C., Johnston, S., Reese, G. et Sapiro, B. (2010). Cloudaudit 1.0 - Automated Audit, Assertion, Assessment, and Assurance API (A6). *Internet Engineering Task Force, Internet-Draft draft-hoff-cloudaudit-00*.

[Horn, 2001] Horn, P. (2001). Autonomic computing : Ibm\'s perspective on the state of information technology.

[Hu *et al.*, 2014] Hu, V. C., Ferraiolo, D., Kuhn, R., Schnitzer, A., Sandlin, K., Miller, R. et Scarfone, K. (2014). Guide to attribute based access control (abac) definition and considerations. *NIST Special Publication*, 800:162.

[Hu *et al.*, 2007] Hu, V. C., Martin, E., Hwang, J. et Xie, T. (2007). Conformance checking of access control policies specified in xacml. *In Computer Software and Applications Conference, 2007. COMPSAC 2007. 31st Annual International*, volume 2, pages 275–280. IEEE.

[Huebscher et McCann, 2008] Huebscher, M. C. et McCann, J. A. (2008). A survey of autonomic computing—degrees, models, and applications. *ACM Computing Surveys (CSUR)*, 40(3):7.

[IBM, 2003] IBM (2003). An architectural blueprint for autonomic computing. *IBM Publication*.

[Ibrahim *et al.*, 2011] Ibrahim, A. S., Hamlyn-Harris, J., Grundy, J. et Almorsy, M. (2011). Cloudsec : a security monitoring appliance for virtual machines in the iaas cloud model. *In Network and System Security (NSS), 2011 5th International Conference on*, pages 113–120. IEEE.

[ISO, 2013a] ISO (2013a). ISO27001 : Information Security Management System (ISMS) standard.

[ISO, 2013b] ISO (2013b). ISO27002 : Information technology – Security techniques – Code of practice for information security controls.

[ISO, 2015] ISO (2015). ISO27017 : Information technology – Security techniques – Code of practice for information security controls based on ISO/IEC 27002 for cloud services.

[ITSEC, 1991] ITSEC (1991). Information Technology Security Evaluation Criteria (ITSEC) v1.2. Technical report.

[Jacob *et al.*, 2004] Jacob, B., Lanyon-Hogg, R., Nadgir, D. K. et Yassin, A. F. (2004). A practical guide to the ibm autonomic computing toolkit.

[Jennings, 2000] Jennings, N. R. (2000). On agent-based software engineering. *Artificial intelligence*, 117(2):277–296.

[Jin *et al.*, 2002] Jin, L.-j., Machiraju, V. et Sahai, A. (2002). Analysis on service level agreement of web services. *HP June*, page 19.

[Jin *et al.*, 2012] Jin, X., Krishnan, R. et Sandhu, R. S. (2012). A unified attribute-based access control model covering dac, mac and rbac. *DBSec*, 12:41–55.

[Kalam *et al.*, 2003] Kalam, A. A. E., Baida, R., Balbiani, P., Benferhat, S., Cuppens, F., Deswarte, Y., Miege, A., Saurel, C. et Trouessin, G. (2003). Organization based access control. *In Policies for Distributed Systems and Networks, 2003. Proceedings. POLICY 2003. IEEE 4th International Workshop on*, pages 120–131. IEEE.

[Kamp et Watson, 2000] Kamp, P.-H. et Watson, R. N. (2000). Jails : Confining the omnipotent root. *In Proceedings of the 2nd International SANE Conference*, volume 43, page 116.

[Karp, 1972] Karp, R. M. (1972). *Reducibility among combinatorial problems*. Springer.

[Kearney *et al.*, 2010] Kearney, K. T., Torelli, F. et Kotsokalis, C. (2010). Sla* : An abstract syntax for service level agreements. *In Grid Computing (GRID), 2010 11th IEEE/ACM International Conference on*, pages 217–224. IEEE.

[Kephart et Chess, 2003] Kephart, J. O. et Chess, D. M. (2003). The vision of autonomic computing. *Computer*, 36(1):41–50.

[Keromytis *et al.*, 2012] Keromytis, A. D., Geambasu, R., Sethumadhavan, S., Stolfo, S. J., Yang, J., Benameur, A., Dacier, M., Elder, M., Kienzle, D. et Stavrou, A. (2012). The meerkats cloud security architecture. *In Distributed Computing Systems Workshops (ICDCSW), 2012 32nd International Conference on*, pages 446–450. IEEE.

[Kivity *et al.*, 2007] Kivity, A., Kamay, Y., Laor, D., Lublin, U. et Liguori, A. (2007). kvm : the linux virtual machine monitor. *In Proceedings of the Linux Symposium*, volume 1, pages 225–230.

[Kolyshkin, 2006] Kolyshkin, K. (2006). Virtualization in linux. *White paper, OpenVZ*.

[KPMG, 2014] KPMG (2014). Cloud Survey Report : Elevating Business in the Cloud.

[Krishnan et Gonzalez, 2015] Krishnan, S. et Gonzalez, J. L. U. (2015). Google compute engine. *In Building Your Next Big Thing with Google Cloud Platform*, pages 53–81. Springer.

[Kumar et Cohen, 2000] Kumar, S. et Cohen, P. R. (2000). Towards a fault-tolerant multi-agent system architecture. *In Proceedings of the fourth international conference on Autonomous agents*, pages 459–466. ACM.

[Lampson, 1971] LAMPSON, B. (1971). Protection. *In Proc. 5th Princeton Conf. on Information Sciences and Systems*, pages 18–24. Princeton.

[Lampson, 1969] LAMPSON, B. W. (1969). Dynamic protection structures. *In Proceedings of the November 18-20, 1969, fall joint computer conference*, pages 27–38. ACM.

[Lampson, 1973] LAMPSON, B. W. (1973). A note on the confinement problem. *Communications of the ACM*, 16(10):613–615.

[Lefray *et al.*, 2013a] LEFRAY, A., CARON, E., ROUZAUD-CORNABAS, J., HUAXI YULIN, Z., BOUSQUET, A., BRIFFAUT, J. et TOINARD, C. (2013a). Security-Aware Models for Clouds. ACM Symposium on High-Performance Parallel and Distributed Computing.

[Lefray *et al.*, 2013b] LEFRAY, A., ROUZAUD-CORNABAS, J., BRIFFAUT, J. et TOINARD, C. (2013b). Security for cloud environment through information flow properties formalization with a first-order temporal logic.

[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*, volume 10, pages 303–314.

[Ludwig *et al.*, 2003] LUDWIG, H., KELLER, A., DAN, A., KING, R. P. et FRANCK, R. (2003). Web service level agreement (wsla) language specification. *IBM Corporation*, pages 815–824.

[McCarty, 2004] MCCARTY, B. (2004). *SELinux*. O&Reilly.

[Mell et Grance, 2011] MELL, P. et GRANCE, T. (2011). The nist definition of cloud computing.

[Merkel, 2014] MERKEL, D. (2014). Docker : lightweight linux containers for consistent development and deployment. *Linux Journal*, 2014(239):2.

[Microsoft, 2015] MICROSOFT (2015). Microsoft azure. `https://azure.microsoft.com/`.

[Miner et Athey, 2007] MINER, D. et ATHEY, J. (2007). FCGlob : A New SELinux File Context Syntax. *In Proceedings of the Third Annual Security Enhanced Linux Symposium*.

[Moore, 2015] MOORE, P. (2015). The State of SELinux. *Linux Security Summit*.

[Moreno-Vozmediano *et al.*, 2012] MORENO-VOZMEDIANO, R., MONTERO, R. S. et LLORENTE, I. M. (2012). Iaas cloud architecture : From virtualized datacenters to federated cloud infrastructures. *Computer*, (12):65–72.

[Morris, 2009] MORRIS, J. (2009). svirt : Hardening linux virtualization with mandatory access control. *In Linux. conf. au Conference*.

[Moses *et al.*, 2005] MOSES, T. *et al.* (2005). Extensible access control markup language (xacml) version 2.0. *Oasis Standard*, 200502.

[Murray, 2011] MURRAY, K. (2011). *Microsoft Office 365 : Connect and collaborate virtually anywhere, anytime*. Microsoft Press.

[Neuman et Ts' O, 1994] NEUMAN, B. C. et TS' O, T. (1994). Kerberos : An authentication service for computer networks. *Communications Magazine, IEEE*, 32(9):33–38.

[NIST, 2014] NIST (2014). ShellShock. `https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-6271`.

[NIST, 2015] NIST (2015). VENOM. `https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2015-3456`.

[Nurmi *et al.*, 2009] Nurmi, D., Wolski, R., Grzegorczyk, C., Obertelli, G., Soman, S., Youseff, L. et Zagorodnov, D. (2009). The eucalyptus open-source cloud-computing system. *In Cluster Computing and the Grid, 2009. CCGRID'09. 9th IEEE/ACM International Symposium on*, pages 124–131. IEEE.

[OASIS, 2015] OASIS (2015). OASIS. https://www.oasis-open.org/.

[OneCMDB, 2015] OneCMDB (2015). OneCMDB. http://www.onecmdb.org.

[OpenSSH, 2014] OpenSSH (2014). Openssh website. http ://www.openssh.com/.

[OpenStack, 2015] OpenStack (2015). OpenStack Juno. https://www.openstack.org/software/juno/.

[Östergård, 2001] Östergård, P. R. (2001). A new algorithm for the maximum-weight clique problem. *Nordic Journal of Computing*, 8(4):424–436.

[OVAL, 2014] OVAL (2014). Oval language. http ://oval.mitre.org/.

[PaX Team, 2003] PaX Team (2003). Pax address space layout randomization (aslr).

[Pearson *et al.*, 2012] Pearson, S., Tountopoulos, V., Catteddu, D., Südholt, M., Molva, R., Reich, C., Fischer-Hübner, S., Millard, C., Lotz, V., Jaatun, M. G. *et al.* (2012). Accountability for cloud and other future internet services. *In CloudCom*, pages 629–632.

[Pepple, 2011] Pepple, K. (2011). *Deploying openstack.* " O'Reilly Media, Inc.".

[Perez *et al.*, 2006] Perez, R., Sailer, R., van Doorn, L. *et al.* (2006). vtpm : virtualizing the trusted platform module. *In Proc. 15th Conf. on USENIX Security Symposium*, pages 305–320.

[PHP, 2015] PHP (2015). phpMyAdmin. https://www.phpmyadmin.net/.

[Popek et Goldberg, 1974] Popek, G. J. et Goldberg, R. P. (1974). Formal requirements for virtualizable third generation architectures. *Communications of the ACM*, 17(7):412–421.

[RedHat, 2015] RedHat (2015). Firewalld. https://fedoraproject.org/wiki/FirewallD.

[RedHat, 2015] RedHat (2015). Openshift. https://www.openshift.com/.

[Ristic, 2010] Ristic, I. (2010). *ModSecurity Handbook.* Feisty Duck.

[Roblee *et al.*, 2005] Roblee, C., Berk, V. et Cybenko, G. (2005). Implementing large-scale autonomic server monitoring using process query systems. *In Autonomic Computing, 2005. ICAC 2005. Proceedings. Second International Conference on*, pages 123–133. IEEE.

[Russell et Welte, 2002] Russell, R. et Welte, H. (2002). Linux netfilter hacking howto. *Disponıvel em http ://www. netfilter. org/documentation/HOWTO//netfilter-hacking-HOWTO. letter. ps (Junho de 2005).*

[Sahoo *et al.*, 2010] Sahoo, J., Mohapatra, S. et Lath, R. (2010). Virtualization : A survey on concepts, taxonomy and associated security issues. *In Computer and Network Technology (ICCNT), 2010 Second International Conference on*, pages 222–226. IEEE.

[Salesforce, 2015a] Salesforce (2015a). Crm. https://www.salesforce.com/.

[Salesforce, 2015b] Salesforce (2015b). Heroku. https://www.heroku.com/.

[Samar, 1996] Samar, V. (1996). Unified login with pluggable authentication modules (pam). *In Proceedings of the 3rd ACM conference on Computer and communications security*, pages 1–10. ACM.

[Sandhu *et al.*, 2000] Sandhu, R., Ferraiolo, D. et Kuhn, R. (2000). The nist model for role-based access control : towards a unified standard. *In ACM workshop on Role-based access control*, volume 2000.

[Sandhu, 1988] Sandhu, R. S. (1988). The schematic protection model : its definition and analysis for acyclic attenuating schemes. *Journal of the ACM (JACM)*, 35(2):404–432.

[Sandhu, 1992] Sandhu, R. S. (1992). The typed access matrix model. *In Research in Security and Privacy, 1992. Proceedings., 1992 IEEE Computer Society Symposium on*, pages 122–136. IEEE.

[Sandhu *et al.*, 1996] Sandhu, R. S., Coyne, E. J., Feinstein, H. L. et Youman, C. E. (1996). Role-based access control models. *Computer*, (2):38–47.

[Schaufler, 2008] Schaufler, C. (2008). The simplified mandatory access control kernel. *White Paper*, pages 1–11.

[Schmerl et Garlan, 2002] Schmerl, B. et Garlan, D. (2002). Exploiting architectural design knowledge to support self-repairing systems. *In Proceedings of the 14th international conference on Software engineering and knowledge engineering*, pages 241–248. ACM.

[Smalley *et al.*, 2001] Smalley, S., Vance, C. et Salamon, W. (2001). Implementing selinux as a linux security module. *NAI Labs Report*, 1(43):139.

[Smith et Nair, 2005a] Smith, J. et Nair, R. (2005a). *Virtual machines : versatile platforms for systems and processes*. Elsevier.

[Smith et Nair, 2005b] Smith, J. E. et Nair, R. (2005b). The architecture of virtual machines. *Computer*, 38(5):32–38.

[Soshi *et al.*, 2004] Soshi, M., Maekawa, M. et Okamoto, E. (2004). The dynamic-typed access matrix model and decidability of the safety problem. *IEICE transactions on fundamentals of electronics, communications and computer sciences*, 87(1):190–203.

[Spencer *et al.*, 1999] Spencer, R., Smalley, S., Loscocco, P., Hibler, M. et Lepreau, J. (1999). The flask security architecture : System support for diverse policies. *In Proceedings of the Eighth USENIX Security Symposium*.

[Spengler, 2002] Spengler, B. (2002). Detection, prevention, and containment : A study of grsecurity. *In Libre Software Meeting*.

[Standard, 2004] Standard, R. (2004). Incits 359-2004. *ANSI INCITS*, pages 359–2004.

[Sterritt *et al.*, 2005] Sterritt, R., Smyth, B. et Bradley, M. (2005). Pact : personal autonomic computing tools. *In Engineering of Computer-Based Systems, 2005. ECBS'05. 12th IEEE International Conference and Workshops on the*, pages 519–527. IEEE.

[Stoneburner, 2001] Stoneburner, G. (2001). *Underlying technical models for information technology security : recommendation of the National Institute of Standards and Technology*. US Department of Commerce, Computer Security Division, Information Technology, National Institute of Standards and Technology.

[Strassner *et al.*, 2006] Strassner, J., Agoulmine, N. et Lehtihet, E. (2006). Focale : A novel autonomic networking architecture.

[Strassner *et al.*, 2008] Strassner, J., Samudrala, S., Cox, G., Liu, Y., Jiang, M., Zhang, J., van der Meer, S., Donnelly, W. *et al.* (2008). The design of a new context-aware policy model for autonomic networking. *In International Conference on Autonomic Computing*, pages 119–128. IEEE.

[Sturm *et al.*, 2000] Sturm, R., Morris, W. et Jander, M. (2000). {Foundations of Service Level Management}.

[Tai, 1979] Tai, K.-C. (1979). The tree-to-tree correction problem. *Journal of the ACM (JACM)*, 26(3):422–433.

[Tcsec, 1985] Tcsec, D. (1985). Trusted computer system evaluation criteria. *DoD 5200.28-STD*, 83.

[Tomlinson, 2008] Tomlinson, A. (2008). Introduction to the tpm. *In Smart Cards, Tokens, Security and Applications*, pages 155–172. Springer.

[Twidle *et al.*, 2009] Twidle, K., Dulay, N., Lupu, E. et Sloman, M. (2009). Ponder2 : A policy system for autonomous pervasive environments. *In Autonomic and Autonomous Systems, 2009. ICAS'09. Fifth International Conference on*, pages 330–335. IEEE.

[Vaquero *et al.*, 2008] Vaquero, L. M., Rodero-Merino, L., Caceres, J. et Lindner, M. (2008). A break in the clouds : towards a cloud definition. *ACM SIGCOMM Computer Communication Review*, 39(1):50–55.

[Velte et Velte, 2009] Velte, A. et Velte, T. (2009). *Microsoft virtualization with Hyper-V*. McGraw-Hill, Inc.

[Venelle *et al.*, 2013] Venelle, B., Briffaut, J., Clévy, L. et Toinard, C. (2013). Security enhanced java : Mandatory access control for the java virtual machine. *In Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC), 2013 IEEE 16th International Symposium on*, pages 1–7. IEEE.

[Viega *et al.*, 2002] Viega, J., Messier, M. et Chandra, P. (2002). *Network Security with OpenSSL : Cryptography for Secure Communications.* " O'Reilly Media, Inc.".

[Wailly *et al.*, 2012] Wailly, A., Lacoste, M. et Debar, H. (2012). Vespa : multi-layered self-protection for cloud resources. *In Proceedings of the 9th international conference on Autonomic computing*, pages 155–160. ACM.

[Wains, 2013] Wains, S. (2013). tcpdump advanced filters. `http://www.wains.be/pub/networking/tcpdump_advanced_filters.txt`.

[Waltermire *et al.*, 2011a] Waltermire, D., Quinn, S., Scarfone, K. et Halbardier, A. (2011a). The technical specification for the Security Content Automation Protocol (SCAP). *NIST Special Publication*, 800:126.

[Waltermire *et al.*, 2011b] Waltermire, D., Schmidt, C., Scarfone, K. et Ziring, N. (2011b). Specification for the extensible configuration checklist description format (xccdf) version 1.2 (draft).

[Wooldridge, 2009] Wooldridge, M. (2009). *An introduction to multiagent systems*. John Wiley & Sons.

[Wu et Buyya, 2012] Wu, L. et Buyya, R. (2012). Service level agreement (sla) in utility computing systems. *IGI Global*.

[Wustenhoff et BluePrints, 2002] Wustenhoff, E. et BluePrints, S. (2002). Service level agreement in the data center. *Sun Microsystems Professional Series*.

[Xu *et al.*, 2005] Xu, J., Adabala, S. et Fortes, J. A. (2005). Towards autonomic virtual applications in the in-vigo system. *In Autonomic Computing, 2005. ICAC 2005. Proceedings. Second International Conference on*, pages 15–26. IEEE.

[Ylonen et Lonvick, 2006] Ylonen, T. et Lonvick, C. (2006). The secure shell (ssh) connection protocol.

[Zahariev, 2009] Zahariev, A. (2009). Google app engine. *Helsinki University of Technology.*

[Zenoss, 2014] Zenoss (2014). The State of the Open Source Cloud . http://www.zenoss.com/documents/2014-State-OS-Cloud-Report.pdf.

[Zhang *et al.*, 2010] Zhang, Q., Cheng, L. et Boutaba, R. (2010). Cloud computing : state-of-the-art and research challenges. *Journal of internet services and applications*, 1(1):7–18.

Annexes

A Langage

A.1 Capacités

Cette section présente trois tables contenant les capacités d'application, les capacités d'assurance et les capacités internes qui sont définies. Ces tables ne sont pas exhaustives : de nouvelles capacités peuvent être définies et, pour chacune d'entre elles, d'autres mécanismes que ceux cités peuvent être utilisés. L'expert en sécurité est chargé de définir de nouvelles capacités ou de nouveaux mécanismes : l'administrateur de l'architecture logicielle n'a pas accès aux informations de ces différentes tables. La table 1 présente les capacités d'application qui ont été définies ainsi que des mécanismes (chapitre 2) qui leur sont associés.

Capacité	**Mécanismes**	**Description**
allow_access	DAC SELinux grsecurity WebAccess	Autorise les accès à une ressource
allow_connection	PAM Kerberos LDAP	Autorise une connexion sur une machine
allow_exec_access	DAC SELinux grsecurity	Autorise l'exécution d'une ressource système
allow_read_access	DAC SELinux grsecurity	Autorise les accès en lecture à une ressource système
allow_tcp_networking	Iptables pf Firewalld	Autorise les connexions TCP
allow_udp_networking	Iptables pf Firewalld	Autorise les connexions UDP
allow_write_access	DAC SELinux grsecurity	Autorise les accès en écriture à une ressource système
close_interface	Iptables pf Firewalld	Ferme une interface réseau
close_port	Iptables pf Firewalld	Ferme un port

collect_assurance_results	Oscap SAINT AssuranceExec	Collecte des résultats des tests d'assurance
create_domain	SELinux	Crée un domaine système isolé
decrypt_on_access	TPM SE GPG JCE OpenSSL	Déchiffre un fichier lors d'un accès autorisé
deny_all_accesses	DAC SELinux grsecurity	Interdit les accès à une ressources système
deny_all_exec_accesses	DAC SELinux grsecurity	Interdit l'exécution d'une ressource système
deny_all_read_accesses	DAC SELinux grsecurity	Interdit les accès en lecture à une ressource système
deny_all_write_accesses	DAC SELinux grsecurity	Interdit les accès en écriture à une ressource système
deny_connection	PAM Kerberos LDAP SSH	Interdit une connexion sur une machine
detect_authentication	PAM	Détecte une authentification réussie
encrypt_file	TPM SE GPG JCE OpenSSL	Chiffre un fichier
encrypt_flow	OpenVPN Ipsec OpenSSH	Chiffre un flux réseau
generate_key	TPM SE GPG JCE OpenSSL	Génère une clef de chiffrement
hmac_communication	SSH OpenSSL	Assure l'intégrité d'un message
isolate	SELinux sVirt	Isole une ressource
open_interface	Iptables pf Firewalld	Ouvre une interface réseau
open_port	Iptables pf Firewalld	Ouvre un port
redirect_flow	Iptables pf Firewalld	Redirige un flux réseau
run_assurance_tests	Oscap SAINT AssuranceExec	Exécute des tests d'assurance

sticky_policy_based_AC	DPM	Active un contrôle d'accès lié à une politique stockée dans la ressource

Table 1 – Capacités d'application et mécanismes associés

La table 2 présente les capacités d'assurance et certains des mécanismes qui peuvent les fournir.

Capacité	Mécanismes	Description
check_encrypted	GPG OpenSSL	Vérifie qu'une ressource système est chiffrée
check_encrypted_flow	tcpdump	Vérifie qu'un flux réseau est chiffré
check_exec	test	Vérifie que l'exécution est autorisée
check_integrity_sum	md5deep	Vérifie l'intégrité d'une ressource
check_isolated	test	Vérifie qu'une ressource est isolée
check_port_opened	nmap netstat	Vérifie qu'un port est ouvert
check_read	test	Vérifie que la lecture est autorisée
check_write	test	Vérifie que l'écriture est autorisée
try_login	su ssh	Vérifie qu'un utilisateur peut se connecter

Table 2 – Capacités d'assurance et mécanismes associés

Enfin, la table 3 présente des capacités internes du langage.

Capacité	Description
consistent_plugin	Sélectionne des plugins compatibles sur plusieurs nœuds
get_all_users	Récupère la liste de tous les utilisateurs du système
get_files	Récupère la liste des fichiers ayant un contexte donné
get_local_ip	Récupère la liste des adresses IP d'un nœud
get_published	Récupère des données d'un SE^E distant
get_system_version	Récupère la version du système du nœud
get_users	Récupère la liste des utilisateurs ayant un contexte donné
publish	Publie des données pour un SE^E distant
require_plugin	Force l'utilisation d'un plugin

Table 3 – Capacités internes

A.2 Classes et Propriétés

Plusieurs propriétés sont définies dans notre langage. Cette section en présente certaines, mais de nouvelles peuvent être définies par le responsable de la sécurité afin de répondre à un nouveau besoin.

A.2.1 Confidentialité

La classe **`@Confidentiality`** inclut deux sous-classes : **`@Confidentiality_System`** et **`@Confidentiality_Network`**. Leurs propriétés ont été détaillées à la section 3.5.

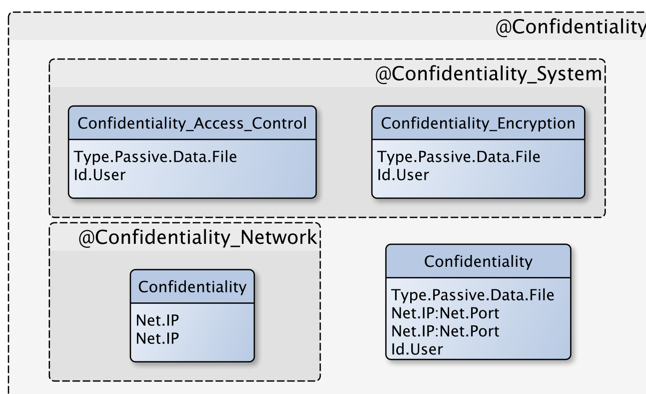

FIGURE 1 – Classe de confidentialité

La propriété **`Confidentiality_access_control`** (listing 1) protège un fichier en contrôlant les accès en lecture sur celui-ci.

```
1  boolean Confidentiality_access_control (Type.Passive.Data.File SCFile, Id.User
       SCUser) {
2   enforcement {
3    deny_all_read_accesses (SCFile);
4    allow_read_access (SCFile, SCUser);
5   }
6   assurance {
7    boolean c = true;
8    authorized_users = get_users(SCUser);
9    for (SCFileTmp IN get_files(SCFile) {
10    for (SCUserTmp IN get_all_users()) {
11     if (SCUserTmp IN authorized_users) {
12      c &= check_read (SCFileTmp, SCUserTmp);
13     }
14     else {
15      c &= (NOT check_read (SCFileTmp, SCUserTmp));
16     }
17    }
18   }
19   return c;
20  }
21 }
```

Listing 1 – Propriété de confidentialité d'un fichier (contrôle d'accès)

La propriété du listing 2 vise également à limiter le lecture du fichier, mais cette fois en utilisant du chiffrement.

```
1  boolean Confidentiality_encryption (Type.Passive.Data.File SCFile, Id.User
       SCUser) {
2   enforcement {
3    Context SCKey;
4    SCKey = Type.Passive.Data.Key.Symmetric;
5    SCKey += generate_key (SCKey);
6    Confidentiality_access_control (SCKey, SCUser);
7    encrypt_file (SCFile, SCKey);
8    decrypt_on_access(SCFile, SCUser);
9   }
10  assurance {
11   boolean c = true;
```

```
12  authorized_users = get_users(SCUser);
13  for (SCFileTmp IN get_files(SCFile) {
14   for (SCUserTmp IN get_all_users()) {
15    if (SCUserTmp IN authorized_users) {
16     c &= check_encrypted (SCFileTmp, SCUserTmp);
17    }
18    else {
19     c &= (NOT check_encrypted (SCFileTmp, SCUserTmp));
20    }
21   }
22  }
23  return c;
24 }
25 }
```

Listing 2 – Propriété de confidentialité d'un fichier (chiffrement)

La propriété du listing 3 chiffre des communications réseaux pour qu'elles soient confidentielles.

```
1 boolean Confidentiality (Net.IP SC1, Net.IP SC2) {
2  enforcement {
3   Type.Passive.Data.Key SCKey;
4   if (SC1.Net.IP == get_local_ip()) {
5    SCKey += generate_key (Type.Passive.Data.Key.[Symmetric|Asymmetric]);
6    publish(SCKey, SC2);
7    Plugin plug = consistent_plugin (SC1, SC2, encrypt_flow, SC1, SC2, SCKey);
8    plug->encrypt_flow (SC1, SC2, SCKey);
9   }
10  else {
11   get_published(SC1, SCKey);
12   Plugin plug = consistent_plugin (SC1, SC2, encrypt_flow, SC1, SC2, SCKey);
13   plug->encrypt_flow (SC1, SC2, SCKey);
14  }
15 }
16 assurance {
17  return check_encrypted_flow (SC1, SC2);
18 }
19 }
```

Listing 3 – Propriété de confidentialité réseau

La propriété du listing 4 définit la confidentialité d'un fichier vis-à-vis d'utilisateurs situés sur une machine distante.

```
1 boolean Confidentiality (Type.Passive.Data.File SCFile, Net.IP:Net.Port SC1,
      Net.IP:Net.Port SC2, Id.User SCUser) {
2  enforcement && assurance {
3   if (SC1.Net.IP == get_local_IP()) {
4    Confidentiality_Access_Control (SCFile:SC1, SCUser:SC1);
5    Confidentiality (SC1, SC2);
6   }
7   else {
8    Confidentiality (SC1, SC2);
9    Confidentiality_Access_Control (SCFile:SC2, SCUser:SC2);
10  }
11 }
12 }
```

Listing 4 – Propriété de confidentialité hybride

A.2.2 Intégrité

La classe d'intégrité contient des propriétés d'intégrité système et réseau.

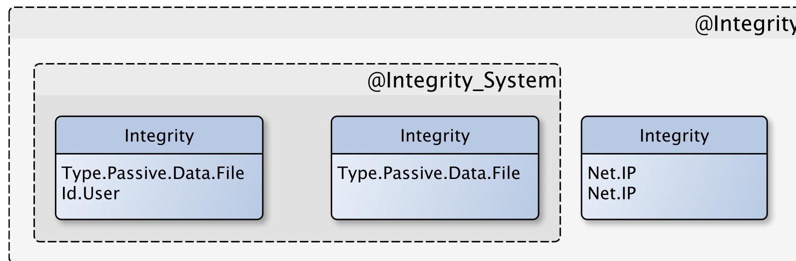

FIGURE 2 – Classe d'intégrité

Le listing 5 définit une propriété empêchant toutes les écritures sur un fichier, tandis que le listing 6 autorise les écritures pour certains utilisateurs. Dans les deux cas, les propriétés sont vérifiées en tentant d'ouvrir les fichiers en écriture.

```
1 boolean Integrity (Type.Passive.Data.File SCFile) {
2  enforcement {
3   return deny_all_write_accesses(SCFile);
4  }
5  assurance {
6    boolean c = true;
7    for (SCFileTmp IN get_files(SCFile) {
8     return check_integrity_sum(SCFileTmp);
9    }
10 }
11 }
```

Listing 5 – Propriété d'intégrité totale d'un fichier

```
1 boolean Integrity (Type.Passive.Data.File SCFile, Id.User SCUser) {
2  enforcement {
3   deny_all_write_accesses(SCFile);
4   return allow_write_access(SCFile, SCUser);
5  }
6  assurance {
7    boolean c = true;
8    authorized_users = get_users(SCUser);
9    for (SCFileTmp IN get_files(SCFile) {
10    for (SCUserTmp IN get_all_users()) {
11     if (SCUserTmp IN authorized_users) {
12      c &= check_write (SCFileTmp, SCUserTmp);
13     }
14     else {
15      c &= (NOT check_write (SCFileTmp, SCUserTmp));
16     }
17    }
18   }
19   return c;
20 }
21 }
```

Listing 6 – Propriété d'intégrité d'un fichier

La propriété du listing 7 exprime l'intégrité des communications réseaux.

```
1 boolean Integrity (Net.IP SC1, Net.IP SC2) {
```

```
2  enforcement {
3    hmac_communication (SC1, SC2);
4  }
5  assurance {
6    return check_integrity_sum(SC1, SC2);
7  }
8 }
```

Listing 7 – Propriété d'intégrité réseau

A.2.3 Exécutable

Le listing 8 autorise l'exécution d'un fichier par un utilisateur, mais l'interdit pour les autres utilisateurs.

```
1 boolean Executable (Type.Passive.Data.File SCFile, Id.User SCUser) {
2  enforcement {
3   deny_all_exec_accesses(SCFile);
4   return allow_exec_access(SCFile, SCUser);
5  }
6  assurance {
7    boolean c = true;
8    authorized_users = get_users(SCUser);
9    for (SCFileTmp IN get_files(SCFile) {
10    for (SCUserTmp IN get_all_users()) {
11     if (SCUserTmp IN authorized_users) {
12      c &= check_exec (SCFileTmp, SCUserTmp);
13     }
14     else {
15      c &= (NOT check_exec (SCFileTmp, SCUserTmp));
16     }
17    }
18   }
19   return c;
20  }
21 }
```

Listing 8 – Propriété rendant un fichier exécutable

A.2.4 Isolation

La classe d'isolation permet d'isoler un ensemble d'éléments.

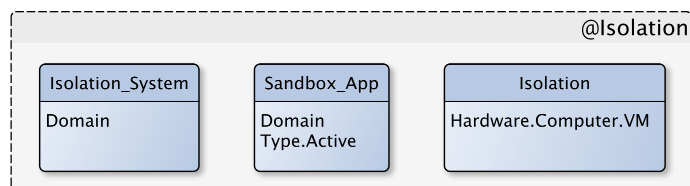

Figure 3 – Classe d'isolation

Le listing 9

```
1 boolean Isolation_System (Domain SCDom) {
2  enforcement {
3    create_domain (SCDom);
4    isolate (SCDom);
```

```
5  }
6  assurance {
7    return check_isolated (SCDom);
8  }
9 }
```

Listing 9 – Propriété d'isolation système d'un domaine

La propriété **Sandbox_App** du listing 10 permet d'isoler une application ou un service du reste du système. La propriété garantit la confidentialité de l'ensemble des fichiers de l'application, ainsi que leur intégrité. Certains fichiers peuvent cependant être lus, édités ou exécutés par un administrateur de l'application.

```
1 boolean Sandbox_App (Domain SCDom, Type.Active SCApp) {
2  enforcement && assurance {
3     Isolation_System (SCDom);
4     Confidentiality_access_control (SCDom:(Type.Passive.Data.File=".*"), SCDom:
         SCApp);
5     Integrity (SCDom:(Type.Passive.Data.File.Configuration=".*"), SCDom:SCApp);
6     Integrity (SCDom:(Type.Passive.Data.File.Logs=".*"), SCDom:SCApp);
7     Integrity (SCDom:(Type.Passive.Data.File.Key=".*"), SCDom:SCApp);
8     Integrity (SCDom:(Type.Passive.Data.File.Temporary=".*"), SCDom:SCApp);
9     Integrity (SCDom:(Type.Passive.Data.File.Binary=".*"));
10    Integrity (SCDom:(Type.Passive.Data.File.Executable=".*"));
11    Executable (SCDom:(Type.Passive.Data.File.Executable=".*"), SCDom:SCApp);
12  }
13 }
```

Listing 10 – Propriété d'isolation système d'une application

La propriété du listing 11 permet d'isoler des machines virtuelles du reste du système.

```
1 boolean Isolation (Hardware.Computer.VM SCvm) {
2  enforcement {
3    isolate (SCvm);
4    Confidentiality_access_control (SCvm:(Type.Passive.Data.File=".*"), SCvm:(
        Type.Active=".*"));
5    Integrity (SCvm:(Type.Passive.Data.File=".*"), SCvm:(Type.Active=".*"));
6  }
7  assurance {
8    return check_isolated (SCvm);
9  }
10 }
```

Listing 11 – Propriété d'isolation système de VM

A.2.5 Accès

La propriété d'accès définie au listing 12 permet l'ouverture d'un port réseau pour un ensemble d'adresses IP distantes.

```
1 boolean Access (Net.Port SCLocal, Net.IP SCRemote) {
2  enforcement {
3   open_port (SCLocal, SCRemote);
4  }
5  assurance {
6   return check_port_opened (SCLocal, SCRemote);
7  }
8 }
```

Listing 12 – Propriété d'accès réseau

A.2.6 Authentification

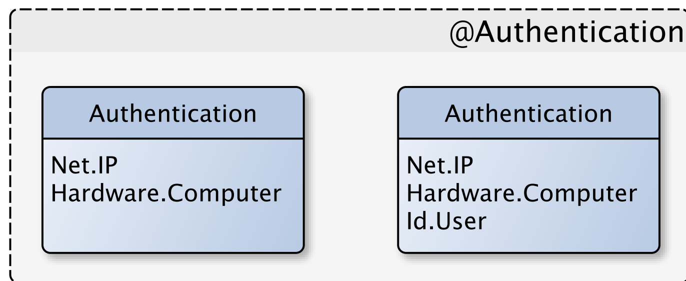

FIGURE 4 – Classe d'authentification

Les propriétés d'authentification des listings 13 et 14 permettent d'autoriser des connections pour certaines machines. La seconde propriété permet également de spécifier quelles identités peuvent être prises après la connexion.

```
1 boolean Authentication (Net.IP SC1, Hardware.Computer SC2) {
2  enforcement {
3   allow_connection (SC1, SC2);
4  }
5  assurance {
6   return try_login (SC1, SC2);
7  }
8 }
```

Listing 13 – Propriété d'authentification

```
1 boolean Authentication (Net.IP SC1, Hardware.Computer SC2, Id.User SC3) {
2  enforcement {
3   allow_connection (SC1, SC2, SC3);
4  }
5  assurance {
6   return try_login (SC1, SC2, SC3);
7  }
8 }
```

Listing 14 – Propriété d'authentification avec mise à jour du contexte

A.2.7 Assurance

La classe d'assurance décrit comment les tests d'assurance sont effectués et comment leur résultats sont récupérés.

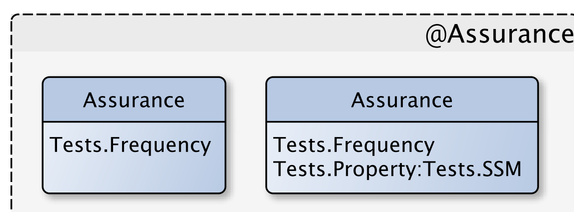

FIGURE 5 – Classe d'assurance

La propriété du listing 15 exécute les tests et récupère les résultats à une fréquence fixe. Celle du listing 16 restreint les tests qui sont effectués en précisant des propriétés et/ou des mécanismes.

```
boolean Assurance (Tests.Frequency SCFreq) {
 enforcement {
  run_assurance_tests (SCFreq);
  collect_assurance_results (SCFreq);
 }
}
```

Listing 15 – Propriété d'assurance

```
boolean Assurance (Tests.Frequency SCFreq, Tests.Property:Tests.SSM SC2) {
 enforcement {
  run_assurance_tests (SCFreq, SC2);
  collect_assurance_results (SCFreq, SC2);
 }
}
```

Listing 16 – Propriété d'assurance

A.3 Grammaire

Les différents éléments du langage ont été définis au chapitre 3. Cette section présente la grammaire du langage d'expression des besoins de sécurité. Ce langage est constitué de deux sous-langages : celui permettant de définir la politique de sécurité (utilisé par l'administrateur de l'architecture logicielle) et celui permettant de configurer les propriétés (utilisé par l'expert en sécurité).

Le listing 17 présente la grammaire d'expression de la politique. Il s'agit donc de la partie écrite par l'administrateur de l'architecture logicielle (ou générée à partir d'une interface graphique) qui décrit les besoins de sécurité d'une architecture logicielle. La grammaire explicite comment exprimer des contextes et comment les utiliser pour définir des instances de propriétés.

```
<attribute>        ::= ( <key> = <value> )
<key>              ::= string(.string)*
<value>            ::= "regex"

<context>          ::= <contextName> = <contextDef> ;
<contextName>      ::= string
<contextDef>       ::= <attribute>(:<attribute>)*
                   | $<contextName>(:$<contextName>)*(:<attribute>)*

<property>         ::= (''|@)<propName> ( <argumentsList> ) ;
<propName>         ::= string
<argumentsList>    ::= <argument>(, <argument>)*
<argument>         ::= <contextName>
                   | " <contextNameRegex> "
```

Listing 17 – Grammaire de la politique

Le listing 18 détaille la grammaire utilisée pour définir les classes et les propriétés de sécurité. Les classes sont composées d'autres classes ou de propriétés, auxquelles on associe un score. Les propriétés sont formées de variables, de capacités, d'appels à d'autres propriétés, de blocs conditionnels et de boucles itératives. Leur fonctionnement peut ainsi être décrit de manière détaillée. Cette grammaire correspond au langage tel que vu par l'expert en sécurité.

```
<class>            ::= @<className> { (<classStatement>)+ } }
```

```
2  <className>       ::= string
3  <classStatement> ::= <score> <propName> (<propArgs>) ;
4                     | <score> @<className> ;
5  <score>          ::= integer | ''
6
7  <property>       ::= <type> <propName> (<propArgs>) <block>
8  <type>           ::= boolean | <contextType>
9  <contextType>    ::= <key>(:<key>)*
10 <key>            ::= string(.string)*
11 <propName>       ::= string
12 <propArgs>       ::= <argument> (, <argument>)*  | ''
13 <argument>       ::= <type> <argName>
14 <argName>        ::= string
15 <block>          ::= { (<statement>)* }
16 <statement>      ::= <variableDecl> ;
17                    | <capability> ;
18                    | <propCall> ;
19                    | <ifBlock>
20                    | <forBlock>
21                    | <returnStatement> ;
22
23 <variableDecl>   ::= <type> <varName> ((=|+=|&=)(<contextType>|<capability>))?
24 <varName>        ::= string
25
26 <capability>     ::= (<varName> ->)? <capName> ( <argumentsList> )
27 <capName>        ::= string
28 <argumentList>   ::= <argument> (, <argument>)*
29 <argument>       ::= <argName>(:<argName>)*
30
31 <propCall>       ::= <propName> (<argumentsList>)
32
33 <ifBlock>        ::= if (<testStatement>) <block> (else <block>)?
34 <testStatement>  ::= <var>
35                    | (NOT)? <inStatement>
36                    | (NOT)? <contextArg> == (<contextArg>|<capability>)
37                    | (NOT)? <andStatement>
38 <var>            ::= (NOT)? <varName>
39 <contextArg>     ::= <varName>(.<contextType>)?
40 <andStatement>   ::= AND { (<propCall>; | <capability>;)* }
41
42 <forBlock>       ::= for (<inStatement>) <block>
43 <inStatement>    ::= <name> IN (<capability> | <name>)
44
45 <returnStatement>::= return (true | false | <capability> | <var> | <propCall> |
       <andStatement>)?
```

Listing 18 – Grammaire de définition des propriétés

B Expérimentation

Cette section présente les politiques de sécurité utilisées dans le cas d'usage développé au chapitre 6.

B.1 Contextes

Le listing 19 contient les différents contextes utilisés pour définir la politique de sécurité. Par simplicité, ces contextes sont définis sur toutes les machines du cas d'usage, mais des sous-ensembles pourraient être définis pour chacune d'entre elles.

```
// Machines
HOST                   = (Hardware.Computer = "nova-compute-node");
HostReverseProxy       = (Hardware.Computer.VM = "reverse-proxy-nginx");
HostClient             = (Hardware.Computer.VM = "client");
HostServerWEB          = (Hardware.Computer.VM = "apache-see");
HostServerSQL          = (Hardware.Computer.VM = "mysql-see");
VMs                    = (Hardware.Computer.VM = ".*");

// Utilisateurs
CloudProvider  = (Identity.Username="idCloudProvider"):(Identity.Role="
    StandardUser|PackageAdmin|SysnetAdmin");
TenantOperator = (Identity.Username="idTenantOperator"):(Identity.Role="
    StandardUser|PackageAdmin|SysnetAdmin");
TenantAdmin    = (Identity.Username="idTenantAdmin"):(Identity.Role="
    StandardUser|LoggingAdmin|WEBAdmin");
User           = (Identity.Username="idUser"):(Identity.Role="StandardUser");
anyone         = (*);

// Domaines
DomainWEB     = (Domain="App_Web");
DomainWEBAPPS = (Domain="App_Web_Apps");
DomainSQL     = (Domain="App_Database");

// Resources
FileWEB       = DomainWEB:(Type.Passive.Data.File="default");
BinaryWEB     = DomainWEB:(Type.Passive.Data.File.Binary="bin_exec");
ConfigWEB     = DomainWEB:(Type.Passive.Data.File.Configuration="conf");
LogWEB        = DomainWEB:(Type.Passive.Data.File.Logs="logs");

FileWEBAPPS   = DomainWEBAPPS:(Type.Passive.Data.File="default");
BinaryWEBAPPS = DomainWEBAPPS:(Type.Passive.Data.File.Binary="bin_exec");
ConfigWEBAPPS = DomainWEBAPPS:(Type.Passive.Data.File.Configuration="conf");
TmpWEBAPPS    = DomainWEBAPPS:(Type.Passive.Data.File.Temporary="tmp");
LogWEBAPPS    = DomainWEBAPPS:(Type.Passive.Data.File.Logs="logs");

FileSQL       = DomainSQL:(Type.Passive.Data.File="default");
BinarySQL     = DomainSQL:(Type.Passive.Data.File.Binary="bin_exec");
ConfigSQL     = DomainSQL:(Type.Passive.Data.File.Configuration="conf");
LogSQL        = DomainSQL:(Type.Passive.Data.File.Logs="logs");

// Services
ServiceSSH     = (Type.Active.Service="SSH");
ServiceWEB     = HostServerWEB:DomainWEB:(Type.Active.Service="App_Web"):(
    Identity.Role="System");
ServiceWEBAPPS = HostServerWEB:DomainWEBAPPS:(Type.Active.Service="App_Web_Apps
    "):(Identity.Role="System");
ServiceSQL     = HostServerSQL:DomainSQL:(Type.Active.Service="App_Database"):(
    Identity.Role="System");

// Ports et IP
WEBPort     = (Network.Port="80");
WEBAPPSPort = (Network.Port="9999");
MysqlPort   = (Network.Port="3306");
SSHPort     = (Network.Port="22");
AnyIP       = (Network.IP=".*");

// Assurance
Frequency = (Tests.Frequency="10m");
```

Listing 19 – Contextes des machines

B.2 Politiques

Cette section contient les politiques de sécurité pour les machines du cas d'usage (chapitre 6). Des détails sont donnés à la section 6.2. Le listing 20 présente la politique pour les machines physiques.

```
1 Authentication (anyone, serviceSSH, "systemUser|cloudProvider");
2 Access (SSHPort, anyIP);
3 Isolation (VMs);
4
5 Assurance (Frequency);
```

Listing 20 – Politique des machines physiques

Le listing 21 présente la politique pour la VM Proxy, dans le cas où un seul triplet de VM est déployé. Dans les tests utilisant plus de triplets, la propriété de confidentialité réseau est dupliquée autant de fois que nécessaire.

```
1 @Isolation (DomainWEB, ServiceWEB);
2
3 Confidentiality_access_control (LogWEB, TenantOperator);
4 Confidentiality_access_control (LogWEB, TenantAdmin);
5
6 Confidentiality_access_control (ConfigWEB, TenantAdmin);
7 Confidentiality_access_control (FileWEB, TenantAdmin);
8
9 Integrity (ConfigWEB, TenantAdmin);
10
11 Executable (BinaryWEB, TenantAdmin);
12
13 Authentication (HostReverseProxy, ServiceSSH, "TenantAdmin|TenantOperator" );
14 Authentication (HostClient, ServiceSSH, "TenantAdmin|TenantOperator" );
15
16 Access (WEBPort, AnyIP);
17 Access (SSHPort, HostReverseProxy);
18 Access (SSHPort, HostClient);
19
20 Confidentiality (HostReverseProxy, "HostClient|HostServerWEB|HostServerSQL");
21
22 Assurance (Frequency);
```

Listing 21 – Politique des VM Proxy

Les listings 22, 23 et 24 présentent les politiques pour les VM Apache, MySQL, et Client.

```
1 @Isolation (DomainWEB, ServiceWEB);
2 @Isolation (DomainWEBAPPS, ServiceWEBAPPS);
3
4 Confidentiality_access_control (LogWEB, TenantOperator);
5 Confidentiality_access_control (LogWEB, TenantAdmin);
6 Confidentiality_access_control (LogWEBAPPS, TenantAdmin);
7
8 Confidentiality_access_control (ConfigWEB, TenantAdmin);
9 Confidentiality_access_control (FileWEB, TenantAdmin);
10 Confidentiality_access_control (ConfigWEBAPPS, TenantAdmin);
11 Confidentiality_access_control (FileWEBAPPS, TenantAdmin);
12
13 Integrity (ConfigWEB, TenantAdmin);
14 Integrity (ConfigWEBAPPS, TenantAdmin);
15
```

```
16 Executable (BinaryWEB, TenantAdmin);
17 Executable (BinaryWEBAPPS, TenantAdmin);
18
19 Authentication (HostServerWEB, ServiceSSH, "TenantAdmin|TenantOperator");
20 Authentication (HostReverseProxy, ServiceSSH, "TenantAdmin|TenantOperator");
21 Authentication (HostClient, ServiceWEBAPPS, "User");
22
23 Access (SSHPort, HostServerWEB);
24 Access (SSHPort, HostReverseProxy);
25 Access (WEBAPPSPort, HostClient);
26 Access (WEBPort, AnyIP);
27
28 Confidentiality (HostReverseProxy, "HostClient|HostServerWEB|HostServerSQL");
29
30 Assurance (Frequency);
```

Listing 22 – Politique système des VM Apache

```
1 @Isolation (DomainSQL, ServiceSQL);
2
3 Confidentiality_access_control (LogSQL, TenantOperator);
4 Confidentiality_access_control (LogSQL, TenantAdmin);
5
6 Confidentiality_access_control (ConfigSQL, TenantAdmin);
7 Confidentiality_access_control (FileSQL, TenantAdmin);
8
9 Integrity (ConfigSQL, TenantAdmin);
10
11 Executable (BinarySQL, TenantAdmin);
12
13 Authentication (HostReverseProxy, ServiceSSH, "TenantAdmin|TenantOperator" );
14 Authentication (HostServerSQL, ServiceSSH, "TenantAdmin|TenantOperator" );
15 Authentication (HostServiceWEB, ServiceSQL, "User" );
16
17 Access (MysqlPort, HostServerWEB);
18 Access (SSHPort, HostReverseProxy);
19 Access (SSHPort, HostServerDB);
20
21 Confidentiality (HostReverseProxy, "HostClient|HostServerWEB|HostServerSQL");
22
23 Assurance (Frequency);
```

Listing 23 – Politique des VM MySQL

```
1 Access (SSHPort, AnyIP);
2
3 Confidentiality (HostReverseProxy, "HostClient|HostServerWEB|HostServerSQL");
4
5 Assurance (Frequency);
```

Listing 24 – Politique des VM Clientes

Aline BOUSQUET

Application et assurance autonomes de propriétés de sécurité dans un environnement d'informatique en nuage

Les environnements d'informatique en nuage sont des environnements hétérogènes et dynamiques, ce qui les rend complexes à sécuriser. Dans cette thèse, nous proposons un langage et une architecture permettant d'exprimer et d'appliquer des propriétés de sécurité dans un environnement en nuage. Le langage permet à un client de l'informatique en nuage d'exprimer des besoins de sécurité sans préciser comment ils seront appliqués. Le langage se base sur des contextes abstrayant les ressources et des propriétés correspondant aux besoins de sécurité.
Les propriétés sont ensuite appliquées en utilisant les mécanismes de sécurité disponibles (tels que SELinux, PAM, iptables ou firewalld) via une architecture autonome. Cette architecture permet d'abstraire et de réutiliser les capacités de sécurité des mécanismes existants. Une propriété de sécurité est ainsi définie comme une combinaison de capacités et peut être appliquée grâce à la collaboration de plusieurs mécanismes. Les mécanismes sont alors automatiquement configurés en accord avec les propriétés établies par l'utilisateur.
L'architecture dispose aussi d'un système d'assurance qui lui permet de détecter une défaillance d'un mécanisme ou une erreur d'application. L'architecture peut ainsi répondre aux problèmes rencontrés, par exemple en ré-appliquant des propriétés avec d'autres mécanismes. De plus, le système d'assurance fournit une évaluation de l'application des propriétés.
La thèse propose ainsi un système autonome d'application et d'assurance de la sécurité dans des environnements hétérogènes.
Mots clés : sécurité, propriétés de sécurité, assurance, nuage informatique, systèmes hétérogènes

Autonomic enforcement and assurance of security properties in a Cloud

Cloud environnements are heterogeneous and dynamic, which makes them difficult to protect. In this thesis, we introduce a language and an architecture that can be used to express and enforce security properties in a Cloud. The language allows a Cloud user to express his security requirements without specifying how they will be enforced. The language is based on contexts (to abstract the resources) and properties (to express the security requirements).
The properties are then enforced through an autonomic architecture using existing and available security mechanisms (such as SELinux, PAM, iptables, or firewalld). This architecture abstracts and reuses the security capabilities of existing mechanisms. A security property is thus defined by a combination of capabilities and can be enforced through the collaboration of several mechanisms. The mechanisms are then automatically configured according to the user-defined properties.
Moreover, the architecture offers an assurance system to detect the failure of a mechanism or an enforcement error. Therefore, the architecture can address any problem, for instance by re-applying a property using different mechanisms. Lastly, the assurance system provides an evaluation of the properties enforcement.
This thesis hence offers an autonomic architecture to enforce and assure security in Cloud environnements.
Keywords : security, security properties, assurance, cloud computing, heterogeneous systems

INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES CENTRE VAL DE LOIRE

Laboratoire d'Informatique
Fondamentale d'Orléans
Bâtiment IIIA Rue Léonard de Vinci
B.P. 6759
F-45067 ORLEANS Cedex 2

LABORATOIRE D'INFORMATIQUE LIFO FONDAMENTALE D'ORLEANS