Formalisation et garantie de propriétés de sécurité système : application à la détection d'intrusions

Jérémy Briffaut

UNIVERSITE D'ORLEANS

THESE PRESENTEE A L'UNIVERSITE D'ORLEANS

POUR OBTENIR LE GRADE DE

DOCTEUR DE L'UNIVERSITE D'ORLEANS

Discipline : Informatique

PAR

Jérémy BRIFFAUT

Titre de la Thèse :

Formalisation et garantie de propriétés de sécurité système : application à la détection d'intrusions

Soutenue le **: 13 décembre 2007**

MEMBRES DU JURY

- M. Mathieu Blanc	Ingénieur de recherche CEA	**Examinateurs**
- M. Jean-Michel Couvreur	Professeur	
- M. Jean-François Lalande	Maître de conférence	
- M. Jean Goubault-Larrecq	Professeur	**Président du jury**
- M. Christian Toinard	Professeur	**Directeur de thèse**
- M. Fabrice Bouquet	Maître de conférence habilité	**Rapporteurs**
- M. Gaétan Hains	Professeur	

Remerciements

Je remercie M. Fabrice Bouquet et M. Gaetan Hains pour avoir accepté d'évaluer mes travaux en étant rapporteurs de cette thèse. Je tiens à remercier les membres de mon jury de thèse de doctorat, M. Mathieu Blanc, M. Jean-Michel Couvreur, M. Jean-François Lalande et M. Jean Goubault-Larrecq pour avoir présidé mon jury de thèse.

Je tiens à remercier mon directeur de thèse, M. Christian Toinard pour son soutien et son aide durant la durée de mes recherches et m'avoir offert la possibilité de réaliser cette thèse, mais aussi la région Centre et le LIFO pour avoir financé cette thèse et m'avoir ainsi permis de réaliser ces travaux de recherche. Je tiens à remercier tous les membres de l'équipe Sécurité et Distribution des Système, M. Mathieu Blanc, M. Patrice Clemente, M. Jean-François Lalande et M. David Teller, ainsi que M. Martial Szpieg professeur à l'ENSI de Bourges, pour les nombreuses discussions enrichissantes et pour toute l'aide qu'ils m'ont fourni durant cette thèse. Je remercie aussi l'ENSI de Bourges pour m'avoir accueilli dans ses locaux et m'avoir offert un cadre agréable de travail.

Je voudrais remercier mes amis, ma famille pour leurs encouragements et leur soutien et surtout ma femme Kathaleen pour ses nombreuses relectures.

Table des matières

Table des figures

Liste des Algorithmes

Liste des tableaux

Glossaire

ACL	*Access Control List* : Liste de contrôle d'accès	page 40
ASLR	*Address Space Layout Randomization* : mécanisme de randomisation de l'espace d'adressage présent dans grsecurity	page 188
AVC	*Access Vector Cache* : Composant de SELinux servant de cache pour les vecteurs d'interactions	page 185
Biba	Modèle d'intégrité conçu par K. J. Biba	page 44
BLP	Modèle de confidentialité Bell-LaPadula, nommé d'après ses auteurs D. E. Bell et L. J. La Padula	page 42
DAC	*Discretionary Access Control* : Contrôle d'accès discrétionnaire	page 39
DTE	*Domain and Type Enforcement* : Modèle de protection associant des domaines aux sujets et des types aux objets	page 45
DTEL	*Domain and Type Enforcement Language* : Langage de configuration du modèle DTE	page 45
DTOS	*Distributed Trusted Operating System :* système d'exploitation sécurisé développé par la NSA	page 185
HIDS	*Host Based Intrusion Detection System :* Système de détection d'intrusions hôte	page 27
HRU	Modèle théorique de DAC établi par Harrison, Ruzzo et Ullman	page 40
IDS	*Intrusion Detection System :* Système de détection d'intrusions	page 27
IPSEC	*Internet Protocol Security* : Sécurité du protocole Internet (IP)	page 46
ITSEC	*Information Technology Security Evaluation Criteria* : Critères d'évaluation de la sécurité des systèmes d'information, définis par un comité de pays européens	page 21
LSM	*Linux Security Modules* : Ensemble de fonction pour l'implantation de modules de sécurité dans le noyau Linux	page 185
MAC	*Mandatory Access Control* : Contrôle d'accès mandataire	page 42
MLS	*MultiLevel Security* : Modèle de sécurité multi-niveaux	page 44

MTAM	*Monotonic Typed Access Matrix* : Modèle de matrice de contrôle d'accès typée à action mono-atomique conçu par Ravi S. Sandhu	page 41
NIDES	Outils de détection d'intrusions	page 29
NIDS	*Network Based Intrusion Detection System :* Système de détection d'intrusions réseau	page 27
NSA	*National Security Agency* : Agence de sécurité nationale américaine, à l'origine du projet SELinux	page 185
PaX	Patch pour le noyau Linux apportant notamment la protection des pages mémoire non-exécutables	page 188
ROC	*Receiver Operating Characteristic :* Courbe représentant le taux de détection d'un IDS en fonction du taux de faux positif	page 27
SELinux	*Security-Enhanced Linux :* mécanisme de contrôle d'accès mandataire pour le système Linux développée par la NSA	page 185
SPM	*Sandhu's Schematic Protection Model* : Modèle de protection conçu par Ravi S. Sandhu	page 41
TAM	*Typed Access Matrix* : Modèle de matrice de contrôle d'accès typée conçu par Ravi S. Sandhu	page 41
TCP	*Transmission Control Protocol* : protocole de transmission réseau en mode connecté	page 188
TCSEC	*Trusted Computer System Evaluation Criteria* : Critères d'évaluation de la sécurité des systèmes informatiques, édités par le Ministère de la Défense américain. Aussi appelé *Orange Book*	page 21
TPE	*Trusted Path Execution* : mécanisme de sécurité intégré à grsecurity	page 185
UDP	*User Datagram Protocol* : protocole de transmission réseau en mode non connecté	page 188
UID	*User IDentifier* : identifiant numérique unique à chaque utilisateur dans les systèmes d'exploitation de type Unix	page 185

Chapitre 1

Introduction

1.1 Contexte des travaux

En terme de sécurité, il existe trois grandes classes de propriétés : la confidentialité, l'intégrité et la disponibilité. Les autres propriétés peuvent être vues comme des cas particuliers des ces trois classes (par exemple, l'authentification peut être exprimée comme l'intégrité de la source, et la non répudiation comme l'intégrité de la source et de la destination). Dans notre étude nous nous intéressons aux propriétés de confidentialité et d'intégrité. Même s'il peut y avoir des perspectives de nos travaux en direction de la disponibilité, dans cette thèse nous ne traiterons pas cette classe de propriété.

La confidentialité et l'intégrité concernent à la fois les opérations locales à un système et les transmissions sur un réseau informatique. Dans cette thèse, nous traitons les actions effectuées localement sur un système et les propriétés associées. Les transmissions informatiques sortent du cadre de cette étude. Il s'agit en général de protocoles qui sont standardisés pour garantir par exemple la confidentialité des données circulant sur le réseau. Cependant, les deux aspects sont indissociables. En effet, si les machines n'offrent aucune garantie, on ne peut pas obtenir de propriété de sécurité à l'échelle d'un système réparti c'est-à-dire d'un ensemble de machines reliées par un réseau. De plus, de nombreuses attaques des protocoles de sécurité exploitent des failles du système et des applications pour violer les propriétés de sécurité. Par exemple, sur IPSec une attaque du système permettant de voler les clés compromet la confidentialité et l'intégrité des échanges sur le réseau. On voit donc que la sécurisation d'une machine (système et application) est indispensable à la sécurité des réseaux informatiques.

L'objectif de cette thèse est de garantir des propriétés d'intégrité et de confidentialité à l'échelle d'une machine, c'est-à-dire des systèmes pris individuellement. Il s'agit donc d'une approche système où 1) le noyau du système d'exploitation garantit certaines propriétés d'intégrité et de confidentialité et 2) d'autres propriétés peuvent être obtenues par analyse des actions réalisées par les processus. Pour nous, il s'agit en pratique de contrôler les interactions réalisées au sein d'un système pour 1) offrir certaines propriétés correspondant à un mécanisme de protection système et 2) offrir les autres propriétés via un mécanisme extérieur au noyau en détectant les interactions qui violent les propriétés attendues. La protection permet d'éviter que certaines actions illégales apparaissent tandis que la détection fournit des alertes lorsque des actions illégales sont effectuées. Notre étude permet à la fois la protection et la détection. Cependant, l'approche proposée est tout à fait générale car elle repose sur une analyse des interactions qui peut éventuellement être implantée au niveau du noyau. Lorsque les performances requises par les processus ne permettent pas une telle implantation au sein du noyau, l'analyse est effectuée en dehors de l'espace noyau sous forme d'une détection des violations.

Lors d'une attaque ou d'une tentative d'intrusion sur un système, le ou les attaquants engendrent

un ensemble d'actions (i.e. d'interactions) violant une des propriétés de sécurité. Cette violation se traduit par l'exécution d'une seule interaction ou d'un ensemble d'interactions. Notre étude des propriétés de sécurité fait clairement apparaître la nécessité de considérer les séquences d'interactions ou leurs corrélations. Nous montrons qu'une formalisation manque pour exprimer, comparer et étendre les propriétés de sécurité de la littérature. Notre travail propose une méthode générale qui garantit plus de propriétés, qui fonctionne en pratique sur des systèmes et qui a été expérimentée pendant plus d'un an afin de détecter des attaques qui ne sont pas traitées par les autres solutions.

S'il s'agit d'une approche système, notre méthode permet aussi de traiter des actions associées à une transmission. En effet, les interactions que nous analysons sont les appels des processus aux services du système d'exploitation et certaines de ces interactions sont effectuées sur des ressources réseau. Il peut par exemple s'agir d'une interaction entre un processus (sujet) et un descripteur réseau (objet) pour effectuer des opérations d'émissions et de réceptions de message. Ces interactions entre les processus et les ressources réseau peuvent ainsi être protégées. De plus, nous verrons en conclusion que l'approche étant générale, elle a d'hors et déjà été étendue 1) pour permettre le déploiement d'une politique de sécurité système sur tout un réseau et 2) pour faciliter l'administration d'une politique de sécurité réseau.

1.2 Apports de la thèse

Nous montrerons à travers l'état de l'art que les solutions de protection actuelles offrent des propriétés limitées au contrôle d'une seule interaction. Par ailleurs, les approches de détection d'intrusions offrent des propriétés d'intégrité et de confidentialité qui ne traitent pas les actions complexes composées d'une suite d'interactions. Elles se limitent en pratique à l'analyse d'une interaction ou d'une combinaison logique d'interactions. Seuls quelques IDS traitent d'une suite d'interactions particulière, les flux d'informations. Ainsi, tout un ensemble de propriétés de sécurité reste sans réponse. Par exemple, les propriétés générales d'intégrité et de confidentialité définies par [ITSEC 1991] ne sont actuellement garanties ni par les approches contrôle d'accès ni par les approches détection d'intrusions. En effet, pour garantir ces propriétés il faut pouvoir analyser une suite d'interactions qui par transitivité permet de violer l'intégrité ou la confidentialité. Par exemple, il peut s'agir d'un processus qui en exécutant différents binaires arrive à obtenir des privilèges plus élevés. En pratique, cela correspond à une suite d'appels système qui forme une fermeture transivite permettant d'accéder au privilège. Ainsi, les solutions actuelles de contrôle d'accès ou de détection d'intrusions ne traitent que des interactions et ne prennent pas en compte les séquences, c'est-à-dire les fermetures transitives des dépendances causales.

Par ailleurs, il manque un formalisme permettant de comparer les propriétés des différentes solutions de la littérature. En effet, non seulement chaque auteur adopte son propre formalisme pour décrire les propriétés de sa solution, mais souvent la formalisation des propriétés n'est pas précise et elle n'est pas réutilisable pour décrire d'autres propriétés de sécurité. Cette absence d'un formalisme ne permet pas de définir les propriétés sans ambiguïtés et ne facilite pas leurs comparaisons. Si au départ nous nous sommes intéressés à fournir un mécanisme de détection d'intrusions, nous sommes arrivés à la nécessité de définir un langage pour décrire les actions du système. Ce langage permet non seulement de définir formellement toutes les propriétés classiquement évoquées dans la littérature mais aussi de les étendre et d'en proposer de nouvelles. L'avantage est que la formalisation des propriétés devient précise. Cette précision permet non seulement d'éviter les ambiguïtés mais nous a aussi permis de proposer une méthode générale pour implanter les propriétés d'intégrité et de confidentialité d'un système. Ainsi, toute propriété exprimée dans notre langage peut être implantée soit

sous forme de contrôle d'accès soit, pour des raisons de performances, sous forme d'une détection d'intrusions. Par ailleurs, et même si ce n'était pas l'objectif de ce langage, celui-ci nous a permis de comparer les propriétés des solutions de la littérature et d'en définir précisément les limites.

Notre langage repose essentiellement sur une notion de fermeture transitive d'une relation de dépendance causale entre interactions. Notre définition de la dépendance causale est originale, en effet même si elle repose sur le principe de causalité utilisé par différents auteurs, il s'agit d'une propriété adaptée aux interactions du système. Cette définition est plus précise que celles utilisées par ailleurs dans la littérature. En effet, on trouve la plupart du temps une définition informelle de la relation de causalité ou une simple citation de la notion définie par [Lamport 1978] pour les messages. La fermeture transitive de notre dépendance causale permet de définir la notion de séquence c'est-à-dire d'une suite d'interactions causalement liées qui relie deux entités. Par exemple, un processus A peut envoyer des signaux à d'autres processus pour finalement déclencher une écriture dans un fichier B. Cette séquence de signaux est donc la cause de la modification du fichier et il peut s'agir d'une violation de l'intégrité de B si A n'a pas le droit de modifier ce fichier. Notre langage offre une modélisation tout à fait générale des différentes classes d'activités observables sur un système. Ce langage distingue les interactions, les séquences et leurs corrélations. Notre étude porte donc sur des propriétés correspondant à des actions observables, c'est-à-dire où l'ensemble des événements est visible sur le système. Pour les activités non observables d'un point de vue système, les dépendances entre les interactions doivent être déduites. Cette classe concerne en général les propriétés de disponibilité ou des attaques utilisant des éléments extérieurs non observables sur le système (notamment, des échanges verbaux). Nous ne traitons pas ces cas.

L'expressivité de notre langage, est démontrée par le fait qu'il nous permet de définir toutes les propriétés d'intégrité et de confidentialité système rencontrées classiquement dans la littérature. Il permet une formalisation de la propriété générale d'intégrité définie dans [ITSEC 1991] ainsi que les cas particuliers de *non-interférence* (intégrité des sujets), la *base de confiance* (intégrité des domaines) et le modèle BIBA. Il supporte aussi la propriété générale de confidentialité définie dans [ITSEC 1991] ainsi que les modèles BLP. Il traite aussi les *abus de privilèges*, où le privilège concerne soit l'intégrité soit la confidentialité. Pour cette dernière classe, nous traitons des propriétés classiques comme la *séparation de privilèges*, les *exécutables de confiance* et le *respect des règles de contrôle d'accès*.

Ce langage offre un pouvoir d'expressivité très large puisqu'il nous permet d'étendre des propriétés existantes et de proposer de nouvelles propriétés. Ainsi nous définissons deux nouvelles propriétés, la *cohérence d'accès aux données* pour la confidentialité et l'*absence de changement de contexte* pour l'abus de privilèges.

La thèse propose une méthode pour mettre en oeuvre ce langage sur un système afin de garantir les propriétés de sécurité qu'il peut exprimer. La méthode part de la définition d'une politique de contrôle d'accès. Cette politique peut être minimale, c'est-à-dire réduite à la seule énumération de tous les exécutables (sujets) et de toutes les autres ressources (objets). Cette politique permet de calculer un graphe des interactions entre sujets et objets. Ce graphe nous sert alors à calculer un graphe de dépendance causale au moyen d'une simple table qui permet d'orienter les interactions dans le sens de leur effet. Nous montrons qu'il y a équivalence entre ce graphe et toutes les relations de causalité observables. Nous calculons aussi différentes réductions de ce graphe pour exprimer des cas particuliers de séquence (transitions de contexte ou flux d'informations). A partir du moment où nous sommes capables de manipuler les interactions et les séquences qui sont les terminaux de notre langage, nous pouvons proposer une implantation pour toutes les propriétés exprimées dans ce langage.

Cette mise en oeuvre du langage, est appliquée à la détection d'intrusions. Ainsi, nous proposons un outil appelé PIGA (Policy Interaction Graph Analysis) qui prend en entrée une politique de sécurité (incluant des propriétés exprimées dans notre langage) et une politique de contrôle d'accès. Cette

dernière permet la construction des graphes de dépendance causale. PIGA extrait de ces graphes une base des activités qui violent les propriétés de sécurité. L'outil utilise les traces des appels système pour détecter l'apparition de ces activités illicites.

L'outil PIGA est facilement utilisable. En effet, l'administrateur n'a qu'à définir un ensemble de propriétés dans un langage textuel simple. Ainsi, par la simple définition de deux lignes de propriétés, il peut obtenir la détection de toutes les violations de l'intégrité ou de la confidentialité du système. Si cette politique génère trop de fausses alertes, l'administrateur peut la raffiner. Pour cela, il dispose de toutes les propriétés formalisées dans notre langage.

Notre outil a été expérimenté pendant un an sur un pot de miel. Il a permis de détecter toutes les attaques qui violent l'intégrité et la confidentialité. Ces attaques conduisent à des sessions complètes d'actions illégales. Grâce aux alertes levées par PIGA, nous avons pu aisément identifier et caractériser des sessions complètes d'attaques. Notre expérimentation montre donc l'efficacité de l'approche puisqu'elle permet de garantir un ensemble de propriétés bien plus large que les autres outils de détection d'intrusions système en détectant des attaques non connues et qui violent les propriétés souhaitées.

Les résultats de cette thèse contiennent non seulement un cadre général et formel pour garantir des propriétés de sécurité mais aussi un outil directement utilisable en pratique pour détecter les violations de ces propriétés. De nombreuses perspectives sont possibles pour rendre son usage encore plus aisé et optimiser le travail des administrateurs, notamment l'implantation d'un compilateur permettant de générer automatiquement les algorithmes pour les propriétés.

1.3 Plan du mémoire

Chaque chapitre peut se lire indépendamment. Nous recommandons de lire le chapitre état de l'art, en dernier. En effet, s'il donne de nombreux détails sur les différentes solutions de détection d'intrusions et de protection, seule sa conclusion est utile pour la suite du manuscrit. Le lecteur peut donc aller directement au chapitre 3 puisque celui-ci propose une méthode pour comparer les solutions de la littérature.

Dans le chapitre 2, nous définissons les propriétés de sécurité et la notion de politique de sécurité. Ensuite, nous décrivons les systèmes de détection d'intrusions (détection par scénarios, détection d'anomalies, détection paramétrée par une politique) et les modèles de contrôles d'accès (les familles DAC, MAC, RBAC). Ce chapitre conclut sur l'absence de solutions pour garantir les propriétés d'intégrité et de confidentialité d'un système tel que défini dans [ITSEC 1991].

Le chapitre 3 propose une formalisation permettant de classifier les actions qui peuvent être conduites sur un système. Notre classification définit les interactions (appels système), ce qui réutilise largement les notions de la littérature. Ensuite, nous donnons une proposition originale pour 1) la dépendance causale entre interactions, 2) la notion de séquence (fermeture transitive des dépendances causales) et 3) les corrélations mêlant interactions et séquences. Nous définissons ensuite un langage supportant trois classes d'activité : interactions, séquences et corrélations. Nous utilisons ce langage pour définir les propriétés de sécurité. Ainsi, un ensemble de propriétés d'intégrité et de confidentialité sont ainsi définies formellement. Cette formalisation permet alors de comparer les différentes solutions de la littérature. Ce chapitre conclut sur la nécessité de définir un nouveau modèle afin de pouvoir garantir des propriétés ou détecter leurs violations.

Le chapitre 4 propose un modèle permettant de garantir toute propriété de sécurité exprimée dans notre langage. Notre proposition est de construire le graphe d'interactions à partir d'une politique de protection. Ensuite, nous construisons le *graphe de dépendance causale* qui contient l'ensemble des dépendances causales de la politique de protection. Nous contractons ce graphe pour deux dépen-

dances particulières (la transition et le transfert d'information). Nous montrerons qu'il y a équivalence entre l'existence d'un chemin dans un de ces trois graphes et une séquence observable sur le système. Cela a une importance pratique importante puisque toute séquence observable est contenue dans un de ces graphes. Pour chaque propriété formalisée dans notre langage, nous proposons un algorithme qui permet d'énumérer toutes les activités violant cette propriété. Ces algorithmes correspondent à une implantation de notre langage. Ces algorithmes peuvent être utilisés par un mécanisme de protection pour garantir les propriétés. Ils peuvent aussi servir à réaliser un mécanisme pour détecter les violations des propriétés comme présenté dans le chapitre suivant.

Le chapitre 5 présente l'outil de détection d'intrusions PIGA utilisant notre modèle. PIGA prend en entrée une politique de protection et une politique de détection. La politique de détection contient un ensemble de propriétés de sécurité telles que définies préalablement. Ainsi, un administrateur définit les propriétés de sécurité souhaitées. PIGA génère les différents graphes, énumère les activités qui violent les propriétés et utilise les traces des appels système pour lever des alertes.

Le chapitre 6 décrit l'implantation et l'expérimentation de PIGA. L'implantation réutilise les politiques de protection pour deux systèmes mandataires (SELinux et GRSECURITY) afin de montrer que notre solution fonctionne pour les mécanismes de protection les plus avancés. Nous présentons l'expérimentation sur une année de ce système. Nous montrons que l'outil permet de détecter des attaques qui correspondent à une signature non reconnue par d'autres IDS. L'outil permet de détecter des attaques complexes et de reconstruire des sessions complètes d'attaque. Enfin, le chapitre 7 résume l'apport de la thèse. Il décrit des travaux annexes concernant les politiques dynamiques et l'administration des systèmes répartis. Ces travaux n'ont pas été décrits dans ce document mais font l'objet de plusieurs publications. Enfin, ce chapitre présente les perspectives. PIGA a été expérimenté sur des politiques MAC parce qu'il s'agit de système innovant et performant en terme de protection. Cependant, l'approche est très facilement transposable pour utiliser des politiques DAC (disponibles sur des systèmes Windows ou Unix traditionnels). Il s'agit sûrement de la perspective la plus immédiate et la plus facilement réalisable. Par ailleurs, nous présentons des améliorations pour des politiques dynamiques.

Chapitre 2

État de l'art

Le besoin de politiques de sécurité pour les systèmes d'information est apparu dès les premières études sur les problématiques de confidentialité et d'intégrité. Elles font partie intégrante des critères d'évaluation de la sécurité, tant au niveau international dans le TCSEC [TCSEC 1985], qu'au niveau européen dans ITSEC [ITSEC 1991]. Ce chapitre présente les travaux menés dans les domaines liés à cette thèse, c'est-à-dire aux travaux visant l'application d'une politique de sécurité.

Nous proposons tout d'abord de définir les notions de propriété de sécurité et de politique de sécurité. Puis nous étudierons les travaux de référence, dans les domaines de la détection d'intrusions et du contrôle d'accès qui visent respectivement à détecter ou endiguer tout abus de la politique. Nous nous intéressons tout d'abord aux mécanismes de détection d'intrusions classiques, puis nous présenterons les approches basées sur la définition d'une politique de sécurité. Enfin, nous étudierons la problématique du contrôle d'accès, d'une part les différents modèles théoriques qui ont été développés pour y répondre, et d'autre part les implantations qui en découlent.

2.1 Politique de sécurité

La définition et l'application d'une politique de sécurité représente le coeur de la sécurité d'un système d'information. Une *politique de sécurité* définit un ensemble des *propriétés de sécurité*, chaque propriété représentant un ensemble de conditions que le système doit respecter pour rester dans un état considéré comme sûr. Une définition incorrecte ou l'application partielle d'une politique peut entraîner le système dans un état non-sûr, autorisant le vol d'informations ou de ressources, la modification d'informations ou la destruction du système. Dans cette section, nous donnons une définition générale des propriétés de sécurité, d'une politique de sécurité et des mécanismes utilisés pour l'application d'une politique.

2.1.1 Propriétés de sécurité

La sécurité des systèmes d'information repose sur trois propriétés fondamentales : la *confidentialité*, l'*intégrité* et la *disponibilité*. L'interprétation de ces trois aspects varie suivant le contexte dans lequel elles sont utilisées. Cette représentation est liée aux besoins des utilisateurs, des services et des lois en vigueur. La définition et l'application de ces propriétés font partie intégrante des critères d'évaluation de la sécurité, tant au niveau international dans le TCSEC [TCSEC 1985], qu'au niveau européen dans ITSEC [ITSEC 1991]. Plusieurs définitions de ces propriétés existent [ITSEC 1991, TCSEC 1985, Bishop 2003], nous proposons dans cette section une synthèse de ces propriétés.

2.1.1.1 Confidentialité

La *confidentialité* repose sur la prévention des accès non autorisés à une information. La nécessité de confidentialité est apparue suite à l'intégration des systèmes d'informations critiques, telles que les organisations gouvernementales ou les industries, dans des zones sensibles. La confidentialité est ainsi définie par l'Organisation Internationale de Normalisation (ISO) comme "*le fait de s'assurer que l'information est seulement accessible qu'aux entités dont l'accès est autorisé*". Plus précisement, la *propriété de confidentialité* peut être définie comme suit :

Définition 2.1.1 (Confidentialité) *Soit I de l'information et soit X un ensemble d'entités non autorisées à accéder à I. La propriété de confidentialité de X envers I est respectée si aucun membre de X ne peut obtenir de l'information de I.*

La propriété de confidentialité implique que l'information ne doit pas être accessible par certaines entités, mais doit être accessible par d'autres. Les membres de l'ensemble X sont généralement définis de manière implicite. Par exemple, lorsque l'on parle d'un document confidentiel, cela signifie que seulement certaines entités connues ont accès à ce document. Les autres, qui par définition ne doivent pas avoir accès à ce document, font partie de l'ensemble X.

2.1.1.2 Intégrité

D'une manière générale, l'*intégrité* désigne l'état de données qui, lors de leur traitement, de leur conservation ou de leur transmission, ne subissent aucune altération ou destruction volontaire ou accidentelle, et conservent un format permettant leur utilisation. Dans le cas d'une ressource, l'intégrité signifie que la ressource "fonctionne" correctement, c'est à dire qu'elle respecte sa spécification. La propriété d'intégrité des données vise à prévenir toute modification non autorisée d'une information. La garantie de la fidélité des informations vis à vis de leur conteneur est connue sous le nom d'*intégrité des données*. La garantie des informations en rapport avec la création ou les propriétaires est connue sous le nom d'*intégrité de l'origine*, plus communément appelée *authenticité*. Plus précisement, la *propriété d'intégrité* peut être définie comme suit :

Définition 2.1.2 (Intégrité) *Soit X un ensemble d'entités et soit I de l'information ou une ressource. Alors la propriété d'intégrité de X envers I est respectée si aucun membre de X ne peut modifier I.*

Tout comme la propriété de confidentialité, les membres de X sont généralement définis de manière implicite. Ainsi, les utilisateurs ayant le droit de modifier une information ou une ressource sont définis de manière explicite (propriétaires de cette information, etc.). Les autres utilisateurs forment alors l'ensemble X.

2.1.1.3 Disponibilité

La *disponibilité* se réfère à la possibilité d'utiliser l'information ou une ressource désirée. Cette propriété est à mettre en parallèle avec la fiabilité car le fait d'avoir un système qui n'est plus disponible est un système défaillant. Dans le cadre de la sécurité, la propriété de disponibilité se réfère au cas où un individu peut délibérément interdire l'accès à certaines informations ou ressources d'un système. Plus précisement, la *propriété de disponibilité* peut être définie comme suit :

Définition 2.1.3 (Disponibilité) *Soit X un ensemble d'entités et soit I une ressource. Alors la propriété de disponibilité de X envers I est respectée si tous les membres de X ont accès à I.*

La définition exacte du terme “accès” dépend des besoins des membres de X, de la nature de la ressource et de l'usage de cette ressource. Par exemple, si un site d'achat en ligne peut nécessiter une heure pour répondre à une requête, alors la propriété de disponibilité peut être considérée comme respectée. Si un serveur médical peut mettre une heure pour répondre à une requête concernant une possible allergie à un anesthésique, alors cette propriété peut être considérée comme non respectée.

2.1.1.4 Principe et Propriétés dérivées

La confidentialité, l'intégrité et la disponibilité sont des concepts fondamentaux de la sécurité. En se basant sur ces définitions, plusieurs cas particuliers, qui ne considèrent qu'un sous-ensemble de conditions d'intégrité, de confidentialité ou de disponibilité, peuvent ainsi être définies. Dans cette partie, nous donnons une définition de quatre de ces cas particuliers : 1) le confinement de processus, 2) le principe du moindre privilège, 3) la séparation de privilège et 4) la non-interférence. Ces propriétés peuvent ainsi être classées dans une des trois classes précédentes en fonction des privilèges qui sont contrôlés.

Confinement de processus Lampson [Lampson 1973] caractérise le problème du confinement par :

Définition 2.1.4 (Problème du confinement) *Le problème du confinement concerne la prévention de la divulgation, par un service (ou un processus), d'information considéré comme confidentiel par les utilisateurs de ce service.*

Lampson définit alors les caractéristiques nécessaires à un processus pour qu'il ne puisse pas divulguer de l'information. Une de ces caractéristiques est liée au fait qu'un processus observable ne doit pas stoker de l'information pour la réutiliser ultérieurement. En effet, si un processus stocke de l'information liée à un utilisateur A et qu'un autre utilisateur B peut observer ce processus, il y a alors un risque que B puisse obtenir cette information. Un processus qui ne stocke pas d'information ne peut donc pas divulguer cette information. A l'extrême, si un processus peut être observé, il ne doit pas non plus effectuer d'opérations. En effet, dans certains cas, un analyste peut reconstituer le flux des événements (ou l'état d'un processus) et en déduire des informations sur les entrées de ce processus. En conclusion, un processus qui ne peut pas être observé et qui ne peut pas communiquer [1] avec d'autres processus ne peut pas divulguer de l'information. Cette propriété est alors appelée l'*isolation totale*.

En pratique, l'isolation totale est difficilement applicable sur un système. Les processus confinés partagent généralement des ressources tels que le processeur, le réseau ou le disque avec d'autres processus non confinés. Les processus non confinés peuvent alors divulguer de l'information provenant de ces ressources partagées. De plus, deux processus peuvent s'échanger de l'information de manière indirecte (sans communication explicite), via des canaux cachés.

Définition 2.1.5 (Canal caché) *Un canal caché est un canal de communication possible mais qui n'était pas prévu, lors de la conception du système, comme canal de communication.*

Supposons, par exemple, qu'un processus p veut transmettre de l'information à un processus q et que ces processus ne peuvent pas communiquer directement. Si ces deux processus partagent une même ressource, par exemple un système de fichiers, alors p peut transmettre de l'information à q

[1] L'observation d'un processus A par un processus B étant considérée comme une communication entre A et B.

via un canal caché. Par exemple, p peut créer un fichier 0 ou 1 pour représenter les bit 0 ou 1 et end pour indiquer la fin de la communication. Ainsi à chaque création d'un fichier, q obtient un bit d'information et supprime le fichier pour obtenir le bit suivant jusqu'à la fin du message symbolisé par end. De ce fait q obtient de l'information de p sans communication directe par seule observation du comportement de p.

Définition 2.1.6 (Confinement de processus) *Soit p un processus, x et y deux entités. La propriété de confinement d'un processus p est garantie si p ne peut pas transférer de l'information de x à une autre entité y.*

Finalement, comme l'indique Lampson, la notion de *confinement de processus* est transitive. Si un processus q est considéré comme confiné contre la divulgation d'information, alors si ce second processus invoque un autre processus q, ce processus doit aussi être confiné sinon il peut divulguer de l'information provenant de p.

De part la difficulté d'implanter l'isolation *totale*, le problème des canaux cachés et la difficulté pour avoir la transitivité, le confinement est une propriété difficile à appliquer sur un système réel. De ce fait, le confinement est généralement appliqué à un sous-ensemble du système, par exemple les processus privilégiés. On parle alors de *confinement partiel* du système.

Moindre privilège Le principe du *moindre privilège* [Saltzer et Schroeder 1975] établit qu'une entité ne devrait jamais avoir plus de privilèges que ceux requis pour compléter sa tâche. L'application de ce principe vise à minimaliser les privilèges associés à chaque entité ou processus du système. Ce principe est lié au confinement de processus. En effet, pour être appliqué, il requiert que les processus soient confinés dans le plus petit domaine d'exécution possible. Ce principe peut ainsi être défini comme suit :

Définition 2.1.7 (Moindre privilège) *Soit x une entité, P un ensemble de privilèges assignés à x et T un ensemble de tâches attribuées à x. Alors, le principe du moindre privilège sur x pour les tâches T est respecté si tous les privilèges de P sont nécessaires pour réaliser T.*

Cette propriété implique qu'une entité qui n'a pas besoin d'un droit d'accès ne doit pas avoir la possibilité d'obtenir ce droit. De plus, si une action requiert l'augmentation des privilèges d'une entité, ces droits supplémentaires doivent être supprimés lorsque cette action est terminée. Par exemple, si une entité a le droit d'écrire ou d'ajouter de l'information dans un fichier et qu'une tâche donnée ne nécessite pas la modification mais seulement l'ajout de données, cette tâche devra être exécutée avec le privilège d'ajout et non d'écriture. Le privilège d'écriture ne devra être accordé que si une modification des données existantes est nécessaire.

Séparation de privilèges Une première définition du principe de la *séparation de privilèges* implique qu'*un système ne doit pas permettre l'acquisition de privilèges via un contrôle basé sur une unique condition*. Par exemple, sous GNU/Linux, l'acquisition des droits administrateurs nécessite de : 1) connaître le mot de passe administrateur, 2) appartenir au groupe `wheel`.

Une seconde définition de ce principe, donnée dans [Clark et Wilson 1987a] (page 187), sous-entend que : "*Une règle élémentaire de séparation de privilèges peut être que toute personne ayant le droit de créer ou modifier des objets n'ait pas le droit de les exécuter*". De même dans [Sandhu 1990] : "*La séparation de privilèges implique que différentes opérations réalisées sur un même objet doivent être effectuées par des utilisateurs différents*". D'une manière générale, ce principe peut être défini par :

Définition 2.1.8 (Séparation de privilège) *Soit o un objet, P un ensemble de privilèges associés à o et X un ensemble d'entités. Alors, le principe de séparation de privilège implique que les privilèges P sur o soient distribués sur l'ensemble des utilisateurs X.*

Cette définition sous-entend que l'ensemble des privilèges d'un système doit être distribué sur l'ensemble des utilisateurs. Par exemple, une règle, généralement appliquée aux services système (ou démons système), implique qu'un service ne doit pas pouvoir créer/modifier un fichier puis l'exécuter. Ce principe, qui peut être étendu au cas des utilisateurs système, nécessite donc que les services systèmes ne possèdent pas le privilège de modification et d'exécution sur un même objet. Ce principe permet d'endiguer des attaques visant la génération d'un script par un service du système (ayant les droits administrateur) en vue de son exécution. Dans le cas des utilisateurs, ceci implique que les utilisateurs qui créent des objets soient différents de ceux qui les exécutent.

Non-interférence Le principe de la *non-interférence* [Focardi et Gorrieri 2001] implique que deux ou plusieurs processus puissent s'exécuter simultanément sans *interférer*, c'est-à-dire sans modifier la vision des données ou le comportement des autres processus. D'une manière générale, la non-interférence peut être définie par :

Définition 2.1.9 (Non-interférence) *Soit X un ensemble d'entités et soit I un ensemble de données. Un ensemble d'entités X n'interfère pas avec un ensemble de données I si les valeurs de I sont indépendantes des actions effectuées par X.*

Dans le cas d'un système, si nous considérons deux groupes de processus : les processus de haut niveau (processus système) et de bas niveau (utilisateur). La non-interférence peut correspondre à : l'exécution des processus utilisateurs n'a pas d'incidence sur l'exécution des processus système. Cela permet d'endiguer certaines attaques des utilisateurs sur les services système.

2.1.2 Politique de sécurité

Une politique de sécurité spécifie ce que l'on entend par "sécurisé", pour un système ou un ensemble de systèmes, et peut être définie de manière informelle (langage naturel) ou formelle (par des relations mathématiques). Une politique de sécurité considère les différentes propriétés de sécurité, définies dans la section précédente, relatives au système à protéger. En ce qui concerne la confidentialité, une politique identifie les différents états du système permettant l'accès à des données non-autorisées. Il ne faut alors pas seulement considérer le vol d'information direct, mais aussi une suite de transferts d'informations qui, par transitivité, peut conduire à un *flux d'informations*. Dans le cas de l'intégrité, une politique de sécurité identifie quels sont les moyens autorisés pour modifier de l'information, mais aussi quelles entités ont l'autorisation de modifier ces informations. Dans le cas de la disponibilité, une politique de sécurité définit quels services doivent être fournis. Des paramètres supplémentaires peuvent aussi être définis, comme un interval de temps pendant lequel un service doit être accessible, ou un temps minimal de réponse. Ce type de propriété a un lien étroit avec la qualité de service.

La déclaration d'une politique de sécurité doit permettre de définir formellement les propriétés de sécurité désirées sur un système. Si l'on doit prouver que le système est sûr, la définition de la politique doit pouvoir permettre de prouver que les différentes propriétés définies et leurs implantations sont correctes. Si une preuve formelle est impossible les propriétés de sécurité peuvent être garantis par des tests.

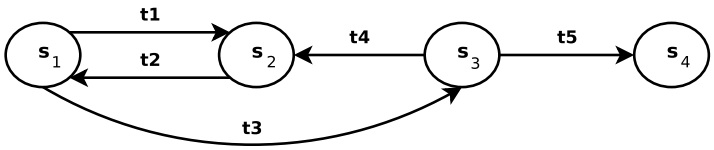


FIG. 2.1 – Exemple d'automate à états finis représentant les états d'un système.

En pratique, peu de politiques de sécurité définissent formellement les propriétés. La politique définit alors l'ensemble des actions légales entre entités. Le problème de vérification de ces règles reste en général entier, au regard de la confidentialité mais surtout vis à vis de l'intégrité.

2.1.2.1 Définition générale

Considérons un système d'information comme un automate à états finis avec un ensemble de fonctions de transition changeant l'état du système. Dans cette représentation, nous pouvons définir une *politique de sécurité* par :

Définition 2.1.10 (Politique de sécurité) *Une **politique de sécurité** est une déclaration qui partitionne les états d'un système en un ensemble d'états **autorisés** (ou **sûrs**) et un ensemble d'états **non-authorisés** (ou **non-sûrs**).*

Une politique de sécurité fixe donc le contexte dans lequel on peut définir un système "sûr".

Définition 2.1.11 (Système sûr) *Un **système sûr** est un système qui, partant d'un état sûr, ne pourra jamais entrer dans un état non-sûr.*

Considérons l'automate présent dans la figure 2.1 contenant quatre états et cinq transitions représentant un système. Prenons l'exemple d'une politique de sécurité qui partitionne ce système en un ensemble d'états autorisés $A = \{s_1, s_2\}$ et un ensemble d'états non-autorisés $NA = \{s_3, s_4\}$. Ce système est considéré comme non-sûr car partant d'un état sûr, il existe une transition amenant le système dans un état non-sûr. Néanmoins, si l'arc entre s_1 et s_3 est supprimé, le système sera considéré comme sûr. En effet, dans ce cas, il n'est plus possible d'atteindre un état non-sûr à partir d'un état sûr. Le passage d'un état sûr à un état non-sûr est alors appelé une brèche de sécurité :

Définition 2.1.12 (Brèche de sécurité) *Une **brèche de sécurité** apparaît lorsqu'un système entre dans un état non-autorisé.*

2.1.2.2 Garantie d'une politique de sécurité

La garantie et la définition d'une politique de sécurité sont deux aspects différents. La garantie dépend du mécanisme de sécurité.

Définition 2.1.13 (Mécanisme de sécurité) *Un **mécanisme de sécurité** est une entité ou une procédure dont l'objectif est de respecter chaque propriété de la politique ou d'en détecter les violations.*

Un tel mécanisme peut appliquer de manière active l'ensemble des propriétés de politique de sécurité ou passivement en détecter les violations. Ainsi, un mécanisme de contrôle d'accès garantit une propriété de sécurité alors qu'un système de détection d'intrusions en détectera les violations.

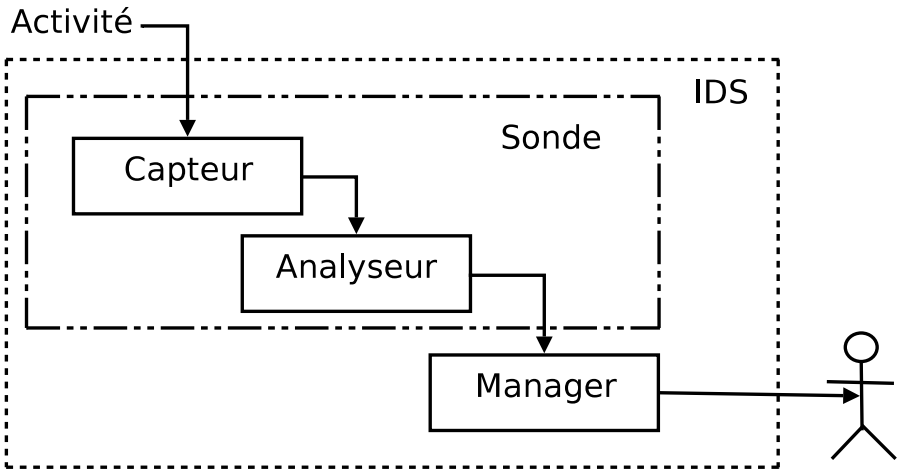


FIG. 2.2 – Architecture d'un système de détection d'intrusions selon l'IDWG

Dans les sections suivantes, nous étudions deux types de mécanismes de sécurité : les systèmes de *détection d'intrusions* et les mécanismes de *contrôle d'accès*.

2.2 Détection d'Intrusions

Les systèmes de détection d'intrusions (IDS pour *Intrusion Detection System*) ont pour objectif de révéler, généralement via des alertes, toute activité pouvant être considérée comme intrusive, depuis ou vers un système d'information, par analyse de données. Les sources de ces données correspondent à des événements générés par différents services ou utilisateurs. Les premiers travaux en détection d'intrusions ont débuté avec Anderson [Anderson 1980] en 1980 et Denning [Denning 1987] en 1987. Aujourd'hui, il existe plus de 140 systèmes de détection d'intrusions [Meier 2004].

Les systèmes de détection d'intrusions sont généralement classés en deux grandes catégories suivant le type de données à analyser [Bace et Mell 2001] : les systèmes de détection d'intrusions système (HIDS pour Host-based IDS) et les systèmes de détection d'intrusions réseau (NIDS pour Network-based IDS). Les HIDS sont caractérisés par l'analyse des événements/traces générés par le système. Les NIDS analysent les données qui transitent sur le réseau. Un IDS réalise une analyse passive. L'analyse passive est à mettre en opposition avec l'analyse active, c'est par exemple le cas pour un pare-feu qui bloque certains paquets.

La figure 2.2 illustre, de manière simplifiée, les différents modules composant un système de détection d'intrusions selon la normalisation proposée par l'*Intrusion Detection Working Group* [Wood et Erlinger 2007]. Cette architecture est composée de 3 modules communs à la majorité des IDS. L'*Activité* du système d'information fournit une source de données à des *Capteurs*. Ces *Capteurs* ont alors pour rôle d'extraire et de transformer certaines informations afin de les transmettre sous forme d'événements à un *Analyseur*. Le module d'analyse utilise alors ces événements afin de déceler une possible intrusion et génère en conséquence des alertes. Ces alertes sont finalement envoyées à un gestionnaire d'alertes (*Manager*). Ce dernier est chargé de traiter les alertes émanant des différents analyseurs et de notifier toute activité suspecte sur le système d'information à l'opérateur de sécurité. Notons enfin qu'un système de détection d'intrusions peut être constitué de plusieurs *Capteurs* traitant des sources de données différentes, de plusieurs *Analyseurs* appliquant différentes méthodes

d'analyse et de plusieurs *Managers* centralisant les alertes de sites distincts.

La performance d'un système de détection d'intrusions, et notamment de sa méthode d'analyse, est liée à deux notions importantes qui permettent d'"évaluer" ces performances [Bishop 2003] (page 11) :

Définition 2.2.1 (Faux négatif) *Idéalement, toute intrusion doit donner lieu à une alerte. Une intrusion non détectée, c'est-à-dire n'ayant pas généré d'alerte, constitue alors un faux négatif. La fiabilité (ou couverture) d'un Analyseur est liée à son taux de faux négatifs qui représente alors le pourcentage d'intrusions non détectées, ce taux devant être le plus bas possible.*

Définition 2.2.2 (Faux positif) *Toute alerte doit correspondre à une intrusion effective. Lorsqu'un système de détection d'intrusions génère une alerte qui n'a pas lieu d'être, cette alerte est qualifiée de faux positif. La pertinence (ou crédibilité) d'un Analyseur est liée à son taux de faux positifs qui représente alors le pourcentage de fausses alertes.*

La complétude d'un système de détection d'intrusions peut être définie comme son taux de détection, cette notion est ainsi liée au taux de faux négatifs. La relation entre la complétude et la pertinence d'un tel système peut ainsi être visualisée à l'aide d'une courbe ROC [Mell *et al.* 2003] (*Receiver Operating Characteristic*) représentant le taux de détection en fonction du taux de faux positifs. Une courbe ROC permet de trouver la configuration optimale d'un IDS minimisant le nombre de faux positifs et maximisant le taux de détections pour un contexte donné (réseau ou hôte spécifique). Mais l'estimation des taux reste difficile en pratique.

Nous venons de voir que le type de données permet de classer les IDS en deux grandes catégories (système et réseau). La méthode d'analyse permet d'obtenir une autre classification en distinguant les approches basées sur la détection d'anomalie (*Anomaly detection*), de celles utilisant des scénarios d'attaques (Misuse detection) et enfin de celles exploitant la définition d'une politique de sécurité (Policy-based detection). Dans la suite de cette section, nous utiliserons cette seconde classification pour étudier ces trois catégories de systèmes de détection d'intrusions.

2.2.1 Détection d'anomalies

Cette approche proposée par Anderson [Anderson 1980], puis étendue par Denning [Denning 1987], part d'un constat simple selon lequel l'exploitation d'une vulnérabilité sur un système, ou une tentative d'intrusion, implique la modification du *comportement* d'un service, d'une application ou même d'un utilisateur. Par exemple, le détournement d'un service peut donner lieu à : une utilisation anormale de ressources, l'accès à des fichiers ou une utilisation anormale du réseau.

L'une des premières approches de détection d'anomalies fût proposée par Anderson [Anderson 1980]. Partant du principe que les politiques de sécurité peuvent être violées, notamment grâce à l'abus des mécanismes système, l'auteur propose une nouvelle approche basée sur la comparaison des comportements des utilisateurs à un comportement de référence appelé *profil*. L'approche par détection d'anomalies, ou détection comportementale (*anomaly detection*), part donc de l'hypothèse que l'on peut définir un comportement dit "normal" de l'entité à surveiller (utilisateur, service, application, etc.). Toute déviation vis-à-vis de ce profil peut alors être considérée comme intrusive. Cette approche comporte deux phases : la première étant la construction d'un profil pertinent, la seconde correspond à l'évaluation de la déviation du comportement observé par rapport à ce profil.

La définition du profil détermine le comportement attendu du système. Ainsi toute déviation par rapport à ce profil donnera lieu à une alerte sans aucune connaissance de la cause de cette déviation.

Cette approche permet donc de détecter une attaque sans aucune connaissance du fonctionnement de celle-ci et donc de pouvoir potentiellement détecter des attaques inconnues. De plus, en supposant que ce profil n'évolue que rarement, cette approche ne requiert qu'une maintenance minimale. Cependant, la définition du profil et l'évaluation des déviations sont deux points critiques qui influent sur la "qualité" de la détection.

2.2.1.1 Construction du profil

Par définition, le profil est constitué d'un ensemble de mesures empiriques correspondant à un comportement "normal" pouvant caractériser un utilisateur, un service, une application ou un système.

La construction de ce profil résulte d'une phase d'apprentissage, au cours de laquelle le système est "observé" afin de recueillir des informations sur une utilisation "type" ou "normale" de ce système. Ces informations sont ensuite utilisées afin de générer un profil d'utilisation caractéristique. Ce profil correspond à un ensemble de paramètres pouvant être de natures différentes :

Statistique Le profil est constitué de données statistiques caractérisant l'évolution dans le temps d'une application, d'un utilisateur ou du système. Ces caractéristiques sont généralement obtenues par observation à intervalle régulier d'un système. Ces données peuvent prendre en compte le temps processeur utilisé, le taux de mémoire utilisé, la date de connexions/déconnexions des utilisateurs, etc. Cette approche permet ainsi de déterminer au vu de n observations $x_0, ..., x_{n-1}$ d'un paramètre x, si la valeur x_n de la n$^{\text{ème}}$ observation est normale ou non.

Les auteurs de [Lunt 1990] proposent de concevoir un IDS nommé IDES utilisant un profil basé sur une modélisation statistique des utilisateurs. Ces travaux ont aboutit à NIDES [Javitz et Valdes 1993, Anderson *et al.* 1994], un IDS qui utilise un modèle statistique (basé sur la covariance) afin de construire un profil composé de paramètres caractéristiques des utilisateurs. NIDES définit ainsi un modèle statistique de mesure d'évènements prenant en compte l'intensité, l'énumération ou la fréquence d'apparition de ces événements. NIDES s'intéresse principalement à la définition d'un profil par utilisateur. Ces paramètres concernent principalement l'accès aux fichiers et répertoires, le taux d'utilisation des ressources systèmes, les connections/déconnections des utilisateurs, les commandes utilisées et les activités réseaux.

L'environnement Emerald [Porras et Neumann 1997, Neumann et Porras 1999, Lindqvist et Porras 1999] est une suite d'outils distribués et modulaires permettant de traquer les activités malicieuses dans un réseau à large échelle. Emerald [Porras et Neumann 1997] contient ainsi un module de détection d'anomalies basé sur les travaux initiés pour NIDES. Alors que NIDES est utilisé pour définir un profil utilisateur, le module intégré dans Emerald a été étendu afin de pouvoir définir un profil réseau. Emerald est ainsi constitué d'un ensemble de moteurs de génération de profil (*engine profiler*) permettant une séparation totale entre la représentation du profil et les algorithmes mathématiques utilisés pour évaluer les nouvelles observations.

Règle logique d'infèrence Un ensemble de règles logiques d'infèrence définit le profil normal d'un utilisateur au vu de ses précédentes activités. Ces règles peuvent être définies manuellement ou générées automatiquement à partir des observations effectuées. Les règles ainsi générées décrivent des comportements et sont rafraîchies régulièrement. Notons qu'une règle peut décrire la distribution probabiliste du prochain événement attendu.

L'outil Wisdom & Sense [Vaccaro et Liepins 1989] utilise un ensemble de règles où chaque règle décrit la distribution probabiliste du prochain événement attendu. Ce qui différencie cet IDS d'une approche de détection par scénario vient du fait que l'ensemble des règles est généré

automatiquement par analyse de l'historique des événements système. Par exemple, si dans les traces système un utilisateur tente d'ouvrir une session et que l'événement suivant représente à 95% l'ouverture de la session, à 3% une seconde tentative d'ouverture et à 2% une troisième tentative, trois règles sont déduites des traces : 1) **si** `ouverture_session` = 1 **alors** `session_ouverte` = 95%, 2) **si** `ouverture_session` = 2 **alors** `session_ouverte` = 3%, et 3) **si** `ouverture_session` = 3 **alors** `session_ouverte` = 2%. Tout comportement déviant de ces règles sera alors considéré comme intrusif.

Apprentissage automatique L'apprentissage automatique est utilisé dans l'approche de détection d'anomalie afin d'apprendre un profil correspondant au comportement normal d'un utilisateur. La phase d'apprentissage utilise alors un ensemble d'observations d'un utilisateur ou du système afin de générer ce profil. Le profil peut être, par exemple, représenté par un réseau de neurones ayant comme entrée un ensemble de paramètres à observer (temps processeur utilisé, mémoire disponible, taille et type de paquet, etc.). En utilisant comme échantillon d'entraînement la période d'observation, cette approche fournit après apprentissage, un réseau de neurones "théoriquement" capable de détecter des intrusions. Les méthodes d'apprentissage généralement utilisées pour la détection d'anomalie sont les réseaux de neurones, les machines à vecteurs de support (SVM pour *Support Vector Machine*) ou les arbres de décision. L'objectif étant d'apprendre, de manière supervisée ou non, c'est-à-dire après étiquetage de l'échantillon de test par un expert ou sur les données brutes, le comportement normal de l'utilisateur. Ces méthodes étant généralement considérées comme des "boites noires", il est difficile de prouver si le résultat de l'apprentissage est correct ou pas.

Les auteurs [Fox *et al.* 1990, Debar *et al.* 1992, Debar et Dorizzi 1992] proposent de générer un profil représentant le comportement des utilisateurs via une approche de classification automatique (apprentissage non supervisé). L'outil Hyperview [Debar *et al.* 1992, Debar et Dorizzi 1992] comporte un module ayant pour objectif de prédire le comportement des utilisateurs, c'est-à-dire les prochaines actions qu'ils devraient accomplir. Hyperview utilise un réseau de neurones afin de prédire la prochaine commande d'un utilisateur en fonction des précédentes. Le réseau de neurones reçoit ainsi en entrée les $c_0, \ldots, c_{n-1}$ dernières commandes et prédit en sortie la prochaine c_n. De plus, la sortie du réseau est réutilisée en entrée pour les prédictions suivantes. De ce fait, le réseau adapte le profil aux nouveaux comportements et oublie les anciens.

Les auteurs de [Ryan *et al.* 1998] proposent une implantation d'un système de détection d'anomalies nommé NNID (pour *Neural Network Intrusion Detector*). Cet outil utilise un réseau de neurones afin d'identifier un utilisateur légitime en fonction de la distribution des commandes utilisées et non l'ordre de ces commandes.

Finalement, Forrest [Forrest *et al.* 1997] s'inspire de l'immunologie biologique et propose de mettre en oeuvre des algorithmes du type *pattern matching* afin d'obtenir un profil constitué de séquences d'appels système caractéristiques. Le profil, représentant un système sain, est alors constitué d'un ensemble de séquences d'appels système valides obtenu par observation du système durant une phase de test. Lors de la détection, Forrest propose d'utiliser des algorithmes de *pattern matching* afin de reconnaître les séquences d'appels système non présentes dans le profil et qui représentent alors des anomalies.

Data Mining Appliqué à la détection d'anomalies, le Data Mining a pour vocation d'extraire des observations les caractéristiques des données représentant alors le profil de l'utilisateur, d'une application ou d'un système.

Les auteurs de [Lee *et al.* 1998] génèrent ainsi un profil, composé de *patterns* représentant des

séquences d'appels système, à partir de traces d'exécution. Un pattern correspond ainsi à une expression régulière utilisée pour reconnaître les séquences valides. Une séquence ne correspondant à aucun pattern est alors considérée comme malicieuse. Le Data Mining permet ainsi d'extraire le plus petit ensemble de patterns caractéristiques à partir des traces d'appels système générées lors de l'exécution d'une application. Ainsi, plutôt que d'énumérer toutes les séquences possibles, l'idée est de découvrir le plus petit ensemble de patterns caractéristiques de cet ensemble. Une des conséquences de l'utilisation de patterns est que si une séquence malicieuse correspond à un pattern du profil, cette séquence ne sera pas détectée. De même, l'utilisation d'une petite base d'apprentissage peut avoir comme conséquence l'obtention de patterns trop génériques ne permettant pas de différencier les séquences normales et des malicieuses.

2.2.1.2 Détection des déviations

Après la construction du profil, il s'agit de détecter les déviations. Les méthodes d'estimation des déviations sont de deux natures :

Statistique - Associé à la construction du profil statistique Une différence importante entre la valeur statistique attendue et celle observée peut représenter une première estimation simple. De même, l'apparition trop fréquente d'un événement à probabilité très basse, ou au contraire, la non-apparition d'un événement à probabilité élevée peuvent représenter un autre type de mesure. L'utilisation de la moyenne et de l'écart type σ, proposée dans l'approche de Denning [Denning 1987], permet de considérer l'observation x_n comme anormale si x_n sort de l'intervalle de confiance défini par $(\pm\Delta \times \sigma)$ autour de la moyenne (où Δ est une valeur définie manuellement).

IDES assigne un score statistique à chaque type d'événements de chaque session utilisateur par analyse des traces système. Lorsque ce score dépasse un seuil prédéfini, une alerte est générée. Lors de la détection, IDES compare la nouvelle distribution de fréquences d'apparition ou d'intensité de ces événements, obtenue lors d'une observation à court-terme, à un historique de distribution des fréquences obtenue à long-terme. Lors des observations, un historique est mis-à-jour, en affectant par exemple un poids plus élevé aux observations récentes, afin d'obtenir une distribution de fréquences dynamiques. Un avantage de cette approche vient donc de cette dynamicité qui ne nécessite aucune connaissance (ou estimation) des fréquences anormales. En contrepartie, un utilisateur malveillant peut abuser de cette dynamicité pour faire apprendre à IDES un ensemble de séquences correspondant à une attaque.

Évaluation de l'apprentissage - Associé à la construction par apprentissage automatique Lors de l'utilisation d'algorithmes d'apprentissage automatique, les nouvelles observations du système sont utilisées comme source de la phase de reconnaissance. La méthode utilisée fournit alors une valuation de l'activité du système. Dans le cas de l'utilisation d'un réseau de neurones, le réseau de neurones construit servira de méthode de détection, il en va de même pour les arbres de décision. Une alerte est alors générée lorsque le réseau de neurones ou l'arbre de décision considère les observations comme suspectes. Dans le cas d'un réseau de neurones entraîné, une indication (valeur entre 0 et 1) peut correspondre à la probabilité d'une activité intrusive, il faut alors définir un seuil au delà duquel on considère qu'il y a réellement intrusion.

Certaines approches [Fox *et al.* 1990, Debar *et al.* 1992, Debar et Dorizzi 1992] ne s'intéressent pas à l'estimation de l'activité intrusive, mais prédisent le comportement normal qui doit être observé ensuite. Dans ce cas, une alerte est générée lorsque le comportement prévu ne

correspond pas à la réalité.

Finalement, les approches basées sur la génération d'un profil composé de *pattern* [Forrest *et al.* 1997, Lee *et al.* 1998] utilisent des méthodes de *pattern-matching* pour reconnaître ces patterns dans les observations du système. Le filtrage par motif (en anglais, *pattern-matching*) est la vérification de la présence de constituants d'un motif (en anglais, *pattern*). De tels motifs concernent conventionnellement soit des séquences, soit des arbres. Les séquences (particulièrement les chaînes de caractères) sont souvent décrites par des expressions rationnelles. Un ensemble d'événements ne correspondant pas à un pattern du profil est alors considéré comme un comportement intrusif et donne lieu à la génération d'une alerte.

2.2.1.3 Discussion

Ces méthodes étant basées sur des observations rarement exhaustives et limitées dans le temps, elles produisent généralement un taux de *faux positifs*[2] élevé. En outre, une des hypothèses forte de cette approche est que, durant la phase d'observation, aucune attaque ou intrusion n'a eu lieu. Ainsi, la phase d'observation doit être effectuée sur des systèmes sûrs et avec des utilisateurs de confiance. Dans le cas contraire, on se heurte à un problème de *faux négatifs*[3], certaines attaques ou intrusions n'étant pas détectées car faisant partie du profil. Un utilisateur pourrait ainsi introduire délibérément un biais dans son profil afin de pouvoir l'exploiter par la suite. Un sur-apprentissage entraîne un taux de faux positifs élevés. Inversement, un profil sous-entrainé aura pour conséquence un taux de faux-négatifs élevé. De plus, le choix des paramètres du profil représentatif des entités surveillées est un problème majeur.

La détection des déviations pose aussi un problème. Ainsi, un seuil de déclenchement trop bas (haut) peut donner lieu à un taux de faux positifs (négatifs) élevé. Les mises-à-jour des profils posent aussi problème. Dans le cas pratique, ces systèmes sont peu sûr.

2.2.2 Détection par scénarios

La détection par scénarios (ou *misuse detection*) permet de détecter une attaque connue via la définition d'un *scénario*. Cette approche utilise une base de connaissances, appelée base de *signatures d'attaques* et une méthode de recherche de motifs permettant de reconnaître les signatures définies. Un détecteur d'intrusions par scénario est alors composé de :

- un ensemble de sondes produisant un flux d'évènements
- une base de signatures d'attaques
- un algorithme de recherche de motif, comparant le flux d'événements aux signatures contenues dans la base

2.2.2.1 Base de signatures d'attaques

Chaque signature peut être vue comme une suite d'événements caractéristiques d'une attaque permettant de la différencier du comportement normal du système. La construction de la base de signatures exige des connaissances précises des attaques et de leurs paramètres. Ces informations peuvent, par exemple, être obtenues par veille technologique.

Les scénarios ou la base de signatures d'attaques peuvent être de nature différente :

[2] un faux positif est une alerte ne correspondant pas réellement à une tentative d'intrusion (alertes non fondées).

[3] Un faux négatif correspondant à une intrusion ne générant pas d'alerte (intrusions non détectées)

Motif Un motif peut exprimer une séquence d'appels système, une suite de commandes utilisateurs ou même un ensemble de paquets réseaux spécifique à une attaque. Par exemple, un paquet réseau contenant les drapeaux `SYN` et `FIN` sont typiques d'un scan réseau. De même, un paquet contenant les mots-clés `password :` ou `/etc/shadow` peut indiquer qu'un utilisateur distant a accédé à un fichier contenant des mots de passe. La base de signatures est ainsi composée d'un ensemble de motifs d'activité illicite sur le système. Ces motifs peuvent être définis à l'aide d'expressions régulières ou utiliser un langage proche des événements remonté par les capteurs.

Par exemple, un IDS réseau (NIDS) utilisera un langage permettant de facilement décrire le contenu des paquets réseau, tel que l'adresse IP source ou destination ou les drapeaux utilisés, à l'aide de mots-clés prédéfinis ou d'expressions régulières. Cette approche est donc la plus couramment utilisée dans les NIDS tels que Snort [Roesch 1999], Realsecure[4] ou NetRanger[5]. Notons que dans tous ces cas, les signatures sont définies manuellement par des experts.

Règle logique d'infèrence La base de signatures peut être représentée à l'aide d'un ensemble de règles logiques d'infèrence servant d'entrée à un système expert. Ces règles sont définies manuellement. Ces règles traduisent l'expertise d'un opérateur de sécurité et sont écrites dans un langage qui peut être spécifique à la détection d'intrusions.

OSIRIS [Baur et Weiss 1988] utilise ainsi des règles écrites en Prolog, IDES [Lunt *et al.* 1992] utilise le langage du système expert PBEST, alors que ASAX [Habra *et al.* 1992] utilise son propre langage RUSSEL. Ce langage, qui peut être vu comme un langage procédural incluant une structure de contrôle prédéfinie, permet d'exprimer des comportements anormaux. Il est, par exemple, possible d'exprimer une règle détectant X authentifications incorrectes dans une période de temps inférieure à Y. Cette signature correspond à une tentative d'attaque par force-brute qui a pour objectif de déterminer le mot-de-passe d'un utilisateur en essayant un ensemble de mot de passe provenant d'un *dictionnaire*. Ainsi une telle règle peut être : "**si** `tentative_ouverture_session` ≥ 10 **et** `temps` ≤ 60 **alors** `intrusion` $=$ `vrai`".

Automate Un scénario d'attaque peut aussi être représenté à l'aide d'automates et de machines à états finis. Un automate représente l'enchaînement d'actions nécessaires à la réalisation de l'attaque [Eckmann *et al.* 2000]. Cette approche permet d'exprimer des signatures complexes comportant plusieurs possibilités d'étapes dans l'automate pour arriver à un même état. L'automate peut aussi être exprimé à l'aide de langage spécifique tel que dans [Pouzol et Ducass 2002].

Plusieurs approches [Carrasco et Oncina 1994, Kosoresow et Hofmeyr 1997, Ron *et al.* 1996] utilisent un automate à états finis. L'automate peut aussi être représenté sous forme d'une variante des réseaux de Pétri colorés comme dans IDIOT [Kumar et Spafford 1994] ou sous forme de diagrammes de transitions d'états tel qu'implanté dans l'outil NetSTAT [Vigna et Kemmerer 1998]. Les états de cet automate représentent l'historique récent des symboles (des appels système) qui ont été observés, une transition d'un état vers un autre caractérise l'ensemble de traces qui doivent être produit après cet état.

Notons que de nombreux travaux portent sur le développement de langage de description d'attaques [Lunt *et al.* 1992] [Habra *et al.* 1992] [Eckmann *et al.* 2002]. Ces langages sont à la fois expressifs, pour permettre une description précise et concise d'un scénario, et adaptés à l'environnement considéré (trafic réseau, événement système, etc.). Ils permettent la prise en compte des spécificités liées à l'environnement (différentes couches et options d'un paquet réseau, arguments possibles d'appels système).

[4] http ://www.realsecure.net/

[5] http ://www.cisco.com/univercd/cc/td/doc/product/iaabu/netrangr/index.htm

2.2.2.2 Détection des scénarios

La détection par scénarios lève une alerte lorsqu'un scénario est reconnu. Pour chacun des trois types de signatures précédants, une méthode spécifique est utilisée :

Pattern Matching La reconnaissance de signature utilise des algorithmes de type *pattern matching* pour reconnaître une signature dans une trace correspondant à une suite d'événements. Cette approche classique pose un problème lorsque plusieurs scénarios donnent lieu à une même signature. Pour pallier à ce problème, certaines approches utilisent des algorithmes de reconnaissance basés sur des algorithmes génétiques [Mé 1998], des réseaux bayésiens [DuMouchel et Schonlau 1998] ou encore des approches intégrant l'analyse de la configuration du système [Mounji et Charlier 1997]. D'autres approches utilisent un système de corrélation "multi-événements", intégrant des pré-conditions, post-conditons et des assertions [Michel et Mé 2001] [Cuppens et Ortalo 2000] afin de préciser la définition des scénarios.

Cette approche permet d'avoir des performances élevées en termes d'analyse, mais elle est généralement la source d'un taux élevé de faux positifs. En effet, une des limites de cette approche vient du fait qu'il est difficile d'écrire une signature couvrant plusieurs variantes d'une même attaque sans générer de faux positifs.

Système expert Le flux d'événements généré par les capteurs sert d'entrée au système expert. Son moteur d'inférence décide si une attaque répertoriée s'est produite. Lorsque le système expert reconnaît une règle, une alerte est émise. EMERALD [Lindqvist et Porras 1999] intègre une variante du système expert PBEST, les règles étant définies à l'aide du langage spécifique à PBEST.

Automate Lorsque l'état final est atteint, le scénario d'attaque s'est totalement déroulé et la signature est considérée comme validée. Une alerte correspondant à ce scénario est alors générée.

2.2.2.3 Discussion

La détection par scénario permet d'avoir un taux de détection d'attaques connues (contenues dans la base) élevé. De plus lorsque les signatures sont correctement écrites et précises, il en résulte un faible taux de faux positifs. Mais elle ne permet pas la détection d'attaques inconnues (dont on ne connaît pas le scénario) ce qui se traduit par un taux de faux négatifs élevé. Contrairement à l'approche de détection d'anomalies, la détection par signature demande une maintenance active de la base de signatures. Ces solutions ne garantissent pas de propriétés de sécurité. Celles-ci ne sont même pas exprimées. L'objectif est simplement de détecter des violations connues.

2.2.3 Détection paramétrée par une politique

L'approche de détection d'intrusions paramétrée par la politique (*Policy-based* IDS) utilise une politique de sécurité comme définition du comportement de référence d'un système. Contrairement aux approches classiques de détection d'intrusions qui détectent des violations sans qu'une politique ne soit explicitée, cette approche détecte les violations d'une politique définissant un ensemble de propriétés. Prenons le cas d'une modification proscrite par la politique de sécurité, d'un document par un utilisateur. Un HIDS de ce type vérifie l'intégrité du document.

L'approche de détection d'intrusions paramétrée par la politique peut aussi être vue comme une approche de détection d'anomalies utilisant comme profil la politique de sécurité et ne nécessitant donc aucune phase d'apprentissage. Si la politique est correctement définie, l'IDS en détectera les violations.

Dans la suite de cette section, nous étudierons les trois approches actuelles de détection paramétrée par une politique : *abus de privilèges*, *non-interférence* et *flux de références*.

2.2.3.1 Abus de privilèges

La détection d'intrusions par spécification de comportement (*Specification-based* IDS), initialement proposée par Ko [Ko *et al.* 1994], a pour objectif de détecter les abus de privilèges. L'approche de [Ko *et al.* 1994] définit un langage de spécification, fondé sur une logique du premier ordre. Ce langage permet de spécifier le comportement normal d'un programme en définissant l'ensemble des appels système autorisés et les arguments typiques de ces appels système. La détection utilise un moteur de *pattern matching* assurant la reconnaissance des appels non conformes. Cette approche a été appliquée au cas des systèmes distribués [Ko *et al.* 1997].

Les auteurs de [Ko *et al.* 1994] proposent de détecter les abus de privilèges en spécifiant manuellement le comportant normal des applications. Cette spécification sera ensuite comparée aux traces d'exécution de l'application. Toute action, violant cette spécification, aura pour conséquence la génération d'une alerte. Cette spécification n'est donc pas un comportement observé empiriquement, comme dans l'approche de Forrest [Forrest *et al.* 1997], mais une spécification du comportement "normal".

Application à la détection d'intrusions : Le langage de spécification peut être composé d'expressions régulières [Sekar *et al.* 1999] représentant les appels système et leurs arguments autorisés, ou d'une grammaire [Ko *et al.* 1994] [Ko *et al.* 1997] permettant d'exprimer des relations et des conditions sur les arguments des différents appels système. Des implantations avancées [Uppuluri et Sekar 2001] proposent des macro-définitons d'opérations générales composées d'un ensemble d'appels système (écrire, lire, interférer avec un autre processus).

Lors de la détection, les traces d'appels système sont, par exemple, obtenues via un programme de supervision de type *strace*[6] ou via un module du noyau détournant les appels système. Ces traces sont ensuite soumises à un module de reconnaissance. En fonction du langage utilisé, il peut s'agir d'un simple algorithme de *pattern matching*, un automate à états finis ou une machine de Turing. Lorsqu'un comportement non conforme à la spécification de l'application est détecté, c'est-à-dire lorsqu'une trace n'est pas reconnue, une alerte est levée. Une approche [Sekar *et al.* 1998] plus avancée permet d'isoler l'application fautive via, par exemple, une réduction des privilèges de cette application.

Discussion De même que la détection d'anomalie, la détection d'abus de privilèges a l'avantage de pouvoir détecter des attaques inconnues. Les *flux d'informations illégaux* ne sont pas traités. Cette méthode ne traite que le cas particulier des abus de privilèges pris en compte dans la spécification.

2.2.3.2 Non-interférence

L'approche de détection par contrôle de non-interférence, proposée par Ko et Redmond dans [Ko et Redmond 2002], définit une politique comme un ensemble de règles. Elle permet de détecter des attaques de type *race-condition*, où une application interfère sur les données d'une autre

[6] *strace* est une application GNU/Linux qui permet de monitorer/surveiller les appels système ou les signaux d'une autre application.

application, [7] en utilisant le concept de *non-interférence* entre processus. En général, ce type d'attaque a lieu lorsqu'un processus *non-privilégié* NP oblige un processus *privilégié* P à effectuer des opérations illégales, c'est-à-dire non conformes à son comportement normal. Pour se faire, NP utilise une suite d'opérations spécifiques effectuée dans un labs de temps court. [Ko et Redmond 2002] utilise alors la non-interférence (cf. def 2.1.9) pour détecter ce type d'attaque. Cette approche constitue l'un des premiers travaux en détection d'intrusions bâti sur l'idée de garantir une propriété de sécurité spécifique.

Modèle Les auteurs s'intéressent ici à des interférences entre opérations *privilégiés*, c'est-à-dire s'exécutant avec des droits autorisés à modifier une donnée D, et *non-privilégiés*, c'est-à-dire s'exécutant avec des droits ne permettant pas la modification de cette donnée D. Par extension, les auteurs parlent de processus *privilégiés* et *non-privilégiés*. D'autre part, un état du système est dit "sûr" si cet état est cohérent vis-à-vis de la définition de la politique. Étant donné que cette approche est basée sur une politique contrôlant l'*intégrité des données*, cet état sera "sûr" si aucune donnée privilégiée n'a été modifiée par un processus non-privilégié.

Lors de l'exécution d'un processus privilégié, les opérations de processus non-privilégiés seront considérées comme non-interférentes si chaque opération du processus privilégié *commute* avec l'ensemble des opérations effectuées par les processus non-privilégiés. Une opération A commute avec une opération B si elle n'a pas d'incidence sur les données utilisées par B, c'est-à-dire si les séquences d'opérations A, B et B, A ont le même résultat. Par exemple, une opération `read` (ouvrir un fichier) commute avec une opération de lecture (`read`). Par contre, une opération d'écriture (`write`) ne commute pas avec une opération de lecture (`read`).

Cette approche introduit donc la définition d'une *table de commutativité* (X, Y) des appels système permettant de vérifier si une opération privilégiée X et une opération non-privilégiée Y commutent. La table 2.1 contient un exemple de table de commutativité pour les appels système sous GNU/Linux. Dans cette table, un $\surd$ signifie que deux opérations commutent sans condition, par exemple l'opération d'ouverture en lecture (`o/read`) commute avec toute autre opération. Un c signifie que deux opérations ne commutent pas, il faut alors vérifier certaines conditions portant généralement sur la résolution des noms des données manipulées par ces deux appels système (vérifier qu'ils ne concernent pas le même fichier). Par exemple, une ouverture en écriture (`o/write`) ne commute avec une ouverture en lecture (`o/read`) que s'il ne s'agit pas du même fichier.

Application à la détection d'intrusions Le système de détection d'intrusions décrit dans [Ko et Redmond 2002] utilise donc cette table de commutativité afin de vérifier que les événements d'un processus privilégié commutent avec ceux des processus non-privilégiés. En pratique, le détecteur consiste en une sonde qui intercepte les appels système exécutés par les processus applicatifs, et qui alimente l'algorithme de détection proprement dit. Cette algorithme vérifie alors que chaque opération réalisée par le processus privilégié commute avec l'ensemble des opérations exécutées par les autres processus non-privilégiés. Si deux opérations ne commutent pas, une alerte est générée.

[7] Une *situation de compétition* ou *race-condition* est un problème de sécurité dans lequel une application protège ses données de façon insuffisante. Une autre application peut alors en profiter pour les lui voler ou les détourner, si elle y accède au bon moment, c'est-à-dire avant que l'autre application n'ait eu le temps de les protéger. Ce problème est généralement lié au problème de concurrence et d'ordonnancement des opérations non prises en compte lors de l'implantation de l'application. Pour éliminer les *race conditions*, il faut s'assurer que les opérations que l'on veut effectuer successivement sont atomiques.

	o/write	o/read	close	chmod	chown	slink	link	rename	unlink
o/write	c	c	√	c	c	c	c	c	c
o/read	√	√	√	√	√	√	√	√	√
close	√	√	√	√	√	√	√	√	√
chmod	c	c	√	c	c	c	c	c	c
chown	c	c	√	c	c	c	c	c	c
slink	c	c	√	c	c	c	c	c	c
link	c	c	√	c	c	c	c	c	c
rename	c	c	√	c	c	c	c	c	c
unlink	c	c	√	c	c	c	c	c	c

TAB. 2.1 – Table de commutativité pour les appels système sous GNU/Linux

Discussion Le principal avantage de cette méthode est de garantir un type d'intégrité des processus. Cependant, cette solution ne garantit pas la propriété d'intégrité au sens de [ITSEC 1991] et ne traite pas la confidentialité.

2.2.3.3 Contrôle de flux de référence

La détection par contrôle de flux de référence [Zimmermann *et al.* 2003b, Zimmermann *et al.* 2003a] détecte certains flux d'informations illégaux. Un flux d'informations est défini comme une séquence d'appels système permettant de transférer de l'information d'un objet A vers un objet B. Le contrôle de flux de référence détecte uniquement les *attaques par délégation*, étendant ainsi l'approche de non-interférence à la confidentialité. Une attaque par délégation est alors définie comme une séquence d'opérations aboutissant à la création d'un flux illégal. Un flux d'informations entre A et B est considéré comme illégal si il ne peut être directement créé par A, c'est à dire si A ne peut pas accéder directement à B.

Modèle Le contrôle de flux de référence utilise une représentation objet du système, où chaque objet est vu comme un conteneur d'informations sur lequel on peut appliquer des méthodes (lire, écrire). Un accès à de l'information est ainsi modélisé par un *appel de méthode* sur l'objet contenant cette information. Un flux d'informations est alors représenté par une série d'appels de méthode.

Un *domaine* correspond à un ensemble de flux légaux. Ces domaines sont obtenus via une politique de contrôle d'accès. La table 2.2 contient un exemple de politique de contrôle d'accès. La figure 2.3 représente, à gauche, l'interprétation de la politique de contrôle d'accès en termes de flux d'informations et, à droite, l'ensemble de domaines résultant de cette interprétation. On peut ainsi remarquer que l'on obtient deux domaines correspondant aux deux utilisateurs. Le domaine A correspond ainsi à l'ensemble $\{(o_1, lecture), (o_2, ecriture), (o_3, ecriture)\}$ et le domaine B à l'ensemble $\{(o_1, ecriture), (o_2, lecture), (o_4, lecture), (o_4, ecriture)\}$

Par définition, pour être légal, un flux d'informations ne doit faire appel qu'à des méthodes d'un même domaine. Toute violation de cette propriété génère une alerte. Par exemple le flux composé de la séquence $(o_1, lecture), (o_4, ecriture)$ est illégal car $(o_1, lecture)$ appartient au domaine A et $(o_4, ecriture)$ au domaine B. À chaque objet est associé un *ensemble de références*, déterminant dans quel(s) domaine(s) chaque méthode de l'objet peut être utilisée. Un flux entre deux domaines génère une alerte.

	Alice	**Bob**
o_1	lecture	écriture
o_2	écriture	lecture
o_3	écriture	$\emptyset$
o_4	$\emptyset$	lecture,écriture

TAB. 2.2 – Exemple de politique de contrôle d'accès.

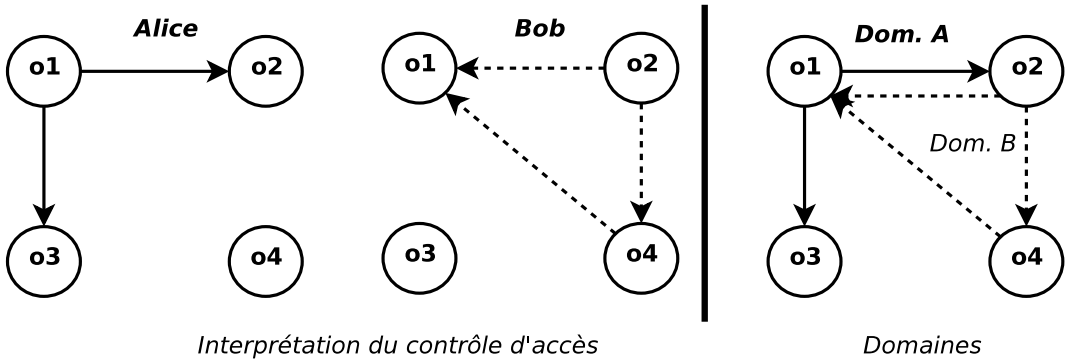


FIG. 2.3 – Flux et domaines correspondant à la politique

Discussion L'une des limitations importantes de cette solution est que tout flux entre domaines est illégal et tout flux à l'intérieur d'un domaine est légal. Il ne s'agit donc que d'une propriété particulière de confidentialité. De plus, il n'est pas possible d'obtenir la propriété de confidentialité générale telle que définie dans[ITSEC 1991], ni même de détecter une autre propriété que celle de l'absence de flux entre deux domaines.

2.3 Contrôle d'accès

Comme nous l'avons vu dans la section 2.1.1, la sécurité des systèmes d'information repose sur trois propriétés fondamentales : la *confidentialité*, l'*intégrité* et la *disponibilité*. Le *contrôle d'accès* a pour objectif de garantir deux de ces trois propriétés fondamentales (confidentialité et d'intégrité) via la définition et l'implantation d'une politique sécurité (cf. section 2.1) régissant les accès : *une politique de contrôle d'accès*. L'exécution d'un système d'exploitation peut être vue comme un ensemble de sujets et d'objets interagissants ensemble dans le but d'effectuer des actions qui modifieront (ou non) l'état du système. Ces entités peuvent être séparées en deux ensembles : celui des *sujets*, qui sont les entités actives (processus), et celui des *objets*, qui correspondent aux entités passives (ressources, fichiers, sockets). Une opération effectuée par un sujet sur un objet peut alors être représentée par un triplet *(sujet, objet, type d'accès)*. Une politique de contrôle d'accès est composée d'un ensemble d'opérations permises. Un modèle de contrôle d'accès classique peut être modélisé par :

Un ensemble de *sujets* Un *sujet* est une entité active du système (essentiellement un processus) ;

Un ensemble d'*objets* Un *objet* est une entité passive, un conteneur d'informations, sur lequel un sujet peut effectuer des actions (les fichiers, sockets de communication, périphériques matériels, etc.) ;

Un ensemble de *permissions* Une *permission* représente une action autorisée entre un *sujet* et un *objet* (par exemple, lecture, écriture, exécution, etc.), ou entre deux *sujets* (envoi de signal ou de message inter-processus) ;

Un ensemble de *commandes de modification* . Ces commandes permettent de modifier les ensembles précédents (création et destruction de sujets, d'objets et de privilèges).

Les *commandes de modification* permettent de faire évoluer ces politiques de sécurité pour, par exemple, prendre en compte l'intégration de nouveaux utilisateurs dans le système d'exploitation. Cette évolution doit être contrôlée pour prévenir les abus de la politique ou les configurations indésirables favorisant les intrusions. C'est pourquoi, chaque mécanisme de contrôle d'accès définit des règles de modification plus ou moins restrictives suivant le but recherché, et nécessitant éventuellement certains privilèges comme un niveau administrateur.

Les parties suivantes présentent les deux premières familles de modèles de contrôle d'accès. On en distingue principalement trois :

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

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

Contrôle d'accès basé sur les rôles Le modèle RBAC pour *Role-Based Access Control* a pour but de simplifier l'administration des droits d'accès des utilisateurs individuels en fournissant un niveau d'indirection supplémentaire. Plutôt que de donner directement des permissions aux utilisateurs, on définit différents rôles possibles pour l'utilisation du système d'exploitation, avec des droits d'accès associés. Ensuite, chaque utilisateur a accès à un ensemble de rôles suivant son activité, et donc à un sous-ensemble de permissions correspondant.

2.3.1 Contrôle d'accès discrétionnaire

Le contrôle d'accès *discrétionnaire* (DAC) est actuellement le modèle implanté dans la majorité des systèmes d'exploitation. Son nom vient du fait que la définition des droits d'accès à une ressource du système est réalisée à la *discrétion* de son propriétaire. Typiquement, les droits d'accès sur un fichier sont positionnés par l'utilisateur déclaré comme propriétaire de ce fichier. Par exemple, sous Unix, le propriétaire d'un fichier gère les droits de *lecture*, d'*écriture* et d'*exécution* de ses fichiers pour *lui-même*, les membres d'un *groupe* et tous les *autres* utilisateurs du système.

2.3.1.1 Modèle de Lampson

Une manière naturelle pour représenter un modèle de contrôle d'accès est sous forme de *matrice de contrôle d'accès*. Dans cette représentation, chaque ligne représente un sujet, chaque colonne représente un objet ou un sujet et chaque élément correspond à un ensemble de privilèges.

Le premier modèle de ce type fût proposé par Lampson en 1971 [Lampson 1971]. Lampson propose ainsi de placer, dans une matrice A, l'ensemble D des domaines de protection (qui représentent

des contextes d'exécution pour les programmes, i.e. les sujets) sur les lignes, et en colonnes l'ensemble X des objets (incluant les domaines). Il pose ensuite deux définitions :

Définition 2.3.1 (Capabilities Lists) *Étant donné un domaine $d \in D$, la liste des capacités (capabilities) pour le domaine d est l'ensemble des couples $(o, A[d, o]), \forall o \in X$.*

Définition 2.3.2 (Access Control Lists) *Étant donné un objet $o \in X$, la liste de contrôle d'accès (*ACL*) pour l'objet o est l'ensemble des couples $(d, A[d, o]), \forall d \in D$.*

La définition 2.3.1 lie un domaine d avec l'ensemble des permissions qu'il possède sur chaque objet o. Une liste des capacités représente l'ensemble des actions permises pour un sujet sur le système. La définition 2.3.2 lie un objet o avec l'ensemble des permissions accordé à chaque domaine d. Une liste de contrôle d'accès contient ainsi l'ensemble des permissions sur un objet défini pour chaque sujet du système.

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

Dans ce modèle la modification de la politique de sécurité, via l'application des règles de modification, nécessite la modification de la matrice de contrôle d'accès. Par exemple, la prise en compte d'un nouvel utilisateur passe par l'ajout d'une ligne de la matrice. Par conséquent, la mise à jour d'une telle politique est quelque peu fastidieuse. Le modèle de Lampson a été progressivement amélioré pour donner naissance à d'autres modèles tel que le modèle HRU [Harrison *et al.* 1976].

2.3.1.2 Modèle HRU

Établi par [Harrison *et al.* 1976], le modèle HRU est traditionnellement utilisé pour décrire les politiques de contrôle d'accès discrétionnaires. Dans ce modèle, une matrice P représente l'ensemble des droits d'accès des sujets sur des objets. Ainsi chaque sujet possède des droits d'accès sur certains objets. Mais dans ce modèle, les sujets sont eux-mêmes des objets. Ainsi chaque sujet possède des droits de modification de la matrice de contrôle d'accès afin de créer ou de détruire des sujets ou des objets (ajout ou suppression de colonne). Mais un sujet a aussi la possibilité de modifier les droits que possède un autre sujet sur un objet.
HRU propose de modéliser la protection dans les systèmes d'exploitation comme suit :

- une matrice de contrôle d'accès P ;
- l'ensemble des sujets S et l'ensemble des objets O modélisés par l'ensemble des entiers de 1 à k ;
- l'ensemble des droits génériques R tels que `possession`, `lecture`, `écriture`, `exécution` ;
- un ensemble fini C de commandes $c_1, \ldots, c_n$, représente l'ensemble des opérations fournies par le système d'exploitation (création de fichier, modification des droits...) ;
- un ensemble d'actions élémentaires E : `enter` et `delete` pour l'ajout et la suppression de droits, `create subject` et `create object` pour la création de nouveaux sujets et objets et enfin `destroy subject` et `destroy object` pour la destruction de sujets et objets.

Du point de vue de la protection, les commandes de l'ensemble C ont toutes la même forme : elles prennent en paramètres une liste de sujets et objets, et suivant la présence de certains droits dans la matrice P, elles effectuent des actions élémentaires sur le système. La configuration du système de protection est représentée par le triplet (S, O, P).

Afin d'étudier le problème de la sûreté d'un système de protection, HRU s'intéresse au *transfert de privilège* (droit), qui se produit lorsqu'une commande insère un droit particulier r, dans une case de la matrice P où il était précédemment absent. Ce problème de sûreté peut être défini comme : "étant donné 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 du modèle HRU ont ainsi prouvé que :

- dans le cas d'un système de protection *mono-opérationnel*, i.e. dans lequel toutes les commandes ne contiennent qu'une seule action élémentaire, le problème de sûreté est *décidable*. Toutefois ce problème de vérification est *NP-complet*.
- dans le cas général, le problème de la sûreté d'un système de protection est *indécidable*.

Même si le problème de protection du modèle HRU à mono-opération est décidable et que ce système est facilement manipulable, il reste trop simple pour couvrir des politiques de sécurité réelles. Par exemple, la seule commande de création de fichiers d'un système d'exploitation de type UNIX se compose déjà de deux actions : création d'un nouvel objet, et positionnement des droits sur celui-ci. En outre, les auteurs mentionnent le fait que la taille du problème est réduite à une taille polynomiale dès lors que les actions de création de sujet ou d'objet sont retirées du système de protection.

2.3.1.3 Modèle TAM

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

Cette approche apporte au modèle HRU la notion de *type*. Un ensemble fini T de type de sécurité et un ensemble d'opérations élémentaires, permettant la gestion de ces types, est alors ajouté à la modélisation de HRU vue précédemment. En outre, cet ensemble est fini : la création ou la suppression de nouveaux types n'est pas possible. De plus, lorsqu'un objet est créé, son type est défini et n'est jamais modifié. Les opérations autorisées sur la matrice de contrôle d'accès dépendent alors des types de sujets et d'objets concernés.

Sandhu démontre ensuite que le problème de sûreté dans le cas de la version monotone (qui ne possède pas de requête de destruction ou de suppression) de ce modèle est décidable. Ce nouveau modèle, MTAM (Monotonic Typed Access Matrix), est obtenu en ôtant les opérations de suppression du modèle TAM. Toutefois la complexité de ce problème reste NP. C'est pourquoi, Sandhu définit le modèle MTAM ternaire, dans lequel toutes les commandes ont au maximum trois arguments. Au prix d'une perte d'expressivité, le problème de sûreté voit sa complexité ramenée à un degré polynomial.

Une version dite *augmentée* de TAM, appelée ATAM [Ammann et Sandhu 1992], a ensuite été proposée afin de fournir un moyen simple de détecter l'absence de droit dans une matrice de contrôle d'accès. L'objectif de cette démarche étant de pouvoir facilement modéliser la *séparation de privilèges*, celle-ci préconisant l'intervention de plusieurs utilisateurs pour mener à bien une tâche.

2.3.1.4 Discussion

Le modèle de contrôle d'accès couramment utilisé sur les systèmes d'exploitation actuels est le *Discretionary Access Control*. Le DAC délègue l'accord des permissions d'accès à la discrétion des propriétaires des ressources du système.

En pratique, ce modèle de contrôle d'accès a clairement montré ses limites. En effet, les attaques possibles contre les systèmes d'exploitation visent à obtenir un accès de niveau *super-utilisateur*

par abus de services système afin de gagner de nouveaux privilèges (en anglais, *privilege escalation*). Lorsqu'une telle attaque est réussie, l'attaquant obtient des pouvoirs qui outrepassent le DAC et donnent un accès complet à l'ensemble des ressources du système d'information.

De plus, diverses études[TCSEC 1985, Ferraiolo et Kuhn 1992, Loscocco *et al.* 1998] ont établi la faiblesse des modèles DAC. En effet, le contrôle d'accès discrétionnaire repose sur la capacité des utilisateurs à définir correctement les permissions sur les fichiers dont ils sont propriétaires. Toute erreur peut mener à une défaillance de sécurité. Par exemple si lorsque le fichier qui contient les mots de passe (`/etc/shadow` sous GNU/Linux) venait à être autorisé en écriture pour tous les utilisateurs, ceci pourrait modifier le mot de passe du super-utilisateur et obtenir ses droits.

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

L'introduction de la notion de *rôle* par [Ferraiolo et Kuhn 1992] se justifie par le fait que les modèles DAC ne répondent pas aux exigences de sécurité des systèmes d'exploitation, même non-militaires. Enfin, les critères d'évaluation de la sécurité des systèmes [TCSEC 1985, ITSEC 1991] établissent une nette différence de niveau de sécurité entre un système implantant seulement DAC, et un système implantant également le modèle MAC. Il ressort donc de l'analyse des modèles DAC que ceux-ci n'offrent pas de réelle garantie quant à la confidentialité et l'intégrité. Pour que les propriétés puissent être garanties, il est nécessaire d'utiliser un modèle MAC permettant de restreindre les droits accordés au *super-utilisateur* et éviter que les utilisateurs modifient les permissions de leurs fichiers de façon erronée.

2.3.2 Contrôle d'accès mandataire

L'objectif du contrôle d'accès mandataire (MAC) est d'imposer donc une politique non modifiable par les utilisateurs finaux, dans le but de garantir la sûreté du système. La politique définit l'ensemble des *interactions* valides entre sujets et objets. Pour contrôler les accès entre sujets et objets, Anderson propose d'utiliser un *Moniteur de Référence* (*Reference Monitor*). Cette matrice étant fixe, il devient alors possible de garantir que des droits d'accès considérés dangereux ne pourront être donnés ou obtenus par erreur. Ce concept de Moniteur de Référence est au cœur de la définition du Contrôle d'accès mandataire. Ce terme, qui désignait initialement le modèle défini par Bell&LaPadula, a vu sa signification évoluer et représente aujourd'hui tout mécanisme qui place la gestion de l'attribution des permissions d'accès hors d'atteinte des utilisateurs concernés par ces permissions.

Les différents modèles, présentés dans cette section, ont tous pour objectif de spécifier une politique de contrôle d'accès.

2.3.2.1 Modèle Bell et LaPadula

Le modèle Bell-LaPadula, établi par [Bell et La Padula 1973], plus couramment appelé BLP, est issu des besoins en confidentialité des données du monde militaire. Il a ainsi pour objectif de prévenir toute divulgation d'informations. Ce modèle repose sur le modèle HRU et exclut toute création ou destruction de sujets ou d'objets.

Ce modèle introduit la notion de *label* associé à chaque sujet et objet du système. Un label représente un *niveau de sécurité* et contient deux types d'identifiants de sécurité :

1. **un identifiant hiérarchique.** Cet identifiant représente le *niveau de classification* pour les objets (i.e. son niveau de sensibilité) ; et le *niveau d'habilitation* pour les sujets, qui sont typiquement : `non classifié`, `confidentiel`, `secret`, `top secret`. Ces niveaux de classification sont classés sous forme hiérarchique.
2. **des identifiants de catégories.** Ces différentes catégories d'informations correspondent aux différentes organisations manipulant les données, par exemple `militaire`, `privé`, `public`. Ces identifiants sont indépendants de la hiérarchie de confidentialité.

En plus du contrôle effectué à l'aide de la matrice de contrôle d'accès classique, ce modèle repose sur le respect de deux nouvelles règles :

ss-property ou *simple security property* : lorsqu'un sujet demande un accès en lecture sur un objet, son *niveau d'habilitation* doit être supérieur ou égal à celui de l'objet. Cette règle assure la confidentialité de l'information.

***-property** ou *star-property* : seuls les transferts d'informations depuis des objets de classification inférieure vers des objets de classification supérieure sont autorisés. Cette règle assure donc la prévention contre la divulgation d'informations.

Le système est donc modélisé de la façon suivante : un ensemble de sujets S, un ensemble d'objets O, une matrice d'accès M et une fonction $f : S \cup O \rightarrow 1 \ldots n$ retournant l'identifiant hiérarchique d'un sujet ou d'un objet. On dispose également d'un ensemble de permissions d'accès $A = \{e, r, a, w\}$ classées suivant leur capacité d'observation (lecture) et d'altération (écriture) de l'information :

e Ni observation ni altération (*execute*) ;

r Observation sans altération (*read*) ;

a Altération sans observation (*append*) ;

w Observation et altération (*write*).

Les règles précédentes, qui doivent être vérifiées par le mécanisme de contrôle d'accès, peuvent donc s'écrire :

$$r \in M[s, o] \Rightarrow f(s) \geq f(o)$$
$$r \in M[s, o_1] \wedge w \in M[s, o_2] \Rightarrow f(o_2) \geq f(o_1)$$

La vérification de la propriété * nécessite le contrôle de tous les flux d'informations entre sujets et objets possibles sur le système. Lors de l'implantation de ce modèle, l'existence de canaux cachés[8], dans le système d'exploitation cible, peut ainsi entraîner des problèmes de flux d'informations non contrôlables. Afin de prévenir ce problème, une version plus restrictive de la politique BLP utilise les règles suivantes :

No Read Up Lorsqu'un sujet demande un accès en lecture sur un objet, son *niveau d'habilitation* doit être supérieur ou égal à celui de l'objet.

No Write Down Lorsqu'un sujet demande un accès en écriture sur un objet, son niveau doit être inférieur ou égal à celui de l'objet.

[8] Les canaux cachés représentent les parties du système non-contrôlable par le mécanisme de contrôle d'accès, par exemple un tampon mémoire, un périphérique de type *raw*, etc.

Ce qui peut se traduire par :

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

L'article de Bell et LaPadula décrit l'intégration de ce modèle dans MULTICS, on le trouve également dans certaines versions de Solaris, HPUX ou autres systèmes UNIX. Généralement, ce modèle n'est pas désigné sous le nom BLP, mais plutôt sous l'acronyme MLS (Multi-Level Security pour modèle de sécurité multi-niveaux) comme dans l'Unix System V/MLS.

Cependant, l'implantation de ce modèle, sans aucune adaptation à l'environnement utilisé, peut être difficile. L'attribution des labels à certains sujets ou objets peut poser des problèmes, c'est par exemple le cas du dossier `/tmp` dans lequel n'importe quel processus est supposé pouvoir créer des fichiers. Certaines propriétés ont donc été ajoutées au modèle. Par exemple, le niveau de classification d'un objet passe à un niveau supérieur (celui du sujet) lorsqu'il est accédé en écriture par un sujet de niveau supérieur. Notons que dans le modèle initial, il est normalement interdit de modifier un fichier de niveau inférieur. L'effet néfaste lié à cet aménagement est que les objets ont tendance à être "tirés vers le haut", et donc se trouver sur le même niveau (le plus élevé) après un certain temps. On doit attirer l'attention sur le fait que le modèle BLP ne traite que la confidentialité et comme nous le verrons par la suite il ne garantit pas la confidentialité telle que définie dans[ITSEC 1991].

2.3.2.2 Modèle Biba

Alors que le modèle BLP vu précédemment ne répond qu'à des besoins de *confidentialité*, le modèle dit "Biba" [Biba 1975] vise à garantir l'*intégrité*. Il s'agit en réalité d'un modèle similaire à BLP qualifié de modèle *dual* à BLP. A chaque sujet et objet est associé un *niveau d'intégrité* qui correspond respectivement au "pouvoir d'accès" et au niveau d'intégrité (du sujet qui l'a créé). Les objectifs de sécurité de cette politique visent à :

- interdire toute propagation d'information d'un objet vers un autre objet de niveau d'intégrité inférieur ;
- interdire à tout sujet de modifier des objets possédant un niveau d'intégrité supérieur.

Ainsi, les règles relatives à la matrice de contrôle d'accès n'autorisent la modification du contenu d'un objet qu'aux sujets possédant un niveau d'intégrité suffisant. De plus, la communication entre sujets est prise en compte via la notion *"d'invocation de s_2 sur s_1"* représentant un flux d'informations unidirectionnel de s_1 vers s_2. L'ensemble de permissions d'accès A se voit alors enrichi d'un élément i correspondant à l'invocation. Deux règles permettent alors d'assurer l'intégrité :

No Read Down Un sujet n'est pas autorisé à accéder en lecture à un objet d'intégrité strictement inférieure, car cela pourrait corrompre sa propre intégrité ;

No Write Up Un sujet n'est pas autorisé à altérer le contenu d'un objet d'intégrité strictement supérieure.

Ces règles pouvant être écrites sous la forme :

$$r \in M[s,o] \Rightarrow f(s) \leq f(o)$$
$$w \in M[s,o] \Rightarrow f(s) \geq f(o)$$
$$i \in M[s_1,s_2] \Rightarrow f(s_1) \geq f(s_2)$$

Ces règles signifient que :

1. pour qu'un sujet s ait accès en lecture à un objet o, son niveau d'intégrité $f(s)$ doit être inférieur ou égal au niveau d'intégrité $f(o)$ de l'objet ;
2. pour qu'un sujet s ait accès en écriture à un objet o, son niveau d'intégrité doit être supérieur ou égal au niveau d'intégrité de l'objet ;
3. pour qu'un sujet s_1 puisse invoquer un sujet s_2, son niveau d'intégrité doit être supérieur ou égal au niveau d'intégrité du sujet invoqué.

Ces règles évitent à tout moment que ne se produise un transfert d'informations d'un niveau d'intégrité bas vers un niveau d'intégrité haut, ce qui signifierait une compromission de l'intégrité du niveau haut. La troisième règle découle du fait que si tous les canaux de flux d'informations apparaissent sous forme d'objets, alors une invocation de s_2 par s_1 s'apparente à une écriture par s_1 dans un certain objet o suivie d'une lecture de o par s_2.

A l'instar du modèle BLP, le modèle Biba est difficile à utiliser tel quel dans un système d'exploitation. Ici l'idée est de migrer un sujet vers un niveau d'intégrité inférieur lorsqu'il accède à un objet de niveau inférieur. L'effet néfaste est que l'ensemble des sujets va rapidement se trouver en bas de l'échelle des niveaux d'intégrité. Dés lors, l'intérêt du modèle Biba est fortement remis en cause, puisqu'il n'y a plus de différence entre les sujets. Finalement, le modèle Biba n'est qu'un modèle d'intégrité. Comme nous le verrons ensuite, il ne traite ni la propriété d'intégrité telle que définie dans [ITSEC 1991], ni la confidentialité.

2.3.2.3 Modèle DTE

Le modèle *Domain and Type Enforcement* (DTE) [Boebert et Kain 1985] est un modèle de contrôle d'accès de haut niveau. Disponible depuis des années dans certains systèmes d'exploitation commerciaux [Badger *et al.* 1995], il se rapproche de l'utilisation du "typage fort" disponible dans le modèle TAM et constitue une plate-forme sur laquelle peuvent être implantées des politiques de contrôle d'accès telles que Bell & LaPadula ou Biba. Concrètement, dans un système d'exploitation, les politiques de sécurité définies via le modèle DTE visent à :

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

Le modèle DTE reprend le principe de matrice de contrôle d'accès, mais la notion de *sujet* et *objet* est remplacée ici par la notion de *domaine* et de *type*. Ainsi chaque objet (fichier, message, mémoire partagée, socket, etc.) du système possède un type, et chaque processus s'exécute dans un domaine précis, dont découlent ses droits d'accès.

Le modèle DTE définit ainsi trois tables :

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

La définition des trois tables en DTEL est effectuée par un administrateur, et les utilisateurs ne peuvent y déroger. C'est pourquoi DTE est un modèle de contrôle d'accès mandataire (MAC). Notons finalement que la définition de ces tables pour un système complet est une tâche difficile et fastidieuse qui

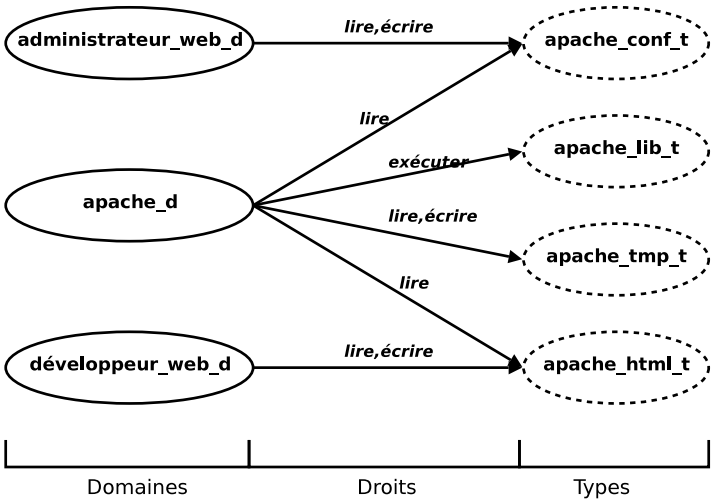


FIG. 2.4 – Configuration DTE pour apache

nécessite des connaissances précises sur le fonctionnement de chaque application et une modélisation de bas niveau du système.

Lorsqu'un nouveau processus est créé, il hérite automatiquement du domaine de son père. La DDT définit également les "points d'entrée" des domaines, c'est à dire quels sont les fichiers qui, une fois exécutés, vont créer un processus qui aura un domaine spécifique pouvant être différent du processus l'ayant invoqué. Dans la définition de ce modèle, il n'existe aucun autre moyen pour un processus d'opérer un changement automatique de domaine (une transition). La figure 2.4 est un exemple de politique DTE pour le serveur Web `apache`. Cette politique définit trois domaines distincts : un domaine `administrateur_web_d` consacré à l'administration de la configuration d'`apache` ; un domaine `développeur_web_d` destiné à l'écriture des pages Web ; et un domaine `apache_d` consacré à l'exécution du serveur.

Cette politique permet de garantir l'intégrité des fichiers de configuration et des pages Web en instaurant une *séparation de privilèges*. Ainsi chaque domaine est destiné à une tâche qui lui est propre, et ils ne peuvent interférer entre eux. En conséquence, même si `apache` s'exécute sous le compte administrateur `root`, il n'a accès qu'à ses fichiers de configuration, ses librairies et aux pages Web. De même seul `apache` a accès à son fichier de configuration, prévenant ainsi le système contre les attaques internes visant l'*intégrité* de ce fichier. De plus, ce type de confinement permettra d'endiguer certaines attaques sur le processus `apache` ayant pour objectif d'obtenir les privilèges administrateur. Par exemple une attaque de type *Buffer Overflow* sur `apache` peut entraîner l'exécution d'un *shellcode* afin d'ouvrir un terminal sera bloquée par le mécanisme de contrôle d'accès. Notons que ce n'est pas l'attaque (le *Buffer Overflow*) qui sera détectée ou bloquée mais la nécessité de droits supplémentaires, non nécessaires à `apache` et donc non présents dans la politique, afin de réussir la deuxième partie de l'attaque (ouverture d'un terminal). Mais ce confinement ne permettra pas d'empêcher une attaque ne nécessitant pas plus de droits que ceux accordés à un processus. Ainsi un vol d'information, par exemple dans une base de données ou un fichier accessible par apache, ne pourra être empêché.

2.3.2.4 Discussion

Les modèles *Mandatory Access Control* (MAC) ont été envisagés afin de répondre aux faiblesses des modèles DAC (cf. partie 2.3.1.4, page 43). On distingue deux orientations dans les modèles de type MAC. Une première orientation [Bell et La Padula 1973, Biba 1975, Clark et Wilson 1987b, Brewer et Nash 1989] répond à des problématiques spécifiques, par exemple la nécessité de disposer d'une habilitation adéquate pour la lecture de documents classifiés dans BLP. Cependant, les systèmes d'exploitation relèvent de problématiques plus complexes.

Une seconde orientation est le modèle DTE [Boebert et Kain 1985]. Plutôt que de considérer une propriété spécifique, DTE permet la spécification de politiques. Le problème de cette technique est qu'elle ne permet pas de spécifier les propriétés requises.

Par ailleurs, nous ne traitons pas les modèles à base de rôle (RBAC pour *Role-Based Access Control*) car ils n'apportent rien en terme de propriété de sécurité. Ces modèles simplifient uniquement l'administration.

Les modèles MAC fournissent des propriétés que l'on peut prouver. Par exemple, BLP repose sur deux lois qui empêchent la transmission directe d'information [Bell et La Padula 1973]. En revanche, les propriétés qu'ils permettent de garantir ne sont pas suffisamment larges. Il s'agit soit de garantir la confidentialité, soit l'intégrité. De plus, nous verrons que ces modèles ne traitent pas les séquences et donc ne garantissent pas les propriétés définies dans [ITSEC 1991].

2.4 Modèles d'administration de politiques de sécurité

Dans les parties précédentes, nous avons vu que le principe de politique de sécurité est un concept central dans le monde de la sécurité. Que ce soit dans le cadre du contrôle d'accès, où une telle politique définit l'ensemble des actions autorisées sur le système, ou dans le cadre de la détection d'intrusions, pour par exemple représenter l'ensemble des actions/comportements illégaux sur un système. Ainsi, de nombreuses approches reposent sur la définition d'une politique. Le principe de *Policy-Based Management* (PBM) [Sloman 1994, Lupu et Sloman 1999], ou administration par politique, s'applique particulièrement bien dans le cadre de systèmes homogènes. Malheureusement dans la tendance actuelle, les systèmes sont massivement répartis, en termes de services et d'utilisateurs, et hétérogènes, aussi bien au niveau équipement qu'au niveau applicatif. Dans cette optique, ce principe se révèle généralement problématique, notamment en termes d'administration et d'évaluation des politiques.

Afin de résoudre ces problèmes, de nombreux travaux portent sur des modèles d'administration de politiques de sécurité. Cette section présente deux modèles qui répondent à ces problématiques d'administration de la sécurité d'un système réparti.

Le premier modèle, appelé *Multi-domaines*, utilise un découpage en domaines du système réparti. Un domaine représente alors une organisation, une entreprise, un ensemble arbitraire d'équipements, etc. Chaque domaine possède alors une politique de sécurité qui lui est propre. Ce découpage en domaine relève typiquement d'une représentation hiérarchique du système réparti. Un mécanisme d'héritage est alors utilisé pour propager les politiques au sein ou entre domaines.

Le second modèle, appelé *Méta-politiques*, considère que différentes politiques de sécurité sont disponibles dans le cadre d'un système réparti. Des contraintes ou des opérations sont alors exploitées pour choisir quelles politiques ou combinaisons de politiques seront appliquées en fonction de l'environnement visé. On parle alors ici de "politiques de politiques", suivant l'expression donnée par [Hosmer 1992b].

2.4.1 Modèles multi-domaines

Cette approche part du principe suivant : "la gestion d'un système réparti ne peut pas être complètement centralisée, mais doit au contraire être partitionnée, de façon à refléter les sous-ensembles gérés par des administrateurs différents". C'est ici qu'apparaît la notion de *domaine* représentant un regroupement d'objets répondant à des besoins communs en termes d'administration.

Un domaine se définit donc comme un ensemble de références à des objets. Un objet est dit *membre direct* d'un domaine si celui-ci contient sa référence. De plus, un domaine peut également contenir des *sous-domaines* et un objet peut appartenir à plusieurs domaines. Les modèles multi-domaines apportent une solution au problème de l'administration des réseaux et systèmes distribués.

La notion de *domaine* d'application de politique a été formalisée pour la première fois par [Moffett *et al.* 1990]. Plusieurs articles [Moffett et Sloman 1991, Moffett et Sloman 1993, Sloman 1994] précisent ensuite cette formalisation. Selon [Sloman 1994], un tel modèle doit être composé de deux types de politiques :

1. Les *politiques d'autorisation* qui représentent l'ensemble des permissions d'un administrateur. Elles posent des contraintes, d'une part, sur les informations disponibles, et d'autre part, sur les opérations pouvant être effectuées.
2. Les *politiques d'obligation* qui définissent ce qu'un administrateur doit ou ne doit pas faire : elles guident ses décisions.

Une politique est alors définie comme un ensemble de règles régissant les interactions entre un *sujet* et une *cible* et de *contraintes* telles que des *contraintes temporelles* (intervalle légal d'ouverture d'une session) ou des *contraintes paramétriques* (ne pas autoriser plus de 10 connexions d'utilisateurs en même temps). En outre, une politique peut définir des *relations* avec d'autres politiques tel que l'*héritage de règles*, ou un *champ d'application* d'une politique.

2.4.1.1 Exemples d'implantations

Afin de représenter et d'appliquer ce modèle de politiques, différents travaux portent sur la définition et l'implantation de langage d'expression de politiques. Dans cette section, nous présentons trois exemples d'implantations de modèles multi-domaines : Ponder, LGI et Or-BAC. Notons que d'autres modèles pour l'expression de politique d'administration ont été proposés (*Management Policy*) [Marriott 1993, Wies 1994].

Ponder Ponder [Damianou *et al.* 2000] est conçu pour définir des politiques de sécurité à l'échelle de grands systèmes répartis, dans un langage abstrait. Ces politiques sont ensuite converties en configuration pour les équipements au moyen d'agents de traduction.

Le langage défini pour *Ponder* [Damianou *et al.* 2000] fournit une première implantation du modèle issu des travaux de Sloman. Il s'agit d'un langage déclaratif, orienté-objet, permettant la définition de politiques de sécurité et d'administration. En particulier, il spécifie des règles pour l'écriture de politiques d'autorisation, d'obligations événementielles, de restriction, et de politiques de délégation. Ce langage inclut également la définition de méta-politiques, mais celles-ci concernent la gestion de conflits entre politiques, ou des contraintes sur l'application d'ensembles de politiques sur un même élément, ou d'application d'une politique sur un ensemble d'éléments. Par exemple, ces méta-politiques peuvent contrôler le nombre maximum d'éléments sur lesquels une politique peut être appliquée. En outre, Ponder ne permet pas de contrôler la mise à jour de ces politiques.

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

LGI [Minsky et Ungureanu 2000] vise à résoudre les problématiques de négociation de politiques mutuelles lors d'interactions entre systèmes d'information. Par exemple, lorsque deux entreprises veulent partager des ressources, elles partageront leur politique LGI et une politique mutuelle sera automatiquement générée et appliquée sur chaque entreprise. Ainsi, LGI ne concerne donc pas l'administration d'un domaine de protection, mais les relations entre domaines.

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

Ici encore, il s'agit ni de contrôler ni de garantir des propriétés de sécurité tels que les flux d'informations interdits ou l'intégrité d'un système (cf. section 2.1.1).

2.4.2 Modèles Méta-Politique

Les modèles Méta-Politiques utilisent un second type de politique ayant pour objectif de décrire comment les différentes politiques disponibles peuvent être utilisées ou combinées.

Cette notion de méta-politique, introduit par Hosmer [Hosmer 1992a, Hosmer 1992b], doit permettre d'une part, de spécifier explicitement les conditions d'utilisation des politiques (par exemple, informations générales, contraintes temporelles ou organisationnelles, etc.), et d'autre part définir des règles de composition (relations entre politiques, coordination multi-politique, etc.).

[Belokosztolszki et Moody 2002] étudient l'application des méta-politiques à la description générique d'une politique afin de décrire une politique sans donner explicitement son contenu. Cet article s'intéresse alors aux accès inter-domaines, accès depuis un domaine à un service d'un autre domaine, dans des environnements multi-domaines où chaque domaine est administré de façon autonome. Afin de négocier des accords d'accès aux services inter-domaines , également appelés SLA (*Service-Level Agreement*), il est alors nécessaire d'échanger des informations sur les politiques de sécurité des domaines en interaction. C'est dans ce but que [Belokosztolszki et Moody 2002] proposent l'utilisation de méta-politiques. Dans un premier temps, elles sont employées pour diffuser des informations sur les politiques des domaines, permettant de les évaluer, mais sans pour autant diffuser les règles effectives de la politique. Ainsi ces méta-politiques contiennent des informations accessibles tel que le type d'une donnée ou d'un objet du domaine, les fonctions disponibles, les rôles, les règles explicites et les règles invariantes. Dans un second temps, des méta-politiques de conformité et d'interface sont établies, et sont utilisées pour évaluer les méta-politiques diffusées par les domaines. Ces méta-politiques de conformité contiennent par exemple la description des objets et des types d'accès associés, la classification des accès (lecture / modification), la description abstraite des utilisateurs, etc. Ces trois types de méta-politiques permettent ainsi de vérifier la légalité d'une demande de service inter-domaine et finalement d'accepter ou de refuser l'accès à ce service.

[Blanc *et al.* 2004] proposent un autre modèle de méta-politique pour l'administration décentralisée des nœuds d'un système réparti. L'implantation proposée dans [Blanc 2006] permet ainsi de déployer une politique de contrôle d'accès de type MAC dans un système réparti. Cette méta-politique repose sur l'utilisation de deux niveaux de contrôle définissant deux types de politique. La définition de deux niveaux de contrôle est ainsi étroitement liée à la définition de la méta-politique. En effet, son but est d'assurer simultanément le contrôle d'accès sur l'ensemble des nœuds d'un système réparti, et de garantir que les évolutions locales des politiques de contrôle d'accès ne violeront pas les objectifs de sécurité définis globalement. Deux types de politiques, la politique de protection et la politique de modification, permettent ainsi de générer une politique de contrôle d'accès et de contraindre l'évolution de cette politique.

La *politique locale de protection* est composée d'un ensemble de règles de contrôle d'accès. Contrairement aux modèles d'administration de politiques de sécurité précédentes, cet ensemble peut évoluer localement et indifféremment sur chaque nœud du système réparti. Cet ensemble contient toutes les interactions légales du système. Une interaction est alors définie comme une action légale entre un contexte de sécurité source (processus) et un contexte cible (tout objet du système y compris processus, mémoire partagée, socket, etc.). La notion de contexte de sécurité permet alors d'englober d'autres modèles de contrôle d'accès basés sur la définition d'attribut tels que DTE, BIBA, BLP ou le modèle RBAC. Ces règles, liées aux applications et services installés, peuvent différer d'un nœud à l'autre ; l'installation de nouvelles applications nécessite la modification de la politique de protection locale.

Une *politique de modification* permet alors de contraindre l'évolution de la politique locale de protection sur chacun des nœuds. Elle est déployée de façon identique sur chaque nœud, et ne peut être modifiée. Ainsi, dans le cadre de l'activité normale des nœuds, les opérations d'évolution ne peuvent à aucun moment violer les propriétés globales de sécurité visées par la définition de la politique de modification. La politique de modification définit des règles qui permettent à certains sujets d'ajouter, de modifier ou de supprimer des sujets, des objets ou des permissions entre sujets et objets, via un langage d'expressions régulières.

Cette Méta-Politique permet l'application d'une politique de contrôle d'accès sur des systèmes hétérogènes. Dans ce but, la politique de protection est exprimée dans un langage neutre, indépendant du mécanisme de sécurité cible. Un mécanisme, le *Language Adapter*, est alors en charge de la projection de ces règles vers le langage de configuration du système cible.

2.4.2.1 Discussion

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

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

Tel qu'exposé dans les articles introduisant le concept de Méta-Politique [Hosmer 1992a, Hosmer 1992b], l'objectif des Méta-Politiques est de décrire comment un ensemble de politiques doit être utilisé ou composé. Il s'agit ainsi de politiques "de haut niveau" qui réglementent l'utilisation des politiques "de bas niveau". Les articles concernant les modèles multi-domaines tels que [Sloman *et al.* 1992, Marriott 1993, Wies 1994] fournissent également des outils de caractérisation et de manipulation des politiques, comparables aux méta-politiques. Ce terme est ainsi cité dans [Sloman 1994], et intégré au langage Ponder [Damianou *et al.* 2000]. Il s'agit là de cas d'applications des concepts de méta-politiques beaucoup plus concrets que les notions données dans [Hosmer 1992a], bien que leur emploi se limite à la définition de contraintes sur la gestion des ensembles de politiques.

Des études ultérieures s'intéressent à la notion de méta-politique, dans le but d'en développer les usages. Selon [Avitabile 1998], les méta-politiques peuvent fournir des informations sur le cycle de vie des politiques (raffinement, délégation, conflits, exceptions, tests, application). Toutefois, il ne s'agit pas de gérer l'évolution d'une politique de sécurité ni de garantir des propriétés de sécurité système. L'article considère plutôt des procédés d'administration de politiques de haut niveau, et la façon dont ces procédés interviennent au cours de la vie de la politique.

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

Les solutions actuelles d'administration de politique ne prennent pas en compte la définition et la garantie de propriétés de sécurité. La problématique concerne les politiques réparties et la gestion des conflits entre domaines. Même si l'on peut garantir certaines propriétés de sécurité, ce n'est pas l'objectif visé par ces modèles. Il s'agit principalement de méthodes d'organisation ou de gestion de conflits simples entre règles de contrôle d'accès. Aucune de ces solutions ne propose de formalisation des propriétés de sécurité, ni un cadre général pour garantir ces propriétés.

2.5 Conclusion

Dans ce chapitre, nous avons présenté le concept général de politique de sécurité et des propriétés de sécurité qui en découlent. Nous avons ainsi énoncé les trois grandes classes de propriétés de sécurité : l'intégrité, la confidentialité et la disponibilité. Pour chacune de ces classes, nous avons défini une propriété générale et des cas particuliers. La garantie d'une politique de sécurité induit la garantie de toutes les propriétés qui la compose. Nous avons ainsi présenté les travaux de référence des domaines de la détection d'intrusions et du contrôle d'accès.

Le contrôle d'accès ne traite qu'une interaction prise isolément et ne contrôle pas les séquences d'interactions. Il est donc incapable de garantir des propriétés avancées de sécurité prenant en compte les séquences d'interactions. Nous avons présenté les différents modèles suivant trois axes : DAC, MAC et RBAC. Nous avons ensuite détaillé deux implantations système de contrôle d'accès obligatoire pour le noyau GNU/Linux. Les modèles de contrôle d'accès ont fait l'objet de nombreuses recherches, en particulier dans le domaine de la *vérification de sécurité*. Leur objectif est de permettre la vérification d'une certaine politique et d'offrir des garanties. Cependant ces vérifications portent sur un ensemble restreint de propriétés de sécurité. Ainsi, certains modèles ne visent que la garantie de propriétés de confidentialité ou ne visent que la garantie de propriétés d'intégrité. Surtout ces modèles ne traitent

pas des séquences de façon générale.

Nous avons montré que toutes les approches de détection d'intrusions actuelles présentent des limitations importantes :

- la détection à base de spécification de comportement peut se rapprocher du contrôle d'accès au sens où elle contrôle les interactions au sein d'une application. Cependant, elle ne permet pas de contrôler les séquences et ne couvre pas la sécurité de tout un système.
- le contrôle de non-interférence ne détecte que des violations d'une propriété d'intégrité particulière, c'est-à-dire l'intégrité d'exécution des processus privilégiés.
- le contrôle de flux de référence se restreint aux flux d'informations illégaux. Cette solution ne traite qu'un cas particulier de la confidentialité.

Ainsi, nous voyons que toutes ces solutions, qu'elles soient de type contrôle d'accès ou détection d'intrusions, ne traitent pas des séquences. Or comme nous le verrons par la suite la définition de propriétés d'intégrité et de confidentialité nécessitent la prise en compte des séquences. De plus toutes ces approches ne permettent de contrôler qu'une sous partie du système, c'est-à-dire soit le domaine utilisateur, soit les processus privilégiés. Les solutions d'administration de politiques s'intéressent quand à elle principalement aux conflits de politique. Il s'agit de conflits simples comme deux règles d'accès conflictuelles. Ces approches ne traitent pas non plus des séquences ou des propriétés de sécurité générales.

Enfin, l'état de l'art fait apparaître une grande difficulté à comparer les solutions car d'une part les auteurs adoptent leur propre formalisme et d'autre part ils ne traitent que des cas particuliers de propriété. Il est d'ailleurs souvent très difficile de faire confiance aux propriétés annoncées par les auteurs car s'ils annoncent des propriétés générales, on s'aperçoit en les analysant qu'il ne s'agit que de cas particuliers de propriété d'intégrité ou de confidentialité. Cette difficulté est liée à l'absence d'un formalisme applicable à toutes les propriétés définies par les auteurs et à un faible souci de précision du formalisme adopté par les auteurs. La formalisation que nous proposons par la suite mettra en évidence que les solutions actuelles ne fournissent pas de réponse générale au problème d'intégrité ou de confidentialité d'un système car elles ne traitent pas la notion de séquence. Dans cette thèse, nous avançons qu'un détecteur d'intrusions paramétré par une politique de sécurité doit signaler toutes les violations pour toutes les propriétés d'intégrité ou de confidentialité.

Dans le chapitre 3, nous proposons une modélisation d'un système et un langage de description des activités permettant d'exprimer l'ensemble des propriétés de sécurité définies dans ce chapitre, de les étendre et d'en proposer de nouvelles. Grâce à ce langage nous pourrons comparer plus finement les différentes approches de détection d'intrusions et de contrôle d'accès de la littérature.

Chapitre 3

Formalisation de propriétés de sécurité

Dans ce chapitre, nous proposons une formalisation permettant de classifier les actions qui peuvent être conduites sur un système. Elle permet de définir des propriétés de sécurité telle que la confidentialité ou l'intégrité d'un système. Notre classification des activités utilise : 1) des interactions (appels système), 2) des fermetures transitives causales d'interactions (séquences) et 3) des corrélations. Les corrélations permettent de modéliser des schémas d'attaques ou des propriétés de sécurité complexes reposant sur des combinaisons logiques et des compositions :

- d'interactions ;
- de séquences ;
- de séquences et d'interactions.

Ainsi, nous proposons une modélisation tout à fait générale des différentes classes d'activités observables sur un système. Cette modélisation inclut les classes d'activités basées sur des interactions, des séquences et des corrélations. Elle inclut aussi une classe où les activités ne peuvent pas être représentées par des séquences ou des corrélations. Pour cette dernière, il s'agit d'activités exploitant des événements qui apparaissent non liés causalement ou non corrélés. Les activités de cette dernière classe reposent en général sur des éléments non observables d'un point de vue système et dont les relations entre éléments doivent être déduites. Il s'agit d'attaques beaucoup moins fréquentes et plus complexes à mettre en œuvre, par exemple, un déni de service distribué où les attaquants se sont concertés par des moyens qui ne sont pas observables sur le système (par exemple, par téléphone) pour envoyer des messages en masse sans qu'il soit possible de trouver de liens entre les différents messages. En général, dans les attaques distribuées classiques, les différents messages ou événements peuvent tout de même être reliés entre eux ou être corrélés. Cette dernière classe d'attaque ou de propriété sort du cadre de cette étude. Il s'agit en général soit de propriété de disponibilité, soit d'attaque nécessitant des éléments extérieurs non observables sur le système (vol de mot de passe lors d'une communication orale, etc.). Nous ne traiterons donc que des attaques ou des propriétés correspondant à des actions observables, c'est-à-dire où l'ensemble des événements est visible sur le système ou peut être obtenu par corrélation.

Cette section reprend des notions existantes dans la littérature (contexte de sécurité, opérations élémentaires, interactions, transfert d'informations, transition) pour en donner une définition générale. Dans la littérature, ces notions ne sont pas toujours formalisées et elles utilisent des notations ou des appellations différentes. C'est pourquoi nous prenons le temps de les définir avec précision. Dans la section 3.2, ces notations seront utilisées pour définir notre langage de description d'activités. Ce langage inclut une relation de dépendance causale entre interactions, une relation de fermeture transitive de la dépendance causale et des opérateurs de corrélation pour ces relations. Les définitions

données ici sont tout à fait originale. Si la notion de dépendance causale est classique, nous en proposons une définition adaptée aux interactions d'un système. La notion de séquence permet de retrouver la transitivité du principe de causalité. Les opérateurs de corrélation permettent de décrire des activités complexes composées de différentes séquences et interactions combinées entre elles. Le langage ainsi proprosé est original et il a un sens pratique certain car il permet de formaliser, dans la section 3.3, toutes les propriétés de sécurité système rencontrées dans la littérature, de les étendre et d'en proposer de nouvelles. Son utilité est ainsi démontrée et nous en proposons une implantation dans le chapitre suivant.

La formalisation de propriétés de sécurité, développée dans ce chapitre, a également été exposée dans les articles [Briffaut 2005, Blanc *et al.* 2006, Briffaut *et al.* 2006c].

3.1 Représentation du système

L'exécution d'un système d'exploitation peut être vue comme un ensemble de sujets et d'objets interagissant ensemble dans le but d'effectuer des actions qui modifieront (ou non) l'état du système. Ces entités peuvent être séparées en deux ensembles : celui des *sujets*, qui sont les entités actives (processus), et celui des *objets*, qui correspondent aux entités passives (ressources, fichiers, sockets). Une opération effectuée par un sujet sur un objet peut alors être représentée par un triplet *(sujet, objet, type d'accès)*.

Dans cette section, nous proposons une modélisation des actions effectuées par les entités d'un système en terme d'*interactions* entre *contextes de sécurité*. Chaque contexte, représentant un *objet* ou un *sujet* du système, fait référence à un ensemble d'attributs. Un contexte fait référence à un ensemble de processus ou d'objets du système. Par exemple, le contexte sc_{apache} est associé aux processus `apache` et le contexte sc_{var_www} aux fichiers Web contenus dans `/var/www/`. Une interaction représente une action particulière (lire, écrire, exécuter, envoyer un signal, etc.) effectuée par un contexte sur un autre. De plus, ces interactions pouvent être catégorisées suivant leurs effets sur le système (transfert d'informations, changement de labels, etc.).

Basé sur ces définitions, nous proposons une modélisation d'un système en tant qu'ensemble d'interactions observables. Cette modélisation permet d'introduire, dans la section 3.2, une formalisation et une classification des activités possibles sur ce système. Finalement, dans la section 3.3, nous proposons une formalisation d'un ensemble de propriétés de sécurité pour un système utilisant cette représentation. Nous montrons, dans le chapitre 4, que cette modélisation peut être utilisée pour représenter une politique de contrôle d'accès. Ainsi, nous proposons une méthode d'analyse d'une politique de contrôle d'accès permettant d'extraire les descriptions d'activités qui correspondent à des violations possibles des propriétés de sécurité d'un système.

3.1.1 Contexte de sécurité

Un *contexte de sécurité* (*security context* en anglais), est un ensemble d'attributs ($attr_i$) associé à chaque entité du système. Chaque attribut est un *identifiant de sécurité* qui se réfère à un modèle de contrôle d'accès de type MAC ou une autre caractéristique tel qu'un nom de fichier ou de processus. Un attribut peut par exemple correspondre à l'identité (nom d'utilisateur) d'une entité, le rôle de l'entité dans le modèle RBAC, le domaine ou le type de l'entité du modèle DTE, un niveau de classification ou d'habilitation des modèles Biba ou Bell&LaPadula, etc. Un contexte de sécurité, constitué de plusieurs attributs, peut ainsi faire référence à plusieurs modèles de contrôle d'accès MAC. Chaque entité du système est étiquetée par un unique contexte de sécurité, noté sc_{label} et plusieurs entités

peuvent avoir le même contexte. Par exemple, tous les processus `apache` seront identifiés par le même contexte sc_{apache}. Nous noterons un contexte, constitué de n attribut, par :

$$sc_{label} = \{attr_1, attr_2, \ldots, attr_n\} \tag{3.1}$$

Sur un système, nous distinguons deux ensembles de contextes de sécurité correspondant aux deux types d'entité d'un système : l'ensemble des contextes de sécurité *sujets*, noté SC_S, et l'ensemble des contextes de sécurité *objets*, noté SC_O. L'ensemble des contextes de sécurité SC d'un système est donc l'union des deux ensembles précédents :

$$SC = SC_S \cup SC_O \tag{3.2}$$

L'utilisation des contextes de sécurité permet de représenter un système indépendemment du système d'exploitation sous-jacent. Ainsi, nous identifierons les entités d'un système par leur contexte de sécurité et non par leur "nom" ou leur chemin du système de fichiers.

3.1.2 Opération élémentaire

Chaque système d'exploitation définit un ensemble d'opérations, généralement appelées *appels système*, qui forment les briques de base de ce système. Ces opérations permettent de communiquer avec le noyau de ce système afin d'effectuer des opérations d'entrée/sortie (lecture/écriture sur un fichier), des communications inter-processus, etc. Une telle opération peut être vue comme une action qu'une entité du système peut effectuer sur une autre entité. Nous appelons ainsi *opération élémentaire*, notée eo, une opération "observable" du système et nous distinguons deux types d'opérations élémentaires :

- Les opérations élémentaires liées au système (i.e. appel système) ;
- Les opérations élémentaires liées au mécanisme de contrôle d'accès.

Une opération liée au système entraîne une modification de l'état du système (ouverture d'un fichier, envoi d'un signal, etc.). Les opérations liées au mécanisme de contrôle d'accès n'ont d'incidence que sur la représentation, interne au mécanisme de contrôle d'accès, des entités et n'entraînent pas de modification du système. C'est par exemple le cas lors d'une transition de contexte (changement de contexte de sécurité) où seul le contexte du processus, qui n'a d'existence qu'au niveau du mécanisme de contrôle d'accès, est modifié.

Par exemple, nous notons $\{x : y\}$ une opération élémentaire qui correspond à une opération y sur une classe d'objet x. De même, nous utilisons la notation $\{y\}$ pour ne caractériser que l'opération, par exemple $\{lire\}$. L'opération de lecture sur un fichier correspond à l'opération : $eo_1 = \{file : read\}$, l'opération d'écriture étant $eo_2 = \{file : write\}$. Ces deux exemples d'opérations élémentaires correspondent à des actions que les processus peuvent exécuter via le système d'exploitation. Certains mécanismes de contrôle d'accès, tel que SELinux, définissent un ensemble d'opérations élémentaires supplémentaires. Par exemple, l'opération de changement de contexte de sécurité pour un sujet correspond à l'opération élémentaire $eo_3 = \{process : transition\}$.

Dans la suite, nous notons IS (pour *Interaction Set*) l'ensemble des opérations élémentaires disponibles sur un système. Les éléments contenus dans cet ensemble dépendent du mécanisme de contrôle d'accès et du système utilisés.

3.1.3 Action élémentaire

Lors de l'exécution du système, les entités de ce système effectuent des actions, chaque action correspond alors à une opération élémentaire effectuée par une entité sur une autre. Nous appelons *action élémentaire*, une opération élémentaire exécutée par une entité sur une autre.

L'influence de ces actions élémentaires sur le système dépend implicitement du type de l'opération élémentaire effectuée. Nous distinguons ainsi trois classes d'actions élémentaires :

Interactions cette action correspond au cas général et représente une opération élémentaire quelconque ;

Transferts d'informations cette action est une interaction qui implique un *transfert d'informations* d'un contexte vers un autre ;

Transitions cette action est une interaction qui implique un *changement de contexte* d'un contexte vers un autre ;

Notons que le transfert d'informations et la transition sont des cas particuliers d'interactions. Nous allons maintenant détailler ces trois types d'actions élémentaires et leurs influences sur le système.

3.1.3.1 Interaction

Une *interaction* it, c'est-à-dire une opération élémentaire eo effectuée par un contexte de sécurité sc_{source} sur un contexte de sécurité sc_{cible}, est représentée par un triplet :

$$it = (sc_{source} \in SC_S, sc_{cible} \in SC, eo \in IS)$$

Une interaction correspond à une opération élémentaire eo quelconque. Le contexte, qui *effectue* l'action, est une entité active du système (un processus) ; le contexte de sécurité sc_{source} appartient donc à l'ensemble des contextes sujets. Le contexte cible, *subissant* l'action, peut être un sujet, par exemple lors d'une communication entre deux processus, ou un objet, par exemple lors de la lecture d'un fichier.

Définition 3.1.1 *Une interaction, notée $sc_{source} \xrightarrow{eo} sc_{cible}$, est une action élémentaire eo effectuée par un contexte de sécurité source sc_{source} sur un contexte de sécurité cible sc_{cible}.*

$$sc_{source} \xrightarrow{eo} sc_{cible} \quad \equiv_{def} \quad \begin{cases} it = (sc_{source} \in SC_S, sc_{cible} \in SC, eo \in IS), \\ sc_{source} \textit{ effectue } eo \textit{ sur } sc_{cible} \end{cases}$$

Par exemple, l'action élémentaire de type "interaction" $it = (sc_{apache}, sc_{var\_www}, \{file : read\})$ se note $sc_{apache} \xrightarrow{file:read} sc_{var\_www}$. Cet exemple correspond à l'opération de lecture $\{file : read\}$ effectuée par le processus identifié par le contexte sc_{apache} sur un contexte objet sc_{var_www} de type fichier.

Nous notons IT l'ensemble de toutes les interactions qu'il est possible de définir en prenant toutes les paires de contextes de sécurité et toutes les opérations élémentaires possibles entre ces paires.

3.1.3.2 Transfert d'informations

Sur un système d'exploitation, chaque entité peut être vue comme un conteneur d'informations ayant la capacité d'obtenir ou de transmettre de l'information à une autre entité. Cette information peut être stockée de manière persistante (système de fichiers, etc.) ou de manière volatile (pour un processus, mémoire partagée, etc.).

Deux types d'opérations permettent d'engendrer des transferts d'informations. Tout d'abord, l'opération de type "lecture" effectuée par un contexte de sécurité sc_1 sur un contexte sc_2 permet à sc_1 d'acquérir de l'information de sc_2. A l'inverse, une opération de type "écriture" effectuée par un contexte de sécurité sc_1 sur un contexte sc_2 transfert de l'information de sc_1 vers sc_2. Nous avons

déjà vu dans la section 3.1.2 (page 57) qu'une opération élémentaire, contrôlée par le mécanisme de contrôle d'accès, correspond à un appel système. Ces appels système peuvent induire la réception ou l'envoi d'informations (ou les deux). Dans notre représentation, chaque opération élémentaire eo peut être classée dans l'une des quatre catégories suivantes :

- `write_like` : envoi d'informations. Le contexte qui l'invoque est source du transfert d'informations ;
- `read_like` : réception d'informations. Le contexte qui l'invoque est cible du transfert d'informations ;
- `both` : envoi et réception d'informations, correspondant respectivement à `write_like` et `read_like`. Le contexte qui l'invoque est source ou cible du transfert d'informations. Notons que cette catégorie concerne peu d'opération élémentaire.
- `none` : n'induit aucun transfert d'informations.

Nous définissons ainsi deux fonctions permettant d'identifier la catégorie d'une opération élémentaire :

Fonction 3.1.1 `is_read_like : IS` $\rightarrow \{vrai, faux\}$*, est une fonction telle qu'étant donné une opération élémentaire, cette fonction retourne* `vrai` *si cette opération est du type* `read_like`*, sinon elle retourne* `faux`*.*

Fonction 3.1.2 `is_write_like : IS` $\rightarrow \{vrai, faux\}$*, est une fonction telle qu'étant donné une opération élémentaire, cette fonction retourne* `vrai` *si cette opération est du type* `write_like`*, sinon elle retourne* `faux`*.*

Notons que ces fonctions sont implantées à l'aide d'une table qui associe à chaque opération une catégorie. Des exemples de cette table seront donnés dans les chapitres 4 et 6.

De ce fait, l'interaction $it_1 = (sc_1, sc_2, eo_1)$ correspond à un transfert d'informations du contexte de sécurité sc_1 vers le contexte sc_2 , noté[1] $sc_1 > sc_2$, si l'opération élémentaire eo_1 est du type `write_like` ou `both` ; de même cette interaction correspond à un transfert d'informations du contexte sc_2 vers le contexte sc_1 ($sc_2 > sc_1$) si l'opération élémentaire eo_1 est du type `read_like` ou `both`.

Définition 3.1.2 *Un transfert d'informations, noté* $sc_{source} > sc_{cible}$*, est une interaction impliquant un transfert d'informations du contexte* sc_{source} *vers le contexte* sc_{cible}

$$sc_{source} > sc_{cible} \quad \equiv_{def} \quad \begin{cases} sc_{source} \xrightarrow{eo} sc_{cible}, \textbf{\textit{t.q.}}\ is\_write\_like(eo) \\ \vee \\ sc_{cible} \xrightarrow{eo} sc_{source}, \textbf{\textit{t.q.}}\ is\_read\_like(eo) \end{cases}$$

Chaque interaction, $sc_{source} \xrightarrow{eo} sc_{cible}$, correspond donc, en fonction du type de eo, à un des quatre cas suivant :

- $sc_{source} > sc_{cible}$;
- $sc_{cible} > sc_{source}$;
- $sc_{source} > sc_{cible}$ et $sc_{cible} > sc_{source}$;
- aucun transfert ;

Par exemple, l'interaction de lecture $sc_{apache} \xrightarrow{file:read} sc_{var\_www}$ est un transfert d'informations $sc_{var\_www} > sc_{apache}$ et l'interaction d'écriture $sc_{apache} \xrightarrow{file:write} sc_{var\_www}$ est un transfert d'informations $sc_{apache} > sc_{var\_www}$

[1] Nous utilisons le symbole $>$ pour représenter un transfert d'informations au lieu de $\xrightarrow{transfert}$ car il s'agit du symbole communément utilisé dans la littérature, par exemple dans [Bishop 2003] ou dans [Zimmermann 2003].

3.1.3.3 Transition

Les modèles de contrôle d'accès de type MAC utilisant RBAC, tel que SELinux ou GRSECURITY, permettent à une entité du système d'accéder à différents "rôles" (cf. annexe D). Chaque rôle est associé à un ensemble de privilèges différents sur le système. L'introduction de la notion de rôle a pour objectif de limiter au strict minimum les droits d'un utilisateur lorsqu'il exécute certaines tâches. Cette notion de rôle est à mettre en parallèle avec la notion de contexte de sécurité. En effet, dans notre représentation, le rôle d'une entité est un des attributs du contexte associé à cette entité. Le changement de rôle d'une entité se traduit donc par le changement du contexte associé à cette entité. La différence entre l'ancien et le nouveau contexte ne concerne alors que l'attribut associé au rôle. Ainsi, lorsqu'un processus ou un objet change de contexte, il effectue une interaction spécifique que nous appellerons une *transition*. Il en va de même pour la notion de domaine dans le modèle DTE. Lorsqu'une *transition* de domaine a lieu, l'attribut lié au domaine est modifié et le processus ayant effectué la transition obtient les nouveaux privilèges liés à ce domaine. Dans ces deux cas, il s'agit bien d'un changement du contexte associé à une entité du système via une interaction spécifique.

Par définition, une *transition de contexte* est une interaction, spécifique à certains systèmes, permettant à un processus P de changer son contexte. Cette transition entraîne l'obtention des privilèges liés au domaine cible et la perte des privilèges du domaine source. Une transition de contexte permet donc à un processus de raffiner ces privilèges en vue de l'exécution d'une tâche spécifique qui ne nécessite pas les mêmes privilèges que le contexte actuel.

Afin d'identifier les opérations élémentaires correspondant à une transition de contextes, nous définissons la fonction 3.1.3.

Fonction 3.1.3 $is\_transition : IS \rightarrow \{vrai, faux\}$ *est une fonction telle qu'étant donné une opération élémentaire* eo*, cette fonction retourne* `vrai` *si cette opération correspond à une transition,* `faux` *sinon.*

Définition 3.1.3 *Une transition, notée* $sc_{source} \xrightarrow{trans} sc_{cible}$*, est une interaction permettant à un processus, ayant le contexte* sc_{source}*, d'obtenir le contexte* sc_{cible}

$$sc_{source} \xrightarrow{trans} sc_{cible} \equiv_{def} sc_{source} \xrightarrow{eo} sc_{cible}, \textbf{\textit{t.q.}}\ is\_transition(eo)$$

Sous SELinux, l'opération élémentaire $\{process : transition\}$ est une opération qui correspond à une transition de contexte pour un processus. Par exemple, l'interaction $sc_{init} \xrightarrow{process:transition} sc_{apache}$, est une interaction qui permet au processus `init`, identifié par sc_{init}, d'obtenir le contexte sc_{apache}. Cette transition, notée $sc_{init} \xrightarrow{trans} sc_{apache}$, permet donc au processus `init` d'obtenir le contexte sc_{apache} associé au domaine de `apache`. Par exemple, lors du démarrage du système, lorsque le processus `init` exécute le service Web apache, le processus résultant (`apache`) est tout d'abord étiqueté par le contexte sc_{init} (par héritage du processus `init`). Cette transition, réalisée après l'exécution, permet de fixer le contexte du processus apache (sc_{apache}) et donc son domaine d'exécution (les droits associés à son nouveau contexte).

3.2 Langage de description d'activités

Dans cette thèse, nous nous intéressons aux activités résultant d'un ensemble d'interactions (d'appels système). Parmi ces activités, nous nous intéressons aux ensembles d'interactions ordonnés et à leurs corrélations. Un ensemble d'interactions ordonné se traduit par une suite d'interactions où chaque interaction est causalement liée à la précédente.

Par exemple, si sc_2 peut écrire dans un objet sc_3, alors sc_2 peut transférer de l'information vers sc_3 ($sc_2 > sc_3$). De plus, si sc_1 peut lire l'information contenue dans sc_3, alors la suite d'interactions $(sc_2, sc_3, \{file : write\})$ puis $(sc_1, sc_3, \{file : read\})$ permet à sc_1 d'obtenir de l'information provenant de sc_2. Nous montrerons que cette suite correspond à une dépendance causale entre deux interactions, que nous appellerons *séquence*.

La combinaison d'ensembles ordonnés se fait alors au moyen d'opérateurs de *corrélation* : *composition* ($\circ$), *et logique* ($\wedge$) et *ou logique* ($\vee$). Ces opérateurs permettent de décrire des activités complexes combinant des séquences et des interactions. Par exemple, si un contexte sc_1 souhaite exécuter un objet sc_4, il peut le faire par combinaison de deux séquences. Notamment, sc_1 peut transiter vers le contexte sc_3 au moyen d'une séquence composée de la fermeture transitive de deux interactions : $f = \{(sc_1, sc_2, \{process : transition\}), (sc_2, sc_3, \{process : transition\})\}$. Ensuite, sc_3 peut exécuter sc_4 via l'interaction $g = \{(sc_3, sc_4, \{file : execute\})\}$. Ainsi, la composition de ces deux séquences ($g \circ f$) permet à sc_1 d'exécuter sc_4.

L'approche que nous proposons permet de modéliser les interactions (appels système) observables sur une machine.Cependant, cette méthode permet aussi de traiter les échanges réseau dans la mesure où ceux-ci se traduisent bien par un appel système telle que la lecture ou l'écriture d'un message sur un descripteur réseau (une socket TCP/IP). Dans cette thèse, nous traiterons essentiellement des fermetures transitives causales, que nous appellerons *séquences*, et de leurs combinaisons. Finalement, nous proposons un langage de description des activités. Ce langage permettra ensuite de définir formellement un ensemble de propriétés de sécurité et de caractériser les attaques violant ces propriétés.

3.2.1 Catégorie d'activité

Nous définissons par *activité* un ensemble d'interactions. Nous proposons, dans cette partie, de catégoriser une activité en fonction des relations de causalité entre les interactions qui la composent. De ce fait, nous nous intéressons essentiellement aux ensembles ordonnés causalement et aux corrélations de ces ensembles.

Nous proposons de distinguer quatre classes d'activités, la figure 3.1 représente ainsi les inclusions entre ces différentes classes. La classe des *interactions* incluse dans toutes les autres classes, c'est-à-dire la classe élémentaire, correspond aux activités composées d'une seule interaction. Ces interactions ont été décrites dans la section 3.1.3.1. Une activité de ce type est donc réalisée à l'aide d'une seule interaction, par exemple sc_1 exécute sc_2 ($it = (sc_1, sc_2, \{file : execute\})$).

La seconde classe, la classe des *séquences*, qui englobe les interactions, correspond aux séquences d'interactions, c'est-à-dire aux fermetures transitives causales d'interactions. Ainsi, une activité de cette classe est décrite comme une suite d'interactions où chaque interaction dépend causalement de la précédente. Les interactions composant cette suite peuvent être de nature différentes. Il peut, par exemple, s'agir de la fermeture transitive d'une opération d'écriture dans un fichier ($(sc_1, sc_2, \{file : write\})$) et d'une opération de lecture ($(sc_3, sc_2, \{file : read\})$) permettant à un contexte sc_3 d'obtenir de l'information depuis sc_1.

La classe englobante, la classe des *corrélations*, est celle qui compose des séquences avec des *opérateurs de corrélation*. Par exemple, une activité $((it \circ seq_{trans_1}) \wedge (seq_{trans_2} \wedge (flux \vee seq_{int}))$ met en jeu une *séquence de transition* seq_{trans_1} afin d'accéder à un privilège correspondant à l'*interaction* it, et, une *séquence de transition* seq_{trans_2} combinée à, une *séquence de transfert d'informations* $flux$ ou une *séquence d'interactions* seq_{int}.

Finalement, la classe la plus globale inclut les corrélations, les séquences, les interactions, et les autres activités non exprimables dans notre modélisation. Ces autres activités peuvent, par exemple, nécessiter de prendre en compte la fréquence d'apparition pour être catégorisées. Pour l'instant, nos

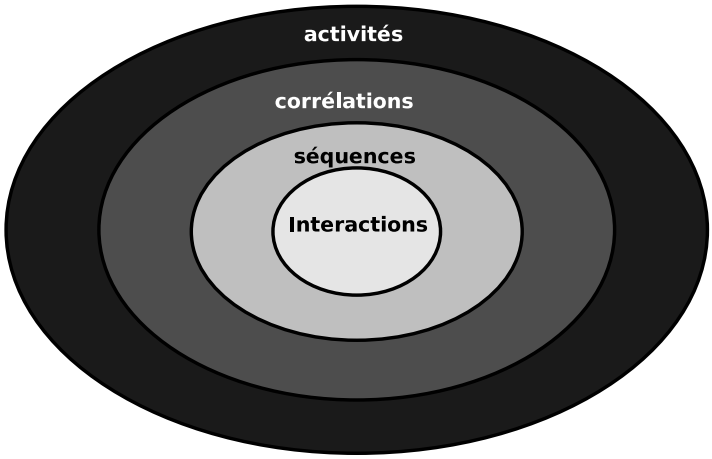


FIG. 3.1 – Classes d'activités et méthodes de contrôle

opérateurs de corrélation ne prennent pas en compte la fréquence ou l'intensité d'apparition des événements (interactions). Cependant, cette modélisaton pourrait aisément être étendue afin d'intégrer ces différents facteurs. Ces facteurs sont en général utilisés pour définir les propriétés de sécurité relatives à la disponibilité et notre étude se limite principalement aux propriétés d'intégrité et de confidentialité.

Les cas les plus intéressants, d'un point de vue système, correspondent aux trois premières classes d'activités. En effet, ces trois premières classes permettent de décrire des activités composées d'ensembles d'interactions causalement dépendantes et observables sur le système. De ce fait, les ensembles non ordonnés, qui nécessitent d'autres facteurs pour être catégorisés, sortent du cadre de cette étude.

3.2.2 Dépendance causale

Le principe de causalité spécifie que "la cause doit précéder l'effet". C'est l'aspect temporel de la relation qui nous intéresse ici. En effet, sur un système on peut tenir compte des dates des événements en se fiant à l'horloge locale du système. Le fait qu'un événement en précède temporellement un autre est une condition nécessaire pour avoir une dépendance causale entre ces deux événements. Cependant, la dépendance temporelle n'est pas une condition suffisante. Un événement A peut ainsi précéder l'événement B sans que A change les effets de B.

Dans la pratique, sur un système informatique, on ne peut tenir compte que du temps, sinon il faudrait connaître la sémantique des opérations (ce qui reviendrait, par exemple, à analyser les paramètres des appels système). Ainsi, on considère qu'une opération d'écriture a pour effet de modifier un objet même si cette opération ne modifie pas réellement d'information. On surestime donc les causes associées à un effet. Par exemple, s'il y a eu une séquence d'interactions $it_1 = sc_1 \xrightarrow{ecrire} sc_2$ puis $it_2 = sc_3 \xrightarrow{lire} sc_2$ sans que l'écriture ait modifié l'information contenue dans sc_2, nous considérons qu'il y a dépendance causale entre it_1 et it_2 en surestimant que l'écriture a un effet sur la lecture.

Quelque soit les définitions proposées par différents auteurs, la dépendance causale est généralement une surestimation de la relation de causalité. Par exemple, dans [Lamport 1978], un événement e précède causalement un événement e' correspond à l'estimation que e est une cause de e', mais cet estimateur n'est pas sûr. Nous nous plaçons dans ce même contexte, c'est-à-dire nous considérons que les dépendances causales sont toujours des surestimations de la relation de causalité.

Dans notre cas, une interaction it_1 précède causalement une interaction it_2 si it_1 précède temporellement it_2, et si it_1 et it_2 partage un même contexte. Ainsi, **nous dirons que it_1 est une "cause possible"** de l'apparition de it_2 ou du changement de l'effet de it_2. Bien sûr, cela n'implique nullement que it_1 est l'unique cause de it_2. Il peut toujours y avoir plusieurs causes possibles pour une interaction it_2, il s'agit de toutes les interactions précédant temporellement it_2 et ayant un contexte partagé avec celle-ci.

Dans cette partie, nous proposons deux définitions pour la dépendance causale. La première définition fournit un estimateur simple de la relation de cause à effet entre interactions, mais nous verrons que cet estimateur renvoie de fausses dépendances. C'est pourquoi nous sommes amenés à définir un second estimateur. Bien qu'étant toujours une surestimation, cette seconde définition supprime les fausses dépendances induites par la première définition. Afin d'avoir une relation de causalité transitive, nous définirons ensuite la notion de séquence. Si notre dépendance n'est pas une relation transitive, les séquences permettent d'exprimer les fermetures transitives d'interactions causalement dépendantes. Les séquences répondent donc à la nécessité d'un principe de causalité qui soit transitif.

3.2.2.1 Premier estimateur de la relation de causalité

Nous définissons un premier estimateur de la relation de causalité. Nous montrerons sur un exemple que cet estimateur ajoute des causes it_1 pour une interaction it_2 alors que it_1 ne modifie pas l'effet de it_2. En effet, ce premier estimateur ne minimise pas suffisamment les causes d'une interaction it_2 donnée.

Définition 3.2.1 *Soit deux interactions it_1 et it_2, nous dirons que l'interaction it_1 précède causalement l'interaction it_2, notée $it_1 \twoheadrightarrow it_2$, si :*

$$it_1 \twoheadrightarrow it_2 \quad \equiv_{def} \quad \begin{cases} it_1 \neq it_2, \\ sc_2 \in it_1, \\ sc_2 \in it_2, \\ t_{sc_2}(debut(eo_1)) \leqslant t_{sc_2}(fin(eo_2)) \end{cases}$$

Le contexte sc_2 est un contexte partagé par it_1 et it_2. La notation $t_{sc_2}(debut(eo_1)) \leqslant t_{sc_2}(fin(eo_2))$ signifie que l'action eo_1 débute sur sc_2 avant que eo_2 se termine sur sc_2. Les dates utilisées sont celles mesurées par ce contexte partagé qui peut être considéré comme un observateur temporel des interactions it_1 et it_2. Afin d'éviter la réflexivité de la relation $\twoheadrightarrow$, la définition considère que les deux interactions sont différentes $it_1 \neq it_2$ (c'est-à-dire, au moins un des trois éléments différents).

La figure 3.2 donne une représentation générale de cette notion de dépendance causale pour deux interactions it_1 et it_2. L'échelle des abscisses représente le temps, ainsi nous retrouvons que l'interaction it_1 est effectuée avant it_2. Le contexte sc_2 est le *contexte partagé* par ces deux interactions. Considérons par exemple, l'interaction $it_1 = (sc_1, sc_2, \{process : signal\})$ qui envoie un signal à sc_2 et l'interaction $it_2 = (sc_2, sc_3, \{process : signal\})$ qui envoie un signal à sc_3. Dans cet exemple, le premier signal eo_1 est bien une cause possible de l'apparition du signal eo_2. Ceci est bien sûr une surestimation de la notion de dépendance causale, il n'y a pas de certitude que it_1 soit une cause de l'apparition de it_2. Cependant, l'inverse est impossible car it_2 débute après la fin de it_1. Il est donc légitime de dire que l'interaction it_1 précède causalement it_2 : $it_1 \twoheadrightarrow it_2$.

Cette première définition surestime les causes car elle ne prend pas en considération les changements d'états des contextes mis en jeu. Pour illustrer cette surestimation, nous prenons deux exemples. Le premier montre que cette définition de la dépendance causale traduit une relation de causalité possible. Le second exemple montre, à l'inverse, que cette définition induit une relation de causalité

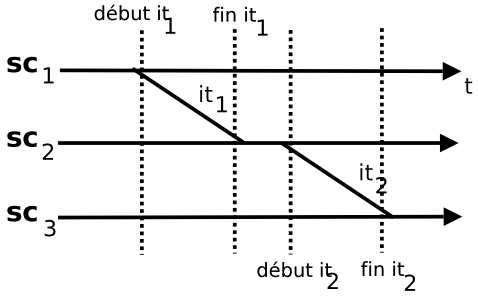


FIG. 3.2 – Dépendance causale entre deux interactions

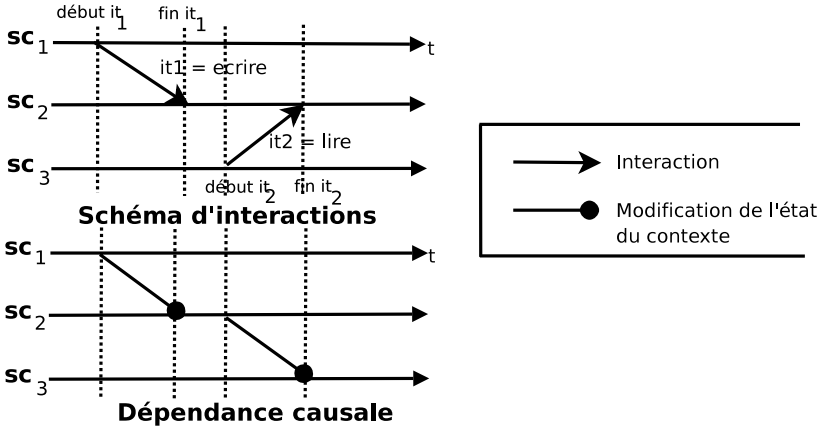


FIG. 3.3 – Interactions de lecture et d'écriture / Dépendance causale

impossible. C'est pourquoi nous serons amenés à proposer une seconde définition qui élimine ces fausses dépendances. Il est bien sûr évident que cette seconde définition correspond à un estimateur qui doit surestimer les dépendances. En effet, s'il est acceptable de les surestimer, il est inacceptable de les sousestimer. En effet, notre objectif étant de détecter l'apparition des activités illicites, la surestimation des dépendances causales peut entraîner de fausses alertes, mais la sousestimation aboutit à la non détection de certaines attaques.

Considérons un premier exemple avec les interactions $it_1 = (sc_1, sc_2, \{ecrire\})$ et $it_2 = (sc_3, sc_2, \{lire\})$. Ces deux interactions sont exécutées de manière ordonnée comme l'indique la figure 3.3. L'action eo_1 correspond à l'écriture par sc_1 dans l'objet sc_2 et l'action eo_2 correspond à la lecture par sc_3 de l'objet sc_2. Ainsi, c'est bien le changement d'état de sc_2, obtenu par l'opération d'écriture lors de it_1, qui est la cause du changement d'état possible de sc_3 (il s'agit dans ce cas d'un transfert d'informations). Il est donc légitime de dire que l'interaction it_1 a causé le changement d'état associé à it_2, it_1 précède donc causalement it_2 : $it_1 \twoheadrightarrow it_2$. Puisque l'opération d'écriture débute avant l'opération de lecture, la lecture ne peut pas précéder causalement l'écriture.

Considérons un second exemple avec les interactions $it_1 = (sc_1, sc_2, eo_1 = \{lire\})$ et $it_2 = (sc_3, sc_2, eo_2 = \{lire\})$. Nous pouvons dire que it_1 précède causalement it_2 car eo_1 apparaît avant eo_2, comme l'illustre la figure 3.4. Cependant la première interaction ne change pas l'état de sc_2 (il s'agit d'une lecture), son exécution n'influe donc pas sur it_2. Dans ce cas, la première définition de la

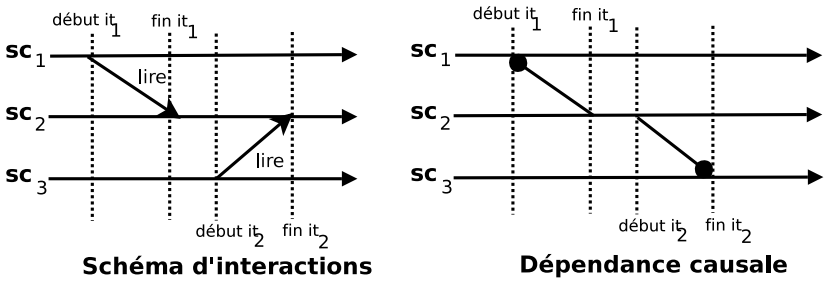


FIG. 3.4 – Interactions de lecture / Dépendance causale

dépendance causale estime mal la notion de cause à effet. En effet, comme l'illustre la figure 3.4, la modification de l'état du contexte sc_3 n'est pas une cause de l'interaction it_1.

3.2.2.2 Second estimateur de la relation de causalité

Ainsi, nous proposons une seconde définition qui raffine la relation de dépendance causale en éliminant les fausses dépendances. Cette seconde définition impose qu'it_1 modifie l'état du *contexte partagé* sc_2 et que it_2 modifie l'état du contexte final. Il s'agit là encore d'une surestimation qui cependant élimine les fausses dépendances impliquées par la définition précédente.

Définition 3.2.2 *Nous définissons la notion de dépendance causale, notée $\twoheadrightarrow$, comme :*

$$it_1 \twoheadrightarrow it_2 \quad \equiv_{def} \quad \begin{cases} sc_2 \in it_1, \\ sc_2 \in it_2, sc_3 \in it_2, \\ t_{sc_2}(debut(eo_1)) \leqslant t_{sc_2}(fin(eo_2)), \\ it_1 \textit{ modifie l'état du contexte partagé } sc_2, \\ it_2 \textit{ modifie l'état du contexte } sc_3 \end{cases}$$

Par la suite, nous utilisons cette seconde définition comme relation de dépendance causale.

Notons qu'avec cette nouvelle définition, l'exemple de la figure 3.3 conserve bien la notion de dépendance causale $it_1 \twoheadrightarrow it_2$, puisque l'opération d'écriture eo_1 modifie l'état du contexte sc_2 qui est ensuite lu par sc_3 lors de l'interaction eo_2. La seconde interaction it_2 modifie en conséquence l'état de sc_3. Par contre, dans l'exemple de la figure 3.4, il n'y a plus de dépendance causale entre it_1 et it_2 puisque l'opération de lecture it_1 ne modifie pas l'état de sc_2. Cette nouvelle définition élimine donc cette fausse dépendance. Cette définition de la dépendance causale n'est pas transitive. C'est pourquoi nous avons besoin de définir la notion de séquence qui représente une fermeture transitive d'interactions causalement liées.

3.2.3 Séquence d'interactions

Un élément de cette classe correspond à une suite d'interactions ayant des relations de cause à effet. Il s'agit en fait d'une fermeture transitive d'interactions où chaque interaction peut être de nature différente et où chaque interaction est liée causalement à la précédente.

Définition 3.2.3 *Une séquence de n interactions, notée $sc_{source} \Rightarrow sc_{cible}$, d'un contexte sc_{source} vers un sc_{cible} est une fermeture transitive de $n-1$ dépendances causales :*

$$sc_{source} \Rightarrow sc_{cible} \quad \equiv_{def} \quad \{it_1, \ldots, it_n\}, \begin{cases} n \geq 2, \\ sc_{source} \in it_1, \\ sc_{cible} \in it_n, \\ \forall k = 1..n-1, it_k \twoheadrightarrow it_{k+1} \end{cases}$$

La formulation fait clairement apparaître que, dans une séquence, les différentes opérations sont quelconques, c'est-à-dire que les n interactions it_k peuvent être différentes. De plus, les $n-1$ interactions sont dépendantes causalement : $sc_k \xrightarrow{eo_k} sc_{k+1}$ *précède causalement* $sc_{k+1} \xrightarrow{eo_{k+1}} sc_{k+2}$.

Nous proposons de définir différentes catégories de séquences en fonction du type d'interactions qui les composent. Ainsi, nous distinguons deux cas particuliers de séquences : le *flux d'informations* et la *séquence de transitions*. Nous allons maintenant détailler ces deux types particuliers de séquence et leur intérêt.

3.2.3.1 Flux d'informations

Un *flux d'informations*, noté $sc_{source} \gg sc_{cible}$, correspond à une fermeture transitive où chaque dépendance causale $it_k \twoheadrightarrow it_{k+1}$ correspond à deux transferts d'informations $sc_k > sc_{k+1} > sc_{k+2}$. Ainsi, les n interactions correspondent à n transfert d'informations : $(sc_1 = sc_{source}) > sc_2 > \ldots > (sc_{n+1} = sc_{cible})$. Ce type de séquence représente ainsi un transfert d'informations depuis sc_{source} vers sc_{cible} par séquence de transferts successifs.

La présence d'un *flux d'informations* entre deux contextes de sécurité sc_{source} et sc_{cible}, noté $sc_{source} \gg sc_{cible}$, correspond à la possibilité de transférer de l'information depuis sc_{source} vers sc_{cible} par composition d'au moins deux transferts d'informations.

Définition 3.2.4 *Un flux d'informations entre sc_{source} et sc_{cible}, noté $sc_{source} \gg sc_{cible}$, est une séquence $sc_{source} \Rightarrow sc_{cible}$, telle que chaque interaction de cette séquence correspond à un transfert d'informations :*

$$sc_{source} \gg sc_{cible} \quad \equiv_{def} \quad sc_{source} \Rightarrow sc_{cible} \begin{cases} sc_{source} \Rightarrow sc_{cible} = \{it_1, \ldots, it_n\}, \\ \forall k = 1..n, it_k = sc_k > sc_{k+1} \end{cases}$$

Un flux d'informations est donc une séquence d'interactions où chaque interaction it_k correspond à un transfert d'informations. Par exemple, la séquence d'interactions $\{(sc_1, sc_2, \{file : write\}), (sc_3, sc_2, \{file : read\})\}$ correspond au flux d'informations composé de deux transferts d'informations : $sc_1 > sc_2$ et $sc_2 > sc_3$. $sc_1 > sc_2$ transfert de l'information depuis sc_1 vers sc_2 et $sc_2 > sc_3$ transfert de l'information depuis sc_2 vers sc_3. Ce flux d'informations, $sc_1 \gg sc_3$ permet donc de transférer de l'information de sc_1 à sc_3 via sc_2.

3.2.3.2 Séquence de transitions

Une *séquence de transitions*, notée $sc_{source} \Rightarrow_{trans} sc_{cible}$, correspond à une fermeture transitive où chaque dépendance causale $it_k \twoheadrightarrow it_{k+1}$ correspond à deux transitions $sc_k \xrightarrow{trans} sc_{k+1} \xrightarrow{trans} sc_{k+2}$. Ainsi, les n interactions correspondent à n transitions : $(sc_1 = sc_{source}) \xrightarrow{trans} sc_2 \xrightarrow{trans} \ldots \xrightarrow{trans} (sc_{n+1} = sc_{cible})$. Ce type de séquence est un changement de contexte possible de sc_{source} en sc_{cible} par séquence de transition.

Nous appelons ainsi *séquence de transitions*, notée $sc_{source} \Rightarrow_{trans} sc_{cible}$, la fermeture transitive d'un ensemble de *transitions*. Ce type de séquence permet à un processus, étiqueté par le contexte sc_{source}, d'"obtenir" le contexte sc_{cible} par composition d'au moins deux transitions.

Définition 3.2.5 *Une séquence de transitions entre sc_{source} et sc_{cible}, notée $sc_{source} \Rightarrow_{trans} sc_{cible}$, est une séquence $sc_{source} \Rightarrow sc_{cible}$ telle que chaque interaction de la séquence correspond à une transition :*

$$sc_{source} \Rightarrow_{trans} sc_{cible} \quad \equiv_{def} \quad sc_{source} \Rightarrow sc_{cible} \begin{cases} sc_{source} \Rightarrow sc_{cible} = \{it_1, \ldots, it_n\}, \\ \forall k = 1..n, it_k = sc_k \xrightarrow{trans} sc_{k+1} \end{cases}$$

Une séquence de transitions est donc une séquence où chaque interaction it_k correspond à une transition. Par exemple, la séquence d'interactions $\{(sc_1, sc_2, \{process : transition\}), (sc_2, sc_3, \{process : transition\})\}$ correspond à une *séquence de transitions* composée de deux transitions : $sc_1 \xrightarrow{trans} sc_2$ et $sc_2 \xrightarrow{trans} sc_3$. $sc_1 \xrightarrow{trans} sc_2$ correspond à un changement de contexte de sc_1 vers sc_2 et $sc_2 \xrightarrow{trans} sc_3$ correspond à un changement de contexte de sc_2 vers sc_3. Cette séquence de transitions, $sc_1 \Rightarrow_{trans} sc_3$ correspond donc à un changement de contexte de sc_1 vers sc_3. Le processus, initialement étiqueté par sc_1, qui exécute cette séquence, sera donc au final étiqueté par sc_3. C'est pourquoi nous disons que cette séquence correspond à un changement de contexte d'un processus.

3.2.4 Corrélation

Les deux premières classes d'activités sont constituées des interactions seules et des séquences. La troisième classe est constituée des corrélations de séquences ou d'interactions. Dans cette partie, nous définissons tout d'abord nos opérateurs de corrélation et nous donnerons deux cas particuliers de corrélations de séquences : l'*accès à un privilège* et l'*accès à l'information*.

L'*accès à un privilège* eo correspond à une composition qui surestime la possibilité, pour un contexte, d'accéder à un privilège. Il s'agit de la composition d'une séquence et d'une interaction : $sc_{source} \xrightarrow{eo_1} \ldots \xrightarrow{eo_n} sc_{inter}$ et $sc_{inter} \xrightarrow{eo} sc_{cible}$. Cette composition caractérise une activité dont l'objectif est d'obtenir (utiliser) le privilège eo sur sc_{cible}. Dans ce cas, il y a dépendance temporelle entre la séquence d'interactions et l'obtention du privilège. En effet, pour obtenir le privilège eo, la séquence doit précéder l'obtention du privilège. En fait, l'accès à un privilège peut être considéré comme une séquence $sc_{source} \Rightarrow sc_{cible}$ particulière où la dernière opération correspond à eo.

L'*accès à l'information* correspond à la corrélation qui surestime la possibilité, pour un contexte, d'accéder à de l'information. Il s'agit de la composition d'une séquence entre sc_{acces} et sc_{inter} et d'un flux d'informations entre sc_{info} et sc_{inter} : $sc_{acces} \xrightarrow{eo_1} \ldots \xrightarrow{eo_n} sc_{inter}$ et $sc_{info} > \ldots > sc_{inter}$. Contrairement à l'accès à un privilège, il n'y a cette fois pas de contrainte temporelle entre la fermeture transitive et le transfert d'informations. En effet, l'information peut arriver par transfert dans le contexte sc_{inter}, puis le contexte sc_{acces} peut par exemple transiter jusqu'à ce contexte pour y accéder ; ou inversement, le contexte sc_{acces} peut transiter jusqu'au contexte sc_{inter} et peut ensuite recevoir de l'information depuis sc_{info}. Cette corrélation peut donc se produire sans dépendance temporelle.

3.2.4.1 Opérateurs de corrélation

Les opérateurs de corrélation permettent de traiter les activités qui combinent plusieurs séquences ou interactions. Ces opérateurs permettent ainsi de "corréler" plusieurs séquences ou interactions afin

de décrire une activité complexe. Nous définissons tout d'abord l'opérateur de composition $\circ$, puis les deux opérateurs logiques $\wedge$ et $\vee$.

A toute séquence $f = sc_1 \xrightarrow{eo_1} \ldots \xrightarrow{eo_{n-1}} sc_n$, aussi notée $sc_1 \Rightarrow sc_n$, nous associons une fonction F qui, pour le premier contexte sc_1 de cette séquence, retourne le dernier contexte sc_n, nous avons donc : $F(sc_1) = sc_n$.

Définition 3.2.6 *Soit $f = sc_{f_1} \Rightarrow sc_{f_n}$ et $g = sc_{g_1} \Rightarrow sc_{g_m}$ deux séquences tel que $sc_{f_n} = sc_{g_1}$.*
La composition des deux séquences f et g correspond à la séquence $sc_{f_1} \Rightarrow (sc_{f_n} = sc_{g_1}) \Rightarrow sc_{g_m}$ pour laquelle la composition des deux fonctions $G \circ F$ s'applique : $G \circ F(sc_{f_1}) = G[F(sc_{f_1})] = G(sc_{f_n}) = G(sc_{g_1}) = sc_{g_m}$. Étant donné que la composition des fonctions F et G s'applique à la séquence résultante, nous notons $g \circ f$ cette nouvelle séquence $sc_{f_1} \Rightarrow sc_{g_m}$.

Cette définition implique une dépendance causale entre la dernière interaction de f et la première interaction de g : $it_{f_{n-1}} \twoheadrightarrow it_{g_1}$

Les opérateurs $\wedge$ et $\vee$ permettent de définir des corrélations entre séquences, interactions ou toutes autres corrélations. La notion de dépendance causale entre les interactions d'une séquence implique que chaque interaction doit être observée en respectant l'ordre causal. Ainsi, la dépendance $it_1 \twoheadrightarrow it_2$ implique que l'interaction it_1 doit être observée avant it_2. Nous définissons ainsi la notion de *validité* d'une séquence :

Définition 3.2.7 *Soit une fonction V qui, appliquée à une séquence, renseigne sur la validité d'une séquence :*

$$V : IT^n \rightarrow \{vrai, faux\}$$

Soit $s = it_1, \ldots, it_n$ une séquence, la fonction V retourne $vrai$ si toutes les interactions de s ont été observées dans l'ordre sur le système, c'est-à-dire si it_1 est observée avant it_2 qui est observée avant it_3 etc., $v = faux$ sinon.

A partir de cette notion de validité d'une séquence, nous pouvons définir nos opérateurs logiques :

Définition 3.2.8 *Soit deux interactions, séquences ou corrélations a et b, $(V(a) \wedge V(b))$ est une formule booléenne qui est* `vraie` *si $V(a)$ est* `vraie` ***et*** *$V(b)$ est* `vraie`*.*
Par abus, nous dirons qu'une $(a \wedge b)$ est `vraie` *si $(V(a) \wedge V(b))$ est* `vraie`*.*

Définition 3.2.9 *Soit deux interactions, séquences ou corrélations a et b, $(V(a) \vee V(b))$ est une formule booléenne qui est vraie si $V(a)$ est* `vraie` ***ou*** *$V(b)$ est* `vraie`*.*
Par abus, nous dirons qu'une $(a \vee b)$ est `vraie` *si $(V(a) \vee V(b))$ est* `vraie`*.*

Pour ces opérateurs, il n'y a pas de dépendance temporelle lors de la réalisation de a et b. Ainsi, a peut se dérouler avant b ou après b, de même a et b peuvent se dérouler simultanément. Ainsi, une activité $(a \wedge b)$ correspond à la réalisation de a et de b, une activité $(a \vee b)$ correspond à la réalisation de a ou de b.

3.2.4.2 Langage de description d'activités

Nous avons vu qu'une activité pouvait être exprimée sous forme d'une interaction (action élémentaire) ou d'une séquence. De plus, les trois opérateurs de corrélation permettent d'agrandir le spectre des descriptions d'activités. Ainsi une activité peut être décrite comme une corrélation de plusieurs

séquences, compositions, ou corrélations. Nous proposons donc une grammaire pour le langage d'expressions des activités :

$$
\begin{array}{rl}
activite & ::= [\ description\ "="\ ]\ correlation \\
correlation & ::= (correlation \wedge correlation)|(correlation \vee correlation)|composition \\
composition & ::= (composition \circ composition)|terminal \\
terminal & ::= sequence|action\_elementaire \\
action\_elementaire & ::= sc \xrightarrow{eo} sc|sc > sc|sc \xrightarrow{trans} sc \\
sequence & ::= sc \Rightarrow sc|sc \gg sc|sc \Rightarrow_{trans} sc \\
sc & ::= "\ contexte\ de\ securite\ " \\
eo & ::= "\ operation\ elementaire\ " \\
description & ::= "\ nom\ ou\ type\ de\ l'activite\ "
\end{array}
$$

Ainsi, un terminal de ce langage est une action élémentaire ou une séquence, ces terminaux pouvant être composés avec l'opérateur $\circ$ afin de décrire de nouvelles séquences. Ces séquences peuvent ensuite être composées avec les opérateurs $\wedge$ et $\vee$ afin de décrire des activités complexes. Une description d'activité associe donc à une description textuelle, une corrélation de séquences. Ainsi, il est possible de composer ces différents opérateurs pour obtenir, par exemple, une activité qui correspond à la combinaison d'une séquence de transitions, d'une séquence d'interactions et d'un flux d'informations. Une telle activité peut, par exemple, correspondre à l'ouverture d'une session par un utilisateur distant, suivie de l'installation d'un logiciel de *scan* réseau et finalement de la récupération des résultats du *scan*. Cette grammaire prend en compte les cas particuliers d'interactions (transfert d'informations et transitions) et de séquences (flux d'informations et séquences de transitions). Bien que cette grammaire pourrait ne prendre en compte que les interactions et les séquences générales ($sc \xrightarrow{eo} sc$ et $sc \Rightarrow sc$), les cas particuliers permettent de raffiner la description d'une activité. Ainsi, cette grammaire permet de décrire des activités complexes combinant ces diverses séquences et interactions.

Nous allons maintenant utiliser cette grammaire, afin de définir deux types de corrélations particulières : l'*accès au privilège* et l'*accès à l'information*. L'*accès à un privilège* eo correspond à une composition surestimant la possibilité, pour un contexte, d'accéder au privilège *d'effectuer* eo *sur un contexte donné*. L'*accès à l'information* correspond à la corrélation qui surestime la possibilité, pour un contexte, d'accéder à de l'information.

3.2.4.3 Accès à un privilège

Les privilèges accessibles par une entité sc_{source} ne dépendent pas seulement des interactions qu'elle peut réaliser mais aussi des privilèges atteignables via une séquence. Nous appelons ainsi *accès à un privilège* eo sur sc_{cible}, notée $sc_{source} \Rightarrow_{eo} sc_{cible}$, la composition d'une séquence d'interactions ($sc_{source} \Rightarrow sc_{inter}$) et d'une interaction accédant au privilège eo de sc_{cible} ($sc_{inter} \xrightarrow{eo} sc_{cible}$).

Définition 3.2.10 *Soit* $seq = sc_{source} \Rightarrow sc_{inter}$ *une séquence d'interactions et* $it = sc_{inter} \xrightarrow{eo} sc_{cible}$ *une interaction, la composition* $sc_{source} \Rightarrow_{eo} sc_{cible} = (it \circ seq)$ *est une surestimation de la possibilité d'accès au privilège* eo *sur* sc_{cible} *par* sc_{source}.

Notons qu'il y a une contrainte de précédence temporelle entre la séquence et l'interaction.

Ce type de composition peut, par exemple, correspondre à l'envoi de messages entre plusieurs processus allant de sc_{source} à sc_{inter} qui a pour conséquence l'exécution de sc_{cible} par le processus

sc_{inter} ($sc_{inter} \xrightarrow{execute} sc_{cible}$).

Cette définition de l'*accès à un privilège* représente le cas général de la possibilité d'accès à un privilège. Notamment, dans l'exemple précédent, les envois de messages entre processus ne sont pas forcément la cause de l'exécution du privilège par sc_{inter}. Il s'agit bien sûr d'une surestimation de la possibilité réelle d'obtention du privilège. Nous pouvons cependant définir un cas particulier d'accès à un privilège où la séquence d'interaction correspond à une séquence de transitions. Comme indiqué dans la partie 3.1.3.3 (page 60), une transition de contexte ne modifie pas l'état d'un processus mais le contexte qui lui est associé et donc son ensemble de privilèges. Cette composition particulière correspond donc bien à un accès direct à ce privilège. Ainsi, il ne s'agit plus ici d'une surestimation d'accès à un privilège.

Nous appelons ainsi *accès à un privilège par transition*, notée $sc_{source} \Rightarrow_{trans\_eo} sc_{cible}$, une séquence de transitions $sc_{source} \Rightarrow_{trans} sc_{inter}$ suivie d'une interaction $sc_{inter} \xrightarrow{eo} sc_{cible}$.

Définition 3.2.11 *Soit $seq_{trans} = sc_{source} \Rightarrow_{trans} sc_{inter}$ une séquence de transitions et $it = sc_{inter} \xrightarrow{eo} sc_{cible}$ une interaction, la composition $sc_{source} \Rightarrow_{trans\_eo} sc_{cible} = (it \circ seq_{trans})$ est un accès direct au privilège eo de sc_{cible} par sc_{source}.*

3.2.4.4 Accès à l'information

Sur un système, une information est accessible par *flux d'informations* ($sc_{info} \gg sc_{acces}$) mais aussi par corrélation d'un flux et d'une séquence. Nous appelons ainsi *accès à l'information*, notée $sc_{acces} \Rrightarrow sc_{info}$, la corrélation d'une séquence $sc_{acces} \Rightarrow sc_{inter}$ et d'un flux d'informations $sc_{info} \gg sc_{inter}$.

Définition 3.2.12 *Soit $seq = sc_{acces} \Rightarrow sc_{inter}$ une séquence d'interactions et $flux = sc_{info} \gg sc_{inter}$ un flux d'informations, la corrélation $sc_{acces} \Rrightarrow sc_{info} = (seq \wedge flux)$ est une surestimation de la possibilité d'accès à l'information contenue dans sc_{info} par sc_{acces}.*

Notons que l'opérateur $\wedge$ n'implique pas de contrainte de précédence temporelle entre ces séquences. Une telle corrélation peut, par exemple, consister à la mise en place d'un canal caché entre sc_{acces} et sc_{inter}. Cela permet indirectement au processus sc_{acces} d'obtenir les informations transmises par sc_{info} à sc_{inter}.

Cette définition d'un *accès à l'information* modélise de façon générale une activité qui permet d'accéder à l'information d'un autre contexte. Cependant, il s'agit d'une surestimation de la possibilité d'accès à l'information. Nous proposons donc un cas particulier d'*accès à l'information* où chaque interaction composant la première séquence correspond à une transition. Ainsi, lorsque ce processus accède à un flux d'informations, il y accède directement.

Définition 3.2.13 *Soit $seq_{trans} = sc_{acces} \Rightarrow_{trans} sc_{inter}$ une séquence de transitions et $flux = sc_{info} \gg sc_{inter}$ un flux d'informations, la corrélation $sc_{acces} \Rrightarrow_{trans} sc_{info} = (seq_{trans} \wedge flux)$ est une possibilité d'accès direct à l'information contenue dans sc_{info} par sc_{acces}.*

Nous appelons ainsi, *accès à l'information par transition*, ce cas particulier.

3.2.5 Synthèse

Dans cette section, nous avons proposé une classification des activités sous forme d'ensembles ordonnés. Cette classification permet de décrire une activité comme une interaction unique, une sé-

	Interaction	**Séquence**	**Corrélation**
Cas général	Interaction $sc_1 \xrightarrow{eo} sc_2$	Séquence d'interactions $sc_{source} \Rightarrow sc_{cible}$	Accès à l'information $sc_{source} \Rrightarrow sc_{cible}$
			Accès à un privilège $sc_{source} \Rightarrow_{eo} sc_{cible}$
Cas Particulier	Transfert d'informations $sc_1 > sc_2$	Flux d'informations $sc_{source} \gg sc_{cible}$	Accès à l'information par transition $sc_{source} \Rrightarrow_{trans} sc_{cible}$
	Transition $sc_1 \xrightarrow{trans} sc_2$	Séquence de transitions $sc_{source} \Rightarrow_{trans} sc_{cible}$	
			Accès à un privilège par transition $sc_{source} \Rightarrow_{trans\_eo} sc_{cible}$

TAB. 3.1 – Synthèse des notations des activités

quence ou une corrélation. Pour chaque classe, nous avons proposé des cas particuliers, tel que le transfert d'informations ou le flux d'informations, afin de prendre en compte des comportements spécifiques.

La table 3.1 fait la synthèse des classes d'activités que nous distinguons. Trois colonnes séparent les activités basées sur une *Interaction*, une *Séquence*, ou une *Corrélation* de ces dernières.

Considérons pour l'instant le cas général, c'est-à-dire une activité quelconque (première ligne du tableau). Dans la colonne de la classe *Interaction*, on ne trouve que les activités basées sur une seule *interaction*. De même, la classe séquence fait apparaître les notations utilisées pour les activités utilisant des *séquences d'interactions*. Enfin, dans la colonne *Corrélation*, on retrouve les deux catégories générales d'activités : l'*accès à l'information* et l'*accès à un privilège*.

A partir de ces descriptions générales de classes d'activités, nous résumons les cas particuliers d'activités qui utilisent des opérations élémentaires spécifiques. Dans la colonne *Interaction*, nous retrouvons les activités du type *transfert d'informations* et du type *transition*. Dans la colonne *Séquence*, nous retrouvons les activités du type *flux d'informations* et *séquence de transitions*. Finalement, la colonne *Corrélation* contient les activités du type *accès à l'information par transition* et *accès à un privilège par transition*.

Nous proposons, dans la section suivante, une formalisation des propriétés de confidentialité et d'intégrité. La classification des activités et le langage de description d'activités servent de base à cette formalisation.

3.3 Propriétés de sécurité

Dans la section 2.1.1, nous avons défini, de façon générale, les trois classes principales de propriétés de sécurité : l'*intégrité*, la *confidentialité* et la *disponibilité*. De plus, dans le chapitre 2 nous avons étudié, à travers des systèmes de détection d'intrusions et des mécanismes de contrôle d'accès, diverses propriétés de sécurité tels que la condition de non-interférence, le modèle de confidentialité BLP ou le modèle d'intégrité BIBA. Nous proposons, dans cette section, de formaliser un ensemble de propriétés dont l'objectif est de garantir l'*intégrité* ou la *confidentialité* d'un système. Nous nous intéressons, plus particulièrement, aux propriétés du chapitre 2, et nous proposerons de nouvelles

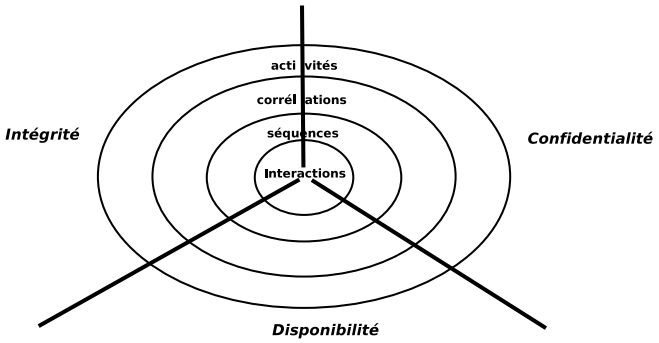


FIG. 3.5 – Classes d'activités et propriétés de sécurité

propriétés.

Ainsi, nous allons utiliser notre formalisation des activités, définie dans la section précédente, afin d'exprimer différentes propriétés de sécurité visant l'intégrité et la confidentialité. Tout d'abord, nous proposons une définition de la propriété d'*intégrité* et de *confidentialité* qui respecte les définitions faites dans [ITSEC 1991, TCSEC 1985, Bishop 2003] (cf. section 2.1). La formalisation de ces deux propriétés exploitent les relations de cause à effet observables sur un système. Nous nous limitons donc à ce qui est observable par un système d'exploitation, ce qui signifie qu'il existe toujours des violations possibles de ces propriétés basées notamment sur des échanges non observables par le système (par exemple, la communication verbale d'un mot de passe ou la transmission papier de documents confidentiels).

Par ailleurs, nous montrons que notre formalisation s'applique aussi pour d'autres modèles d'intégrité et de confidentialité. Pour l'intégrité, nous considérons la *non-interférence* (intégrité des sujets), la *base de confiance* (intégrité des domaines) et le modèle BIBA. Pour la confidentialité, nous traitons les modèles BLP et BLP restrictif. De plus, nous considérons une troisième classe de propriétés, connue sous le nom d'*abus de privilèges*, qui permet d'assurer, en fonction du type de privilège considéré, soit l'intégrité soit la confidentialité. Pour cette classe, nous traitons des propriétés classiques comme la *séparation de privilèges*, les *exécutables de confiance* et le *respect des règles de contrôle d'accès*.

Nous montrons enfin que notre formalisation permet soit d'étendre des propriétés existantes, soit de proposer de nouveaux modèles. Par exemple, pour la confidentialité, nous proposons une nouvelle propriété de *cohérence d'accès aux données* et pour l'abus de privilèges, nous proposons une nouvelle propriété d'*absence de changement de contexte*.

La définition de ces propriétés repose uniquement sur la formalisation des activités proposée dans la section précédente. Cela montre clairement que les différentes classes d'activités (interaction, séquence et corrélation) que nous avons formalisées, permettent non seulement de bien définir les propriétés classiques de la littérature, mais aussi, de les étendre et d'en proposer de nouvelles. Comme le montre la figure 3.5, nous proposons de catégoriser ces propriétés en termes de classes d'activités. Lors de la définition de chaque propriété, nous définissons ainsi des conditions liées à chaque classe d'activité. Ainsi, nous démontrons le pouvoir d'expressivité de notre langage de formalisation des activités.

La formalisation des propriétés de sécurité servira de base à l'analyse d'une politique de sécurité proposée dans le chapitre 4. Nous proposerons ainsi une méthode basée sur l'analyse de graphe, qui permet d'extraire les violations possibles de ces propriétés. Ces violations seront alors exprimées dans

notre langage de description d'activités.

Notons que pour formaliser nos propriétés, nous utilisons l'accolade ({) pour signifier un *et logique* entre plusieurs conditions. Par exemple, la condition $C_{3.3}$ prend en paramètres 2 arguments x, y. Cette condition est $vraie$ si : "Pour tout a appartenant à l'ensemble A, tels que les conditions $cond1$ et $cond2$ sont respectées, cela implique que la condition $cond3$ doit être respectée". Si cette dernière condition est fausse, la condition $C_{3.3}$ est fausse.

$$C_{3.3}(x,y) \text{ est vraie } \textbf{ssi} \quad \forall a \in A \textbf{ t.q. } \left\{ \begin{array}{l} cond1(x) \\ cond2(y) \end{array} \right. , \quad cond3(x,y) \tag{3.3}$$

3.3.1 Intégrité

La classe des propriétés d'*intégrité* a pour objectif de prévenir toute *modification* non autorisée d'une entité d'un système. Une entité peut correspondre à un objet (fichier, socket, ressource partagée, etc.) ou un sujet (processus, application, etc.). Dans notre représentation, une entité correspond donc à un contexte de sécurité, qui peut être du type sujet ou objet. Nous proposons de décliner cette classe en quatre propriétés, afin de garantir sur un système :

- *Intégrité des objets* : intégrité telle que définie dans [ITSEC 1991] ;
- *Biba* : intégrité selon le modèle de BIBA ;
- *Intégrité des sujets* : intégrité des sujets basée sur le modèle de non-interférence ;
- *Intégrité des domaines* : intégrité d'un ensemble de contextes.

L'intégrité des objets correspond à la définition générale de l'intégrité (cf. section 2.1). De plus, dans cette section, nous proposons d'étendre la définition générale d'intégrité et la définition de la non-interférence afin de prendre en compte les activités complexes correspondant à des séquences ou des combinaisons de séquences. Finalement, nous proposons de projeter le modèle de BIBA dans notre représentation.

3.3.1.1 Intégrité des objets

D'une manière générale, l'intégrité permet de prévenir un système contre toute modification illicite d'un objet. Dans la section 2.1.1, nous avons défini la propriété générale d'intégrité comme :

Définition 3.3.1 (Intégrité) *Soit X un ensemble d'entités et soit I de l'information ou une ressource. Alors la propriété d'intégrité de X envers I est respectée si aucun membre de X ne peut modifier I.*

Dans notre représentation, les ensembles X et I correspondent à des ensembles de contextes. Afin de projeter cette propriété dans notre modèle, nous proposons de formaliser la propriété d'intégrité d'un objet envers un sujet, puis nous définirons la propriété d'intégrité d'un système.

La propriété d'*intégrité d'un objet* se traduit par l'impossibilité, pour un sujet, de porter atteinte à l'intégrité d'un objet. En général, nous dirons qu'un contexte peut porter atteinte à un autre contexte, si il peut le modifier.

Dans notre représentation, un sujet scs_1 ne peut pas porter atteinte à l'*intégrité d'un objet* sco_1 si cette entité ne possède pas directement le privilège de modification sur cet objet, ce que nous appelons la condition $C_{3.4}(scs_1, sco_1)$.

$$C_{3.4}(scs_1, sco_1) \text{ est vraie } \textbf{ssi} \quad \forall eo \in IS \textbf{ t.q. } scs_1 \xrightarrow{eo} sco_1, \quad \neg is\_write\_like(eo) \tag{3.4}$$

De plus, nous proposons d'étendre cette propriété afin de prendre en compte les *accès à un privilège*. Ainsi, un sujet scs_1 ne peut pas porter atteinte à l'intégrité d'un objet sco_1 si cette entité ne peut pas obtenir le privilège de modification, ce que nous appelons la condition $C_{3.5}(scs_1, sco_1)$.

$$C_{3.5}(scs_1, sco_1) \text{ est vraie } \textbf{ssi} \quad \forall eo \in IS \textbf{ t.q. } scs_1 \Rightarrow_{eo} sco_1, \quad \neg is\_write\_like(eo) \qquad (3.5)$$

La propriété d'*intégrité d'un objet* vis-à-vis d'un sujet est respectée si les conditions (3.4) et (3.5) ne sont pas violées.

Propriété 3.3.1 *Un **objet** $sco_1 \in SC_O$ est dit **intègre** vis-à-vis d'un sujet $scs_1 \in SC_S$ si et seulement si les conditions* (3.4) *et* (3.5) *sont respectées.*

$$P_{3.3.1}(scs_1, sco_1) \textit{ est vraie } \textbf{ssi} \quad C_{3.4}(scs_1, sco_1) \wedge C_{3.5}(scs_1, sco_1)$$

La condition (3.4) est violée si il existe une interaction contenant une opération élémentaire eo permettant à un sujet scs_1 de modifier l'objet sco_1. La condition (3.5) est violée si il existe un *accès au privilège* de modification.
A partir de cette propriété, nous pouvons définir la propriété d'*intégrité d'un système* :

Propriété 3.3.2 *Soit $S = \{(scs_1, sco_1), \ldots, (scs_n, sco_n)\}$ un ensemble de couples de contextes. Un **système** est dit **intègre** si et seulement si pour tout couple (scs_i, sco_i), sco_i est intègre vis-à-vis de scs_i.*

$$P_{3.3.2}(S) \textit{ est vraie } \textbf{ssi} \quad \forall i = 1..n, P_{3.3.1}(scs_i, sco_i)$$

Cette propriété est violée si un des couples (scs_i, sco_i) ne respecte pas une des conditions (3.4) et (3.5).

Notons que la condition (3.5) concerne les *accès au privilège* au sens général. Étant donné que cette classe d'activité surestime les possibilités d'accès à un privilège, cette condition peut être restreinte au cas des *accès par transition* ($scs_i \Rightarrow_{trans\_eo} sco_i$). Dans ce cas, seuls les *accès au privilège* composant une séquence de transition et une interaction du type modification seront pris en compte.

3.3.1.2 Biba

Le modèle BIBA (cf. section 2.3.2.2) a pour objectif de garantir l'intégrité d'un système via le respect de trois règles de contrôle des opérations. Dans cette section, nous proposons une projection du modèle d'intégrité dit BIBA dans notre représentation. Rappelons que ce modèle associe à chaque entité du système un niveau d'intégrité. Trois règles de contrôle régissent ainsi les accès sur le système :

biba-1 Pour qu'un sujet scs_1 ait accès en **lecture** à un objet sco_1, son niveau d'intégrité doit être **inférieur ou égal** au niveau d'intégrité de l'objet ;

biba-2 Pour qu'un sujet scs_1 ait accès en **écriture** à un objet sco_1, son niveau d'intégrité doit être **supérieur ou égal** au niveau d'intégrité de l'objet ;

biba-3 Pour qu'un sujet scs_1 puisse **invoquer** un sujet scs_2, son niveau d'intégrité doit être **supérieur ou égal** au niveau d'intégrité du sujet invoqué.

Il y aura atteinte à l'intégrité du système si l'une de ces trois règles est violée.

Dans la définition de ces règles, la notion de sujet ou d'objet correspond à notre notion de contexte de sécurité, le niveau d'intégrité étant alors un attribut de ce contexte. Nous définissons ainsi la fonction $\texttt{int} : SC \rightarrow \mathbb{N}$ qui associe à un contexte son niveau d'intégrité et nous proposons une formalisation des trois conditions de la *propriété d'intégrité de Biba* dans notre représentation.

La première condition, $C_{3.6}(scs_1, sco_1)$ (condition **biba-1**), implique que pour toute interaction du type *lecture* entre un contexte sujet scs_1 et un objet sco_1, le niveau d'intégrité du sujet est inférieur ou égal au niveau d'intégrité de l'objet.

$$C_{3.6}(scs_1, sco_1) \text{ est vraie } \textbf{ssi } \forall eo \in IS \textbf{ t.q. } \left\{ \begin{array}{l} scs_1 \xrightarrow{eo} sco_1 \\ is\_read\_like(eo) \end{array} \right. , int(scs_1) \leqslant int(sco_1) \quad (3.6)$$

La seconde condition, $C_{3.7}(scs_1, sco_1)$ (condition **biba-2**), implique que pour toute interaction du type *modification* entre un contexte sujet scs_1 et un objet sco_1 , le niveau d'intégrité du sujet est supérieur ou égal au niveau d'intégrité de l'objet :

$$C_{3.7}(scs_1, sco_1) \text{ est vraie } \textbf{ssi } \forall eo \in IS \textbf{ t.q. } \left\{ \begin{array}{l} scs_1 \xrightarrow{eo} sco_1 \\ is\_write\_like(eo) \end{array} \right. , int(scs_1) \geqslant int(sco_1) \quad (3.7)$$

Finalement, la troisième condition, $C_{3.8}(scs_1, sco_1)$ (condition **biba-3**), implique que pour toute *invocation* d'un contexte sujet scs_2 par un sujet scs_1, le niveau d'intégrité du sujet scs_1 est supérieur ou égal au niveau d'intégrité du sujet invoqué.

$$C_{3.8}(scs_1, scs_2) \text{ est vraie } \textbf{ssi } \forall eo \in IS \textbf{ t.q. } \left\{ \begin{array}{l} scs_1 \xrightarrow{eo} scs_2 \\ is\_execute\_like(eo) \end{array} \right. , int(scs_1) \geqslant int(scs_2) \quad (3.8)$$

La propriété d'intégrité de BIBA est donc respectée si les conditions (3.6), (3.7) et (3.8) ne sont pas violées. Nous définissons ainsi la propriété d'intégrité de BIBA :

Propriété 3.3.3 *Un **système**, composé d'un ensemble SC de contextes, est dit intègre selon Biba si et seulement si les conditions* (3.6)*,* (3.7) *et* (3.8) *sont respectées.*

$$P_{3.3.3}(SC) \textit{ est vraie } \textbf{\textit{ssi}} \ \forall sc_1, sc_2 \in SC, C_{3.6}(sc_1, sc_2) \wedge C_{3.7}(sc_1, sc_2) \wedge C_{3.8}(sc_1, sc_2)$$

Notons finalement que dans notre modèle, un contexte sujet peut effectuer une opération de lecture/modification sur un contexte objet (fichier, socket, etc.) ou sur un autre sujet (envoi/réception de signal, etc.). Nous généralisons donc les trois conditions définissant le modèle *intégrité selon Biba* en considérant toute interaction entre deux contextes quelconques (sujet ou objet).

3.3.1.3 Intégrité des sujets/Non-interférence

La propriété d'*intégrité des sujets* a pour objectif de garantir l'intégrité d'exécution des contextes sujets (i.e. un processus) sur un système. Cette propriété est une adaptation de la propriété de non-interférence (cf. section 2.1.9, page 27) dans notre représentation.

La propriété de non-interférence est basée sur la notion de commutativité des opérations. D'une manière générale, il peut y avoir interférence entre deux sujets sc_1 et sc_2 si ces deux contextes "utilisent" un objet commun sc_3 et que les actions effectuées sur sc_3 par sc_1 et sc_2 ne commutent pas. Soit $it_1 = (scs_1, sco_1, eo_1)$ et $it_2 = (scs_2, sco_1, eo_2)$ deux interactions, l'opération eo_1 commute avec l'opération eo_2 si elle ne modifie pas la vision qu'a scs_2 de sco_1, c'est-à-dire si eo_1 ne modifie pas l'état du contexte partagé sco_1. Par exemple, les opérations $eo_1 = \{file : write\}$ et

$eo_2 = \{file : read\}$ ne commutent pas car eo_1 peut modifier les données lues par eo_2, alors que les opérations $eo_1 = \{file : read\}$ et $eo_2 = \{file : read\}$ commutent car une lecture ne modifie pas l'état de sco_1.

Nous définissons ainsi la fonction `commute` $: IS \times IS \rightarrow \{vrai, faux\}$ qui associe à deux opérations élémentaires la valeur `vrai` si ces deux opérations commutent, `faux` sinon. Cette fonction utilise une table de commutativité, définie dans [Ko et Redmond 2002] (cf. partie 2.2.3.2, page 37), et adaptée aux opérations de notre représentation.

Dans notre représentation, un sujet sc_1 ne peut pas porter atteinte à l'*intégrité d'un sujet* sc_2 si ces deux entités ne peuvent pas effectuer directement des opérations qui ne commutent pas. Nous définissons ainsi la condition $C_{3.9}(sc_1, sc_2)$.

$$C_{3.9}(sc_1, sc_2) \text{ est vraie } \textbf{ssi} \begin{cases} \forall sco_1 \in SC_O \\ \forall eo_1, eo_2 \in IS \end{cases} \textbf{t.q.} \begin{cases} sc_1 \xrightarrow{eo_1} sco_1 \\ sc_2 \xrightarrow{eo_2} sco_1 \end{cases}, commute(eo_1, eo_2) \quad (3.9)$$

Cette condition est violée si il existe deux interactions accédant à un même objet dont les opérations élémentaires ne commutent pas.

De même, un sujet sc_1 ne peut pas porter atteinte à l'*intégrité d'un sujet* sc_2 si ces deux entités ne peuvent pas effectuer indirectement des opérations qui ne commutent pas. Nous définissons ainsi la condition $C_{3.10}(sc_1, sc_2)$.

$$C_{3.10}(sc_1, sc_2) \text{ est vraie } \textbf{ssi} \begin{cases} \forall sco_1 \in SC_O \\ \forall eo_1, eo_2 \in IS \end{cases} \textbf{t.q.} \begin{cases} sc_1 \Rightarrow_{eo_1} sco_1 \\ sc_2 \Rightarrow_{eo_2} sco_1 \end{cases}, commute(eo_1, eo_2) \quad (3.10)$$

Cette condition est violée si il existe deux *accès à des privilèges*, sur un même objet, qui ne commutent pas.

Ces deux conditions permettent de définir la propriété d'*intégrité d'un sujet* sc_1 vis-à-vis d'un sujet sc_2 et la *propriété d'intégrité des sujets* d'un système.

Propriété 3.3.4 *L'exécution d'un* ***sujet*** sc_1 *est dit* ***intègre*** *vis-à-vis d'un sujet* sc_2 *si* $P_{3.3.4}(sc_1, sc_2)$ *est vraie.*

$$P_{3.3.4}(sc_1, sc_2) \text{ est vraie } \textbf{ssi } C_{3.9}(sc_1, sc_2) \wedge C_{3.10}(sc_1, sc_2)$$

Propriété 3.3.5 *Les* ***sujets*** SC_S *d'un* ***système*** *sont* ***intègres*** *si et seulement si chaque couple de sujet respecte la propriété 3.3.4.*

$$P_{3.3.5}(SC_S) \text{ est vraie } \textbf{ssi } \forall sc_1, sc_2 \in SC_S \textbf{ t.q. } sc_1 \neq sc_2, P_{3.3.4}(sc_1, sc_2)$$

Notons que sur un système, cette propriété n'est pas réellement applicable. En effet, certains ensembles de contextes partagent délibérément des ressources et utilisent sur ces ressources des opérations qui ne commutent pas. Ce qui peut alors être considéré comme une violation de la propriété de non-interférence. C'est pourquoi nous définissons une dernière propriété de non-interférence fidèle à la définition originale de [Ko et Redmond 2002].

Cette propriété, moins générale que la *propriété d'intégrité des sujets* d'un système, se limite à l'intégrité processus privilégiés (ayant les droits administrateur) et au cas d'interférence entre ces processus et les processus dit non-privilégiés (utilisateurs).

Propriété 3.3.6 *Soit* $P = \{sc_{p_1}, \ldots, sc_{p_n}\}$ *un ensemble de sujets privilégiés et* $NP = \{sc_{np_1}, \ldots, sc_{np_m}\}$ *un ensemble de sujets non privilégiés L'exécution des* ***sujets privilégiés*** *est dit* ***intègre*** *si chaque sujet privilégié est intègre vis-à-vis d'un sujet non privilégié.*

$$P_{3.3.6}(P, NP) \text{ est vraie } \textbf{ssi } \forall sc_p \in P, \forall sc_{np} \in NP, P_{3.3.4}(sc_p, sc_{np})$$

3.3.1.4 Intégrité des domaines

La notion d'*intégrité des domaines* se réfère à la notion de *noyau de sécurité* ou *base de confiance* (en anglais TCB pour *Trusted Computing Base*). Nous définissons par TCB un ensemble de contextes de sécurité TCB_{in} qui représente un *domaine* à surveiller. Par définition, toute interaction entre un contexte de cet ensemble et un contexte n'appartenant pas à cet ensemble représente une violation de l'intégrité de ce domaine. De par cette définition, un TCB peut être comparé à un *chroot virtuel* où les interactions entre les éléments de ce *chroot* et le reste du système sont impossibles. Nous définissons ainsi la propriété d'*intégrité d'un domaine*.

Propriété 3.3.7 *Soit $TCB_{in} \subset SC$ un ensemble de contextes définissant un domaine.*
*Un **domaine** est dit **intègre** si aucune interaction n'existe entre un contexte de ce domaine et le reste du système.*

$$P_{3.3.7}(TCB_{in}) \text{ est vraie } \textbf{ssi} \begin{cases} \forall eo \in IS \\ \forall sc_1, sc_2 \in SC \end{cases}, \textbf{t.q. } sc_1 \xrightarrow{eo} sc_2, \begin{cases} sc_1 \in TCB_{in} \Longrightarrow sc_2 \in TCB_{in} \\ \vee \\ sc_2 \in TCB_{in} \Longrightarrow sc_1 \in TCB_{in} \end{cases}$$

Étant donné un ensemble de contextes constituant un *domaine*, cette propriété est donc respectée si aucune interaction n'est possible entre un contexte du domaine et un autre contexte extérieur à ce domaine.
En fonction de cette propriété, nous définissons la *propriété d'intégrité des domaines* d'un système.

Propriété 3.3.8 *Soit TCB_{sys} un ensemble de domaines, les **domaines** d'un **système** sont **intègres** si et seulement si chaque domaine $d \in TCB_{sys}$ respecte la propriété 3.3.7.*

$$P_{3.3.8}(TCB_{sys}) \text{ est vraie } \textbf{ssi } \forall d \in TCB_{sys}, P_{3.3.7}(d)$$

Notons que sur un système réel, afin d'initialiser les services qui peuvent appartenir à un TCB, il est nécessaire d'avoir des interactions entre les contextes système et les contextes de ce TCB. Ainsi, l'apparition de ces interactions, lors de l'initialisation du système, représente des faux positifs, c'est-à-dire de fausses violations de la propriété 3.3.7. C'est pourquoi le système de détection d'intrusion, que nous proposons dans le chapitre 5, ne peut considérer ces violations qu'après l'initialisation du système.

3.3.2 Confidentialité

La classe des propriétés de confidentialité a pour objectif de prévenir un système contre tout accès d'une entité à une information non autorisée. Nous proposons de décliner cette classe en quatre propriétés dans le but de garantir :

- *Confidentialité des contextes* : confidentialité de contextes du système ;
- *Bell&LaPadula* confidentialité : selon le modèle de *Bell&LaPadula* ;
- *Bell&LaPadula restrictive* : confidentialité selon la version restrictive du modèle de *Bell&LaPadula* ;
- *Cohérence d'accès aux données* : cohérence entre les informations accessibles directement et par corrélation.

La confidentialité des contextes correspond à la définition générale de la confidentialité (cf. section 2.1). De plus, dans cette section, nous proposons d'étendre la définition générale de la confidentialité afin de prendre en compte les activités complexes correspondant à des séquences ou des

combinaisons de séquences. Nous proposons aussi de projeter les modèles BLP et BLP restrictif dans notre représentation. Nous proposons finalement une nouvelle propriété permettant de garantir la cohérence des accès aux données.

3.3.2.1 Confidentialité des contextes

D'une manière générale, la confidentialité permet de prévenir un système contre tout accès illicite à un objet. Dans la section 2.1.1, nous avons défini la propriété générale de confidentialité comme :

Définition 3.3.2 (Confidentialité) *Soit I de l'information et soit X un ensemble d'entités non autorisées à accéder à I. La propriété de confidentialité de X envers I est respectée si aucun membre de X ne peut obtenir de l'information de I.*

Dans notre représentation, les ensembles X et I correspondent à des ensembles de contextes. Afin de projeter cette propriété dans notre modèle, nous proposons de formaliser la propriété de confidentialité d'un contexte envers un autre contexte, puis nous définirons la propriété de confidentialité d'un système.

Dans un système, une entité peut obtenir de l'information d'une autre entité si elle peut directement accéder à l'information contenue dans cette entité, ou si il existe un flux d'informations valide entre ces deux entités. La propriété de confidentialité a pour objectif de proscire certains flux d'informations sur un système. Par exemple, les flux entre le domaine utilisateur et les fichiers système ou entre deux types d'utilisateurs différents d'une entreprise (service comptable et service administratif) peuvent être considérés comme illégaux. La propriété de *confidentialité d'un contexte* se traduit alors par l'impossibilité, pour un contexte, d'obtenir de l'information depuis un autre contexte.

Une entité sc_1 ne peut porter atteinte à la propriété de *confidentialité du contexte* sc_2 si cette entité ne peut obtenir directement de l'information, ce que nous appelons la condition $C_{3.11}(sc_1, sc_2)$.

$$C_{3.11}(sc_1, sc_2) \text{ est vraie } \textbf{ssi } \neg sc_2 > sc_1 \tag{3.11}$$

De même une entité sc_1 ne peut obtenir de l'information depuis une entité sc_2 si toute séquence d'information n'induit pas de flux d'informations, ce que nous appelons la condition $C_{3.12}(sc_1, sc_2)$.

$$C_{3.12}(sc_1, sc_2) \text{ est vraie } \textbf{ssi } \neg scs_2 \gg sc_1 \tag{3.12}$$

De plus, nous proposons d'étendre cette propriété afin de prendre en compte les *accès à l'information*. Finalement, une entité sc_1 ne peut obtenir de l'information d'une entité sc_2 si toute corrélation n'induit pas un accès à l'information, ce que nous appelons la condition $C_{3.13}(sc_1, sc_2)$.

$$C_{3.13}(sc_1, sc_2) \text{ est vraie } \textbf{ssi } \neg sc_1 \Rrightarrow sc_2 \tag{3.13}$$

La propriété de confidentialité d'un contexte sc_2 envers un contexte sc_1 est respectée si les conditions (3.11), (3.12) et (3.13) sont respectées.

Propriété 3.3.9 *Un **contexte** sc_1 ne peut obtenir d'information depuis un contexte sc_2 si et seulement si les conditions* (3.11)*,* (3.12) *et* (3.13) *sont respectées.*

$$P_{3.3.9}(sc_1, sc_2) \textit{ est vraie } \textbf{\textit{ssi}} \quad C_{3.11}(sc_1, sc_2) \wedge C_{3.12}(sc_1, sc_2) \wedge C_{3.13}(sc_1, sc_2)$$

Cette propriété est donc violée si il existe un transfert d'informations, un flux d'informations ou une corrélation d'accès à l'information entre sc_1 et sc_2.

A partir de cette propriété, nous pouvons définir la propriété de *confidentialité d'un système* :

Propriété 3.3.10 *Soit $S = \{(scs_1, scd_1), \ldots, (scs_n, scd_n)\}$ un ensemble de couples de contextes. La propriété de **confidentialité** de ce **système** est assurée si et seulement si, pour tout couple (scs_i, scd_i), scs_i ne peut obtenir d'information de scd_i.*

$$P_{3.3.10}(S) \text{ est vraie } \textbf{ssi } \forall i = 1..n, P_{3.3.9}(scs_i, scd_i)$$

Cette propriété est violée si un des couples (scs_i, scd_i) ne respecte pas une des conditions (3.11), (3.12) ou (3.13).

Notons que la condition (3.13) concerne les corrélations d'*accès à l'information* au sens général. Étant donné que ces corrélations surestiment les possibilités d'accès à l'information, cette condition peut être restreinte au cas des corrélations d'*accès à l'information par transition*. Dans ce cas, seules les corrélations d'une séquence de transitions et d'un flux d'informations sont prises en compte.

3.3.2.2 Bell&LaPadula

Le modèle de confidentialité dit BLP (cf. section 2.3.2.1, page 44) a pour objectif de garantir la confidentialité d'un système via le respect de deux règles de contrôle des opérations. Ainsi nous notons BLP le modèle de confidentialité défini par Bell&LaPadula. Dans cette partie, nous proposons une projection de ce modèle dans notre représentation. Rappelons que ce modèle associe à chaque entité un niveau d'intégrité correspondant à un *niveau d'habilitation* pour les sujets ou à un *niveau de sensibilité* pour les sujets. Deux règles de contrôle régissent ainsi les accès sur le système :

blp-1 Lorsqu'un sujet scs_1 demande un accès en **lecture** sur un objet sco_1, son *niveau d'habilitation* doit être **supérieur ou égal** à celui de l'objet. Cette règle assure la confidentialité de l'information ;

blp-2 Lorsqu'un **transfert d'informations** est effectué d'un objet sco_1 vers un objet sco_2, le niveau d'habilitation du contexte cible doit être **supérieur ou égal** au niveau d'habilitation du contexte source. Seuls les transferts d'informations depuis des objets de classification inférieure vers des objets de classification supérieure sont autorisés. Cette règle assure donc la prévention contre la divulgation d'informations.

Il y aura atteinte à la confidentialité du système si l'une de ces règles est violée. Dans notre représentation, un sujet ou un objet correspond à un contexte de sécurité, son niveau d'habilitation étant alors un attribut de ce contexte. Nous définissons ainsi la fonction $\texttt{hab} : SC \rightarrow \mathbb{N}$ qui associe à un contexte son niveau d'habilitation et nous proposons une formalisation des deux règles de la *propriété de confidentialité de* BLP dans notre représentation.

La première condition, $C_{3.14}(scs_1, sco_1)$ (condition **blp-1**), implique que pour toute interaction du type *lecture* entre un contexte sujet scs_1 et un objet sco_1, le niveau d'habilitation du sujet est supérieur ou égal au niveau d'habilitation de l'objet

$$C_{3.14}(scs_1, sco_1) \text{ est vraie } \textbf{ssi } \forall eo \in IS \textbf{ t.q. } \left\{ \begin{array}{l} scs_1 \xrightarrow{eo} sco_1 \\ is\_read\_like(eo) \end{array} \right. , hab(scs_1) \geqslant hab(sco_1) \quad (3.14)$$

La seconde condition, $C_{3.15}(sco_1, sco_2)$ (condition **blp-2**), implique que pour tout flux d'informations $sco_1 \gg sco_2$ entre deux objets sco_1 et sco_2, le niveau d'habilitation de la source du flux $hab(sco_1)$

est supérieur ou égal au niveau d'habilitation de la cible $hab(sco_2)$.

$$C_{3.15}(sco_1, sco_2) \text{ est vraie } \textbf{ssi} \quad \forall sco_2 \gg sco_1, \quad hab(sco_1) \geqslant hab(sco_2) \tag{3.15}$$

La propriété de confidentialité de BLP est respectée si les conditions (3.14) et (3.15) ne sont pas violées. Nous définissons ainsi la propriété de confidentialité d'un système selon BLP.

Propriété 3.3.11 *Un **système**, composé d'un ensemble de contextes $SC = SC_S \cup SC_O$, est dit **confidentiel** selon* BLP *si et seulement si les conditions* (3.14) *et* (3.15) *sont respectées.*

$$P_{3.3.11}(SC) \textit{ est vraie } \textbf{ssi} \begin{cases} \forall scs_1 \in SC_S \\ \forall sco_1, sco_2 \in SC_O \end{cases}, C_{3.14}(scs_1, sco_1) \wedge C_{3.15}(sco_1, sco_2)$$

3.3.2.3 Bell&LaPadula restrictif

Le modèle de confidentialité dit BLP (cf. section 2.3.2.1, page 44) a pour objectif de garantir la confidentialité d'un système via le respect de trois règles "simples" de contrôle des opérations. Ainsi nous notons BLPR la version restrictive du modèle de confidentialité défini par Bell&LaPadula. Dans cette partie, nous proposons une projection de ce modèle dans notre représentation. Les trois règles de contrôle d'accès de la version restrictive de ce modèle correspondent dans notre modèle à :

blpr-1 Pour qu'un sujet scs_1 ait accès en **lecture** à un objet sco_1, son niveau d'habilitation $hab(scs_1)$ doit être **supérieur ou égal** au niveau d'habilitation $hab(sco_1)$ de l'objet ;

blpr-2 Pour qu'un sujet scs_1 ait accès en **ajout** (modification sans lecture) à un objet sco_1, son niveau d'habilitation doit être **inférieur ou égal** au niveau d'habilitation de l'objet ;

blpr-3 Pour qu'un sujet scs_1 ait accès en **modification** à un objet sco_1, son niveau d'habilitation doit être **égal** au niveau d'habilitation de l'objet ;

Il peut y avoir atteinte à la confidentialité du système si l'une de ces trois règles est violée.

La première condition, $C_{3.16}(scs_1, sco_1)$ (condition **blpr-1**), implique que pour toute interaction de type *lecture* entre un sujet et un objet, le niveau d'habilitation du sujet ($hab(scs_1)$) est supérieur ou égal à celui de l'objet ($hab(sco_1)$).

$$C_{3.16}(scs_1, sco_1) \text{ est vraie } \textbf{ssi } \forall eo \in IS \textbf{ t.q. } \begin{cases} scs_1 \xrightarrow{eo} sco_1 \\ is\_read\_like(eo) \end{cases}, hab(scs_1) \geqslant hab(sco_1) \tag{3.16}$$

La seconde condition, $C_{3.17}(scs_1, sco_1)$ (condition **blpr-2**), implique que pour toute interaction d'ajout entre un sujet et un objet, le niveau d'habilitation du sujet ($hab(scs_1)$) est inférieur ou égal à celui de l'objet ($hab(sco_1)$).

$$C_{3.17}(scs_1, sco_1) \text{ est vraie } \textbf{ssi } \forall eo \in IS \textbf{ t.q. } \begin{cases} scs_1 \xrightarrow{eo} sco_1 \\ is\_add\_like(eo) \end{cases}, hab(scs_1) \leqslant hab(sco_2) \tag{3.17}$$

Finalement, la troisième condition, $C_{3.18}(scs_1, sco_1)$ (condition **blpr-3**), implique que pour toute interaction de type modification entre un sujet et un objet, le niveau d'habilitation du sujet $hab(scs_1)$ est égal de celui de l'objet $hab(sco_1)$.

$$C_{3.18}(scs_1, sco_1) \text{ est vraie } \textbf{ssi } \forall eo \in IS \textbf{ t.q. } \begin{cases} scs_1 \xrightarrow{eo} sco_1 \\ is\_write\_like(eo) \end{cases}, hab(scs_1) = hab(sco_1) \tag{3.18}$$

Un système respecte la propriété de confidentialité BLP-restrictive si aucune interaction ne viole l'une de ces trois conditions. Nous définissons ainsi la propriété de confidentialité d'un système selon BLP-restrictive.

Propriété 3.3.12 *Un **système** est dit **confidentiel** selon* BLP*-**restrictive** si et seulement si les conditions* (3.16)*,* (3.17) *ou* (3.18) *sont respectées.*

$$P_{3.3.12}(SC) \text{ est vraie ssi } \begin{cases} \forall sc_1 \in SC_S \\ \forall sc_2 \in SC_O \end{cases}, C_{3.16}(sc_1, sc_2) \wedge C_{3.17}(sc_1, sc_2) \wedge C_{3.18}(sc_1, sc_2)$$

3.3.2.4 Cohérence d'accès aux données

Un contexte peut, par combinaison, accéder à des données auxquelles il n'a pas directement accès (via une interaction). La propriété de cohérence d'accès aux données vise à garantir qu'une donnée non accessible directement par un contexte ne l'est pas par combinaison de séquences. Si les données accessibles par combinaison sont les mêmes que celles accessibles directement, nous dirons donc qu'il y a *cohérence d'accès aux données*. Nous définissons donc la propriété de *cohérence d'accès à une donnée*.

Propriété 3.3.13 *La propriété de cohérence d'accès à une donnée sc_2 par sc_1 est respectée si pour toute corrélation d'accès à l'information entre sc_1 et sc_2, il existe un transfert d'informations de sc_2 vers sc_1.*

$$P_{3.3.13}(sc_1, sc_2) \text{ est vraie ssi } sc_1 \Rrightarrow sc_2 \text{ implique } sc_2 > sc_1$$

Tout comme précédemment, cette propriété peut être restreinte au cas des *accès à l'information par transition*. A partir de cette propriété, nous définissons la propriété de *cohérence d'accès aux données d'un système* :

Propriété 3.3.14 *Sur un **système**, composé d'un ensemble de contextes $SC = SC_S \cup SC_O$, la propriété de **cohérence d'accès aux données** est respectée si et seulement si :*

$$P_{3.3.14}(SC) \text{ est vraie ssi } \forall sc_1, sc_2 \in SC, P_{3.3.13}(sc_1, sc_2)$$

Une politique de sécurité respecte cette propriété si pour toute corrélation d'accès à l'information, il existe un transfert d'informations équivalent.

Pour illustrer cette propriété, prenons l'exemple de deux contextes sujets scs_1 et scs_2, et deux contextes objets sco_1 et sco_2. Supposons que scs_1 a accès en lecture à sco_1 et que scs_2 a accès en lecture à sco_1 et sco_2. Supposons que le contexte scs_1 puisse transiter vers un contexte scs_2. Si scs_1 transite vers scs_2, il peut avoir accès aux deux contextes sco_1 et sco_2, or il n'avait normalement accès qu'à sco_1. Il peut donc, par transition, obtenir de l'information depuis sco_2 alors qu'il n'était pas supposé avoir accès à cette information directement. Ce type de corrélation peut aboutir à une violation de la confidentialité d'un système.

3.3.3 Abus de privilèges

La classe des propriétés d'abus de privilèges a pour objectif de prévenir un système contre toute action ayant pour objectif d'abuser des privilèges disponibles. Dans notre représentation, nous proposons quatre propriétés permettant de garantir :

- *Séparation de privilèges* : une séparation entre certains privilèges tels que les privilèges de modification et d'exécution ;
- *Absence de changement de contexte* : l'impossibilité d'obtenir les privilèges d'un autre contexte par transition ;

- *Exécutables de confiance* l'impossibilité : de l'exécution d'application considérée comme non sûre ;
- *Respect d'une politique de contrôle d'accès* : le respect de l'application d'une politique de contrôle d'accès ;
- *Respect d'une Méta-Politique de contrôle d'accès* : le respect de l'application d'une Méta-Politique de contrôle d'accès.

Notons que chaque propriété d'abus de privilèges peut aussi être classée dans une des deux classes précédentes : *intégrité* ou *confidentialité*. Nous préférons utiliser le terme d'*abus de privilèges* pour ces propriétés car l'exploitation de ces propriétés est généralement liée à un abus des privilèges disponibles. Nous proposons d'étendre la définition de la séparation de privilèges afin de prendre en compte les activités complexes correspondant à des séquences ou des combinaisons de séquences. Nous proposons aussi une nouvelle propriété permettant de garantir l'absence de changement de contexte et de respect d'une Méta-Politique.

3.3.3.1 Séparation de privilèges

La propriété de séparation de privilèges, définie dans [Clark et Wilson 1987a] (page 187), implique que : "*Une règle élémentaire de séparation de privilèges peut être que toute personne ayant le droit de créer ou modifier des objets n'ont pas le droit de les exécuter*". Une seconde définition donnée dans [Sandhu 1990] spécifie que : "*La séparation de privilèges implique que les opérations réalisées sur un même objet doivent être effectuées par des utilisateurs différents*". La propriété de *séparation de privilèges* peut être considérée comme une propriété d'intégrité ou de confidentialité suivant le type des opérations qui doivent être séparées.

Un exemple *simple* de séparation de privilèges implique donc qu'un contexte qui a eu accès en *écriture* à une donnée, ne puisse pas ensuite l'*exécuter*. En effet, certaines attaques visant les services du système exploitent la possibilité de créer un fichier, par exemple un fichier contenant un script ouvrant un terminal distant, pour ensuite faire exécuter ce script par un service système.

Une première condition, $C_{3.19}(scs_1, sco_1)$, implique qu'un sujet scs_1 ne peut pas modifier puis exécuter un objet sco_1 si il n'existe pas de composition d'interactions permettant la modification puis l'exécution d'un contexte.

$$C_{3.19}(scs_1, sco_1) \text{ est vraie } \textbf{ssi } \forall eo_1, eo_2 \in IS \textbf{ t.q. } \begin{cases} it_1 = scs_1 \xrightarrow{eo_1} sco_1 \\ it_2 = scs_1 \xrightarrow{eo_2} sco_1 \\ is\_write\_like(eo_1) \\ (it_2 \circ it_1) \end{cases}, \neg is\_execute\_like(eo_2) \quad (3.19)$$

De plus, nous proposons d'étendre cette propriété afin de prendre en compte les *accès à un privilège*. Une seconde condition, $C_{3.20}(scs_1, sco_1)$, implique qu'un sujet scs_1 ne peut pas modifier puis exécuter un objet sco_1 si il n'existe pas d'*accès à un privilège* permettant la modification puis l'exécution d'un contexte :

$$C_{3.20}(scs_1, sco_1) \text{ est vraie } \textbf{ssi } \forall eo_1, eo_2 \in IS \textbf{ t.q. } \begin{cases} ac_1 = scs_1 \Rightarrow_{eo_1} sco_1 \\ ac_2 = scs_1 \Rightarrow_{eo_2} sco_1 \\ is\_write\_like(eo_1) \\ (ac_2 \circ ac_1) \end{cases}, \neg is\_execute\_like(eo_2) \quad (3.20)$$

Basé sur ces deux conditions, nous définissons la propriété de *séparation des privilèges d'un contexte*.

Propriété 3.3.15 *Un contexte scs_1 respecte la propriété de séparation de privilèges sur un objet sco_1 si et seulement si les conditions* (3.19) *et* (3.20) *sont respectées.*

$$P_{3.3.15}(scs_1, sco_1) \text{ est vraie } \textbf{ssi} \quad C_{3.19}(scs_1, sco_1) \wedge C_{3.20}(scs_1, sco_1)$$

Finalement, nous pouvons définir la propriété *de séparation des privilèges sur un système* comme.

Propriété 3.3.16 *Un* ***système****, composé d'un ensemble de contextes $SC = SC_S \cup SC_O$, respecte la propriété* ***de séparation des privilèges*** *si $P_{3.3.15}$ est vraie.*

$$P_{3.3.16}(SC) \text{ est vraie } \textbf{ssi} \begin{cases} \forall sc_s \in SC_S \\ \forall sc_o \in SC_O \end{cases}, \quad P_{3.3.15}(sc_s, sc_o)$$

Notons que la propriété de séparation de privilèges *simple* (3.3.15) ne vérifie que la présence des privilèges de modification et d'exécution. Cette propriété peut être généralisée pour vérifier que n'importe quel couple ou ensemble d'opérations élémentaires ne peut être utilisé séquentiellement.

Propriété 3.3.17 *Soit un ensemble de couples de contextes et $OP_1, OP_2 \subset IS$ deux ensembles d'opérations élémentaires. Un* ***système****, composé d'un ensemble de contextes $SC = SC_S \cup SC_O$, respecte la propriété* ***de séparation des privilèges entre*** *OP_1* ***et*** *OP_2 si $P_{3.3.17}$ est vraie.*

$$P_{3.3.17}(S, OP_1, OP_2) \text{ est vraie } \textbf{ssi} \begin{cases} \forall sc_s \in SC_S \\ \forall sc_o \in SC_O \\ \forall eo_1, eo_2 \in IS \end{cases} \textbf{t.q.} \quad \wedge \begin{cases} it_1 = sc_s \xrightarrow{eo_1} sc_o \\ it_2 = sc_s \xrightarrow{eo_2} sc_o \\ (it_2 \circ it_1) \\ eo_1 \in OP_1 \end{cases}, eo_2 \notin OP_2$$

$$\wedge \begin{cases} act_1 = sc_s \Rightarrow_{eo_1} sc_o \\ act_2 = sc_s \Rightarrow_{eo_2} sc_o \\ (ac_2 \circ ac_1) \\ eo_1 \in OP_1 \end{cases}, eo_2 \notin OP_2$$

Cette propriété implique que 1) pour toute composition de deux interactions $(it_2 \circ it_1)$, si l'opération de it_1 appartient à l'ensemble OP_1 alors l'opération de it_2 ne doit pas appartenir à l'ensemble OP_2, 2) pour toute composition de deux séquences d'accès à un privilège $(ac_2 \circ ac_1)$, si l'opération de ac_1 appartient à l'ensemble OP_1 alors l'opération de ac_2 ne doit pas appartenir à l'ensemble OP_2.

3.3.3.2 Absence de changement de contexte

Nous proposons de définir une nouvelle propriété permettant de garantir l'*absence de changement de contexte*. Cette propriété a pour objectif d'empêcher un contexte d'obtenir, par transition, les privilèges associés à un autre contexte. La propriété d'*absence de changement d'un contexte* peut ainsi être considérée comme une propriété d'intégrité ou de confidentialité suivant les privilèges accessibles.

Dans notre représentation, un sujet scs_1 ne peut pas changer son contexte en scs_2 s'il n'existe pas de *transition* entre ces deux contextes, ce que nous appelons la condition $C_{3.21}(scs_1, scs_2)$.

$$C_{3.21}(scs_1, scs_2) \text{ est vraie } \textbf{ssi} \quad \neg scs_1 \xrightarrow{trans} scs_2 \tag{3.21}$$

De même, un sujet scs_1 ne peut pas obtenir le contexte scs_2 si il n'existe pas de *séquence de transitions* entre ces deux contextes, ce que nous appelons la condition $C_{3.22}(scs_1, scs_2)$.

$$C_{3.22}(scs_1, scs_2) \text{ est vraie } \textbf{ssi} \quad \neg scs_1 \Rightarrow_{trans} scs_2 \tag{3.22}$$

Nous définissons ainsi la propriété d'absence de changement de contexte entre deux sujets.

Propriété 3.3.18 *Un contexte sujet scs_1 ne peut pas obtenir le contexte scs_2 si et seulement si les conditions* (3.21) *et* (3.22) *sont respectées.*

$$P_{3.3.18}(scs_1, scs_2) \text{ est vraie ssi } \quad C_{3.21}(scs_1, scs_2) \wedge C_{3.22}(scs_1, scs_2)$$

A partir de cette propriété, nous pouvons définir la propriété de respect de l'*absence de changement de contexte sur un système*.

Propriété 3.3.19 *Soit $S = \{(scs_1, scd_1), \ldots, (scs_n, scd_n)\}$ un ensemble de couples de contextes. Un **système** respecte la propriété d'**absence de changement de contexte** si et seulement si pour tout couple (scs_i, scd_i), scs_i ne peut pas transiter vers scd_i.*

$$P_{3.3.19}(S) \text{ est vraie ssi } \quad \forall i = 1..n, P_{3.3.18}(scs_i, scd_i)$$

3.3.3.3 Exécutables de confiance

La propriété d'*exécutable de confiance* TPE (en anglais, *Trusted Path Execution*) a pour objectif de prévenir l'exécution d'applications auxquelles l'administrateur ne fait pas confiance (cf. annexe D.2). Cette notion de TPE peut être représentée dans notre modèle par un sous-ensemble de contextes de sécurité : $TPE \subset SC$, chaque contexte de cet ensemble est alors un *contexte de confiance*. Par exemple, sous GNU/Linux, un exemple d'ensemble de contextes de confiance TPE serait constitué des contextes associés aux répertoires `/sbin :/bin :/usr/bin :/usr/bin`. Les applications typées par ces contextes sont alors les seules dont l'exécution est autorisée par l'administrateur.

La propriété d'*exécutables de confiance* implique que pour toute interaction it autorisant l'exécution d'un contexte sc_2, ce contexte fait partie de l'ensemble des contextes de confiance TPE.

Propriété 3.3.20 *Soit un ensemble de contextes de confiance $TPE \subset SC$. Le **système** est dit **exécutable de confiance** si tout les objets exécutables appartiennent à l'ensemble des exécutables de confiance.*

$$P_{3.3.20}(TPE) \text{ est vraie ssi } \quad \forall sc_1, sc_2 \in SC, \forall eo \in IS \text{ t.q. } \left\{ \begin{array}{l} sc_1 \xrightarrow{eo} sc_2 \\ is\_execute\_like(eo) \end{array} \right. , \quad sc_2 \in TPE$$

Cette propriété est alors vérifiée si un contexte sc_1 exécute un contexte sc_2 qui n'appartient pas à TPE. Notons que cette propriété nécessite la définition d'un ensemble de contextes (l'ensemble TPE) pour pouvoir être appliquée.

3.3.3.4 Respect d'une politique de contrôle d'accès

Une règle de contrôle d'accès r correspond à une interaction $it = (sc_1, sc_2, eo)$ légale, c'est à dire autorisée sur un système. Une politique de contrôle d'accès $\mathcal{POL}$ définit un ensemble de règles de contrôle d'accès. $\mathcal{POL}$ est donc un ensemble d'interactions $\{it_1, \ldots, it_n\}$ légales. Il y a violation de la politique de contrôle d'accès lorsqu'une interaction it, non présente dans la politique, est effectuée. Nous définissons ainsi la propriété de *respect d'une politique de contrôle d'accès* comme :

Propriété 3.3.21 *Soit $\mathcal{POL} = \{it_1, \ldots, it_n\}$ une politique de contrôle d'accès. Un système respecte une politique de contrôle d'accès si toute interaction appartient à la politique.*

$$P_{3.3.21}(POL) \text{ est vraie ssi } \quad \forall sc_1, sc_2 \in SC, \forall eo \in IS \text{ t.q. } sc_1 \xrightarrow{eo} sc_2, \quad sc_1 \xrightarrow{eo} sc_2 \in \mathcal{POL}$$

Cette propriété peut être considérée comme une propriété d'intégrité d'une politique de contrôle d'accès. Cette propriété est violée si une interaction *observée* n'appartient pas à l'ensemble d'interactions défini par la politique.

3.3.3.5 Respect d'une Méta-Politique de contrôle d'accès

Finalement, nous proposons de définir une nouvelle propriété permettant de garantir le respect d'une Méta-Politique [Blanc 2006]. Une Méta-Politique de contrôle d'accès autorise l'utilisation d'un ensemble de politiques. Nous définissons ainsi la propriété de *respect d'une Méta-Politique de contrôle d'accès* comme :

Propriété 3.3.22 *Soit $\mathcal{MP} = \{\mathcal{POL}_1, \mathcal{POL}_2, \ldots, \mathcal{POL}_n\}$ une Méta-Politique de contrôle d'accès. Un système respecte une Méta-Politique de contrôle d'accès si toute interaction appartient à une des politiques de la Méta-Politique.*

$$P_{3.3.22}(MP) \text{ est vraie } \textbf{ssi} \quad \left\{ \begin{array}{l} \forall sc_1, sc_2 \in SC \\ \forall eo \in IS \end{array} \right. , \textbf{t.q.} \; sc_1 \xrightarrow{eo} sc_2, \quad \left\{ \begin{array}{l} \exists \mathcal{POL}_x \in \mathcal{MP} \\ sc_1 \xrightarrow{eo} sc_2 \in \mathcal{POL}_x \end{array} \right.$$

Dans la pratique, une Méta-Politique représente l'ensemble des interactions possibles en prenant en compte le changement dynamique de politique. Nous pouvons donc dire que $\mathcal{MP} \in \mathcal{P}(POL)$. Cette propriété est donc violée si il existe une interaction qui n'est pas incluse dans une des politiques $\mathcal{POL}_x$ possibles.

3.4 Discussion

Dans cette section, nous résumons l'ensemble des solutions existantes en termes de contrôle d'accès et de détection d'intrusions (cf. chapitre 2). Nous nous intéressons uniquement aux solutions qui exploitent la définition de propriétés de sécurité, sachant que nous avons formalisé ces propriétés au moyen de notre langage de description d'activités (cf. section 3.3). Ainsi, nous classifions assez aisément les solutions de la littérature en distinguant, leur portée (c'est-à-dire le fait qu'elles soient applicables à tout un système ou seulement à certains processus), leur caractère discrétionnaire (DAC) ou obligatoire (MAC), et les propriétés applicables. Cela nous permet de fournir un tableau comparatif qui fait clairement apparaître que toutes ces solutions sont limitées au contrôle des interactions et ne traitent pas des activités complexes basées sur des séquences ou des corrélations. Cette section détaille les limitations des approches existantes de contrôle d'accès et de détection d'intrusions.

3.4.1 Limite des solutions actuelles

Nous proposons, dans la table 3.2, une comparaison des modèles de contrôle d'accès et de détection d'intrusion (étudiés dans le chapitre 1) en fonction des propriétés de sécurité (définies dans ce chapitre). Dans cette table, nous distinguons, tout d'abord, les modèles MAC (mandataires) des modèles DAC (discrétionnaires). Nous donnons ensuite la portée de chaque modèle en distinguant trois classes :

- Système : Applicable sur l'ensemble des objets/sujets du système ;
- Processus privilégiés : Restreint aux processus administrateur ou aux services système ;
- Utilisateur : Restreint aux processus utilisateurs.

Finalement, nous précisons, pour chaque modèle, la liste des propriétés de sécurité qui peuvent être appliquées et pour chaque propriété, les classes d'activités contrôlables. Dans la table 3.2, nous utilisons les notations suivantes pour indiquer les classes d'activités contrôlables :

- $\surd$: la propriété peut être contrôlée totalement ;
- it : la propriété peut contrôler des *interactions* isolées ;
- un $flux$: la propriété peut contrôler les *séquences* de type *flux d'informations*.

	Contrôle d'Accès				**Détection d'Intrusions**		
Catégorie	**Modèle MAC**		**Implantation MAC**		**paramétrée par une politique DAC**		
Portée	Système				Processus privilégiés		Utilisateurs
Nom	Bell & LaPadula	Biba	SELinux	GRSECURITY	Spécification de comportement	Non-interférence	Contrôle de flux de référence
Intégrité							
- Objets			it	it	it		
- Sujets						it	
- No Read Down/No Write		$\surd\,(it)$	$\surd\,(it)$ MLS				
- Domaines			$\surd\,(it)$	$\surd\,(it)$			
Confidentialité							
- Objets			it it	it it	it it		it $flux$
- No Read Up/No Write Down	$\surd\,(it)$						
- No Read Up/No Write Down restrictif	$\surd\,(it)$		$\surd\,(it)$ MCS				
- Cohérence d'accès aux données							
Abus de privilèges							
- Séparation de privilèges			it	it			
- Absence de changement de contexte			it	it			
- Exécutables de confiance				$\surd\,(it)$			
- Respect d'une politique			$\surd\,(it)$	$\surd\,(it)$			
- Respect d'une Méta-Politique							

TAB. 3.2 – Propriétés de sécurité et modèles existants

Nous allons maintenant détailler les limitations des solutions de contrôle d'accès et de détection d'intrusions évoquées dans le chapitre 2.

3.4.1.1 Contrôle d'accès

Un mécanisme de contrôle d'accès MAC (cf. section 2.3, page 40) utilise une *politique de contrôle d'accès* afin de définir l'ensemble des appels système légaux entre entités (les interactions) et ainsi interdire les autres. Une politique de contrôle d'accès permet de contrôler des appels système, mais ne considère jamais des séquences ou des corrélations complexes d'appels système. Comme indiqué dans la table 3.2, les modèles ou les mécanismes de contrôle d'accès existants ne peuvent appliquer que certaines propriétés de sécurité. Certains mécanismes, tel que SELinux, peuvent garantir différentes propriétés de sécurité, mais seulement celles n'utilisant pas la notion de dépendance causale. Ainsi, les séquences d'interactions ou les corrélations ne sont pas prises en compte par ces modèles. Comme indiqué dans la figure 3.6, le spectre des attaques qui peuvent être bloquées par un mécanisme de contrôle d'accès se limite aux activités composées d'une seule interaction (un seul appel système).

3.4.1.2 Détection d'intrusions

La définition et l'usage d'une politique de contrôle d'accès ne sont pas limités au cas des systèmes MAC. En effet, même lorsque le système d'exploitation (par exemple, pour un système DAC), ne peut garantir cette politique, un IDS pourra en détecter les violations. Comme indiqué dans la table 3.2, les systèmes de détection d'intrusions paramétrés par une politique de contrôle d'accès sont généralement conçus pour des systèmes de type DAC.

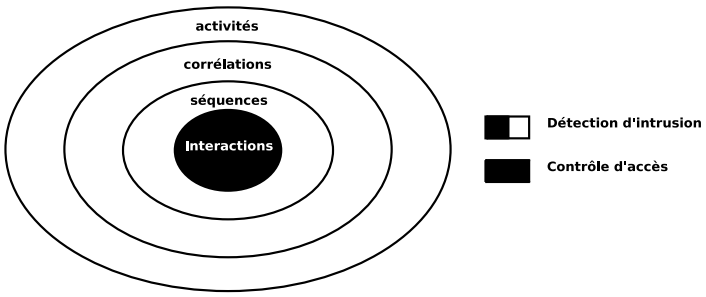


FIG. 3.6 – Classification des ensembles d'interactions

Ainsi, la *spécification de comportement* permet de détecter des violations de propriétés d'intégrité ou de confidentialité correspondant à des attaques simples (une interaction de modification ou de transfert d'informations, etc.). Cette approche ne permet pas de détecter des attaques complexes correspondant à des séquences causales ou des combinaisons de séquences. L'approche de *non-interférence* permet de détecter les attaques violant uniquement l'intégrité des sujets (processus). Finalement, ces deux approches étant basées sur l'analyse des appels système, elles nécessitent l'audit (l'observation) complet des processus du système. De ce fait, ces approches ne sont applicables qu'à une sous partie du système (les processus privilégiés).

L'approche de détection par *contrôle de flux de références* permet de détecter les flux d'informations correspondant à la propriété de confidentialité des objets. Elle ne garantit pas la propriété générale de confidentialité définie dans [ITSEC 1991] et elle ne traite pas les propriétés de type abus de privilèges. De plus, cette approche se limite au modèle DAC et ne prend donc pas en compte des interactions spécifiques aux systèmes MAC telles que les *transitions*. Finalement, cette approche se limite à l'ensemble des contextes utilisateurs et ne peut pas être appliquée à l'utilisateur administrateur ou aux services système (cf. section 2.2.3.3).

Ainsi, les approches existantes de détection d'intrusions paramétrées par une politique de sécurité ne permettent pas de détecter des attaques complexes reposant sur des séquences ou des corrélations. Finalement, ces approches ne sont applicables qu'à un sous ensemble du système, c'est-à-dire soit les processus privilégiés, soit les processus utilisateurs.

3.4.2 Conclusion

Dans notre modélisation d'un système, une *interaction* est une *action élémentaire* que peut effectuer un *contexte sujet* (processus) sur un *contexte objet* (qui peut être un autre processus ou une ressource du système). L'exécution d'un système correspond alors à une *suite* d'interactions.

Lors d'une attaque ou d'une tentative d'intrusion sur un système, le ou les attaquants engendrent un ensemble d'actions (i.e. d'interactions) violant une des *propriétés de sécurité*. Cette violation se traduit par l'exécution d'une seule interaction ou d'un ensemble d'interactions. Notre formalisation des propriétés de sécurité fait clairement apparaître la nécessité de considérer les séquences d'interactions ou leurs corrélations. Les attaques violant ces propriétés sont de complexité croissante : celles utilisant seulement une *interaction*, celles utilisant une *séquence* d'interactions et celles correspondant à une *combinaison* de séquences. Une attaque constituée d'une seule interaction peut être endiguée par un mécanisme de contrôle d'accès. Si certains systèmes de détection d'intrusions peuvent contrôler des cas particuliers de séquences, ils ne garantissent qu'une ou deux propriétés et ne s'appliquent que sur une sous partie du système (privilégié ou non privilégié). De plus, ces systèmes sont conçus

pour des mécanismes de contrôle d'accès de type DAC et ne prennent pas en compte des interactions spécifiques au modèle MAC, par exemple les transitions de processus.

Ainsi, notre formalisation des activités (cf. section 3.1) nous a permis de définir formellement l'ensemble des propriétés de sécurité (cf. section 3.3) rencontrées classiquement dans la littérature. Cette formalisation va bien au delà puisqu'elle nous a permis de définir de nouvelles propriétés de sécurité. De plus, elle nous permet de comparer l'ensemble des solutions existantes et de montrer clairement qu'aucune ne traître des séquences d'interactions, ni des activités complexes résultant des compositions de séquences et d'interactions. Notre langage de description d'activités manipule des notions de séquences causales et propose des opérateurs de corrélation qui permettent de les combiner. Il offre donc un pouvoir d'expression qui est bien adapté à ce que perçoit un système d'exploitation. En effet, les seules relations de cause à effet qu'il ne peut pas traiter sont celles qui ne laissent pas de trace sur le système et qui reposent donc sur des éléments extérieurs. Dans ce cas, on peut parler d'événements indépendants qui nécessitent de prendre en compte d'autres éléments que les traces système.

Dans le chapitre suivant, nous proposons une méthode générique, pour garantir des propriétés de sécurité, reposant sur la définition d'une politique de contrôle d'accès. Cette politique sert d'entrée pour le calcul des séquences violant les propriétés de sécurité souhaitées. Nous montrons que le calcul des séquences se ramène à calculer différents graphes. En effet, notre formalisation des activités permet d'énoncer un théorème d'équivalence entre les séquences du système et les chemins du graphe. Ensuite, notre formalisation des propriétés est utilisée pour développer des algorithmes qui énumèrent l'ensemble des violations possibles de ces propriétés. Ainsi, dans le chapitre 5, nous proposons une méthode pour détecter l'apparition des activités illicites. En résumé, grâce à notre langage de description d'activités et notre formalisation des propriétés, nous pouvons énumérer puis détecter toutes les violations de ces propriétés. Nous proposons donc une méthode générale qui détecte toutes les violations observables d'un point de vue système dès lors que la politique de contrôle d'accès est correctement définie.

Chapitre 4

Analyse d'une politique de contrôle d'accès

Dans ce chapitre, nous nous intéressons aux garanties des propriétés d'*intégrité*, de *confidentialité* et d'*abus de privilèges* définies dans la section 3.3. Nous avons vu dans la section 3.4 qu'un mécanisme de contrôle d'accès (qu'il soit de type DAC ou MAC) ne peut pas contrôler des éléments complexes telles que les séquences ou les corrélations. En effet, un tel mécanisme utilise une *politique de contrôle d'accès* afin de définir l'ensemble des actions *légales* sur un système. Chaque règle de contrôle d'accès permet de contrôler une interaction et ceci sans tenir compte des dépendances causales avec les autres règles.

Il est cependant possible de développer des méthodes de vérification basées sur ces règles de contrôle d'accès. Dans [Guttman *et al.* 2003] sont exposés les principes de l'outil SLAT qui permet de faire une telle vérification sur les règles de contrôle d'accès SELinux. Mais l'outil SLAT vérifie uniquement la présence de flux d'informations entre deux contextes. De plus, cet outil retourne un seul flux violant une propriété donnée et n'énumère donc pas toutes les attaques (activités illicites) possibles violant cette propriété.

Dans notre cas, le problème de vérification devient plus difficile puisqu'il s'agit de contrôler les différentes catégories de séquences et de corrélations exposées dans la section 3.2. Dans le cadre de cette thèse, nous souhaitons proposer une approche d'analyse d'une politique de contrôle d'accès permettant d'énumérer toutes les activités violant une des propriétés de sécurité. Il peut alors s'agir des propriétés d'intégrité, de confidentialité et d'abus de privilèges définies dans la section 3.3, ou de toute nouvelle propriété exprimée dans notre langage.

Dans la section 4.1, nous définissons formellement la notion de *politique de contrôle d'accès* en utilisant la modélisation d'un système donné dans la section 3.1. Nous introduirons ensuite la notion de *graphe d'interactions* qui représente, sous forme d'un graphe, l'ensemble des interactions autorisées par la politique de contrôle d'accès. Dans la section 4.2, nous utilisons le graphe d'interactions pour calculer un *graphe de dépendance causale* qui représente les dépendances entre interactions d'une politique de contrôle d'accès. Nous montrons alors qu'il y a équivalence entre les *chemins* de ce graphe et les *séquences* autorisées par la politique. Nous proposerons ensuite deux types de graphe de dépendance causale afin de représenter les cas particuliers de séquences (flux d'informations et séquence de transitions). De même, nous montrons l'équivalence entre ces deux graphes et les séquences de la politique.

Ces trois types de graphe de dépendance causale correspondent aux terminaux de notre langage. De ce fait, l'implantation de notre langage revient à manipuler les chemins dans ces trois graphes.

Ainsi, nous proposons, dans la section 4.3, des algorithmes généraux pour énumérer les terminaux du langage de description d'activités. La définition de ces graphes et la notion d'équivalence entre un chemin et une séquence permettent alors, dans la section 4.4, de définir des algorithmes d'énumération des violations possibles des propriétés de sécurité. Chaque violation correspond ainsi à un élément d'une des classes d'activités. Dans le chapitre 5, nous proposons une nouvelle approche de détection d'intrusions pour détecter l'apparition de ces violations.

L'analyse d'une politique de contrôle d'accès, développée dans ce chapitre, a fait l'objet de plusieurs publications [Briffaut *et al.* 2005a, Briffaut 2005, Blanc *et al.* 2006, Briffaut *et al.* 2006c].

4.1 Politique de contrôle d'accès

Une politique de contrôle d'accès (cf. section 2.1.2) définit un ensemble de règles de contrôle utilisé par le mécanisme de contrôle d'accès d'un système afin d'autoriser ou d'interdire chaque interaction entre entités du système. Nous commençons par définir la notion de *vecteur d'interactions* qui correspond à un ensemble d'*opérations élémentaires* légales entre deux contextes de sécurité. Un vecteur représente ainsi un ensemble de règles de contrôle d'accès définies entre deux entités (contextes) du système. A partir de cette définition, nous proposons une formalisation d'une politique de contrôle d'accès en tant qu'ensembles de vecteurs d'interactions. Cette formalisation sert ensuite de base à la construction du *graphe d'interactions* qui représente l'ensemble des interactions autorisées par une politique de contrôle d'accès.

4.1.1 Vecteur d'interactions

Nous définissons par *vecteur d'interactions*, l'ensemble d'*opérations élémentaires* que peut effectuer un contexte de sécurité sur un autre contexte. Un vecteur d'interactions, noté iv (pour *Interaction Vector*), est alors un triplet composé d'un contexte de sécurité source sc_{source}, d'un contexte de sécurité cible sc_{cible} et d'un ensemble d'opérations élémentaires is :

Définition 4.1.1 *Un vecteur d'interactions iv est de la forme :*

$$iv = (sc_{source} \in SC_S, sc_{cible} \in SC, is \subset IS)$$
$$où \;\; is = \{eo_1, eo_2, \ldots, eo_n\}$$

L'ensemble IS correspond à l'ensemble des opérations élémentaires définies pour un système. Le contexte sc_{source} peut donc utiliser toute opération $eo_i \in is$ sur sc_{cible}.

Par exemple, soit sc_{apache_d} le contexte associé à chaque processus `apache` et $sc_{var\_www\_t}$ le contexte associé à tout fichier du répertoire `/var/www/`. Un vecteur d'interactions $iv_{web} = (sc_{apache\_d}, sc_{var\_www\_t}, \{read, write\})$ autorise le contexte de sécurité sc_{apache_d} à accéder en lecture/écriture à tout fichier ayant comme contexte $sc_{var\_www\_t}$. De ce fait, tout processus étiqueté par le type sc_{apache_d} pourra accéder en lecture ou en écriture aux fichiers du système étiquetés par $sc_{var\_www\_t}$.

Nous notons IV l'ensemble de tous les vecteurs d'interactions qu'il est possible de définir en prenant toutes les paires de contextes de sécurité et tous les ensembles d'opérations possibles entre ces paires.

4.1.2 Politique de contrôle d'accès

Comme nous l'avons vu dans la section 2.3 (page 40), une *politique de contrôle d'accès* définit un ensemble de règles régissant les actions entre entités d'un système. Chaque règle décrit alors un ensemble de *privilèges* entre deux contextes de sécurité : un contexte de sécurité sc_{source} et un contexte de sécurité sc_{cible}. Dans notre modélisation, une règle de contrôle d'accès peut donc se traduire par : *"le contexte de sécurité sc_{source} est autorisé à effectuer les opérations $is = \{eo_1, eo_2, \ldots, eo_n\}$ sur le contexte de sécurité sc_{cible}"*. Ainsi, un *mécanisme de contrôle d'accès* système permet de contrôler les *opérations* effectuées sur un système, c'est-à-dire d'accepter ou de refuser l'utilisation, par une entité, d'une opération, sur une autre entité. Une action effectuée sur un système sera alors *autorisée* si elle correspond à une règle de la politique et *refusée* dans le cas contraire.

Nous avons défini la notion de *vecteur d'interactions* comme un ensemble d'opérations élémentaires entre deux contextes sc_{source} et sc_{cible}, chaque règle de contrôle d'accès correspond ainsi à un vecteur d'interactions. De par cette définition, une politique de contrôle d'accès, notée $\mathcal{POL}$, est donc un sous-ensemble de IV :

$$\mathcal{POL} \subset IV$$

Ainsi, nous dirons qu'une interaction est **légale**, du point de vue de la politique de contrôle d'accès, si et seulement si elle correspond à une règle de la politique de contrôle d'accès $\mathcal{POL}$:

Propriété 4.1.1 *Soit $sc_1 \in SC_S, sc_2 \in SC, eo \in IS$, une interaction $sc_1 \xrightarrow{eo} sc_2$ est **légale** dans la politique ($\mathcal{POL}$) si et seulement si $\exists iv = (sc_1, sc_2, is) \in \mathcal{POL}$ tel que $eo \in is$.*

Une politique de contrôle d'accès étant constituée d'un ensemble de vecteurs d'interactions ($\mathcal{POL} \subset IV$), l'ensemble des permissions autorisées entre deux contextes de sécurité sont contenues dans un vecteur d'interactions ($iv \in \mathcal{POL}$). Nous définissons alors la notion d'appartenance d'une interaction it à un ensemble de vecteurs d'interactions $\mathcal{X}$, notée $it \sqsubset \mathcal{X}$, par :

Définition 4.1.2 *Soit $it = (sc_{source} \in SC_S, sc_{cible} \in SC, eo_1 \in IS)$, $\mathcal{X} \subset IV$, nous avons $it \sqsubset \mathcal{X}$ si et seulement si $\exists is_1 \in IS$ tel que $iv = (sc_{source}, sc_{cible}, is_1) \in \mathcal{X}$ et $eo_1 \in is_1$*

Ainsi une interaction it, correspondant à une opération élémentaire eo_1, appartient à un ensemble de vecteur d'interactions $\mathcal{X}$ si il existe un vecteur d'interactions iv dans cette ensemble ayant les mêmes contextes source et cible et dont l'ensemble d'opérations is_1 contient l'opération élémentaire eo_1.

4.1.3 Exemple de politique de contrôle d'accès

Le listing 4.1 contient un exemple simplifié de politique de contrôle d'accès. Dans cet exemple, chaque contexte ne possède qu'un seul attribut : le domaine pour les sujets (postfixés par `_d`) ou le type pour les objets (postfixés par `_t`). Cet exemple correspond à la définition d'une politique de contrôle d'accès implantant le modèle DTE (*Domain Type Enforcement*) (cf. partie 2.3.2.3). Chaque ligne de ce listing représente un *vecteur d'interactions*. Le système sur lequel est appliqué cette politique est donc constitué de onze contextes de sécurité :

$$SC_S = \{sc_{login\_d}, sc_{ssh\_d}, sc_{admin\_d}, sc_{user\_d}, sc_{apache\_d}, sc_{webserv\_d}\}$$

$$SC_O = \{sc_{apache\_conf\_t}, sc_{var\_www\_t}, sc_{user\_info\_t}, sc_{admin\_info\_t}, sc_{temp\_t}\}$$

```
(login_d, admin_d, {transition})                                    1
(login_d, user_d, {transition})

(ssh_d, user_d, {transition})
                                                                    5
(user_d, admin_d, {transition})
(user_d, webserv_d, {transition})
(user_d, user_info_t, {read, write})
(user_d, temp_t, {read, write})
                                                                    10
(admin_d, apache_d, {transition})
(admin_d, webserv_d, {transition})
(admin_d, apache_conf_t, {read, write})
(admin_d, admin_info_t, {read, write})
(admin_d, temp_t, {read, write})                                    15

(apache_d, apache_conf_t, {read})
(apache_d, webserv_d, {transition})
(apache_d, var_www_t, {read, write, execute})
                                                                    20
(webserv_d, admin_info_t, {read})
(webserv_d, user_info_t, {read})
(webserv_d, var_www_t, {read, write, execute})
```

Listing 4.1 – Exemple de politique de contrôle d'accès.

De plus, cette politique considère quatre opérations élémentaires :

$$IS = \{transition, read, write, execute\}$$

Dans cet exemple, un service Web, accessible par différents utilisateurs, est utilisé pour traiter des informations personnelles stockées dans des fichiers. Ce service est accessible directement par les utilisateurs via l'exécution d'un binaire (une seule transition est alors nécessaire), ou indirectement via une interface Web (une séquence de transitions est alors nécessaire). Cependant, lors de l'exécution de ce service, le processus appelant doit effectuer une transition vers le domaine de ce service (`webserv_d`). Cette transition a pour objectif de limiter les droits du service Web sur le système en empêchant, par exemple, l'accès en écriture aux fichiers personnels.

Deux types de sessions, utilisateur (`user_d`) et administrateur (`admin_d`), sont accessibles via une authentification locale (`login_d`). Une authentification réussie se traduit alors par une transition du domaine `login_d` vers le domaine `user_d` ou `admin_d` (ligne 1 et 2). De plus, l'utilisateur peut se connecter via `ssh`, ce qui correspond à une transition du domaine `ssh_d` vers le domaine `user_d` (ligne 4). Un utilisateur peut aussi obtenir les privilèges administrateur via une transition (ligne 6). Les utilisateurs et les administrateurs peuvent *lire* et *écrire* dans des fichiers (stockant des informations personnelles) qui sont propres à chaque domaine (ligne 8 et 14). De plus, ils ont accès à un répertoire temporaire (ligne 9 et 15) et au service Web via une transition (ligne 7 et 12). L'administrateur peut modifier la configuration d'un serveur Web (ligne 13) et exécuter ce serveur (ligne 11). Ce serveur Web (ligne 17 à 19) peut lire sa configuration, exécuter le service Web et lire/écrire/modifier ses fichiers Web (`var_www_t`). Finalement, le service Web peut lire les fichiers personnels (ligne 21 et 22) des différents utilisateurs et lire/écrire/modifier les fichiers Web (ligne 23).

4.1.4 Graphe d'interactions

Une politique de contrôle d'accès, décrite sous forme d'un ensemble de vecteurs d'interactions ($\mathcal{POL} \subset IV$), peut être représentée à l'aide d'un graphe orienté $G = (N, E)$ où un nœud $n \in N$ est étiqueté par un contexte de sécurité et un arc de $e \in E$ par un ensemble d'opérations élémentaires. Nous définissons par *graphe d'interactions*, noté $G = (N, E)$, un graphe représentant tous les vecteurs d'interactions d'une politique. A chaque vecteur d'interactions $iv = (sc_1 \in SC_S, sc_2 \in SC, is \subset IS) \in \mathcal{POL}$, nous associons donc un arc $e = (sc_1, sc_2) \in E$ étiqueté par is et deux nœuds étiqueté par sc_1 et sc_2. Une interaction $sc_1 \xrightarrow{eo} sc_2$, définie dans la politique de contrôle d'accès, correspond ainsi à un arc étiqueté par eo entre deux nœuds étiquetés par sc_1 et sc_2.

4.1.4.1 Construction du graphe

`graphe_politique`
début
  Données : $\mathcal{POL} \subset IV$
  Résultat : $G = (N, E)$
  N=$\emptyset$, E=$\emptyset$;
  pour chaque $iv = (sc_1, sc_2, is) \in \mathcal{POL}$ **faire**
    e = (sc_1, sc_2);
    E=$E \cup \{e\}$;
    `valuer`$(e, \{is\}, G)$;
    N=$N \cup \{sc_1\} \cup \{sc_2\}$;
  fin
  retourner $G = (N, E)$
fin

Algorithme 1 : Algorithme de construction du graphe d'interactions

Afin d'attribuer un sous-ensemble d'opérations élémentaires à chaque arc d'un graphe, nous définissons tout d'abord deux fonctions qui permettent de lier une valuation à un arc :

Fonction 4.1.1 `valuation` $: E \times \mathcal{G} \rightarrow \mathcal{P}(IS)$ *est une fonction qui, étant donné un arc* $\in$ *et un graphe, retourne l'ensemble des opérations élémentaires associé à cet arc dans ce graphe.*
`valuer` $: E \times \mathcal{P}(IS) \times \mathcal{G}$ *est une fonction qui fixe l'ensemble d'opérations élémentaires d'un arc* $\in$ *dans un graphe.*

Par exemple, pour un graphe G, l'appel à `valuer(`e_1`,` $\{transition\}, G$`)` associera donc l'interaction $transition$ à l'arc e_1, un appel à `valuation(`e_1, G`)` retournera donc $\{transition\}$.

La construction du graphe d'interactions d'une politique de contrôle d'accès $\mathcal{POL}$ est réalisée par l'algorithme 1. Cet algorithme prend en entrée une politique de contrôle d'accès $\mathcal{POL}$ et renvoie le graphe d'interactions correspondant à cette politique. Ainsi, pour chaque vecteur d'interactions $iv \in IV$ de cette politique un arc e est ajouté à l'ensemble des arcs E. Cet arc est étiqueté par l'ensemble des opérations élémentaires $is \in IS$ autorisées entre ces deux contextes. Les contextes source et destination sont ajoutés à l'ensemble des nœuds N du graphe. Notons finalement que le fait qu'un contexte de sécurité objet ne peut être la source d'une interaction, implique que les nœuds objets sont toujours des puits dans ce graphe.

4.1.4.2 Exemple

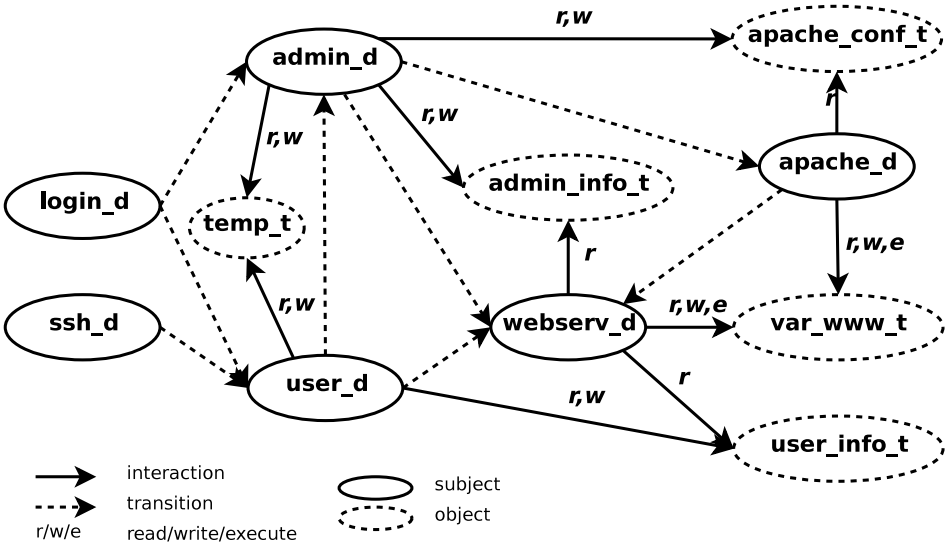


FIG. 4.1 – Graphe d'interactions de la politique du listing 4.1

Le graphe d'interactions correspondant à la politique de contrôles d'accès du listing 4.1 (page. 4.1) est donné dans la figure 4.1. Chaque contexte de sécurité correspond à un nœud unique tracé en pointillé pour les objets et en trait plein pour les sujets. Les ensembles d'opérations entre chaque couple de contextes correspondent à des arcs entre ces nœuds. Les arcs en pointillé correspondent aux transitions de contexte gérées par le mécanisme de contrôle d'accès, les autres arcs correspondent aux opérations système (i.e. appels système).

Par exemple, le vecteur d'interactions `(apache_d, var_www_t, { read, write, execute})` de la politique se traduit dans ce graphe par deux nœuds et un arc. Les nœuds étiquetés par $apache\_d$ et var_www_t correspondent respectivement au contexte de sécurité sujet associé au processus `apache` et au contexte de sécurité objet associé aux fichiers Web. L'arc entre ces deux nœuds étiqueté par `r, w, e` correspond aux opérations légales entre ces deux contextes.

4.2 Graphe de dépendance causale

Nous définissons par *graphe de dépendance causale*, noté $G_c = (N, E)$, un graphe représentant les *dépendances causales* entre chaque interaction d'une politique. Pour chaque vecteur d'interactions $iv = (sc_1 \in SC_S, sc_2 \in SC, is \subset IS) \in \mathcal{POL}$, nous associons un arc $e_1 = (sc_1, sc_2) \in E$ étiqueté par un sous-ensemble d'opérations élémentaires $is_1 \subset is$ et un arc $e_2 = (sc_2, sc_1) \in E$ étiqueté par un sous-ensemble d'opérations élémentaires $is_2 \subset is$. Chaque opération $eo \in is_1$ correspond à une opération qui permet à sc_1 de *modifier l'état* de sc_2. De même, chaque opération $eo \in is_2$ correspond à une opération qui permet à sc_2 de *modifier l'état* de sc_1. Nous définissons ainsi une *table de changement d'état* qui associe à chaque opération élémentaire $eo \in IS$ *le sens de l'effet* de cette opération. Pour chaque opération, cet effet correspond à *le contexte appelant modifie l'état du contexte appelé* (noté $\rightarrowtail$) ou *le contexte appelé modifie l'état du contexte appelant* (noté $\leftarrowtail$). Ainsi, nous définissons la fonction `ordre` qui, en fonction de la table T, associe à une opération élémentaire un arc :

Fonction 4.2.1 `ordre` *est une fonction qui, étant donné une interaction* $it = sc_1 \xrightarrow{eo} sc_2$ *et une table de changement d'état T, retourne l'arc* e *correspondant à l'effet de* eo*.*

Le listing 4.2 contient un exemple de table T_{ex1} pour quatre opérations $eo_1, \ldots, eo_4$. D'après cette table, eo_1 et eo_2 modifient l'état du contexte *appelé*, alors que eo_3 et eo_4 modifient l'état du contexte *appelant*. Ainsi, pour deux interactions $it_1 = (sc_1, sc_2, eo_1)$ et $it_2 = (sc_1, sc_2, eo_4)$, l'appel à `ordre(`$it_1$`,` T_{ex1}`)` renvoie l'arc $e = (sc_1, sc_2)$ alors que l'appel à `ordre(`it_2`,` T_{ex1}`)` renvoie l'arc $e = (sc_2, sc_1)$.

eo_1	$\rightarrowtail$
eo_2	$\rightarrowtail$
eo_3	$\leftarrowtail$
eo_4	$\leftarrowtail$

Listing 4.2 – Exemple de table de changement d'état.

4.2.1 Construction

Nous proposons l'algorithme 2 de construction du *graphe de dépendance causale*. Cet algorithme prend en paramètres un graphe d'interactions G_{int} et d'une table de changement d'état T. Il renvoie alors le graphe G_c associé au graphe d'interactions (et donc à la politique de contrôle d'accès $\mathcal{POL}$).

```
graphe_dependance_causale
début
    Données : G_POL, T
    Résultat : G_c = (N, E)
    N=∅, E=∅;
    pour chaque sc_1 ∈ G_int faire
        pour chaque sc_2 ∈ G_int faire
            is = valuation(sc_1, sc_2, G_int);
            pour chaque eo ∈ is faire
                e = ordre((sc_1, sc_2, eo), T);
                valuer(e, {eo});
                E = E ∪ {e};
                N = N ∪ {sc_1} ∪ {sc_2};
            fin
        fin
    fin
    retourner G_c = (N, E)
fin
```

Algorithme 2 : Algorithme de construction du graphe de dépendance causale.

Dans la suite, nous noterons $sc_1 \longmapsto_{eo_1} sc_2$ un arc de sc_1 vers sc_2 contenant la valuation eo_1. Par construction, l'arc $sc_1 \longmapsto_{eo_1} sc_2$ correspond donc à une des deux interactions suivantes :

– $(sc_1, sc_2, \{eo_1\})$ si eo_1 modifie l'état de sc_2 (l'appelé) ;

– $(sc_2, sc_1, \{eo_1\})$ si eo_1 modifie l'état de sc_1 (l'appelant).

Nous pouvons ainsi définir la propriété d'équivalence entre un *arc* et une *interaction* :

Propriété 4.2.1 *Dans le graphe de dépendance causale G_c, l'arc $sc_1 \longmapsto_{eo_1} sc_2$ correspond à l'interaction $it = (sc_1, sc_2, \{eo_1\})$ ou l'interaction $it = (sc_2, sc_1, \{eo_1\})$.*

De ce fait, chaque interaction $it \sqsubset \mathcal{POL}$ d'une politique de contrôle d'accès correspond à un arc e dans le graphe de dépendance causale (étiqueté par l'opération élémentaire de it).

Nous proposons finalement la propriété d'observation d'un chemin

Propriété 4.2.2 *Nous dirons que le chemin $sc_1 \longmapsto_{eo_1} sc_2 \longmapsto_{eo_2} sc_3$ est observé sur le système, si l'arc $sc_1 \longmapsto_{eo_1} sc_2$ "apparaît" avant l'arc $sc_1 \longmapsto_{eo_1} sc_2$. C'est-à-dire, d'après la propriété 4.2.1, que l'interaction it_1 "apparaît" avant que it_2 soit fini ($t_{sc_2}(debut(eo_1)) \leqslant t_{sc_2}(fin(eo_2))$).*

4.2.2 Exemple

Le listing 4.3 contient un exemple de politique de contrôle d'accès $\mathcal{POL}$ constituée de quatre contextes de sécurité $sc_1, \ldots, sc_4$ et de quatre vecteurs d'interactions.

$(sc_1, sc_2, \{ eo_1 \})$	1
$(sc_2, sc_4, \{ eo_2 \})$	2
$(sc_3, sc_1, \{ eo_3 \})$	3
$(sc_1, sc_3, \{ eo_4 \})$	4

Listing 4.3 – Exemple de politique de contrôle d'accès.

La figure 4.2 représente le graphe de dépendance causale obtenu par application de l'algorithme 2 sur la politique 4.3 et la table (T_{ex1}) contenue dans le listing 4.2. Ce graphe contient ainsi quatre nœuds étiquetés par chaque contexte de sécurité et quatre arcs.

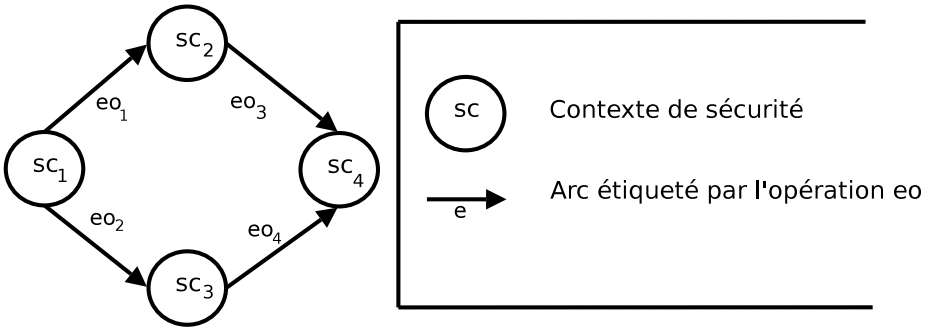


FIG. 4.2 – Exemple de graphe de dépendance causale

Dans cet exemple, si nous prenons le cas des deux interactions $it_1 = (sc_1, sc_3, \{eo_2\})$ et $it_2 = (sc_4, sc_3, \{eo_4\})$. D'après la définition de la dépendance causale (def. 3.2.2, page 65), nous avons $it_1 \twoheadrightarrow it_2$ car it_1 modifie l'état du contexte partagé sc_3 (eo_2 est de type $\rightarrowtail$), it_2 modifie l'état du contexte cible sc_4 (eo_4 est de type $\leftarrowtail$). Nous voyons clairement dans cet exemple que la dépendance causale $it_1 \twoheadrightarrow it_2$ se traduit par un chemin $sc_1 \longmapsto_{eo_2} sc_3 \longmapsto_{eo_4} sc_4$.

Nous proposons donc, dans la partie suivante, de prouver que dans ce graphe, un chemin $sc_1 \longmapsto_{eo_1} sc_2 \longmapsto_{eo_2} sc_3$, où l'arc $sc_1 \longmapsto_{eo_1} sc_2$ correspond à une interaction it_1 et l'arc

$sc_2 \longmapsto_{eo_2} sc_3$ correspond à une interaction it_2, implique une dépendance causale $it_1 \twoheadrightarrow it_2$. Nous montrerons finalement qu'un chemin $sc_1 \longmapsto_{eo_1} \ldots \longmapsto_{eo_{n-1}} sc_n$ dans ce graphe correspond à une fermeture transitive causale $it_1 \twoheadrightarrow \ldots \twoheadrightarrow it_{n-1}$ (une séquence $sc_1 \Rightarrow sc_n$).

4.2.3 Équivalence entre chemin et séquence

Dans cette section, nous proposons de prouver qu'il y a équivalence entre l'ensembles des fermetures transitives causales (séquences) et l'ensembles des chemins dans le graphe de dépendance causale. De ce fait, la recherche de tous les chemins existants entre deux nœuds, nous permettra d'énumérer toutes les séquences réalisables entre deux contextes. Nous proposons donc de prouver qu'un chemin correspond à une séquence, puis nous montrerons qu'une séquence correspond à un chemin. Finalement, nous énoncerons le théorème 4.2.1 d'équivalence entre un chemin du graphe et une séquence.

Nous commençons par montrer qu'un chemin de deux arcs implique une dépendance causale.

Lemme 4.2.1 *Soit it_1 et it_2 deux interactions effectuant respectivement les opérations eo_1 et eo_2, telles que $sc_1, sc_2 \in it_1$ et $sc_2, sc_3 \in it_2$.*
Si, dans le graphe d'interactions $G_c = (N, E)$, il existe un chemin $sc_1 \longmapsto_{eo_1} sc_2 \longmapsto_{eo_2} sc_3$ alors il y a dépendance causale $it_1 \twoheadrightarrow it_2$.

Preuve :
Comme sc_1 et sc_2 appartiennent à l'arc $sc_1 \longmapsto_{eo_1} sc_2$ étiqueté par eo_1, cet arc correspond à l'interaction it_1 (prop. 4.2.1). De même l'arc $sc_2 \longmapsto_{eo_2} sc_3$ correspond à l'interaction it_2.

Par construction, $sc_1 \longmapsto_{eo_1} sc_2$ implique que sc_1 *modifie l'état de* sc_2, donc it_1 *modifie l'état de* sc_2. De même, it_2 *modifie l'état de* sc_3.
De plus, $sc_1 \longmapsto_{eo_1} sc_2 \longmapsto_{eo_2} sc_3$ implique que sc_2 est *le contexte partagé* par it_1 et it_2.
L'apparition de $sc_1 \longmapsto_{eo_1} sc_2 \longmapsto_{eo_2} sc_3$ implique que it_1 apparaîtra avant it_2 et donc que $t_{sc_2}(debut(eo_1)) \leqslant t_{sc_2}(fin(eo_2))$ (prop. 4.2.2).

Finalement, nous avons :

$$sc_1 \longmapsto_{eo_1} sc_2 \longmapsto_{eo_2} sc_3 \Longrightarrow \begin{cases} sc_1 \longmapsto_{eo_1} sc_2 \Leftrightarrow it_1 \\ sc_2 \longmapsto_{eo_2} sc_3 \Leftrightarrow it_2 \\ sc_2 \in it_1 \\ sc_2 \in it_2 \\ t_{sc_2}(debut(eo_1)) \leqslant t_{sc_2}(fin(eo_2)) \\ it_1 \text{ modifie l'état du contexte partagé } sc_2 \\ it_2 \text{ modifie l'état du contexte } sc_3 \end{cases}$$

Ceci correspond à la définition de la dépendance causale (def. 3.2.2, page 65). Nous avons donc bien :

$$sc_1 \longmapsto_{eo_1} sc_2 \longmapsto_{eo_2} sc_3 \Longrightarrow it_1 \twoheadrightarrow it_2$$

□

Nous montrons maintenant qu'un chemin entre n contextes $(sc_1, \ldots, sc_n)$ implique une séquence $sc_1 \Rightarrow sc_n$.

Lemme 4.2.2 *Si, dans le graphe de dépendance causale, il existe un chemin $sc_1 \longmapsto_{eo_1} sc_2 \longmapsto_{eo_2} \ldots \longmapsto_{eo_{n-1}} sc_n$ alors il existe une séquence $sc_1 \Rightarrow sc_n$.*

Preuve :
D'après la propriété. 4.2.1 :

$$\forall i = 1..n, [sc_i \longmapsto_{eo_i} sc_{i+1}] \Longrightarrow [it_i = sc_i \xrightarrow{eo_i} sc_{i+1} \text{ ou } sc_{i+1} \xrightarrow{eo_i} sc_i]$$

Et,

$$sc_1 \in it_1, sc_{n+1} \in it_n$$

D'autre part, dans les suppositions du lemme, nous avons :

$$sc_1 \longmapsto_{eo_1} sc_2 \longmapsto_{eo_2} \ldots \longmapsto_{eo_n} sc_{n+1}$$

Or, d'après le lemme 4.2.1 :

$$\forall i = 1..n-1, [sc_i \longmapsto_{eo_i} sc_{i+1} \longmapsto_{eo_{i+i}} sc_{i+2}] \Longrightarrow [it_i \twoheadrightarrow it_{i+1}]$$

Nous avons donc :

$$\forall i = 1..n-1, it_i \twoheadrightarrow it_{i+1}$$

Finalement, nous avons :

$$[sc_1 \longmapsto_{eo_1} sc_2 \longmapsto_{eo_2} \ldots \longmapsto_{eo_n} sc_{n+1}] \Longrightarrow \begin{cases} \forall i = 1..n, sc_i \longmapsto_{eo_i} sc_{i+1} \\ sc_1 \in it_1 \\ sc_{n+1} \in it_n \\ \forall k = 1..n-1, it_k \twoheadrightarrow it_{k+1} \end{cases}$$

Ceci correspond à la définition d'une séquence (def. 3.2.3, page 65). Nous avons donc bien :

$$[sc_1 \longmapsto_{eo_1} sc_2 \longmapsto_{eo_2} \ldots \longmapsto_{eo_{n-1}} sc_n] \Longrightarrow [sc_1 \Rightarrow sc_n]$$

□

Nous venons de montrer que si il existe un chemin dans le graphe de dépendance causale alors il existe une séquence dans la politique de contrôle d'accès. Nous allons maintenant montrer qu'une séquence implique un chemin dans le graphe de dépendance causale. Pour cela, nous montrons, tout d'abord, qu'une dépendance causale implique un chemin de deux arcs dans le graphe de dépendance causale.

Lemme 4.2.3 *Soit it_1 et it_2 deux interactions effectuant respectivement les opérations eo_1 et eo_2, tels que $sc_1, sc_2 \in it_1$ et $sc_2, sc_3 \in it_2$.*
Si il existe une dépendance causale $it_1 \twoheadrightarrow it_2$ alors il existe un chemin $sc_1 \longmapsto_{eo_1} sc_2 \longmapsto_{eo_2} sc_3$ dans le graphe de dépendance causale.

Preuve :
D'après la définition de la dépendance causale 3.2.2 :

$$it_1 \twoheadrightarrow it_2 \Longleftrightarrow \begin{cases} sc_1, sc_2 \in it_1 \\ sc_2, sc_3 \in it_2 \\ t_{sc_2}(debut(eo_1)) \leqslant t_{sc_2}(fin(eo_2)) \\ it_1 \text{ modifie l'état du contexte partagé } sc_2 \\ it_2 \text{ modifie l'état du contexte } sc_3 \end{cases}$$

Comme it_1 modifie l'état du contexte partagé sc_2 , cela implique, par construction (fonction `ordre`), qu'il existe un arc $sc_1 \longmapsto_{eo_1} sc_2$.
De même, it_2 modifie l'état du contexte sc_3 implique qu'il existe un arc $sc_2 \longmapsto_{eo_2} sc_3$.
Comme chaque contexte correspond à un unique nœud dans le graphe, nous avons donc :

$$[sc_1 \longmapsto_{eo_1} sc_2 \text{ et } sc_2 \longmapsto_{eo_1} sc_3] \Longrightarrow [sc_1 \longmapsto_{eo_1} sc_2 \longmapsto_{eo_2} sc_3]$$

De ce fait, nous avons donc bien :

$$[it_1 \twoheadrightarrow it_2] \Longrightarrow [sc_1 \longmapsto_{eo_1} sc_2 \longmapsto_{eo_2} sc_3]$$

□

Nous allons maintenant montrer qu'une séquence implique un chemin dans le graphe.

Lemme 4.2.4 *Soit n interactions it_i effectuant l'opération eo_i tel que $\forall i = 1..n$, on a $sc_i, sc_{i+1} \in it_i$.*
Si il existe une fermeture transitive causale (séquence) $sc_1 \Rightarrow sc_{n+1}$ alors il existe un chemin $sc_1 \longmapsto_{eo_1} sc_2 \longmapsto_{eo_2} \ldots \longmapsto_{eo_n} sc_{n+1}$ dans un graphe de dépendance causale.

Preuve :
D'après la définition 3.2.3, une fermeture transitive causale $sc_1 \Rightarrow sc_{n+1}$ implique que :

$$\forall i = 1..n-1, it_i \twoheadrightarrow it_{i+1}$$

D'après le lemme 4.2.3,

$$\forall i = 1..n-1, [it_i \twoheadrightarrow it_{i+1}] \Longrightarrow [sc_i \longmapsto_{eo_i} sc_{i+1} \longmapsto_{eo_{i+1}} sc_{i+2}]$$

Chaque contexte étant identifié de manière unique dans le graphe, nous avons donc :

$$sc_1 \longmapsto_{eo_1} sc_2 \longmapsto_{eo_2} \ldots \longmapsto_{eo_n} sc_{n+1}$$

De ce fait, nous avons donc bien :

$$[sc_1 \Rightarrow sc_{n+1}] \Longrightarrow [sc_1 \longmapsto_{eo_1} sc_2 \longmapsto_{eo_2} \ldots \longmapsto_{eo_n} sc_{n+1}]$$

□

A partir de ces quatre lemmes, nous pouvons désormais énoncer le théorème d'équivalence entre un chemin et une fermeture transitive causale.

Théorème 4.2.1 *Dans un graphe de dépendance causale $G_c = (N, E)$, il existe un chemin entre sc_1 et sc_n si et seulement si il existe une séquence (fermeture transitive causale) $sc_1 \Rightarrow sc_n$ dans la politique associée à ce graphe.*

Preuve :
D'après le lemme 4.2.2, si il existe un chemin entre sc_1 et sc_n, alors il existe une séquence $sc_1 \Rightarrow sc_n$:

$$[sc_1 \longmapsto_{eo_1} sc_2 \longmapsto_{eo_2} \ldots \longmapsto_{eo_{n-1}} sc_n] \Longrightarrow [sc_1 \Rightarrow sc_n]$$

D'après le lemme 4.2.4, si il existe une séquence $sc_1 \Rightarrow sc_n$, alors il existe un chemin entre sc_1 et sc_n :

$$[sc_1 \Rightarrow sc_n] \Longrightarrow [sc_1 \longmapsto_{eo_1} sc_2 \longmapsto_{eo_2} \ldots \longmapsto_{eo_{n-1}} sc_n]$$

Nous avons donc bien :

$$[sc_1 \Rightarrow sc_n] \Longleftrightarrow [sc_1 \longmapsto_{eo_1} sc_2 \longmapsto_{eo_2} \ldots \longmapsto_{eo_{n-1}} sc_n]$$

□

Ce théorème montre une équivalence entre le graphe de dépendance causale et les séquences qui sont autorisées par une politique. Si cette preuve est obtenue facilement par construction du graphe, son intérêt pratique est prépondérant. D'après ce théorème d'équivalence entre un chemin et une fermeture transitive causale, nous pouvons donc énumérer toutes les séquences, entre deux contextes, *légales* du point de vue de la politique de contrôle d'accès, par énumération de tous les chemins entre deux nœuds dans le graphe de dépendance causale. Nous avons vu dans la section 3.3, que la formalisation des propriétés de sécurité fait apparaître des conditions portant sur l'existence de séquences. Une violation d'une de ces conditions se traduit ainsi par l'existence d'une séquence qui correspond ainsi à la présence d'un chemin dans le graphe de dépendance causale. Nous avons ainsi un modèle général permettant d'analyser une politique de contrôle d'accès afin de rechercher toute violation possible d'une propriété de sécurité.

4.2.4 Application

Nous proposons de considérer trois graphes de dépendance causale différents qui ne considèrent qu'un sous-ensemble des opérations élémentaires définies dans une politique de contrôle d'accès. Ces graphes nous permettront d'énumérer les trois types de séquences (définies dans la section 3.2) par simple recherche de chemins. Ainsi, chaque graphe sera associé à un type de séquence. Le graphe de dépendance causale que nous venons de définir permet ainsi d'énumérer les séquences générales (*séquence d'interactions*). Nous définirons ensuite deux graphes particuliers : le graphe de flux d'informations (*flux d'informations*) et le graphe de transitions (*séquence de transitions*).

Le *graphe de dépendance causale*, noté $G_{c_{dep}}$, représente ainsi l'ensemble des *opérations élémentaires* autorisées par une politique de contrôle d'accès. Notons que contrairement au *graphe d'interactions* (cf. 4.1.4), ce graphe représente l'effet d'une interaction et non l'interaction elle-même. Ainsi, dans le graphe d'interactions, une interaction $(sc_1, sc_2, \{eo_1\})$ correspond toujours à un arc $sc_1 \longmapsto sc_2$ étiqueté par eo_1. Alors que, dans le graphe de dépendance causale, cette interaction correspondra à un arc $sc_1 \longmapsto sc_2$ si eo_1 est une opération qui *modifie l'état* de sc_2 (type $\rightarrowtail$) ou à un arc $sc_2 \longmapsto sc_1$ si eo_1 *modifie l'état* de sc_1 (type $\leftarrowtail$).

Afin de construire ce graphe, nous définissons une table T_{dep} contenant l'ensemble des opérations disponibles sur le système ($\forall eo \in IS$). Cette table prend en compte les opérations élémentaires mises en jeu dans la politique de contrôle d'accès $\mathcal{POL}$. Ainsi, cette table associe à chaque opération eo du système un *ordre* $\rightarrowtail$ ou $\leftarrowtail$. A partir de la définition de cette table T_{dep}, nous utilisons l'algorithme 2 (page 95) afin de construire le *graphe de dépendance causale* d'un graphe d'interactions (correspondant à une politique de contrôle d'accès $\mathcal{POL}$). Par construction, ce graphe représente donc l'ensemble des dépendances causales entre interactions définies dans une politique de contrôle d'accès.

Étant donné que ce graphe représente l'ensemble des opérations élémentaires et d'après le théorème 4.2.1, nous pouvons définir la propriété d'équivalence entre un *chemin* dans ce graphe et une *séquence d'interactions* :

Propriété 4.2.3 *Soit $\mathcal{POL}$ une politique de contrôle d'accès et $G_{c_{dep}}$ le graphe de dépendance causale qui lui est associé.*

Il existe un ***chemin*** $sc_1 \longmapsto_{eo_1} \ldots \longmapsto_{eo_{n-1}} sc_n$ *dans* $G_{c_{dep}}$ *si et seulement si il existe une* ***séquence d'interactions*** $sc_1 \Rightarrow sc_n$ *dans* $\mathcal{POL}$.

Ce graphe peut donc être utilisé pour énumérer toutes les séquences possibles entre deux contextes.

4.2.4.1 Exemple

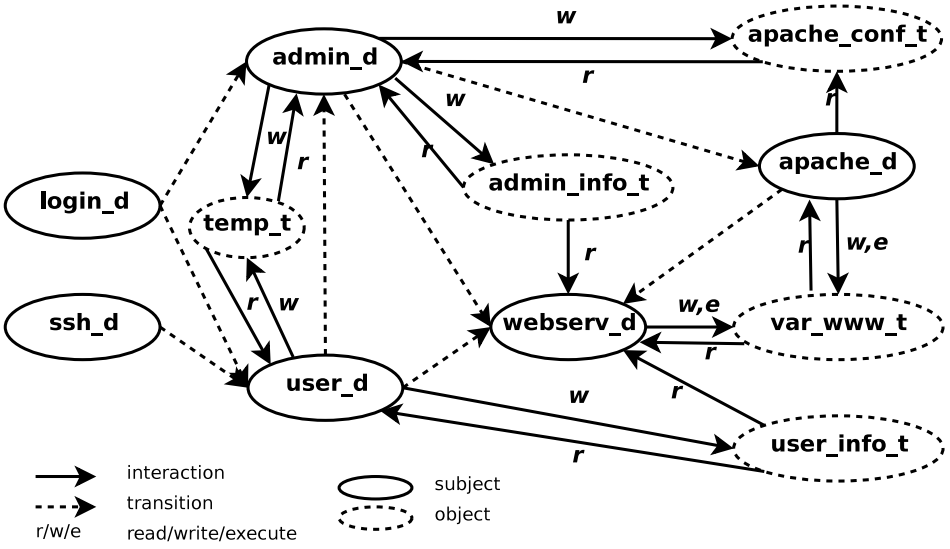


FIG. 4.3 – Graphe de dépendance causale correspondant au listing 4.1

La figure 4.3 représente le *graphe de dépendance causale* obtenu par application de l'algorithme de construction (algo. 2, page 95) à la politique de contrôle d'accès présent dans le listing 4.1. La table T_{dep}, utilisée pour construire ce graphe, est contenue dans le listing 4.4.

Dans cet exemple, nous considérons que seule l'opération de lecture ($read$) peut modifier l'état du contexte *appelant* car cette opération transfert de l'information depuis le contexte appelé vers l'appelant. Les autres interactions modifient l'état du contexte *appelé* (apparition du nouveau contexte ou modification de ce contexte).

```
transition  ⇀
write       ⇀
execute     ⇀
read        ↼
```

Listing 4.4 – Table de changement d'état pour le listing 4.4.

Un arc dans ce graphe, par exemple entre $user\_d$ et $user\_info\_t$, correspond à une modification possible de l'état de $user\_info\_t$ par $user\_d$. Par exemple, une modification (opération $write$) du contexte $user\_info\_t$. Cet arc correspond ainsi à l'interaction $sc_{user\_d} \xrightarrow{write} sc_{user\_info\_t}$.

L'existence d'un chemin entre les nœuds $user\_d$ et $webserv\_d$ implique qu'il existe *une séquence d'interactions* qui peut avoir comme incidence la modification de l'*état* du Web-service par un utilisateur. Par exemple, le chemin ($sc_{user\_d} \longmapsto_{write} sc_{user\_info\_t} \longmapsto_{read} sc_{webserv\_d}$) correspond à une

séquence qui peut permettre à l'utilisateur d'influer sur l'exécution du Web service via un *canal caché* (le fichier $user\_info\_t$). Un attaquant pourrait alors exploiter une vulnérabilité dans le Web service afin de détourner le comportement de cette application. Une telle vulnérabilité peut par exemple correspondre à une mauvaise interprétation des données écrites dans le fichier $user\_info\_t$ permettant l'exécution, par le Web service, de code malveillant fourni par l'utilisateur.

4.2.5 Graphe de flux d'informations

Le *graphe de flux d'informations*, noté $G_{c_{inf}}$, représente l'ensemble des *transferts d'informations* autorisés par une politique de contrôle d'accès. Un arc $e = sc_1 \longmapsto sc_2$ correspond à un transfert d'informations de sc_1 vers sc_2. Cet arc est étiqueté par l'ensemble des opérations pouvant réaliser ce transfert. De ce fait, ce graphe ne considère que les opérations élémentaires ayant pour effet le *transfert d'informations* d'un contexte vers un autre.

4.2.5.1 Construction du graphe

Comme nous l'avons vu dans la section 3.1.3.2 (page 58), une opération de type `read_like`, d'un contexte sc_1 sur un contexte sc_2, a pour *effet* le transfert d'informations de sc_2 vers sc_1 ($sc_2 > sc_1$). Ainsi, une opération de type `read_like` modifie l'état de l'*appelant* (type $\longmapsto$). Une opération de type `write_like` d'un contexte sc_1 sur un contexte sc_2, a pour *effet* le transfert d'informations de sc_1 vers sc_2 ($sc_1 > sc_2$). De ce fait, une opération de type `write_like` modifie l'état de l'*appelé* (type $\rightarrowtail$).

Ainsi, chaque opération élémentaire vérifiant la fonction `is_read_like` est de type $\longmapsto$; chaque opération vérifiant la fonction `is_write_like` est de type $\rightarrowtail$. Ces deux fonctions permettent de construire la table de changement d'état T_{inf}. A partir de la définition de cette table T_{inf}, nous utilisons alors l'algorithme général de construction (algo. 2, page 95) afin de construire le *graphe de flux d'informations* en fonction d'un graphe d'interactions.

Notons que l'implantation réelle des fonctions 3.1.1 et 3.1.2 utilise un fichier de correspondance qui permet d'associer à chaque opération élémentaire une catégorie (`read_like`, `write_like`, `both` ou `none`) et un *niveau* de transfert d'informations. Ce niveau se réfère à la "capacité" de transfert offerte par l'opération élémentaire. Il sera, par exemple, maximal pour une opération élémentaire de type "lecture dans un fichier" et minimal pour un "envoi de signal inter-processus". Ce paramètre permet, lorsqu'il est pris en compte lors de la construction de la table T_{inf}, de réduire le nombre de transferts d'informations possible et donc la taille du graphe. Nous détaillons cet aspect dans le chapitre 6.

Par construction, le *graphe de flux d'informations* représente l'ensemble des transferts d'informations présent dans une politique de contrôle d'accès. De plus, d'après la définition d'un flux d'informations (def. 3.2.4), un flux d'informations n'est constitué que de transfert d'informations. Étant donné qu'un flux d'informations est un cas particulier de séquence, nous pouvons appliquer le théorème 4.2.1 d'équivalence entre un chemin et une séquence. Nous énonçons donc la propriété d'équivalence entre un chemin du graphe de flux d'informations et un flux d'informations.

Propriété 4.2.4 *Soit $\mathcal{POL}$ une politique de contrôle d'accès et $G_{c_{inf}}$ le graphe de flux d'informations qui lui est associée.*
Il existe un ***chemin*** $sc_1 \longmapsto_{eo_1} \ldots \longmapsto_{eo_{n-1}} sc_n$ *dans $G_{c_{inf}}$ si et seulement si il existe un* ***flux d'informations*** $sc_1 \gg sc_n$ *dans $\mathcal{POL}$.*

Ainsi, le graphe de flux d'informations permet d'énumérer l'ensemble des flux d'informations présent dans une politique de contrôle d'accès.

4.2.5.2 Exemple

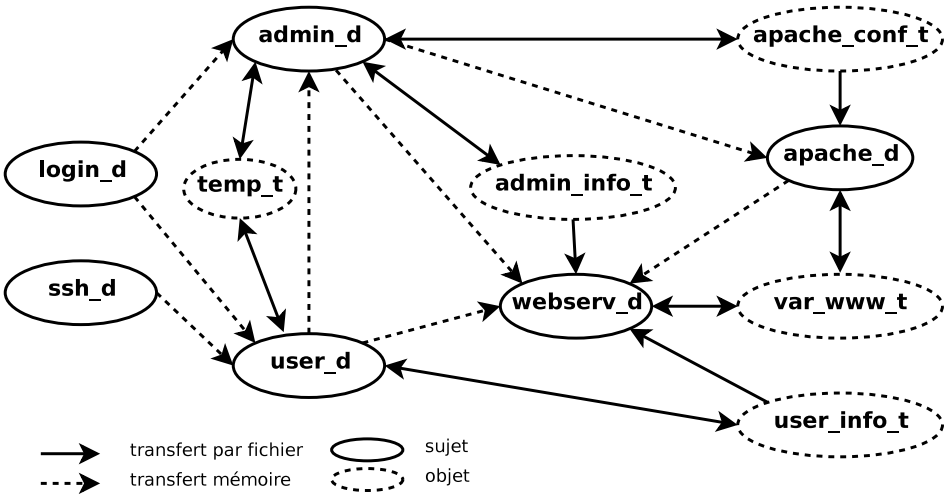


FIG. 4.4 – Graphe de flux d'informations correspondant au listing 4.1

La figure 4.4 représente le graphe de flux d'informations associé à la politique de contrôle d'accès du listing 4.1 (page 92). Dans ce graphe, les *transferts d'informations* entre contexte sujet et objet sont en trait plein ; alors que les transferts liés à des interactions inter-processus sont en pointillés (entre deux sujets). Le fichier de correspondance utilisé lors de la construction de ce graphe est contenu dans le listing 4.5. Ce fichier contient sur la première colonne le nom de l'opération élémentaire, la catégorie du transfert sur la deuxième et finalement la capacité de transfert de cette opération. Ainsi, nous pouvons remarquer qu'une transition de contexte implique un transfert de l'information

```
transition                    write_like          5
write                         write_like          10
execute                       read_like           1
read                          read_like           10
```

Listing 4.5 – Exemple de fichier de correspondance des opérations élémentaires.

du contexte source vers le contexte cible. En effet, comme le processus n'est pas modifié lors de cette transition, il conserve l'information acquise auparavant, impliquant ainsi un transfert d'informations de l'ancien contexte vers le nouveau. La table T_{inf} pour cette politique correspond donc au listing 4.6 (pour un $niveau \geqslant 1$).

```
transition                    ↣
write                         ↣
execute                       ↢
read                          ↢
```

Listing 4.6 – Exemple de table de changement d'état pour les transferts d'informations.

Par exemple l'arc entre $user\_info\_t$ et $webserv\_d$, correspond à un transfert d'informations

légal $sc_{user\_info\_t} > sc_{webserv\_d}$ entre ces deux contextes. Cet arc correspond ainsi à l'interaction $sc_{webserv\_d} \xrightarrow{read} sc_{user\_info\_t}$.

L'existence d'un chemin entre les nœuds $user\_info\_t$ et $admin\_d$ implique que de l'information contenue dans le fichier typé par $user\_info\_t$, appartenant normalement aux utilisateurs, peut être transférée vers le contexte $admin\_d$. Par exemple, le chemin

$$sc_{user\_info\_t} \longmapsto_{read} sc_{user\_d} \longmapsto_{write} sc_{temp\_t} \longmapsto_{read} sc_{admin\_d}$$

4.2.6 Graphe de transitions

Le *graphe de transitions* est une représentation de l'ensemble des *transitions* de contextes autorisées par la politique de sécurité. Un arc $e = sc_1 \longmapsto sc_2$ entre deux nœuds sc_1 et sc_2 correspond à une transition (changement de contexte) valide d'un contexte sc_1 vers un contexte sc_2. Cet arc est étiqueté par l'ensemble des opérations pouvant réaliser cette *transition*. De ce fait, ce graphe ne considère que les opérations élémentaires ayant pour effet une transition d'un contexte vers un autre.

4.2.6.1 Construction du graphe

Comme nous l'avons vu dans la section 3.1.3.3 (page 60), une opération de type *transition*, d'un contexte sc_1 ver un sc_2, a pour *effet* l'obtention du contexte sc_2 par sc_1. Ainsi, une opération de type *transition* a pour effet de changer l'état du contexte sc_2 de *non-utilisé* à *utilisé*, ce contexte représentant le nouvel état du processus ayant effectué le changement de contexte.

De ce fait, chaque opération élémentaire vérifiant la fonction `is_transition` (def. 3.1.3) est de type $\rightarrowtail$. Cette fonction est donc utilisée pour construire la table de changement d'état T_{trans}. A partir de la définition de cette table T_{trans}, nous utilisons l'algorithme général de construction (algo. 2, page 2) afin de construire le *graphe de transitions* en fonction d'un graphe d'interactions. Notons qu'étant donné que seuls les contextes de sécurité sujet, correspondant à des processus, peuvent changer de contexte, ce graphe ne contient que des nœuds correspondant à des contextes de sécurité sujet.

Par construction, le *graphe de transitions* représente l'ensemble des transitions présentes dans une politique de contrôle d'accès. De plus, d'après la définition d'une séquence de transitions (def. 3.2.5), une telle séquence n'est constituée que d'interactions du type transition. Par application du théorème 4.2.1 d'équivalence entre un chemin et une séquence, nous énonçons la propriété d'équivalence entre un chemin du graphe de transitions et une séquence de transitions.

Propriété 4.2.5 *Soit $\mathcal{POL}$ une politique de contrôle d'accès et $G_{c_{trans}}$ le graphe de transition qui lui est associée.*
Il existe un ***chemin*** $sc_1 \longmapsto_{eo_1} \ldots \longmapsto_{eo_{n-1}} sc_n$ *dans $G_{c_{trans}}$ si et seulement si il existe une* ***séquence de transitions*** $sc_1 \Rightarrow_{trans} sc_n$ *dans $\mathcal{POL}$.*

Toute séquence $sc_{source} \Rightarrow_{trans} sc_{cible}$ correspond donc à un chemin de sc_{source} vers sc_{cible} dans le graphe de transitions. Ainsi, le graphe de transitions permet d'énumérer l'ensemble des *séquences de transitions* présent dans une politique de contrôle d'accès.

4.2.6.2 Exemple

La figure 4.5 représente le graphe de transitions associé à la politique de contrôles d'accès du listing 4.1. La table T_{trans}, utilisée par l'algorithme général de construction (algo. 2, page 95), correspondant à cette politique, est contenue dans le listing 4.7. Dans cette table, seule l'opération élémen-

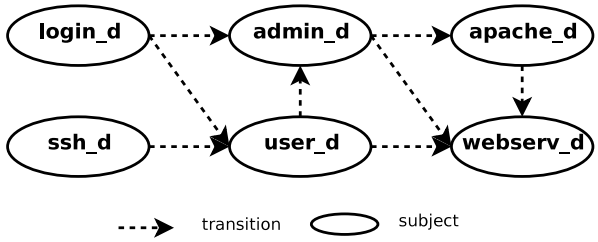


FIG. 4.5 – Graphe de transitions correspondant au listing 4.1

taire $transition$ est une opération vérifiant la fonction `is_transition`. Notons que dans des exemples plus complexes de politiques, plusieurs opérations élémentaires peuvent permettre une transition.

`transition`	$\rightarrowtail$

Listing 4.7 – Exemple de table de changement d'état pour les opérations de transition.

Un arc dans ce graphe, par exemple entre sc_{login_d} et sc_{user_d}, correspond à une transition de contexte légale ($sc_{login\_d} \xrightarrow{trans} sc_{user\_d}$).

Un *chemin* dans ce graphe, par exemple entre ssh_d et $webserv\_d$, correspond à *une séquence de transitions*, permettant à ssh_d d'obtenir le contexte $webserv\_d$. Rappelons qu'une séquence de transitions correspond aux divers changements de contexte effectués par un processus lors de son exécution. Par exemple, un utilisateur qui se connectera via *ssh* puis s'authentifiera en tant qu'administrateur pour invoquer le service Web, engendrera la séquence de transitions :

$$sc_{ssh\_d} \Rightarrow_{trans} sc_{webserv\_d} =$$

$$\{sc_{ssh\_d} \xrightarrow{trans} sc_{user\_d}, sc_{user\_d} \xrightarrow{trans} sc_{admin\_d}, sc_{admin\_d} \xrightarrow{trans} sc_{webserv\_d}\}$$

4.3 Algorithme d'énumération

L'objectif de la construction des *graphes de dépendance causale* est de permettre l'énumération des *interactions* ou des *séquences* violant des propriétés de sécurité exprimées dans notre langage de description des activités. Nous utilisons ainsi les graphes définis dans la section précédente comme support pour l'énumération des différentes classes d'activités (interactions, séquences et corrélations). Dans cette section, nous présentons les algorithmes généraux utilisés pour énumérer ces différentes classes d'activités.

Comme le montre la table 4.1, il y a trois types de dépendance causale et trois types de séquences correspondant à trois graphes. Les graphes de dépendance définis dans la section précédente et les propriétés qui en découlent permettent d'établir que chaque type d'interaction (cf. section 3.1.3, page 57) et chaque type de séquence (cf. section 3.2, page 60) correspondent à un arc ou un chemin dans un des graphes de dépendance causale. Nous reprenons, dans la table 4.1, le type de chaque arc et de chaque chemin en fonction du graphe. Nous distinguons, dans ce tableau, le *graphe d'interactions*, qui ne prend pas en compte la dépendance causale entre les opérations, et les trois graphes de dépendance causale. Ainsi, le *graphe d'interactions* ne permet d'énumérer que les interactions définies dans une

Type de Graphe	Arc	Chemin
Dépendance causale	*Dépendance causale (2 arcs)* $it_1 \twoheadrightarrow it_2$	Séquence d'interactions $sc_{source} \Rightarrow sc_{cible}$
Flux d'informations	Transfert d'informations $sc_1 > sc_2$	Flux d'informations $sc_{source} \gg sc_{cible}$
Transition	Transition $sc_1 \xrightarrow{trans} sc_2$	Séquence de transitions $sc_{source} \Rightarrow_{trans} sc_{cible}$
Graphe d'interactions	Interaction $sc_1 \xrightarrow{eo} sc_2$	

TAB. 4.1 – Synthèse des opérations élémentaires et séquences obtenues selon le type de graphe

politique de contrôle d'accès. En effet, un chemin dans ce graphe correspond à une fermeture transitive quelconque d'interactions qui n'entre pas en jeu dans la définition des propriétés de sécurité. Nous n'utiliserons donc pas ce type de séquence dans la suite.

Dans cette section, nous proposons un algorithme d'énumération des interactions dans la partie 4.3.1 et un algorithme d'énumération des séquences dans la partie 4.3.2. Finalement, nous utilisons ces deux algorithmes pour définir deux algorithmes d'énumération de corrélations spécifiques (accès à un privilège, accès à l'information) dans la partie 4.3.3.

4.3.1 Interaction

Nous proposons, tout d'abord, la fonction `enum_interaction`, définie dans l'algorithme 3, qui permet d'énumérer l'ensemble des *interactions* disponibles entre deux contextes sc_1 et sc_2. Cette fonction prend donc en paramètre deux contextes (sc_1 et sc_2) et un graphe G. Pour chaque opération eo, appartenant à la valuation de l'arc (fonction `valuation`, page 93) entre sc_1 et sc_2, une interaction it est alors construite et ajoutée à un ensemble INT. Finalement, cette fonction retourne l'ensemble INT contenant alors l'ensemble des interactions définies entre sc_1 et sc_2.

```
enum_interaction(sc1, sc2, G)
début
    INT := ∅;
    pour chaque eo ∈ valuation(sc1, sc2, G) faire
        INT := INT ∪ ( ordre_inverse(sc1, sc2, eo, G), {eo});
    fin
    Retourne INT;
fin
```

Algorithme 3 : Algorithme d'énumération des interactions

La conversion d'un arc en une interaction est réalisée par la fonction `ordre_inverse`. En effet, chaque arc représente le sens du changement d'état d'une opération, nous avons donc besoin de retrouver l'interaction qui correspond à ce changement (les deux contextes peuvent alors être invervés).

Définition 4.3.1 `ordre_inverse` *est une fonction qui prend en paramètres deux contextes sc_1 et sc_2, une opération élémentaire eo et un graphe G. Cette fonction renvoie alors le couple (sc_s, sc_d) qui correspond à l'inverse de l'effet de eo dans la table de changement de contexte utilisé pour construire G.*

Suivant le graphe G utilisé, l'algorithme 3 retourne des activités du type :

- interactions $it = sc_1 \xrightarrow{eo} sc_2$, pour le *graphe d'interactions* G_{int} ;
- interactions ayant un effet sur sc_2, pour le *graphe de dépendance causale* $G_{c_{dep}}$;
- transferts d'informations $sc_1 > sc_2$, pour le *graphe de flux d'informations* $G_{c_{inf}}$;
- transitions $sc_1 \xrightarrow{trans} sc_2$, pour le *graphe de transitions* $G_{c_{trans}}$.

Cette fonction est ainsi utilisée pour énumérer la classe d'activité correspondant aux *interactions*.

4.3.2 Séquence

Nous avons montré dans la section 4.2 qu'une *séquence* est équivalente à un *chemin* dans un graphe de dépendance causale. De plus les propriétés énoncées dans cette section font apparaître une relation entre la catégorie du graphe et le type de séquence obtenu. Ainsi, en fonction du graphe de dépendance causale utilisé, un chemin est équivalent à :

- Une séquence d'interactions $sc_{source} \Rightarrow sc_{cible}$, pour le *graphe de dépendance causale* (prop. 4.2.3) ;
- Un flux d'informations $sc_{source} \gg sc_{cible}$, pour le *graphe de flux d'informations* (prop. 4.2.4) ;
- Une séquence de transitions $sc_{source} \Rightarrow_{trans} sc_{cible}$, pour le *graphe de transitions* (prop. 4.2.5).

De ce fait, l'énumération d'une catégorie de séquences utilise le graphe de dépendance causale associé à cette catégorie (cf. table 4.1). Par exemple, pour énumérer toutes les *séquences d'interactions* possibles entre deux contextes sc_{source} et sc_{cible}, nous utiliserons le *graphe de dépendance causale*.

Dans notre implantation, nous utilisons, comme algorithme d'énumération des chemins, l'algorithme des *K-plus court chemins* (en anglais *KShortestPath*) correspondant à la signature de fonction `kshortestpath(`sc_1, sc_2, G`)`. Cette fonction renvoie alors l'ensemble des K chemins entre sc_1 et sc_2 dans le graphe G. Comme le précise la remarque 4.3.1, avec une valeur de K assez grande, nous obtenons en général tous les chemins possibles dans une politique de contrôle d'accès. Dans le cas contraire, K est incrémenté.

Nous définissons alors la fonction `enum_sequence` d'énumération des séquences entre deux contextes. Cette fonction prend en paramètres deux contextes (sc_{source} et sc_{cible}) et un graphe G_c. Elle renvoie alors l'ensemble des séquences possible de sc_{source} vers sc_{cible} par énumération des chemins entre ces deux nœuds dans le graphe G_c.

```
enum_sequence(sc_source, sc_cible, G_c)
début
    SEQ := ∅;
    CHEMIN := kshortestpath(sc_source, sc_cible, G_c) ;
    pour chaque c ∈ CHEMIN faire
        Liste seq;
        pour chaque e = (sc_1, sc_2, eo) ∈ c faire
            seq := seq .push(( ordre_inverse(sc_1, sc_2, eo, G_c), eo) ) ;
        fin
        SEQ := SEQ ∪ seq
    fin
    Retourne SEQ;
fin
```

Algorithme 4 : Algorithme d'énumération des séquences

Cette fonction utilise donc `kshortestpath` afin d'énumérer l'ensemble des chemins existants entre les deux contextes d'entrée (sc_{source} et sc_{cible}) dans le graphe G_c. Chaque chemin est ensuite converti en une séquence seq puis ajouté à l'ensemble des séquences SEQ. Une séquence est ainsi représentée par une liste chaînée d'interactions. Afin d'alléger la définition de cet algorithme, nous supposons que la fonction `kshortestpath` retourne des chemins constitués d'arcs contenant une seule opération : $sc_1 \longmapsto_{eo} sc_2$. Ainsi, pour un arc étiqueté par deux opérations eo_1 et eo_2, cette version retournera

deux chemins $sc_1 \longmapsto_{eo_1} sc_2$ et $sc_1 \longmapsto_{eo_2} sc_2$.

Suivant le type de graphe utilisé, la fonction `enum_sequence` renverra donc l'ensemble des :

- séquences d'interactions $sc_{source} \Rightarrow sc_{cible}$ pour le graphe de dépendance causale $G_{c_{dep}}$;
- flux d'informations $sc_{source} \gg sc_{cible}$ pour le graphe de flux d'informations $G_{c_{inf}}$;
- séquences de transitions $sc_{source} \Rightarrow_{trans} sc_{cible}$ pour le graphe de transitions $G_{c_{trans}}$.

Cette fonction est ainsi utilisée pour énumérer la classe d'activité correspondant aux *séquences*.

Remarque 4.3.1 *L'analyse des graphes obtenus à partir de politiques de contrôle d'accès, telle que* SEL*inux ou* GRSECURITY*, ont conduit à deux observations : 1) ces graphes ne contiennent généralement pas plus de 2000 nœuds et, 2) le diamètre de ces graphes n'excède pas une valeur de 15. De ce fait, l'expérimentation faite avec l'algorithme* `kshortestpath` *montre que cet algorithme permet d'énumérer l'ensemble des chemins entre deux nœuds en quelques secondes en prenant comme valeur maximale de recherche* $k := 5,00,000$ *(cf. chapitre 6). C'est pourquoi nous utilisons cet algorithme pour énumérer l'ensemble des chemins existant entre deux nœuds. Notons finalement que la définition de l'algorithme 4 ne se restreint pas à cet algorithme de recherche de chemins.*

4.3.3 Corrélation

Une corrélation est constituée de plusieurs séquences ou interactions, pouvant être de catégorie différente, combinées à l'aide d'opérateurs ($\circ, \wedge, \vee$) (cf. section 3.2.4, page 67). La définition de l'algorithme général d'énumération des corrélations utilise donc les deux algorithmes précédents pour énumérer les activités correspondant à une corrélation. En effet, cet algorithme correspond à l'implantation de notre langage de description d'activité et nous venons de founir les deux algorithmes d'énumération de ce langage. L'algorithme d'énumération des corrélations correspond donc à une simple implantation de notre grammaire en utilisant les deux algorithmes précédents. L'algorithme d'énumération des corrélations énumère ainsi chaque terminal d'une corrélation afin de recomposer l'ensemble des activités possibles. Par soucis de concision, nous ne détaillerons pas cet algorithme dans ce manuscrit. Cependant, étant donné que la formalisation des propriétés de sécurité présentée dans la section 3.3 fait apparaître deux types de corrélation spécifiques, nous proposons, dans la suite de cette section, deux algorithmes d'énumération spécifiques pour les *accès aux privilèges* et les *accès à l'information*. Ces algorithmes sont en effet suffisants pour définir ces propriétés de securité.

4.3.3.1 Accès à un privilège

Nous avons défini, dans la partie 3.2.4.3 (page 69), la corrélation *d'accès à un privilège* eo comme la composition d'une *séquence d'interactions* et d'une *interaction* (contenant eo). L'opérateur de composition ($\circ$) implique que la séquence d'interactions doit être réalisée avant l'interaction finale contenant eo. Ce type de composition peut donc être vu comme une *séquence d'interactions* où la dernière interaction correspond au privilège eo.

Nous proposons donc la fonction `enum_accespriv` (algo. 5) d'énumération des *accès à un privilège* eo. Cette fonction prend en paramètres deux contextes (sc_{source} et sc_{cible}), le privilège concerné (eo), le graphe d'interactions (G_{int}) et un graphe de dépendance causale (G_c). Cette fonction retourne alors

l'ensemble des "séquences" (SEQ) d'accès au privilège eo.

```
enum_accespriv(sc_source, sc_cible, eo, G_int, G_c)
début
    SEQ := ∅;
    pour chaque sc ∈ voisins(sc_cible) faire
        pour chaque it ∈ enum_interaction(sc, sc_cible, G_int) faire
            si it == (sc, sc_cible, eo) alors
                pour chaque seq ∈ enum_sequence(sc_source, sc, G_c) faire
                    SEQ := SEQ ∪ (seq.push(it))
                fin
            fin
        fin
    fin
    Retourne SEQ;
fin
```

Algorithme 5 : Algorithme d'énumération des accès à un privilège

Cette fonction recherche donc, dans le graphe d'interactions (G_{int}), toute *interaction* (it) permettant d'utiliser eo sur sc_{cible} (en considérant les nœuds voisins de sc_{cible}) ; puis, dans le graphe G_c, toute *séquence d'interactions* (seq) entre sc_{source} et le contexte source (sc) de l'interaction it. Pour chaque séquence d'interactions possible seq, une nouvelle "séquence" (composée de seq puis it) est alors ajoutée à l'ensemble SEQ. De ce fait, chaque accès aux privilèges de l'ensemble SEQ correspond à une séquence d'interaction où la dernière interaction correspond à l'opération eo.

Notons finalement que cette fonction permet d'énumérer les corrélations d'*accès à un privilège*, si le graphe de dépendance (G_c) correspond au *graphe de dépendance causale*, et les corrélations d'*accès à un privilège par transitions*, si le graphe de dépendance correspond au *graphe de transitions*.

4.3.3.2 Accès à l'information

Nous avons défini, dans la partie 3.2.4.4 (page 70), la corrélation d'accès à une information sc_{info} comme la combinaison d'une *séquence d'interactions* et d'un *flux d'informations*. L'opérateur de composition utilisé ($\wedge$) n'implique aucune dépendance temporelle entre ces deux séquences.

Ainsi, afin d'énumérer une corrélation d'*accès à l'information* ($sc_{acces} \Rightarrow_{trans} sc_{inter} \wedge sc_{info} \gg sc_{inter}$), nous définissons l'algorithme 6. Cet algorithme permet de contruire l'ensemble des corrélations de type accès à une information d'un contexte sc_{acces} sur un contexte sc_{info}. Les deux terminaux de cette corrélation étant des séquences, cet algorithme utilise donc l'algorihme d'énumération des séquences. Il retourne alors un ensemble d'accès à l'information COR. Nous utilisons alors la notation $[seq_1, \wedge', seq_2]$ pour représenter deux séquences seq_1 et se_1 et l'opérateur de corrélation $\wedge$.

```
enum_accesinf(sc_acces, sc_info, G_int(N, E), G_flux, G_c)
début
    COR := ∅;
    pour chaque sc_inter ∈ N faire
        pour chaque ∀seq_1 ∈ enum_sequence(sc_acces, sc_inter, G_c) faire
            pour chaque ∀seq_2 ∈ enum_sequence(sc_info, sc_inter, G_flux) faire
                COR := COR ∪ [seq_1, ∧', seq_2]
            fin
        fin
    fin
    Retourne COR;
fin
```

Algorithme 6 : Algorithme d'énumération des accès à l'information

Notons finalement que cet algorithme permet d'énumérer les corrélations d'*accès à l'information*, si le graphe de dépendance (G_c) correspond au *graphe de dépendance causale*, et les corrélations

d'*accès à l'information par transitions*, si le graphe de dépendance correspond au *graphe de transitions*.

4.4 Analyse des propriétés de sécurité

Nous avons défini dans la section 3.3, un ensemble de propriétés de sécurité qui peuvent être appliquées sur un système. Lorsqu'elles sont respectées, ces propriétés de sécurité permettent alors de garantir la *confidentialité* ou l'*intégrité* du système. Nous avons formalisé chaque propriété de sécurité comme un ensemble de conditions, où chaque condition repose sur l'absence ou la présence d'interactions, de *séquences* ou de corrélations. Nous dirons ainsi qu'une *propriété de sécurité* peut être violée si il existe *activité* ne vérifiant pas cet ensemble de conditions.

Dans ce chapitre, nous avons proposé une modélisation d'une politique de contrôle d'accès sous forme de graphes. De plus, nous avons montré que chaque *interaction* ou chaque *séquence* d'une politique de contrôle d'accès correspond à un *arc* ou à un *chemin* dans l'un des graphes. Dans la section précédente, nous avons proposé des algorithmes généraux permettant d'énumérer toutes les interactions et toutes les séquences présentes dans un graphe et donc dans une politique de contrôle d'accès.

Dans cette partie, nous proposons d'appliquer ces algorithmes d'énumération afin de calculer l'ensemble des interactions, séquences ou corrélations violant les propriétés de sécurité définies dans la section 3.3. Chacune de ces violations, que nous appellerons *signature d'attaque*, correspond alors à une *activité* possible qui permet de violer une propriété de sécurité, par abus de la politique de contrôle d'accès. Chaque *signature d'attaque* appartient alors à l'une des *classes d'activités*.

Ainsi, pour chaque propriété de sécurité exprimée dans notre langage, nous associons une fonction, que nous appelons une *règle de détection*. Chaque règle de détection permet d'énumérer l'ensemble des violations d'une propriété de sécurité et elle renvoie donc un ensemble de signatures d'attaques. Nous proposerons, dans le chapitre 5, un système de détection d'intrusions, permettant de détecter l'apparition de ces signatures et donc de garantir le respect d'un ensemble de propriétés de sécurité.

Dans la suite de cette section, nous détaillons le fonctionnement de chaque *règle de détection*, les paramètres de ces fonctions et la classe des signatures renvoyées.

4.4.1 Intégrité

Le premier ensemble de *règles de détection* correspond aux propriétés d'intégrité définies dans la section 3.3.1. Chaque règle de détection renvoie un ensemble de signatures correspondant à des modifications du système autorisées par la politique de contrôle d'accès, mais violant ces propriétés d'intégrité. Le tableau B.1 (page.190) reprend, pour chaque règle de détection, les conditions recherchées, le type de graphe utilisé et le type de signature renvoyée. Dans cette partie, nous détaillons l'algorithme associé à chaque règle de détection. Lors de la définition de ces algorithmes, nous considérons que le graphe d'interactions (G_{int}) et les graphes de dépendance causale (G_{dep}, G_{flux} et G_{trans}) sont définis en tant que variables globales. Lorsqu'un graphe G_c est passé en paramètre à un algorithme, cela indique que l'algorithme peut être paramétré par un graphe spécifique dans le but d'énumérer une catégorie de séquences. Par exemple, ce paramètre permet d'énumérer les *accès à un privilège* ou de se restreindre aux *accès à un privilège par transition*.

4.4.1.1 Intégrité des objets

La *règle de détection* `integrity` permet d'énumérer toutes les violations de la propriété d'*intégrité des objets* (cf. prop. 3.3.1, page 74). Rappelons que cette propriété a pour objectif de prévenir toute modification, directe ou par séquence, d'un objet (sc_2) par un sujet (sc_1). Cette règle prend donc en paramètres deux contextes sc_1 et sc_2 et renvoie un ensemble de signatures constitué d'*interactions* et de *accès à un privilège*. Chaque signature correspond alors à une violation possible de la propriété d'intégrité.

La propriété d'*intégrité des objets* étant composée de deux conditions, nous décomposons la règle de détection `integrity` en deux fonctions. La première fonction, `integrity_it`, énumère dans le graphe d'interactions l'ensemble des arcs entre sc_1 et sc_2 étiquetés par une opération élémentaire de type *modification*.

```
integrity_it(sc_1, sc_2)
début
    IT := ∅;
    pour chaque it ∈ enum_interaction(sc_1, sc_2, G_int) faire
        si is_write_like(it.eo) alors
            IT := IT ∪ it;
        fin
    fin
    Retourne IT;
fin
```

La seconde fonction, `integrity_seq`, énumère l'ensemble des *accès au privilège* de *modification* entre sc_1 et sc_2. Cette fonction correspond alors à la condition (3.5) de la propriété d'intégrité des objets. Le paramètre G_c correspond soit au *graphe de dépendance causale*, soit au *graphe de transitions*. Dans le premier cas, cette règle renvoie un ensemble d'*accès au privilège* et dans le second, un ensemble d'*accès au privilège par transition*.

```
integrity_seq(sc_source, sc_cible, G_c)
début
    SEQ := ∅;
    pour chaque eo ∈ IS faire
        si is_write_like(eo) alors
            pour chaque it ∈ enum_accespriv(sc_source, sc_cible, eo, G_int, G_c) faire
                SEQ := SEQ ∪ seq;
            fin
        fin
    fin
    Retourne SEQ;
fin
```

Nous définissons ainsi la règle de détection `integrity` qui correspond à la propriété d'intégrité des objets. Cette règle de détection prend alors en paramètres deux contextes de sécurité (sc_{source} et sc_{cible}) et applique les deux fonctions précédentes. Elle renvoie ainsi un ensemble d'interactions et d'accès à un privilège.

```
integrity(sc_source, sc_cible, G_c)
début
    IT := integrity_it(sc_source, sc_cible);
    SEQ := integrity_seq(sc_source, sc_cible, G_c);
    Retourne IT ∪ SEQ;
fin
```

Les contextes sc_{source} et sc_{cible} peuvent, par exemple, être définis par un administrateur qui désirerait détecter les modifications possibles de certains fichiers critiques du système par les contextes

utilisateurs.

La propriété d'intégrité des objets d'un système (prop. 3.3.2) est alors appliquée par appels successifs à cette règle de recherche (un appel par couple passé en paramètres). Ceci permet alors, via la définition de deux ensembles de contextes SC_{source} et SC_{cible}, d'énumérer toutes les violations possibles des éléments de SC_{source} sur les éléments de SC_{cible}.

Une application de cette seconde règle consiste à classer les contextes de sécurité sujets et objets en deux catégories *système* et *utilisateur*. Il est alors possible de détecter toutes les modifications faites par des sujets utilisateurs sur les objets système et inversement.

Exemple Pour l'exemple de politique de contrôle d'accès (cf. listing 4.1, page 92), nous pouvons définir les ensembles de contextes suivants :

$$SC_{user} = \{sc_{user\_d}, sc_{user\_info\_t}\}$$

$$SC_{admin} = \{sc_{admin\_d}, sc_{admin\_info\_t}\}$$

$$SC_{apache} = \{sc_{apache\_d}, sc_{webserv\_d}, sc_{apache\_conf\_t}, sc_{var\_www\_t}\}$$

L'ensemble SC_{user} correspond alors au "domaine" utilisateur, l'ensemble SC_{admin} correspond au "domaine" administrateur et l'ensemble SC_{apache} au "domaine" du serveur Web. Nous définissons tout d'abord deux règles afin de garantir que les domaines utilisateur et serveur Web ne peuvent pas porter atteinte à l'intégrité du domaine administrateur (ligne 1). De plus, la ligne 2 permet de garantir que les domaines utilisateurs et Web serveur ne peuvent se porter atteinte mutuellement :

integrity(SC_{user}, SC_{admin}, G_{trans})	integrity(SC_{apache}, SC_{admin}, G_{trans})
integrity(SC_{apache}, SC_{user}, G_{trans})	integrity(SC_{user}, SC_{apache}, G_{trans})

Prenons l'exemple de la première règle de recherche et du graphe contenu dans la figure 4.3 (page 101). L'utilisateur ne pouvant modifier directement les données du domaine administrateur, cette règle ne renvoie aucune interaction. Par contre, elle renvoie la séquence suivante :

$$sc_{user\_d} \xrightarrow{trans} sc_{admin\_d} \xrightarrow{write} sc_{admin\_info\_t}$$

Cet *accès à un privilège* autorise la modification des fichiers d'informations de l'administrateur par l'utilisateur via une transition.

4.4.1.2 Biba

La *règle de détection* `int_biba` permet d'énumérer toutes les violations de la propriété d'*intégrité de* BIBA (cf. prop. 3.3.3, page 75). Cette règle requiert, comme paramètres, un ensemble de contextes SC_{int} possédant un attribut de type *intégrité* et elle renvoie un ensemble de signatures constitué d'*interactions*. Rappelons que cette propriété repose sur la définition de *niveau d'intégrité* et trois lois régissant les accès sur le système :

1. pour qu'un sujet sc_1 ait accès en lecture à un objet sc_2, son niveau d'intégrité $int(sc_1)$ doit être inférieur ou égal au niveau d'intégrité $int(sc_2)$ de l'objet ;
2. pour qu'un sujet sc_1 ait accès en écriture à un objet sc_2, son niveau d'intégrité doit être supérieur ou égal au niveau d'intégrité de l'objet ;
3. pour qu'un sujet sc_1 puisse invoquer un sujet sc_2, son niveau d'intégrité doit être supérieur ou égal au niveau d'intégrité du sujet invoqué.

Ces trois conditions permettent de définir la *règle de détection* `int_biba` :

```
int_biba(SC_int)
début
    IT := ∅;
    pour chaque sc_1 ∈ SC_int faire
        pour chaque sc_2 ∈ SC_int faire
            pour chaque it ∈ enum_interaction(sc_1, sc_2, G_int) faire
                si is_read_like(it.eo) et int(sc_1) > int(sc_2) alors
                    IT := IT ∪ it;
                fin
                si is_write_like(it.eo) et int(sc_1) < int(sc_2) alors
                    IT := IT ∪ it;
                fin
                si is_execute_like(it.eo) et int(sc_1) < int(sc_2) alors
                    IT := IT ∪ it;
                fin
            fin
        fin
    fin
    Retourne IT;
fin
```

Cette règle de détection renvoie donc l'ensemble des interactions, c'est-à-dire des arcs dans le *graphe d'interactions*, en contradiction avec une des trois lois de BIBA.

Notons que la définition de cette *règle de détection* permet d'appliquer la propriété d'intégrité de BIBA à un sous-ensemble de contextes du système SC_{int}. Ainsi, il est possible d'appliquer cette propriété à l'ensemble des contextes utilisateurs pour assurer l'intégrité des données utilisateurs, ou sur un ensemble de contextes système pour assurer l'intégrité des fichiers de configuration.

Exemple Pour l'exemple de politique de contrôle d'accès (cf. listing 4.1, page 92), nous définissons les niveaux d'intégrité suivants :

Contexte(s)	Niveau d'intégrité
$sc_{user\_info\_t}$, $sc_{admin\_info\_t}$	0
$sc_{apache\_conf\_t}$	1
$sc_{var\_www\_t}$, sc_{apache_d}, $sc_{webserv\_d}$	2

Pour cet exemple, aucune règle de la politique de contrôle d'accès n'est en contradiction avec la propriété d'intégrité de BIBA, ainsi la règle `int_biba(SC)` ne renvoie aucune signature.

4.4.1.3 Intégrité des sujets

La *règle de détection* `int_subject` permet d'énumérer toutes les violations de la propriété d'*intégrité d'un sujet* (cf. prop. 3.3.4, page 76). Cette règle requiert, comme paramètres, un contexte sc_1 et elle renvoie un ensemble de signatures constitué de *séquences* de deux interactions. Cette règle énumère tous les contextes scs partageant un objet sco avec sc_1. Chaque couple d'interactions it_1 et

	write	read	setattr	link	rename	unlink
write	c	c	c	c	c	c
read	√	√	√	√	√	√
setattr	c	c	c	c	c	c
link	c	c	c	c	c	c
rename	c	c	c	c	c	c
unlink	c	c	c	c	c	c

TAB. 4.2 – Table de commutativité pour la classe *file* sous SELinux

it_2 qui ne commutent pas, est alors ajouté à la liste des violations SEQ.

```
int_subject(sc_1)
début
    SEQ := ∅;
    pour chaque sco ∈ SC_O faire
        pour chaque scs ∈ SC_S faire
            pour chaque it_1 ∈ enum_interaction(sc_1, sco, G_int) faire
                pour chaque it_2 ∈ enum_interaction(scs, sco, G_int) faire
                    si ¬commute(it_1, it_2) et sc_1 ≠ scs alors
                        SEQ := SEQ ∪ {it_1, it_2};
                    fin
                fin
            fin
        fin
    fin
    Retourne SEQ;
fin
```

La propriété d'*intégrité des sujets d'un système* peut être vérifiée par appels successifs à la règle de détection `int_subject` (un appel par sujet du système). Cette seconde règle ne prend ainsi aucun paramètre.

Notons finalement que cette règle de détection requiert la définition d'une table de commutativité pour les opérations élémentaires servant de base à la fonction `commute`. La table 4.2 contient un exemple de table pour les opérations élémentaires sur les fichiers pour SELinux. Le symbole √ signifie alors que les deux opérations commutent et un c signifie que les opérations ne commutent pas.

Exemple Si nous reprenons l'exemple de politique de contrôle d'accès (cf. listing 4.1, page 92), la règle `int_subject(`$SC_{systeme}$`)` applique la propriété d'intégrité des sujets à l'ensemble des contextes du système :

$$SC_{systeme} = \{sc_{login\_d}, sc_{ssh\_d}, sc_{apache\_d}, sc_{webserv\_d}\}$$

En utilisant la table 4.2, les contextes sc_{apache_d} et $sc_{webserv\_d}$ peuvent interférer sur le contexte objet $sc_{var\_www\_t}$. Dans ce cas, cette règle générera les signatures suivantes :

$$\{sc_{apache\_d} \xrightarrow{write} sc_{var\_www\_t}, sc_{webserv\_d} \xrightarrow{read} sc_{var\_www\_t}\}$$

$$\{sc_{webserv\_d} \xrightarrow{write} sc_{var\_www\_t}, sc_{apache\_d} \xrightarrow{read} sc_{var\_www\_t}\}$$

4.4.1.4 Intégrité des domaines

La *règle de détection* `int_domain` permet d'énumérer toutes les violations de la propriété d'*intégrité des domaines* (cf. prop. 3.3.7, page 77). Cette règle prend en paramètres un ensemble de contextes TCB_{in} représentant le domaine (TCB) à surveiller. Ainsi, cette règle énumère toutes les interactions possibles entre un élément de ce domaine et le reste du système.

`int_domain`(TCB_{in})
début
  $IT := \emptyset$;
  pour chaque $sc_1 \in TCB_{in}$ **faire**
    pour chaque $sc_2 \notin TCB_{in}$ **faire**
      pour chaque $it_1 \in$ `enum_interaction`(sc_1, sc_2, G_{int}) **faire**
        $IT := IT \cup it$;
      fin
      pour chaque $it_2 \in$ `enum_interaction`(sc_2, sc_1, G_{int}) **faire**
        $IT := IT \cup it$;
      fin
    fin
  fin
  Retourne IT;
fin

Il y aura donc violation de la propriété d'*intégrité des domaines* si l'une des interactions de IT est observée sur le système.

Exemple Si nous reprenons l'exemple de politique de contrôle d'accès (cf. listing 4.1, page 92), la règle `int_domain`(SC_{TCB}) applique la propriété des domaines au domaine "Web" définie par :

$$SC_{TCB} = \{sc_{apache\_d}, sc_{webserv\_d}, sc_{apache\_conf\_t}, sc_{var\_www\_t}, sc_{admin\_info\_t}, sc_{user\_info\_t}\}$$

Ainsi, cette règle générera des signatures telles que :

$$sc_{admin\_d} \xrightarrow{trans} sc_{apache\_d}$$

$$sc_{admin\_d} \xrightarrow{write} sc_{apache\_conf\_t}$$

Comme nous le verrons dans le chapitre 5, cette règle permet, d'isoler le serveur Web `apache`, et ses dépendances (fichiers de configuration et services Web), sans nécessiter la création d'un `chroot`, notre approche permet de créer un TCB constitué de l'ensemble des contextes de ces applications et ainsi de détecter les interactions entre ces contextes et le reste du système.

Notons que certaines interactions ne peuvent être supprimées de la politique car elles sont nécessaires lors de l'initialisation du système. Ainsi, lors de l'initialisation du système, des alertes correspondant à des *faux positifs* seront levées et devront être ignorées. Il peut par exemple s'agir d'interactions liées au processus d'intialisation du système qui a pour tâche d'exécuter tous les services système (et donc le serveur `apache`). Seules les alertes levées après initialisation complète du système devront être prises en compte. Notons que ces faux positifs peuvent être écartés en exécutant la détection de ces signatures après l'initialisation du système.

4.4.2 Confidentialité

Le second ensemble de *règles de détection* correspond aux propriétés de confidentialité définies dans la section 3.3.2. Chaque règle de détection renvoie un ensemble de signatures correspondant à des

accès autorisés par la politique de contrôle d'accès, mais violant ces propriétés de confidentialité. Le tableau B.2 (page. 191) reprend, pour chaque règle de détection, les conditions recherchées, le type de graphe utilisé et le type de signatures renvoyées. Dans cette partie, nous détaillons le fonctionnement de chaque règle de détection.

4.4.2.1 Confidentialité des contextes

La *règle de détection* `confidentiality` permet d'énumérer toutes les violations de la propriété de *confidentialité d'un contexte* (cf. prop. 3.3.9, page 78). Rappelons que cette propriété a pour objectif de prévenir tout accès par un sujet (sc_1) à un objet (sc_2). Cette règle prend donc en paramètres deux contextes sc_1 et sc_2 et renvoie un ensemble de signatures constitué d'*interactions*, d'*accès à un privilège* et d'*accès à l'information*.

La propriété de *confidentialité d'un contexte* étant composée de trois conditions, nous décomposons la règle de détection `confidentiality` en trois fonctions. La première fonction, `confidentiality_it`, énumère l'ensemble des transferts d'informations entre sc_1 et sc_2 en utilisant le *graphe de flux d'informations*. Cette fonction correspond alors à la condition (3.11) de la propriété de confidentialité des contextes.

`confidentiality_it`(sc_1, sc_2)
début
| **Retourne** `enum_interaction`$(sc_2, sc_1, G_{c_{flux}})$;
fin

La seconde fonction, `confidentiality_seq`, énumère l'ensemble des *flux d'informations* de sc_{cible} à sc_{source}. Cette fonction correspond alors à la condition (3.12) de la propriété de confidentialité des contextes.

`confidentiality_seq`$(sc_{source}, sc_{cible})$
début
| **Retourne** `enum_sequence`$(sc_{cible}, sc_{source}, G_{c_{flux}})$;
fin

La dernière fonction, `confidentiality_acces`, énumère l'ensemble des *accès à l'information* de sc_{acces} sur sc_{info}. Cette fonction correspond alors à la condition (3.13) de la propriété de confidentialité des contextes. Le paramètre G_c correspond soit au *graphe de dépendance causale*, soit au *graphe de transitions*. Dans le premier cas, cette règle renvoie un ensemble d'*accès à l'information* et dans le second, un ensemble d'*accès à l'information par transition*.

`confidentiality_acces`$(sc_{acces}, sc_{info}, G_c)$
début
| **Retourne** `enum_accesinf`$(sc_{acces}, sc_{info}, G_{int}, G_{flux}, G_c)$;
fin

Nous définissons ainsi la règle de détection `confidentiality` qui correspond à la propriété de confidentialité d'un contexte sc_{cible} vis à vis d'un autre contexte sc_{source}. Cette règle de détection prend en paramètres deux contextes de sécurité (sc_{source} et sc_{cible}) et applique les trois fonctions

précédentes. Elle renvoie ainsi un ensemble d'interactions et d'accès à un privilège.

`confidentiality`$(sc_{source}, sc_{cible}, G_c)$
début
  $IT :=$ `confidentiality_it`$(sc_{source}, sc_{cible})$;
  $SEQ :=$ `confidentiality_seq`$(sc_{source}, sc_{cible})$;
  $COR :=$ `confidentiality_acces`$(sc_{source}, sc_{cible}, G_c)$;
  Retourne $IT \cup SEQ \cup COR$;
fin

La propriété de confidentialité des contextes d'un système (prop. 3.3.10) est alors appliquée par appels successifs à cette règle de recherche (un appel par couple passé en paramètres). Cette règle de détection peut ainsi prendre comme argument deux ensembles de contextes de sécurité, SC_{source} et SC_{cible}, entre lesquels les flux d'informations sont prohibés. Même si l'application de cet algorithme entraîne un grand nombre d'énumérations, nous verrons, dans le chapitre 6, que les caractéristiques des graphes d'une politique de contrôle d'accès (nombre de nœuds et d'arcs) permettent l'utilisation de ce type de d'énumération.

Par exemple, les fichiers de configuration système contiennent généralement des informations critiques tels que des mots-de-passe administrateur qui peuvent être stockés en clair. Pour éviter que les utilisateurs n'accèdent à ces fichiers, les ensembles de contextes SC_{source} et SC_{cible} peuvent correspondre à l'ensemble des contextes utilisateurs et l'ensemble des contextes système. Cette règle permet ainsi de détecter tout "transfert d'informations" d'un fichier de configuration du système vers l'espace utilisateur.

Exemple Dans l'exemple de politique de contrôle d'accès (cf. listing 4.1, page 92), nous pouvons définir les règles suivantes afin de garantir l'absence de flux d'informations entre les fichiers d'informations et les contextes qui n'en sont pas propriétaires :

$$\texttt{confidentiality}(sc_{admin\_d},\ sc_{user\_info\_t}, G_{trans})$$
$$\texttt{confidentiality}(sc_{user\_d},\ sc_{admin\_info\_t}, G_{trans})$$

La première règle de détection ne renvoie aucune signature de type *interaction* car il n'existe aucun transfert d'informations (direct) entre ces deux contextes. Mais cette règle renvoie des signatures correspondant à des *flux d'informations*, par exemple :

$$sc_{user\_info\_t} > sc_{user\_d} > sc_{admin\_d}$$

$$sc_{user\_info\_t} > sc_{user\_d} > sc_{temp\_t} > sc_{admin\_d}$$

Finalement, cette règle renvoie des signatures correspondant à des *accès à l'information*, par exemple :

$$sc_{admin\_d} \xrightarrow{trans} sc_{webserv\_d}, sc_{user\_info\_t} > sc_{webserv\_d}$$

$$sc_{admin\_d} \xrightarrow{trans} sc_{apache\_d}, sc_{apache\_d} \xrightarrow{trans} sc_{webserv\_d}, sc_{user\_info\_t} > sc_{webserv\_d}$$

Chacune de ces signatures permet à l'administrateur d'obtenir des informations dans le fichier utilisateur.

4.4.2.2 Bell&Lapadula

La *règle de détection* `conf_blp` permet d'énumérer toutes les violations de la propriété de *confidentialité de* BLP (cf. prop. 3.3.11, page 80). Cette règle requiert, comme paramètres, un ensemble de contextes SC_{hab} possédant un attribut de type *habilitation* et elle renvoie un ensemble de signatures constitué d'*interactions*.

```
conf_blp(SC_hab)
début
    IT := ∅;
    SEQ := ∅;
    pour chaque sc1 ∈ SC_hab faire
        pour chaque sc2 ∈ SC_hab faire
            pour chaque it ∈ enum_interaction(sc1, sc2, G_int) faire
                si is_read_like(it.eo) et hab(sc1) < hab(sc2) alors
                    IT := IT ∪ it;
                fin
            fin
            pour chaque seq ∈ enum_sequence(sc2, sc1, G_c_flux) faire
                si hab(sc1) < hab(sc2) alors
                    SEQ := SEQ ∪ seq;
                fin
            fin
        fin
    fin
    Retourne IT ∪ SEQ;
fin
```

La première partie de cette règle de détection permet d'extraire du graphe d'interactions l'ensemble des interactions qui violent la première lois du modèle BLP, c'est-à-dire toute interaction du type lecture entre un contexte sujet scs_1 et un objet sco_1 tel que le niveau d'habilitation du sujet et inférieur au niveau d'habilitation de l'objet.

La seconde partie de cette règle de détection permet d'extraire du graphe de flux d'informations tous les flux d'informations $sc_2 \gg sc_1$ tel que le niveau d'habilitation de la source du flux $hab(sc_1)$ est inférieur au niveau d'habilitation de la cible $hab(sc_2)$. Les signatures validant cette règle correspondent à des *interactions* (de type lecture) et des *flux d'informations*. Cette règle de détection renvoie donc un ensemble de signatures composé d'interactions et un ensemble de signatures composé de flux d'informations.

Notons que tout comme pour la propriété de BIBA, cette règle de détection peut être appliquée à un sous-ensemble de contextes du système SC_{hab}. Ainsi, il est possible d'appliquer cette propriété à l'ensemble des contextes utilisateurs pour assurer la confidentialité des données utilisateurs, ou sur un ensemble de contextes système pour assurer la confidentialité des fichiers de configuration.

Exemple Pour l'exemple de politique de contrôle d'accès (cf. listing 4.1, page 92), nous pouvons définir les niveaux d'habilitation suivants :

Contexte(s)	Niveau d'habilitation
$sc_{apache\_conf\_t}$	0
sc_{apache_d}, $sc_{var\_www\_t}$	1
$sc_{webserv\_d}$, $sc_{user\_info\_t}$, $sc_{admin\_info\_t}$	2

Ainsi, la règle `conf_blp(`SC_{web}`)` applique la propriété de confidentialité au domaine Web :

$$SC_{web} = SC_{apache} \cup \{sc_{user\_info\_t}, sc_{admin\_info\_t}\}$$

Avec cette définition des niveaux d'habilitation, la première condition du modèle BLP n'est pas violée. Mais la seconde condition est violée pour les flux d'informations suivants :

$$sc_{user\_info\_t} > sc_{webserv\_d} > sc_{var\_www\_t}$$

$$sc_{admin\_info\_t} > sc_{webserv\_d} > sc_{var\_www\_t}$$

Ces deux signatures correspondent à des transferts d'informations des fichiers personnels vers les fichiers Web réalisables (valides d'après la définition de la politique de contrôle d'accès). En effet, elles correspondent à un flux d'un objet vers un objet de niveau inférieur

4.4.2.3 Bell&Lapadula restrictif

La *règle de détection* `conf_blpr` permet d'énumérer toutes les violations de la propriété de *confidentialité de* BLP *restrictif* (cf. prop. 3.3.12, page 81). Tout comme la règle précédente, cette règle requiert, comme paramètres, un ensemble de contextes SC_{hab} possédant un attribut de type *habilitation* et elle renvoie un ensemble de signatures constitué d'*interactions*.

`conf_blpr(`SC_{hab}`)`
début
  $IT := \emptyset$;
  pour chaque $sc_1 \in SC_{hab}$ **faire**
    pour chaque $sc_2 \in SC_{hab}$ **faire**
      pour chaque $it \in$ `enum_interaction(`sc_1, sc_2, G_{int}`)` **faire**
        si `is_read_like(it.eo)` **et** $hab(sc_1) < hab(sc_2)$ **alors**
          $IT := IT \cup it$;
        fin
        si `is_add_like(it.eo)` **et** $hab(sc_1) > hab(sc_2)$ **alors**
          $IT := IT \cup it$;
        fin
        si `is_write_like(it.eo)` **et** $hab(sc_1) \neq hab(sc_2)$ **alors**
          $IT := IT \cup it$;
        fin
      fin
    fin
  fin
  Retourne IT;
fin

Nous définissons ainsi trois conditions permettant d'extraire les interactions dans le graphe d'interactions violant les trois lois du modèle BLP restrictif. La première correspond aux interactions de lecture entre sujets et objets dont le niveau d'habilitation du sujet $hab(sc_1)$ est inférieur à celui de l'objet $hab(sc_2)$. La seconde correspond aux interactions d'ajouts entre sujets et objets dont le niveau d'habilitation du sujet $hab(sc_1)$ est supérieur à celui de l'objet $hab(sc_2)$. La dernière correspond aux interactions d'exécution où le niveau d'habilitation du sujet appelant $hab(sc_1)$ est différent de celui de l'appelé $hab(sc_2)$. Chaque signature est alors composée d'une interaction validant l'une des trois conditions précédentes.

4.4.2.4 Cohérence d'accès aux données

La *règle de détection* `conf_data` permet d'énumérer toutes les violations de la propriété de *cohérence d'accès aux données* (cf. prop. 3.3.13, page 81). Cette règle prend en paramètres deux contextes de sécurité sc_{source} et sc_{cible}. Les signatures correspondent alors à des *accès à l'information* possibles

entre ces deux contextes auxquels aucun transfert d'informations directe n'est associé.

```
conf_data(sc_source, sc_cible, G_c)
début
    COMB := ∅;
    pour chaque cmb = enum_accesinf(sc_source, sc_cible, G_int, G_flux, G_c) faire
        si cmb ≠ ∅ et enum_interaction(sc_cible, sc_source, G_c_flux) == ∅ alors
            COMB := COMB ∪ cmb;
        fin
    fin
    Retourne COMB;
fin
```

Cette règle de recherche vérifie que pour chaque accès à l'information cmb, un transfert d'informations est possible. Dans le cas contraire, chaque *accès à l'information* est alors ajouté à l'ensemble des violations ($COMB$).

La propriété de cohérence d'accès aux données d'un système (prop. 3.3.14) est alors appliquée par appels successifs à cette règle de recherche (un appel par couple passé en paramètres). Ceci permet alors, via la définition de deux ensembles de contextes SC_{source} et SC_{cible}, d'énumérer toutes les violations possibles des éléments de SC_{source} sur les éléments de SC_{cible}.

Exemple Pour l'exemple de politique de contrôle d'accès (cf. listing 4.1, page 92), nous pouvons définir les ensembles de contextes suivants :

$$SC_{user} = \{sc_{user\_d}, sc_{user\_info\_t}\}$$

$$SC_{admin} = \{sc_{admin\_d}, sc_{admin\_info\_t}\}$$

La règle `conf_data(`SC_{user}`,` SC_{admin}`)` applique la propriété de cohérence d'accès aux données de l'*administrateur* pour les *utilisateurs*. Ainsi, l'utilisateur n'ayant pas un accès direct aux fichiers d'informations $sc_{admin\_info\_t}$, la corrélation suivante sera considérée comme illégale et correspondra donc à une signature :

$$sc_{user\_d} \xrightarrow{trans} sc_{webserv\_d} \wedge sc_{admin\_info\_t} > sc_{webserv\_d}$$

4.4.3 Abus de privilèges

Le dernier ensemble de *règles de détection* correspond aux propriétés d'abus de privilèges définies dans la section 3.3.3. Chaque règle de détection renvoie un ensemble de signatures correspondant à un abus de privilèges autorisé par la politique de contrôle d'accès. Le tableau B.3 (page 192) reprend, pour chaque règle de détection, les conditions recherchées, le type de graphe utilisé et le type de signature renvoyé. Dans cette partie, nous détaillons le fonctionnement de chaque règle de détection d'abus de privilèges.

4.4.3.1 Séparation de privilèges

La *règle de détection* `duties_sep` permet d'énumérer toutes les violations de la propriété de *séparation de privilèges* (cf. prop. 3.3.16, page 83). Rappelons que cette propriété a pour objectif de prévenir toute *modification* puis *exécution*, directe ou par séquence, d'un objet par un sujet. Chaque

signature est donc constituée d'une séquence de deux interactions ou de deux corrélations d'accès à un privilège.

```
duties_sep(G_c)
début
    SEQ := ∅;
    CMB := ∅;
    pour chaque sc_1 ∈ SC_S faire
        pour chaque sc_2 ∈ SC_O faire
            pour chaque it_1 ∈ enum_interaction(sc_1, sc_2, G_int) faire
                pour chaque it_2 ∈ enum_interaction(sc_1, sc_2, G_int) faire
                    si is_write_like(it_1.eo) et is_execute_like(it_2.eo) alors
                        SEQ := SEQ ∪ {it_1, it_2};
                    fin
                fin
            fin
            pour chaque eo_1 ∈ IS faire
                pour chaque eo_2 ∈ IS faire
                    si is_write_like(eo_1) et is_execute_like(eo_2) alors
                        pour chaque seq_1 ∈ enum_accespriv(sc_1, sc_2, eo_1, G_int, G_c) faire
                            pour chaque seq_2 ∈ enum_accespriv(sc_1, sc_2, eo_2, G_int, G_c) faire
                                CMB := CMB ∪ {seq_1, seq_2};
                            fin
                        fin
                    fin
                fin
            fin
        fin
    fin
    Retourne SEQ ∪ CMB;
fin
```

Chaque signature correspond à la possibilité pour un contexte sujet de modifier puis exécuter un contexte objet, violant ainsi la propriété de séparation de privilèges. La première partie de cette règle analyse le graphe d'interactions afin d'extraire les arcs étiquetés par les opérations élémentaires de modification et d'exécution. La seconde partie de cette règle énumère les accès aux privilèges de modification et d'exécution.

Notons que cette règle peut être raffinée pour prendre en paramètres qu'un ensemble de contextes SC_{daemon} afin de n'appliquer cette propriété qu'à cet ensemble, tel qu'énoncé dans la propriété 3.3.17 (page 83).

Notons finalement que cet algorithme n'est pas celui réellement implanté. Pour faciliter son écriture, dans ce manuscrit de thèse, nous considérons en effet tous les couples possibles d'opérations élementaires lors de la seconde énumération et nous ne considérons pas la combinaison d'un accès à un privilège et d'une interaction.

Exemple Pour l'exemple de politique de contrôle d'accès (cf. listing 4.1, page 92), nous pouvons définir les ensembles de contextes suivants :

$$SC_{systeme} = \{sc_{login\_d}, sc_{ssh\_d}, sc_{apache\_d}, sc_{webserv\_d}\}$$

Ainsi, la règle `duties_sep(`$SC_{systeme}$`)` n'applique cette propriété qu'aux services système. Dans cet exemple de politique de contrôle d'accès, les contextes sc_{apache_d} et $sc_{webserv\_d}$ peuvent violer la propriété de séparation de privilèges. Ainsi les deux signatures suivantes permettront ce type d'abus :

$$sc_{apache\_d} \xrightarrow{write} sc_{var\_www\_t}, sc_{apache\_d} \xrightarrow{execute} sc_{var\_www\_t}$$

$$sc_{webserv\_d} \xrightarrow{write} sc_{var\_www\_t}, sc_{webserv\_d} \xrightarrow{execute} sc_{var\_www\_t}$$

De même pour les combinaisons d'accès aux privilèges :

$$seq_1 = sc_{user\_d} \xrightarrow{trans} sc_{apache\_d}, sc_{apache\_d} \xrightarrow{write} sc_{var\_www\_t}$$

$$seq_2 = sc_{user\_d} \xrightarrow{trans} sc_{apache\_d}, sc_{apache\_d} \xrightarrow{execute} sc_{var\_www\_t}$$

4.4.3.2 Absence de changement de contexte

La *règle de détection* `bad_transition` permet d'énumérer toutes les violations de la propriété d'*absence de changement de contexte* (cf. prop. 3.3.19, page 84). Rappelons que cette propriété a pour objectif de prévenir tout changement d'un contexte sc_1 vers un contexte sc_2. Cette règle prend donc en paramètres deux contextes sc_1 et sc_2 et renvoie un ensemble de signatures constitué de *transitions* et de *séquences de transitions*.

`bad_transition`(sc_1, sc_2)
début
| **Retourne** `enum_interaction`(sc_1, sc_2, G_{trans}) ∪ `enum_sequence`$(sc_1, sc_2, G_{c_{trans}})$;
fin

Exemple Sur un système, nous pouvons interdire que le domaine `administrateur` ou les services Web soit accessibles suite à une connexion `ssh`. Ainsi toutes les opérations de maintenance devront être réalisées localement (via `login`). Si nous reprenons l'exemple de politique de contrôle d'accès (cf. listing 4.1, page 92), les règles suivantes permettent d'appliquer cette propriété au système :

`bad_transition`$(sc_{ssh\_d},\ sc_{admin\_d})$
`bad_transition`$(sc_{ssh\_d},\ sc_{apache\_d})$
`bad_transition`$(sc_{ssh\_d},\ sc_{webserv\_d})$

Par exemple, ces règles renverront :

$$sc_{ssh\_d} \xrightarrow{trans} sc_{user\_d} \xrightarrow{trans} sc_{admin\_d}$$

$$sc_{ssh\_d} \xrightarrow{trans} sc_{user\_d} \xrightarrow{trans} sc_{admin\_d} \xrightarrow{trans} sc_{apache\_d}$$

$$sc_{ssh\_d} \xrightarrow{trans} sc_{user\_d} \xrightarrow{trans} sc_{webserv\_d}$$

4.4.3.3 Exécutables de confiance

La *règle de détection* `tpe` permet d'énumérer toutes les violations de la propriété d'*exécutables de confiance* (cf. prop. 3.3.20, page 84). Rappelons que cette propriété a pour objectif de prévenir toute exécution d'une application non présente dans l'ensemble TPE. Cette règle prend donc en paramètres un ensemble de contextes de confiance TPE et renvoie un ensemble de signatures constitué

d'*interactions*.

```
tpe(TPE)
début
    IT := ∅;
    pour chaque sc1 ∈ SC_S faire
        pour chaque sc2 ∈ SC_O faire
            si sc2 ∉ TPE alors
                pour chaque it ∈ enum_interaction(sc1, sc2, G_int) faire
                    si is_execute(it.eo) alors
                        IT := IT ∪ it;
                    fin
                fin
            fin
        fin
    fin
    Retourne IT;
fin
```

Une signature correspond ainsi à une interaction it dont le contexte cible n'appartient pas à l'ensemble TPE et qui est exécutable.

4.4.3.4 Respect des règles de contrôle d'accès

La *règle de détection* `conformity` permet d'énumérer toutes les violations de la propriété de *respect des règles de contrôle d'accès* (cf. prop. 3.3.21, page 84). Rappelons que cette propriété a pour objectif de prévenir toute exécution d'une interaction non présente dans la politique de contrôle d'accès. Contrairement aux autres propriétés de sécurité, cette règle de détection s'applique sur les traces d'interactions du système. Ainsi, lorsqu'une interaction it est exécutée sur le système (appartenant à l'ensemble des traces T), cette interaction est considérée comme illégale si elle n'appartient pas à la politique (au graphe d'interactions). Cette règle de détection est l'une des plus simples de notre approche mais elle n'en est pas pour autant moins importante. En effet, des tentatives répétées d'interactions illégales sont soit signe d'un problème de la politique de contrôle d'accès, soit des tentatives d'abus de privilèges dues à une intrusion. Cette règle prend donc en paramètres un ensemble de traces d'interactions T et renvoie un ensemble d'alertes constitué d'*interactions*.

```
conformity(T)
début
    IT := ∅;
    pour chaque it ∈ T faire
        si it ∉ G_int alors
            IT := IT ∪ it;
        fin
    fin
    Retourne IT;
fin
```

4.4.4 Complexité des algorithmes d'énumération

Afin d'étudier la complexité des différents algorithmes, nous notons $m = |IV|$ le nombre d'arcs du graphe de dépendance causale, $n = |SC|$ le nombre de contextes de sécurité, k le nombre maximum de chemins recherchés et $op = |IS|$ le nombre d'opérations constituant la politique de contrôle d'accès. L'implantation de l'algorithme d'énumération des chemins (*kshortestpath*) a une complexité de : $O(m + nlog(n) + k)$ (cf. [Eppstein 1997]). L'algorithme d'énumération des interactions `enum_interaction` est ainsi en $O(op)$, l'algorithme d'énumération des séquences `enum_sequence`

Propriété de sécurité	Complexité de l'algorithme d'analyse
Intégrité	
- `integrity`	$O(op + op^2 * n * m + op^2 * n^2 log(n) + n * op^2 * k)$
- `int_biba`	$O(n^2 * op)$
- `int_subject`	$O(n^2 * op^2)$
- `int_domain`	$O(n^2 * op)$
Confidentialité	
- `confidentiality`	$O(op + m + nlog(n) + k + n * (m + nlog(n) + k)^2)$
- `conf_blp`	$O(n^2 * op + m * n^2 + n^3 log(n) + n^2 * k))$
- `conf_blpr`	$O(n^2 * op + m * n^2 + n^3 * log(n) + n^2 * op)$
- `conf_data`	$O(n * op^2 * m + n^2 * op^2 log(n) + k * n * op^2)$
Abus de privilèges	
- `duties_sep`	$O(n^2 * op^2 + n^3 * (m + nlog(n) + k)^2)$
- `bad_transition`	$O(op + m + nlog(n) + k)$
- `tpe`	$O(n^2 * op)$

TAB. 4.3 – Complexité des algorithmes d'énumération

est en $O(m + nlog(n) + k)$ (i.e. la même complexité que l'algorithme *kshortestpath*). Les algorithmes d'énumération d'accès à un privilège et d'accès à l'information sont respectivement en $O(n * op * m + op * n^2 log(n) + n * op * k)$ et $O(n * (m + nlog(n) + k)^2)$.

La table 4.3 donne la complexité des algorithmes d'énumération associés à chaque propriété. Nous pouvons remarquer que les deux algorithmes ayant la complexité la plus importante sont ceux associés aux propriétés d'intégrité et de confidentialité générale ainsi que l'algorithme de cohérence d'accès aux données. Notons que la complexité de ces algorithmes prend en compte l'énumération de l'ensemble des contextes, alors que ces règles sont en fait appliquées sur un sous-ensemble de contexte du système. De plus, la plupart des boucles imbriquées ne sont pas réalisées grâce à la présence de test logique avant chaque boucle, comme dans l'algorithme `conf_data`, ce qui permet de réduire énormément le nombre d'appel aux algorithmes d'énumération des séquences et des corrélations. Finalement, nous verrons dans le chapitre 6, que les politiques de contrôle d'accès étudiées ont pour caractéristiques $op < 200$,$n < 3.500$ et $m < 400.000$. Ces caractéristiques nous ont permis d'utiliser tous ces algorithmes sur des politiques réelles, malgré la complexité de certains algorithmes.

4.5 Conclusion

Dans ce chapitre nous avons proposé une formalisation d'une politique de contrôle d'accès en tant qu'ensemble d'interactions et la notion de graphe d'interactions. Ce graphe contient l'ensemble des interactions autorisées par la politique de contrôle d'accès.

De plus, nous définissons la notion de *graphe de dépendance causale* qui représente l'ensemble des dépendances entre interactions de la politique. Ce graphe conduit à deux cas particuliers : le graphe de flux d'informations et le graphe de transitions. Nous avons montré qu'il y avait équivalence entre l'existence d'un chemin dans un de ces graphes et une séquence observable.

Nous proposons différents algorithmes permettant l'énumération des interactions, des séquences et des corrélations. Pour chaque propriété de sécurité formalisée dans notre langage d'activités, nous proposons un algorithme qui permet d'énumérer toutes les activités illicites (signatures) violant cette propriété. Ces algorithmes correspondent à une implantation des terminaux de notre langage et nous traitons ainsi les trois classes d'activités définies précédemment : les *interactions*, les *séquences* et les *corrélations*.

Nous proposons, dans le chapitre 5, une approche de détection d'intrusions afin de détecter la

"réalisation" des activités violant une propriété de sécurité. Cette approche exploite la définition d'*une politique de détection* et d'*une politique de contrôle d'accès* afin de calculer une base de signatures d'activités illicites. Une politique de détection définit alors l'ensemble des propriétés de sécurité qui doivent être respectées sur un système. Ainsi, par analyse des traces du système, cette approche permet de détecter l'apparition des activités qui correspondent à des violations effectives des propriétés de sécurité.

Chapitre 5

Application à la détection d'intrusions

La détection d'intrusions proposée dans ce chapitre prend en entrée une politique de contrôle d'accès et une politique de détection. La politique de détection contient un ensemble de propriétés de sécurité tel que définies dans le chapitre précédent. L'intérêt de cette politique de détection est donc de permettre à un administrateur de définir les propriétés de sécurité souhaitées qu'il ne peut pas exprimer dans la politique de contrôle d'accès. La politique de contrôle d'accès permet de générer les trois graphes vus précédemment. Les graphes permettent alors d'énumérer les activités qui violent les propriétés définies dans la politique de détection. Ces activités sont ensuite utilisées pour détecter les violations apparaissant sur le système. La détection analyse les traces des interactions (*log* d'appels système) afin de signaler l'apparition de ces activités illicites. Notre langage de description d'activités sert non seulement à exprimer des propriétés de sécurité mais permet aussi d'exprimer les activités illicites (attaques). La méthode proprosée est tout à fait générale puisqu'elle repose sur un langage qui permet d'exprimer toute propriété d'intégrité ou de confidentialité mettant en jeu des interactions, des séquences ou des combinaisons observables sur un système.

Notre approche de détection d'intrusions comprend deux grandes étapes. En premier lieu, l'administrateur du système doit utiliser les propriétés de sécurité définies précédemment, c'est-à-dire toutes celles exprimables dans notre langage, pour définir des règles de sécurité qui vont constituer la politique de détection. Dans un second temps, cette politique est utilisée par notre solution de détection d'intrusions afin de construire une base d'activités illicites.

Ce chapitre détaillera dans un premier temps la politique de détection et dans un second temps la solution générique de détection. La figure 5.1 fait apparaître les deux grandes étapes d'administration et de détection. Pour l'étape de détection, il montre les trois phases permettant de lever une alerte lorsqu'une propriété définie dans la politique de détection est violée :

1. *Construction des graphes* : la *politique de contrôle d'accès* permet de construire le *graphe d'interactions* et les différents *graphes de dépendance causale* ;
2. *Génération de la base de signatures* : chaque règle de la politique de détection permet d'énumérer les activités qui violent cette règle. Chaque activité correspond à une signature et l'ensemble de ces signatures constitue la base de signatures violant la politique de détection ;
3. *Détection des activités* : nous utilisons les traces d'interactions pour détecter les activités qui correspondent à des signatures de la base. Lorsqu'une telle activité est reconstituée, une alerte est alors levée.

Dans ce chapitre, nous détaillons l'étape d'administration qui correspond à la définition d'une politique de détection dans la section 5.1. Ensuite, nous présentons les trois phases de l'étape de détection.

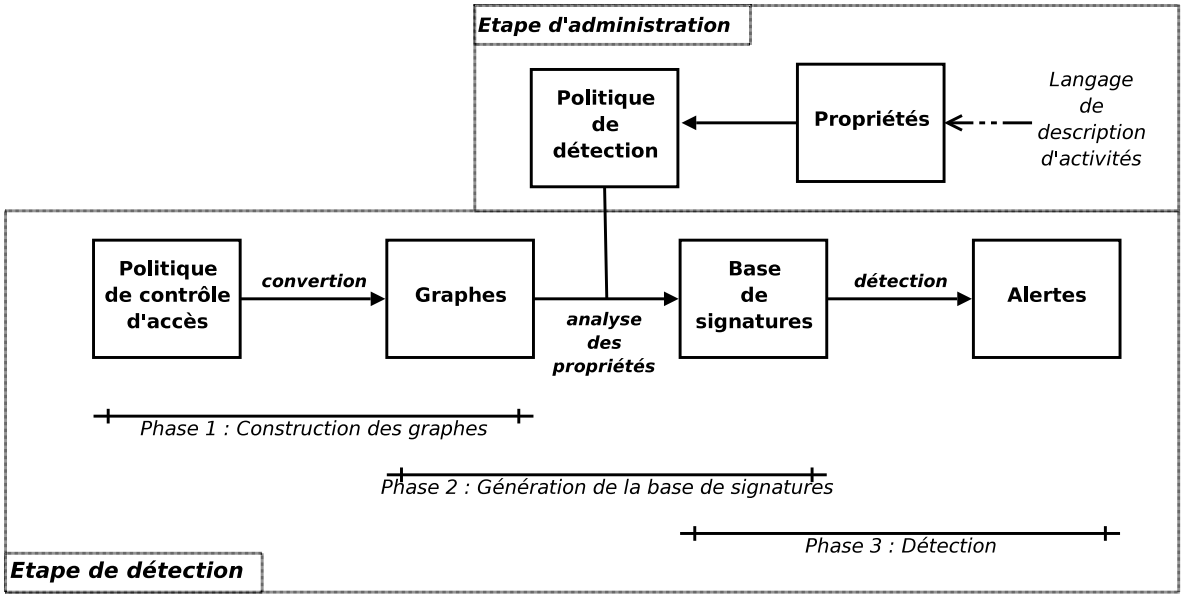


FIG. 5.1 – Fonctionnement général.

L'application de l'analyse d'une politique de contrôle d'accès à la détection d'intrusion, développée dans ce chapitre, a également été exposée dans les articles [Briffaut *et al.* 2005a, Briffaut 2005, Blanc *et al.* 2006, Briffaut *et al.* 2006c].

5.1 Politique de détection

Nous définissons par *politique de détection*, un ensemble de propriétés de sécurité qui doivent être respectées sur un système. Ces propriétés sont toutes celles exprimées dans notre langage de description d'activités. Notre *politique de détection* définit ainsi un ensemble de propriétés à satisfaire. La table A.1 (page 188) synthétise l'ensemble des propriétés de sécurité qui peuvent être exprimées. La définition de ces propriétés repose sur le nom des règles correspondant aux algorithmes d'énumération des violations définis dans la section 4.4.

Une règle est une de ces propriétés pour laquelle on définit la valeur des paramètres. Pour chaque propriété, nous indiquons le nom à utiliser dans la politique de détection, les paramètres nécessaires et l'objectif de cette règle. Par exemple, la propriété `integrity` (ligne 1) correspond à la propriété d'*intégrité des objets* et prend en arguments un couple de contextes ou un couple d'ensemble de contextes. Cette propriété permet par exemple de définir la règle `integrity`$(SC_{user}, SC_{admin}, G_c)$ pour garantir l'intégrité des contextes administrateurs.

5.1.1 Définition de la politique de détection

L'administrateur doit définir l'ensemble des règles de sécurité qu'il veut inclure dans la politique de détection. Une règle correspond à une notion facilement utilisable par un administrateur. Par exemple, grâce à la règle `integrity`$(SC_{user}, SC_{admin}, G_c)$, l'administrateur peut garantir l'intégrité des fichiers système en interdisant leur modification par les utilisateurs.

Si le nombre de règles à définir peut être important selon les besoins d'intégrité et de confidentialité, il s'agit, en tout cas, de notions facilement compréhensibles par un administrateur système. De plus, l'existence de propriétés génériques, comme `integrity` ou `confidentiality`, permet de garantir, avec peu de règles, des propriétés pour un ensemble de contextes. Ainsi la seule règle `integrity(`SC_{user}`,` SC_{admin}`,` G_c`)` protège de toutes les violations d'intégrité du domaine administrateur. Cependant, des règles plus fines permettent de définir des propriétés restreintes mais plus ciblées vis à vis des besoins de sécurité. L'administrateur dispose donc de règles générales qui surestiment les risques, mais qui sont simples à définir, et de règles spécialisées qui affinent les propriétés, mais qui demandent une meilleure expertise du système.

Afin d'illustrer cette notion de politique de détection, nous proposons, dans la partie suivante, de définir un exemple de politique. Nous reprenons ainsi les exemples de règles définies dans la section section 4.4 (page 110).

5.1.2 Exemple de politique de détection

Considérons les ensembles de contextes de sécurité sujets (SC_S) et objets (SC_O) suivants :

$SC_S = \{sc_{admin\_d}, sc_{user\_d}, sc_{login\_d}, sc_{ssh\_d}, sc_{apache\_d}, sc_{webserv\_d}\}$
$SC_O = \{sc_{apache\_conf\_t}, sc_{var\_www\_t}, sc_{admin\_info\_t}, sc_{user\_info\_t}\}$

Afin de paramétrer les règles de détection, l'administrateur définit différents ensembles de contextes. Les ensembles SC_{user} et SC_{admin} correspondent respectivement aux contextes associés aux utilisateurs et aux administrateurs. L'ensemble SC_{apache} contient tous les contextes liés au domaine apache. L'ensemble $SC_{systeme}$ définit l'ensemble des services système et SC_{web} l'ensemble des contextes lié au service Web. Finalement, l'ensemble SC_{TCB} contient l'ensemble des contextes ayant trait au service Web. L'administrateur définit ainsi six ensembles de contextes, en fonction de ses besoins de sécurité :

$$
\begin{array}{ll}
SC_{user} & = \{sc_{user\_d}, sc_{user\_info\_t}\} \\
SC_{admin} & = \{sc_{admin\_d}, sc_{admin\_info\_t}\} \\
SC_{apache} & = \{sc_{apache\_d}, sc_{webserv\_d}, sc_{apache\_conf\_t}, sc_{var\_www\_t}\} \\
SC_{systeme} & = \{sc_{login\_d}, sc_{ssh\_d}, sc_{apache\_d}, sc_{webserv\_d}\} \\
SC_{web} & = SC_{apache} \cup \{sc_{user\_info\_t}, sc_{admin\_info\_t}\} \\
SC_{TCB} & = \{sc_{apache\_d}, sc_{webserv\_d}, sc_{apache\_conf\_t}, sc_{var\_www\_t}, sc_{admin\_info\_t}, sc_{user\_info\_t}\}
\end{array}
$$

L'administrateur définit une politique de détection constituée de trois catégories de règles dans le but de garantir des propriétés d'*intégrité*, de *confidentialité* et *d'abus de privilèges*. Lors de la définition de ces règles, l'administrateur limite la portée de certaines règles grâce au type de graphe passé en paramètres. Il utilise alors soit le graphe de dépendance causale (G_{dep}) ou soit le graphe de transitions (G_{trans}). Dans cet exemple de politique de détection, l'administrateur se restreint aux *accès à un privilège* et aux *accès à l'information* par transition, c'est pourquoi il passe en paramètres le *graphe de transitions* (G_{trans}) lors de la définition de certaines règles.

Nous allons maintenant détailler chaque catégorie de règles en distinguant le type de propriété visé.

5.1.2.1 Règles d'intégrité

Dans la politique de détection du listing 5.1, l'administrateur considère un premier ensemble de règles (règles 1 à 7) afin de garantir des propriétés d'*intégrité* sur le système.

Ainsi, quatre règles d'intégrité des objets (`integrity(...)`) (cf. page 73) sont définies. Les règles 1) et 2) ont pour conséquence d'interdire toute modification, faite par un *utilisateur*, d'un contexte du domaine *apache* ou *admin*. Les règles 3) et 4) empêchent que le domaine *apache* ne viole l'intégrité

```
integrity(SC_user, SC_admin, G_trans)                                    1
integrity(SC_user, SC_apache, G_trans)                                   2
integrity(SC_apache, SC_user, G_trans)                                   3
integrity(SC_apache, SC_admin, G_trans)                                  4
int_subject(SC_systeme)                                                  5
int_domain(SC_TCB)                                                       6
int_biba(SC_web)                                                         7
                                                                         8
confidentiality(sc_admin_info_t, sc_user_d, G_trans)                     9
confidentiality(sc_user_info_t, sc_admin_d, G_trans)                     10
conf_data(SC_user, SC_admin)                                             11
conf_blp(SC_web, G_trans)                                                12
                                                                         13
duties_sep(SC_systeme, G_trans)                                          14
bad_transition(sc_ssh_d, sc_admin_d)                                     15
bad_transition(sc_ssh_d, sc_apache_d)                                    16
bad_transition(sc_ssh_d, sc_webserv_d)                                   17
conformity()                                                             18
```

Listing 5.1 – Exemple de politique de détection.

des domaines *utilisateurs* et *administrateurs*. Ainsi, si un attaquant prend le contrôle du serveur Web, l'intégrité des contextes SC_{user} ou SC_{admin} est garantie.

La règle `int_subject(`$SC_{systeme}$`)` permet d'empêcher deux sujets d'interférer et ainsi d'assurer l'*intégrité des sujets* du système.

La règle `int_domain(`SC_{TCB}`)` a pour objectif d'isoler le serveur et le service Web du reste du système en les cloisonnant (tel un *chroot virtuel*). Cette règle interdit alors toute interaction entre les services Web et le reste du système et assure l'*intégrité du domaine* Web. Ce type de règle peut permettre d'isoler temporairement ces services. Par exemple, cette règle peut être activée la nuit lorsqu'aucun utilisateur ne doit accéder à l'hôte.

Nous pouvons finalement appliquer la propriété d'*intégrité de* BIBA au domaine Web (SC_{web}) via la règle `int_biba(`SC_{web}`)`. Pour appliquer cette règle, nous définissons alors la table suivante afin d'associer un niveau d'intégrité à chaque contexte :

Contexte(s)	Niveau d'intégrité
$sc_{user\_info\_t}$, $sc_{admin\_info\_t}$	0
$sc_{apache\_conf\_t}$	1
$sc_{var\_www\_t}$, sc_{apache_d}, $sc_{webserv\_d}$	2

Cette règle permet ainsi de garantir l'intégrité des fichiers de configuration (niveau 1) et des fichiers utilisateurs et administrateurs (niveau 2). Notons que ce type de règle est plus complexe à définir mais peut être appliqué sur un sous ensemble du système. Ici, l'administrateur n'applique cette propriété que sur les données critiques.

5.1.2.2 Règles de confidentialité

Un second ensemble de règles permet d'appliquer des propriétés de *confidentialité* sur le système. Les deux premières règles (ligne 9 et 10) ont pour objectif d'interdire l'accès, par les utilisateurs, aux fichiers d'informations de l'administrateur (`confidentiality(`$sc_{admin\_info\_t}$`,` sc_{user_d}`,` G_{trans}`))`) et réciproquement interdire aux administrateurs l'accès aux informations utilisateurs (`confidentiality(`$sc_{user\_info\_t}$`,` sc_{admin_d}`,` G_{trans}`))`).

La règle conf_data(SC_{user}, SC_{admin}, G_{trans}) a pour objectif d'interdire tout accès par transition à une donnée de l'administrateur à un utilisateur. De ce fait, l'utilisateur ne peut accéder qu'aux données auxquelles il a directement accès (par interaction).

L'administrateur peut finalement appliquer la propriété de *confidentialité de* BLP au domaine Web SC_{web} via la règle conf_blp(SC_{web}). Cette règle utilise, par exemple, la table suivante afin d'associer un niveau d'habilitation à chaque contexte :

Contexte(s)	Niveau d'habilitation
$sc_{apache\_conf\_t}$	0
$sc_{apache\_d} sc_{var\_www\_t}$	1
$sc_{webserv\_d} sc_{user\_info\_t} sc_{admin\_info\_t}$	2

Cette règle permet alors de garantir la confidentialité des fichiers d'informations de niveau 2 ($sc_{user\_info\_t}$ et $sc_{admin\_info\_t}$).

5.1.2.3 Règles d'abus de privilèges

Finalement, l'administrateur définit un ensemble de règles afin d'appliquer des propriétés d'*abus de privilèges*. Nous pouvons empêcher les tentatives de création d'objets suivies de leur exécution par les services système via la règle de *séparation de privilèges* duties_sep($SC_{systeme}$, G_{trans}).

Sur ce système, nous pouvons aussi interdire que le domaine *administrateur* ou les services Web soient accessibles suite à une connexion ssh. Cette règle peut, par exemple, s'appliquer sur un système où les opérations de maintenance doivent être réalisées localement (via login). La règle bad_transition(sc_{ssh_d}, sc_{admin_d}) interdit ainsi toute transition depuis *ssh* vers le contexte administrateur. De même, les règles bad_transition(sc_{ssh_d}, sc_{apache_d}) et bad_transition(sc_{ssh_d}, $sc_{webserv\_d}$) interdisent toute transition depuis *ssh* vers les contextes associés aux serveurs Web et aux services Web. Finalement, la règle conformity() permet de garantir le respect de la politique de contrôle d'accès. Ainsi, toute interaction pourra soit être prévenue, soit donner lieu à une alerte.

Comme nous le verrons dans la section 5.3, cette *politique de détection* permet de générer un ensemble de *signatures* constituant ainsi une base de signatures d'attaques réalisables. Ces signatures, obtenue par l'analyse des graphes de politique et de dépendance causale, correspondent à des *activités* réalisables du point de vue de la *politique de contrôle d'accès*, mais violant une des propriétés de sécurité définies dans la *politique de détection*. Les sections suivantes présentent les différentes phases de notre solution de détection.

5.2 Construction des graphes

Nous détaillons tout d'abord la phase de *construction des graphes*. La figure 5.2 représente le fonctionnement de la phase de construction des graphes. La politique de contrôle d'accès consiste en un ensemble d'interactions qui permet de calculer le *graphe d'interactions*. Ce graphe sert alors d'entrée à la construction des trois graphes de dépendance causale.

La politique de contôle d'accès peut être définie par différents moyens. Soit elle est définie manuellement par l'administrateur, soit elle est calculée par réutilisation de politiques de contrôle d'accès existantes. Dans le chapitre 6, nous présenterons comment nous réutilisons les politiques de contrôle d'accès MAC tel que SELinux ou GRSECURITY. Dans ce chapitre, nous considérons qu'une politique, correspondant à un ensemble de vecteurs d'interactions $\mathcal{POL} \subset IV$, est disponible.

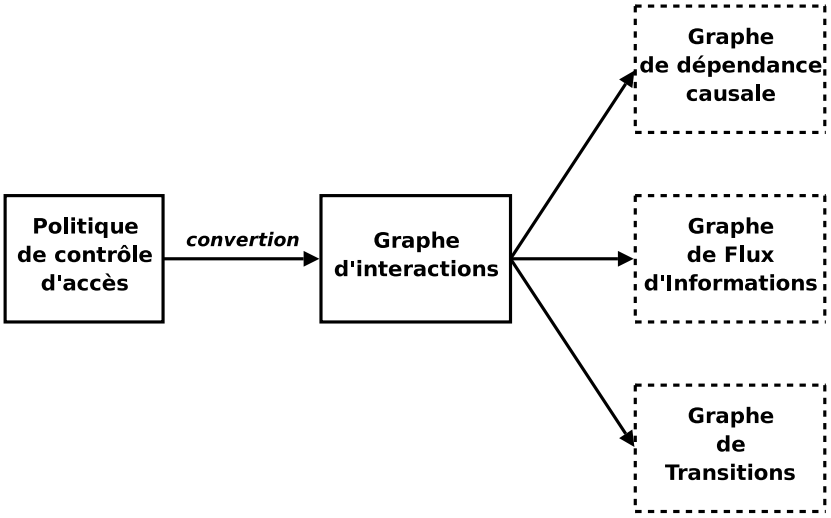


FIG. 5.2 – Phase 1 : Génération des graphes de politique et de dépendance causale.

L'algorithme 7 de construction des graphes réutilise les algorithmes `graphe_politique` et `graphe_dependance_causale` et les tables de changement d'état définies dans le chapitre 4 . Cet algorithme prend en paramètres une politique de contrôle d'accès $\mathcal{POL}$ et les trois tables de changement d'état $T_{dep}, T_{inf}, T_{trans}$. Il retourne le graphe d'interactions (G_{int}), le graphe de dépendance causale (G_{dep}), le graphe d'informations (G_{inf}) et le graphe de transitions (G_{trans}).

`construire_graphes`$(\mathcal{POL} \subset IV, T_{dep}, T_{inf}, T_{trans})$
début
  G_{int} :=`graphe_politique`$(\mathcal{POL})$;
  G_{dep} :=`graphe_dependance_causale`(G_{int}, T_{dep});
  G_{inf} :=`graphe_dependance_causale`(G_{int}, T_{inf});
  G_{trans} :=`graphe_dependance_causale`(G_{int}, T_{trans});
  retourner $G_{int}, G_{dep}, G_{inf}, G_{trans}$;
fin

Algorithme 7 : Algorithme de construction des graphes de politique et de dépendance causale.

La politique de contrôle d'accès sert ainsi d'entrée à l'algorithme de construction du *graphe d'interactions* (algo. 1, page 93). Ce graphe représente alors l'ensemble des interactions autorisées par cette politique. Ce graphe d'interactions sert ensuite d'entrée à l'algorithme de construction des différents *graphes de dépendance causale* (algo. 2, page 95). Ce second algorithme permet alors, via la définition des *tables de changement d'état*, de générer le *graphe de dépendance causale* général, le *graphe de flux d'informations* et le *graphe de transitions*. Ces trois graphes représentent respectivement l'ensemble des *séquences d'interactions*, des *flux d'informations* et des *séquences de transitions* réalisables sur le système, c'est-à-dire légales du point de vue de la politique de contrôle d'accès.

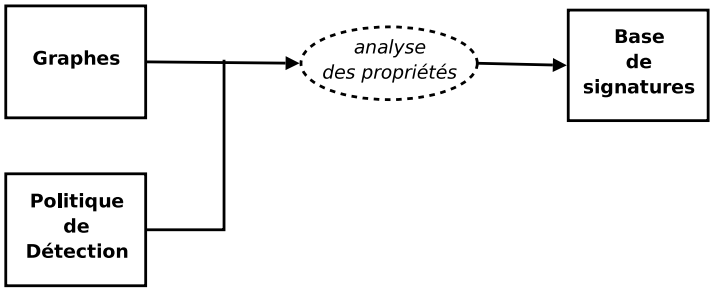


FIG. 5.3 – Phase 2 : Construction de la base de signatures.

5.3 Génération de la base de signatures

La phase de génération de la *base de signatures* a pour objectif de calculer l'ensemble des activités violant les propriétés de sécurité définies dans la politique de détection. Chaque *signature* correspond ainsi à une activité (c'est-à-dire une interaction, une séquence ou une corrélation) violant une propriété de sécurité.

La figure 5.3 représente le processus de génération de la base de signatures (phase 2). Cette phase utilise les graphes construits durant la première phase et la politique de détection.

Dans cette section, nous définissons notre représentation de la *base de signatures*, puis nous proposerons un algorithme de construction de cette base. Nous donnons ensuite un exemple de construction de cette base.

5.3.1 Base de signatures

Une *signature* représente une *activité illicite* violant une des propriétés de sécurité définies dans la *politique de détection*. Dans notre modèle, chaque signature appartient à une des classes d'activités définies dans la section 3.1 (page 56). De ce fait, chaque signature correspond à une *action élémentaire*, une *séquence* ou une *corrélation*.

Une base de signatures ($BASE$) est un ensemble d'*activités* exprimées dans notre *langage de description d'activités* (cf. section 3.2.4.1, page 67).

$$BASE = \{activite_1, \ldots, activite_n\}$$

Chaque activité peut être vue comme un ensemble d'interactions tel que le modélise la classe `Activite`. Cette classe permettra de définir, dans la section 5.4, des algorithmes de reconstruction des séquences. Pour ce faire, nous définissons une méthode (`get(int pos)`) qui renvoie la $pos^{\text{ème}}$ interaction d'une activité.

```
Classe Activite
début
    Liste <Interaction> interactions;

    Interaction get(int pos)
    début
        Retourne interactions.get(pos);
    fin
fin
```

Afin de générer automatiquement cette base, nous allons maintenant proposer un algorithme de construction de la base de signatures.

5.3.2 Construction de la base de signatures

En fonction d'une *politique de détection* et des différents *graphes*, calculés précédemment, nous proposons l'algorithme 8 de *construction de la base de signatures*.

`construire_base` ($\mathcal{POL} \subset IV, \mathcal{POL}_{detection}, G_{int}, G_{dep}, G_{inf}, G_{trans}$)
début
  $BASE := \emptyset$;
  pour chaque $regle \in \mathcal{POL}_{detection}$ **faire**
    $BASE := BASE \cup regle()$;
  fin
  retourner $BASE$;
fin

Algorithme 8 : Algorithme de construction de la base de signatures.

Cet algorithme appelle chaque règle de détection ($regle$) de la politique de détection ($\mathcal{POL}_{detection}$). L'appel d'une règle correspond à l'exécution de l'algorithme d'énumération associé (cf. section 4.4). Le résultat de l'énumération est alors ajouté à la base de signatures. Ainsi, pour la première règle de notre exemple de politique de détection (cf. listing 5.1, page 130), cela correspond à l'appel de l'algorithme `integrity(`SC_{user}`,` SC_{admin}`,` G_{trans}`)`.

5.3.3 Exemple

Dans cette partie, nous donnons la base correspondant à notre exemple de politique de détection (cf. listing 5.1, page 130) en prenant comme politique de contrôle d'accès celle définie dans le chapitre 4 (cf. listing 4.1, page 92).

La base résultante est constituée de 37 signatures qui correspondent chacune à une violation possible d'une propriété de sécurité.

$$BASE = \{activite_1, activite_2, \ldots, activite_{37}\}$$

Le listing 5.2 contient l'ensemble des signatures pour les règles d'intégrité de cette politique de détection. Dans ce listing, les signatures sont regroupées en fonction de la règle de détection violée (commençant par le caractère '#'). Ainsi, l'application de la règle `integrity(`SC_{user}`,` SC_{admin}`,` G_{trans}`)` renvoie une seule signature qui représente un *accès au privilège de modification*, la seconde règle (ligne 4) retourne deux signatures du même type. La règle pour la propriété d'intégrité des sujets (ligne 8) renvoie deux signatures correspondant à des corrélations (la composition d'une modification et d'une exécution). Finalement, la règle de la ligne 12 (propriété d'intégrité des domaines) retourne neuf signatures (des interactions) correspondant à des possibles interactions entre le domaine Web et le reste du système. Chacune de ces signatures est donc une activité réalisable de part la définition de la politique de contrôle d'accès, mais qui qui viole une des propriétés de sécurité définies dans la politique de détection.

Le listing 5.3 contient l'ensemble des signatures correspondant aux règles de confidentialité. La première règle (confidentialité des contextes) renvoie cinq signatures dont un *flux d'informations* et

1 #regle integrity($SC_{user}, SC_{admin}, G_{trans}$)
2 $activite_1 = (sc_{user\_d} \xrightarrow{trans} sc_{admin\_d} \xrightarrow{write} sc_{admin\_info\_t})$
3
4 #regle integrity($SC_{user}, SC_{apache}, G_{trans}$)
5 $activite_2 = (sc_{user\_d} \xrightarrow{trans} sc_{admin\_d} \xrightarrow{write} sc_{apache\_conf\_t})$
6 $activite_3 = (sc_{user\_d} \xrightarrow{trans} sc_{admin\_d} \xrightarrow{trans} sc_{apache\_d} \xrightarrow{write} sc_{var\_www\_t})$
7
8 #regle int_subject($SC_{systeme}$)
9 $activite_4 = (sc_{webserv\_d} \xrightarrow{read} sc_{var\_www\_t}) \circ (sc_{apache\_d} \xrightarrow{write} sc_{var\_www\_t})$
10 $activite_5 = (sc_{apache\_d} \xrightarrow{read} sc_{var\_www\_t}) \circ (sc_{webserv\_d} \xrightarrow{write} sc_{var\_www\_t})$
11
12 #regle int_domain(SC_{TCB})
13 $activite_6 = (sc_{admin\_d} \xrightarrow{read} sc_{apache\_conf\_t})$
14 $activite_7 = (sc_{admin\_d} \xrightarrow{write} sc_{apache\_conf\_t})$
15 $activite_8 = (sc_{admin\_d} \xrightarrow{trans} sc_{apache\_d})$
16 $activite_9 = (sc_{admin\_d} \xrightarrow{trans} sc_{webserv\_d})$
17 $activite_{10} = (sc_{admin\_d} \xrightarrow{read} sc_{admin\_info\_t})$
18 $activite_{11} = (sc_{admin\_d} \xrightarrow{write} sc_{admin\_info\_t})$
19 $activite_{12} = (sc_{user\_d} \xrightarrow{trans} sc_{webserv\_d})$
20 $activite_{13} = (sc_{user\_d} \xrightarrow{read} sc_{user\_info\_t})$
21 $activite_{14} = (sc_{user\_d} \xrightarrow{write} sc_{user\_info\_t})$

Listing 5.2 – Exemple de base de signatures(partie 1).

quatre *accès à l'information*. Il en va de même pour la seconde règle (ligne 8). Ainsi, l'existence de ces signatures montre que malgré l'utilisation d'une politique de contrôle d'accès, il est tout de même possible que l'utilisateur accède aux données personnelles de l'administrateur et inversement. La règle de la propriété de confidentialité des données (ligne 16) retourne quatre signatures qui représentent des accès indirects aux données de l'administrateur par l'utilisateur, alors que l'accès direct n'est pas possible. Notons que dans cet exemple, les signatures 26 à 29 sont redondantes vis-à-vis des signatures 16 à 19. En effet une signature peut correspondre à la violation de plusieurs propriétés. Ici, la troisième règle est un cas particulier de la première. La quatrième règle (ligne 22) énumère deux *flux d'informations* violant la propriété de confidentialité du modèle BLP. Elles correspondent à un flux d'un objet vers un objet de niveau inférieur (cf. section 3.3.2.2, page 79).

Finalement, le listing 5.4 contient l'ensemble des signatures pour les règles d'abus de privilèges. La règle pour la propriété de séparation de privilèges renvoie ainsi trois signatures impliquant une exécution possible d'un objet préalablement modifié (ou créé). Ces signatures sont donc à des *corrélations* (la combinaison d'une modification et d'une exécution). La seconde règle (propriété d'absence de changement de contexte) renvoie une *séquence de transitions* correspondant à l'acquisition du contexte *administrateur* à partir du contexte *ssh*. De même, les deux dernières règles (ligne 9 et 12) retournent une signature du même type. Ces signatures représentent ainsi à un accès possible aux contextes d'administration ou de service système via une connexion ssh.

Ainsi, nous avons obtenu automatiquement un ensemble de 37 signatures permettant de violer les propriétés de sécurité. Dans la section suivante, nous utilisons les traces système pour détecter les violations effectives et en conséquence générer des alertes.

1 #regle confidentiality($sc_{admin\_info\_t}, sc_{user\_d}, G_{trans}$)
2 $activite_{15} = (sc_{admin\_info\_t} > sc_{temp\_t} > sc_{user\_d})$
3 $activite_{16} = (sc_{user\_d} \xrightarrow{trans} sc_{webserv\_d}) \wedge (sc_{admin\_info\_t} > sc_{webserv\_d})$
4 $activite_{17} = (sc_{user\_d} \xrightarrow{trans} sc_{admin\_d}) \wedge (sc_{admin\_info\_t} \xrightarrow{trans} sc_{admin\_d})$
5 $activite_{18} = (sc_{user\_d} \xrightarrow{trans} sc_{admin\_d} \xrightarrow{trans} sc_{webserv\_d}) \wedge (sc_{admin\_info\_t} > sc_{webserv\_d})$
6 $activite_{19} = (sc_{user\_d} \xrightarrow{trans} sc_{admin\_d} \xrightarrow{trans} sc_{apache\_d} \xrightarrow{trans} sc_{webserv\_d}) \wedge (sc_{admin\_info\_t} > sc_{webserv\_d})$
7
8 #regle confidentiality($sc_{user\_info\_t}, sc_{admin\_d}, G_{trans}$)
9 $activite_{20} = (sc_{user\_info\_t} > sc_{user\_d} > sc_{admin\_d})$
10 $activite_{21} = (sc_{user\_info\_t} > sc_{user\_d} > sc_{temp\_t} > sc_{admin\_d})$
11 $activite_{22} = (sc_{admin\_d} \xrightarrow{trans} sc_{webserv\_d}) \wedge (sc_{user\_info\_t} > sc_{webserv\_d})$
12 $activite_{23} = (sc_{admin\_d} \xrightarrow{trans} sc_{apache\_d} \xrightarrow{trans} sc_{webserv\_d}) \wedge (sc_{user\_info\_t} > sc_{webserv\_d})$
13 $activite_{24} = (sc_{admin\_d} \xrightarrow{trans} sc_{webserv\_d}) \wedge (sc_{user\_info\_t} > sc_{webserv\_d})$
14 $activite_{25} = (sc_{admin\_d} \xrightarrow{trans} sc_{apache\_d} \xrightarrow{trans} sc_{webserv\_d}) \wedge (sc_{user\_info\_t} > sc_{webserv\_d})$
15
16 #regle conf_data(SC_{user}, SC_{admin})
17 $activite_{26} = (sc_{user\_d} \xrightarrow{trans} sc_{webserv\_d}) \wedge (sc_{admin\_info\_t} > sc_{webserv\_d})$
18 $activite_{27} = (sc_{user\_d} \xrightarrow{trans} sc_{admin\_d}) \wedge (sc_{admin\_info\_t} \xrightarrow{trans} sc_{admin\_d})$
19 $activite_{28} = (sc_{user\_d} \xrightarrow{trans} sc_{admin\_d} \xrightarrow{trans} sc_{webserv\_d}) \wedge (sc_{admin\_info\_t} > sc_{webserv\_d})$
20 $activite_{29} = (sc_{user\_d} \xrightarrow{trans} sc_{admin\_d} \xrightarrow{trans} sc_{apache\_d} \xrightarrow{trans} sc_{webserv\_d}) \wedge (sc_{admin\_info\_t} > sc_{webserv\_d})$
21
22 #regle conf_blp(SC_{web}, G_{trans})
23 $activite_{30} = (sc_{user\_info\_t} > sc_{webserv\_d} > sc_{var\_www\_t})$
24 $activite_{31} = (sc_{admin\_info\_t} > sc_{webserv\_d} > sc_{var\_www\_t})$

Listing 5.3 – Exemple de base de signatures(partie 2).

5.4 Détection des activités illicites

Cette phase peut ainsi être considérée comme une approche de détection par scénario (cf. partie 2.2.2) où chaque scénario correspond à un ensemble d'interactions réalisable mais illégal du point de vue de la politique de détection.

Afin de détecter l'apparition des signatures, nous proposons d'utiliser les traces d'interactions. En effet, les systèmes permettent généralement d'auditer les interactions réalisées au niveau du noyau (appels système). Nous verrons deux exemples de ce type d'audit dans le chapitre 6. La phase de détection utilise donc ces traces afin de détecter l'apparition d'une activité de la base. L'apparition d'une telle activité dans les traces du système entraînera alors la génération d'une alerte. Lorsqu'une interaction est auditée, une trace est générée par le système. Un trace t contient au moins trois informations :

- une interaction $it = (sc_{source} \in SC_S, sc_{source} \in SC, eo \in IS)$, que nous noterons $t.it$, qui représente l'interaction effectuée ;
- le *PID* (identifiant du processus) et *PPID* (identifiant du processus père) du processus étiqueté par sc_{source}. Nous noterons $t.pid$ et $t.ppid$ ces deux informations ;
- la date $date$ d'apparition de l'interaction notée $t.date$.

Nous notons $T = \{t_1, t_2, \ldots, t_n\}$ l'ensemble des traces générées par le système.

Comme le représente la figure 5.4, cette phase utilise en entrée l'ensemble des traces générées et une base de signatures. Un mécanisme de reconstruction de signatures permet alors de détecter l'apparition des signatures.

```
1  #regle duties_sep(SC_systeme, G_trans)
2  activite_32 = (sc_apache_d --execute--> sc_var_www_t) ∘ (sc_apache_d --write--> sc_var_www_t)
3  activite_33 = (sc_webserv_d --execute--> sc_var_www_t) ∘ (sc_webserv_d --write--> sc_var_www_t)
4  activite_34 = (sc_user_d --trans--> sc_apache_d --write--> sc_var_www_t) ∘ (sc_user_d --trans--> sc_apache_d --execute--> sc_var_www_t)
5
6  #regle bad_transition(sc_ssh_d, sc_admin_d)
7  activite_35 = (sc_ssh_d --trans--> sc_user_d --trans--> sc_admin_d)
8
9  #regle bad_transition(sc_ssh_d, sc_apache_d)
10 activite_36 = (sc_ssh_d --trans--> sc_user_d --trans--> sc_admin_d --trans--> sc_apache_d)
11
12 #regle bad_transition(sc_ssh_d, sc_webserv_d)
13 activite_37 = (sc_ssh_d --trans--> sc_user_d --trans--> sc_webserv_d)
```

Listing 5.4 – Exemple de base de signatures (partie 3).

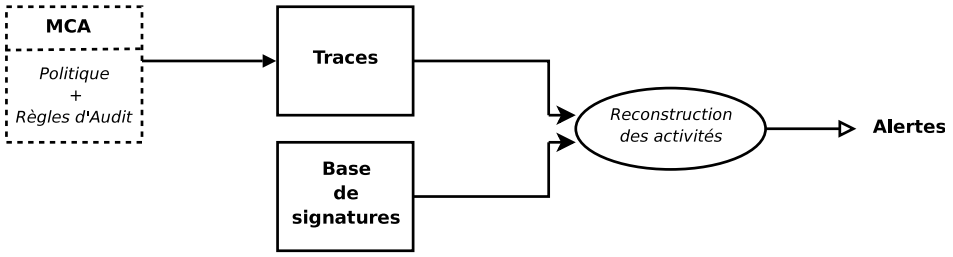


FIG. 5.4 – Phase 3 : Détection des activités.

Afin de détecter l'apparition des activités illicites (correspond à une signature), nous proposons tout d'abord une représentation objet des activités de notre langage de description d'activités. Cette modélisation permet d'associer à chaque activité en cours un objet qui rend compte de l'avancement de l'activité. Nous appelons alors ces objets des *observateurs* d'activités. Ces observateurs permettront ensuite de définir les différentes méthodes de reconstruction des activités.

5.4.1 Observation des activités

Un observateur est un *objet* lié à une activité (issue d'une signature) et qui correspond à l'état actuel d'une activité sur le système. Un observateur permet ainsi de connaître si l'activité a été partiellement ou complètement observée et la date d'apparition de cette activité. Les observateurs permettent de faire le lien entre les activités (classe `Activite`, cf. partie 5.3.1) et les signatures.

Dans cette partie, nous proposons une représentation *objet* des différents observateurs liés aux classes d'activités : *interaction*, *séquence* et *corrélation*. Cette représentation permettra ensuite de proposer différents algorithmes de reconstruction des signatures (activités). Nous proposerons finalement un algorithme d'analyse des traces système permettant de détecter l'apparition de ces signatures. Nous définissons ainsi une classe `Observateur` qui correspond à une classe générale autorisant l'implantation de notre langage de description. Cette classe sera utilisée dans la suite de cette section afin

de définir les différents éléments de notre langage de description.

```
Classe Observateur
début
    booleen vu := faux;
    int date;
    booleen observe();
    int date();
    void analyse(Trace t);
fin
```

La phase de reconstruction des signatures nécessite de connaître si une activité est apparue (son état) et la date de son apparition. Nous définissons ainsi deux attributs (vu et $date$) associés à deux méthodes, `observe()` et `date()`, qui renvoient respectivement l'état de l'activité (observé ou pas) et sa date d'apparition. Finalement, nous associons à chaque activité une méthode `analyse(Trace t)` qui analyse une trace t. Cette méthode modifiera alors l'état d'observation d'une activité (booléen vu) et la date d'apparition ($date$) de cette activité.

Nous proposons maintenant d'étendre cette classe afin de modéliser les trois classes d'activité : *interaction*, *séquence* et *corrélation*.

5.4.1.1 Classe d'interaction

Un observateur *Interaction* est constitué d'une `Activite` de type interaction (it) qui peut correspondre à une *interaction générale*, un *transfert d'informations* ou une *transition*. La définition de la classe des interactions est simple : une interaction it est observée lorsqu'elle correspond à l'interaction de la trace ($t.it$). Dans ce cas, cette interaction passe dans l'état *observé* et la date d'apparition est fixée. La classe triviale qui en découle est la suivante :

```
Classe Interaction etend Observateur
début
    Activite it;

    booleen observe()
    début
        Retourne vu;
    fin

    booleen analyse(Trace t)
    début
        si t.it == it alors
            vu := vrai;
            date := t.date;
        fin
    fin

    int date()
    début
        Retourne date;
    fin
fin
```

5.4.1.2 Classe Séquence

Une séquence est une fermeture transitive causale d'un ensemble d'interactions. Ainsi, une séquence peut être représentée par une liste chainée d'interactions où chaque interaction ne pourra être

observée que si l'interaction précédente l'est déjà. La classe `Sequence` permet de représenter l'observateur lié à une séquence d'interactions, un flux d'informations ou une séquence de transitions.

```
Classe Sequence etend Observateur
début
    Activite seq;
    int obs := 1;

    booleen observe()
    début
        Retourne obs ==taille(seq);
    fin

    void analyse(Trace t)
    début
        seq.get(obs).analyse(t);
        si seq.get(obs).observe() alors
            obs := obs + 1;
        fin
    fin

    int date()
    début
        Retourne (seq.it_obs.date());
    fin
fin
```

Cette classe est donc constituée d'une `Activite` (seq) et possède un compteur qui correspond à l'état d'*avancement* de la séquence. Ce compteur est donc initialisé à 1 lors de la construction de l'objet correspondant à une séquence, puis il est incrémenté lorsque l'interaction en cours ($seq.get(obs)$) correspond à l'interaction de la trace (t). Ceci permet de prendre en compte la dépendance temporelle entre chaque interaction. Ainsi la méthode `analyse(Trace t)` incrémente le compteur obs lorsque la trace correspond à la prochaine interaction de la séquence. Finalement, la méthode `observe()` retourne $vrai$ lorsque le nombre d'interactions observé est égal à la taille de la séquence.

5.4.1.3 Corrélations

Une corrélation, telle que définie dans notre langage de description d'activité, correspond à une combinaison d'interactions, de séquences ou de corrélations réalisée avec trois opérateurs : $\circ, \vee, \wedge$. Les terminaux d'une corrélation correspondent toujours à une interaction ou une séquence. La classe `Correlation` est ainsi constituée de deux `Observateur` ($gauche$ et $droite$) et d'un opérateur de corrélation. Chacun de ces deux `Observateur` correspond ainsi à une `Interaction`, une `Sequence` ou une `Correlation`. Cette classe permet de représenter l'ensemble des corrélations exprimables dans notre

langage de description d'activités.

```
Classe Correlation etend Observateur
début
    Observateur gauche ;
    Observateur droite ;
    Char op;

    booleen observe()
    début
        si op ==' ∧' alors
            Retourne gauche.observe()∧droite.observe() ;
        fin
        si op ==' ∨' alors
            Retourne gauche.observe()∨droite.observe() ;
        fin
        si op ==' ∘' alors
            Retourne gauche.observe()∧droite.observe()∧(gauche.date() > droite.date()) ;
        fin
    fin

    void analyse(Trace t)
    début
        si op ==' ∘' alors
            si droite.observe() alors
                gauche.analyse(t);
            fin
            sinon
                droite.analyse(t);
            fin
        fin
        sinon
            gauche.analyse(t);
            droite.analyse(t);
        fin
    fin
fin
```

Dans cette classe, l'opérateur de corrélation correspond au caractère $'\wedge'$,$'\vee'$, ou $'\circ'$. En fonction de cet opérateur, une corrélation sera donc observée si les deux activités de cette corrélation ont été observées ($\wedge$), si l'une des activités a été observée ($\vee$). L'opérateur $\circ$ implique que les deux activités soient observées et que la date d'apparition de l'activité correspondant à l'opérande de gauche apparaissent après l'opérande de droite.

Finalement, une trace est analysée différemment lorsqu'il s'agit d'un opérateur booléen ($\wedge$ ou $\vee$) ou de l'opérateur de composition ($\circ$). En effet, celui-ci implique une dépendance temporelle entre les deux séquences, la trace n'est analysée par l'opérande de gauche que si l'opérante de droite est déjà observée, sinon elle est d'abord et seulement analysée par l'opérande de droite.

5.4.2 Reconstruction des activités

L'objectif de la reconstruction est de générer des alertes lorsqu'une signature est reconnue, c'est-à-dire lorsque toutes les interactions composant cette signature ont été réalisées. La phase de reconstruction analyse donc l'ensemble des traces d'interactions afin d'y déceler d'éventuelles attaques. Lorsqu'une signature est reconnue, la fonction 5.4.1 est exécutée afin de générer une alerte. La génération de l'alerte peut, par exemple, correspondre à l'écriture d'une alerte dans un fichier de traces, l'envoi d'un courriel à un administrateur, etc.

Fonction 5.4.1 `generer_alerte` : $BASE \times T$ *est une fonction telle qu'étant donné une signature de la base et une trace, cette fonction génère une alerte.*

La phase de reconstruction utilise la représentation objet des *observateurs* et nous définissons deux méthodes différentes de reconstruction. Chaque méthode prend en paramètres une trace $t \in T$ (générée par le mécanisme de contrôle d'accès) et un sous ensemble de la base de signatures ($B \subset BASE$). Une alerte est alors levée lorsqu'une signature de cette base est reconnue.

Comme nous l'avons vu dans la sections 3.2 (page 60), une séquence est une fermeture transitive d'un ensemble d'opérations élémentaires causalement liées. Pour reconstruire une séquence, il faut que l'opération it_{j-1} ait été observée avant l'interaction it_j. Pour cela, nous tenons compte des dates d'apparition des interactions.

5.4.2.1 Influence des méthodes de reconstruction

Pour reconstruire correctement *les séquences de transitions*, nous sommes obligés de tenir compte de l'héritage des processus. En effet, dans le cas contraire, deux problèmes différents se posent. Nous proposons d'illustrer chaque problème par un exemple. Le premier problème concerne les fausses séquences, la figure 5.5 illustre ce problème. Cette figure représente deux processus P_1 et P_2 indé-

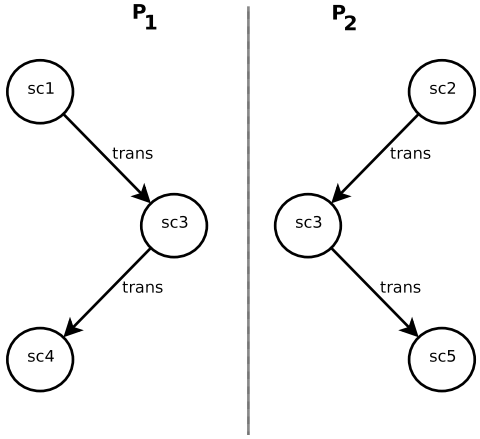


FIG. 5.5 – Reconstruction de processus indépendants.

pendants, étiquetés par sc_1 et sc_2, qui transitent tous les deux vers le contexte sc_3. Ensuite, ces deux processus transitent vers les contextes sc_4 et sc_5. Si l'on ne tient pas compte des processus, la reconstruction des séquences peut conduire à deux fausses séquences : $sc_1 \xrightarrow{trans} sc_3 \xrightarrow{trans} sc_5$ et $sc_2 \xrightarrow{trans} sc_3 \xrightarrow{trans} sc_4$. En effet, ces deux séquences de transitions correspondent alors à des changements de contexte de sc_1 vers sc_5 et de sc_2 vers sc_4 qui n'ont pas eu lieu. On voit que ce problème est aisément éliminé si l'on tient compte des processus en plus des transitions.

Le second problème concerne les séquences qui peuvent manquer, avec le raisonnement précédent, si l'on ne tient pas compte de l'héritage des processus. La figure 5.6 illustre alors ce problème. Dans cette figure, un processus P_1, étiqueté par sc_1, transite vers un contexte sc_2 puis crée un fils. Ce fils transite ensuite vers sc_3. En appliquant le raisonnement précédent, les seules séquences recomposées seraient $sc_1 \xrightarrow{trans} sc_2$ et $sc_2 \xrightarrow{trans} sc_3$. Or comme P_2 est un fils de P_1, il y a bien une transition de sc_1 vers sc_3. Ce problème est alors réglé en tenant compte de l'héritage des processus.

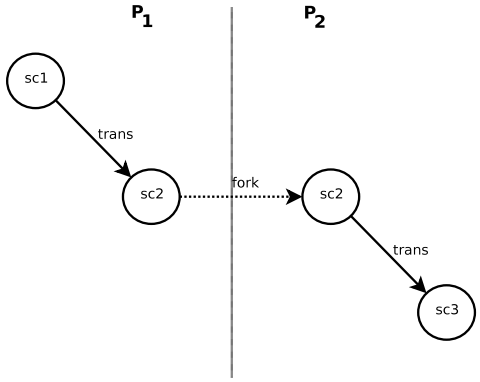


FIG. 5.6 – Héritage de processus

Ces deux problèmes sont traités par une méthode de reconstruction particulière, le *reconstructeur arborescent*, qui tient compte de l'héritage des processus. En ce qui concerne la reconstruction des autres types de séquences, nous n'avons pas à tenir compte de l'héritage puisqu'il faut justement prendre en compte les interactions entre tous les processus du système. Pour les autres types de séquences, nous utilisons un reconstructeur système où les processus n'interviennent pas lors de la reconstruction. Ainsi nous proposons deux méthodes de reconstruction des activités :

Reconstructeur Système : permet de reconnaître une activité qui apparaît sur le système. Cette activité ne différencie pas les processus mis-en-jeu ;

Reconstructeur Arborescent : permet de reconnaître une activité qui met en jeu une descendance de processus.

Nous allons maintenant détailler le fonctionnement de ces deux reconstructeurs.

5.4.2.2 Reconstructeur Système

La reconstruction système utilise les activités de la base de signatures et les traces pour calculer les activités en cours d'observation. Cette reconstruction nécessite donc la définition d'une liste d'activités en cours d'observation : `Liste<Observateur>` OBS_{sys}. Une signature (activité) de la base est alors ajoutée à cette liste lorsque la première interaction de la signature correspond à la trace. Ainsi, dans ce reconstructeur, il n'y a qu'un seul ensemble d'observateurs (OBS_{sys}), dit "global", initialisé avant l'analyse des traces et partagé par l'ensemble des processus du système. La trace analysée permet aussi de faire évoluer les activités en cours d'observation. Une alerte est alors générée lorsqu'une activité est totalement observée. Tous les processus du système participeront donc à l'évolution des activités de cet ensemble. Nous définissons ainsi la classe `Reconstructeur_Systeme` qui correspond à

ce type de reconstructeur.

```
Classe Reconstructeur_Systeme etend Reconstructeur
reconstruire(TRACE t, ACT ⊂ BASE) début
    pour chaque act ∈ ACT faire
        si t.it == act.get(1) alors
            OBS_sys.ajouter(act);
        fin
    fin
    pour chaque act ∈ OBS_sys faire
        act.analyse(t);
        si act.observe() alors
            generer_alerte(act, t) ;
        fin
    fin
fin
```

Cette classe possède une seule méthode `reconstruire` qui prend en paramètres une trace et un ensemble de signatures (les activités nécessitant une reconstruction système). Tout d'abord, cette méthode ajoute les nouveaux observateurs. La trace t est ensuite analysée par chaque activité présente dans la liste OBS_{sys}. Lorsqu'une activité est observée, c'est-à-dire complètement réalisée, une alerte est alors générée.

5.4.2.3 Reconstructeur Arborescent

Afin de prendre en compte la descendance des processus, nous définissons un second reconstructeur que nous appellerons *reconstructeur Arborescent*. Lors de l'analyse des traces, ce reconstructeur construit un arbre "généalogique" des processus du système : `Arbre` *arbre_proc*. L'arbre de processus est alors initialisé (via l'appel à la méthode `init_arbre()`) qui crée un nouvel arbre ayant comme nœud racine le processus initial du système. Par exemple, sous GNU/Linux le processus initial correspond à *init* et son *PID* vaut 1.

La construction de cet arbre nécessite de prendre en compte les opérations élémentaires de création et de destruction de processus. Chaque nœud de cet arbre correspond ainsi à un processus du système identifié par son *PID* et il est relié à son père identifié par le numéro *PPID*. De plus, chaque nœud de l'arbre stocke sa propre liste d'observateurs OBS_{arb}, dite "locale" au nœud. Les observateurs d'un nœud n'évoluent que si la trace correspond au processus du nœud. Lors de l'ajout d'un nouveau nœud, correspondant à l'apparition d'un nouveau processus, le nœud fils hérite d'une copie des observateurs OBS_{arb} de son père. L'ajout d'un nouveau nœud dans l'arbre est alors réalisé par la procédure suivante :

```
ajouter_proc(arbre, t)
début
    arbre.ajouter_noeud(t.ppid, t.pid);
    arbre.get(t.pid).OBS_arb := PROC_TREE.get(t.ppid).OBS_arb.copy();
fin
```

Cette procédure ajoute le nouveau nœud correspondant au processus fils de la trace, puis elle initialise l'ensemble OBS_{arb} du nouveau nœud avec une copie de celui du processus père.

La reconstruction des activités est alors réalisée par la méthode `reconstruire` de la classe `Reconstructeur_Arborescent`. Cette méthode prend en paramètres une trace et une base de signatures. Si la trace correspond à un nouveau processus, ce processus est alors ajouté à l'arbre des processus et hérite d'une copie des activités en cours de son père. L'appel à OBS_{arb} := *arbre_proc*.get(*t.pid*) renvoie alors les observateurs locaux du processus courant (le processus ayant réalisé l'interaction). Ensuite,

les nouvelles activités sont ajoutées à cet ensemble d'observateurs. Finalement, seuls les observateurs de cet ensemble évoluent. Une alerte est alors générée pour chaque activité totalement observée.

```
Classe Reconstructeur_Arborescent etend Reconstructeur
global arbre_proc;
reconstruire(TRACE t, ACT ⊂ BASE) début
    si t.p ∉ arbre_proc alors
        ajouter_proc(arbre_proc, t.ppid, t.p);
    fin
    OBS_arb := arbre_proc.get(t.pid);
    pour chaque act ∈ ACT faire
        si t.it == act.get(1) alors
            OBS_arb.ajouter(act);
        fin
    fin
    pour chaque act ∈ OBS_arb faire
        act.analyse(t);
        si act.observe() alors
            generer_alerte(act, t);
        fin
    fin
fin
```

Notons que l'algorithme présentée ici est une version simplifiée de l'algorithme réel de reconstruction. Notamment, cette version ne prend pas en compte la destruction des processus.

Nous pouvons finalement remarquer que les séquences que le reconstructeur arborescent peut reconstruire peuvent aussi être reconstruites par le reconstructeur système. En effet, le reconstructeur système calcule toutes les séquences alors que le reconstructeur arborescent calcule uniquement les séquences d'une descendance de processus. Bien que le reconstructeur système est suffisant pour reconstruire n'importe quel type de séquence, nous avons vu dans la section 5.4.2.1 que le reconstructeur arborescent permet d'éliminer les fausses séquences de transitions. C'est pourquoi c'est ce reconstructeur qui est utilisé pour reconstruire les *séquences de transitions*.

5.4.3 Algorithme d'analyse des traces d'interactions

Les deux reconstructeurs, que nous venons de décrire, sont utilisés pour analyser les traces d'interactions d'un système en fonction d'une base de signatures. Nous proposons donc l'algorithme 9 d'analyse des traces d'interactions. Cet algorithme prend en paramètres un ensemble de traces T et une base de signatures $BASE$. Une liste d'observateurs (OBS_{sys}) et un arbre de processus ($arbre\_proc$) sont tout d'abord initialisés afin de créer un *reconstructeur système* et un *reconstructeur arborescent*.

De plus, deux listes de signatures (ACT_{sys} et ACT_{arb}) sont créées. Ces deux listes permettent de séparer les signatures nécessitant un reconstructeur système, des signatures nécessitant un reconstructeur arborescent (les séquences de transitions, via la méthode `estTransition()`).

Finalement, pour chaque trace de l'ensemble des traces T, la méthode `reconstruire` des deux reconstructeurs est appelée pour la trace t et le sous ensemble des signatures alloué à ce reconstructeur. Les reconstructeurs généreront alors des alertes pour chaque activité correspondant à une signature de

la base des signatures.

```
detection(T, BASE)
début
    List<Activite> OBS_sys := ∅;
    Arbre arbre_proc :=init_arbre();
    Reconstructeur reconstructeur_systeme;
    Reconstructeur reconstructeur_arborescent;
    List<Activite> ACT_sys := ∅;
    List<Activite> ACT_arb := ∅;

    pour chaque sig ∈ BASE faire
        Activite act := init(sig) si sig.estTransition() alors
            ACT_arb.ajouter(act);
        fin
        sinon
            ACT_sys.ajouter(act);
        fin
    fin

    pour chaque t ∈ T faire
        reconstructeur_systeme.reconstruire(t, ACT_sys);
        reconstructeur_arborescent.reconstruire(t, ACT_arb);
    fin
fin
```

Algorithme 9 : Algorithme d'analyse des traces d'interactions.

5.4.4 Algorithme général de détection

Jusqu'ici, nous avons défini les trois phases de notre approche de détection des violations d'un ensemble de propriétés de sécurité. L'algorithme 10 synthètise l'application de ces trois phase et il représente ainsi notre algorithme général de détection. Ainsi, il utilise les trois algorithmes associés aux trois phases de notre approche de détection. Notre approche nécessite la définition d'une politique de contrôle d'accès, une politique de détection, trois tables de changement d'état et un ensemble de traces obtenu lors de l'exécution du système. La politique de contrôle d'accès est alors exprimée sous forme d'un ensemble de vecteurs d'interactions $\mathcal{POL}$. Les trois tables de changement d'état $T_{dep}, T_{flux}, T_{trans}$ permettent alors de construire un ensemble de graphes (phase 1) associés à cette politique de contrôle d'accès. Ces graphes correspondent ainsi au graphe d'interactions G_{int}, au graphe de dépendance causale G_{dep}, au graphe de flux d'informations G_{flux} et au graphe de transitions G_{flux}.

La seconde phase utilise alors ces graphes et une politique de détection $\mathcal{POL}_{detection}$ qui définit l'ensemble des propriétés de sécurité nécessaires. La politique de détection définit alors l'ensemble des règles d'analyse de ces graphes. Cette phase génère donc une base de signatures $BASE$ contenant l'ensemble des activités violant les propriétés de sécurité.

La troisième phase permet, par analyse des traces d'interactions du système T, de détecter l'apparition des éléments de la base de signatures $BASE$. Cette phase utilise les reconstructeurs afin de génèrer une alerte pour chaque signature observée. L'observation d'une activité correspondant à une signature de la base est alors une violation d'une des propriétés de sécurité définies dans la politique

de détection.

```
detection_globale(POL ⊂ IV, POL_detection, T_dep, T_flux, T_trans, T)
début
    G_int, G_dep, G_flux, G_trans :=construire_graphes(POL ⊂ IV, T_dep, T_flux, T_trans);
    BASE :=construire_base(POL ⊂ IV, POL_detection, G_int, G_dep, G_flux, G_trans);
    detection(T, BASE);
fin
```

Algorithme 10 : Algorithme général de détection

5.5 Conclusion

L'algorithme 10 résume notre approche. Nous pouvons remarquer que les trois phases proposées dans ce chapitre peuvent être utilisées séparément. Les phases de *construction des graphes* et de *construction de la base de signatures* doivent être appelées avant la phase de *détection*. Cependant, une phase s'applique indépendemment tant que les données de la phase précédente n'ont pas été modifiées.

Ainsi sur un système, ou un ensemble de système, nous n'avons pas besoin de reconstruire les graphes et la base tant que la *politique de contrôle d'accès* et la *politique de détection* sont fixes. De plus, nous conservons l'état des observations en cours afin d'éviter tout faux négatif lié au redémarrage du système (un flux d'informations dont chaque moitié est réalisée avant et après le redémarrage).

Finalement, lorsque la politique de contrôle d'accès est fixe, les modifications de la politique de détection n'impliquent que la reconstruction de la base. Cette nouvelle base sert alors de nouvelle entrée à la phase de détection et l'ancienne base peut être conservée afin de terminer la reconstruction des observations en cours.

Nous proposons, dans le chapitre suivant, une implantation de cette méthode de détection pour deux systèmes de contrôle d'accès : SELinux et GRSECURITY. Nous conclurons alors ce chapitre par les résultats de l'expérimentation de cette implantation.

Chapitre 6

Implantation et Expérimentation

Dans le chapitre 5, nous avons proposé une approche de détection des activités illicites sur un système. Cette approche utilise la politique de contrôle d'accès d'un système et nécessite la définition d'une politique de détection. Cette politique de détection définit un ensemble de propriétés de sécurité qui doivent être respectées par ce système. L'approche d'analyse d'une politique de contrôle d'accès, proposée dans le chapitre 4, permet alors d'énumérer l'ensemble des violations possibles de ces propriétés de sécurité. Ces violations correspondent à des activités décrites dans notre langage de description d'activités (cf. chapitre 3). Ainsi, nous proposons un outil, que nous appelons PIGA (pour *Policy Interaction Graph Analysis*), qui implante cette approche de détection. Cet outil, écrit en `Java`, implante les trois phases de détection proposées dans le chapitre 4.

Dans ce chapitre, nous proposons de décrire l'implantation de cet outil et l'expérimentation qui en a été faite. Nous proposons ainsi d'appliquer notre approche à deux systèmes cibles qui sont SELinux et GRSECURITY. Dans la section 6.1, nous détaillons le mécanisme de conversion d'une politique SELinux ou GRSECURITY en un ensemble de vecteurs d'interactions. La section 6.2 décrit le fonctionnement des trois phases de détection. Finalement, dans la section 6.3, nous proposons d'étudier une application de cet IDS dans le cadre des *pots-de-miel*.

L'implantation de notre approche de détection d'intrusions, développée dans ce chapitre, a également été exposée dans les articles [Briffaut *et al.* 2005b, Blanc *et al.* 2006, Briffaut *et al.* 2006b, Briffaut *et al.* 2006c, Briffaut *et al.* 2006a].

6.1 Conversion d'une politique en langage neutre

Dans notre approche, la *politique de contrôle d'accès* d'un système cible doit tout d'abord être projetée en une *politique neutre* qui consiste en un ensemble de vecteurs d'interactions $\mathcal{POL} \subset IV$. Comme l'indique la figure 6.1, cette projection est alors réalisée par un mécanisme de traduction de la politique cible. Ainsi, nous proposons deux outils afin de convertir les politiques SELinux et GRSECURITY en une politique neutre.

Dans cette partie, nous donnons tout d'abord le *langage neutre* utilisé pour définir les vecteurs d'interactions. Puis, pour chaque système cible, nous revenons sur les langages propres à ces systèmes. Nous montrons ensuite comment projeter le langage cible dans le langage neutre. Finalement, nous présentons des exemples pour SELinux et GRSECURITY et nous étudierons les politiques neutres associées.

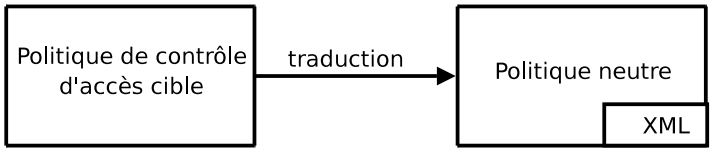


FIG. 6.1 – Projection d'une politique de contrôle d'accès cible en politique neutre.

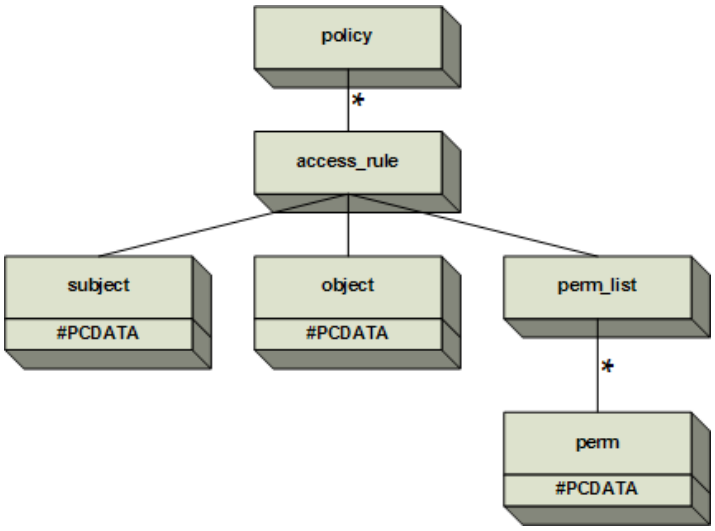


FIG. 6.2 – DTD pour la politique neutre.

6.1.1 Langage neutre d'expression d'une politique de contrôle d'accès

La définition de la syntaxe du langage neutre utilise le langage XML et les DTD. La DTD pour le format XML de cette politique neutre, donnée en annexe C, est illustrée par la figure 6.2. L'élément racine `policy` correspond à la description d'une politique de contrôle d'accès. Au-dessous, l'élément fils est `access_rule`, avec une relation de type $\star$ qui indique que dans le document XML, le nombre d'éléments `access_rule` n'est pas limité. L'élément `access_rule` représente en fait un vecteur d'interactions. Le format de ces vecteurs correspond à un ensemble d'opérations élémentaires entre deux contextes ($sc_{subject}$ et sc_{object}) (cf. def. 4.1.1, page 90).

La DTD contient donc les trois éléments suivants : `subject` désigne le contexte de sécurité sujet $sc_{subject}$, `object` le contexte de sécurité objet sc_{object}, et `perm_list` représente is. Ce dernier étant un ensemble d'opérations élémentaires, l'élément fils `perm` est donc lié à `perm_list` par une relation $\star$. Les éléments `perm` peuvent contenir eux-mêmes une liste de droits d'accès du type `{file : read, write}`. L'intérêt d'avoir une liste est d'autoriser la définition de plusieurs classes de permissions entre deux contextes.

Le listing 6.1 représente la forme neutre de l'exemple de politique proposée dans le listing 4.1 (cf. section 4.1.4, page 93). Les lignes 4 à 10 représentent un vecteur d'interactions qui autorise la transition du contexte de sécurité `(system_u, system_r, login_d)` vers le contexte `(root, admin_r, admin_d)`. Ce vecteur contient donc l'opération de transition : `{process : transition}`. Les lignes 11 à 18 représentent un autre vecteur d'interactions qui autorise le contexte `(system_u, system_r, apache_d)` à lire, écrire ou exécuter tout fichier ayant le contexte `(root, object_r, var_www_t)` et à lire ou écrire dans tout répertoire ayant ce

même contexte.

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <!DOCTYPE policy SYSTEM 'meta.dtd'>
3  <policy>
4     <access_rule>
5        <subject>(system_u, system_r, login_d)</subject>
6        <object>(root, admin_r, admin_d)</object>
7        <perm_list>
8           <perm>{process : transition}</perm>
9        </perm_list>
10    </access_rule>
11    <access_rule>
12       <subject>(system_u, system_r, apache_d)</subject>
13       <object>(system_u, object_r, var_www_t)</object>
14       <perm_list>
15          <perm>{file : read, write, execute}</perm>
16          <perm>{dir : read, write}</perm>
17       </perm_list>
18    </access_rule>
19 [...]
20 </policy>
```

Listing 6.1 – Politique neutre en XML.

Ce format XML permet de définir une politique sous forme neutre. C'est cette forme neutre qui est utilisée en entrée de la phase de construction du *graphe d'interactions*. Nous allons maintenant décrire comment les politiques cibles SELinux et GRSECURITY sont converties en politique neutre.

6.1.2 Application à SELinux

SELinux est une implantation de contrôle d'accès obligatoire pour Linux, qui intègre notamment les modèles RBAC et DTE. Dans cette partie, nous étudions tout d'abord la syntaxe du langage d'expressions de politique SELinux. Puis, nous montrerons comment notre outil projette une politique SELinux en une politique neutre. Une description de SELinux est proposé dans l'annexe D.1 (page 195).

Politique cible SELinux La ligne 1 correspond à la déclaration d'un rôle (`admin_r`) et autorise le type `admin_d` à utiliser ce rôle. Les lignes 3 à 5 sont des déclarations de type (règles commençant par `type`). Une déclaration de type contient le mot clé `type`, puis le nom du type suivis d'un ensemble d'attributs. Par exemple, la ligne 3 déclare un type `httpd_d` qui possède les attributs `domain, privlog, daemon, privmail`. Les types sujets se distinguent par la présence d'un attribut `domain`, les autres types qui ne possèdent pas l'attribut `domain` sont alors considérés comme des objets. Ainsi, le type `httpd_d` est un sujet, les types `httpd_exec_t` et `service_u_exec_t` sont des objets.

Le listing 6.2 contient un exemple de politique SELinux. Les lignes 7 à 11 correspondent à des règles de contrôle d'accès (règles commençant par `allow`). Ces règles correspondent à des interactions autorisées entre un *type* sujet et un *type* objet (le *type* étant un attribut d'un contexte). Ainsi, nous pouvons voir que la définition des règles de contrôle d'accès SELinux est basée sur les *types*, elle ne tient compte ni des *rôles*, ni des *utilisateurs*. Nous verrons que le fait que les règles ne portent pas sur le contexte complet, c'est-à-dire le triplet (`u:r:t`), peut conduire à des failles de protection. Les interactions sont composées d'opérations élémentaires, par exemple `file{read}`. Ces opérations élémentaires correspondent aux appels système disponibles dans GNU/Linux et aux opérations

```
1  role admin_r types admin_d;
2   [...]
3  type httpd_d, domain, privlog, daemon, privmail;
4  type httpd_exec_t, file_type, exec_type;
5  type service_u_exec_t, file_type, exec_type;
6   [...]
7  allow httpd_d httpd_exec_t:file { read getattr lock execute ioctl entrypoint };
8  allow httpd_d webserv_d:process transition;
9  allow webserv_d user_info_t:file { read };
10 allow webserv_d user_info_t:dir{ read };
11 allow user_d webserv_d:process transition;
12  [...]
13 type_transition httpd_d httpd_php_exec_t:process httpd_php_d;
14 type_transition httpd_d webserv_exec_t:process webserv_d;
15  [...]
```

Listing 6.2 – Extrait de politique SELinux.

liées au mécanisme de contrôle d'accès (par exemple, l'opération `process{ transition }`) qui induit un changement de contexte).

Enfin les lignes 13 et 14 présentent des règles de transition (règles commençant par `type_transition`). Il s'agit de règles qui permettent d'automatiser les changements de contexte de sécurité. Par exemple, pour la ligne 13, lorsqu'un processus ayant le type `httpd_d` exécute un fichier de type `httpd_php_exec_t`, le mécanisme de contrôle d'accès étiquette automatiquement le processus avec le contexte `httpd_php_d`.

6.1.2.1 Projection en langage neutre

La projection d'une politique cible SELinux en langage neutre se fait par énumération des contextes puis des vecteurs d'interactions. Ces vecteurs sont ensuite traduits en XML afin de former la politique neutre. Dans cette partie, nous détaillons donc comment obtenir les contextes et les vecteurs d'interactions d'une politique SELinux.

A) Énumération des contextes de sécurité Un contexte de sécurité SELinux est constitué de trois attributs : une *identité*, un *rôle* et un *type*. Le début d'une politique SELinux contient la déclaration des *rôles* et *types*. De plus, un fichier de configuration (`seusers`) associe à chaque utilisateur un ensemble de rôles. Ces déclarations permettent d'énumérer l'ensemble des contextes sujets. L'énumération des contextes de sécurité sujets est donc réalisée comme suit :

- pour chaque utilisateur u,
- pour chaque rôle r accessible par u,
- pour chaque type t accessible par r, on obtient un contexte sujet `u:r:t`.

Par exemple, la politique du listing 6.2 définit le rôle `admin_r` qui peut accéder au type `admin_d` (ligne 1). Supposons que le fichier `seusers` associe le rôle `admin_r` à l'utilisateur `root`, nous obtenons un contexte sujet : `root:admin_r:admin_d`

```
/var/www          -d    system_u:object_r:var_www_t
/etc/apache2      -d    system_u:object_r:apache_conf_t
[...]
```

Listing 6.3 – Extrait du fichier `file_contexts`.

Les contextes de sécurité objets sont déclarés dans un fichier spécifique : `file_contexts`. Le listing 6.3 contient un exemple de ce fichier. Par exemple ce fichier déclare, sur la ligne 1, le contexte `system_u:object_r:var_www_t` associé au répertoire `/var/www`. Ce fichier fournit directement l'ensemble des contextes de sécurité objets pour tous les objets du système. En effet, sous `unix` toutes les ressources accessibles par les processus sont désignées avec un nom de fichier. Par exemple, le répertoire `/proc` permet d'accéder aux ressources du noyau. Ainsi le fichier `file_contexts` définit les contextes de l'ensemble des ressources d'un système.

B) Énumération des vecteurs d'interactions Comme indiqué dans le listing 6.2, une règle `allow` définit un ensemble d'opérations élémentaires autorisées entre deux *types*. Un vecteur d'interactions, entre deux contextes de sécurité, correspond donc à un ensemble de règles `allow` entre les deux *types* de ces contextes. Par exemple, les lignes 9 et 10 correspondent à deux ensembles d'opérations entre le type `webserv_d` et le contexte `user_info_t`. Ainsi, pour deux contextes `(system_u:system_r:webserv_d)` et `(user_u:object_r:user_info_t)`, ces deux lignes permettent de définir le vecteur `(system_u:system_r:webserv_d, user_u :user_r:user_info_t, {file:read, dir:read})`. Dans la politique neutre, ce vecteur correspond à :

```
<access_rule>
   <subject>(system_u, system_r, webserv_d)</subject>
   <object>(user_u, object_r, user_info_t)</object>
   <perm_list>
      <perm>{file : read}</perm>
      <perm>{dir : read}</perm>
   </perm_list>
</access_rule>
```

L'ensemble des vecteurs d'interactions est ainsi obtenu par énumération de tous les ensembles d'opérations élémentaire déclarés entre chaque couple de contextes. Cette énumération est transcrite dans le format XML (cf. section 6.1.1), ce qui fournit la politique neutre.

6.1.2.2 Exemples de conversion

Nous avons écrit un outil (en C et utilisant la librairie *selinux*) qui convertit une politique SELinux dans le langage neutre. Tout d'abord, cet outil calcule l'ensemble des contextes de sécurité, puis l'ensemble des vecteurs d'interactions, en utilisant les deux méthodes d'énumération décrites précédemment. Cet outil prend en entrée une politique de contrôle d'accès SELinux et génère une politique neutre (au format XML).

La table 6.1 présente les caractéristiques de politiques SELinux et des langages neutres obtenus. Dans cet exemple, nous considérons six machines utilisant des distributions différentes et du type de services fournis (serveur ou utilisateur). Afin de comparer nos résultats, nous considérons trois types d'installation :

- *Minimale* : installation de base sans service ni application utilisateur ;
- *Standard* : installation avec les services et applications classiques ;
- *Complète* : tous les services et applications disponibles sont installés.

La machine *SEL 5* correspond à une installation minimale de GNU/Linux. *SEL 1, 3, 4* correspondent à des installations standards. La machine *SEL 2* correspond à une installation complète. Du fait de la taille de la politique obtenue, ce dernier cas correspond donc *au pire cas* de notre étude. Finalement, la machine *SEL 6* est une machine standard utilisant un nouveau système de gestion des politiques

Nom		SEL 1	SEL 2	SEL 3	SEL 4	SEL 5	SEL 6
Description	Distribution	Gentoo	Debian	Gentoo/vmware	Gentoo	Gentoo	Gentoo
	Version	2004	Etch	2004	2005	2005	2006
	Type	Serveur	Serveur Utilisateur	Serveur	Serveur	Utilisateur	Serveur
	Installation	Standard	Complète	Standard	Standard	Minimale	Standard
Politique Cible	Utilisateur	3	3	5	3	3	5
	Rôle	5	6	5	5	5	5
	Type	637	2086	637	656	523	871
	Règle	17 639	235 691	17 650	19 497	11 655	22 465
	Temps	47s	10min31s	1min2s	52s	27s	2min39
Politique Neutre	Contexte objet	438	1940	459	452	351	614
	Contexte sujet	139	1077	165	143	121	264
	Contexte	577	3017	624	595	472	878
	Vecteurs	17 684	314 582	21 359	18 215	13 843	58 483

TAB. 6.1 – Exemple de conversion de politique SELinux

SELinux (gestion modulaire). Cette dernière machine comprend donc plus de règles que les versions antérieures équivalentes (version 2004 ou 2005).

Dans la table 6.1, nous pouvons constater qu'une politique SELinux définit généralement 3 à 5 utilisateurs (attribut *identité*) et 5 ou 6 *rôles*. Concernant les *types*, une installation minimale ou standard contient environ 650 types alors qu'une installation complète en définit plus de 2000. Finalement, une politique SELinux définit entre 10,000 et 250,000 règles, pour une valeur standard de 20,000 règles lors d'une installation standard.

L'outil de conversion nécessite ainsi moins de deux minutes pour effectuer la conversion de politiques minimales ou standards. Le pire cas étant celui de l'installation complète qui, de part ses 235,691 règles, nécessite environ 10 minutes de traitement.

Nous obtenons ainsi des politiques neutres contenant en moyenne 420 contextes de sécurité objets pour une politique standard et 150 contextes de sécurité sujets. Le nombre moyen de vecteurs d'interactions est ainsi de 17,000 pour 570 contextes. Le pire cas contient alors 3,017 contextes pour 314,582 vecteurs d'interactions, ce qui explique les 10 minutes de traitement nécessaires.

6.1.3 Application à GRSECURITY

Comme SELinux, GRSECURITY est une implantation de contrôle d'accès obligatoire. GRSECURITY implante une version allégée des modèles RBAC et DTE. Dans cette partie, nous étudions tout d'abord la syntaxe du langage d'expressions de politique GRSECURITY. Puis, nous montrerons comment notre outil projette une politique GRSECURITY en une politique neutre. Une description de GRSECURITY est proposé dans l'annexe D.2 (page 198).

6.1.3.1 Politique cible GRSECURITY

Le listing 6.4 contient un exemple de politique cible GRSECURITY. Les lignes 1 à 5 définissent les règles associées au rôle `system_r` (déclaré à la ligne 1). Une déclaration de rôle débute par le mot clé `role`, suivie du nom du rôle puis d'un ensemble d'attributs. Cette déclaration est valide jusqu'à la prochaine déclaration de rôle et permet de définir toutes les règles associées à ce rôle. Les lignes commençant par `subject` correspondent aux règles d'accès, qui ont pour contexte sujet

```
1 role system_r G
2 role_transitions user_r, admin_r
3 subject /usr/bin/httpd
4    /var/www  rwe
5    /etc/apache2 r
6 [...]
```

Listing 6.4 – Extrait de configuration grsecurity.

le programme qui suit le mot-clé `subject`, et pour contexte objet chacune des lignes suivantes, jusqu'au prochain mot-clé `subject`. Par exemple, la ligne 5 définit un accès en lecture (`r`) entre le processus `/usr/bin/httpd` et les fichiers contenus dans le répertoire `/etc/apache2`. Cette règle concerne le rôle englobant `system_r`.

Les lignes débutants par `role_transitions` définissent les changements de contextes légaux. Par exemple, la ligne 2 autorise le contexte ayant le rôle `system_r` à transiter vers le rôle `user_r` ou `admin_r` .

6.1.3.2 Projection en langage neutre

Comme pour SELinux, la projection d'une politique GRSECURITY se fait par énumération des contextes et des vecteurs d'interactions. Ces vecteurs sont ensuite traduits en XML afin d'obtenir une politique neutre. Dans cette partie, nous détaillons donc comment énumérer les contextes et les vecteurs d'interactions d'une politique GRSECURITY.

A) Énumération des contextes de sécurité GRSECURITY ne définit pas les contextes de sécurité de la même façon que SELinux. GRSECURITY désigne les contextes par leur chemin dans le système de fichier. Pour comparaison, SELinux désigne le processus du serveur Apache par le contexte `(system_u, system_r, httpd_d)` et GRSECURITY le désigne simplement par le chemin du fichier exécutable d'Apache, soit `/usr/bin/httpd`. L'avantage pour GRSECURITY est que la désignation des contextes est directe. Cependant, comme les contextes de sécurité permettent des classes d'équivalence dans SELinux, il faudra davantage de règles dans GRSECURITY pour écrire une politique équivalente à celle de SELinux.

De ce fait, nous avons besoin d'un mécanisme pour traduire les chemins en contextes sujets ou objets. Pour cela nous proposons une solution de traduction des contextes GRSECURITY en contextes neutres.

a) Énumération des contextes objets Nous utilisons le fichier SELinux `file_contexts` (cf. listing 6.3) pour donner la correspondance entre un contexte GRSECURITY (chemin) et un contexte neutre (le triplet `u:r:t` inclut dans le fichier `file_contexts`). Si nous prenons la troisième ligne de la politique 6.4, le chemin correspond à `/var/www`. Dans le fichier `file_contexts` du listing 6.3, la ligne 1 correspond à `/var/www` et donne le contexte neutre `system_u:object_r:var_www_t`. Cette méthode permet ainsi d'énumérer l'ensemble des contextes objets de notre politique neutre.

b) Énumération des contextes sujets Pour énumérer les sujets, nous réutilisons les fichiers de configuration de SELinux : `file_contexts` et `policy.conf`. Le fichier `policy.conf` permet d'énumérer tous les types exécutables (qui possèdent l'attribut `file_exec`). Cette énumération

permet d'obtenir, via le fichier `file_contexts`, tous les chemins associés à ces types. Finalement, le fichier `policy.conf` permet d'obtenir le type sujet (domaine) associé aux types exécutables. Ainsi, nous obtenons une association (chemin, type sujet). Toutes ces associations sont stockées dans le fichier `sc_contexts`, tel que l'illustre le listing 6.5. Dans ce listing, chaque ligne définit un chemin et le type sujet associé. L'énumération des contextes de sécurité sujets se fait alors de la manière

```
/usr/sbin/httpd      httpd_d
/usr/sbin/ssh     ssh_d
/usr/bin/login    login_d
[...]
```

Listing 6.5 – Table `sc_contexts` de correspondance entre chemins et types sujets..

suivante :

- Pour chaque chemin, on extrait le type t qui lui est associé (avec la table `sc_contexts`) ;
- Pour chaque rôle r accédant à ce chemin (cf. listing 6.4), on énumère les utilisateurs u accédant à ce rôle (via le fichier SELinux `seusers`) afin d'obtenir tous les contextes sujet `u:r:t` correspondant à ce chemin.

Cette méthode permet ainsi d'énumérer l'ensemble des contextes sujets d'une politique neutre. Par exemple, le listing 6.4 déclare le sujet `/usr/bin/httpd`, qui correspond au type `httpd_d` d'après la table `sc_contexts`. Ce sujet est déclaré dans rôle `system_r`, si l'utilisateur `system_u` peut accéder à ce rôle (cf. fichier `seusers`), nous obtenons alors le contexte : `system_u:system_r:httpd_d`.

B) Énumération des vecteurs d'interactions Le format des règles d'accès de GRSECURITY est simple (cf. listing 6.4) : on trouve une première ligne commençant par `subject` déclarant un chemin sujet, les lignes suivantes désignent les chemins objets et les permissions d'accès. Afin de convertir ces permissions sous forme neutre, nous utilisons une table de traduction. Le listing 6.6 contient un extrait de cette table.

```
r     file : read
w     file : write
x     file : execute
[...]
```

Listing 6.6 – Extrait de la table de traduction des permissions GRSECURITY en opérations neutres.

La conversion d'une règle GRSECURITY en un vecteur neutre est réalisée de la manière suivante :

- Les permissions GRSECURITY sont traduites en forme neutre via la table de traduction 6.6 ;
- Les sujets GRSECURITY sont traduits en forme neutre via l'énumération des sujets ;
- Les objets GRSECURITY sont traduits en forme neutre via l'énumération des objets.

Par exemple, les permissions de la ligne 4 sont converties en l'ensemble $is = \{file : read, write, execute\}$. Le sujet et l'objet GRSECURITY sont convertis en forme neutre `system_u:system_r:httpd_d` et `system_u:object_r:var_www_t` par les méthodes d'énumérations précédentes. Nous obtenons donc le vecteur : `(system_u:system_r:httpd_d, system_u:object_r:var_www_t, { file:read, write, execute })`. Ce vecteur est alors traduit en XML :

Nom		GR 1	GR 2	GR 3	GR 4
Description	Distribution	Gentoo	Défaut	Fedora	Fedora
	Type	Serveur/Utilisateur	Serveur	Serveur	Serveur
	Installation	Complète	Minimale	Standard	Standard
Politique Cible	Sujet	201	21	18	21
	Rôle	19	2	9	13
	Règle	6563	81	509	925
	Temps	1min33s	6s	9s	14s
Politique Neutre	Contexte objet	104	34	44	46
	Contexte sujet	104	5	16	24
	Contexte	208	39	60	70
	Vecteur	1399	89	206	318

TAB. 6.2 – Exemple de conversion de politique GRSECURITY

```
<access_rule>
   <subject>(system_u, system_r, httpd_d)</subject>
   <object>(system_u, object_r, var_www_t)</object>
   <perm_list>
      <perm>{file : read,write,execute}</perm>
   </perm_list>
</access_rule>
```

En appliquant cette méthode sur l'ensemble de la politique GRSECURITY, nous obtenons la politique neutre au format XML.

6.1.3.3 Exemples de conversion

Nous avons écrit un outil (en *shell-script*) qui permet de convertir une politique au format GRSECURITY en politique neutre. Nous avons appliqué cet outil sur quatre politiques GRSECURITY provenant de quatre machines différentes. La table 6.2 contient une synthèse des résultats obtenus lors de l'utilisation de cet outil. Nous considérons ainsi quatre politiques :

- *GR 1* est une politique qui provient d'une machine contenant des applications serveurs (serveur Web, mail, etc.) et utilisateurs (environnement X, outils de développement, etc.) ;
- *GR 2* est la politique livrée avec GRSECURITY ;
- *GR 3* est la politique d'un serveur mail ;
- *GR 4* est la politique d'un serveur Web et DNS.

Les politiques 1, 3 et 4 ont été obtenues avec la phase d'apprentissage de GRSECURITY, la politique 2 a été définie manuellement par le créateur de GRSECURITY.

Une politique standard (GR 3 et 4) définit en moyenne 20 sujets, 10 rôles et moins de 1000 règles de contrôle d'accès. Notre exemple de politique complexe (GR 1) définit 201 sujets (un par application) pour 19 rôles et contient 6563 règles de contrôle d'accès. Finalement, la politique livrée avec GRSECURITY (GR 2) définit 21 sujets et 2 rôles (administrateur et utilisateur) pour un total de 81 règles.

Pour effectuer la conversion du langage de GRSECURITY vers notre langage neutre, notre outil nécessite quelques secondes pour une politique standard à quelques minutes pour une politique complexe. La politique neutre résultante contient ainsi moins de 100 contextes pour une politique standard à plus de 200 pour une politique complexe.

6.1.4 Discussion

Nous pouvons tout d'abord remarquer qu'une politique GRSECURITY définit beaucoup moins de règles qu'une politique SELinux. Ceci vient du fait qu'une politique GRSECURITY est obtenue par apprentissage alors qu'une politique SELinux est générique. Elle est donc plus complète et adaptée à un parc de machines. Une politique SELinux contient des règles qui ne sont pas nécessairement utiles à chaque machine.

De plus, SELinux comprend un ensemble d'opérations élémentaires beaucoup plus important que GRSECURITY (500 contre 10). SELinux permet ainsi de contrôler plus finement les appels système que GRSECURITY. En contrepartie, une politique SELinux nécessite la définition de beaucoup de règles et elle est donc beaucoup plus complexe à définir.

Les conversions, présentées dans cette section, ont été obtenues sur une machine de bureau équipée d'un processeur à 3Ghz et de 2Go de mémoire vive. Étant donné que cette phase de conversion n'est nécessaire que lorsque la politique de contrôle d'accès est modifiée, ces expérimentations montrent que les temps de calcul sont raisonnables et que cette phase peut être réalisée aisément.

Dans les sections suivantes, nous détaillons le fonctionnement de l'outil, nommé PIGA, qui implante la solution de détection définie dans le chapitre 5 (page 127) et met donc en œuvre les trois phases définies : 1) construction des graphes, 2) génération de la base de signatures, et 3) détection des activités illicites.

6.2 Implantation de l'outil de détection

PIGA est une implantation de l'algorithme de détection défini dans la section 5.4.4 (cf. algo 10, page 146). Cet outil prend en entrée une politique neutre (en XML) et réalise les trois phases de notre approche :

- 1) Construire le graphe d'interactions et les graphes de dépendance causale ;
- 2) Générer la base de signatures ;
- 3) Détecter les activités illicites.

Dans cette section, nous détaillons le fonctionnement de l'implantation de notre outil.

6.2.1 Phase 1 : Construction des graphes

Afin de d'effectuer la phase de *construction des graphes* (phase 1, cf. section 5.2, page 131), PIGA prend en entrée une politique neutre. Cette politique permet de construire le graphe d'interactions. Ce graphe est ensuite utilisé pour construire les trois graphes de dépendance causale (graphe de dépendance causale, graphe de flux d'informations et graphe de transitions).

6.2.1.1 Construction du graphe d'interactions

Tout d'abord, la politique neutre permet de construire le *graphe d'interactions* via l'algorithme de construction du graphe d'interactions (cf. algo 2, page 95). La table 6.3 donne, pour les 6 exemples de politiques SELinux définies précédemment, le temps nécessaire pour convertir la politique neutre en un graphe d'interactions et la mémoire nécessaire pour stocker le graphe résultant. Notre outil nécessite ainsi environ 3 secondes pour une politique standard ou minimale et environ 30 secondes pour une politique complexe. L'espace mémoire nécessaire au stockage du graphe est de 10Mo pour construire le graphe d'une politique standard ou minimale et de 200Mo dans le pire des cas. Notons que notre outil propose plusieurs manières de stocker un graphe. Celle qui est utilisée dans cette

Nom	SEL 1	SEL 2	SEL 3	SEL 4	SEL 5	SEL 6
SC	577	3017	624	595	472	878
IV	17 684	314 582	21 359	18 215	13 843	58 483
Taille du graphe	10Mo	201Mo	13Mo	11Mo	11Mo	36Mo
Temps de calcul	3s	33s	3,8s	3,4s	1,8s	3,6s

TAB. 6.3 – Calcul d'un graphe d'interaction pour SELinux

Nom	GR 1	GR 2	GR 3	GR 4
SC	208	39	60	70
IV	1399	89	206	318
Taille du graphe	2Mo	<1Mo	<1Mo	<1Mo
Temps de calcul	<1s	<1s	<1s	<1s

TAB. 6.4 – Calcul d'un graphe d'interaction pour GRSECURITY

expérimentation est la plus coûteuse en mémoire (du fait de la présence d'un grand nombre de tables de *hashage* afin d'accéder plus rapidement aux arcs et aux sommets) mais la plus rapide pour l'analyse du graphe (pour énumérer les chemins). Ainsi, dans une autre représentation, le graphe d'une politique complexe nécessite moins de 40Mo de mémoire, mais son analyse requiert quatre fois plus de temps.

La table 6.4 indique le temps et l'espace nécessaires à la conversion d'une politique neutre issue de GRSECURITY en graphe d'interactions. Du fait de la taille de ces politiques, ces graphes requièrent peu de mémoire (1Mo) et leur conversion est extrêmement rapide (moins de 1 seconde). De ce fait, dans la suite de nos expérimentations, nous ne présenterons plus que les résultats liés aux politiques SELinux qui sont plus complexes et donc plus intéressantes pour estimer les performances et les capacités de notre outil de détection.

6.2.1.2 Construction des graphes de dépendance causale

Le graphe d'interactions est converti en *graphe de dépendance causale* via l'algorithme de construction de graphe de dépendance causale (cf. algo 2, page 95). Cet algorithme nécessite la définition d'une table de changement d'état qui permet de définir l'effet des opérations, c'est-à-dire d'orienter les arcs dans le sens de l'effet (cf. section 4.2, page 94). Le listing 6.7 contient un exemple de table utilisée lors de la construction du graphe de flux d'informations. Cette table a été à l'origine écrite pour l'outil SLAT et nous l'avons légèrement modifiée. Chaque ligne de cette table débute avec la classe de l'opération élémentaire concernée (par exemple *file* pour les fichiers) et le nombre d'éléments de cette classe. Puis pour chaque élément, nous indiquons le nom de l'opération et le sens de l'effet : $\leftarrowtail$ pour *lecture*, $\rightarrowtail$ pour *écriture*, $\leftrightarrow$ pour *lecture* et *écriture*, et $-$ pour aucun). De plus, nous donnons une valeur (entre 1 et 40), que nous appellerons *Mapping*, qui permet de pondérer la probabilité de l'effet. Par exemple, l'opération $entrypoint \leftarrowtail 1$ (qui correspond à l'ouverture d'un descripteur de fichier) change l'état du sujet, mais ce changement d'état est mineur car il n'y a pas réellement de transfert d'informations. Inversement, l'opération $read \leftarrowtail 40$ change l'état du sujet et transfert clairement de l'information.

La table 6.5 présente les caractéristiques des graphes de dépendance causale obtenus à partir des exemples précédents de politiques SELinux. Dans cette table, nous n'indiquons ni le temps nécessaire à la construction de ces graphes, ni l'espace nécessaire. En effet, cette construction "réutilise" le graphe d'interactions construit précédemment. De ce fait, le temps de construction ne dépasse pas une

```
file execute_no_trans ↤ 1 entrypoint ↤ 1 execmod − 1 ioctl − 1 read ↤ 40 write ↣ 40
    create ↣ 1 getattr ↤ 7 setattr ↣ 7 lock − 1 relabelfrom ↔ 1 relabelto ↣ 5 append ↣
     1 unlink ↣ 1 link ↣ 1 rename ↣ 5 execute ↤ 1 swapon ↔ 1 quotaon ↔ 1 mounton ↔ 1
process fork − 1 transition ↣ 5 sigchld ↣ 1 sigkill ↣ 1 sigstop ↣ 1 signull − 1 signal
    ↣ 5 ptrace ↔ 30 getsched ↤ 1 setsched ↣ 1 getsession ↤ 1 getpgid ↤ 1 setpgid ↣ 5
    getcap ↤ 3 setcap ↣ 1 share ↔ 1 getattr ↤ 1 setexec ↣ 1 setfscreate ↣ 1 noatsecure
     − 1 siginh − 1 setrlimit − 1 rlimitinh − 1 dyntransition ↣ 30 setcurrent ↣ 1
    execmem − 1 execstack − 1 execheap − 1
[...]
```

Listing 6.7 – Extrait de table de changement d'état pour le graphe de flux d'informations.

Nom		SEL 1	SEL 2	SEL 3	SEL 4	SEL 5	SEL 6
Graphe de	SC	577	3017	624	595	472	878
dépendance causale	IV	17 684	314 582	21 359	18 215	13 843	58 483
Graphe de	SC	139	1077	165	143	121	264
transitions	IV	256	2269	300	263	231	527
Graphe de	SC	411	1893	428	423	333	469
flux d'informations	IV	451	8795	448	451	403	924
(Mapping > 36)							
Graphe de	SC	446	1995	462	460	360	622
flux d'informations	IV	1563	25816	1560	1593	1297	3130
(Mapping > 1)							

TAB. 6.5 – Graphe de dépendance causale pour SELinux

seconde et la mémoire impliquée par cette construction n'excède pas 1 Mo.

Dans cette table, nous indiquons le nombre de contextes et de vecteurs d'interactions de chaque graphe. Ainsi le graphe de transitions n'excède pas les 2500 arcs pour 1000 nœuds. De plus, nous donnons les caractéristiques de deux types de graphe de flux d'informations : un graphe qui considère toutes les opérations élémentaires (*Mapping* > 1), et un graphe qui ne considère que les opérations *fortes* de transfert d'informations (*Mapping* > 36). Ainsi, comme le montre cette table, ce paramètre permet de diviser par quatre la taille du graphe (le nombre d'arc). Le facteur de *Mapping* influe sur la taille du graphe et donc sur le nombre de signatures générées lors de la phase d'énumération des chemins. Mais ce facteur a surtout un intérêt, lors de la phase de détection, car il évite l'énumération des activités qui surestiment la possibilité de transfert d'informations. Ainsi, une séquence provenant d'un graphe de facteur fort, par exemple une suite de lecture et d'écriture, aura plus de chance de correspondre à un flux d'informations qu'une séquence provenant d'un graphe de facteur faible, par exemple une suite de signaux inter processus.

6.2.2 Phase 2 : Génération de la base de signatures

Nous avons implanté, dans notre outil, les algorithmes d'énumération des activités, c'est-à-dire des interactions et des séquences (cf. algo. 3 et 4, page. 106). Nous rappelons que ces algorithmes prennent en entrée un contexte source, un contexte cible et un graphe. Ils énumèrent alors les activités, c'est-à-dire les interactions et les séquences, qui relient ces deux contextes dans le graphe. Ces énumérations sont utilisées pour la phase de construction de la base de signatures qui prend en entrée une politique de détection et calcule les signatures associées à chaque propriété de sécurité requise par la politique de détection (cf. section 4.4, page 110).

Profondeur	Durée de l'énumération	Nb. de chemins	Mémoire utilisée	Taille de la base
6	2s	20 592	4,4Mo	2Mo
7	12s	160 877	34Mo	16,3Mo
8	34s	423 020	88Mo	42Mo
9	1min26s	982 173	170Mo	99Mo
10	3min40s	2 234 558	350Mo	205Mo

TAB. 6.6 – Performance de la recherche des chemins

Dans cette partie, nous ne présentons que les performances de l'algorithme d'énumération des chemins, c'est-à-dire des activités illicites reliant deux nœuds. La table 6.6 présente les performances de notre algorithme d'énumération des chemins, en utilisant l'algorithme des *K-plus court chemins*. Cette table représente la moyenne des résultats obtenus sur 50 graphes générés aléatoirement et contenant 2000 nœuds et 100,000 arcs orientés. Nous fixons la valeur de K à 5,000,000 et nous faisons varier la profondeur afin d'énumérer tous les chemins possibles. La profondeur de recherche varie alors de 6 à 10. Dans cette table, nous donnons la durée nécessaire à l'énumération de tous les chemins entre deux nœuds quelconques du graphe, le nombre de chemins obtenus, la quantité maximum de mémoire utilisée lors de cette énumération et la taille de la base obtenue. Cette base contient alors l'ensemble des chemins énumérés.

Ainsi, cet algorithme nécessite 2 secondes pour énumérer 20,000 chemins ce qui correspond à une base de signatures d'une taille de 2Mo et 1 minute 30 secondes pour énumérer 1,000,000 de chemins pour une taille totale de la base de 99Mo. Comme nous le verrons dans la section suivante, lors de l'utilisation de cet algorithme sur des politiques réelles, l'énumération d'une propriété de sécurité n'excède pas 20,000 chemins pour une politique standard et 800,000 dans le pire des cas. Ainsi, notre algorithme est tout à fait utilisable en condition réelle et nécessite quelques minutes pour énumérer l'ensemble des activités violant un ensemble de propriétés de sécurité. Nous donnons, dans la section 6.3, des exemples réels de calcul de la base de signatures.

6.2.3 Phase 3 : Détection des activités illicites

Dans cette section, nous proposons de décrire le fonctionnement de la phase de détection. Cette phase utilise la base de signatures obtenue pour une politique de détection et une politique neutre données. Cette phase requiert aussi des traces d'interactions. C'est pourquoi nous décrivons d'abord l'audit des interactions et ensuite l'utilisation pour la détection des activités illicites.

6.2.3.1 Audit des interactions

Pour obtenir les traces des interactions, nous avons besoin d'un mécanisme d'audit. Pour cela, nous générons un ensemble de règles d'audit pour les interactions qui composent la base de signatures. Ces règles d'audit permettent de configurer le système cible qui génère alors une trace lorsqu'une interaction auditée est réalisée. L'avantage de cette approche est de n'auditer que les interactions susceptibles de donner lieu à une alerte, les autres interactions ne sont pas auditées. De plus, l'audit est indépendant de la reconstruction des activités illicites, ce qui présente l'avantage de pouvoir réutiliser tout mécanisme d'audit disponible.

Le listing 6.8 contient un exemple de règles d'audit pour SELinux. Une telle règle commence

par `auditallow` suivie du vecteur d'interactions qui doit être audité. Par exemple, la ligne 1 permet d'auditer toute transition du contexte `login_d` vers le contexte `user_d`, la ligne 2 permet d'auditer les opérations de lecture et d'écriture effectuées par un contexte `user_d` sur un objet ayant le contexte `user_info_t`. Ce fichier contient donc tous les audits d'interactions pour la base de signatures, par simple traduction des interactions de cette base dans le format permettant l'audit.

```
auditallow login_d user_d : process { transition } ;
auditallow user_d user_info_t : file { read write  } ;
```

Listing 6.8 – Exemple de règles d'audit pour SELinux.

Le listing 6.9 contient un exemple de règle d'audit pour GRSECURITY. Dans ce cas, une règle d'audit est définie par une opération en majuscule. Par exemple, la ligne 1 correspond à l'audit des opérations de lecture (R), d'écriture (W), d'exécution (E), de création (C) et de destruction (D) sur tout objet contenu dans le répertoire des utilisateurs. Nous voyons qu'ici le mécanisme pour configurer l'audit de GRSECURITY est assez différent de celui de SELinux.

```
/home     RWXCD
/mnt      RW
```

Listing 6.9 – Exemple de règles d'audit pour GRSECURITY.

Le listing 6.10 présente un exemple de traces générées par SELinux pour deux interactions auditées. La première interaction correspond à une transition du contexte `system_u :system_r :httpd_d` vers le contexte `system_u :system_r :webserv_d`, la seconde correspond à une lecture de l'objet `user_u :object_r :user_info_t` par ce second contexte. Le listing 6.11 présente un exemple de traces

```
audit(1134743631.096:0): avc:  allowed  { transition } for  pid=15008 ppid=14002 exe=/usr/
    sbin/apache2 name=root dev=hda2 ino=635233 scontext=system_u:system_r:httpd_d tcontext=
    system_u:system_r:webserv_d tclass=process
audit(1134743635.187:0): avc:  allowed  { read } for  pid=15008 ppid=14002 exe=/usr/bin/
    service_urx scontext=system_u:system_r:webserv_d tcontext=user_u:object_r:user_info_t
    tclass=file
```

Listing 6.10 – Exemple de traces d'audit pour SELinux.

générées par GRSECURITY pour ces deux interactions. Ainsi, nous pouvons remarquer que les traces de SELinux contiennent directement les contextes de sécurité mis en jeu (mots-clé `scontext` et `tcontext`), alors que les traces de GRSECURITY ne contiennent que les chemins correspondant à ces contextes. De ce fait, nous introduisons un composant de conversion des traces du système cible

```
Dec 12 17:36:29 test kernel: grsec: exec of /usr/bin/service_urx by /usr/sbin/apache2[
    apache2:28013] uid/euid:0/0 gid/egid:0/0, parent /bin/bash[bash:21663] uid/euid:0/0 gid/
    egid:0/0
Dec 12 17:36:29 test kernel: grsec: read of /etc/user_info_t by /usr/bin/service_urx[
    service_urx:28035] uid/euid:0/0 gid/egid:0/0, parent /usr/sbin/apache2[apache2:28013]
    uid/euid:0/0 gid/egid:0/0
```

Listing 6.11 – Exemple de traces d'audit pour GRSECURITY.

en *traces neutres d'interactions*. Dans le cas de SELinux, cette conversion est assez directe. Mais dans le cas de GRSECURITY, la conversion nécessite la traduction des chemins en contextes (via la table définie dans la partie 6.1.3). Finalement, cette phase ajoute dans la trace d'interactions le *pid*, le *ppid* et la date d'apparition nécessaires au *reconstructeur arborescent*. Notons que normalement, les traces

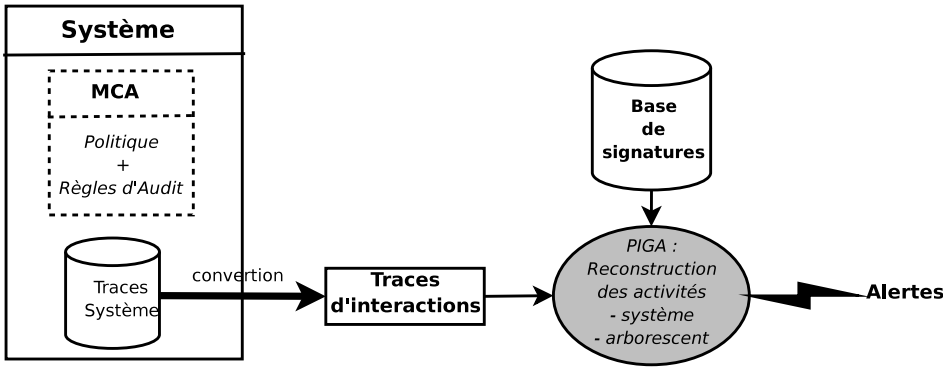


FIG. 6.3 – Fonctionnement de la phase de détection des activités.

```
confidentiality = (( system_u:system_r:httpd_d --trans--> system_u:system_r:webserv_d ) ∧ (
    system_u:object_r:user_info_t > system_u:system_r:webserv_d ))
```

Listing 6.12 – Exemple d'alerte générée.

de SELinux ne fournissent pas d'information sur le *ppid*, nous avons donc créé un correctif noyau afin d'ajouter ces données aux traces système. Les traces converties ont alors la forme suivante :

$$t = ((sc_s, sc_o, eo), pid, ppid, date)$$

6.2.3.2 Reconstruction des activités

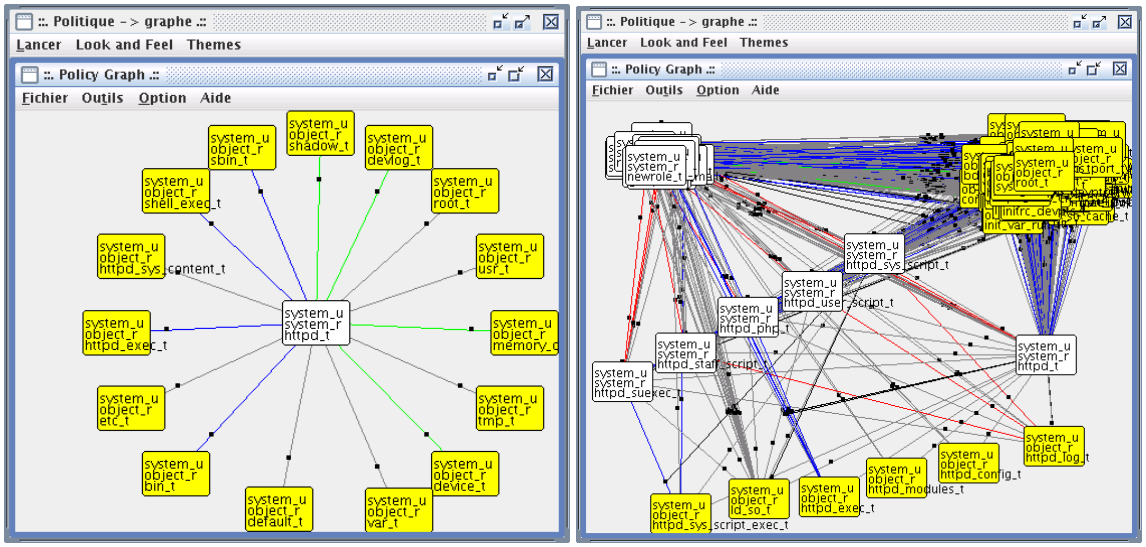
La figure 6.3 illustre le fonctionnement de la phase de détection des activités illicites. Pour cela, notre outil implante les différents reconstructeurs, définis dans la section 5.4 (page 136), et l'algorithme de reconstruction (cf. algo 9, page 9). Lors de cette phase, notre outil prend en paramètres la base de signatures, obtenue lors de la phase 2, et un ensemble de traces d'interactions.

Comme indiqué dans la figure 6.3, ces traces sont analysées par les différents *reconstructeurs* d'activités (cf. 5.4.3, page 144). Pour chaque activité reconstruite, c'est-à-dire correspondant à une signature de la base, le reconstructeur génère une alerte. Ces alertes, exprimées dans notre langage de description d'activité, contiennent le type de propriété violée et l'activité ayant engendré la violation. Le listing 6.12 contient un exemple d'alerte générée par notre système de détection. Cette alerte correspond ainsi à une violation par le serveur Apache de la propriété de confidentialité du fichier d'information utilisateur via un *accès à l'information*.

Finalement, notre IDS peut, soit analyser un ensemble de traces résultants de l'exécution d'un système (analyse indirecte), soit directement analyser les traces lors de l'exécution du système (analyse directe). Dans le cas de l'analyse directe, notre outil peut être utilisé de deux manières différentes :

- **Locale** : détection sur le système audité ;
- **Distante** : détection décentralisée sur un autre système.

Dans le cas de l'utilisation locale, notre IDS est directement installé sur le système analysé et utilise directement les traces générées par le mécanisme de contrôle (fichier `/var/log/avc.log`). Dans le cas de l'utilisation distante, notre IDS est installé sur une machine dédiée à l'analyse des traces. Le mécanisme de traçage du système analysé est alors configuré pour envoyer les traces système à notre IDS (via le protocole de `syslog` distant). Notre IDS fonctionne alors comme un serveur `syslog`.



(a) Grsecurity (b) SELinux

FIG. 6.4 – Visualisation des graphes d'interactions

Notons finalement que d'après nos expérimentations, la phase de détection nécessite en moyenne 40Mo de mémoire. Cet espace est essentiellement alloué afin de stocker la base de signatures et l'arbre des processus du reconstructeur arborescent. De plus, l'analyse indirecte d'un an d'activités nécessite en moyenne 4 heures d'analyse par machine analysée. De part l'utilisation mémoire de notre IDS, et de part sa vitesse d'analyse, notre IDS peut être utilisé en conditions réelles, et ceci localement ou de manière distante. Les expérimentations directes, présentées dans la section suivante, montrent clairement l'efficacité de cette solution et sa capacité à détecter des attaques complexes.

6.2.4 Chargement et visualisation du graphe d'interactions

Finalement, nous avons aussi inclus, dans notre outil, une interface de visualisation et d'édition des graphes. Cette interface permet de manipuler les différents graphes d'interactions et de dépendances causales. Dans notre interface, nous avons inclus des modules permettant de :

- Charger une politique neutre et créer les graphes correspondants ;
- Visualiser et manipuler ces graphes ;
- Éditer un graphe : ajout/suppression de contextes ou de vecteurs d'interactions ;
- Charger et appliquer une politique de détection afin de générer la base de signatures ;
- Visualiser les alertes générées lors de la phase de détection.

La figure 6.4(a) contient ainsi un exemple de visualisation de politique GRSECURITY. Cette visualisation est restreinte au domaine Web (`http`). La figure 6.4(b) représente le même domaine pour une politique SELinux. Ces deux images font ainsi clairement apparaître la différence de complexité entre SELinux et GRSECURITY.

6.3 Expérimentation

Afin de tester notre IDS, nous avons utilisé une plateforme expérimentale de type pots-de-miel (en anglais *honeypots*). Un pot-de-miel peut être défini comme : *Un système informatique public*

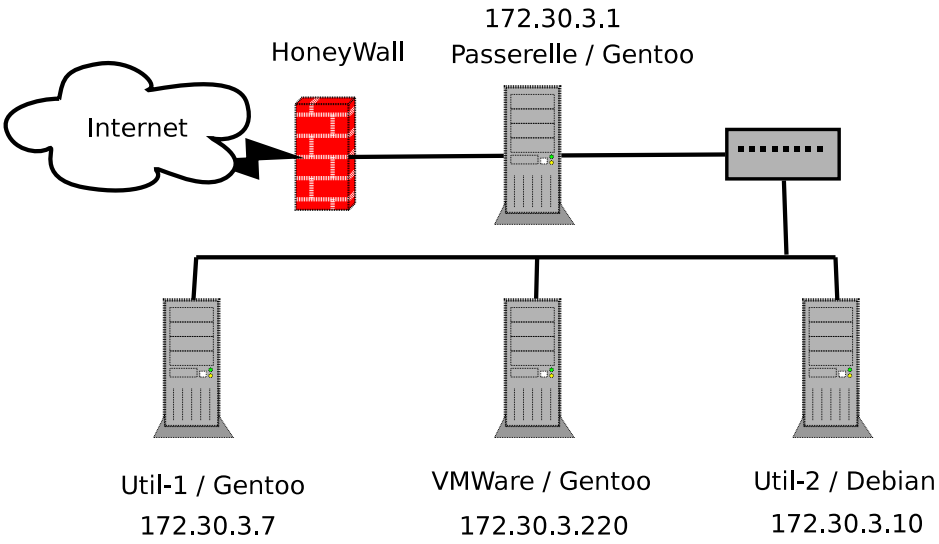


FIG. 6.5 – Schéma de notre plateforme d'expérimentation.

volontairement vulnérable à une ou plusieurs failles connues visant à attirer les pirates afin d'étudier leurs stratégies d'attaque pour mieux les comprendre et les anticiper. En simulant un ordinateur ou un réseau entier vulnérable à une ou plusieurs failles, les honeypots attirent les pirates qui tenteront alors d'exploiter ces dernières dans le but d'arrêter ou corrompre le dit système..

Dans cette section, nous présentons cette plateforme d'expérimentation dans la partie 6.3.1. Dans la partie 6.3.2, nous présenterons les politiques de protection et de détection utilisées sur chaque hôte de cette plateforme ainsi que les bases de signatures obtenues. Nous commenterons ensuite les attaques détectées durant un an dans la partie 6.3.3. Finalement, nous discuterons des résultats de cette détection dans la partie 6.3.4.

6.3.1 Plateforme d'expérimentation

La figure 6.5 représente la plateforme d'expérimentation définie pour tester notre outil de détection d'intrusion. Cette plateforme est constituée de quatre systèmes :

- Une passerelle connectée à Internet (4 IP publics) et à un sous réseau (172.30.3.0/24) ;
- Une machine contenant des applications utilisateurs sous Gentoo (*Util-1*) ;
- Une deuxième machine utilisateur sous Debian (*Util-2*) ;
- Un serveur sur lequel est installé l'outil de virtualisation `VMWare`. Cette machine héberge ainsi les deux machines utilisateurs (*Util-1* et *Util-2*).

Ces systèmes ont accès à Internet, directement ou via la passerelle, et chaque machine possède un service `sshd` modifié. Ce service a été modifié pour accepter aléatoirement des demandes de connexions `ssh` (en moyenne 1% des demandes). De ce fait, ce service autorise aléatoirement la connexion d'un attaquant qui tente d'ouvrir une session `ssh` sur un hôte. Les attaquants peuvent ensuite se connecter par rebond sur les trois autres systèmes. Notre IDS est installé sur le serveur `VMWare/Gentoo` et il analyse alors les traces des quatre systèmes de cette plateforme. Ces quatre systèmes sont donc configurés pour envoyer les traces du mécanisme de contrôle d'accès vers notre IDS via le mode *syslog* distant.

```
integrity(SC_exec)                                                    1
int_domain(SC_.*:.*:user.*)                                           2
confidentiality(SC_S, sc_.*:.*:user.*, G_trans)                       3
duties_sep(SC_S, G_trans)                                             4
bad_transition(.* : .* : user.*, SC_S)                                5
conformity()                                                          6
```

Listing 6.13 – Exemple de politique de détection.

Cette plateforme a été utilisée durant un an, de juillet 2006 à juin 2007, et a reçu de nombreuses attaques suite à la découverte de ces systèmes ouverts lors de *scan ssh*. Ce type de *scan* est utilisé par les attaquants afin de découvrir les systèmes ayant des comptes `ssh` ouverts. Notre *pot-de-miel*, avec son serveur `sshd` modifié, offre alors une cible de choix pour les attaquants et a subi de nombreuses connections illicites.

Dans cette section, nous détaillons tout d'abord les politiques de contrôle d'accès et la politique de détection utilisées sur ces systèmes. Nous détaillerons ensuite les caractéristiques des bases de signatures obtenues pour notre IDS. Finalement, nous présenterons les statistiques d'attaques sur la durée de l'expérimentation, puis nous étudierons un cas particulier d'attaque.

6.3.2 Politiques utilisées

Nous commençons par détailler la politique de détection et les politiques de contrôle d'accès installées sur nos systèmes. Notre définition de la politique de détection permet de générer une base de signatures adaptée au cas des *pots-de-miel*.

6.3.2.1 Politique de détection

Notre plateforme d'expérimentations étant considérée comme un pot-de-miel, nous avons défini une politique de détection dont l'objectif est de garantir :

1. L'intégrité du système ;
2. L'isolation du domaine utilisateur ;
3. La confidentialité du système ;
4. La séparation de privilèges ;
5. L'absence de transitions pour l'utilisateur ;
6. Le respect de la politique de protection.

Ces règles permettent ainsi de détecter l'ensemble des activités réalisées par les utilisateurs, c'est-à-dire les attaquants s'étant connectés sur un système. En effet, dans le cadre d'une utilisation normale, nul n'est sensé se connecter sur un de ces systèmes (définition même d'un pot-de-miel). Les activités des utilisateurs sont donc considérées comme illicites.

Le listing 6.13 contient la politique de détection définie afin d'appliquer ces propriétés de sécurité. Ces règles ont été définies dans la section 4.4 (page. 110), la table A.1 (page. 188) en contient une synthèse. La première règle concerne l'*intégrité du système*. Ainsi, toute modification directe, ou par séquence, d'un objet exécutable du système donnera lieu à une alerte. La seconde règle correspond à la propriété d'*intégrité des domaines* qui permet d'isoler tous les contextes dont le type commence par `user`. Ainsi, nous détectons toutes les actions réalisées par les utilisateurs. La troisième règle permet de garantir la *confidentialité* du système. Ainsi, nous détectons tous les accès à l'information

Nom		Passerelle	Util-1	VMware	Util-2
Description	Distribution	Gentoo	Debian	Gentoo/vmware	Gentoo
	Version	2004		2004	2005
	Type	Serveur	Serveur Utilisateur	Serveur Etch	Utilisateur
Politique Cible	Utilisateur	3	3	5	3
	Rôle	5	6	5	5
	Type	637	2086	637	656
	Règle	17 639	235 691	17 650	19 497
Politique Neutre	SC_O	438	1940	459	452
	SC_S	139	1077	165	143
	SC	577	3017	624	595
	IV	17 684	314 582	21 359	18 215
Graphes de dépendance causale	Graphe de transitions	139	1077	165	143
		256	2269	300	263
	Graphe de flux d'informations (Mapping > 36)	411	1893	428	423
		451	8795	448	451

TAB. 6.7 – Systèmes installés sur notre réseau.

du système réalisés par les utilisateurs. La quatrième règle correspond à la propriété de *séparation de privilèges* simple. Ainsi, nous détectons les modifications d'objets suivies de leurs exécutions. Avec la règle `bad_transition`, nous détectons toutes les transitions ou séquences de transitions réalisées par l'utilisateur. Finalement, la règle `conformity` garantit le respect de la politique cible. Ainsi, nous détectons les activités qui violent la politique de protection.

6.3.2.2 Synthèse des politiques de contrôle d'accès

Les systèmes installés sur cette plateforme correspondent aux systèmes SELinux étudiés dans la section précédente (cf. section 6.1.2.2). La table 6.7 contient une synthèse des caractéristiques de leur politique de contrôle d'accès. Ainsi, nous avons pris en considération trois systèmes standards (*Passerelle*, *VMWare*, *Util-2*) et un système complet *Util-1*.

6.3.2.3 Base de signatures

Les politiques de contrôle d'accès (de la table 6.7) et la politique de détection (du listing 6.13) ont ainsi servi de source à la phase de génération de la base de signatures. La table 6.8 synthétise les caractéristiques des différentes bases obtenues. La règle générant le plus de signatures est ainsi la règle concernant la *confidentialité* du système, la règle qui en génère le moins est la règle d'*intégrité*. En effet, dans la définition de notre politique de détection, cette seconde règle est restreinte à l'intégrité des objets exécutables.

Nous obtenons ainsi quatre bases de signatures. Pour chaque base, nous donnons la taille de la base stockée sous forme d'un fichier compressé et l'espace réel utilisé en mémoire. Ainsi, nous pouvons remarquer que la base occupe environ 5Mo pour un système standard contre 30Mo pour un système complet.

Nous précisons ensuite le nombre de règles d'audit qui doivent être ajoutées dans la politique de contrôle d'accès. Sur la première ligne nous indiquons le nombre d'audits nécessaires et sur la

Nom		Passerelle	Util-1	VMware	Util-2
Graphe	SC	577	3017	624	595
	IV	17 684	314 582	21 359	18 215
Règles de détection	`integrity`	137	9 461	186	140
	`int_domain`	16 283	510 215	18 130	16 546
	`confidentiality`	29510	726 842	29510	29510
	`duties_sep`	243	16 405	320	270
	`bad_transition`	3555	126 228	4250	3941
Résultat	Taille de la base	1,1Mo	3,6Mo	1,2Mo	1,1Mo
		4,7Mo	28Mo	4,9Mo	4,8Mo
	Nb d'audit	13 664	44 503	14 417	14051
		3 624	51 616	3 491	3 764
	Durée de l'analyse	47s	10min31s	1min2s	52s

TAB. 6.8 – Base de signatures obtenues.

seconde, le nombre de vecteurs d'audits correspondant. Ainsi, avec notre approche, moins d'un quart des opérations système sont auditées contrairement aux autres approches de détection qui nécessitent l'audit complet du système (cf. section 3.4, page 85). C'est pourquoi notre approche est appliquée sur l'ensemble du système. De plus, comme nous le verrons dans la section suivante, notre approche permet de détecter des activités complexes telles que des séquences d'interactions.

Finalement, la dernière ligne indique le temps nécessaire pour la création de la base de signatures. Ainsi, sur un système standard, cette construction nécessite moins de 1 minute. Dans le pire cas étudié, cette construction nécessite 10 minutes. Étant donné que cette phase n'est réalisée que lorsque la politique de contrôle d'accès est modifiée, ces performances sont tout à fait adaptées à une utilisation réelle de notre outil. Dans le cas de cette plateforme, la base de signatures n'a été régénérée que lors de mises-à-jour nécessaires à la phase d'installation (au mois de juillet). Une fois les politiques correctement définies, cette phase n'a plus été ré-effectuée.

6.3.3 Analyse des résultats

Dans cette section, nous présentons les résultats obtenus lors de la phase de détection sur cette plateforme d'expérimentation. Nous proposerons ainsi une analyse des attaques subies. Dans certains graphiques, nous ignorons volontairement les alertes qui correspondent à la règle `int_domain`. Cette règle génère une alerte par interaction réalisée par un utilisateur, SELinux étant un système de contrôle d'accès fin, les autres alertes sont noyées dans le flux généré par cette règle. En effet, la présence de cette règle est plus liée à des perspectives de corrélation.

6.3.3.1 Analyse sur un an

Tout d'abord, la figure 6.6 représente la répartition des alertes obtenues sur un an d'expérimentation. La majorité des alertes correspondent à des violations de l'intégrité des contextes exécutables. Cette proportion peut s'expliquer par le fait que, dans la majorité des attaques, lorsque l'attaquant accède à notre pot-de-miel, il l'utilise ensuite pour installer un outil de *scan* ou un robot *irc*. De ce fait, il télécharge une archive dans le répertoire temporaire ou dans son répertoire utilisateur, puis l'extrait. Or, lors de cette extraction, il crée (donc modifie), pour chaque fichier extrait, un nouvel objet qui est du type exécutable. Ainsi, pour chaque fichier extrait, une alerte de type `integrity` est générée puisque cela correspond à une modification d'un fichier exécutable.

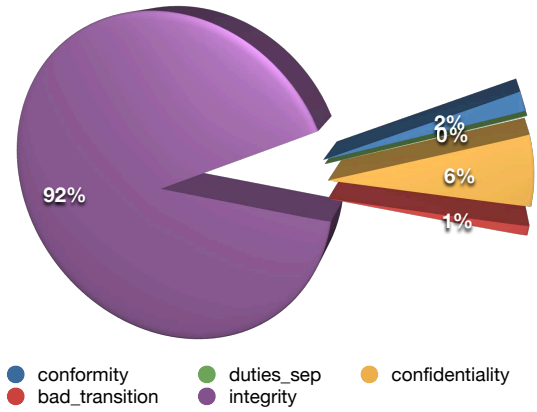


FIG. 6.6 – Répartition des alertes sur un an.

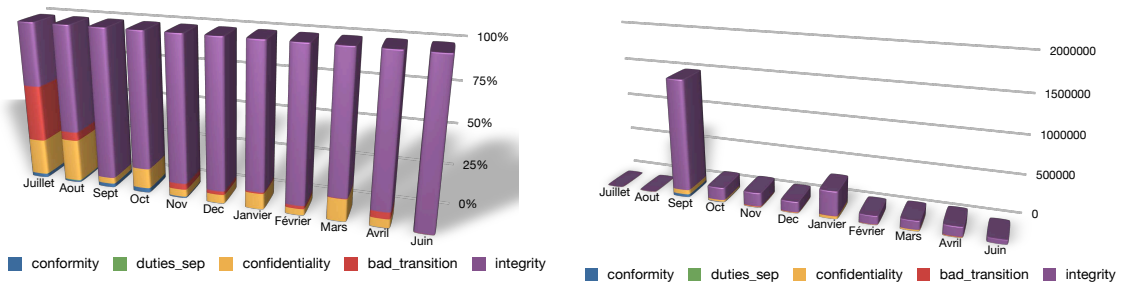
Les violations de confidentialité représentent 6% des alertes générées. Après une étude détaillée des traces système, nous avons constaté que ces alertes résultent, dans la plus grande majorité des cas, à des *flux d'informations* ou à des *accès à l'information* de fichiers temporaires. En effet, les outils installés par les attaquants génèrent et stockent des données dans des répertoires temporaires. Ces données sont, par la suite, récupérées par l'attaquant, générant ainsi des *flux d'informations* interdits entre les fichiers temporaires et l'utilisateur.

Dans notre expérimentation, les tentatives de violation de politiques de contrôle d'accès représentent 2% des attaques. Ces alertes sont généralement liées à des tentatives d'obtention du privilège administrateur ou d'accès à des fichiers système proscrits par la politique (par exemple, `/etc/shadow`).

Les alertes liées aux changements de contextes (*séquences de transitions*) correspondent à 1% des alertes observées. Elles sont majoritairement liées à des connections `ssh` ou `ftp`. Dans notre étude, ces alertes sont caractéristiques d'une connexion réussie, ce qui représente un total de 45 590 connections sur notre pot-de-miel en un an. Ce résultat démontre la nécessité d'avoir des mots-de-passe forts pour un système ouvert sur Internet.

Finalement, les violations qui correspondent à des modifications puis à des exécutions d'un objet représentent moins de 1% des alertes, soit au total 2219 alertes de ce type. Pourtant ces alertes sont significatives de l'activité d'un attaquant. En effet, elles correspondent à l'installation d'outils suivie de leurs exécutions. Ainsi sur 45 590 connections sur notre système, les attaquants n'ont installé puis exécuté que 2219 outils. Ceci est du au fait que la majorité des attaquants ne donnent pas de suite à leur connexion, d'une part parce qu'ils se doutent, après analyse du système, qu'il s'agit d'un pot-de-miel, d'autre part parce que la présence de SELinux les empêche d'avoir trop de privilèges.

Notons que l'utilisation de la règle `conformity` nous a permis d'identifier des problèmes liés à la définition des règles de protection SELinux. En effet, ce langage définit des règles entre *type* et non entre *contextes de sécurité*. Lors de l'utilisation de SELinux, nous nous sommes rendus compte que certains contextes objets changeaient d'attribut *identité* lorsqu'ils étaient édités par des utilisateurs autre que le propriétaire. Ce qui conduit ensuite à l'apparition de contextes (identité,role,type) qui ne sont pas déclarés dans la politique, mais tout de même autorisés par le mécanisme de contrôle d'accès (car le contrôle ne se fait que sur les *types*). La règle `conformity`, de notre IDS, permet de détecter ce genre d'abus car notre contrôle s'effectue sur les contextes et pas seulement sur un attribut. Notre langage neutre, étant plus général que celui utilisé par SELinux, permet donc de détecter des



(a) Répartition des alertes par mois.

(b) Nombre d'alertes par mois.

incohérences du système cible qui peuvent aboutir à des abus de ce système.

6.3.3.2 Répartition des alertes

Les figures 6.7(a) et 6.7(b) représentent respectivement la répartition des alertes et le nombre d'alertes sur cette année d'observation. La figure 6.7(a) montre que le nombre de connexions sur notre plateforme est plus important pour certains mois, par exemple les mois de juillet, août, décembre et avril ; et presque inexistant au mois de septembre octobre et juin. De plus, les mois de juillet et d'août ont été propices à l'installation d'outils de *scan*. Ce qui se traduit par un grand nombre de violations de confidentialité.

Dans la figure 6.7(b), nous observons deux pics d'alertes. Tout d'abord le pic du mois de septembre correspond à une attaque importante qui a eu pour conséquence de remplir tout l'espace disque de la partition stockant les comptes utilisateurs. Cette attaque correspond ainsi à un déni de service local. Le second pic (janvier) correspond à plusieurs installations d'outils de `scan ssh`, dont l'objectif était de scanner d'autres réseaux et ainsi de découvrir d'autres systèmes ouverts. Ces outils ont eu pour conséquence de générer un grand nombre de fichiers de *log*.

6.3.3.3 Exemple d'attaque

Dans cette partie, nous proposons d'étudier les alertes générées le 10 novembre 2006. La figure 6.7 représente les alertes du type changement de contexte (*séquence de transitions*), générées par la règle `bad_transition`, en fonction du temps pour cette journée. Lors de cette journée, un attaquant a tout d'abord scanné le serveur `sshd` de notre pot-de-miel à 6h. Ce *scan* correspond à une attaque par force brute sur les comptes utilisateurs du serveur dans le but d'ouvrir une connexion `ssh`. Lors de ce *scan*, la version modifiée de notre serveur `sshd` a alors créé 11 comptes utilisateurs, impliquant 11 connections réussies de l'attaquant. Notre IDS génère alors, pour chaque connexion réussie, une alerte qui correspond à une activité de type *séquence de transitions* vers le contexte associé au compte utilisateur (`user_t`) :

`bad_transition` = `system_u:system_r:init_t` $\xrightarrow{trans}$ `system_u:system_r:initrc_t` $\xrightarrow{trans}$ `system_u:system_r:sshd_t` $\xrightarrow{trans}$ `user_u:user_r:user_t`

Chaque alerte de ce type correspond donc à une connexion `ssh` d'un utilisateur sur notre système qui viole la *propriété de changements de contexte* définie dans la politique de détection. Dans la figure 6.7, nous voyons que l'attaquant s'est ensuite connecté à 17h, en utilisant un compte obtenu lors du *scan* de 6h. Nous pouvons d'ailleurs remarquer qu'il n'y a eu aucune autre activité illicite entre 6h et 17h. Lors de cette deuxième connexion, l'attaquant a tout d'abord vérifié qu'il pouvait envoyer des paquets

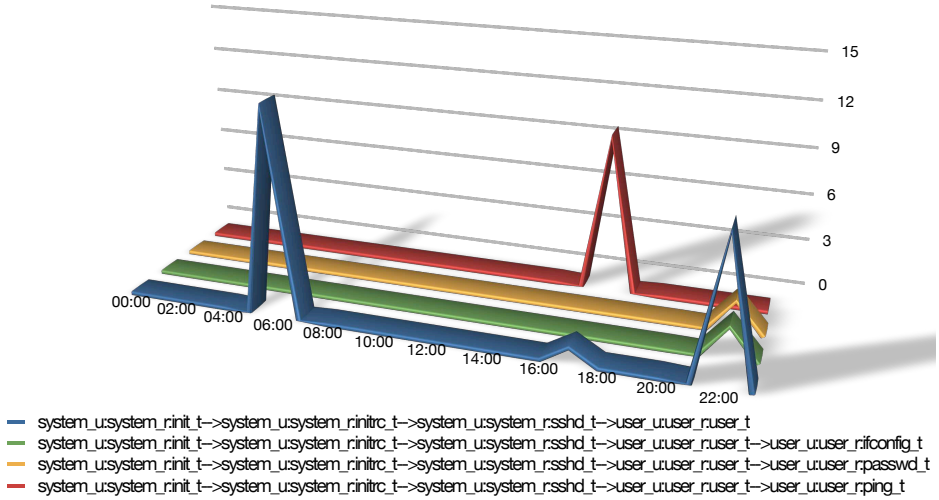


FIG. 6.7 – Alertes de changement de contextes du 10 novembre 2006.

vers Internet. Cette activité se traduit alors par 9 alertes impliquant l'utilisation de la commande `ping` associée au domaine `ping_t`, ces alertes se traduisent par la séquence de transitions suivante :

bad_transition = `system_u:system_r:init_t` $\xrightarrow{trans}$ `system_u:system_r:initrc_t` $\xrightarrow{trans}$ `system_u:system_r:sshd_t` $\xrightarrow{trans}$ `user_u:user_r:user_t` $\xrightarrow{trans}$ `user_u:user_r:ping_t`

Il a finalement ouvert quatre connexions `ssh` entre 22h et 23h. Lors de cette troisième connexion, il a tout d'abord changé son mot-de-passe (accès au contexte `user_u :user_r :passwd_t`) afin d'éviter que quelqu'un réutilise ce compte. Puis, il a vérifié la configuration réseau du système via la commande `ifconfig` (accès au contexte `user_u :user_r :ifconfig_t`).

Afin de comprendre ce que l'utilisateur a effectué lors de ce dernier accès au système, nous proposons d'étudier la figure 6.8. Cette figure contient les alertes violant la propriété de confidentialité (règle `confidentiality`) et un sous ensemble des alertes liées à la propriété d'intégrité du domaine utilisateur (règle `int_domain`). Tout d'abord, les alertes `user_u:user_r:user_t` $\xrightarrow{file:write}$ `user_u :user_r:user_tmp_t` impliquent que l'attaquant utilise des fichiers temporaires pour stocker de l'information. De plus, nous observons la présence des alertes :

- `user_u:user_r:user_t` $\xrightarrow{fd:use}$ `system_u:system_r:sshd_t` qui correspond à l'utilisation de la commande `ssh` ;
- `user_u:user_r:user_t` $\xrightarrow{node:rawip\_recv}$ `system_u:system_r:node_t` qui implique la réception de paquet réseau.

La présence de ces alertes est caractéristique de l'utilisation d'un outil de *scan* `ssh`. Ainsi, nous pouvons supposer que l'attaquant ait installé puis exécuté un outil de *scan*. Cette hypothèse a ensuite été vérifié par l'analyse des traces système et du répertoire temporaire. Finalement, l'attaquant a ouvert une connexion `ssh` vers 22h45 pour venir récupérer les résultats de son *scan*. En effet, la présence d'alertes `user_u:user_r:user_tmp_t > user_u:user_r:user_t` implique que l'attaquant a obtenu de l'information depuis des fichiers temporaires. Or ces fichiers ont été générés par l'outil de `scan` installé précédemment. L'attaquant est donc venu récupérer les résultats de son `scan` afin de découvrir d'autres systèmes ouverts. De plus, par combinaison, ce flux a engendré l'alerte suivante :

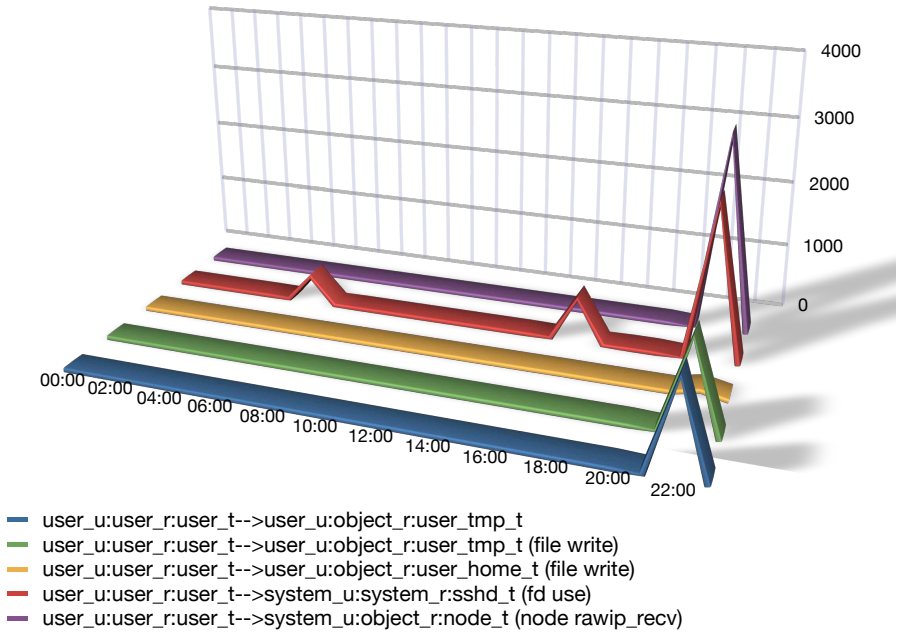


FIG. 6.8 – Alertes complémentaires du 10 novembre 2006.

$$\texttt{confidentiality} = (\ (\texttt{system\_u:system\_r:init\_t} \xrightarrow{trans} \texttt{system\_u:system\_r:initrc\_t} \xrightarrow{trans} \texttt{system\_u:system\_r:sshd\_t} \xrightarrow{trans} \texttt{user\_u:user\_r:user\_t}) \wedge (\texttt{user\_u:user\_r:user\_tmp\_t} > \texttt{user\_u:user\_r:user\_t})\ )$$

Cette alerte correspond à un *accès à l'information par transition* d'un fichier temporaire. La détection de cette activité complexe (composition de deux séquences) démontre la capacité de détection de notre solution.

6.3.4 Discussion

L'exemple précédent d'attaque permet d'illustrer le fonctionnement de notre système de détection d'intrusion. Il permet, de plus, d'identifier les quatre phases qui composent cette attaque :

1. Découverte d'un hôte ouvert via un *scan* réseau (à 6h) ;
2. Connexion et analyse du système découvert (à 17h) ;
3. Installation d'une application malicieuse tel qu'un logiciel de *scan* ou d'un robot *irc*, suivie de son exécution (à 22h) ;
4. Récupération des résultats de l'exécution de l'application malicieuse (à 22h45).

Nous avons observé que la majorité des attaques portées sur notre système respectent ce comportement. L'expérimentation effectuée sur notre plateforme sur une durée d'un an et l'analyse des alertes générées par notre IDS permettent de tirer plusieurs conclusions sur le comportement des attaquants :

- 99% des attaques subies ont pour objectif d'installer un outil de `scan` et/ou un robot `irc` ;
- Le comportement des attaquants respecte généralement les quatre phases évoquées précédemment ;

– Les attaquants qui se rendent compte que le système est un pot-de-miel quittent immédiatement le système ;
– Seuls deux attaquants ont essayé de se connecter sur le sous-réseau interne.

Ainsi, la majorité des attaquants installent un robot `irc`. L'attaquant contrôle ensuite ce robot via un canal `irc` dans le but de lancer des attaques de type *déni de service distribué*. De plus, ces attaquants utilisent le système afin de découvrir d'autres systèmes ouverts et ainsi installer un maximum de robots dans le but de construire ou d'agrandir un *botnet*[1].

Cette expérimentation montre que notre IDS est capable de détecter un grand nombre de violations de propriétés de sécurité et que certaines propriétés sont caractéristiques d'une activité malicieuse. Néanmoins, elle montre la nécessité d'une phase d'agrégation des alertes, afin de réduire le nombre d'alertes du même type, et d'une phase de corrélation, afin de reconstruire une *session d'attaque*. Une *session d'attaque* correspond alors à un ensemble d'alertes qui met en jeu plusieurs activités. Par exemple, l'attaque du 10 novembre correspond à une session d'attaque. Notons que le langage d'expression d'activités énoncé dans la section 3.2.4 permet d'exprimer une session. Par exemple, l'attaque précédente pourrait être décrite par la corrélation suivante :

installation_scanner_ssh =

$$
\begin{aligned}
&((\texttt{system\_u:system\_r:init\_t} \xrightarrow{trans} \texttt{system\_u:system\_r:initrc\_t} \xrightarrow{trans} \texttt{system\_u:system\_r:sshd\_t} \xrightarrow{trans} \texttt{user\_u:user\_r:user\_t}) \wedge \\
&(\texttt{system\_u:system\_r:init\_t} \xrightarrow{trans} \texttt{system\_u:system\_r:initrc\_t} \xrightarrow{trans} \texttt{system\_u:system\_r:sshd\_t} \xrightarrow{trans} \texttt{user\_u:user\_r:user\_t} \xrightarrow{trans} \texttt{user\_u:user\_r:ping\_t} \vee (\texttt{system\_u:system\_r:init\_t} \\
&\xrightarrow{trans} \texttt{system\_u:system\_r:initrc\_t} \xrightarrow{trans} \texttt{system\_u:system\_r:sshd\_t} \xrightarrow{trans} \texttt{user\_u:user\_r:user\_t} \xrightarrow{trans} \texttt{user\_u:user\_r:passwd\_t} \vee \texttt{system\_u:system\_r:init\_t} \xrightarrow{trans} \texttt{system\_u:system\_r:initrc\_t} \\
&\xrightarrow{trans} \texttt{system\_u:system\_r:sshd\_t} \xrightarrow{trans} \texttt{user\_u:user\_r:user\_t} \xrightarrow{trans} \texttt{user\_u:user\_r:ifconfig\_t}))) \wedge \\
&((\texttt{user\_u:user\_r:user\_t} \xrightarrow{fd:use} \texttt{system\_u:system\_r:sshd\_t} \vee \texttt{user\_u:user\_r:user\_t} \xrightarrow{node:rawip_r ecv} \texttt{system\_u:system\_r:node\_t}) \wedge \\
&(\texttt{system\_u:system\_r:init\_t} \xrightarrow{trans} \texttt{system\_u:system\_r:initrc\_t} \xrightarrow{trans} \texttt{system\_u:system\_r:sshd\_t} \xrightarrow{trans} \texttt{user\_u:user\_r:user\_t} \wedge \texttt{user\_u:user\_r:user\_tmp\_t} > \texttt{user\_u:user\_r:user\_t}))
\end{aligned}
$$

Cette session peut donc être décrite avec la corrélation : $((ssh) \wedge (ping \vee (passwd \vee ifconfig))) \wedge ((paquet\_ssh) \wedge (ssh \wedge flux))$. Cette corrélation correspond alors à une connexion `ssh` (phase 1), et l'utilisation de `ping` ou de `passwd` ou de `ifconfig` (phase 2), et le lancement du `scan ssh` (phase 3), et finalement une connexion `ssh` combinée avec un flux d'informations (phase 4). Cependant, la définition des activités correspondant à des sessions doit, actuellement, être réalisée manuellement. Ces corrélations pourraient cependant être obtenues par apprentissage. Nous discuterons de ces perspectives dans la conclusion de cette thèse.

6.4 Conclusion

Dans ce chapitre, nous avons proposé une implantation et une expérimentation de notre modèle de détection. Tout d'abord, nous avons montré que notre analyse des propriétés de sécurité était applicable sur des systèmes réels, tel que SELinux ou GRSECURITY. De plus, nous avons montré que cette implantation était capable d'énumérer et de détecter les activités illicites violant un ensemble de

[1] Un *botnet* est un réseau d'ordinateurs zombies détournés à l'insu de leurs propriétaires. En plus de servir à paralyser le trafic (attaque par déni de service), de moteur à la diffusion de spam, les botnets peuvent également être utilisés pour commettre des délits comme le vol de données bancaires et identitaires à grande échelle. Les botnets sont parfois loués à des tiers peu scrupuleux.

propriétés de sécurité. Finalement l'implantation montre que la phase de détection nécessite un temps de calcul et une quantité de mémoire adaptés à une utilisation sur un système réel.

L'expérimentation présentée dans la chapitre 6.3 a montré l'intérêt de ce mécanisme de détection. En effet, notre approche permet de détecter des attaques correspondant à des activités complexes. Contrairement aux systèmes de détections utilisés couramment, notre approche permet de détecter des activités qui correspondent réellement à des violations de propriétés de sécurité. De plus, elle s'applique à l'ensemble du système sans provoquer de surcharge importante. En effet, dans notre expérimentation, notre IDS analyse les traces de quatre systèmes tout en étant installé sur une machine qui héberge deux autres hôtes. Ainsi, notre approche montre sa capacité à détecter des intrusions effectives en se basant uniquement sur la définition d'une politique de contrôle d'accès et d'une politique de détection.

Chapitre 7

Conclusion

La contribution majeure de cette thèse est la formalisation des propriétés de sécurité système de la littérature et l'élaboration de nouvelles propriétés de sécurité qui n'existent dans aucun système de contrôle d'accès ou système de détection d'intrusions. Outre le formalisme proposé qui dessine un cadre formel unique à l'ensemble des propriétés étudiées par la communauté de recherche en sécurité, nous démontrons que nos nouvelles propriétés de sécurité peuvent être garanties par un nouveau système de détection d'intrusions. Cet outil de détection d'intrusions exploite les politiques système et le cadre formel développé dans cette thèse pour détecter toute activité illicite violant une des propriétés de sécurité proposées. Nos expérimentations qui se sont déroulées sur une année d'attaques contre un pot-de-miel démontrent la faisabilité et la précision de notre système de détection d'intrusions.

Tout d'abord, nous récapitulons les principales contributions de cette thèse. Dans la section "Travaux complémentaires", nous concluons ce résumé en donnant un bref aperçu des autres travaux réalisés durant cette thèse mais qui n'ont pas été détaillés dans ce manuscrit. Enfin, nous terminons ce chapitre en détaillant les différentes pistes qui nous semblent prometteuses compte tenu des résultats obtenus.

Dans le chapitre 2, nous avons établi un état de l'art des modèles de détection d'intrusions et de contrôle d'accès. Cet état de l'art nous permet de conclure à l'absence de solutions traitant des séquences. Dans le chapitre 3, nous avons proposé une formalisation et un langage pour décrire les activités d'un système. La formalisation repose sur une définition originale pour décrire les dépendances causales entre interactions d'un système. Le langage manipule notre notion de séquences causales et propose des opérateurs de corrélation qui permettent de combiner séquences et interactions. Les seules relations de cause à effet que notre langage ne peut pas traiter sont celles qui ne laissent pas de trace sur le système et qui reposent donc sur des éléments extérieurs. Ce langage permet non seulement de formaliser l'ensemble des propriétés de sécurité rencontrées classiquement dans la littérature, mais aussi de les étendre et d'en proposer de nouvelles. De plus, cette formalisation a permis de comparer l'ensemble des solutions existantes et de montrer clairement qu'aucune ne traite des séquences d'interactions ou des activités complexes résultant de la composition de séquences et d'interactions.

Dans le chapitre 4, nous montrons comment calculer le *graphe de dépendance causale* qui représente l'ensemble des dépendances autorisées. Nous montrons qu'il y a équivalence entre un chemin dans un graphe et une séquence observable. Pour chaque propriété de sécurité formalisée dans notre langage d'activités, nous proposons un algorithme d'énumération des activités illicites, c'est-à-dire ne respectant pas la définition de cette propriété. Ainsi, ces algorithmes correspondent à une implantation de notre langage de description d'activités. Notre méthode permet non seulement d'énumérer les activités simples qui correspondent à une violation d'une propriété de sécurité (interactions), mais

aussi les fermetures transitives de ces interactions (séquences) et leurs corrélations.

Dans le chapitre 5, nous proposons une application de l'analyse des propriétés de sécurité à la détection d'intrusions. Ainsi, nous proposons un système de détection d'intrusions basé sur trois phases : 1) *construction des graphes*, 2) *construction de la base de signatures* et 3) *détection des activités illicites*. La première phase utilise une politique de contrôle d'accès existante afin de construire un ensemble de graphes. Ces graphes sont ensuite analysés lors de la deuxième phase et permettent l'énumération de l'ensemble des activités violant une politique de détection. Cette *politique de détection*, définie par un administrateur, explicite l'ensemble des propriétés de sécurité qu'un système doit respecter. L'administrateur définit simplement un ensemble de propriétés de sécurité. Par exemple, la seule définition des deux propriétés générales d'intégrité et de confidentialité permet de garantir la protection des objets système. Si l'administrateur estime que cette définition surestime les activités illégales, il peut alors réutiliser les autres propriétés formalisées dans cette thèse. De plus, notre langage permet, à un administrateur, de définir de nouvelles propriétés suivant les objectifs de sécurité visés. Ainsi, lorsque la politique de détection ou la définition des propriétés amène à une surestimation des attaques et donc un fort taux de faux positifs, notre langage permet de raffiner la définition de ces propriétés. Notre approche est donc extensible aux besoins désirés.

La définition de la politique de détection est utilisée pour énumérer l'ensemble des activités qui peuvent être réalisées sur un système pour violer une des propriétés de sécurité. Les graphes fournissent alors un support pour l'énumération de ces activités illicites. Les algorithmes d'énumération proposés permettent de construire une base de signatures qui est ensuite utilisée lors de la phase de *détection des activités illicites*. Cette troisième phase analyse les traces d'interactions d'un système et génère des alertes pour chaque activité correspondant à une signature de la base et qui viole une des propriétés de sécurité. Notre approche permet donc de détecter les violations effectives des propriétés de sécurité requises pour un système.

Les phases de *construction des graphes* et de *construction de la base de signatures* doivent être appelées avant la phase de *détection*. Sur un système, ou un ensemble de système, nous n'avons pas besoin de reconstruire les graphes et la base tant que la *politique de contrôle d'accès* et la *politique de détection* ne changent pas. De plus, nous conservons l'état des observations en cours afin d'éviter tout faux négatif lié au redémarrage du système (un flux d'informations dont chaque moitié est réalisée avant et après le redémarrage). Finalement, lorsque la politique de contrôle d'accès ne change pas, les modifications de la politique de détection n'impliquent que la reconstruction de la base. Cette nouvelle base servira de nouvelle entrée pour la phase de détection et l'ancienne base peut être conservée afin de terminer la reconstruction des observations en cours.

L'application de l'analyse des propriétés à la détection d'intrusions nécessite l'existence d'une politique de contrôle d'accès. Par exemple, dans le cas de SELinux, cette politique est livrée avec le système et dans le cas de GRSECURITY, cette politique peut être obtenue par un mécanisme d'apprentissage. Ainsi, nous avons détaillé, dans le chapitre 6, l'application de notre approche à ces deux mécanismes MAC. Bien que les deux systèmes étudiés dans cette thèse reposent sur un mécanisme MAC, cette approche pourrait aussi être utilisée dans le cas d'un système DAC. Dans ce cas, la conversion de la politique cible en politique neutre est alors similaire à celle proposée pour GRSECURITY et elle nécessite de parcourir le système pour obtenir les droits d'accès aux objets. Notre solution s'applique donc à tous les systèmes qu'ils soient discrétionnaires ou mandataires. Nous avons choisi de l'illustrer sur des systèmes mandataires car ils sont plus avancés en terme de propriétés de sécurité et plus difficiles à prendre en compte. Ces systèmes DAC beaucoup plus rudimentaires ne doivent pas générer de problèmes supplémentaires. Les politiques de protection qu'ils utilisent sont seulement plus simples. Pour éviter les problèmes de génération d'un nouveau graphe d'interactions quand la politique de protection change, il est possible de définir une politique minimale qui liste tous les

contextes et autorise toutes les opérations.

Finalement, l'expérimentation, détaillée dans le chapitre 6, a montré que notre implantation permettait, non seulement de détecter les violations effectives des propriétés de sécurité, mais aussi les attaques correspondant à des activités complexes. De plus, les performances et les ressources nécessaires lors de la phase de détection montrent que cette approche est applicable sur un système réel. Ainsi, notre approche permet de détecter toutes les attaques violant une propriété, la seule condition pour une détection correcte étant que le système doit pouvoir générer les traces d'interactions nécessaires à cette phase de détection.

Travaux complémentaires

Dans le cadre de cette thèse, nous avons réalisé d'autres travaux relatifs à la détection d'intrusions et au contrôle d'accès. Ces travaux visent l'intégration et l'extension de l'approche de Méta-Politique, définie dans [Blanc 2006], au cas de la détection d'intrusions.

Approche Méta-Politique pour la configuration d'IDS

Nous avons tout d'abord appliqué le modèle Méta-Politique d'administration d'une politique de sécurité, défini dans [Blanc 2006], au cas des systèmes de détection d'intrusions. Ainsi, nous avons proposé une extension de ce modèle qui permet d'exprimer des politiques prenant en compte l'ajout et la suppression dynamique de propriétés pour les IDS. Cette extension a ainsi pour objectif de faciliter l'administration des mécanismes de détection en permettant que les propriétés évoluent dynamiquement sur chacun des nœuds. Ces travaux ont fait l'objet de plusieurs publications [Briffaut *et al.* 2006c, Blanc *et al.* 2006]

Architecture multi-niveaux

Une Méta-Politique définit un ensemble de règles de protection qui sont exprimées dans un format neutre. Afin de projeter ces règles sur les composants d'un système cible, nous avons proposé une architecture multi-niveaux. Cette architecture définit ainsi quatre niveaux de gestion d'une Méta-Politique dans un cadre distribué : 1) administration et répartition de la Méta-Politique, 2) déploiement de la politique de protection, 3) administration des outils sur étagère 4) gestion des alertes. Le premier niveau concerne l'administration d'une Méta-Politique, c'est à dire la définition des règles de protection et de modification. Le second niveau a pour objectif d'appliquer un contrôle des accès au niveau noyau en utilisant un mécanisme de contrôle d'accès MAC. Ce mécanisme est alors complété par notre système de détection (PIGA). Ce niveau a ainsi pour objectif de garantir le respect d'une politique de sécurité, qui comprend la définition de propriétés de sécurité, et de détecter les tentatives d'abus de la Méta-Politique. Le troisième niveau utilise des outils de protection et de détection classiques afin d'effectuer une détection supplémentaire (par exemple dans les paquets réseaux). Finalement les alertes générées par les mécanismes de niveau 2) et 3) sont stockées dans une base en vue d'une phase de corrélation globale. Cette architecture permet ainsi d'appliquer l'ensemble des règles définies dans une Méta-Politique sur un système qui réutilisent des outils existants (snort, netfilter, etc.). Ces travaux ont fait l'objet de plusieurs publications [Briffaut *et al.* 2005a, Briffaut *et al.* 2005b]

Plateforme Multi-Agent pour la détection d'intrusions

Finalement, nous avons défini et implanté une plateforme multi-agent [Briffaut *et al.* 2006b, Briffaut *et al.* 2006a] qui intègre les concepts de Méta-Politique et l'architecture multi-niveaux. Cette plateforme permet de gérer la configuration des systèmes de détection d'intrusions et des systèmes de protection via la définition d'une Méta-Politique. Pour ce faire, un administrateur définit, tout d'abord, une Méta-Politique qui explicite l'ensemble des règles de protection à appliquer sur le réseau, l'ensemble des machines disponibles et les outils de protection qui peuvent être installés sur chaque hôte (IDS ou mécanisme de contrôle d'accès). Les règles de protection de la Méta-Politique sont alors distribuées localement sur l'ensemble des outils disponibles via un mécanisme de répartition de ces règles. Ces règles étant définies dans un langage neutre, notre répartiteur les distribue sur l'ensemble des outils installés et génère alors la configuration de ces outils via un mécanisme de traduction. Lorsqu'une règle ne peut pas être appliquée par un outil, cette règle est alors projetée sur un autre outil. Les règles non applicables donnent alors lieu à des alertes spécifiques. De plus, les administrateurs locaux peuvent adapter les règles de protection locales en utilisant des règles de modification incluses dans la Méta-Politique. Ces règles permettent ainsi d'adapter les règles de protection, définies dans la Méta-Politique, au système local sans nécessiter de communication réseau. Cette approche permet aussi une administration entièrement décentralisée de tout un système réparti. Cela permet de garantir des propriétés à l'échelle du système réparti tout en autorisant des modifications locales des politiques de sécurité.

Perspective

Afin d'améliorer la pertinence de nos travaux, et d'étendre son champ d'applicabilité, des perspectives s'ouvrent sur plusieurs axes : compiler des propriétés définies dans notre langage de description d'activités en algorithmes d'énumérations, étendre le langage de description d'activités afin de prendre en compte des facteurs telle que la fréquence d'apparition, corréler les alertes générées lors de la phase de détection, obtenir les activités complexes du type session d'attaques par apprentissage, appliquer notre approche au contrôle actif des activités.

Application aux politiques dynamiques
Nous travaillons actuellement sur une extension de notre modèle afin de prendre en compte le cas des politiques dynamiques. Nous réutilisons ainsi le concept de Méta-Graphe, défini dans [Blanc 2006], afin de construire le graphe d'interactions associé à une Méta-Politique. Ce graphe est alors utilisé pour calculer des séquences dynamiques, c'est-à-dire contenant des expressions régulières, et une base de signatures dynamique. Cette base contient alors l'ensemble des séquences qui peuvent apparaître sur un système lors d'opérations de modification de la politique de contrôle d'accès (ajout ou suppression de règles). Lors de telle modification, un ensemble de nouvelles signatures est alors calculé à partir des signatures dynamiques et ajouté à la base de signatures. L'avantage de cette approche est donc que la base est créée lors du déploiement d'une Méta-Politique et n'est plus recalculée lors d'opération sur la politique de contrôle d'accès. En effet, seules les séquences dynamiques sont recalculées. De plus, cette approche permet de représenter plus fidèlement des systèmes DAC sans faire l'hypothèse que tous les accès soient autorisés. Nous avons actuellement défini le modèle associé à cette détection et son implantation est en cours.

Compilateur du langage de description d'activités Actuellement, dans notre implantation, les algorithmes d'énumération des activités illicites sont définis manuellement. Or, les propriétés de sécurité correspondant à ces algorithmes sont définies dans notre langage de description d'activités. Nous

proposons donc, tout d'abord, d'étendre ce langage afin de prendre en compte les quantificateurs et les structures de contrôle utilisés dans la définition des propriétés de sécurité. Nous comptons ensuite établir un compilateur de ce langage afin de générer automatiquement l'algorithme d'énumération associé à une propriété de sécurité. Ainsi, un administrateur pourra définir plus facilement via le langage de nouvelles propriétés de sécurité adaptées à ces systèmes ou raffiner les propriétés existantes afin de supprimer d'éventuels faux positifs.

Extension du langage de description d'activités Dans le chapitre 3, nous avons évoqué le fait que la formalisation des propriétés de disponibilité nécessite la prise en compte d'autres facteurs non présents dans le langage actuel. Nous proposons donc d'étendre notre langage de description d'activités afin de prendre en compte des facteurs supplémentaires telle que la fréquence ou l'intensité d'apparition des activités. Ces extensions nécessitent, de plus, la définition de nouveau reconstructeur temporel prenant en compte la durée et la fréquence des activités. Ainsi, notre approche pourrait être étendue à un spectre de propriétés plus large incluant les attaques distribuées et la disponibilité.

Corrélation des alertes Nous avons vu, dans le chapitre 6, que la présence de certaines signatures aboutit à la génération d'un grand nombre d'alertes. En effet, notre approche étant basée sur l'analyse des traces d'interactions, la reconstruction génère actuellement une alerte par activité illicite reconstruite. Tout d'abord, nous comptons inclure une phase d'agrégation des alertes permettant de regrouper les alertes identiques et ainsi réduire le nombre d'alertes. De plus, nous comptons introduire une phase de corrélation de ces alertes dans le but de 1) réduire le nombre d'alertes, 2) caractériser les alertes spécifiques à une attaque, et 3) caractériser des sessions d'attaques. Ces trois points ont pour objectif de faciliter la gestion des alertes et leurs traitements.

Apprentissage des activités complexes Un second aspect qui découle de la corrélation concerne l'apprentissage des activités complexes. Comme évoqué dans le chapitre 6, notre langage permet de décrire une session d'attaque complexe. Ainsi, nous étudions la possibilité de générer automatiquement les signatures pour ces sessions, une phase de corrélation et d'apprentissage des alertes. Une thèse est actuellement en cours sur ce sujet.

Vers un contrôle actif des activités Dans cette thèse, nous avons proposé une application de notre formalisme des propriétés de sécurité au cas de la détection d'intrusions. Actuellement, une alerte est générée lorsqu'une activité illicite est reconstruite, ce qui correspond à un contrôle passif des activités illicites. Cependant, nous souhaitons proposer une seconde application qui vise le contrôle actif des activités. Nous étudions actuellement l'intégration de nos reconstructeurs d'activités en tant que module du noyau GNU/Linux afin de bloquer les activités illicites, par exemple en interdisant la dernière interaction d'une activité. Cette intégration exploite alors la possibilité, offerte par le noyau GNU/Linux, d'interagir avec le contrôle de SELinux ou GRSECURITY via un module de sécurité noyau.

Chapitre 8

Bibliographie

[Abou El Kalam *et al.* 2003] Abou El Kalam, A., Baida, R. E., Balbiani, P., Benferhat, S., Cuppens, F., Deswarte, Y., Miège, A., Saurel, C. et Trouessin, G. (2003). Organization based access control. *In 4th IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY 2003)*, pages 120–131. IEEE Computer Society Press.

[Alpers et Plansky 1994] Alpers, B. et Plansky, H. (1994). Domain and policy based management : Concepts and implementation architecture. *In The Fifth IFIP/IEEE International Workshop on Distributed Systems : Operations and Management (DSOM '94)*, Toulouse, France.

[Ammann et Sandhu 1992] Ammann, P. et Sandhu, R. S. (1992). The extended schematic protection model. 1(3-4):335–384.

[Anderson *et al.* 1994] Anderson, D., Frivold, T., Tamaru, A. et Valdes, A. (1994). Next-generation intrusion detection expert system (nides), software users manual, beta-update release. Rapport technique SRI–CSL–95–07, Computer Science Laboratory, SRI International, 333 Ravenswood Avenue, Menlo Park, CA 94025-3493.

[Anderson 1980] Anderson, J. (1980). Computer security threat monitoring and surveillance. Rapport technique, James P. Anderson Company, Fort Washington, Pennsylvania.

[Anderson 1972] Anderson, J. P. (1972). Computer security technology planning study. Technical Report ESD-TR-73-51, Air Force Electronic Systems Division, Bedford, MA.

[Avitabile 1998] Avitabile, M. (1998). An examination of requirements for metapolicies in policy-based management. Master's thesis, Munich University of Technology.

[Bace et Mell 2001] Bace, R. et Mell, P. (2001). Intrusion detection systems. Rapport technique, National Institute of Standards and Technology (NIST).

[Badger *et al.* 1995] Badger, L., Sterne, D. F., Sherman, D. L. et Walker, K. M. (1995). A domain and type enforcement UNIX prototype. *In Proceedings of the 5th USENIX UNIX Security Symposium*, pages 127–140, Salt Lake City, Utah, USA.

[Baur et Weiss 1988] Baur, A. et Weiss, W. (1988). Audit analysis tool for systems with high demands regarding security and access control. Rapport technique ZFE F2 SOF 42, Siemens Nixdorf Software.

[Bell et La Padula 1973] Bell, D. E. et La Padula, L. J. (1973). Secure computer systems : Mathematical foundations and model. Technical Report M74-244, The MITRE Corporation, Bedford, MA.

[Belokosztolszki et Moody 2002] Belokosztolszki, A. et Moody, K. (2002). Meta-policies for distributed role-based access control systems. *In 3rd IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY 2002)*, pages 106–115, Monterey, California, USA. IEEE Computer Society Press.

[Biba 1975] Biba, K. J. (1975). Integrity considerations for secure computer systems. Technical Report MTR-3153, The MITRE Corporation.

[Bishop 2003] Bishop, M. (2003). *Computer Security Art and Science*. Numéro ISBN 0201440997. Addison-Wesley Professional.

[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, Orléans, France.

[Blanc *et al.* 2006] Blanc, M., Briffaut, J., Lalande, J.-F. et Toinard, C. (2006). Distributed control enabling consistent MAC policies and IDS based on a meta-policy approach. *In Proceedings of the Seventh IEEE International Workshop on Policies for Distributed Systems and Networks (Policy 2006)*, pages 153–156, London, Canada. IEEE Computer Society.

[Blanc *et al.* 2004] Blanc, M., Clemente, P., Courtieu, P., Franche, S., Oudot, L., Toinard, C. et Vessiller, L. (2004). Hardening large-scale networks security through a meta-policy framework. *In* Wysocki, B. J. et Wysocki, T. A., éditeurs : *Third Workshop on the Internet, Telecommunications and Signal Processing (WITSP'04)*, pages 132–137, Adelaide, Australia. DSP for Communication Systems.

[Boebert et Kain 1985] Boebert, W. E. et Kain, R. Y. (1985). A practical alternative to hierarchical integrity policies. *In The 8th National Computer Security Conference*, pages 18–27, Gaithersburg, MD, USA.

[Brewer et Nash 1989] Brewer, D. F. C. et Nash, M. J. (1989). The chinese wall security policy. *In Proceedings of the IEEE Symposium on Security and Privacy*, pages 206–214, Oakland, CA, USA. IEEE.

[Briffaut 2005] Briffaut, J. (2005). Meta-policy oriented intrusion detection. *In The First Colloquium on Risk and Security of the Internet and Systems (CRiSIS 2005)*, pages 77–94, Bourges, France. Ecole Nationale Supérieure d'Ingénieurs de Bourges.

[Briffaut *et al.* 2005a] Briffaut, J., Abou El Kalam, A., Toinard, C. et Blanc, M. (2005a). Intrusion detection and security policy framework for distributed environments. *In Proceedings of the 2005 International Symposium on Collaborative Technologies and Systems (CTS'05)*, pages 100–106, Saint Louis, USA.

[Briffaut *et al.* 2005b] Briffaut, J., Abou El Kalam, A., Toinard, C., Blanc, M. et Oudot, L. (2005b). Multi-level intrusion detection system (MIDS). *In The 4th Conference on Security and Network Architectures (SAR'05)*, pages 145–155, Batz sur Mer, France.

[Briffaut *et al.* 2006a] Briffaut, J., Clement, P., Gad El Rab, M., Toinard, C. et Blanc, M. (2006a). A multi-agent and multi-level architecture to secure distributed systems. *In Proceedings of the First International Workshop on Privacy and Security in Agent-based Collaborative Environments (PSACE 2006)*, Hakodate, Japan.

[Briffaut *et al.* 2006b] Briffaut, J., Clement, P., Rab, M. G. E., Toinard, C. et Blanc, M. (2006b). A collaborative approach for access control, intrusion detection and security testing. *In* Smari, W. W. et McQuay, W., éditeurs : *Proceedings of the 2006 International Symposium on Collaborative Technologies and Systems, Special Session on Multi Agent Systems and Collaboration (MASC 2006)*, pages 270–278, Las Vegas, USA. IEEE Computer Society.

[Briffaut *et al.* 2006c] Briffaut, J., Lalande, J.-F., Toinard, C. et Blanc, M. (2006c). Collaboration between MAC policies and ids based on a meta-policy approach. *In* Smari, W. W. et McQuay, W., éditeurs : *Proceedings of the Workshop on Collaboration and Security (COLSEC'06)*, pages 48–55, Las Vegas, USA. IEEE Computer Society.

[Carrasco et Oncina 1994] Carrasco, R. C. et Oncina, J. (1994). Learning stochastic regular grammars by means of a state merging method. *In* Carrasco, R. C. et Oncina, J., éditeurs : *Grammatical Inference and Applications, Second International Colloquium, ICGI-94, Alicante, Spain, September 21-23, 1994, Proceedings*, pages 139–152. Springer.

[Clark et Wilson 1987a] Clark, D. D. et Wilson, D. R. (1987a). A comparison of commercial and military computer security policies. *In the Symposium on Security and Privacy 1987*, pages 184–193. IEEE Press.

[Clark et Wilson 1987b] Clark, D. D. et Wilson, D. R. (1987b). A comparison of commercial and military computer security policies. *In Proceedings of the IEEE Symposium on Security and Privacy*, pages 184–194, Oakland, CA, USA. IEEE.

[Cuppens et Miège 2003] Cuppens, F. et Miège, A. (2003). Modelling contexts in the Or-BAC model. *In 19th Annual Computer Security Applications Conference (ACSAC 2003)*, Las Vegas, Nevada, USA.

[Cuppens et Ortalo 2000] Cuppens, F. et Ortalo, R. (2000). Lambda : A language to model a database for detection of attacks. *In* Debar, H., Mé, L. et Wu, S. F., éditeurs : *Recent Advances in Intrusion Detection, Third International Workshop, RAID 2000, Toulouse, France, October 2-4, 2000, Proceedings*, pages 197–216. Springer.

[Damianou *et al.* 2000] Damianou, N., Dulay, N., Lupu, E. et Sloman, M. (2000). Ponder : A language for specifying security and management policies for distributed systems. Rapport technique Research Report DoC 2000/1, Imperial College.

[Debar *et al.* 1992] Debar, H., Becker, M. et Siboni, D. (1992). A neural network component for an intrusion detection system. *In IEEE Symposium on Research in Security and Privacy*, pages 1 – 11. Oakland.

[Debar et Dorizzi 1992] Debar, H. et Dorizzi, B. (1992). An application of a recurrent network to an intrusion detection system. volume 2, pages 478–483 vol.2.

[Denning 1987] Denning, D. E. (1987). An intrusion-detection model. 13(2):222–232.

[Department of Defense 1991] Department of Defense (1991). Integrity in Automated Information Systems. Technical Report C 79-91, National Computer Security Center.

[DuMouchel et Schonlau 1998] DuMouchel, W. et Schonlau, M. (1998). A fast computer intrusion detection algorithm based on hypothesis testing of command transition probabilities. *In KDD*, pages 189–193.

[Eckmann *et al.* 2000] Eckmann, S., Vigna, G. et Kemmerer, R. (2000). Statl : An attack language for state-based intrusion detection.

[Eckmann *et al.* 2002] Eckmann, S. T., Vigna, G. et Kemmerer, R. A. (2002). Statl : An attack language for state-based intrusion detection. 10(1/2):71–104.

[Eppstein 1997] Eppstein, D. (March 31 1997). Finding the k shortest paths.

[F. Cuppens et A. Miège 2003] F. Cuppens et A. Miège (2003). Administration Model for Or-BAC. *In Workshop on Metadata for Security, International Federated Conferences (OTM'03)*, pages 754–768, Catania, Sicily, Italy.

[F. Cuppens et A. Miège 2004] F. Cuppens et A. Miège (2004). AdOrBAC : An Administration Model for Or-BAC. *Special issue of the International Journal of Computer Systems Science and Engineering*, 19(3).

[Ferraiolo et Kuhn 1992] Ferraiolo, D. F. et Kuhn, D. R. (1992). Role-based access controls. *In 15th National Computer Security Conference*, pages 554–563, Baltimore, MD, USA.

[Focardi et Gorrieri 2001] Focardi, R. et Gorrieri, R. (2001). Classification of security properties (part i : Information flow).

[Forrest *et al.* 1997] Forrest, S., Hofmeyr, S. A. et Somayaji, A. (1997). Computer immunology. *Communications of the ACM*, 40(10):88–96.

[Fox *et al.* 1990] Fox, K., Henning., R. et Reed, J. (1990). A neural network approach towards intrusion detection. *In 1990 Symposium on Research in Security and Privacy*, pages 125–134.

[Guttman *et al.* 2003] Guttman, J., Herzog, A. et Ramsdell, J. (2003). Information Flow in Operating Systems : Eager Formal Methods. *In Workshop on Issues in the Theory of Security (WITS'03)*, Warsaw, Poland.

[Habra *et al.* 1992] Habra, N., Charlier, B. L., Mounji, A. et Mathieu, I. (1992). Asax : Software architecture and rule-based language for universal audit trail analysis. *In* Deswarte, Y., Eizenberg, G. et Quisquater, J.-J., éditeurs : *Computer Security - ESORICS 92, Second European Symposium on Research in Computer Security, Toulouse, France, November 23-25, 1992, Proceedings*, pages 435–450. Springer.

[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.

[Hosmer 1992a] Hosmer, H. H. (1992a). Metapolicies I. *ACM SIGSAC Review*, 10(2–3):18–43.

[Hosmer 1992b] Hosmer, H. H. (1992b). Metapolicies II. *In The 15th National Computer Security Conference*, pages 369–378, Baltimore, MD.

[ITSEC 1991] ITSEC (1991). Information Technology Security Evaluation Criteria (ITSEC) v1.2. Technical report.

[Javitz et Valdes 1993] Javitz, H. S. et Valdes, A. (1993). The NIDES statistical component : description and justification.

[Ko *et al.* 1994] Ko, C., Fink, G. et Levitt, K. (1994). Automated detection of vulnerabilities in privileged programs by execution monitoring. *In Proceedings of the 10th Annual Computer Security Applications Conference*, pages 134–144, Orlando, FL. IEEE Computer Society Press.

[Ko et Redmond 2002] Ko, C. et Redmond, T. (2002). Noninterference and intrusion detection. *In IEEE Symposium on Security and Privacy*, pages 177–187.

[Ko *et al.* 1997] Ko, C., Ruschitzka, M. et Levitt, K. (1997). Execution monitoring of security-critical programs in distributed systems : A specification-based approach. pages 175–187.

[Kosoresow et Hofmeyr 1997] Kosoresow, A. P. et Hofmeyr, S. A. (1997). Intrusion detection via system call traces. 14(5):35–42.

[Kumar et Spafford 1994] Kumar, S. et Spafford, E. H. (1994). A Pattern Matching Model for Misuse Intrusion Detection. *In Proceedings of the 17th National Computer Security Conference*, pages 11–21.

[Lamport 1978] Lamport, L. (1978). Time, clocks, and the ordering of events in a distributed system. 21(7):558–565.

[Lampson 1969] Lampson, B. W. (1969). Dynamic protection structures. *In AFIPS Fall Joint Computer Conference (FJCC 1969)*, volume 35, pages 27–38, Las Vegas, Nevada, USA. AFIPS Press.

[Lampson 1971] Lampson, B. W. (1971). Protection. *In The 5th Symposium on Information Sciences and Systems*, pages 437–443, Princeton University.

[Lampson 1973] Lampson, B. W. (1973). A note on the confinement problem. *Commun. ACM*, 16(10):613–615.

[Lee *et al.* 1998] Lee, W., Stolfo, S. J. et Mok, K. W. (1998). Mining audit data to build intrusion detection models. *In KDD*, pages 66–72.

[Lindqvist et Porras 1999] Lindqvist, U. et Porras, P. A. (1999). Detecting computer and network misuse through the production-based expert system toolset (p-best). *In Proceedings of the 1999 IEEE Symposium on Security and Privacy*, pages 146–161, Oakland, California. IEEE Computer Society Press, Los Alamitos, California.

[Loscocco et Smalley 2001] Loscocco, P. et Smalley, S. (2001). Integrating flexible support for security policies into the linux operating system. *In Proceedings of the FREENIX Track : 2001 USENIX Annual Technical Conference (FREENIX '01)*. USENIX.

[Loscocco *et al.* 1998] Loscocco, P. A., Smalley, S. D., Muckelbauer, P. A., Taylor, R. C., Turner, S. J. et Farrell, J. F. (1998). The Inevitability of Failure : The Flawed Assumption of Security in Modern Computing Environments. *In Proceedings of the 21st National Information Systems Security Conference*, pages 303–314, Arlington, Virginia, USA.

[Lunt *et al.* 1992] Lunt, T., Tamaru, A., Gilham, F., Javitz, R. J. P. N. H., Valdes, A. et Garvey, T. (1992). A real-time intrusion detection expert system (ides). Rapport technique, SRI International,Computer Science Laboratory.

[Lunt 1990] Lunt, T. F. (1990). Ides : An intelligent system for detecting intruders. *In Symposium : Computer Security*, Rome, Italy. Threat and Countermeasures.

[Lupu et Sloman 1999] Lupu, E. C. et Sloman, M. (1999). Conflicts in policy-based distributed systems management. *IEEE Transactions on Software Engineering*, 25(6):852–896.

[Marriott 1993] Marriott, D. A. (1993). Management policy specification. Rapport technique DoC 94/1, Imperial College, London.

[Meier 2004] Meier, M. (2004). Intrusion detection systems list and bibliography.

[Mell *et al.* 2003] Mell, P., Hu, V., Lippmann, R., Haines, J. et Zissman, M. (2003). An overview of issues in testing intrusion detection systems. Rapport technique IR 7007, National Institute of Standards and Technology / Massachusetts Institute of Technology Lincoln Laboratory.

[Michel et Mé 2001] Michel, C. et Mé, L. (2001). Adele : An attack description language for knowledge-based intrusion detection. *In* Dupuy, M. et Paradinas, P., éditeurs : *Trusted Information : The New Decade Challenge, IFIP TC11 Sixteenth Annual Working Conference on Information Security (IFIP/Sec'01), June 11-13, 2001, Paris, France*, pages 353–368. Kluwer.

[Minsky et Ungureanu 2000] Minsky, N. H. et Ungureanu, V. (2000). Law-governed interaction : a coordination and control mechanism for heterogeneous distributed systems. *ACM Transactions on Software Engineering and Methodology*, 9(3):273–305.

[Moffett et Sloman 1991] Moffett, J. D. et Sloman, M. S. (1991). The representation of policies as system objects. *In The Conference on Organizational Computer Systems*, pages 171–184, Atlanta, Georgia, USA.

[Moffett et Sloman 1993] Moffett, J. D. et Sloman, M. S. (1993). Policy hierarchies for distributed system management. *IEEE Journal on Selected Areas in Communications, Special Issue on Network Management and Control*, 11(9):1404–1414.

[Moffett *et al.* 1990] Moffett, J. D., Sloman, M. S. et Twidle, K. P. (1990). Specifying discretionary access control policy for distributed systems. *Computer Communications*, 13(9):571–580.

[Morris 2004] Morris, J. (2004). Recent developments in selinux kernel performance.

[Mounji et Charlier 1997] Mounji, A. et Charlier, B. L. (1997). Continuous assessment of a unix configuration : Integrating intrusion detection and configuration analysis. *In Proceedings of the Network and Distributed System Security Symposium, NDSS 1997, San Diego, California, USA*. IEEE Computer Society.

[Mé 1998] Mé, L. (1998). Gassata, a genetic algorithm as an alternative tool for security audit trail analysis. *In the first international workshop on Recent Advances in Intrusion Detection*.

[Neumann et Porras 1999] Neumann, P. G. et Porras, P. A. (1999). Experience with emerald to date. *In Proceedings of the Workshop on Intrusion Detection and Network Monitoring, Santa Clara, CA, USA, April 9-12, 1999*, pages 73–80. USENIX.

[Porras et Neumann 1997] Porras, P. A. et Neumann, P. G. (1997). Emerald : Event monitoring enabling responses to anomalous live disturbances. *In Proc. 20th NIST-NCSC National Information Systems Security Conference*, pages 353–365.

[Pouzol et Ducass 2002] Pouzol, J.-P. et Ducass, M. (2002). Formal specification of intrusion signatures and detection rules. *In CSFW '02 : Proceedings of the 15th IEEE Computer Security Foundations Workshop (CSFW'02)*, page 64, Washington, DC, USA. IEEE Computer Society.

[Roesch 1999] Roesch, M. (1999). Snort - lightweight intrusion detection for networks. *In LISA '99 : Proceedings of the 13th USENIX conference on System administration*, pages 229–238, Berkeley, CA, USA. USENIX Association.

[Ron *et al.* 1996] Ron, D., Singer, Y. et Tishby, N. (1996). The power of amnesia : Learning probabilistic automata with variable memory length. 25(2-3):117–149.

[Ryan *et al.* 1998] Ryan, J., Lin, M.-J. et Miikkulainen, R. (1998). Intrusion detection with neural networks. *In* Jordan, M. I., Kearns, M. J. et Solla, S. A., éditeurs : *Advances in Neural Information Processing Systems*, volume 10. The MIT Press.

[Saltzer et Schroeder 1975] Saltzer, J. et Schroeder, M. (1975). The protection of information in computer systems. *Proc. IEEE*, 63(9):1278–1308.

[Sandhu 1990] Sandhu, R. (1990). Separation of duties in computerized information systems. *In IFIP WG11.3 Workshop on Database Security*.

[Sandhu 1988] Sandhu, R. S. (1988). The schematic protection model : Its definition and analysis for acyclic attenuating schemes. *Journal of the ACM*, 35(2):404–432.

[Sandhu 1992] Sandhu, R. S. (1992). The Typed Access Matrix Model. *In Proceedings of the IEEE Symposium on Research in Security and Privacy*, pages 122–136, Oakland, CA, USA. IEEE.

[Secure Computing Corporation 1997] Secure Computing Corporation (1997). DTOS generalized security policy specification. Rapport technique, Secure Computing Corporation.

[Sekar *et al.* 1999] Sekar, R., Bowen, T. F. et Segal, M. E. (1999). On preventing intrusions by process behavior monitoring. *In Proceedings of the Workshop on Intrusion Detection and Network Monitoring, Santa Clara, CA, USA, April 9-12, 1999*, pages 29–40. USENIX.

[Sekar *et al.* 1998] Sekar, R., Cai, Y. et Segal, M. (1998). A specification-based approach for building survivable systems. *In Proc. 21st NIST-NCSC National Information Systems Security Conference*, pages 338–347.

[Sloman 1994] Sloman, M. (1994). Policy driven management for distributed systems. *Journal of Network and Systems Management*, 2(4):333–360.

[Sloman *et al.* 1992] Sloman, M., Moffett, J. et Twidle, K. (1992). Domino domains and policies : An introduction to the project results. Rapport technique Domino Report Arch/IC/4, Imperial College, London.

[Smalley *et al.* 2001] Smalley, S., Vance, C. et Salamon, W. (2001). Implementing SELinux as a linux security module. Rapport technique, NSA.

[Spencer *et al.* 1999] Spencer, R., Smalley, S., Loscocco, P., Hibler, M., Andersen, D. et Lepreau, J. (1999). The Flask security architecture : System support for diverse security policies. *In Proceedings of The Eighth USENIX Security Symposium*, pages 123–129, Washington, D.C., USA.

[Spengler 2002] Spengler, B. (2002). Detection, prevention, and containment : A study of grsecurity. *In Libre Software Meeting 2002 (LSM2002)*, Bordeaux, France. `http://www.grsecurity.net/papers.php`.

[TCSEC 1985] TCSEC (1985). Trusted Computer System Evaluation Criteria. Technical Report DoD 5200.28-STD, Department of Defense.

[Uppuluri et Sekar 2001] Uppuluri, P. et Sekar, R. (2001). Experiences with specification-based intrusion detection. *In* Lee, W., Mé, L. et Wespi, A., éditeurs : *Recent Advances in Intrusion Detection, 4th International Symposium, RAID 2001 Davis, CA, USA, October 10-12, 2001, Proceedings*, pages 172–189. Springer.

[Vaccaro et Liepins 1989] Vaccaro, H. S. et Liepins, G. E. (1989). Detection of anomalous computer session activity. *In IEEE Symposium on Security and Privacy*, pages 280–289.

[Vigna et Kemmerer 1998] Vigna, G. et Kemmerer, R. A. (1998). Netstat : A network-based intrusion detection approach. *In 14th Annual Computer Security Applications Conference (ACSAC 1998), 7-11 December 1998, Scottsdale, AZ, USA*, pages 25–36. IEEE Computer Society.

[Wies 1994] Wies, R. (1994). Policies in network and systems management - formal definition and architecture. *Journal of Network and Systems Management*, 2(1):63–83.

[Wood et Erlinger 2007] Wood, M. et Erlinger, M. (2007). Intrusion detection message exchange requirements. IETF Intrusion Detection Exchange Format Working Group. Request for Comments. Reference : 'rfc4766'.

[Wright *et al.* 2002] Wright, C., Cowan, C., Smalley, S., Morris, J. et Kroah-Hartman, G. (2002). Linux security modules : General security support for the linux kernel. *In Proceedings of the 11th USENIX Security Symposium*, San Francisco, CA, USA.

[Zimmermann 2003] Zimmermann, J. (2003). Détection d'intrusion paramétrée par la politique par controle de flux de références. Mémoire de D.E.A., Supélec Rennes.

[Zimmermann *et al.* 2003a] Zimmermann, J., Mé, L. et Bidan, C. (2003a). Experimenting with a policy-based hids based on an information flow control model. *In 19th Annual Computer Security Applications Conference (ACSAC 2003), 8-12 December 2003, Las Vegas, NV, USA*, pages 364–373. IEEE Computer Society.

[Zimmermann *et al.* 2003b] Zimmermann, J., Mé, L. et Bidan, C. (2003b). An improved reference flow control model for policy-based intrusion detection. *In* Snekkenes, E. et Gollmann, D., éditeurs : *Computer Security - ESORICS 2003, 8th European Symposium on Research in Computer Security, Gjøvik, Norway, October 13-15, 2003, Proceedings*, pages 291–308. Springer.

Annexe A

Tableaux récapitulatifs des propriétés de sécurité

Propriété	Type	Règle de recherche	Paramètre	Objectif
Intégrité	Intégrité des objets	`integrity`	$\{sc_{source} \in SC_S, sc_{cible} \in SC_O, G_c\}$	sc_{source} ne doit pas modifier sc_{cible}
			$\{SC_{source} \subset SC_S, SC_{cible} \subset SC_O, G_c\}$	Chaque $sc_{source} \in SC_{source}$ ne doit pas modifier un $sc_{cible} \in SC_{cible}$
	Intégrité Biba	`int_biba`	$\{SC_{int} \subset SC\}$ $\forall sc \in SC_{int}, attr_{integrite} \in sc$	Application du modèle BIBA au sous-ensemble de contexte SC_{int} Chaque contexte est étiqueté par un niveau d'intégrité
	Intégrité des sujets	`int_subject`		Application du modèle de non-interfèrence sur l'ensemble du système
			$\{SC_{noninter} \subset SC_S\}$	Application du modèle de non-interfèrence sur l'ensemble SC
	Intégrité des domaines	`int_domain`	$\{TCB_{in} \subset SC\}$	Aucune interaction entre TCB_{in} et le reste du système
Confidentialité	Absence de flux d'informations	`confidentiality`	$\{sc_{source} \in SC, sc_{cible} \in SC, G_c\}$	Flux d'informations interdits entre sc_{cible} et sc_{source}
			$\{SC_{source} \subset SC, SC_{cible} \subset SC, G_c\}$	Flux d'informations interdit entre chaque $sc_{cible} \in SC_{source}$ et chaque $sc_{source} \in SC_{cible}$
	Bell&Lapadula	`conf_blp`	$\{SC_{hab} \subset SC\}$ $\forall sc \in SC_{hab}, attr_{habilitation} \in sc$	Application du modèle BLP au sous-ensemble de contexte SC_{hab} Chaque contexte est étiqueté par un niveau de confidentialité
	Bell&Lapadula restrictive	`conf_blpr`	$\{SC_{hab} \subset SC\}$ $\forall sc \in SC_{hab}, attr_{habilitation} \in sc$	Application du modèle BLP restrictif au sous-ensemble de contexte SC_{hab} Chaque contexte est étiqueté par un niveau de confidentialité
	Cohérence d'accès aux données	`conf_data`	$\{sc_{source} \in SC_S, sc_{cible} \in SC_O, G_c\}$	Un contexte ne peut pas accéder à une donnée non accessible directement
			$\{SC_{source} \subset SC_S, SC_{cible} \subset SC, G_c\}$	Un contexte $\{sc_{source}$ ne peut pas accéder à une donnée sur sc_{cible} qui n'est pas accessible directement
Abus de privilège	Séparation de privilèges	`duties_sep`	$\{G_c\}$	Les contextes sujets n'ont pas le droit de modifier puis d'exécuter un objet
			$\{SC_{daemon} \in SC_S, G_c\}$	Les contextes sujets de l'ensemble SC_{daemon} n'ont pas le droit de modifier puis d'exécuter un contexte objet Application de la séparation de privilèges aux services système
	Absence de changement de contexte	`bad_transition`	$\{sc_{source} \in SC_S, sc_{cible} \in SC_S\}$	un contexte sc_{source} ne peut obtenir le contexte sc_{cible}
			$\{SC_{source} \subset SC_S, SC_{cible} \subset SC_S\}$	Tout contexte $sc_{source} \in SC_{source}$ ne peut obtenir le contexte sc_{cible} sur $sc_{cible} \in SC_{cible}$ qui n'est pas accessible directement
	Exécutables de confiance	`tpe`	$\{TPE \subset SC\}$	Seul les objets contenus dans TPE sont exécutables
Abus	Violation de la politique	`conformity`		Détecte les tentatives d'interactions illégales

TAB. A.1 – Synthèse des propriétés de sécurité

Annexe B

Tableaux récapitulatifs des règles de détection

Règle de détection	Propriété	Condition	Graphe				Signature		
			Politique	Dépendance causale	Information	Transition	Int.	Seq.	Cor.
`integrity`	Intégrité des objets	$i = scs_1 \xrightarrow{eo} sco_1$ $is\_write\_like(eo)$	√				√		
		$seq = scs_1 \Rightarrow_{eo} sco_1$ $is\_write\_like(eo)$	√	√(1)		√(2)			√
`int_biba`	Intégrité Biba	$i = scs_1 \xrightarrow{eo} sco_1$ $is\_read\_like(eo)$ $int(scs_1) > int(sco_1)$	√				√		
		$i = scs_1 \xrightarrow{eo} sco_1$ $is\_write\_like(eo)$ $int(scs_1) < int(sco_1)$	√				√		
		$i = scs_1 \xrightarrow{eo} scs_2$ $is\_execute\_like(eo)$ $int(scs_1) < int(scs_2)$	√				√		
`int_subject`	Intégrité des sujets	$i_1 = scs_1 \xrightarrow{eo_1} sco_1$ $i_2 = scs_2 \xrightarrow{eo_2} sco_1$ $\neg commute(eo_1, eo_2)$	√						√
`int_domain`	Intégrité des domaines	$sc_1 \in TCB_{in} \wedge sc_2 \notin TCB_{in}$ $sc_1 \notin TCB_{in} \wedge sc_2 \in TCB_{in}$	√				√		

TAB. B.1 – Synthèse des règles d'intégrité

Règle de détection	Propriété	Condition	Graphe				Signature		
			Politique	Dépendance causale	Information	Transition	Int.	Seq.	Cor.
`confidentiality`	Absence de flux d'informations	$i = sc_1 > sc_2$			√		√		
		$seq = sc_1 \gg sc_2$			√			√	
		$seq = sc_1 \gg_{trans} sc_2$		√(1)	√	√(2)			√
`conf_blp`	Bell&Lapadula	$i = scs_1 \xrightarrow{eo} sco_1$ $is\_read\_like(eo)$ $hab(scs_1) < hab(sco_1)$	√				√		
		$seq = sco_1 \gg sco_2$ $hab(sco_2) < hab(sco_1)$			√			√	
`conf_blp`	Bell&Lapadula restrictive	$i = scs_1 \xrightarrow{eo} sco_1$ $is\_read\_like(eo)$ $hab(scs_1) < hab(sco_1)$	√				√		
		$i = scs_1 \xrightarrow{eo} sco_2$ $is\_append\_like(eo)$ $hab(scs_1) > hab(sco_2)$	√				√		
		$i = scs_1 \xrightarrow{eo} sco_1$ $is\_write\_like(eo)$ $hab(scs_1) != hab(sco_1)$	√				√		
`conf_data`	Cohérence d'accès aux données	$seq = sc_1 \gg_{trans} sc_2$ $sc_2 > sc_1 \notin \mathcal{POL}$		√(1)	√	√(2)			√

TAB. B.2 – Synthèse des règles de confidentialité

Règle de détection	Propriété	Condition	Graphe				Signature		
			Politique	Dépendance causale	Information	Transition	Int.	Seq.	Cor.
`duties_sep`	Séparation de privilèges	$i_1 = scs_1 \xrightarrow{eo_1} sco_1$ $i_2 = scs_1 \xrightarrow{eo_2} sco_1$ $is\_write\_like(eo_1)$ $is\_execute\_like(eo_2)$	√				√		
		$seq_1 = scs_1 \Rightarrow_{eo_1} sco_1$ $seq_2 = scs_1 \Rightarrow_{eo_2} sco_1$ $is\_write\_like(eo_1)$ $is\_execute\_like(eo_2)$	√	√(1)		√(2)			√
`bad_transition`	Absence de changement de contexte	$i = sc_1 \xrightarrow{trans} sc_2$	√				√		
		$seq = scs_1 \Rightarrow_{trans} sc_2$				√		√	
`tpe`	Exécutables de confiance	$i = sc_1 \xrightarrow{eo} sc_2$ $sc_2 \notin TPE$ $is\_execute\_like(eo)$	√				√		
`conformity`	Violation de la politique	$i = scs_1 \xrightarrow{eo} sco_1$ $i \notin \mathcal{POL}$	√				√		

TAB. B.3 – Synthèse des règles d'abus de privilèges

Annexe C

Format de politique neutre

Le listing suivant contient la DTD définissant le format XML des règles contenues dans la politique neutre (cf. 6.1.1).

```
<!-- ************ -->
<!-- * ELEMENTS * -->
<!-- ************ -->

<!-- TOP Level Element -->
<!-- ***************** -->
<!ELEMENT policy ((access_rule)* )>

<!-- Access rules definition -->
<!ELEMENT access_rule (subject, object, perm_list)>
<!ELEMENT subject (#PCDATA)>
<!ELEMENT object (#PCDATA)>
<!ELEMENT perm_list ((perm)* )>
<!ELEMENT perm (#PCDATA)>
```

Annexe D

Exemples de systèmes de contrôle d'accès mandataire

Dans le chapitre 2, nous avons abordé les modèles théoriques de contrôle d'accès. Au-delà de la définition de ces modèles, il est essentiel de disposer d'implantations fiables de ceux-ci sur les systèmes d'exploitation courants pour pouvoir réaliser pleinement leurs objectifs.

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

D.1 Security Enhanced Linux

SELinux est un projet lancé par la NSA, aujourd'hui totalement intégré au noyau Linux [Morris 2004], à l'origine du concept de Linux Security Modules ou LSM [Wright *et al.* 2002]. L'objectif de ce projet est d'implanter un contrôle d'accès de type MAC, robuste et finement configurable, au moyen de l'architecture LSM [Smalley *et al.* 2001, Loscocco et Smalley 2001]. SELinux est issu des travaux portant sur la sécurisation du système d'exploitation DTOS [Secure Computing Corporation 1997].

D.1.1 Linux Security Modules

Le projet LSM intègre dans le noyau des points d'ancrage au niveau des appels système afin de rendre possible l'implantation de nouveaux modèles de contrôle d'accès de façon modulaire sans nécessité de modification du noyau. Dans le but de fournir des mécanismes de contrôle d'accès génériques pour le noyau Linux, le projet LSM relève de deux objectifs précis : l'intégration des nouveaux points d'ancrage dans le code source du noyau (*Security Hooks*), et le support de l'empilement des modules de sécurité :

– Des points de contrôle (*hooks*) sont ajoutés dans les fonctions associées aux appels système, qui constituent les points d'entrée du noyau pour la manipulation des entités nommées du système. Certains sont utilisés pour créer et modifier les champs concernant la sécurité, et les autres

(la majorité) sont responsables du contrôle d'accès. Normalement, ces points de contrôle interviennent après les vérifications liées au contrôle d'accès standard du noyau Linux, ainsi les permissions normales s'appliquent toujours. En conséquence, les modules de sécurité implantés via LSM sont des modules *restrictifs*, ils ne peuvent qu'interdire des actions qui seraient autorisées par Linux.

– Les modules LSM peuvent être "empilés", i.e. les fonctions de sécurité de plusieurs modules peuvent être utilisées en même temps. Cependant, ce mécanisme comporte une restriction majeure. Le premier module chargé sera considéré comme "primaire", et responsable du chargement d'un module secondaire. Chaque module est ainsi responsable du chargement d'un autre module afin de former une chaîne de modules. Or chaque module est responsable de la prise en compte des décisions d'accès produites par son successeur. En conclusion, aucune garantie ne peut être fournie quant à l'application de la politique de sécurité d'un module qui ne serait le premier.

D.1.2 Architecture Flask

Cette architecture [Spencer *et al.* 1999], proposée pour SELinux, repose sur le principe de séparation de la prise de décision (*decision-making*) et d'application (*policy enforcement*) d'une politique de sécurité. Cette architecture repose sur deux composants essentiels : un serveur de sécurité et un module LSM. Le *Security Server*, ou serveur de sécurité, équivalent au *Reference Monitor* d' [Anderson 1972], est responsable de la partie décision. Il contient donc la politique de sécurité du système. Un module LSM, responsable de la partie application, fournit une implantation de l'ensemble des points de contrôle.

SELinux fournit également un système de cache entre ces deux parties, appelé Access Vector Cache (AVC). Grâce à ce système, un vecteur de permissions d'accès entre un sujet et un objet n'est calculé qu'une seule fois et accessible ensuite directement via le cache, améliorant ainsi nettement les performances de SELinux.

D.1.3 Politique de sécurité

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

Dans le modèle de SELinux, un contexte de sécurité est composé d'au moins trois attributs :

– L'$identite$ est liée aux UID standards de Linux, mais diffère dans le sens où il s'agit d'identifiants spécifiques à SELinux. Alors que l'UID d'un utilisateur peut changer au cours de ses actions, par exemple en exécutant des programmes ayant le bit *setuid* activé, l'identité SELinux n'est jamais modifiée et pointe toujours vers l'utilisateur d'origine [1] (celui qui a ouvert une session sur le système d'exploitation).

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

[1] Seules les applications permettant de s'authentifier sur le système tel que *ssh* ou *login* ont la possibilité de changer l'attribut identité des sujets qu'ils créent.

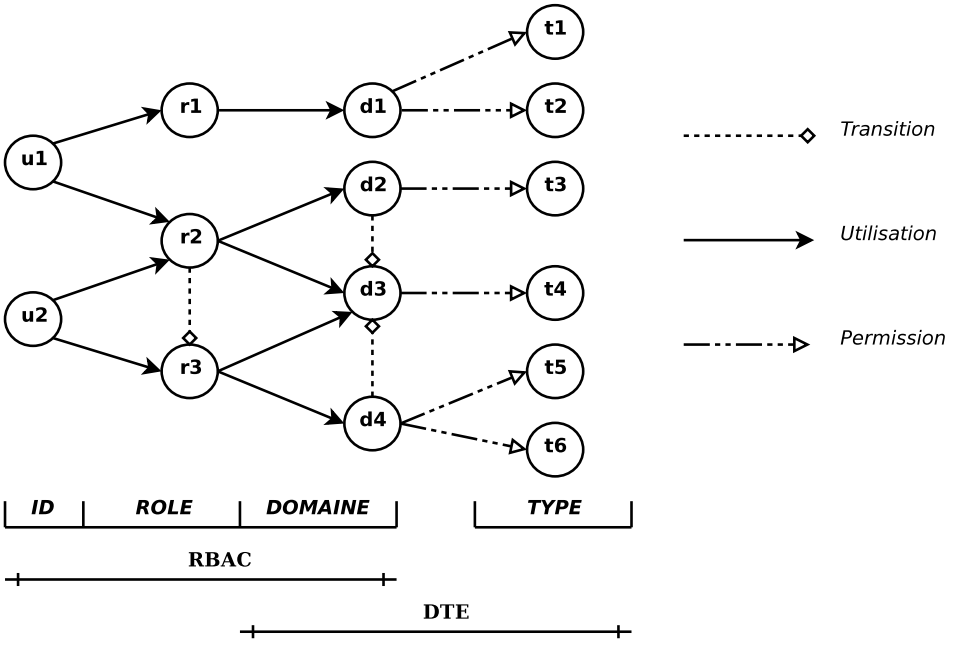


FIG. D.1 – Intégration de RBAC et DTE dans SELinux

peut posséder qu'un seul rôle à la fois. De plus, un utilisateur peut, s'il en a le droit, changer de rôle pendant une même session, moyennant une authentification.

– L'attribut $type$ prend son sens dans le modèle DTE, qui est au cœur de la définition de la politique de sécurité SELinux.

Le format des contextes de sécurité est alors le suivant : `<ID>:<role>:<type>` Cette approche utilise les modèles RBAC et DTE pour respectivement gérer l'attribution des droits aux identités via la définition de rôles et la définition des droits via des permissions par des domaines sur des types. Notons que, dans SELinux, il n'y a pas de notion de domaine ; en effet un domaine n'est qu'un type spécifique associé à un processus.

Une *interaction* légale est une action autorisée entre deux contextes de sécurité *source* et *cible*. Une action autorisée se traduit par une *permission* (lire, écrire, exécuter, etc.) valide sur une *classe* (fichier, répertoire, socket, etc.). Une politique de sécurité pour SELinux définit ainsi un ensemble d'interactions légales.

La figure D.1 contient un exemple de représentation de politique SELinux. Cette politique est composée de deux identités, trois rôles, quatre domaines et six types. Chaque identité peut accéder à deux rôles différents et le rôle r_2 peut transiter vers le rôle r_3. Ainsi u_2 peut passer du rôle r_2 au rôle r_3. Chaque rôle a accès à un ou deux domaines, chaque domaine possède des permissions sur un ou deux types. De plus, les domaines d_2 ou d_4 peuvent transiter vers le domaine d_3, c'est à dire obtenir les privilèges liés à ce domaine. Le triplet `<ID>:<role>:<domaine>` correspond ainsi au modèle RBAC et le couple `<domaine>:<type>` au modèle DTE.

D.1.4 Décision d'accès

Une décision d'accès est le résultat, fourni par le mécanisme de contrôle d'accès, lors d'une tentative d'interaction réalisée par un processus. Le *Security Server*, contenant la politique SELinux,

soit la politique de sécurité, est responsable de ces décisions d'accès. Les informations utilisées pour calculer les permissions d'accès sont les *contextes de sécurité* source et cible, et la classe de la cible. Le résultat est un *vecteur de permissions*, contenant des booléens pour tous les types d'accès associés à la classe de la cible. Les décisions sont de deux types : décisions d'accès et décisions de transition. Les décisions d'accès concernent les tentatives d'accès à un objet, pour accorder ou refuser l'accès. Les décisions de transition ont lieu lorsqu'une nouvelle entité est créée dans le système ou qu'un sujet demande un changement de contexte, afin de calculer son contexte de sécurité. Par exemple, ces calculs ont lieu lorsqu'un nouveau processus est exécuté, ou qu'un nouveau fichier est créé.

D.2 grsecurity

GRSECURITY [Spengler 2002] est un projet débuté en 2001, qui a pour principal objectif de renforcer la sécurité du noyau Linux. Son développement n'est pas intégré au noyau, GRSECURITY se présente alors sous forme d'un patch pour le code source de Linux. Il a pour objectif de fournir trois types de protection :

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

D.2.1 Confinement

GRSECURITY fournit trois mécanismes de confinement : la notion de répertoire d'application de confiance (TPE pour *Trusted Path Execution*), un système d'ACL et un implantation de RBAC.

Le but de TPE est d'éviter que les utilisateurs exécutent des programmes auxquels l'administrateur ne fait pas confiance. Le principe de fonctionnement de TPE est de n'autoriser l'exécution d'applications que lorsqu'elles sont contenues dans un dossier appartenant à `root`, et non modifiables par les utilisateurs. C'est par exemple le cas des dossiers `/bin` et `/usr/bin`.

Le système d'ACL de GRSECURITY permet de restreindre les actions des sujets sur le système. Les sujets sont identifiés par le chemin d'un fichier exécutable, et pour chaque sujet on peut définir des règles. Celles-ci spécifient les droits d'accès sur le système de fichiers, sur les connexions réseau (TCP et UDP), sur les *capabilities POSIX* [2] et sur la quantité de ressources système utilisable (mémoire, nombre de fichiers ouverts, etc.). Il est possible de définir des ACL soit sur des fichiers précis, soit sur des dossiers et ceci de manière récursive. En ce qui concerne l'héritage des droits, lors de la création d'un objet par un processus, l'objet hérite par défaut des droits de son père (répertoire ou processus). Par exemple, en mettant une ACL sur un dossier, celle-ci sera héritée par tous les fichiers qu'il contient, et par les sous-dossiers. Néanmoins, une directive spécifique permet de briser cet héritage. Quand elle est spécifiée lors de la définition d'une règle, seuls les droits définis pour ce sujet particulier seront appliqués.

Les fonctionnalités RBAC de GRSECURITY exploitent la définition de différents ensembles de règles associés aux utilisateurs suivant leur rôle. Cette approche permet de séparer les privilèges d'un

[2] Les *capabilities POSIX* définissent un système de privilèges utilisé sous GNU/Linux. Linux propose un mécanisme (encore incomplet) de capacités, qui scinde les privilèges traditionnellement associés au Super-utilisateur en unités distinctes que l'on peut activer ou inhiber individuellement.

même utilisateur suivant les différents rôles qui lui sont attribués. De plus, un utilisateur peut, s'il en a le droit, changer de rôle pendant une même session, moyennant une authentification.

D.2.2 Prévention

GRSECURITY introduit dans le noyau Linux plusieurs mécanismes de prévention dont l'objectif est très différent de celui visé par le confinement. En effet, le confinement réduit l'impact d'une attaque réussie, alors que la prévention rend plus difficile l'exploitation des vulnérabilités connues, habituellement utilisées par un attaquant. Les mécanismes de prévention introduits dans GRSECURITY sont :

- PaX a pour objectif de rendre certaines pages mémoire non exécutables, notamment la pile et le tas des processus, empêchant ainsi les attaques par dépassement de tampon (*Buffer Overflow*) ;
- ASLR (*Address Space Layout Randomization*) introduit de l'aléa dans le choix des adresses mémoire des processus, notamment au niveau de la pile et des librairies, empêchant ainsi l'utilisation d'adresses mémoire connues lors de l'exécution de code en mémoire.
- Enfin, GRSECURITY peut rendre aléatoire le choix de certains identifiants du système, notamment les identifiants de paquets IP, de processus,, etc.

En outre, GRSECURITY apporte des renforcements à l'appel système `chroot()`, normalement utilisé pour changer la racine du système de fichiers pour un processus particulier afin de séparer son exécution du reste du système. En "emprisonnant" ainsi chaque service critique d'un système, cette approche permet normalement de limiter les dégâts en cas de corruption d'un des services. Mais dans l'implantation actuelle du noyau Linux, il existe plusieurs manières connues [Spengler 2002] permettant de "s'évader" de la nouvelle racine. GRSECURITY apporte donc des restrictions supplémentaires au noyau afin de prévenir ce problème.

D.2.3 Détection

GRSECURITY a de nombreuses possibilités d'audit d'évènements du système, autorisant la surveillance des actions effectuées sur le système d'exploitation, et la détection d'activités suspectes. D'une part, le système d'ACL permet de définir, en parallèle des règles de *contrôle d'accès*, des règles d'*audit*. Il est ainsi possible de générer une trace pour tout droit d'accès pris en considération par GRSECURITY (lecture, écriture, etc.) D'autre part, il est possible d'auditer :

- les appels système effectués par les processus ;
- les évènements liés à PaX, notamment les tentatives d'exécution dans la pile d'un processus, souvent symptomatiques d'une attaque par *Buffer Overflow* ;
- et beaucoup d'autres évènements spécifiques à la sécurité dans le noyau Linux.

Formalisation et garantie de propriétés de sécurité système : application à la détection d'intrusions

Résumé Dans cette thèse, nous nous intéressons à la garantie des propriétés d'intégrité et de confidentialité d'un système d'information. Nous proposons tout d'abord un langage de description des activités système servant de base à la définition d'un ensemble de propriétés de sécurité. Ce langage repose sur une notion de dépendance causale entre appels système et sur des opérateurs de corrélation. Grâce à ce langage, nous pouvons définir toutes les propriétés de sécurité système classiquement rencontrées dans la littérature, étendre ces propriétés et en proposer de nouvelles. Afin de garantir le respect de ces propriétés, une implantation de ce langage est présentée. Nous prouvons que cette implantation capture toutes les dépendances perceptibles par un système. Cette méthode permet ainsi d'énumérer l'ensemble des violations possibles des propriétés modélisables par notre langage. Notre solution exploite la définition d'une politique de contrôle d'accès afin de calculer différents graphes. Ces graphes contiennent les terminaux du langage et permettent de garantir le respect des propriétés exprimables. Nous utilisons alors cette méthode pour fournir un système de détection d'intrusion qui détecte les violations effectives des propriétés. L'outil peut réutiliser les politiques de contrôle d'accès disponibles pour différents systèmes cibles DAC (Windows, Linux) ou MAC tels que SELinux et GRSECURITY. Cet outil a été expérimenté sur un pot de miel durant plusieurs mois et permet de détecter les violations des propriétés souhaitées. **Mots-clés** Sécurité système, Propriété de sécurité, Détection d'intrusion, Contrôle d'accès

Formalization and guaranty of system security properties : application to the detection of intrusions.

Abstract In this thesis, we are interested in the guaranty of the properties of integrity and confidentiality of an information system. We first of all propose a language of description of the system activities used as a basis for the definition of a set of security properties. This language rests on a notion of causal dependence between system calls and on operators of correlation. Thanks to this language, we can define all the system security properties classically met in the literature, extend these properties and propose news of them. In order to guaranty the respect of these properties, an implementation of this language is presented. We prove that this implementation captures all the dependences perceptible by a system. This method thus makes it possible to enumerate the whole of the possible violations of the properties expressed by our language. Our solution exploits the definition of an access control policy in order to compute various graphs. These graphs contain the terminals of the language and make it possible to guaranty the respect of the properties. We then use this method to provide a system of detection of intrusions which detects the effective violations of the properties. The tool can re-use the access control policies available for various target systems DAC (Windows, Linux) or MAC such as SELinux and grsecurity. This tool was tested on a honeypot during several months and makes it possible to detect the violations of the desired properties.

Keywords
System security, Security property, Intrusion detection, Access control

Discipline : Informatique
Laboratoire d'Informatique Fondamentale d'Orléans
Bâtiment IIIA
Rue Léonard de Vinci
B.P. 6759
F-45067 ORLEANS Cedex 2