Formalisation de propriétés de sécurité pour la protection des systèmes d'exploitation

Jonathan Rouzaud-Cornabas

UNIVERSITÉ D'ORLÉANS

ÉCOLE DOCTORALE SCIENCES ET TECHNOLOGIES

LABORATOIRE : LIFO

THÈSE présentée par :

Jonathan ROUZAUD-CORNABAS

soutenue le : **02 Décembre 2010**

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

Discipline : **Informatique**

Formalisation de propriétés de sécurité pour la protection des systèmes d'exploitation

THÈSE DIRIGÉE PAR :

Christian TOINARD	Professeur, LIFO (Directeur)
Patrice CLEMENTE	Maître de Conférences, LIFO (Co-Responsable)

RAPPORTEURS :

Francine KRIEF	Professeur, LaBRI
Jean-Yves MARION	Professeur, LORIA

JURY :

Mathieu BLANC	Docteur, CEA-DAM
Patrice CLEMENTE	Maître de Conférences, ENSI de Bourges
Francine KRIEF	Professeur, ENSEIRB
Sébastien LIMET	Professeur, Université d'Orléans
Jean-Yves MARION	Professeur, Ecole des Mines de Nancy
Christian TOINARD	Professeur, ENSI de Bourges

Remerciements

Je remercie Mme. Francine Krief et M. Jean-Yves Marion 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. Laurent Clevyet M. Sébastien Limet.

Je tiens à remercier mon directeur de thèse, M. Christian Toinard et M. Patrice Clemente, mon co-responsable, pour leur soutien et leur 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 également à remercier M. Jeremy Briffaut pour son aide et son soutien tout au long de ma thèse. Je tiens à remercier tous les membres de l'équipe Sécurité et Distribution des Système, M. Pascal Berthome, M. Jean-François Lalande et M. Yacine Zémali, 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 Stéphanie Ollier pour ses nombreuses relectures et son soutien tout au long de ma thèse.

Table des matières

Table des figures

Liste des Algorithmes

Liste des tableaux

Chapitre 1

Introduction

L'actualité montre tous les jours la grande vulnérabilité des systèmes d'information. Nous pouvons donner trois exemples dans des cadres d'usage différents. Les vols d'information sur les postes de travail des internautes sont quotidiens. Tout d'abord, des arrestations ont lieu régulièrement suite à la mise en place de larges réseaux de vol d'information (botnet) qui se déploient sur les postes client (grand public comme entreprises et gouvernements). En pratique, les utilisateurs sont amenés, par ingénieurie sociale, à exécuter un programme malicieux (malware) qui récupère ensuite les informations personnelles ou confidentielles.

Le second exemple concerne les scénarios d'attaque passant par le poste client pour accéder à l'intranet d'une entreprise. En effet, les points d'entrée sur les intranets sont de plus en plus renforcés. Le poste client devient donc le maillon faible de la sécurité globale des entreprises. C'est ainsi que l'attaque nommée Opération Aurora visant Google (et d'autres grandes sociétés américaines) a permis via une faille côté client d'accéder aux boites mails d'activistes Chinois via l'intranet de Google.

Ces deux exemples illustrent l'importance des problèmes de sécurité coté client. En effet, un même poste de travail servira à des usages très variés. Il est donc nécessaire de pouvoir définir différents domaines, par exemple pour un navigateur web, qui ont chacun des objectifs de sécurité dédiés. De plus, il est important de contrôler les flux entre ces domaines afin de pouvoir garantir les objectifs de sécurité associés à ces domaines.

Le troisième exemple illustre la vulnérabilité des serveurs. Les cas d'attaque de serveurs web sont quotidiens. Ces serveurs sont particulièrement vulnérables puisqu'ils permettent d'exécuter des applications tierces qui sont utilisées pour exécuter du code malveillant sur le serveur. Le risque sur ces serveurs est d'autant plus important qu'ils sont utilisés pour héberger des applications très diverses avec des objectifs incompatibles. On voit donc la nécessité d'avoir des mécanismes de protection sur ces serveurs qui puissent gérer différents domaines avec des politiques spécifiques, tout en garantissant l'absence de flux d'information entre ces domaines.

Un des maillons critiques des systèmes d'information est le système d'exploitation. Actuellement, les systèmes d'exploitation sont les maillons faibles. En effet, ceux-ci sont actuellement incapables de fournir des moyens de protection efficaces pour eux-même mais surtout pour les applications qu'ils hébergent. En pratique, il n'est pas possible de définir les objectifs de sécurité que l'on souhaite pour les applications afin que le système d'exploitation les garantisse. Actuellement, le modèle de protection prédominant est le modèle discrétionnaire où les utilisateurs finaux définissent les droits sur leurs ressources. Avec ce modèle, il est impossible de garantir des objectifs de sécurité. Les approches mandataires visent à permettre la définition d'objectifs de sécurité afin que le système d'exploitation puisse les garantir. Notre état de l'art montrera que les approches proposées répondent très partiellement aux problèmes. D'une part, elles ne traitent qu'un sous-ensemble des objectifs de sécurité requis. D'autre part, elles abordent les problèmes à un niveau qui n'est pas

adapté à la protection système. Soit elles envisagent de colorer toutes les données afin de contrôler les flux d'information. Soit elles définissent des politiques de protection très fines qui traitent les permissions des processus sur les ressources. Dans les deux cas, cela conduit à une complexité importante et cela ne permet pas de traiter efficacement les flux d'information transitifs.

Pour nous, les approches orientées analyses virales ou comportementales sont inadaptées à la protection système. Elles sont actuellement utilisées pour la protection des systèmes d'exploitation parce que ceux-ci n'offrent pas de protection orientée objectifs de sécurité. Les travaux dans ce domaine concernent essentiellement l'analyse des vulnérabilités. Dans de très rares cas, elles analysent les flux d'information transitifs. Cependant, ces travaux s'intéressent peu aux scénarios d'attaque et à des propriétés génériques permettant de garantir des objectifs de sécurité. Ces approches sont complémentaires de la notre. Ils sortent du cadre de cette étude mais nous montrerons des perspectives d'usage de nos résultats dans ces domaines.

Actuellement, il n'existe pas de solutions suffisament ouverte et extensible pour permettre à différents domaines de partager la même infrastructure qu'elle soit cliente ou serveur tout en contrôlant efficacement les flux d'information à l'intérieur et entre les domaines. C'est pourquoi l'Agence Nationale de la Recherche (ANR) a lancé en 2008 un programme de recherche appelé Défi Sécurité [ANR, 2008] (ANR SEC&SI). L'objectif de ce programme était de fournir un poste de travail sécurisé pour l'internaute qui supporte différents domaines d'usages et de criticité (Web, eCommerce, Impôts, Réseaux Sociaux, etc). Cette thèse s'est déroulée en parallèle du projet SPACLik [Briffaut et al., 2009b], vainqueur du Défi Sécurité. Elle a, en partie, contribuée à ce projet. Elle traite certaines difficultés à mettre en oeuvre, en grandeur réelle, un système de protection en profondeur. En effet, un des enseignements du Défi Sécurité est la nécessité d'avoir une protection en profondeur, c'est-à-dire, opérant à tous les niveaux d'un système d'exploitation (noyau, réseau, interface utilisateur, application). La difficulté d'une approche en profondeur est de pouvoir associer aux différents domaines d'usages des politiques de protection variables. On a donc besoin d'une protection dynamique. La dynamicité opère à deux niveaux. Au niveau haut, il s'agit de définir dynamiquement les objectifs de sécurité pour les différents domaines. Au niveau bas, le système d'exploitation doit garantir dynamiquement ces objectifs de sécurité. Ces deux niveaux sont liés. En effet, la garantie dynamique des objectifs de sécurité permet l'expression d'objectifs dynamiques.

Cette thèse propose de faciliter la définition d'objectifs de sécurité. Pour cela, nous proposons un langage qui permet de formaliser les propriétés de sécurité dont on demandera la garantie au système d'exploitation. Ce sont ces propriétés de sécurité qui formalisent les objectifs requis. Ce langage repose sur un ensemble d'opérateurs qui permettent de prendre en compte facilement les flux d'information qu'ils soient directs ou indirects (transitifs). Grâce à ce langage, nous pouvons formaliser aisément l'ensemble des propriétés de protection classiques de la littérature. Ce langage est très ouvert. Il permet de définir de nouvelles propriétés de sécurité. Ainsi, nous proposons une méthode extensible pour décrire les objectifs de sécurité requis. Nous définissons un moyen de compiler ce langage pour analyser les appels systèmes effectués par les processus utilisateurs. La méthode de compilation repose sur la construction d'un graphe de flux d'information. La complexité de ce graphe est polynomiale avec le nombre de type d'entités sur le système. En pratique, sa taille reste faible. La compilation propose une mise en oeuvre de nos opérateurs au moyen de fonctions qui analyse notre graphe de flux d'information. La complexité à la fois de ces fonctions mais aussi du traitement de leurs résultats est polynomiale. En pratique, leur complexité reste faible ce qui permet d'exécuter efficacement l'analyse des propriétés de sécurité. Nous proposons une implantation de ce langage pour la protection d'un système d'exploitation. L'analyse des propriétés de sécurité offre alors une méthode de protection mandataire qui garantit dynamiquement les propriétés requises. Une expérimentation à large échelle a été réalisée sur deux types (client et serveur) de pot de miel à haute interaction. Le pot de miel client permet d'analyser un

large ensemble de propriétés de sécurité associées à un navigateur Web. Afin d'analyser un grand nombre de scénarios d'attaques, les propriétés de sécurité n'empêchent pas le déroulement de ces scénarios. Cependant, nous montrons que les propriétés utilisées pour l'analyse peuvent être utilisées en terme de protection. Le pot de miel serveur permet de tester l'efficacité de notre méthode de protection sur un ensemble de services. Il montre que les machines sans cette protection sont affectées tandis que celles protégées par notre approche mandataire ne sont pas compromises.

La suite du document s'articule en six chapitres. Le chapitre 2 présente l'ensemble des travaux de la littérature dans le domaine de la protection des systèmes. Il conclut sur la nécessité d'un langage permettant de formaliser un large spectre de propriétés de sécurité. Le chapitre 3 propose une modélisation abstraite d'un système d'exploitation ainsi qu'un langage constitué d'opérateurs manipulant les entités abstraites du système. Le chapitre 4 utilise ce langage pour formaliser un large ensemble de propriétés de sécurité. Nous montrons son pouvoir d'expression pour formaliser un nouveau modèle de protection dynamique. Le chapitre 5 définit notre méthode pour compiler dynamiquement les propriétés de sécurité. Il propose une mise en oeuvre reposant sur un graphe de flux d'information et présente la complexité du langage et des propriétés. Le chapitre 6 présente la mise en oeuvre d'un système mandataire reposant sur notre langage d'expression de propriétés. Il développe les expérimentations et les résultats obtenus sur nos pots de miel. Il montre l'efficacité non seulement en terme de performances mais surtout en terme de prévention de scénarios d'attaque à la fois connus et inconnus.

Chapitre 2

État de l'art

Les problématiques de *confidentialité* et d'*intégrité* font partie intégrante des critères d'évaluation de la sécurité. Vis-à-vis des critères d'évaluation de la sécurité des systèmes informatiques [TCSEC, 1985] (*Trusted Computer System Evaluation Criteria* - TSEC), le Ministère de la Défense américain (*Department of Defense* - DoD) a édité un livre (*Orange Book*) mettant en avant la nécessité de politiques de sécurité. Ces politiques doivent pouvoir s'exprimer au moyen de propriétés de confidentialité, d'intégrité et/ou de disponibilité, et ce, même de façon informelle. L'*Orange Book* est toujours une référence pour l'expression des besoins de sécurité des systèmes. Ces critères ont été ensuite repris par un comité de pays européens dans le livre *Information Technology Security Evaluation Criteria* (ITSEC) [ITSEC, 1991]. Ce chapitre présente les travaux existants au regard des objectifs de cette thèse, c'est-à-dire aux travaux visant à ce qu'un système d'exploitation garantisse efficacement des propriétés de sécurité. Pour cela, il est nécessaire de pouvoir exprimer facilement les propriétés de sécurité nécessaires. Le formalisme utilisé doit pouvoir s'appliquer sur un système d'exploitation classique tel que Unix. Il doit permettre de formaliser le spectre le plus large des propriétés attendues par les administrateurs et les utilisateurs finaux du système.

Nous allons découper cet état de l'art en trois parties. Tout d'abord, nous allons nous consacrer aux politiques de sécurité système. Cette première partie va détailler les grandes classes de propriétés de sécurité qu'un système peut garantir. Elle montrera que ces classes peuvent être formalisées de différentes façons et qu'elles nécessitent de pouvoir être adaptées, composées ou étendues facilement. Elle conclura par l'absence d'un moyen adapté pour réaliser cette formalisation. La deuxième partie présentera des modèles de protection classiques qui supportent des cas particuliers des propriétés. Cette partie montrera que les travaux dans ce domaine mêlent à la fois formalisation de la propriété et moyens de la supporter. Elle se conclura en montrant qu'aucun de ces modèles n'est suffisament expressif et qu'ils ne peuvent pas être composés ou étendus facilement. Dans la troisième partie, nous montrerons comment les auteurs proposent d'implanter soit une de ces propriétés soit un de ces modèles. Elle montrera là encore que les solutions proposées ne peuvent pas aisément être étendues pour atteindre les objectifs que nous visons. Nous conclurons ce chapitre en explicitant les points à résoudre pour permettre à la fois de formaliser les propriétés requises et de garantir ces propriétés sur un système d'exploitation.

2.1 Politique de sécurité

La définition et l'application d'une *politique de sécurité* représente le cœur de la sécurité d'un système d'information car elle décrit les *objectifs de sécurité*. Ces objectifs peuvent se décliner sous la forme d'un ensemble de propriétés de sécurité. Chaque propriété représente 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

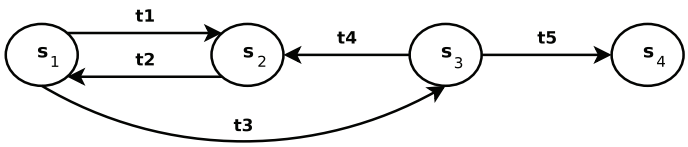

FIGURE 2.1 – Exemple d'automate à états finis représentant les états d'un système.

état non-sûr, autorisant le vol d'informations ou de ressources, la modification d'informations ou la destruction (totale ou partielle) du système. Dans cette section, nous commençons par expliciter en quoi consiste une politique de sécurité puis nous donnons une définition des différentes classes de *propriétés de sécurité* que nous rencontrons dans la littérature.

2.1.1 Politique de sécurité

Une politique de sécurité spécifie ce que l'on entend par "sécurité" du système. Elle définit les objectifs de sécurité que l'on attend du système. Elle peut, par exemple, définir comme objectif de confidentialité que toutes informations issues d'un domaine privilégié (exemple : direction de l'entreprise) ne puissent être lues par un domaine non privilégié (exemple : service de production). Ces objectifs correspondent donc à un ensemble de propriétés qui peuvent être définies de manière informelle sous une forme littéraire ou formelle. Les propriétés de sécurité peuvent être regroupées en trois grandes classes [TCSEC, 1985] : *confidentialité*, *intégrité* et *disponibilité*. La *confidentialité* concerne le contrôle de l'accès à l'information en lecture. L'*intégrité* concerne, elle, les accès en écriture. Quant à la *disponibilité*, elle concerne essentiellement l'accessibilité et les temps de réponse des services. Une propriété peut expliciter les comportements autorisés ou ceux interdits. Il est ainsi possible d'exprimer qu'il est autorisé au domaine privilégié d'accéder à des informations de ce domaine. Il est aussi possible d'interdire l'accès à ces données privilégiées pour un domaine non privilégié. Dans la plupart des cas, les politiques considèrent implicitement que tout ce qui n'est pas autorisé, est interdit. Cependant, dans certains cas, il peut être utile d'expliciter ce qui est interdit. Cette hypothèse privilégiant les autorisations est contradictoire avec l'aspect législatif qui, en général, fixe les interdictions. Donc, il est souvent indispensable de vérifier ou de garantir des interdictions autant que des autorisations. Dans la pratique, il est nécessaire de pouvoir garantir à la fois des interdictions et des autorisations.

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 [TCSEC, 1985]. Dans cette représentation, nous pouvons définir une *politique de sécurité* par :

Définition 2.1.1 (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-autorisés** (ou **non-sûrs**).*

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

Définition 2.1.2 (Système sûr) *Un **système sûr** est un système qui, partant d'un état sûr, ne pourra jamais entrer dans un état non-sûr.*

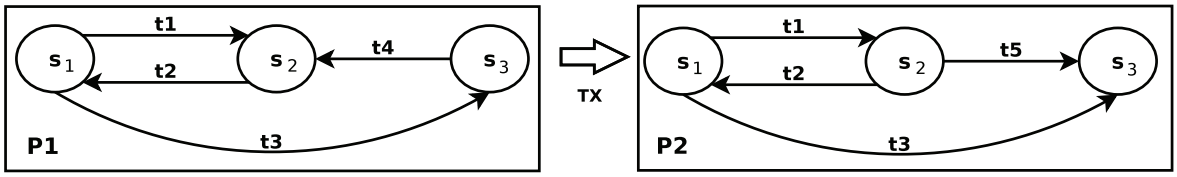

FIGURE 2.2 – Exemple d'automate à état finis dynamiques représentant les états sûrs d'un système

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 à 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 rupture de sécurité ou de propriété :

Définition 2.1.3 (Rupture de sécurité) *Une **rupture de sécurité** apparaît lorsqu'un système entre dans un état non-autorisé.*

Politique de sécurité dynamique

Une politique dynamique est une politique qui évolue avec l'état du système.

Définition 2.1.4 (Politique de sécurité dynamique) *Une politique de sécurité dynamique est une politique de sécurité qui évolue en prenant en compte les états antérieurs du système.*

Considérons l'automate présent dans la figure 2.2 contenant trois états et quatre transitions représentant un système qualifié sûr. Suite à une transition TX, la politique passe de l'état $P1$ à l'état $P2$ qui est aussi un système sûr. Il y a donc évolution de la politique et du système sûr qu'elle décrit au fur et à mesure de l'apparition d'événements sur le système. Certaines approches proposent une évolution encadrée des entités (labels) d'un système. Dans [Foley et al., 1996], les auteurs présentent une approche formelle réalisant des fonctions de changement de labels de sécurité de manière dynamique. Par exemple, si un sujet A utilise un objet B, ce dernier prendra l'étiquette C.

2.1.2 Propriétés de sécurité générales

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

Intégrité

D'une manière générale, l'*intégrité* désigne le fait que les données, lors de leur traitement, de leur conservation ou de leur transmission, ne doivent subir aucune altération ou destruction volontaire ou accidentelle, et conservent un format permettant leur utilisation. 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 que des informations proviennent bien des propriétaires est connue sous le nom d'*intégrité de l'origine*, plus communément appelée *authenticité*. Plus précisément, la *propriété d'intégrité* peut être définie comme suit :

Définition 2.1.5 (Intégrité) *Soit X un ensemble d'entités et soit I de l'information ou une ressource. La propriété d'intégrité de X envers I est respectée si aucun membre de X ne peut modifier I.*

Les membres de X sont généralement définis de manière implicite. Ainsi, les utilisateurs autorisés à modifier une information ou une ressource sont définis de manière explicite (propriétaires de cette information, etc.). Les autres, qui par définition ne doivent pas avoir accès à ce document, forment alors l'ensemble X.

Confidentialité

La *confidentialité* est ainsi définie par [Bishop, 2003] : "elle prévient la divulgation d'information non autorisées". Plus précisément, la *propriété de confidentialité* peut être définie comme suit :

Définition 2.1.6 (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 divulgée à certaines entités, mais peut être divulgée à d'autres.

Disponibilité

La *disponibilité* est à mettre en parallèle avec la fiabilité, et plus spécifiquement, la sûreté de fonctionnement. La disponibilité traite différents modèles de fautes : pannes franches, fautes d'omissions, fautes temporelles et fautes byzantines. Plus précisément, la *propriété de disponibilité* peut être définie comme suit :

Définition 2.1.7 (Disponibilité) *Soit X un ensemble d'entités et soit I une ressource. La propriété de disponibilité de X envers I est respectée si tous les membres de X peuvent accéder à I dans un temps donné.*

Par exemple, la *propriété de disponibilité* peut être garantie pour un site d'achat en ligne dans une durée d'une heure et pour un modèle de fautes donné. Mais cette même propriété de disponibilité est trop faible pour un serveur médical confronté à une possible allergie à un anesthésique, le temps de réponse étant considéré dans ce cas comme trop important. La *propriété de disponibilité* est donc fortement liée au contexte et doit notamment prendre en compte les temps de réponses et les modèles de fautes envisagés. Etant donné qu'il s'agit de problèmes de sûreté et non de sécurité, nous n'aborderons pas cette classe de propriétés par la suite. Cependant, les propriétés d'intégrité et de confidentialité sont souvent considérées comme une condition nécessaire pour la mise en place de propriétés de sûreté de fonctionnement.

2.1.3 Propriétés de sécurité dérivées

Dans l'absolu, un système est parfaitement intègre et confidentiel quand il ne rend aucun service. Dans la pratique, on retrouve le même paradoxe, en effet, garantir des propriétés d'intégrité et de confidentialité sur tout un système revient souvent à que ledit système ne fonctionne plus car trop sécurisé et donc trop contraint. C'est pourquoi il est nécessaire de disposer d'une panoplie suffisament large de propriétés d'intégrité et de confidentialité pour s'adapter aux exigences de sécurité du système tout en lui permettant de fonctionner. Dans la pratique, on trouve donc un grand nombre de cas particuliers de propriété d'intégrité et de confidentialité. Chacun de ces cas particuliers traite un sous-ensemble des besoins d'intégrité ou de confidentialité. Dans cette partie, nous donnons une définition de cinq de ces cas particuliers : 1) le confinement de processus, 2) le principe du moindre privilège, 3) la séparation de privilèges, 4) la non-interférence et 5) l'absence de situation de concurrence.

Confinement de processus

Lampson [Lampson, 1973] caractérise le problème du confinement par :

Définition 2.1.8 (Problème du confinement) *Le problème du confinement concerne la prévention de la divulgation, par un service (ou un processus), d'informations considérées comme confidentielles par les utilisateurs de ce service.*

Lampson définit alors les caractéristiques nécessaires d'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 stocker 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 avec d'autres processus ne peut pas divulguer de l'information. En effet, l'observation d'un processus A par un processus B est considérée comme une communication entre A et B c'est-à-dire qu'un flux d'information allant de A vers B existe. 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 2.1.9 (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 via un canal caché, c'est-à-dire que p peut provoquer un flux d'information transitif vers q. De ce fait, q obtient de l'information de p sans communication directe par seule observation du comportement de p. Finalement, comme l'indique Lampson, la notion de *confinement de processus* est transitive. Si un processus p est considéré comme confiné contre la divulgation d'information,

alors si ce second processus invoque un autre processus r, ce processus doit aussi être confiné sinon il peut divulguer de l'information provenant de p.

De par la difficulté d'implanter l'isolation *totale*, le problème des canaux cachés et la transitivité de la notion de confinement, 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 aux processus privilégiés. On parle alors de *confinement partiel* du système.

Moindre privilège

Ce principe peut ainsi être défini comme suit :

Définition 2.1.10 (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.*

Le principe du *moindre privilège* [Saltzer and 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 minimiser 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.

Séparation de privilèges

L'acceptation la plus répandue [Clark and Wilson, 1987, Sandhu, 1990] correspond à :

Définition 2.1.11 (Séparation de privilèges) *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 peut être définie par :

Définition 2.1.12 (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 ne sont pas affectées par les actions effectuées par X.*

Le principe de la *non-interférence* [Focardi and 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. Dans le cas d'un système, les processus de haut niveau (processus système) et de bas niveau (utilisateur) ne doivent pas interférer. Cela permet d'endiguer certaines attaques des utilisateurs sur les services système.

Situation de Concurrence

Les auteurs dans [Netzer and Miller, 1992] présentent une définition générale :

Définition 2.1.13 (Situation de concurrence) *Une situation de concurrence arrive quand il y a un ordonnancement imprévisible entre les accès de deux entités X et Y à une ressource partagée I avec une X des deux entités provoquant une opération d'écriture sur la ressource partagée I entre deux accès de la seconde entité Y sur cette ressource.*

Une situation de concurrence, appellée Race Condition (RC) en anglais, correspond à une absence d'ordonnancement entre les accès provoqués par deux utilisateurs et/ou processus vers une ressource avec au moins un des deux utilisateurs/processus modifiant cette ressource partagée. L'état du système dépend de l'ordre d'exécution et devient donc imprévisible.

L'absence de situation de concurrence est liée au principe de non-interférence. En effet, il porte également sur l'absence de modification, par un processus, de la vision des données ou de l'état du système par un second processus.

2.1.4 Discussion

Les propriétés de sécurité présentées dans cette section présentent un niveau de formalisme qui n'est pas suffisant pour être utilisable, en pratique, dans le cadre d'un système d'exploitation. Par exemple, les propriétés d'intégrité et de confidentialité ne sont pas suffisament précises dans de nombreux cas et mènent à une sur ou sous-approximation des objectifs de sécurité. Les propriétés dérivées permettent d'outrepasser ce problème mais elles ne traitent d'une part que de cas particuliers et, d'autre part, ne résolvent en rien la difficulté de mise en oeuvre sur un système réel. On voit donc qu'il y a besoin de pouvoir formaliser avec le bon niveau de granularité les propriétés requises. Ce formalisme doit être suffisament général pour prendre en compte tous ces cas particuliers mais aussi pour définir des propriétés plus précises, plus complètes ou plus étendues.

2.2 Modèles de Protection

Dans cette section, nous allons détailler les différents modèles de protection de la littérature. Chacun de ces modèles traite d'une propriété de sécurité ou d'un cas particulier. Un modèle inclut généralement un moyen de formalisation de la propriété et l'algorithmique nécessaire à sa mise en oeuvre. D'une part, nous verrons que les formalisations adoptées ne sont pas forcément adaptées à la protection d'un système d'exploitation. D'autre part, l'algorithmique n'est pas toujours suffisante pour réaliser une mise en oeuvre du modèle. Et finalement, les modèles sont incompatibles au niveau de leur formalisation, ce qui empêche de composer les propriétés de sécurité associées.

2.2.1 Contrôle d'accès discrétionnaire

Le contrôle d'accès *discrétionnaire* (DAC) est le plus ancien modèle de protection. C'est toujours le système de protection majoritairement utilisé dans les systèmes d'exploitation actuels (Unix, MS Windows, Mac OSX, ...). La caractéristique principale est que le propriétaire des ressources définit les droits d'accès sur ces dernières. Les droits d'accès sont donc à la *discrétion* du propriétaire. Par exemple, sous Unix, le propriétaire d'un fichier définit les droits de *lecture*, d'*écriture* et d'*exécution* de ses fichiers pour les différents utilisateurs du système (lui-même, les membres de son groupe et les autres).

Modèle de Lampson

Le premier qui a formalisé le modèle discrétionnaire est Lampson en 1971, [Lampson, 1971] en utilisant des notions qu'il avait introduit précédemment [Lampson, 1969]. 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 (c'est-à-dire les sujets) sur les lignes, et en colonnes l'ensemble X des objets (incluant les domaines). Il pose ensuite deux définitions :

Définition 2.2.1 (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 2.2.2 (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 $(d, A[d, o]), \forall d \in D$.*

Le modèle de Lampson a été progressivement amélioré pour donner naissance à d'autres modèles tels que le modèle HRU [Harrison et al., 1976].

Modèle HRU

L'apport principal du modèle HRU [Harrison et al., 1976] concerne la modification de matrices de contrôle d'accès afin de modifier la liste des sujets et des objets mais aussi des permissions.

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.

HRU s'intéresse aux problèmes de sûreté d'un système de protection.

- Les auteurs ont prouvé que lorsque les commandes ne contiennent qu'une seule action élémentaire, le problème de sûreté est décidable. Toutefois, la vérification de la matrice de protection est *NP-Complet*. Cependant cette hypothèse est irréaliste dans le cas de système réel.
- Dans le cas général d'un système d'exploitation, le problème de vérification d'un système de protection discrétionnaire est indécidable. Ce résultat est essentiel car il prouve qu'il est impossible de garantir des propriétés de sécurité sur un système discrétionnaire.

Modèle TAM

Le modèle TAM (Typed Access Matrix), introduit par [Sandhu, 1992b], 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. Pour R.S. Sandhu, l'ensemble de ces types est fini, il n'est pas possible d'en créer ou d'en supprimer.

R.S. 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 and 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.

Discussion

Le point important est que aussi bien le modèle HRU [Harrison et al., 1976] que le modèle MTAM [Sandhu, 1992b] établissent l'impossibilité de garantir des propriétés de sécurité sur un système réel. En effet, ce n'est qu'au prix de simplifications qui ne correspondent pas à la réalité des systèmes d'exploitation que l'on peut arriver à garantir des propriétés sur un système discrétionnaire.

La notion de *rôle* introduit par [Ferraiolo and Kuhn, 1992] ne change en rien le résultat d'impossibilité. Elle permet juste de simplifier l'écriture des règles de protection.

Diverses études [TCSEC, 1985, Ferraiolo and 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 un 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.

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 est nécessaire d'utiliser un modèle MAC permettant de restreindre les droits accordés au *super-utilisateur* et d'éviter que les utilisateurs ne fixent eux-mêmes les permissions sur leurs fichiers.

2.2.2 Contrôle d'accès mandataire

Le contrôle d'accès mandataire (MAC) impose une politique non modifiable par les utilisateurs finaux. Pour contrôler les accès entre sujets et objets, Anderson [Anderson, 1980] propose d'utiliser un *Moniteur de Référence* (*Reference Monitor*). Ce concept est essentiel à la définition du MAC. Cette notion initialement associée au modèle défini par Bell et LaPadula, a vu sa signification évoluer et représente aujourd'hui tout mécanisme qui place la définition et la gestion de la politique hors d'atteinte des utilisateurs concernés. Par ailleurs, nous ne traiterons pas des modèles à base de rôles (RBAC pour *Role-Based Access Control*) car ils n'apportent rien en terme de propriété de sécurité. Ces modèles permettent uniquement de factoriser des règles et n'offrent pas une expressivité supplémentaire.

Les différents modèles, présentés dans cette section, sont des modèles mandataires supportant une propriété de sécurité ou un cas particulier.

Modèle Bell et LaPadula

Le modèle Bell-La Padula, établi par [Bell and La Padula, 1973], plus couramment appelé BLP, concerne la confidentialité des données du monde militaire. Il a ainsi pour objectif de prévenir toute divulgation d'informations.

Ce modèle introduit la notion de *label* (ou étiquette) 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 (c'est-à-dire 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.

Ce modèle repose sur le respect de deux 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 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 [1], 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.

Ce qui peut se traduire par :

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

1. Dans ce cas, les canaux cachés représentent les parties du système non-contrôlables par le mécanisme de contrôle d'accès, par exemple un tampon mémoire, un périphérique de type *raw*, etc.

L'article de Bell et La Padula [Bell and La Padula, 1973] décrit l'intégration de ce modèle dans MULTICS. On 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 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].

Modèle Biba

Le modèle dit "Biba" [Biba, 1975] vise à garantir l'*intégrité*. Il s'agit d'un 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 inférieure ou égale, 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 supérieure ou égale.

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 suivi 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.

Modéle de la Muraille de Chine

La *Muraille de Chine* permet de garantir qu'une opération malveillante de type *déli d'initié* ("Inside Dealing") ne sera pas possible au sein d'un système d'information. Il est orienté vers le domaine de la finance où ce modèle est aussi important que BLP pour les militaires.

La *Muraille de Chine* est une propriété de sécurité pouvant à la fois se classer dans la catégorie *intégrité* ou *confidentialité* selon la nature du conflit d'intérêt. Cette propriété de sécurité commence par un ensemble de trois définitions qui expriment la manière dont sont séparées les informations.

- Un objet contient une partie de l'information d'une unique entreprise.
- L'ensemble de données d'une entreprise (CD) contient les objets d'une entreprise unique.
- Une classe de conflit d'intérêts (COI) contient les CDs des entreprises en concurrence.

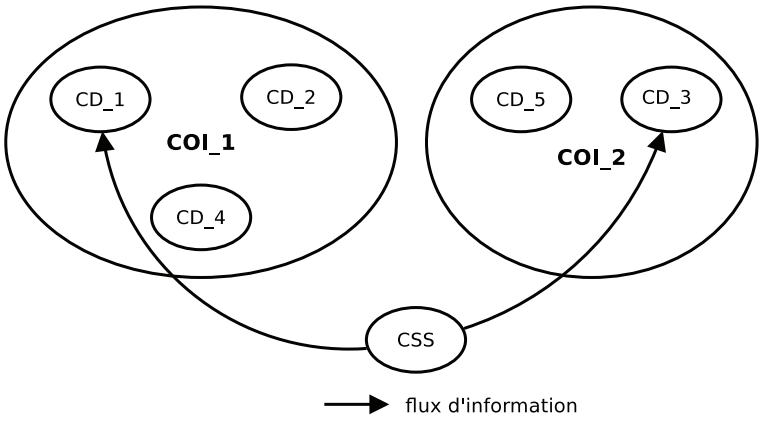

FIGURE 2.3 – Représentation de la séparation des accés

Considérons cinq entreprises (CD_1 à CD_5) séparées en deux groupes de conflit d'intérêt (COI_1 et COI_2), voir la figure 2.3. Un utilisateur CSS qui accéde à un des ensembles de données d'une entreprise, CD_1, appartenant à COI_1, ne pourra pas, par la suite, accéder à d'autres ensembles de données d'entreprises (CD_2 et CD_4) dans ce même COI. Par contre, il pourra accéder à n'importe quels CDs (CD_3 et CD_5) dans un autre COI, COI_2 .

Pour cela, les auteurs de [Brewer and Nash, 1989a] définissent la fonction $PR(S)$ qui retourne l'ensemble des objets qu'un sujet S a déjà lu. Une version préliminaire de la muraille de Chine appelée *Chinese Wall-Simple Security Condition Preliminary Version* s'exprime comme suit :

Définition 2.2.3 (Chinese Wall-Simple Security Condition Preliminary Version) *S peut lire O si et seulement si l'une ou l'autre des deux conditions suivantes est vraie :*

1. *Il existe un objet O' tel que S a accédé à O' et que $CD(O') = CD(O)$.*
2. *Pour tous les objets O', $O' \in PR(S)$ alors $COI(O') \neq COI(O)$.*

Ensuite, la définition 2.2.4 permet la prise en compte de la présence de données publiquement accessibles dans les entreprises. Ces données sont appelées des données nettoyées et sont librement accessibles. Il faut faire la différence au niveau du modèle entre des données dites nettoyées et les autres.

Définition 2.2.4 (Chinese Wall-Simple Security Condition) *S peut lire O si et seulement si une des trois conditions suivante est vraie :*

1. *Il existe un objet O' tel que S a accédé à O' et que $CD(O') = CD(O)$.*
2. *Pour tous les objets O', $O' \in PR(S)$ alors $COI(O') \neq COI(O)$.*
3. *O est un objet nettoyé.*

Finalement, la propriété de la muraille de Chine doit prévenir les flux d'information transitifs permettant de récupérer de l'information indirectement par un échange d'information entre deux sujets via un objet tiers partagé. Pour cela, la définition complète de la Muraille de Chine dite *CW-* Property* est introduite :

Définition 2.2.5 (Chinese Wall-* Property) *S peut écrire dans O si et seulement si les deux conditions suivantes sont vraies :*

1. *Chinese Wall-Simple Security Condition permet à S de lire O.*
2. *Pour tous les objets non-nettoyés O', S peut lire O' si $CD(O') = CD(O)$.*

Cela revient à contraindre l'utilisateur à un seul domaine d'intérêt. Dans ce cas, il ne pourra pas utiliser un second domaine pour faire transiter de l'information.

Dans [Lin and Pan, 2009, Lin, 2007, Lin, 2000, Lin, 1989], T.Y. Lin propose de modifier la méthode de représentation des conflits pour passer d'ensembles disjoints à des relations binaires. En effet, en prenant le cas de la guerre froide et du conflit entre les USA, UK et URSS, il a démontré que les ensembles disjoints ont un manque d'expressivité. Par conséquent, l'utilisation des ensembles mène à un conflit inexistant entre USA et UK car ils sont tous les deux en conflits avec URSS. En prenant des relations binaires, les trois conflits sont modélisés de manière séparée : $USA \rightarrow URSS, UK \rightarrow URSS, URSS \rightarrow USA, UK$ qui les représente correctement les conflits. De plus, T.Y. Lin propose l'utilisation d'un nouveau concept "être allié avec" qui exprime explicitement l'absence de concurrence entre deux entités. Il propose aussi d'étendre le modèle pour y inclure des poids pondérant le niveau de concurrence entre deux entités.

R.S. Sandhu dans [Sandhu, 1992a] exprime le fait que CW est trop restrictif pour être appliqué sur un système réel car, en pratique, chaque utilisateur a forcément des problèmes de concurrence et se retrouve à ne pouvoir que lire des données et ne pas pouvoir en écrire pour respecter la modélisation qui prend en compte les flux d'information indirects de CW. En effet, à partir du moment où un sujet est dans plus d'une entreprise, cela revient à lui supprimer l'autorisation d'écriture. Le système devient donc un large dépôt en lecture seule. De plus, en découpant l'ensemble des sujets en deux sous groupes, R.S. Sandhu montre que le modèle CW est modélisable sous la forme d'un treillis ce qui permet à la propriété d'être évaluée dans le cadre de l'Orange Book [TCSEC, 1985].

Modèle DTE

Le modèle *Domain and Type Enforcement* (DTE) [Boebert and Kain, 1985] est disponible depuis des années dans certains systèmes d'exploitation [Badger et al., 1995]. Il propose un typage

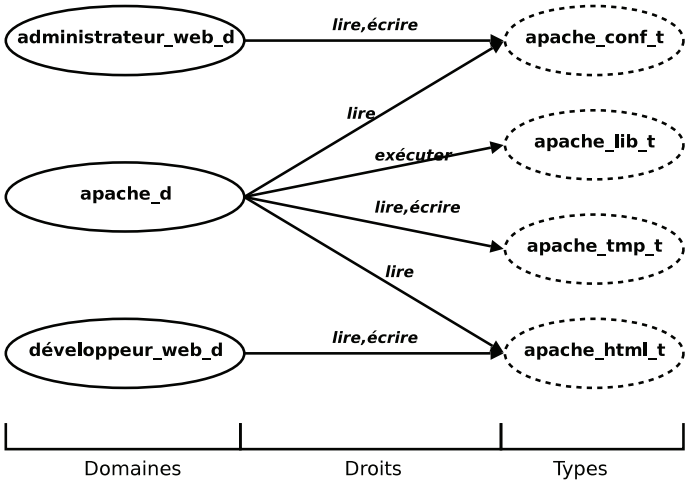

FIGURE 2.4 – Configuration DTE pour apache

fin des objets et permet d'implanter différents modèles de protection. 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. Concrètement, dans un système d'exploitation, les politiques de sécurité définies via le modèle DTE visent, par exemple, à :

- Restreindre les ressources pour les programmes privilégiés (s'exécutant sous le compte `root`) ;
- Contrôler les accès aux ressources sensibles.

Lorsqu'un nouveau processus est créé, il hérite automatiquement du domaine de son père. 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. 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`, avec 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. Celle-ci 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).

Discussion

Les modèles *Mandatory Access Control* (MAC) ont été envisagés afin de répondre aux faiblesses des modèles DAC (cf. partie 2.2.1, page 27). On distingue différents modèles de type MAC. Chacun de ces modèles [Bell and La Padula, 1973, Biba, 1975, Clark and Wilson, 1987],

[Brewer and Nash, 1989b] répond à une problématique spécifique, par exemple la nécessité de confidentialité pour des documents classifiés dans BLP ou la nécessité de contrôler les conflits d'intérêt [Brewer and Nash, 1989b].

La plupart de ces modèles ne prennent pas en compte explicitement les flux indirects. Cependant dans certains cas, ils permettent d'empêcher, implicitement, les flux d'information transitif dans le cadre des modèles MLS, comme démontré dans [Biba, 1975]. En effet, supposons que A et B sont des entités ayant un faible niveau de sensibilité et X et Y sont des entités ayant un fort niveau de sensibilité. Les flux entre A et B ainsi que ceux entre X et Y sont autorisés. Le modèle empêche les flux indirects allant de A ou B vers X ou Y en empêchant simplement les flux directs allant de A ou B vers X ou Y. Un flux d'information transitif tel que $A \rightarrow B \rightarrow X \rightarrow Y$ est alors impossible.

Cependant, les systèmes d'exploitation relèvent de problématiques plus complexes. Comme proposé dans le modèle DTE [Boebert and Kain, 1985], il est nécessaire de pouvoir associer à chaque sujet et chaque objet un type précis afin de pouvoir définir une politique de protection. Cette approche est plus ouverte. En effet, comme le montre [Fraser and Badger, 1998], au moyen de l'approche DTE, on peut représenter les différents modèles de protection et les combiner. En pratique, il manque un formalisme qui permet de facilement manipuler à la fois les flux d'information directs et indirects qui sont observables par le système d'exploitation. La conséquence est la difficulté de spécifier de nouvelles propriétés reposant sur des activités systèmes complexes afin de bien prendre en compte les besoins de protection.

2.3 Protection système orientée objectifs de sécurité

Dans cette section, nous présenterons les différentes façons de mettre en œuvre les modèles de protection vus précédemment. Nous présenterons tout d'abord les approches orientées modélisation d'un système. Ces approches reviennent à un problème complexe de résolution de contraintes sur le modèle. Elles sont, dans la pratique, difficiles à appliquer à tout un système. Ensuite, nous présenterons les approches traitant explicitement de l'intégrité et de la confidentialité. En pratique, les solutions traitent uniquement d'un cas particulier de propriété. Par la suite, nous présenterons les approches permettant de contrôler les flux d'information. Celles-ci s'intéressent exclusivement aux contrôles de flux mais ne s'occupent pas du problème d'expression de politique de sécurité. Finalement, nous présentons 3 propriétés qui sont les plus courantes dans les systèmes de protection. Tout d'abord, l'abus de privilèges est abordé. Puis, les systèmes obligeant la séparation de privilèges, statique ou dynamique, et pour finir, le contrôle des situations de concurrence.

2.3.1 Modélisation par automates

Description

La plupart des approches que nous présentons ici, comme [Clarkson and Schneider, 2008] ou [Foley, 2003], proposent de modéliser un système sous la forme d'un automate d'états. L'analyse des propriétés de sécurité revient à un problème de satisfaction de contraintes [Van Hentenryck, 1989] (Constraint Satisfaction Problem CSP). Il s'agit de rechercher des états satisfaisants des contraintes ou des critères. La résolution de nombreux CSP nécessitent la combinaison d'heuristiques et de méthodes d'optimisation combinatoire pour être résolue en un temps raisonnable. Les propriétés sont donc formalisées sous la forme de contraintes entre les différents états du système. Le problème est le passage à l'échelle sur un système réel. La modélisation d'un système d'exploitation sous la forme d'un automate à états est en pratique irréalisable. Même avec des simplications, les automates restent très complexes et la résolution de contraintes ne peut pas être réalisée en temps réel. Il est donc difficile d'implanter une méthode de protection qui se base sur ce principe.

Dans [Amoroso and Merritt, 1994], les auteurs proposent une méthode permettant d'évaluer l'intégrité d'un système en se basant sur un automate d'états du système. Ils introduisent trois types d'intégrité (A, B, C).

L'intégrité *Type-A* qui interdit toute transition d'un état sûr vers un état non sûr.

L'intégrité *Type-B* qui interdit les transitions d'un état sûr vers un état non sûr si elles sont provoquées par une entité externe au système mais autorisent les transitions provoquées par des entités locales vers des états non sûr.

L'intégrité *Type-C* n'applique aucun contrôle sur les transitions d'un état sûr vers un état non sûr.

En se basant sur l'automate, ils classifient les systèmes suivant ces trois types d'intégrité.

Dans [Schneider, 2000], l'auteur propose un ensemble de propriétés de sécurité orienté vers la sûreté d'un système. Pour cela, il modélise le système sous la forme d'un automate à états. Comme nous l'avons expliqué, le problème de cet automate (comme tout autre automate à états) est qu'il est potentiellement infini, c'est-à-dire non borné. L'auteur déclare explicitement que les propriétés d'intégrité et de confidentialité, qui nous intéressent, ne sont pas prises en compte. Par exemple, les flux d'information ne sont pas contrôlables car ils ne peuvent pas s'exprimer sous la forme de propriété de sûreté. L'auteur propose une implémentation formelle qui se base sur les traces d'exécution d'un système avec une trace représentant un processus. Par la suite, l'auteur dans [Clarkson and Schneider, 2008] propose le concept d'hyperpropriété qui est une propriété composée de plusieurs traces. Ce concept permet de contrôler les flux d'information entre plusieurs processus en composant leurs traces. L'hyperpropriété permet également de se protéger contre les attaques de type non-interférence en combinant les traces des processus privilégiés et non privilégiés. D'autres auteurs ont ensuite repris ce concept dans [Bauer et al., 2002] et proposent une amélioration qui permet par l'ajout ou la suppression d'instructions, de forcer la protection d'une propriété.

L'auteur de [Jacob, 1991] propose une approche formelle de l'intégrité. [Jacob, 1991] définit un mécanisme permettant de garantir l'intégrité d'un système pour chaque nouvelle opération dans ce dernier. Pour cela, elle autorise ou refuse l'opération en décrivant toutes les séquences d'opérations. Mais l'auteur de [Foley, 2003] déclare que cela n'est pas applicable car il faut définir presque manuellement toutes les séquences d'opérations pouvant conduire à cet état. Le fonctionnement est similaire à l'approche de [Schneider, 2000] et la conclusion sur l'impossibilité de contrôler l'intégrité repose sur le même constat que [Clarkson and Schneider, 2008] pour l'expression d'hyperpropriétés comme les flux d'information. En effet, il faut une trace par processus. Il n'est pas clair que ces traces puissent être dynamiques. Enfin, l'analyse de la composition de ces traces est problématique.

Discussion L'utilisation du modèle présenté dans [Amoroso and Merritt, 1994] est plus orienté vers l'évaluation de l'intégrité des systèmes distribués qui contiennent des composants modélisables sous la forme d'automate d'états. L'application de ce modèle sur un système d'exploitation complet n'est pas réaliste car l'expression de son automate complet devient impossible. De plus, la combinatoire liée à l'analyse de l'automate empêche toute application temps réel. Enfin, la combinaison de traces reste un problème ouvert.

2.3.2 Intégrité et Confidentialité

Biba La mise en oeuvre du modèle Biba pour l'intégrité des systèmes patrimoniaux (en opposition aux systèmes militaires) a été explicitée dans [Lee, 1988]. L'auteur met en avant l'importance du concept de sujets sûrs. Un sujet sûr est une entité particulière du système qui est autorisée à contourner la politique de sécurité du système. Ils permettent le fonctionnement réel du système et des opérations d'administration de celui-ci sans pour autant autoriser des actions allant à

l'encontre de l'intégrité du système. Le code de ces sujets doit être prouvé formellement afin de garantir l'absence de violations d'intégrité. En pratique, il est très difficile de prouver formellement une application.

Dans [Liang and Sun, 2001], les auteurs proposent une implémentation du modèle Biba Low Watermark supportant deux niveaux d'intégrité. Ce modèle est implanté sous la forme d'un module noyau (RS-MIP) sous GNU/Linux.

Séparation par niveau Dans [Chaudhuri et al., 2009], les auteurs proposent une version simplifiée du contrôle des flux d'information pour l'intégrité d'un système. La protection qu'ils introduisent se base sur un modèle multi-niveau et une labélisation dynamique des entités. Le modèle appliqué est qu'une entité peut écrire dans des entités ayant un niveau supérieur au sien. Leur protection est implantée sous la forme d'une sonde de protection contre les intrusions IPS sur le système MS-Windows Vista. Par ailleurs, le noyau Windows, à partir de Vista, utilise des niveaux de privilèges qui peuvent servir, soit pour le contrôle d'intégrité, soit pour la confidentialité. Cependant, l'approche n'est pas documentée. Elle ne fait pas l'objet de publications scientifiques. L'ensemble des failles trouvées sur ce type d'OS montre à l'évidence qu'il n'y a pas de contrôle avancé des flux d'information.

Exécution Sûre La propriété d'*intégrité d'exécution*, aussi connue sous le terme *chemin d'exécution sûr* (en anglais, Trusted Path Execution - TPE), permet de s'assurer que tous les binaires exécutés sur un système proviennent de répertoires reconnus comme sûr. Cela évite qu'une personne mal intentionnée puisse télécharger (ou faire télécharger) un binaire puis l'exécuter. Dans [Mohay and Zellers, 1997], l'authenticité du binaire est contrôlée à l'exécution grâce à une liste d'empreintes reconnues comme sûres. Si l'empreinte du binaire est dans cette liste, l'exécution est autorisée sinon elle est refusée. Des approches similaires, comme celle présentée dans [Iglio, 1999], proposent d'augmenter ce modèle en instaurant deux modes : un mode qui n'est pas de confiance où seulement les binaires sûrs peuvent être exécutés et un mode de confiance avec un utilisateur privilégié autorisé à exécuter tous les binaires. Cette approche est relative au concept de *Sujet Sûr* ayant plus de privilèges.

Dans [Rahimi, 2004], les auteurs appliquent le principe d'intégrité d'exécution à GNU/Linux via un module de sécurité LSM. En pratique, ils autorisent l'exécution des binaires provenant uniquement des répertoires systèmes du type `/usr/bin` ou `/bin` et interdisent les exécutions provenant de répertoires temporaires comme `/tmp`. De plus, ils implémentent le concept de *Sujet Sûr* pour un utilisateur privilégié.

Multi Level Security Dans [Wang et al., 2008], les auteurs proposent un modèle simple de la confidentialité qui se repose uniquement sur deux niveaux de sensibilité des informations : haut et bas. L'originalité de leur approche est qu'elle a été conçue pour être appliquée sur un système réel (dans leur cas, Microsoft Windows XP) et qu'elle permet de conserver la confidentialité de données sur un système même si celui-ci est infecté par des spywares. Leur implantation se base sur le suivi des flux implicites et explicites du système et l'étiquettage dynamique des données.

Utilisabilité Le but des auteurs de [Li et al., 2007a], tout comme ceux de [Fraser, 2000] et de [Mao et al., 2009], est de proposer une application du principe de l'intégrité de manière mandataire pour un système d'exploitation. Ils veulent que cette application soit simple à utiliser, contrairement aux autres solutions mandataires qui demandent une configuration complexe. Leur but est de protéger un poste utilisateur contre des attaques provenant du réseau. Ils partent du concept que : "mieux vaut une assez bonne sécurité qui est facilement utilisable plutôt qu'une meilleure sécurité plus complexe à utiliser". Ils utilisent un modèle d'intégrité qui se base sur deux niveaux (haut et

bas). Tous les processus et fichiers interagissant avec le réseau ont un niveau d'intégrité bas qui leur interdit toute communication et échange avec le niveau haut. Toute application ayant le niveau haut et voulant interagir avec le niveau bas descend au niveau bas, sauf exception (SSH et autres solutions d'administrations à distance).

Discussion L'implantation [Liang and Sun, 2001] ne permet que la prise en compte de deux niveaux. Elle ne permet donc pas de prendre en compte les critères tels que présentés dans [TCSEC, 1985] ou [ITSEC, 1991]. Enfin elle ne supporte qu'une propriété dérivée.

La protection présentée dans [Chaudhuri et al., 2009] ne s'applique qu'aux flux directs. Les flux indirects ne sont pas pris en compte. Cette protection est limitée au contrôle des écritures directes.

La solution décrite dans [Rahimi, 2004] ne propose qu'une propriété dérivée. Là encore, elle ne prend pas en compte les suites d'actions conduisant à une exécution illégale d'un binaire par transitivité.

L'approche [Wang et al., 2008] ne permet pas d'exprimer d'autres propriétés et est limitée à deux niveaux de confidentialité. De plus, elle est extrêmement liée à leur implantation sous MS-Windows.

Les approches orientées utilisabilité, telle que celle présentée dans [Li et al., 2007a], ne permettent que de définir une politique de sécurité limitée même si facilement exprimables. De plus, elles réutilisent souvent les politiques discrétionnaires sources de vulnérabilité. Cette protection est donc limitée et présente des vulnérabilités potentielles.

L'ensemble des solutions implante seulement des cas particuliers de propriétés d'intégrité et de confidentialité. En effet, aucune ne propose de langage pour formaliser d'autres propriétés que celles prises en compte. Enfin, la qualité de la protection est souvent discutable et partielle. Elle prend rarement en compte les flux indirects et utilisent dans certains cas des informations facilement compromises.

2.3.3 Contrôle de flux d'information

Contrôle centralisé des flux d'information

Coloration L'une des approches pour suivre les flux d'information est l'utilisation de la *coloration dynamique* des variables puis de suivre leur *propagation* dans le système [Clause et al., 2007]. En pratique, une étiquette, ou couleur, est affectée à une variable, c'est-à-dire un objet. Quand cet objet est lu par un sujet, ce dernier est coloré par l'étiquette de l'objet. Par transitivité, le système est coloré suivant la propagation de l'information.

LIFT [Qin et al., 2006] est une application de cette coloration au niveau matériel pour les systèmes sous MS-Windows. Son but n'est pas de détecter les flux interdits mais d'empêcher l'exécution de failles de bas niveau. Cette approche est plus à rapprocher de
Valgrind [Newsome and Song, 2005] permettant une modélisation et une détection de malware.

Dans [Triem Tong et al., 2010], les auteurs proposent une implantation du principe de coloration des données. Ils peuvent donc suivre la propagation des données au sein du système. Bien que leur principe permet de suivre dynamiquement les flux d'information qu'ils soient directs ou transitifs, leur but est d'empêcher les violations de la politique de contrôle d'accès autorisant des liens directs entre sujets et objets.

Analyse de politique Dans [Guttman et al., 2005], les auteurs proposent de vérifier une politique de contrôle d'accès SELinux pour l'application à la protection contre des flux d'informations illégaux. Pour cela, ils utilisent une approche de model checking qui se base sur une logique temporelle. Bien entendu, leur limitation est qu'ils ne peuvent pas protéger des flux non observables par

le noyau comme toute approche de ce type. Mais il est de tout façon impossible de protéger contre tous les canaux cachés [Wray, 1991]. En se basant sur les opérations effectuées par les sujets, ils peuvent déduire le sens du flux d'information. Ensuite, ils utilisent le principe de causalité pour trouver les liens entre ces flux d'information et en déduire les flux d'information transitifs. Puis, ils sont capables de vérifier que des flux d'information directs ou transitifs n'existent pas dans une politique. Ils sont également capables de vérifier qu'un workflow composé de plusieurs opérations entre un ensemble d'entités passe bien par des entités tierces de confiance. Ils permettent donc de vérifier qu'une politique mandataire respecte des objectifs de sécurité exprimés sous forme de flux d'information. Dans [Knorr, 2000], les auteurs proposent une approche similaire se basant sur l'analyse de politiques pour la détection de flux d'information directs ou transitifs. Leur approche se base sur un réseau de Petri coloré pour modéliser les états du système tel que décrit par la politique.

Dans [Briffaut et al., 2009a], les auteurs utilisent un langage de formalisation des propriétés de sécurité. Ce langage permet d'exprimer des propriétés complexes combinant plusieurs flux. Contrairement aux autres approches, le pouvoir d'expression va plus loin. Une solution est proposée pour analyser des politiques SELinux. L'analyse fournit toutes les activités qui peuvent violer les propriétés de sécurité. De plus, l'approche permet aussi de vérifier des politiques SELinux dynamiques au moyen de contraintes sur les évolutions.

Contrôle décentralisé des flux d'information

Le *contrôle décentralisé des flux d'information* (*Decentralized Information Flow Control* - DIFC) a émergé du besoin de contrôler les flux d'information, non seulement au niveau du système mais aussi au sein des applications. En effet, pour un système de contrôle d'accès, il n'est pas possible de contrôler les flux au sein d'une unique entité. Une partie du contrôle des flux est à faire au niveau de l'application. Ce constat est parti du fait que de plus en plus d'applications sont multi-utilisateurs et sont donc amenées à traiter des informations avec des niveaux de sensibilité différents. C'est donc un problème dû à un trop faible niveau de granularité au niveau système.

Une autre type d'approche [Dalton et al., 2007] de suivi décentralisé des flux d'information dynamiques DIFT (*Decentralized Information Flow Tracking*), propose de marquer (colorer) les données non sûres et de suivre leur propagation dans le système. Les auteurs de [Dalton et al., 2008] proposent d'utiliser cette méthode pour la détection de *buffer overflow* [2] en temps réel et protéger le système contre ses effets. De plus, en partant du constat que les méthodes DIFT provoquent un sûrcout très important allant de 300% à 3700%, les auteurs proposent une implantation matérielle sur FPGA.

Les auteurs de [Hicks et al., 2007] proposent de combiner un système de contrôle mandataire, SELinux, avec un contrôle des flux au niveau applicatif en utilisant un langage typé pour la sécurité, Java Information Flow [Sabelfeld and Myers, 2003] et une politique applicative
JifPol [Hicks et al., 2006]. SELinux va passer les labels de sécurité à Jif qui va se charger d'assurer le respect de la politique vis-à-vis de ces labels au niveau applicatif. Cela permet donc d'avoir une meilleure granularité et donc de formuler des politiques de sécurité plus précises.

Asbestos [Vandebogart et al., 2007] a été la première approche proposant un système d'exploitation orienté vers la protection des flux d'information. Le but initial d'Asbestos est de pouvoir supporter de manière efficace et sécurisée des applications à large échelle. Dans ces applications, les utilisateurs sont isolés les uns des autres par des mécanismes forts au niveau système en se basant sur une politique fournie par l'application. L'isolation étant basée sur des politiques, il est

2. Un dépassement de tampon ou débordement de tampon (en anglais, buffer overflow) est un bug causé par un processus qui, lors de l'écriture dans un tampon, écrit à l'extérieur de l'espace alloué au tampon, écrasant ainsi des informations nécessaires au processus.

possible de définir des flux autorisés entre les utilisateurs permettant un partage de données. Une application même détournée ne pourra pas aller à l'encontre de sa politique, le système de protection se trouvant dans l'espace noyau. Dans ce système, tout est labelisé avec un identifiant unique et quatre niveaux MLS sont également utilisés. Asbestos permet de protéger aussi bien un système mandataire que discrétionnaire contre les flux directs ou transitifs qu'ils soient explicites ou implicites. Comme nous l'avons précédemment dit, il n'est pas possible d'éliminer les canaux cachés mais il est possible, comme c'est le cas dans Asbestos, de rendre le plus difficile possible l'apparition de flux d'information illégaux. En pratique, Asbestos utilise un système de labélisation dynamique et la contamination/colorisation de ces derniers. Grâce à des fonctions de checkpoint/-rollback dans le noyau, un processus peut traiter les requêtes de plusieurs utilisateurs sans pour autant prendre des risques de sécurité. Cela fonctionne en sauvegardant l'état du processus pour un utilisateur, puis en traitant un autre utilisateur et en restaurant l'état précédent quand le premier utilisateur revient. Le processus va donc pouvoir agir au nom de plusieurs utilisateurs sans pour autant remettre en cause la propriété d'isolation. De plus, contrairement aux approches mandataires classiques, la définition de politique dans Asbestos et dans HiStar [Zeldovich et al., 2006] n'est pas faite par une entité centrale mais par chaque application.

Le principe a été ensuite repris sous Linux en introduisant le modèle Flume [Efstathopoulos and Kohler, 2008]. Leur idée est qu'il est plus simple pour les développeurs d'écrire des propriétés de sécurité et d'incorporer la gestion des labels dans leur programme que l'approche actuelle où un administrateur définit la politique de l'ensemble du système et des applications. Pour ce besoin, les auteurs de [Efstathopoulos and Kohler, 2008] proposent un langage de description de politique de flux. Ce langage est simple, il se limite à la description des communications entre les entités du système et ils ne proposent pas d'implantation de modèle de sécurité, comme BLP. En pratique, Flume remplace les appels système par des procédures d'appels inter-processus entre l'application et un moniteur de référence ce qui permet d'éviter la vérification en espace noyau. Les auteurs ont évalué à 2% la quantité de code à réécrire et une augmentation de surcoût allant de 30 à 40%.

Discussion

L'approche par coloration dynamique pour le suivi et le contrôle des flux d'information sur un système se base sur différentes méthodes comme l'instrumentation dynamique de code. Elle ne peut cependant pas être appliquée à un système complet mais seulement à l'échelle d'une ou plusieurs applications. Les seules approches qui permettent le passage à l'échelle d'un système sont des colorations au niveau matériel des registres et demandent une émulation et une instrumentation très lourdes. Elles sont utilisées pour la détection et l'étude de comportement de malwares [Yin et al., 2007] mais ne sont pas applicables à la protection car elles provoquent un surcoût 20 fois supérieur à la normale. De plus, dans la plupart des cas, seuls les flux d'information directs (explicites) sont pris en compte et l'instrumentation du code n'est pas triviale.

LIFT et les autres approches de ce type ne permettent pas de spécifier des objectifs de sécurité mais plutôt de détecter des virus connus ou de générer des signatures pour des virus inconnus.

L'approche présentée dans [Triem Tong et al., 2010] ne propose pas de langage de haut niveau permettant l'expression d'objectifs de sécurité et leur implantation demande de spécifier pour chaque objet une politique et si besoin une coloration initiale. De plus, les problèmes d'implantation réelle des systèmes de colorisation sont connus avec des sur-utilisations très élevées et les auteurs ne proposent aucune implantation dans le système. Seul un exemple simple demandant une écriture complète d'une politique pour chaque objet, est proposé. Un exemple plus complexe est proposé mais ne fonctionne pas avec leur implantation. Pour finir, ils traitent d'une seule propriété qui est la cohérence d'une politique directe (explicite). Cela vise à détecter tous les flux indirects

violant cette politique. Dans la pratique cette approche est inadaptée car elle ne permet pas de trier parmi des flux indirects ceux qui sont légitimes et ceux qui ne le sont pas.

Les approches proposées dans [Guttman et al., 2005] et [Knorr, 2000] visent la vérification de politiques mandataires. Il n'est pas possible de formaliser des propriétés combinant plusieurs flux. Dans [Briffaut et al., 2009a], on s'intéresse à des méthodes qui calculent statiquement les activités illicites. La solution nécessite donc d'avoir une politique obligatoire en entrée afin d'en extraire une liste statique des activités illégales. Par ailleurs, la vérification de politiques dynamiques n'est possible qu'au moyen de contraintes réduisant l'espace de recherche. La méthode n'est donc pas applicable à des politiques entièrement dynamiques comme la muraille de Chine.

Bien que l'approche de [Dalton et al., 2008] ne demande pas de modification du système, elle est orientée vers la détection de buffer overflow (ou d'autres attaques bas niveau). Bien que cette faille soit très dangereuse et très courante, ce n'est pas en soi un objectif de sécurité. Un objectif peut être rompu par l'intermédiaire d'un buffer overflow mais son but est de protéger contre les actions malveillantes découlant de l'utilisation du buffer overflow et non de protéger contre le buffer overflow lui-même. Leur approche ne permet pas de définir des objectifs de sécurité de haut niveau et donc de définir une politique de sécurité complète. Nous exprimons donc l'importance de bien faire la différence entre les systèmes de protection contre les failles et malwares (tel que DIFT) des approches qui protégent un système en se basant sur des objectifs de sécurité comme présentée dans [Hicks et al., 2007].

Le principal problème de l'approche présentée dans [Hicks et al., 2006] est qu'elle demande de redévelopper les logiciels dans des langages fortement typés prenant en compte la sécurité. Etant donné que l'approche présentée sur [Hicks et al., 2007] repose sur l'utilisation de l'approche langage [Hicks et al., 2006] elle présente les mêmes inconvénients.

Le problème des approches [Vandebogart et al., 2007, Zeldovich et al., 2006], [Efstathopoulos and Kohler, 2008] est qu'elles traitent des flux élémentaires et ne présente pas de langage pour formaliser des propriétés de plus haut niveau combinant ces flux élémentaires. De plus, elles demandent une réécriture de tout ou partie des applications. Enfin, à part Flume, les approches introduisent des systèmes d'exploitation nouveaux qui ne sont pas supportés par les vendeurs d'applications et de matériels. Cela pose des problèmes en terme de maintenance et d'évolution de ces systèmes. [Krohn and Tromer, 2009] propose une modélisation de Flume afin de montrer qu'il ne présente pas de canaux cachés directs entre leur deux niveaux de protection. Cela ne permet pas de pouvoir formaliser des propriétés de sécurité puisque le modèle vise à vérifier la sûreté de leur système et non l'expression d'objectifs de sécurité.

2.3.4 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 comportement 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 protection d'un système : 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 and Sekar, 2001] proposent des macro-définitions 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* [3] 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. Cette approche est plus orientée vers la détection mais le même principe est repris dans [Nuansri et al., 1999] pour la protection.

Discussion La détection d'abus de privilèges vise à comparer les appels système aux appels considérés comme légitimes. Cela vise plus la détection d'intrusion que la protection étant donné que la description exhaustive des appels légitimes est impossible. Appliquer ces méthodes à la protection de l'ensemble d'un système est très difficile et reviendrait en pratique à des dénis de service répétés en raison d'un apprentissage non optimal. Les *flux d'informations* ne sont pas traités et les approches semblent difficilement pouvoir être étendues à la formalisation de propriétés de sécurité.

2.3.5 Séparation de privilèges

Introduction La séparation de privilèges a été définie dans [Saltzer and Schroeder, 1975] et requiert, pour qu'une tâche critique soit accomplie, une équipe d'au moins k utilisateurs. Une dérivation simple est qu'il faut deux personnes pour effectuer une tâche critique. Cette propriété ne dit pas quels utilisateurs doivent effectuer quelles tâches mais que plusieurs utilisateurs doivent coopérer pour effectuer une tâche.

La séparation de privilèges est tout particulièrement adaptée dans le modèle RBAC [Sandhu et al., 1996] où la séparation se fait par l'affectation (ou non) de rôles contenant les privilèges. Il suffit ensuite de pratiquer une séparation dans l'affectation des rôles conflictuels pour séparer la séparation de privilèges entre les sujets. Deux taxonomies de la séparation de privilèges pour les systèmes RBAC sont proposées par [Simon and Zurko, 1997] et [Nyanchama and Osborn, 1999]. Nous utiliserons la taxonomie proposée par [Kuhn, 1997] et plus particulièrement le premier axe, c'est-à-dire que nous différencions deux types de séparation de privilèges : statique et dynamique.

Statique La séparation statique donne un ensemble de droit à une entité et un second ensemble à une seconde entité. Il faut que la séparation soit appliquée entre ces deux ensembles. Par exemple, nous voulons appliquer la séparation de privilèges entre deux entités A et B en contrôlant que A ne peut qu'écrire des objets et B les exécuter. Une politique mandataire statique doit donc être définie afin de spécifier tous les privilèges. La vérification d'une politique de séparation statique de privilèges dans un environnement RBAC a été prouvée comme étant un problème coNP-Complet [Li et al., 2007b]. Mais, en modélisant la séparation de privilèges sous forme d'un pro-

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

blème de satisfaction SAT, le problème peut être résolu dans un temps raisonnable. Par exemple, Ahn et Sandhu ont proposé dans [Ahn and Sandhu, 2000], un langage permettant d'exprimer de telles règles de séparation de privilèges.

Dynamique La séparation dynamique de privilèges [Simon and Zurko, 1997] (ou séparation de privilèges opérationnelle) est comparable à une instanciation spéciale de la Muraille de Chine. Les ensembles de permissions sont définis dynamiquement en fonction des accès d'un sujet qui déterminent les privilèges futurs. C'est donc l'historique des accès qui permet de fixer les privilèges. Dans [Basin et al., 2009], les auteurs présentent une modélisation prenant en compte les relations directes et transitives pour le calcul dynamique des privilèges. Dans [Jaeger and Tidswell, 2001], les auteurs proposent une approche formelle permettant de vérifier la séparation de privilèges (statique et dynamique) sur un système en se basant sur une représentation de la politique RBAC sous forme d'un graphe. Puis ils expriment des contraintes sur ce graphe pour représenter la séparation de privilèges.

Une implémentation du principe de la séparation de privilèges dynamique est proposée dans [Radhakrishnan and Solworth, 2006]. Elle est orientée vers le contrôle entre deux groupes de privilèges. Il s'agit d'une version simplifiée de la séparation de privilèges dynamique pour le modèle RBAC.

Discussion La séparation de privilèges qu'elle soit statique ou dynamique est difficilement adaptable à l'ensemble d'un système d'exploitation. D'une part, il n'est pas facile de définir les ensembles de privilèges. Ensuite, il faudrait pouvoir vérifier les politiques associées. Enfin, la prise en compte à la fois des flux et des propriétés plus larges sort du cadre de ces travaux.

2.3.6 Situation de concurrence

Il existe plusieurs types d'attaques de situations de concurrence (Race Condition) sur les données [Yu et al., 2005], sur les signaux [Tahara et al., 2008], ou tout autre ressource partagée.

La situation de concurrence la plus classique est associée aux tests préliminaires que font les programmes sur les ressources. Elle est connue sous le nom time-of-check-to-time-of-use (TOCTTOU) et a été caractérisée par McPhee en 1974 dans [McPhee, 1974]. Sa définition est la suivante : "il existe un ordonnancement imprévisible entre les accès de deux processus à une ressource partagée". La plupart des systèmes de fichiers sont vulnérables à de telles attaques [Bishop, 1995, Bishop and Dilger, 1996]. De plus, avec l'augmentation des systèmes multi-cœur, la détection de races conditions devient de plus en plus complexe. En effet, dans ce cadre, la détection devient un problème NP-Complet [Netzer and Miller, 1990]. Rien que sur les systèmes de fichiers, environ 20 à 40 failles reposant sur des situations de concurrence sont trouvées chaque année. Quatre approches pour la protection contre les situations de concurrence ont été proposées :

Analyse de code En analysant au moment de la compilation le code d'une application, il est possible de détecter ou de trouver les parties de code pouvant potentiellement mener à une vulnérabilité par concurrence [Bishop and Dilger, 1996, Chen and Wagner, 2002, Chess, 2002], [Schwarz et al., 2005] ;

Détection Dynamique En auditant les appels systèmes pouvant potentiellement être utilisés lors d'une situation de concurrence, il est possible de détecter ces dernières. L'approche la plus connue est celle de Ko et Redmond qui permet de détecter les failles à posteriori [Ko and Redmond, 2002]. De nombreuses autres propositions [Lhee and Chapin, 2005], [Joshi et al., 2005] ont suivi reprenant le même principe ;

Protection Dynamique Si on se contente de détecter la concurrence, il est possible de suspendre (ou tuer) les processus ou des appels systèmes suspects. Ainsi,

le système [Cowan et al., 2001, Tsyrklevich and Yee, 2003, Uppuluri et al., 2005] peut être protégé ;

Système de fichiers Il est possible de créer des systèmes de fichiers qui traitent la concurrence. Deux principes sont proposés : construire des systèmes de fichiers transactionnels [Schmuck and Wylie, 1991, Wright et al., 2007] ou en proposant des appels systèmes conçus pour traiter la concurrence [Mazieres and Kaashoek, 1997]. Une solution décrivant une nouvelle API résistante aux situations de concurrence a été proposée [Tsafrir et al., 2008].

Discussion A part les systèmes transactionnels, aucune solution ne traite complètement les problèmes de concurrence. Les approches transactionnelles sont inadaptées pour être intégré dans les systèmes d'exploitation. De plus, les moyens pour traiter efficacement la concurrence doivent permettre de préciser explicitement les situations de concurrences à contrôler. Les approches a la fois de détection et de protection dynamique ne traitent pas tous les cas de concurrence. De plus, l'expression de la concurrence n'est pas aisée dans ses approches.

2.4 Conclusion

Cet état de l'art a montré que les politiques de sécurité qui incluent des propriétés concernant l'intégrité et la confidentialité, ne présentent pas un niveau de formalisation suffisament ouvert pour permettre de les améliorer, d'en définir de nouvelles ou pour s'intégrer dans un système d'exploitation.

Ensuite, nous avons montré la faiblesse des approches discrétionnaires et la nécessité d'avoir des approches mandataires. Cependant, les modèles mandataires existants ne prennent pas bien en compte les flux indirects et sont souvent difficilement applicables à un système d'exploitation.

Les modèles de protection sont des bases permettant la mise en place d'un système de sécurité configuré par des objectifs de sécurité. Que ce soient des contrôles d'accès classique ou se basant sur les flux, ils ne disposent pas de langage permettant de spécifier un large ensemble de propriétés de sécurité formant une politique complexe. Ils sont utilisés ensuite dans les système de protection orienté objectifs comme modélisation du système. Le modèle DTE est plus adapté à un système d'exploitation. Cependant, ce modèle n'introduit pas de moyens de formaliser de façon flexible un large ensemble de propriétés de sécurité.

Parmi les approches de protection système, la plus étudiée est celle reposant sur la modélisation du système sous forme d'un automate d'états. Les usages restent limités à la protection d'un sous-ensemble du système de part la difficulté de modélisation. Les approches qui se concentrent sur un cas ou un sous cas d'intégrité ou de confidentialité ne sont pas suffisamment générales. Elles ne peuvent pas être étendues. Les approches par contrôle de flux d'information visent essentiellement à colorer les flux. Ces méthodes présentent un sûrcout élevé qui rend leur généralisation difficile pour protéger l'ensemble d'un système. Les systèmes d'exploitation orientés contrôle de flux d'information ne proposent pas de langage pour l'expression de propriétés de sécurité. De plus, elles demandent de réécrire une partie du code des applications ou reposent sur des systèmes qui sont peu supportés.

Les approches d'abus, séparation de privilèges ou situation de concurrence ne traitent qu'un cas particulier et prennent difficilement en compte les flux indirects.

Il existe de nombreuses autres méthodes que celles présentées dans cet état de l'art. Elles sont souvent orientées détection d'intrusions et celles qui traitent de protection visent à détecter des comportements anormaux et non pas à garantir des propriétés de sécurité. La virtualisation est parfois présentée comme une solution de protection. En fait, comme le montre les résultats du Défi Sécurité (ANR SEC&SI), la virtualisation n'offre en pratique aucune garantie de sécurité.

En effet, seules les approches incluant des protection mandataires en profondeur, peuvent garantir des propriétés de sécurité et confiner les attaques ou les failles. Dans ce cas, le problème revient à définir un système de protection mandataire efficace. C'est l'idée sur laquelle nous travaillons.

On voit clairement que les manques, concernant la protection des systèmes, reposent d'abord sur l'absence de moyens efficaces pour formaliser les propriétés de sécurité pour un système d'exploitation, ensuite sur la faiblesse des systèmes mandataires existants en terme de couverture des propriétés de sécurité et enfin sur la trop grande complexité des systèmes mandataires en profondeur. Concernant ce dernier point, la solution SPACLik [Briffaut et al., 2009b] du Défi Sécurité a montré l'efficacité d'une approche mandataire en profondeur. Cependant, elle présente un niveau de complexité élevé. En effet, il faut tout d'abord définir des politiques mandataires complexes pour un système SELinux, ensuite compléter la protection au moyen de propriétés de sécurité et enfin fournir des moyens dynamiques pour mettre à jour les politiques concernées en fonction de l'activité.

On voit donc qu'il est nécessaire de fournir des moyens de formaliser les propriétés de sécurité pour un système d'exploitation. Il faut que cette formalisation soit adaptée à la dynamicité du système. Elle doit être simple à utiliser et doit pouvoir s'intégrer efficacement sur un système d'exploitation existant. Enfin, étant donné la difficulté pour évaluer l'efficacité d'une méthode de protection, il est nécessaire de disposer d'expérimentations suffisamment larges et variées prenant en compte des usages et des attaques réelles.

Chapitre 3

Langage de formalisation des activités

Les processus et les ressources d'un système d'exploitation peuvent être vus comme un ensemble d'entités qui interagissent et modifient éventuellement 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, c'est-à-dire un appel système, 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, correspond à un ensemble d'attributs (utilisateur, rôle, type, etc...). Un contexte est associé à un ensemble de processus ou d'objets du système. Par exemple, le contexte cs_{apache} est associé aux processus `apache` et le contexte cs_{var_www} aux fichiers Web contenus dans `/var/www/`. En pratique, un sujet réalise un appel système (du type lecture, écriture, exécution, envoi d'un signal, etc...) sur un objet ou un sujet donné. De plus, les interactions sont catégorisées selon leurs effets sur le système (transfert d'informations, changement de labels, etc.). Nous différencions les interactions potentielles, "c'est-à-dire celles qui pourraient se produire", des interactions réelles, c'est-à-dire celles qui sont liées à des événements systèmes ayant eu lieu. Nous appelons interactions estampillées les interactions réelles.

Le but de ce chapitre est de formaliser ces différentes notions puis de modéliser les interactions observables par le noyau du système d'exploitation, c'est-à-dire les traces observables par le noyau du système d'exploitation. Grâce à cette modélisation, nous proposons une formalisation et une classification des activités observables par le système d'exploitation. Ainsi, par la suite, nous sommes dans la capacité de donner une formalisation des propriétés de sécurité contrôlables au moyen de la capture, dans le noyau, des activités observables.

3.1 Entités du Système d'Exploitation et Opérations

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 d'exploitation. Un attribut peut, par exemple, correspondre à l'identité (nom d'utilisateur) d'une entité, au rôle de l'entité dans le modèle RBAC, au domaine ou au type de l'entité du modèle Domain Type Enforcement (DTE), etc. Il peut également correspondre au niveau de classification ou d'habilitation des modèles MLS (Biba ou Bell&LaPadula). Chaque entité du système est étiquetée par un unique contexte de sécurité, noté cs_{label} et plusieurs entités peuvent avoir le même contexte. Par exemple, tous les processus `apache` sont identifiés par le même contexte cs_{apache}. Il est possible de distinguer tous les processus `apache` en associant à

chacun un contexte spécifique. Cependant dans la pratique, il n'est pas utile de distinguer tous les processus `apache` les uns des autres, bien souvent l'utilisateur ou le rôle du processus permet un niveau de distinction suffisant. Nous notons un contexte, constitué de n attributs, par :

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

Sur un système d'exploitation, nous distinguons deux ensembles de contextes de sécurité correspondant aux deux types d'entité : l'ensemble des contextes de sécurité *sujets*, noté $\mathcal{CSS}$, et l'ensemble des contextes de sécurité *objets*, noté $\mathcal{CSO}$. L'ensemble des contextes de sécurité $\mathcal{CS}$ d'un système est donc l'union des deux ensembles précédents :

$$\mathcal{CS} = \mathcal{CSS} \cup \mathcal{CSO} \tag{3.2}$$

L'utilisation des contextes de sécurité permet de donner une abstraction indépendante 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 dans le système de fichiers. Un contexte devient donc un élément qui désigne, au minima, un type de classe d'entités, si ce n'est un unique objet. Les entités sont donc "fortement" typées. Les changements de type sont possibles. Mais, comme nous le verrons par la suite, ils doivent être contrôlés et observables afin de pouvoir garantir les propriétés de sécurité.

3.1.2 Opération Élémentaire

Définition Chaque système d'exploitation définit un ensemble d'opérations, généralement appelées *appels système*, qui forment les services offert aux processus s'exécutant sur 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 élémentaire*, notée oe, peut être vue comme une action qu'une entité du système effectue sur une autre entité. Certains appels système permettent de gérer les contextes.

Notation ensembliste Nous notons $\mathcal{EO}$ l'ensemble des opérations élémentaires disponibles sur un système.

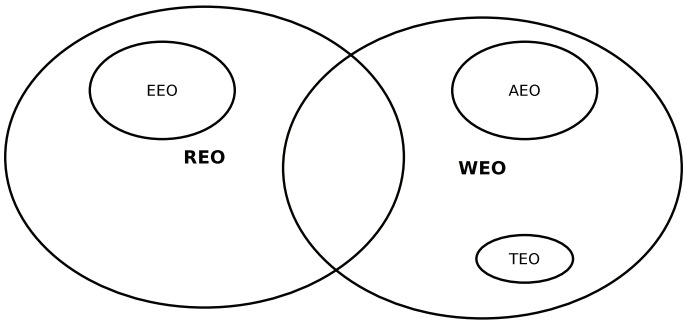

FIGURE 3.1 – Les différents ensembles d'operations élémentaires

Dans cette ensemble, nous notons deux sous-ensembles $\mathcal{REO}$ et $\mathcal{WEO}$ (voir la figure 3.1) contenant respectivement l'ensemble des opérations élémentaires de type lecture et écriture. L'ensemble des opérations de $\mathcal{EO}$ peuvent être classées dans ces sous ensembles. Une opération peut appartenir aux deux ensembles. Nous avons également besoin de distinguer pour l'ensemble $\mathcal{WEO}$ deux sous-ensembles $\mathcal{TEO}$ et $\mathcal{AEO}$ correspondant respectivement aux opérations de transition et

d'ajout d'information. Pour la transition d'un contexte cs_{source} vers un contexte cible cs_{cible}, il y a une écriture de l'état du contexte source dans le contexte cible puisqu'il s'agit du même processus qui transite d'un état vers un autre. Il est évident que l'ajout est bien une opération de type écriture. De la même façon, nous avons besoin pour l'ensemble $\mathcal{REO}$ de distinguer un sous ensemble $\mathcal{EEO}$ contenant les opérations de type exécution. Quand un sujet exécute un objet, il lit cet objet afin de l'exécuter, c'est donc bien une opération de type lecture.

Il peut exister d'autres sous-ensembles de $\mathcal{EO}$ selon les besoins d'expression dans les propriétés de sécurité. Notre formalisation permet facilement d'ajouter de nouveaux ensembles. Par exemple, une hypothétique propriété de sécurité nécessite de pouvoir faire la différence entre les opérations élémentaires de type signal et les autres. Pour cela, nous pouvons déclarer un nouveau sous ensemble $\mathcal{SEO}$ qui contiendra l'ensemble des opérations élémentaires de type signal.

Nous définissons également cinq fonctions qui permettent de faciliter la recherche d'appartenance à un sous-ensemble de $\mathcal{EO}$ pour une opération élémentaire donnée :

- $is\_read\_like : \mathcal{EO} \rightarrow \{V|F\}$ qui retourne vraie (V) si l'opération élémentaire est de type lecture, c'est-à-dire, $eo \in \mathcal{REO}$. Cette fonction est utile pour définir les flux d'information, mais est aussi essentielle pour l'expression d'un certain nombre de propriétés de sécurité comme celles relevant du modèle Biba.
- $is\_write\_like : \mathcal{EO} \rightarrow \{V|F\}$ qui retourne vraie si l'opération élémentaire est de type écriture, c'est-à-dire $eo \in \mathcal{WEO}$. Tout comme la fonction relevant du sous-ensemble de lecture, cette fonction est nécessaire à la définition et à l'orientation des flux d'information. De plus, elle est également utilisée dans un grand nombre de propriétés de sécurité comme celles relevant du modèle Muraille de Chine.
- $is\_append\_like : \mathcal{EO} \rightarrow \{V|F\}$ qui retourne vraie si l'opération élémentaire est de type ajout (concaténation), c'est-à-dire $eo \in \mathcal{AEO}$. Elle nous servira dans le cadre de certaines propriétés de sécurité et plus particulièrement pour l'ensemble des propriétés découlant du modèle Bell&LaPadula restrictif.
- $is\_transition : \mathcal{EO} \rightarrow \{V|F\}$ qui retourne vraie si l'opération élémentaire est de type transition, c'est-à-dire $eo \in \mathcal{TEO}$. Cette fonction permet de modéliser et donc d'observer les changements de contextes des sujets, étape indispensable pour suivre et contrôler les privilèges des processus.
- $is\_exec : \mathcal{EO} \rightarrow \{V|F\}$ qui retourne vraie si l'opération élémentaire est de type exécution c'est-à-dire $eo \in \mathcal{EEO}$. Cette fonction nous permet de définir l'action élémentaire de type exécution mais également des propriétés de sécurité associés à des exécutions.

Tout comme la possibilité d'ajouter de nouveaux sous ensembles à $\mathcal{EO}$, il est simple d'ajouter de nouvelles fonctions pour travailler sur ces sous-ensembles. Par exemple, en reprenant le sous-ensemble $\mathcal{SEO}$ qui contient les opérations élémentaires de type signal, on peut déclarer la fonction $is\_signal : \mathcal{EO} \rightarrow \{V|F\}$ qui retourne vraie si une opération élémentaire est de type signal.

3.1.3 Datation

Nous considérons qu'une date d est un entier qui représentent la date d'un événement observable par le système d'exploitation. Les entiers qui représentent les dates peuvent être incrémentés par le noyau du système. Mais on peut également envisager des mécanismes de datation externes et accessibles au noyau. L'ensemble $\mathcal{D}$ contient l'ensemble des dates du système. Chaque date d fait donc partie de l'ensemble $\mathcal{D}$ c'est-à-dire $d \in \mathcal{D}$. Les dates sont suffisamment précises pour classer historiquement (c'est-à-dire avant, après ou simultanément) deux événements.

En pratique, l'heure UNIX (POSIX Timestamp) est souvent utilisée pour horodater les événements d'un système. Il s'agit du nombre de secondes écoulées depuis le `1er janvier 1970 00:00:00 UTC` jusqu'à l'événement à dater. Sur le même principe, il est possible d'améliorer

assez aisément la précision à la milliseconde, microseconde, voir même en dessous. La datation mène à un second problème : il est nécessaire que l'espace de codage des dates soit suffisamment grand pour éviter d'en atteindre la fin. Une implémentation optimale doit donc permettre d'adapter dynamiquement la taille de cet espace de numérotation pour éviter de remonter dans le temps.

3.1.4 Événements et Datation

Un évènement système est une opération observable par le noyau. Chaque événement est horodaté avec une date unique et deux événements distincts sont donc ordonnables temporellement. Nous distinguons deux types d'événements observables dans notre formalisme : l'événement d'entrée d'un appel système et l'événement de retour d'un appel système. Par exemple, dans la figure 3.2, la fonction `fopen` de la `libc` est invoquée par un processus (cs_{source}). Suite à cet appel de fonction, la `libc` invoque l'appel système `open` pour effectuer effectivement l'action d'ouverture d'un fichier. La première instruction de `open` dans le noyau est désignée par l'événement $EE \in \mathcal{E}$ et est horodatée. Ensuite, l'appel système se déroule et la dernière instruction en espace noyau, c'est-à-dire l'appel à `return`, est désignée par l'événement $ES \in \mathcal{E}$ et horodatée. Suite à cette action, l'exécution de `fopen` retourne en espace utilisateur et la `libc` reprend la main. Finalement, le résultat de `fopen` est renvoyé au processus qui l'a invoqué, c'est-à-dire cs_{source}.

Événement d'entrée dans un appel système

Pour un processus cs_{source} qui exécute un appel système eo sur un objet cs_{cible}, l'événement d'entrée est noté $EE(cs_{source}, cs_{cible}, oe)$. Il correspond à la première instruction qui s'exécute en espace noyau pour l'appel système correspondant. Par souci de concision, nous notons abusivement EE lorsqu'il n'y a pas nécessité de préciser les contextes et l'opération concernée. Suite à l'événement EE, l'exécution, à proprement parler, du code de l'appel système commence. Il est possible pour le noyau de dater EE de manière précise à l'aide d'une date d_{ee}, avec $d_{ee} \in \mathcal{D}$, tel que définie dans la section 3.1.3.

Ainsi soit la fonction $\mathcal{E} \rightarrow \mathcal{D}$, pour un événement d'entrée $EE(cs_{apache}, cs_{var\_www}, file : read)$ qui apparait à la date `2819` : $d_{ee} = date(EE) = 2819$.

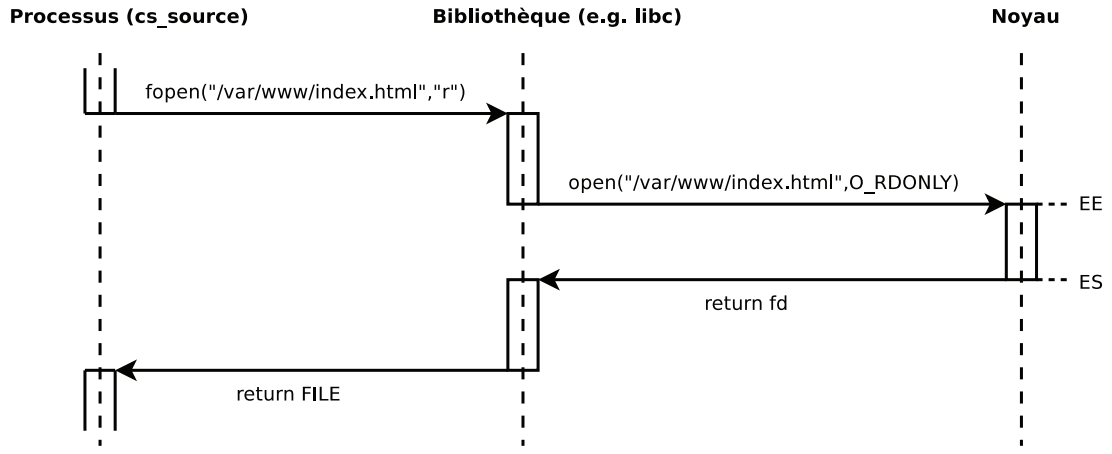

FIGURE 3.2 – Schéma de fonctionnement des événements d'entrée et de sortie de l'appel système `open`

Événement de retour d'un appel système

Pour un processus cs_{source} qui s'exécute un appel système eo sur un objet cs_{cible}, l'événement de retour est noté $ES(cs_{source}, cs_{cible}, eo)$. Il correspond à la dernière instruction qui s'exécute en espace noyau pour l'appel système correspondant. Par souci de concision, nous notons abusivement ES lorsqu'il n'y a pas nécessité de préciser les contextes et l'opération concernée. Suite à cet événement ES, l'exécution continue en espace utilisateur. Tout comme l'événement d'entrée dans un appel système, l'événement de sortie de l'appel système est également horodaté par le noyau par une date d_{es} avec $d_{es} \in \mathcal{D}$.

Ainsi pour la fonction $\mathcal{E} \rightarrow \mathcal{D}$, pour un événement de sortie $ES(cs_{apache}, cs_{var_www}, file : read)$ qui apparait à la date $2973 : d_{es} = date(ES) = 2973$.

3.2 Action Élémentaire

Lors de leurs exécutions, les entités actives du système effectuent des actions. Chacune de ces actions correspond à une opération élémentaire effectuée par une entité active sur une entité active ou passive. Nous appelons *action élémentaire*, une opération élémentaire exécutée par un sujet sur une entité. Une *interaction* est le type général d'action élémentaire entre deux contextes de sécurité.

Nous définissons le terme d'*interaction estampillée* comme l'instanciation d'une interaction donnée, c'est-à-dire l'apparition sur le système de ce type d'action élémentaire. Une interaction estampillée est un événement composé de deux événements particuliers : l'événement d'entrée et l'événement de sortie de l'appel système qui correspond à l'opération élémentaire. Formellement, une interaction estampillée est donc une interaction augmentée de deux dates. En nous basant sur la définition de l'interaction estampillée, nous introduisons le concept de "trace" qui représente l'exécution d'un système, c'est-à-dire l'ensemble des actions en cours ou passées.

L'objectif essentiel de cette section est de présenter la notion de flux d'information associée au concept d'interaction. En effet, chaque interaction conduit à un flux d'information qu'il s'agisse de formaliser à la fois l'expression des propriétés de sécurité ou l'analyse de la trace. C'est pourquoi nous avons besoin de manipuler deux types de flux. Les flux d'interaction sont nécessaires à l'expression des propriétés de sécurité. Les flux d'interaction estampillés permettent, également la formalisation de propriétés de sécurité et sont nécessaires pour mettre en œuvre l'analyse de la trace. Cette analyse sert notamment à la protection d'un système. La notion de flux d'information permet de prendre en compte tous les types d'interaction existants sur le système. Cependant, nous avons besoin de considérer des cas particuliers de ces flux. Nous distinguons deux types de cas particuliers de ces flux associés d'une part aux interactions de type transition et, d'autres part, à celles de type exécution. Par souci de concision, nous ne détaillerons pas ces deux cas particuliers. Cependant, nous les ferons apparaître avec une notation qui utilise les indices t (pour la transition) et x (pour l'exécution).

3.2.1 Interaction

Une *interaction* it est une action élémentaire qui représente une opération élémentaire oe effectuée par un contexte de sécurité cs_{source} sur un contexte de sécurité cs_{cible}. Elle est représentée par un ensemble d'attributs :

$$it = (cs_{source} \in CSS, cs_{cible} \in CS, oe \in \mathcal{EO})$$

Le contexte, qui *effectue* l'action, est une entité active du système (un processus) ; le contexte de sécurité cs_{source} appartient donc à l'ensemble des contextes sujets $\mathcal{CSS}$. Le contexte cible, *subissant* l'action, peut être un sujet, par exemple lors d'une transition d'un contexte source vers un

contexte cible, ou un objet, par exemple lors de la lecture d'un fichier. Le contexte cible appartient donc à l'ensemble des contextes $\mathcal{CS}$. La figure 3.3 présente l'interaction $(cs_{apache}, cs_{var\_www}, \{file : read\})$ où le processus `apache`, c'est-à-dire cs_{apache}, accède en lecture ($read$) à un fichier ($file$) ayant le contexte cs_{var_www} comme $/var/www/index.html$.

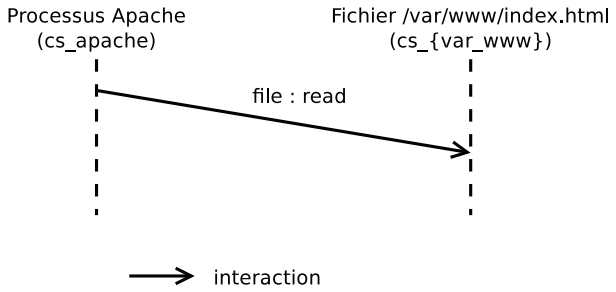

FIGURE 3.3 – Représentation de l'interaction $(cs_{apache}, cs_{var\_www}, \{file : read\})$

Nous définissons formellement une *interaction* comme suit :

Définition 3.2.1 (Interaction) *Une interaction, notée $cs_{source} \overset{oe}{\rightharpoonup} cs_{cible}$, est une action élémentaire oe effectuée par un contexte de sécurité source cs_{source} sur un contexte de sécurité cible cs_{cible}. Elle est définie comme suit :*

$$cs_{source} \overset{oe}{\rightharpoonup} cs_{cible} \quad \equiv_{def} \quad \begin{cases} it = (cs_{source} \in CSS, cs_{cible} \in CS, oe \in \mathcal{EO}), \\ cs_{source} \textit{ effectue } oe \textit{ sur } cs_{cible} \end{cases}$$

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

Nous notons $\mathcal{IT}$ 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. De plus, nous définissons trois fonctions pour les interactions :

- $src : \mathcal{IT} \rightarrow \mathcal{CSS}$ qui retourne le contexte *source* de l'interaction.
- $tgt : \mathcal{IT} \rightarrow \mathcal{CS}$ qui retourne le contexte *destination* de l'interaction.
- $op : \mathcal{IT} \rightarrow \mathcal{OE}$ qui retourne l'*opération élémentaire* liée à l'interaction.

Par exemple, pour l'interaction $it = (cs_{apache}, cs_{var\_www}, \{file : read\})$, la fonction $src(it)$ retourne le contexte cs_{apache}, la fonction $tgt(it)$ retourne le contexte cs_{var_www} et la fonction $op(it)$ retourne l'opération élémentaire $file : read$. Ces trois fonctions sont essentielles pour définir les cas particuliers d'interaction mais également les actions complexes. De plus, elles sont une brique essentielle dans la déclaration de propriétés de sécurité comme l'intégrité des domaines pour lesquels il est nécessaire de vérifier la source ou la cible d'une interaction.

3.2.2 Interaction Estampillée

Une *interaction estampillée* est un macro-événement dont on peut distinguer deux sous événements particuliers qui se succèdent : l'événement d'entrée d'un appel système et l'événement de sortie de ce même appel système. Le couple composé de l'événement d'entrée $EE(cs_{source}, cs_{cible}, oe)$ et de l'événement de sortie $ES(cs_{source}, cs_{cible}, oe)$ donne l'interaction estampillée $(cs_{source}, cs_{cible}, oe, d_{ee}, d_{es})$ avec $d_{ee} = date(EE)$ et $d_{es} = date(ES)$.

En pratique, une interaction estampillée est une occurrence d'une interaction augmentée de deux dates extraites de l'événement d'entrée et de sortie de l'appel système. Une interaction estampillée correspond à l'opération élémentaire effectuée par l'interaction. Ce concept est fondamental afin de modéliser de manière précise les actions se déroulant et s'étant déroulées sur le système : la trace. Pour autant, ce n'est pas une remise en cause des interactions "classiques" qui sont essentielles pour la formalisation et l'expression de propriétés de sécurité comme nous le verrons par la suite. En pratique, la différence entre interaction et interaction estampillée est que, dans un cas, il s'agit d'actions potentielles et, dans l'autre cas, d'actions présentes dans la trace. Les interactions servent essentiellement à formaliser les propriétés requises. Les interactions estampillées peuvent à la fois formaliser les propriétés requises et mettre en oeuvre une méthode d'analyse de la trace.

Par exemple, la figure 3.4 présente l'interaction estampillée $(cs_{apache}, cs_{var\_www}, file : read, 2819, 2973)$ où le processus apache, c'est-à-dire cs_{apache}, accède en lecture $(read)$ à un fichier $(file)$ au contexte cs_{var_www} comme $/var/www/index.html$ avec le début de l'appel système qui commence à la date `2819` et finit à la date `2973`.

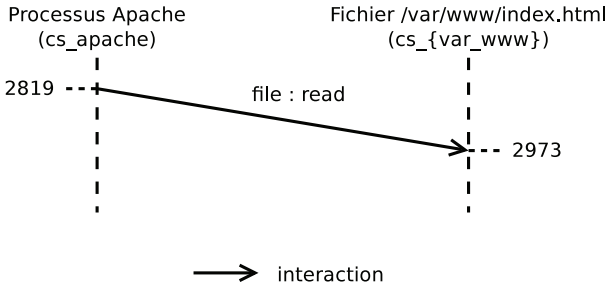

FIGURE 3.4 – Représentation de l'interaction $(cs_{apache}, cs_{var\_www}, file : read, 2819, 2973)$

Nous définissons formellement une *interaction estampillée* comme suit :

Définition 3.2.2 (Interaction Estampillée) *Une interaction estampillée, notée* $cs_{source} \underset{[d_{ee},d_{es}]}{\overset{oe}{\searrow}} cs_{cible}$ *est une action élémentaire* oe *qui commence à la date* d_{ee} *et finit à la date* d_{es}*, effectuée par un contexte de sécurité source* cs_{source} *sur un contexte de sécurité cible* cs_{cible}*. Elle est définie comme suit :*

$$cs_{source} \underset{[d_{ee},d_{es}]}{\overset{oe}{\searrow}} cs_{cible} \equiv_{def} \begin{cases} it = (cs_{source} \in CSS, cs_{cible} \in CS, oe \in \mathcal{EO}, d_{ee} \in \mathcal{D}, d_{es} \in \mathcal{D}), \\ cs_{source} \textit{ effectue } oe \textit{ sur } cs_{cible} \textit{ commençant à } d_{ee} \textit{ et finissant à } d_{es} \end{cases}$$

Par exemple, l'action élémentaire de type "interaction estampillée" $it = (cs_{apache}, cs_{var\_www}, \{file : read\}, 2819, 2973)$ se note $cs_{apache} \underset{[2819,2973]}{\overset{file:read}{\searrow}} cs_{var\_www}$.

Nous notons $\mathcal{ITE}$ l'ensemble de toutes les interactions estampillées qu'il est possible de définir en prenant toutes les paires de contextes de sécurité et toutes les opérations élémentaires possibles entre ces paires, ainsi que toutes les paires de date. De plus, nous définissons cinq fonctions pour les interactions :

- $deb : \mathcal{ITE} \rightarrow \mathcal{D}$ qui retourne la *date de début* de l'interaction estampillée.
- $fin : \mathcal{ITE} \rightarrow \mathcal{D}$ qui retourne la *date de fin* de l'interaction estampillée.
- $src : \mathcal{ITE} \rightarrow \mathcal{CSS}$ qui retourne le contexte *source* de l'interaction estampillée.
- $tgt : \mathcal{ITE} \rightarrow \mathcal{CS}$ qui retourne le contexte *destination* de l'interaction estampillée.
- $op : \mathcal{ITE} \rightarrow \mathcal{OE}$ qui retourne l'*opération élémentaire* liée à l'interaction estampillée.

Par abus de notation, nous noterons "_" une date de début ou de fin qui n'est pas explicitement définie ou utilisée. Par exemple, les notations $cs_{source} \underset{[d_{ee},\_]}{\overset{oe}{\searrow}} cs_{cible}$ $cs_{source} \underset{[\_,d_{es}]}{\overset{oe}{\searrow}} cs_{cible}$ ou

$cs_{source} \overset{oe}{\underset{[\_,\_]}{\rightarrow}} cs_{cible}$ sont respectivement utilisées quand la date d'entrée (d_{ee}) n'est pas définie, la date de sortie (d_{es}) n'est pas définie ou qu'aucune des dates n'est définie.

3.2.3 Traces

Une `trace` observée par le noyau fait référence à un ensemble d'interactions estampillées d'un système d'exploitation.

Définition 3.2.3 (Trace) *Une trace T est un ensemble de n interactions estampillée, $it_n \in \mathcal{ITE}$, représentant l'exécution d'un système.*

$$T \quad \equiv_{def} \quad it_i \in \mathcal{ITE} | i \in 1..n \tag{3.3}$$

Par exemple, la trace T, représentée sur la figure 3.5, est composée de 3 interactions estampillées it_1, it_2, it_3 :

$$T \Leftrightarrow \begin{cases} it_1 = cs_{apache} \overset{file:read}{\underset{[2458,2489]}{\rightarrow}} cs_{apache\_conf} \\ it_2 = cs_{apache} \overset{file:execute}{\underset{[2501,2514]}{\rightarrow}} cs_{php\_bin} \\ it_3 = cs_{apache} \overset{process:transition}{\underset{[2576,2637]}{\rightarrow}} cs_{php} \end{cases} . \tag{3.4}$$

Cette trace représente l'état d'un système où le processus apache (cs_{apache}) a lu ($read$) le fichier ($file$) de configuration d'apache (cs_{apache_conf}). Cette action de lecture a commencé à la date `2458` et a fini à `2489`. Puis, ce même processus a exécuté ($execute$) un fichier ($file$) contenant le binaire relatif à la commande php labellé cs_{php_bin}. Cette action d'exécution a commencé à la date `2501` et a fini à la date `2514`. Finalement, la troisième interaction de la trace exprime le fait que, suite à l'exécution du binaire cs_{php_bin}, le processus apache a transité depuis son contexte cs_{apache} vers un second contexte cs_{php} correspondant à l'espace d'exécution des programmes PHP. Cette action de transition a commencée à la date `2576` et a fini à `2637`.

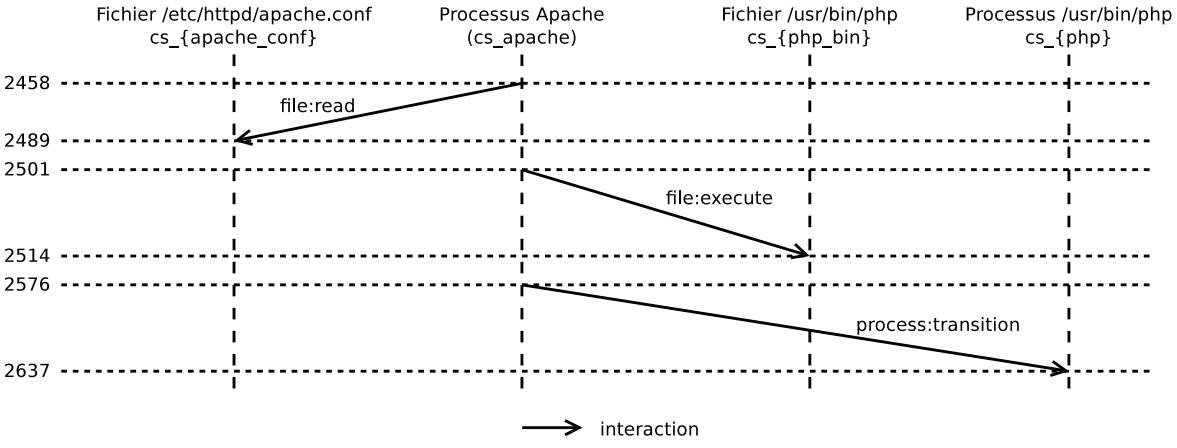

FIGURE 3.5 – Représentation de la trace T

Chaque fois qu'une interaction it_i a lieu sur le système, avant même d'être analysée pour réaliser une protection ou une détection d'intrusion, la trace correspondante au système est mise à jour, c'est-à-dire, T_{i-1} devient $T_i \leftarrow T_{i-1} \cup it_i$. En pratique, la trace T correspond à l'historique

des actions passées sur le système et permet de prendre des décisions en ne se limitant pas à la connaissance unique de l'événement courant. Dans le cas particulier de la protection, l'interaction courante it_i est ajoutée à la trace au moment où l'événement d'entrée est créé dans l'appel système correspondant. La trace est ensuite mise à jour au moment de la création de l'événement de sortie de l'appel système. Si l'interaction a été annulée par le noyau en raison d'une rupture de propriété de sécurité, l'interaction it_i est retirée de la trace T, sinon elle y est conservée. En effet, si l'interaction est annulée par le noyau, cela signifie qu'elle n'a pas eu d'effet sur le système et qu'elle ne doit pas apparaître dans l'historique des appels système. Un historique plus complet contenant également les appels système refusés pourrait être construit pour mettre en œuvre des mécanismes d'audit ou de forensics. Dans le cas d'une protection ou d'une détection d'intrusion, seuls les appels ayant aboutis ont pu accéder ou modifier de l'information et sont utiles pour modéliser le système courant.

3.2.4 Flux d'Information Direct

Quand deux entités interagissent, la conséquence est un flux d'information direct entre ces deux entités. Lorsque l'opération est de type écriture alors l'information va du contexte source vers le contexte cible. Inversement pour la lecture, l'information va de la cible vers la source.

Nous proposons une notation pour le flux d'information qui permet de modéliser à la fois les flux de type lecture et écriture.

Définition 3.2.4 (Flux d'Information Direct) *Une interaction* $(cs_1 \overset{oe}{\rightharpoonup} cs_2)$ *donne lieu à un flux d'information direct entre ces deux contextes* cs_1 *et* cs_2 *noté* $\blacktriangleright$. *Dans le cas d'une opération* $eo \in \mathcal{WEO}$, *le flux est noté* $cs_1 \blacktriangleright cs_2$ *et, dans celui d'une opération* $eo \in \mathcal{REO}$, *le flux est noté* $cs_2 \blacktriangleright cs_1$.

La figure 3.6 présente une interaction de type lecture, $cs_{apache} \overset{file:read}{\rightharpoonup} cs_{var\_www}$, qui est modélisée par le flux d'information direct $cs_{var\_www} \blacktriangleright cs_{apache}$. On voit, comme l'indique le schéma de la figure 3.6, que la flèche représentant le flux d'information, indique le sens du transfert d'information.

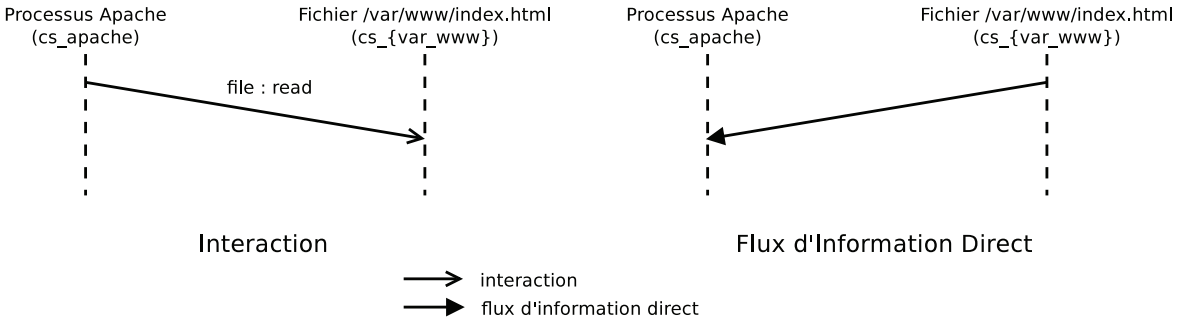

FIGURE 3.6 – Représentation du flux d'information $(cs_{apache}, cs_{var\_www}, file : read)$

Suivant la définitions 3.2.4, nous introduisons l'ensemble $\mathcal{FIDS}$ qui regroupe tous les flux d'information.

Comme nous l'avons précisé en introduction, nous sommes obligés de distinguer deux cas particuliers de flux d'information : la transition et l'exécution. Pour la transition d'un contexte cs_{source} vers un contexte cible cs_{cible}, nous noterons le flux d'information de façon indicé par t tel que $cs_{source} \blacktriangleright_t cs_{cible}$. Pour l'exécution par un contexte cs_1 d'un contexte cs_2 , nous noterons le flux d'information de façon indicée par x tel que $cs_1 \blacktriangleright_x cs_2$.

3.2.5 Flux d'Information Direct Estampillé

Dans le paragraphe précédent, nous avons associé les interactions aux flux d'information directs. Sur le même principe, nous associons maintenant les interactions estampillés aux *Flux d'Information Directs Estampillés*.

Définition 3.2.5 (Flux d'Information Estampillé Direct $\overset{T}{\blacktriangleright}_{[d_{ee},d_{es}]}$) *Une interaction estampillée $it = (cs_1 \overset{oe}{\underset{[d_{ee},d_{es}]}{\searrow}} cs_2)$ qui provient d'une trace T donne lieu à un flux d'information direct estampillé entre les deux contextes cs_1 et cs_2, noté $\overset{T}{\underset{[d_{ee},d_{es}]}{\blacktriangleright}}$. Dans le cas d'une opération $eo \in \mathcal{WEO}$, le flux est noté $cs_1 \overset{T}{\underset{[d_{ee},d_{es}]}{\blacktriangleright}} cs_2$ et, dans celui d'une opération $eo \in \mathcal{REO}$, le flux est noté $cs_2 \overset{T}{\underset{[d_{ee},d_{es}]}{\blacktriangleright}} cs_1$.*

Par exemple, la figure 3.7 présente une interaction de type lecture $cs_{apache} \overset{file:read}{\underset{[2819,2973]}{\searrow}} cs_{var\_www}$ correspondant au flux d'information direct $cs_{var\_www} \overset{T}{\underset{[2819,2973]}{\blacktriangleright}} cs_{apache}$.

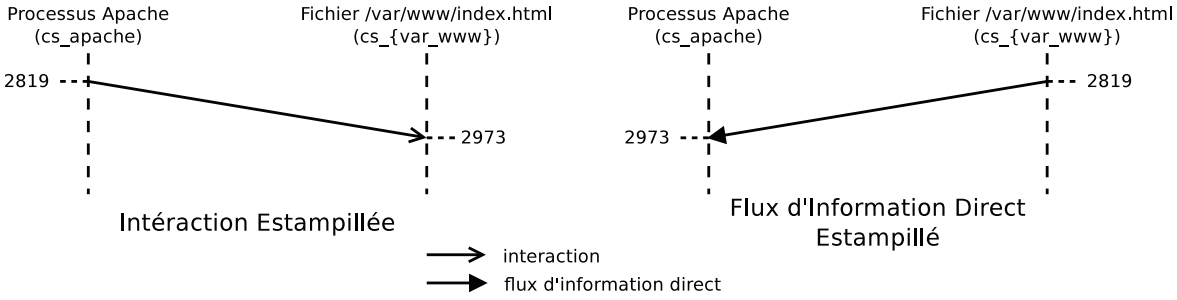

FIGURE 3.7 – Représentation du flux d'information direct estampillé $(cs_{apache}, cs_{var\_www}, file : read, 2819, 2973)$

Suivant la définition 3.2.5, nous introduisons l'ensemble $\mathcal{FIDSE}$ pour regrouper tous les flux d'information ayant eu lieu sur le système.

Nous définissons deux fonctions qui permettent de connaitre les dates de début et de fin d'un flux d'information direct estampillé :

- $deb : \mathcal{FIDSE} \rightarrow \mathcal{D}$ qui retourne la date de début du flux.
- $fin : \mathcal{FIDSE} \rightarrow \mathcal{D}$ qui retourne la date de fin du flux.

Nous noterons la transition estampillée $cs_{source} \overset{T}{\underset{[d_{ee},d_{es}]}{\blacktriangleright_t}} cs_{cible}$, et l'exécution estampillée $cs_1 \overset{T}{\underset{[d_{ee},d_{es}]}{\blacktriangleright_x}} cs_2$.

3.2.6 Flux d'Information Direct Multiple

Nous proposons un moyen de factoriser les flux d'information estampillés. Nous le faisons pour les flux d'information et, pour leurs cas particuliers, les transitions.

Nous avons défini, dans la section 3.2.5, le concept de flux d'information direct estampillé qui peut être qualifié d'élementaire. En effet, ce flux d'information estampillé reflète l'action provoquée par une seule et unique interaction estampillée. Nous introduisons ici le concept de flux d'information direct multiple qui regroupe un ensemble de flux d'information estampillés pour le

même type de flux entre les deux mêmes contextes. Un flux d'information direct multiple regroupe donc plusieurs flux de même nature entre deux mêmes contextes en un flux global entre ces deux contextes.

Définition 3.2.6 (Flux d'Information Direct Multiple $\overset{T}{\blacktriangleright}^{+}$) *A partir d'une trace système T, plusieurs flux d'information de même nature allant d'un contexte cs_1 vers un contexte cs_k avec un premier flux commençant à date d_{ee_1} et un dernier flux finissant à une date d_{es_k} correspondent à un flux d'information direct multiple noté $css \overset{T}{\underset{[d_{ee_1},d_{es_k}]}{\blacktriangleright}^{+}} cs$ et qui est défini par :*

$$cs_1 \overset{T}{\underset{[d_{ee_1},d_{es_k}]}{\blacktriangleright}^{+}} cs_k \overset{def}{\equiv} \left(\begin{array}{c} \exists k, k > 1, \forall i \in [1..k], \\ \bigwedge\left(cs_1 \overset{T}{\underset{[d_{ee_i},d_{es_i}]}{\blacktriangleright}} cs_k \wedge d_{ee_1} \leq d_{ee_i} \leq d_{es_i} \leq d_{es_k}\right) \end{array} \right)$$

La figure 3.8 présente trois flux directs estampillés allant de cs_{var_www} vers cs_{apache}. Du point de vue du transfert d'information, il est possible de considérer que le transfert d'information a commencé avec le début du premier flux d'information à la date `2819` et a fini avec la fin du troisième flux à la date `3254`. Le flux d'information direct multiple résultant de ces trois flux d'information direct estampillés est noté $sc_{var\_www} \overset{T}{\underset{[2819,3254]}{\blacktriangleright}^{+}} sc_{apache}$ et est décrit dans la figure 3.8.

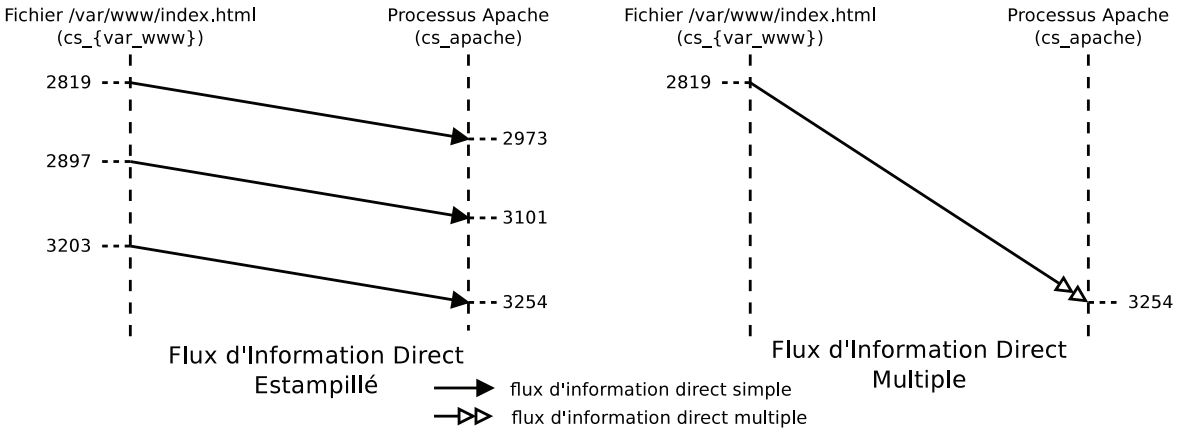

FIGURE 3.8 – Représentation du flux d'information direct multiple $(cs_{apache}, cs_{var\_www}, file : read, 2819, 3254)$

En pratique, cette notation permet de regrouper plusieurs flux en une représentation compacte qui permet de sur-approximer un ensemble de flux de même nature. Ceci est essentiel dans le cadre d'une protection dynamique car elle permet de limiter la taille de la trace et donc son empreinte mémoire. Pour les propriétés qui requièrent uniquement des flux directs, cette notion de flux multiple n'est pas utile.

Nous avons besoin de distinguer un cas particulier de flux d'information direct multiple associé aux transitions que nous notons, avec l'indice t, $ssc \overset{T}{\underset{[d_{ee_1},d_{es_k}]}{\blacktriangleright_t^{+}}} tsc$. Dans ce cas, tous les flux considérés sont de type transition.

3.2.7 Flux d'Information Direct Général

Nous combinons les définitions des flux d'information directs simples (section 3.2.5) et multiples (section 3.2.6) pour introduire le concept de **flux d'information direct général**.

Un flux d'information direct général existe entre deux contextes de sécurité ssc s'il existe un flux d'information direct simple ou multiple entre ces deux contextes. De manière plus précise, il faut noter qu'un flux d'information existe entre deux contextes s'il existe au moins un flux d'information direct simple entre eux.

Définition 3.2.7 (Flux d'Information Direct Général) *Soit T une trace système, un flux d'information direct général allant d'un contexte de sécurité ssc vers un contexte de sécurité tsc commençant à la date d_{ee} et finissant à la date d_{es} existe si il existe un flux direct simple ou multiple de ssc vers tsc à ces dates. Ce flux d'information général est noté $ssc \overset{T}{\underset{[d_{ee},d_{es}]}{\triangleright}} tsc$ et est défini comme suit :*

$$ssc \overset{T}{\underset{[d_{ee},d_{es}]}{\triangleright}} tsc \overset{def}{\equiv} \left(\left((ssc \overset{T}{\underset{[d_{ee},d_{es}]}{\blacktriangleright}} tsc) \vee (ssc \overset{T}{\underset{[d_{ee},d_{es}]}{\blacktriangleright^{+}}} tsc)\right) \right)$$

De la même manière que pour les transitions, nous avons la notation $sc_1 \overset{T}{\underset{[s_i,e_j]}{\triangleright_t}} sc_2$.

3.3 Flux d'Information Indirect

3.3.1 Relation de dépendance causale

Le principe de causalité est que "la cause précède l'effet". Dans le cadre d'un système, une condition nécessaire pour avoir une dépendance causale entre deux événements est que le premier événement précède temporellement le second. Cependant, cette condition temporelle n'est pas suffisante. En effet, A peut précéder temporellement B, sans que l'effet de A sur le système change le comportement et les effets de B.

En pratique, sur un système tel que formalisé ici, il n'est pas possible de tenir compte uniquement des aspects temporels. Pour deux interactions, $cs_1 \overset{ecrire}{\underset{[d_{ee_i},d_{es_i}]}{\rightharpoonup}} cs_2$ et $cs_3 \overset{lire}{\underset{[d_{ee_j},d_{es_j}]}{\rightharpoonup}} cs_2$, nous considérons qu'il y a dépendance causale entre ces deux interactions en estimant que l'écriture a eu un effet sur la lecture du fait du contexte partagé cs_2, en plus de la dépendance temporelle $d_{ee_i} \leq d_{es_j}$. Quelles que soient les définitions proposées par différents auteurs, la dépendance causale est généralement une surestimation de la relation de causalité.

Dans notre modélisation, une interaction it_1 précède causalement une seconde interaction it_2 si deux conditions sont respectées : $deb(it_1) \leq fin(it_2)$ mais aussi si it_1 et it_2 partagent un même contexte. S'il y a relation de causalité, nous considérons que it_1 est une "cause possible" de l'apparition ou du changement d'effet de it_2. Bien sûr, cela n'implique aucunement que it_1 soit l'unique cause de it_2. Il peut toujours y avoir plusieurs causes possibles pour it_2, il s'agit de toutes les interactions présentes dans la trace et qui respectent les deux conditions de la dépendance causale.

Mais ces deux conditions ne sont pas suffisantes et mènent à un grand nombre de faux positifs. Par exemple, avec $cs_1 \overset{lire}{\underset{[d_{ee_i},d_{es_i}]}{\rightharpoonup}} cs_2$ et $cs_3 \overset{lire}{\underset{[d_{ee_i},d_{es_i}]}{\rightharpoonup}} cs_2$, une relation de causalité existe entre ces deux interactions d'après la définition informelle que nous avons donné. Or, dans la réalité, ce n'est pas le cas puisque la première interaction ne modifie pas l'état du contexte partagé cs_2. Nous reprenons la définition donnée par [Briffaut, 2007]. Maisn contrairement à [Briffaut, 2007], nous

considérons les flux estampillés (et pas les interactions) dans notre définition afin de donner une notion de flux d'information indirect qui repose sur les traces.

Définition 3.3.1 *Nous définissons la notion de dépendance causale, notée $\twoheadrightarrow$, comme :*

$$it_1 \twoheadrightarrow it_2 \quad \equiv_{def} \quad \begin{cases} src(it_1) = cs_1, \\ dst(it_1) = src(it_2) = cs_2, \\ dst(it_2) = cs_3, deb(it_1) \leqslant fin(it_2), \\ cs_1 \overset{T}{\underset{[\_,\_]}{\triangleright}} cs_2 \wedge cs_2 \overset{T}{\underset{[\_,\_]}{\triangleright}} cs_3 \end{cases}$$

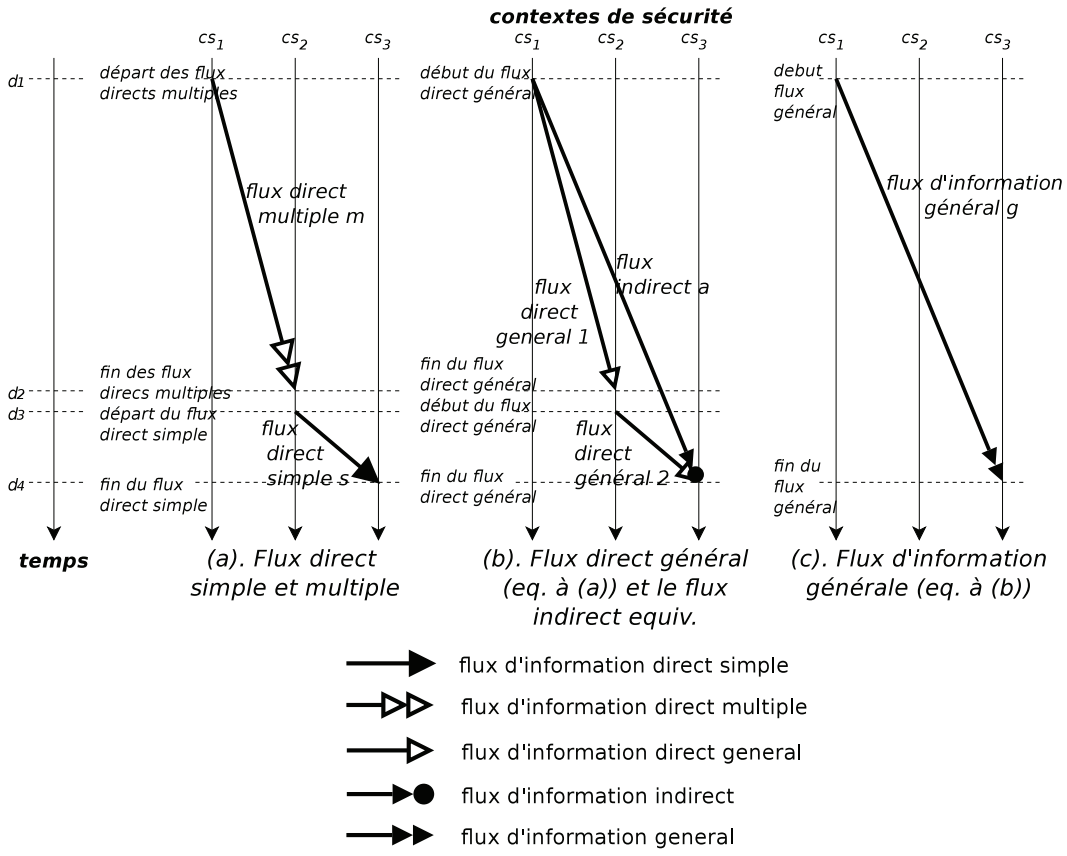

FIGURE 3.9 – Flux d'Information simple et multiple et flux d'information indirects correspondants

3.3.2 Définition

Nous introduisons la définition formelle d'un flux d'information indirect qui permet de transférer de l'information de façon transitive :

Définition 3.3.2 (Flux d'Information Indirect) *Suivant une trace système T, un flux d'information indirect (ou flux d'information transitif) entre un contexte cs_1 et un second contexte cs_k commençant à la date d_{ee_1} et finissant à la date d_{es_k} est noté $cs_1 \overset{T}{\underset{[d_{ee_1}, d_{es_k}]}{\trianglerighteq}} cs_k$, et est défini comme*

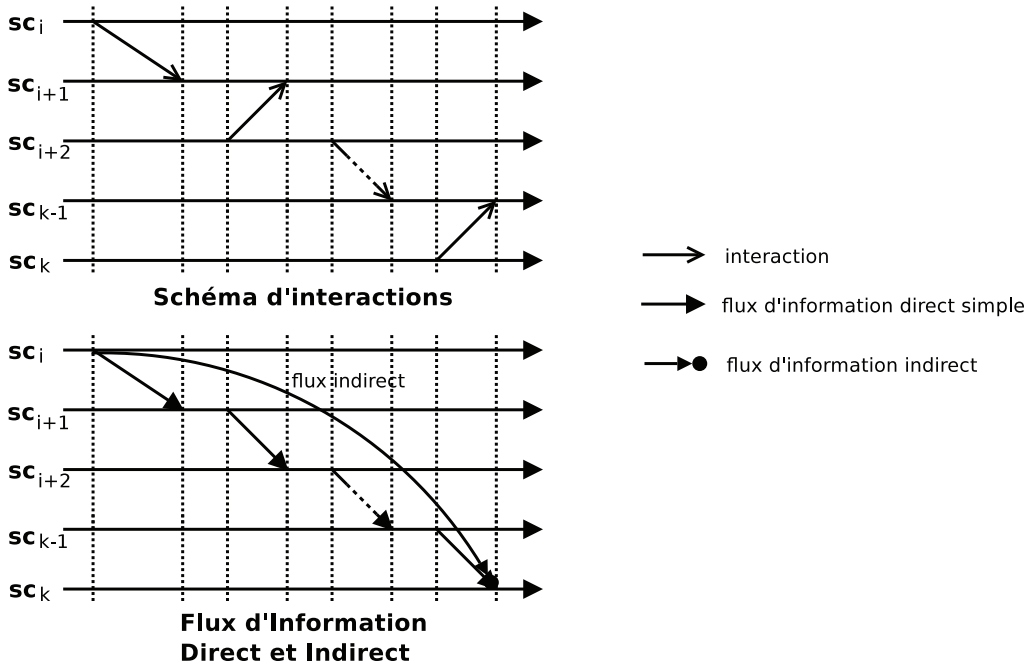

FIGURE 3.10 – Interactions de type Écriture puis Lecture / Flux d'information Indirect

suit :

$$cs_1 \overset{T}{\underset{[d_{ee_1},d_{es_k}]}{\Rrightarrow}} cs_k \overset{def}{\equiv} \left(\begin{array}{c} \exists k \in [3..+\infty], \forall i \in [1..k-2], cs_i \in \mathcal{SC} \\ (cs_i \overset{T}{\underset{[d_{ee_i},d_{es_i}]}{\triangleright}} cs_{i+1}) \wedge (cs_{i+1} \overset{T}{\underset{[d_{ee_{i+1}},d_{es_{i+1}}]}{\triangleright}} cs_{i+2}) \\ \wedge (d_{ee_i} \leq d_{es_{i+1}}) \end{array} \right),$$

On voit que, par définition, $cs_1 \overset{T}{\underset{[d_{ee_1},d_{es_k}]}{\Rrightarrow}} cs_k \Rightarrow \exists k \in [3..+\infty], \forall i \in [1..k-2], it_i \twoheadrightarrow it_{i+1}$: le flux indirect est donc bien une fermeture transitive de $k-1$ interactions estampillées causalement liées. Par la suite, nous notons $\mathcal{FII}$ l'ensemble des flux d'information indirects.

La figure 3.11 présente tous les cas d'ordonnancement temporel entre deux interactions $A = cs_i \overset{T}{\underset{[d_{ee_i},d_{es_i}]}{\triangleright}} cs_{i+1}$ et $B = cs_{i+1} \overset{T}{\underset{[d_{ee_{i+1}},d_{es_{i+1}}]}{\triangleright}} cs_{i+2}$. Le seul cas (figure 3.11.(f)) où il n'y a pas de transfert possible allant de cs_i à cs_{i+2} est quand le flux B se termine avant que A ne commence. Elle correspond bien à la négation de la condition temporelle nécessaire à la dépendance $\neg(deb(A) \leq fin(B)) \Leftrightarrow deb(A) > fin(B)$. Dans le cas figure 3.11.(f), le flux B ne peut donc pas faire transférer de l'information venant du flux A puisque A débute après B.

La figure 3.12 présente deux flux directs estampillés. $cs_{apache} \overset{T}{\underset{[4578,4589]}{\triangleright}} cs_{var\_www}$ correspond à un flux d'information direct estampillé qui va du processus `apache` vers le fichier `/var/www/index.html` et qui commence à la date 4578 et pour finir à 4589.
$cs_{php} \overset{T}{\underset{[4605,4623]}{\triangleright}} cs_{var\_www}$ correspond à un flux d'information allant du fichier `/var/www/index.html` vers le processus `php` commençant à la date 4605 et finissant à 4623. La combinaison de ces deux flux permet de créer le flux transitif allant du processus `apache` vers le processus `php` : $cs_{apache} \overset{T}{\underset{[4578,4623]}{\Rrightarrow}} cs_{php}$.

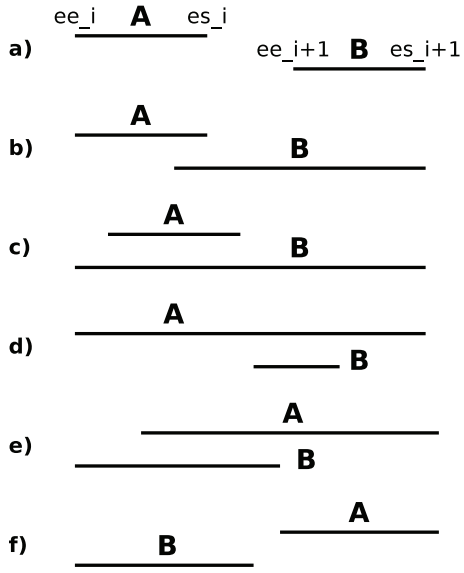

FIGURE 3.11 – Tous les cas d'ordonnancement temporel entre A et B

De manière similaire, nous notons les transitions indirectes : $cs_1 \overset{T}{\underset{[d_{ee_1},d_{es_k}]}{\Rrightarrow_t}} cs_k$.

3.4 Flux d'Information Général

Nous introduisons un opérateur qui généralise les flux d'information en combinant les définitions 3.3.2 (pour le flux d'information direct général) et 3.2.7 (pour le flux d'information indirect). En pratique, un flux d'information général existe s'il existe un flux entre deux contextes de sécurité, c'est-à-dire, s'il existe un flux d'information direct général ou un flux indirect entre ces deux contextes.

Définition 3.4.1 (General Information Flow) *Pour une trace système T, un flux d'information entre un contexte cs_1 et un autre contexte cs_k, commençant à une date d_{ee} et finissant à une date d_{es}, est noté $cs_1 \overset{T}{\underset{[d_{ee},d_{es}]}{\Rrightarrow}} cs_k$ et est défini formellement comme suit :*

$$cs_1 \overset{T}{\underset{[d_{ee},d_{es}]}{\Rrightarrow}} cs_k \overset{def}{\equiv} \left((cs_1 \overset{T}{\underset{[d_{ee},d_{es}]}{\triangleright}} cs_k) \vee (cs_1 \overset{T}{\underset{[d_{ee},d_{es}]}{\Rrightarrow}} cs_k) \right)$$

La figure 3.9.(c). présente un exemple de flux d'information général : $sc_1 \overset{T}{\underset{[d_1,d_4]}{\Rrightarrow}} sc_3$ $(flow_g)$. Ce flux est équivalent au flux d'information indirect $(flow_a)$ de la figure 3.9.(b).

Respectivement, nous notons $sc_1 \overset{T}{\underset{[d_{ee},d_{es}]}{\Rrightarrow_t}} sc_k$ la transition générale.

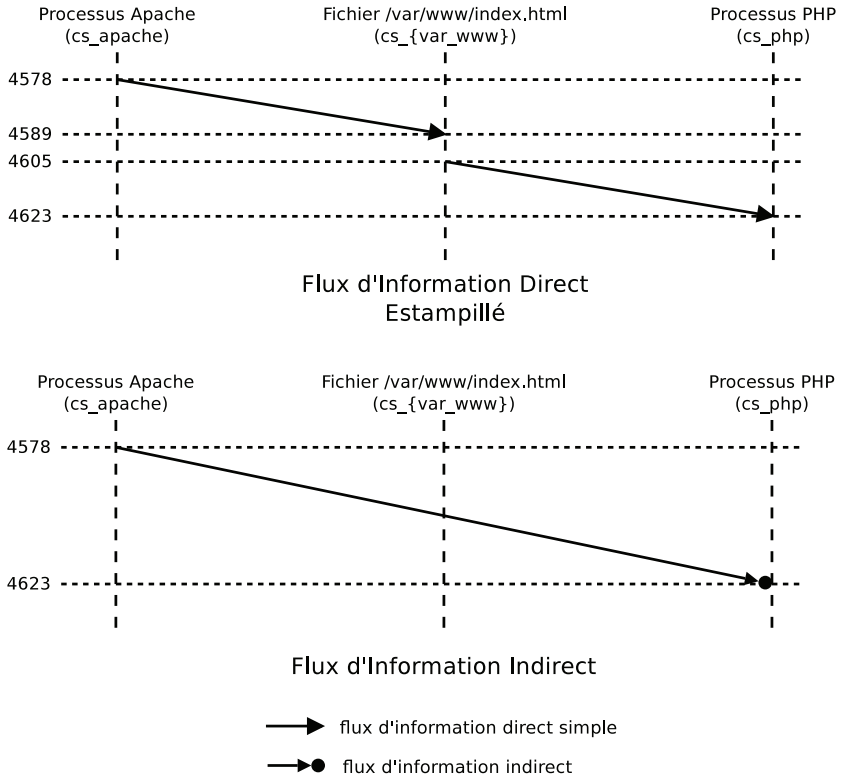

FIGURE 3.12 – Représentation du flux indirect de cs_{apache} à cs_{php}

3.5 Langage de description d'activités

Une activité système peut être exprimée au moyen des différents opérateurs que nous venons de définir. Ces opérateurs permettent de manipuler les notions d'interactions, de flux et de transitions de façon directs et indirects.

Nous proposons ainsi un langage général d'expressions d'activité qui est à la fois utilisé pour exprimer les propriétés de sécurité et pour analyser la trace.

Les terminaux peuvent être intégrés comme de nouveaux opérateurs dans un langage existant. Ce langage autorise également la manipulation des dates associées à la trace. Ainsi, il est possible de définir des contraintes temporelles entre les flux d'information. Nous verrons dans le chapitre suivant que ces opérateurs permettent l'expression formelle de propriétés de sécurité précises.

activité	::= [description " = "] correlation
corrélation	::= (corrélation $\wedge$ correlation)|(correlation $\vee$ correlation)|terminal
terminal	::= interaction|interaction_estampillé|flux_direct
terminal	::= terminal|flux_direct_estampillé|flux_direct_multiple_estampillé
terminal	::= terminal|flux_direct_general|flux_indirect|flux_general
terminal	::= terminal|transition_directe|transition_directe_estampillée
terminal	::= terminal|transition_directe_multiple_estampillée
terminal	::= terminal|transition_directe_generale|transition_indirecte
terminal	::= terminal|transition_general|exécution|exécution_estampillée
interaction	::= $cs \overset{eo}{\frown} cs$
interaction_estampillé	::= $cs \underset{[d_{ee},d_{es}]}{\overset{eo}{\frown}} cs$
flux_direct	::= $cs \blacktriangleright cs$
flux_direct_estampillé	::= $cs \underset{[d_{ee},d_{es}]}{\overset{T}{\blacktriangleright}} cs$
flux_direct_multiple_estampillé	::= $cs \underset{[d_{ee_1},d_{es_k}]}{\overset{T}{\blacktriangleright^{+}}} cs$
flux_direct_général	::= $cs \underset{[d_{ee},d_{es}]}{\overset{T}{\triangleright}} cs$
flux_indirect	::= $cs \underset{[d_{ee_1},d_{es_k}]}{\overset{T}{\blacktriangleright\!\!\blacktriangleright}} cs$
flux_général	::= $cs \underset{[d_{ee},d_{es}]}{\overset{T}{\triangleright\!\!\triangleright}} cs$
transition_directe	::= $cs \blacktriangleright_t cs$
transition_directe_estampillée	::= $cs \underset{[d_{ee},d_{es}]}{\overset{T}{\blacktriangleright_t}} cs$
transition_directe_multiple_estampillée	::= $cs \underset{[d_{ee_1},d_{es_k}]}{\overset{T}{\blacktriangleright_t^{+}}} cs$
transition_directe_générale	::= $cs \underset{[d_{ee},d_{es}]}{\overset{T}{\triangleright_t}} cs$
transition_indirecte	::= $cs \underset{[d_{ee_1},d_{es_k}]}{\overset{T}{\blacktriangleright\!\!\blacktriangleright_t}} cs$
transition_générale	::= $cs \underset{[d_{ee},d_{es}]}{\overset{T}{\underset{t}{\triangleright\!\!\triangleright}}} cs$
exécution	::= $cs \blacktriangleright_x cs$
exécution_estampillée	::= $cs \underset{[d_{ee},d_{es}]}{\overset{T}{\blacktriangleright_x}} cs$
cs	::= " contexte de sécurité "
eo	::= " operation élémentaire "
$description$	::= " nom ou type de l'activité "

Chapitre 4

Formalisation de Propriétés de Sécurité

4.1 Introduction

Nous allons montrer dans ce chapitre comment notre langage permet de formaliser un large ensemble de propriétés de sécurité. Il s'agit d'une approche originale, puisqu'au moyen du même langage, nous pouvons exprimer des propriétés aussi variées que des modèles de protection statique, des modèles dynamiques et définir des nouveaux modèles à la fois statiques et dynamiques. Nous commencerons par présenter une propriété simple, l'exécution par transition, dont l'intérêt est d'illustrer l'efficacité de notre langage pour prendre en compte les flux indirects et la dynamicité du système. Ensuite, nous présentons un large éventail de propriétés d'intégrité. Nous nous contentons d'une seule propriété de confidentialité et d'un modèle multi-niveaux. Par la suite, nous détaillons trois propriétés d'abus de privilèges qui peuvent être nécessaire aussi bien à la confidentialité que l'intégrité. Finalement, nous présentons deux modèles dynamiques : un extrait de la littérature, la muraille de Chine ; et un original, le confinement des données.

Nous verrons dans ce chapitre la nécessité de pouvoir formaliser librement des propriétés de sécurité. Nous avons décidé de présenter ici uniquement un sous ensemble des propriétés qui peuvent être supportées par notre langage. En effet, ce ne peut être qu'après une définition précise des objectifs de sécurité que l'on peut s'attacher à définir les propriétés requises. Par contre, nous verrons à travers les exemples choisis que notre langage permet de formaliser simplement ces propriétés.

4.2 Exécution par Transition

L'exécution par transition peut être assimilée à une propriété de sécurité. Elle est composée d'une séquence de transition suivie d'une exécution directe. La seule relation existante entre les deux actions est une relation de succession temporelle.

4.2.1 Exécution Indirecte

Définition 4.2.1 (Exécution Indirecte) *A partir d'une trace système T donnée, une exécution indirecte effectuée par un sujet cs_1 sur un objet cso, commençant à la date d_{ee_1} et finissant à la date d_{es_k} est notée $cs_1 \overset{T}{\underset{[d_{ee_1},d_{es_k}]}{\Rrightarrow_x}} cso$ et est définie comme suit :*

$$cs_1 \overset{T}{\underset{[d_{ee_1},d_{es_k}]}{\Rrightarrow_x}} cso \overset{def}{\equiv} \left(\begin{array}{c} \exists j,k \in \mathbb{D}, 1 \leq j \leq k, \\ (cs_1 \overset{T}{\underset{[d_{ee_j},d_{es_j}]_t}{\Rrightarrow}} cs_k) \wedge (cs_k \overset{T}{\underset{[d_{ee_k},d_{es_k}]}{\blacktriangleright_x}} cso) \ \wedge \ (d_{es_j} \leq d_{ee_k}) \end{array} \right).$$

La figure 4.1 présente une transition et une exécution. La transition correspond à la transition du processus `apache` vers le contexte du processus `php` qui commence à la date 6125 et pour finir à 6131. L'exécution correspond à l'exécution du fichier `/var/www/login.php` par le processus `php` commençant à la date 6245 et finissant à 6253. Il faut que la date de l'événement de sortie de la transition soit inférieure ou égale à la date de l'événement d'entrée de l'exécution. Comme $6131 \leq 6245$, l'exécution indirecte de `/var/www/login.php` par `apache` existe réellement. L'exécution indirecte est notée $cs_{apache} \overset{T}{\underset{[6125,6253]}{\Rrightarrow_x}} cs_{var\_www\_php}$. Il est important de pouvoir contrôler cette relation puisqu'elle permet de transférer des privilèges et donc de l'information du processus `apache` vers le processus exécutant le script `login.php`.

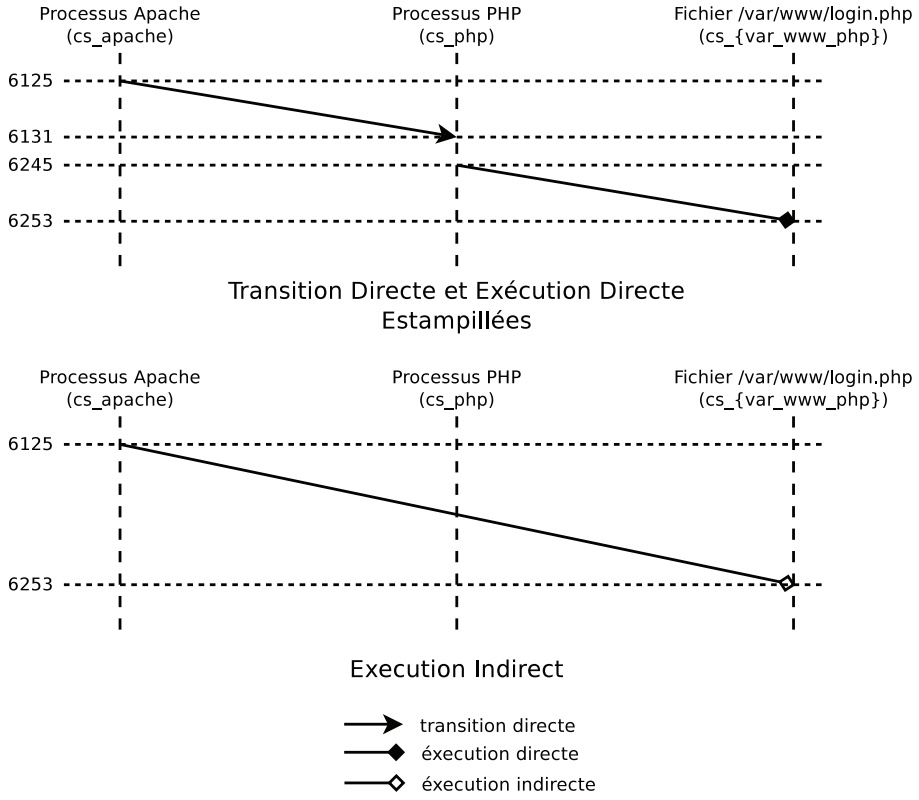

FIGURE 4.1 – Représentation de l'exécution indirecte de $cs_{var\_www\_php}$ par cs_{apache}

4.2.2 Exécution Générale

Nous combinons la définition directe (3.2.4) et indirecte (4.2.1) pour introduire l'opérateur généraliste $\overset{T}{\underset{[s_1,e_k]_x}{\Rrightarrow}}$ qui représente l'exécution générale. De manière informelle, une exécution générale existe si une exécution directe ou indirecte existe entre un sujet et un objet.

4.3 Intégrité

La classe de propriétés de sécurité de type **Intégrité** a pour but de prévenir toute modification non explicitement autorisée d'une entité du système. Cette entité peut aussi bien être un sujet qu'un objet.

Nous proposons les propriétés d'intégrité suivantes [Clemente et al., 2010] :
- Intégrité des objets : intégrité telle que définie dans ITSEC ;
- Intégrité des sujets : Non Interférence ;
- Situation de concurrence : prévenir [Rouzaud Cornabas et al., 2010] les attaques de type concurrence entre deux sujets ;
- Intégrité des domaines : intégrité d'un ensemble de contextes ;
- Exécutable de confiance : Obligation que tous les codes exécutés proviennent d'un ensemble de contextes objets dit *sûrs*.

4.3.1 Intégrité des Objets

Nous rappellons que le principe de cette propriété de sécurité (comme définie dans [ISO/IEC, 2009]) est de contrôler les modifications d'un contexte objet.

Définition 4.3.1 (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 formalisme, X et I sont des ensembles de contexte de sécurité. Plus particulièrement, I est un ensemble de contextes objets et X un ensemble de contextes sujets. La propriété revient donc à garantir qu'un contexte sujet ne puisse pas modifier un contexte objet.

Dans notre représentation, nous projetons cette propriété comme la garantie, pour une trace T, qu'un sujet css n'a pas provoqué et ne provoquera pas une action de type écriture sur un objet cso. Cette propriété est appelée **Intégrité Directe des Données** (IDD) et est définie comme suit :

Définition 4.3.2 (Intégrité Directe des Données) *Un objet $cso \in \mathcal{CSO}$ est dit intègre vis-à-vis d'un sujet $css \in \mathcal{CSS}$ pour une trace T ssi il n'existe pas de flux d'information direct général de css vers cso.*

$$IDD(T, css, cso) \stackrel{def}{\equiv} \neg(css \overset{T}{\rhd} cso)$$

Par exemple, le but de la propriété d'Intégrité Directe des Données peut être d'empêcher toute modification du fichier `/etc/shadow` par un processus utilisateur. En terme de flux d'information, le but est d'empêcher tout flux d'information partant du contexte $user\_t$ vers le contexte $shadow\_t$. La propriété instanciée s'écrit $IDD(T, user\_t, shadow\_t)$. La trace T correspondant au système est décrite dans la figure 4.2 à droite, et à gauche, elle est représentée sous la forme de flux d'information directs. L'arc $user\_t \overset{T}{\rhd} [5131, 5145]shadow\_t$ représente une rupture de la propriété $IDD(T, user\_t, shadow\_t)$. Dans le cas d'une approche par protection, l'arc correspondant est supprimé de la trace puisque l'appel système est annulé.

Par contre, cette propriété d'**Intégrité Directe des Données** ne permet pas de prévenir contre les ruptures indirectes. Par exemple, dans la figure 4.2, il existe deux flux directs $user\_t \overset{T}{\rhd} [6136, 6142]root\_t$ et $root\_t \overset{T}{\rhd} [6155, 6163]shadow\_t$ qui transitivement forment le flux d'information indirect $user\_t \underset{[6136,6163]}{\overset{T}{\rhd\!\!\rhd}} shadow\_t$. Ce flux d'information peut correspondre à une rupture de l'intégrité indirecte. Elle n'est pas prise en compte par la propriété d'**Intégrité Directe des Données**.

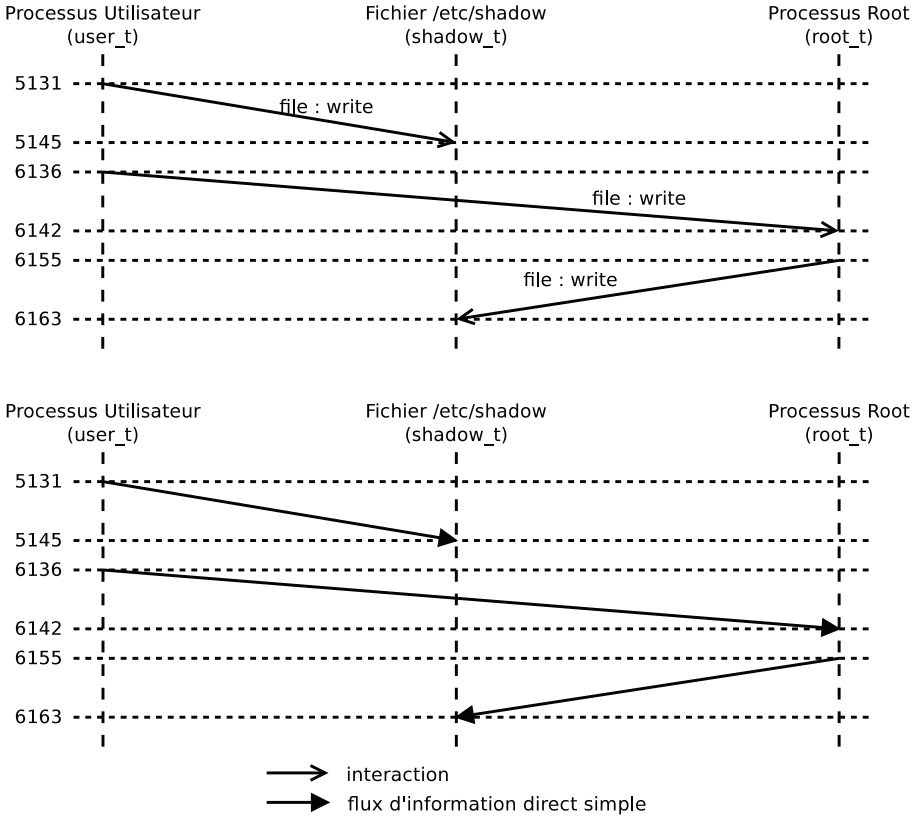

FIGURE 4.2 – Représentation d'une trace système avec rupture d'intégrité directe et indirecte

Pour répondre à cette limitation, nous proposons une seconde propriété : **Intégrité Indirecte des Données** (IID) qui permet de garantir l'intégrité dans le cas de transfert transitif d'information.

Définition 4.3.3 (Intégrité Indirecte des Données) *Un objet cso est dit intègre vis-à-vis d'un sujet css pour une trace T* ***ssi*** *il n'existe pas de flux d'information indirect de css vers cso avec $css \in \mathcal{CSS}$ et $cso \in \mathcal{CSO}$.*

$$IID(T, css, cso) \stackrel{def}{\equiv} \neg(css \overset{T}{\Rrightarrow} cso)$$

Par exemple, nous reprenons l'exemple de la figure 4.2 et la propriété d'intégrité de `/etc/shadow` vis-à-vis d'un processus utilisateur. Le flux d'information indirect $user\_t \overset{T}{\underset{[6136,6163]}{\Rrightarrow}} shadow\_t$ étant détecté, l'opération d'écriture par $root\_t$ dans $shadow\_t$ est interdite.

Définition 4.3.4 (Intégrité Générale des Données) *Un objet $cso \in \mathcal{CSO}$ est dit intègre vis-à-vis d'un sujet $css \in \mathcal{CSS}$ pour une trace T si et seulement si les propriétés IDD et IDO sont respectées pour ces même contextes.*

$$IGD(T, css, cso) \stackrel{def}{\equiv} \neg(css \overset{T}{\Rrightarrow\!\!\!\!\Rightarrow} cso) \Leftrightarrow IDD(css, cso) \wedge IID(css, cso)$$

Cette propriété combine les cas directs et indirects. Dans la suite de ce chapitre, nous définirons, pour chaque propriété, uniquement la version générale qui couvre à la fois le cas direct et indirect.

4.3.2 Intégrité des Sujets

La propriété de sécurité dite **Intégrité des Sujets** a pour but de prévenir la compromission d'un processus (ou tout autre sujet). Par exemple, la propriété peut prévenir contre l'injection de code, dans un processus légitime, par un processus malveillant. Cette propriété est également appelée *non-interférence* par Goguen et Meseguer dans [Goguen and Meseguer, 1982] et est définie comme suit :

Définition 4.3.5 (Non-Interférence) *Soit X un premier ensemble de sujets et Y un second ensemble de sujets, la propriété de non-interférence de X envers Y est respectée si aucun membre de X ne peut modifier Y.*

Dans notre formalisme, X et Y sont des ensembles de contextes de sécurité. Plus particulièrement, X et Y sont des ensembles de contextes sujets. La propriété revient donc à garantir qu'un contexte sujet ne puisse pas modifier un autre contexte sujet

Nous formalisons la propriété d'**Intégrité Générale des Sujets** qui prend en compte les modifications aussi bien directes qu'indirectes :

Définition 4.3.6 (Intégrité Générale des Sujets) *Un sujet $css_2 \in \mathcal{CSS}$ est dit intègre vis-à-vis d'un sujet $css_1 \in \mathcal{CSS}$ pour une trace T si et seulement si il n'existe pas de flux d'information générale allant de css_1 à css_2.*

$$IGS(T, css_1, css_2) \stackrel{def}{\equiv} \neg(css_1 \overset{T}{\lhd\!\lhd\!\lhd} css_2)$$

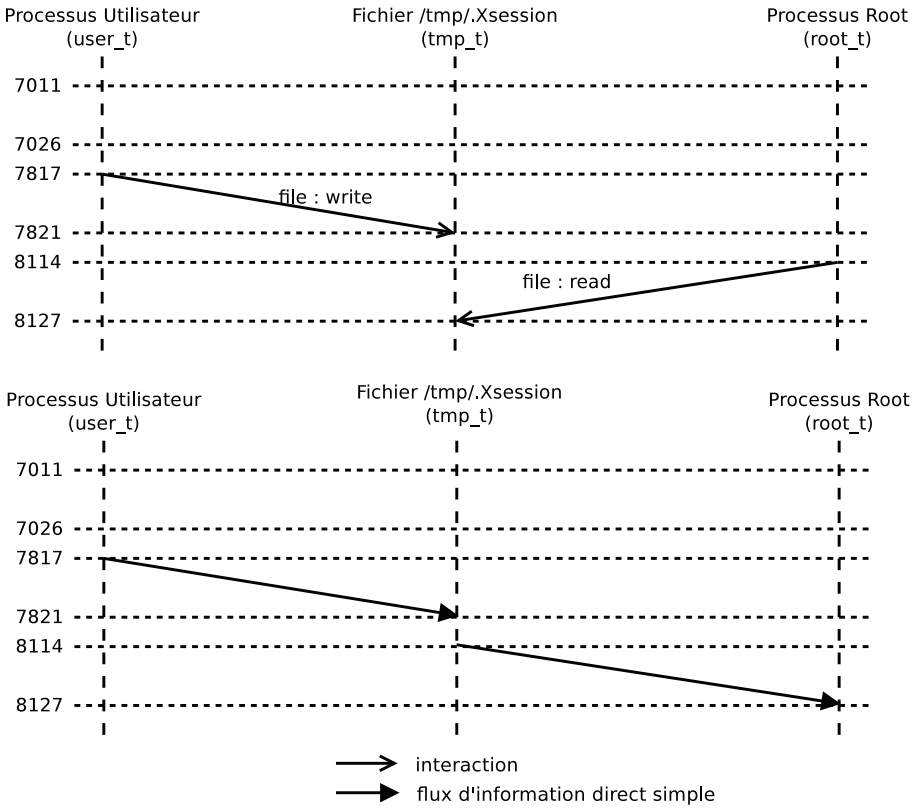

FIGURE 4.3 – Représentation d'une trace système avec rupture d'intégrité

Par exemple, nous souhaitons protéger l'intégrité des processus tournant en super-administrateur ($root\_t$) vis-à-vis des processus tournant en utilisateur ($user\_t$) en instanciant la propriété de sécurité d'**Intégrité Générale des Sujets** comme suit $IGS(T, user\_t, root\_t)$. Ainsi dans la figure 4.3, le flux d'information indirect $user\_t \overset{T}{\underset{[7817,8127]}{\rhd\!\!\rhd}} root\_t$ peut être empêché en interdisant le second flux, à savoir $tmp\_t \overset{T}{\underset{[8114,8127]}{\rhd}} root\_t$.

4.3.3 Intégrité Générale

Les deux propriétés de sécurité précédentes, c'est-à-dire Intégrité des Objets et Intégrité des Sujets, peuvent être respectivement exprimées de manière informelle comme l'interdiction pour un sujet X de modifier un objet Y et comme l'interdiction pour un sujet X de modifier un sujet Z. Les deux propriétés ont comme source un sujet X mais c'est la cible qui change : un objet ou un sujet. Nous proposons d'abstraire ces deux propriétés en une plus générique que nous nommons *Intégrité Générale*. Cette propriété, tout comme les deux autres précédemment citées, a comme source un sujet X mais la cible est indistinctement un sujet ou un objet donc plus généralement une entité. Par conséquent, nous introduisons la définition suivante pour l' intégrité générale :

Définition 4.3.7 (Intégrité Générale) *Une entité $cs_1 \in \mathcal{CS}$ est dit intègre vis-à-vis d'un sujet $css_1 \in \mathcal{CSS}$ pour une trace T si et seulement si il n'y a pas de flux d'information général allant de css_1 vers cs_1.*

$$IG(T, css_1, cs_1) \overset{def}{\equiv} \neg(css_1 \overset{T}{\rhd\!\!\rhd} cs_1)$$

Par exemple, nous souhaitons éviter toute rupture d'intégrité par le processus `firefox` sur l' environnement de l'utilisateur. Nous souhaitons donc prévenir tout flux d'information provenant de $firefox\_t$ et à destination de l'ensemble des contextes reliés à l'utilisateur c'est-à-dire $user\_u : * : *$. Nous instancions donc la propriété d'Intégrité Générale avec ces deux contextes : $IGD(T, firefox\_t, user\_u : * : *)$. En utilisant cette propriété, nous sommes capable de prévenir aussi bien la rupture d'intégrité sur les fichiers comme sur les processus. Par exemple, prenons la projection de la trace système T dans la figure 4.4 en haut et la trace représentée sous la forme de flux d'information en bas. Il existe quatre ruptures de la propriété de sécurité :

- $firefox\_t \overset{T}{\underset{[1147,1152]}{\rhd}} user\_home\_t$ correspond à la rupture de la propriété d'intégrité directe des données. En effet, le contexte destination est un élément de $user\_u : * : *$.
- $firefox\_t \overset{T}{\underset{[1358,1361]}{\rhd\!\!\rhd}} user\_home\_t$ correspond à la rupture de la propriété d'intégrité indirecte des données. En effet, les deux flux $firefox\_t \overset{T}{\underset{[1358,1359]}{\rhd}} thunderbird\_t$ et $thunderbird\_t \overset{T}{\underset{[1360,1361]}{\rhd}} user\_home\_t$ conduisent à cette violation. Le second flux sera interdit par notre propriété.
- $firefox\_t \overset{T}{\underset{[1421,1478]}{\rhd}} user\_exec\_t$ correspond à la rupture de la propriété d'intégrité directe des sujets. Ce flux sera donc interdit.
- $firefox\_t \overset{T}{\underset{[1685,1705]}{\rhd\!\!\rhd}} user\_exec\_t$ correspond à la rupture de la propriété d'intégrité indirecte des sujets. A nouveau, c'est le second flux qui est interdit.

La propriété d'intégrité générale $IGD(T, firefox\_t, user\_u : * : *)$ permet donc, en une seule expression et deux variables d'instanciation, de couvrir un large ensemble de ruptures de

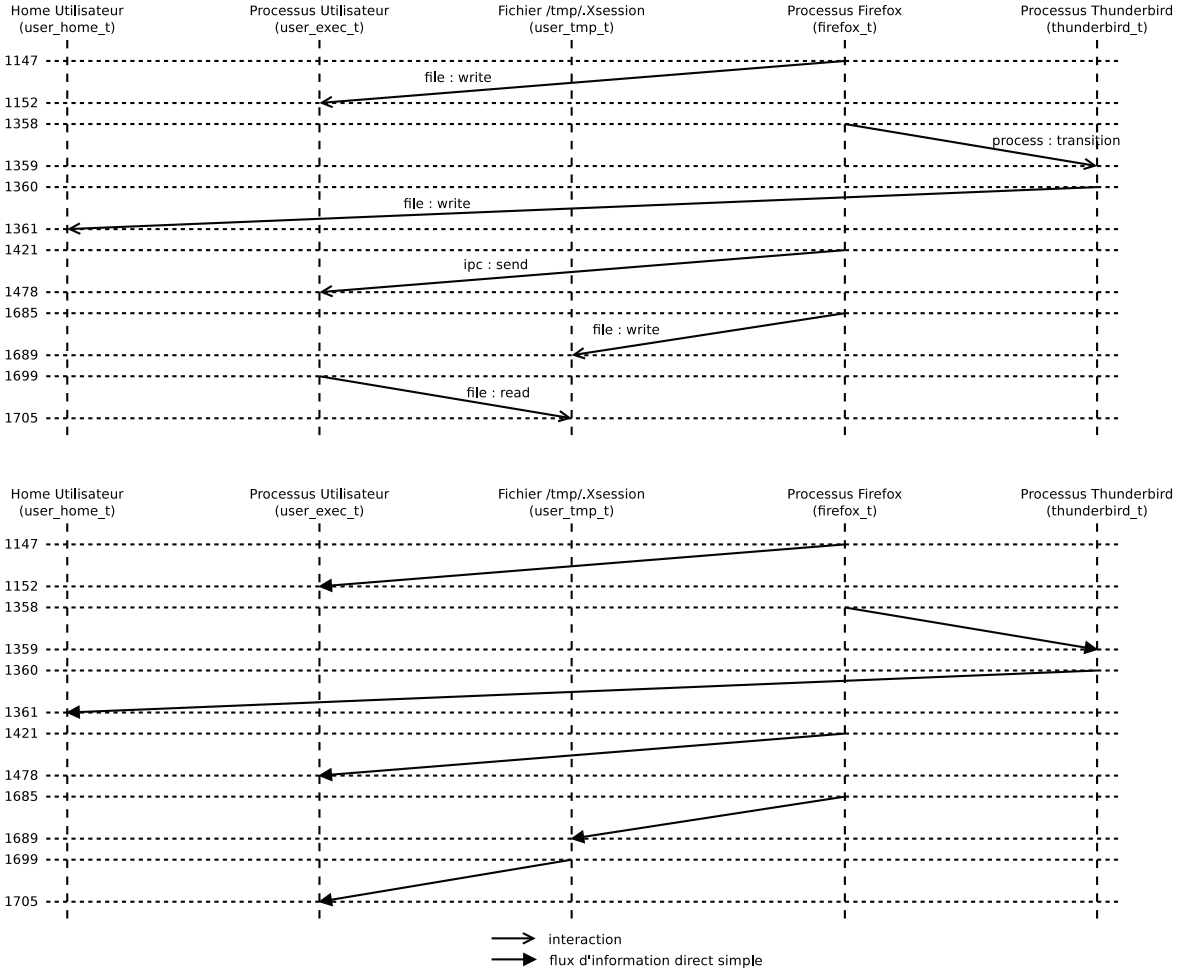

FIGURE 4.4 – Représentation d'une trace système avec rupture d'intégrité sujet et objet

sécurité. Elle permet de facilement protéger un ensemble de contexte contre un autre ensemble. Il est à noter que si par simplification les scénarios ne font pas apparaître de chevauchements des appels systèmes, et donc des interactions, notre formalisme le prend bien en compte. Ainsi par exemple, les deux dernières interactions pourraient se chevaucher telles que $firefox\_t \overset{T}{\underset{[1685,1699]}{\triangleright}} user\_tmp\_t$ et $user\_tmp\_t \overset{T}{\underset{[1698,1705]}{\triangleright}} user\_exec\_t$. La propriété empêcherait la seconde interaction commençant à la date 1698 d'avoir lieu.

4.3.4 Situation de Concurrence

Dans [Netzer and Miller, 1992], une définition générale est donnée pour les situations de concurrence ou races conditions (*RC*) . Nous l'utilisons comme base pour définir la propriété de sécurité dans notre formalisme.

Définition 4.3.8 (Situation de Concurrence Générale) *Une RC arrive quand il existe un ordonnancement imprévisible entre les accès de deux contextes de sécurité à un contexte objet conflictuel (c'est-à-dire un contexte partagé), avec au moins un des deux contextes (e.g. sc_1) provoquant une opération de type $\mathcal{WEO}$ sur le contexte partagée cso entre deux accès au contexte conflictuel cso par l'autre contexte (e.g. sc_2).*

Sur cette base et dans le but de détecter et prévenir les attaques se basant sur des *RC*, nous définissons la propriété :

Définition 4.3.9 (Absence de Situation de Concurrence Générale) *Un contexte de sécurité lsc est protégé contre les RC vis-à-vis d'un autre contexte de sécurité msc si et seulement si msc ne peut pas transférer de l'information vers le contexte objet partagé osc entre deux accès de lsc à ce même contexte partagé osc. La propriété est définie de manière formelle comme suit :*

$\text{No\_Race\_Condition}(T, lsc, msc) \Leftrightarrow$

$$\neg \left(\begin{array}{c} \exists osc \in \mathcal{SC} \wedge \\ \big(A = ((lsc \overset{T}{\underset{[s_1,\_]}{\Rrightarrow}} osc) \vee (osc \overset{T}{\underset{[s_1,\_]}{\Rrightarrow}} lsc)\big) \wedge \\ B = (msc \overset{T}{\underset{[s_2,e_2]}{\Rrightarrow}} osc) \wedge \\ \big(C = ((lsc \overset{T}{\underset{[\_,e_3]}{\Rrightarrow}} osc) \vee (osc \overset{T}{\underset{[\_,e_3]}{\Rrightarrow}} lsc)\big) \wedge \\ \big((s_1 \leq e_2) \ \wedge (s_2 \leq e_3)\big) \end{array} \right)$$

Notre définition couvre tous les ordonnancements incluant les situations de concurrence partielle ou totale.

Par exemple, nous souhaitons empêcher toutes attaques de type *Race Condition* entre les processus utilisateurs (malicieux - mcs), c'est-à-dire $user\_t$, et les processus relevant de `apache` (légaux - lcs), c'est-à-dire $apache\_t$. La trace T représentant le système est présentée dans la figure 4.5 à gauche et les flux d'information provoqués par cette trace sont visibles dans la figure 4.5 à droite. Les trois interactions forment une rupture de la propriété, en effet, les trois flux sont la base de la propriété d'**Absence de Situation de Concurrence Générale**. Le flux A correspond à $apache\_t \overset{T}{\underset{[9025,9056]}{\rhd}} apache\_tmp\_t$, B à $user\_t \overset{T}{\underset{[9069,9110]}{\rhd}} apache\_tmp\_t$ et C à $apache\_tmp\_t \overset{T}{\underset{[9089,9105]}{\rhd}} apache\_t$. Cette ensemble d'interactions forme une Race Condition car la condition temporelle de la propriété n'est pas respectée. En effet, la condition $(s_1 \leq e_2) \wedge (s_2 \leq e_3)$ n'est pas respectée puisque $(9025 \leq 9110) \wedge (9010 \leq 9105)$. Dans ce cas, le dernier flux $apache\_tmp\_t \overset{T}{\underset{[9089,9105]}{\rhd}} apache\_t$ est interdit. L'exemple considére un cas de recouvrement, il en existe bien évidemment quatre autres pouvant conduire à une situation de concurrence. La définition prend bien en compte ces cinq cas de recouvrement.

4.3.5 Intégrité des Domaines

L'**Intégrité des Domaines** permet d'isoler des processus au sein d'un sous ensemble du système, elle correspond à la notion de *chroot virtuel* ou $VCHROOT$. Cette propriété est générale-

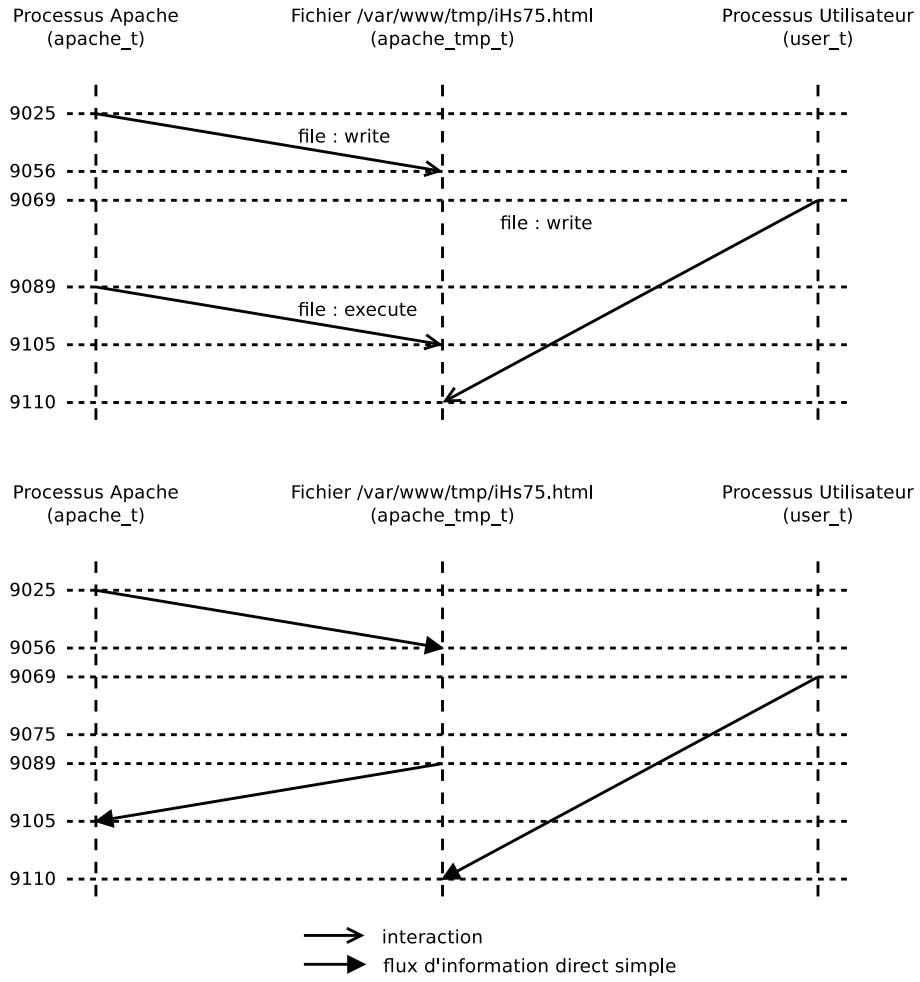

FIGURE 4.5 – Représentation d'une trace système avec situation de concurrence

ment vue, e.g. [Briffaut, 2007], comme un confinement strict.

Nous définissons cette propriété comme suit :

Définition 4.3.10 (Intégrité des Domaines) *Suivant une trace système T et un ensemble VSC de contextes de sécurité, l'intégrité du domaine VSC est garantie si et seulement si chaque membre de VSC partage de l'information seulement avec les membres de VSC.*

$$VCHROOT(T, VSC) \stackrel{def}{=} \left(\begin{array}{c} \forall sc \in VSC, \forall sc_k \in \mathcal{CS} \\ \left((sc \overset{T}{\lhd\!\lhd\!\lhd} sc_k) \vee (sc_k \overset{T}{\lhd\!\lhd\!\lhd} sc) \right) \\ \Rightarrow \\ (sc_k \in VSC) \end{array} \right).$$

En pratique, l'utilisation de cette propriété est difficile. En effet, il faut pouvoir démarrer le système en faisant entrer des processus dans le domaine virtuel ce que ne prend pas en compte cette propriété. C'est pourquoi nous proposons une seconde propriété :

Définition 4.3.11 (Intégrité Fonctionnelle des Domaines) *Suivant une trace système T et un ensemble VSC de contextes de sécurité, l'Intégrité Fonctionnelle des Domaines pour VSC est garantie si et seulement si une des deux conditions est vraie pour chaque interaction de la trace :*

1. *les deux contextes de l'interaction appartiennent à VSC.*
2. *la source de l'interaction n'est pas au sein de VSC.*

$$GVCHROOT(T, VSC) \stackrel{def}{\equiv} \left(\begin{array}{c} \forall sc \in VSC, \forall sc_k \in \mathcal{CS} \\ \left((sc \overset{T}{\blacktriangleright}_{it_i} sc_k) \vee (sc_k \overset{T}{\blacktriangleright}_{it_i} sc) \right) \\ \Rightarrow \\ \left((sc_k \in VSC) \vee (src(it_i) \notin VSC) \right) \end{array} \right).$$

Par exemple, nous souhaitons confiner le processus `firefox` du reste du système (et plus particulièrement de l'utilisateur l'ayant lancé). Ce besoin d'isolation part de l'observation qu'un grand nombre d'attaques sur les machines actuelles sont de type *drive-by download* [1] et utilisent comme vecteur d'entrée sur le système le navigateur Internet. Nous instancions donc la propriété d'**Intégrité Fonctionnelle des Domaines** avec l'ensemble des contextes appartenant à `firefox`, c'est-à-dire l'ensemble de contextes tel que $firefox\_d : * : *$. La propriété d'**Intégrité Fonctionnelle des Domaines** permet à l'utilisateur d'interagir avec `firefox` en lui envoyant des données e.g. des frappes clavier mais empêche `firefox` d'initier un flux d'information avec l'utilisateur e.g. exécution par `firefox` de malware dans l'espace utilisateur, vol d'informations confidentielles par `firefox`. La propriété d'**Intégrité des Domaines** appliquée à `firefox` aurait empêchée le lancement de celui-ci. Elle aurait empêché donc le bon fonctionnement du système.

Dans la figure 4.6 à gauche est représentée une trace T et à droite sous la forme de flux d'information directs. Il existe six flux de données entre le domaine utilisateur et celui de `firefox`. Parmi ces flux, certains sont légitimes et doivent être autorisés. Les autres sont des ruptures de la propriété et sont interdits. Les six flux entre le domaine de `firefox` et le reste du système sont les suivants :

- $user\_u : user\_r : user\_t \overset{T}{\underset{[2401,2468]}{\triangleright}} firefox\_d : firefox\_r : firefox\_t$: il correspond au lancement de `firefox` par l'utilisateur. Ce flux est autorisé puisqu'il correspond à l'entrée dans le domaine.
- $firefox\_d : firefox\_r : firefox\_t \overset{T}{\underset{[2531,2542]}{\triangleright}} user\_u : user\_r : user\_t$: le flux est également légitime puisqu'il s'agit d'un flux d'information du domaine `firefox` vers le domaine utilisateur extérieur au chroot mais initialisé par ce domaine extérieur.
- $firefox\_d : firefox\_r : firefox\_t \overset{T}{\underset{[2587,2601]}{\triangleright}} user\_u : user\_r : user\_home\_t$: le flux est interdit puisqu'il correspond à `firefox` cherchant à faire sortir de l'information du domaine chroot.
- $user\_u : user\_r : user\_home\_t \overset{T}{\underset{[2789,2814]}{\triangleright}} firefox\_d : firefox\_r : firefox\_t$: le flux est aussi interdit puisqu'il correspond à `firefox` cherchant à faire entrer de l'information dans le domaine chroot.
- $user\_u : user\_r : user\_t \overset{T}{\underset{[2845,2853]}{\triangleright}} firefox\_d : firefox\_r : firefox\_t$: le flux est interdit puisqu'il correspond à `firefox` essayant d'obtenir de l'information d'un sujet extérieur au domaine.
- $firefox\_d : firefox\_r : firefox\_t \overset{T}{\underset{[3025,3049]}{\triangleright}} user\_u : user\_r : user\_t$: ce flux est interdit puisqu'il correspond à `firefox` essayant d'envoyer de l'information vers un sujet extérieur au domaine.

On voit que la propriété est fonctionnelle puisqu'elle empêche les attaques issues du domaine chroot tout en permettant le fonctionnnement correct de l'application `firefox`.

1. Drive-by Download : Attaque du coté du client ayant pour cible la plus courante les navigateurs et se lançant au chargement d'une page [Cova et al., 2010].

4.3.6 Executable de Confiance

Le but de la propriété de sécurité **Exécutable de Confiance** (*Trusted Path Execution* en anglais) (TPE) est de garantir qu'un ensemble de sujets peut uniquement exécuter des données provenant d'un ensemble d'objets connus et de confiance tel que défini dans [Rahimi, 2004]. Par exemple, un sujet $user\_t$ est uniquement capable d'exécuter des binaires provenant du contexte objet bin_t.

Historiquement, les versions utilisées de la propriété TPE ne supportent que les exécutions directes. Cette définition est souvent liée à des problèmes d'intégrité car elle a un fort potentiel à influer sur le comportement du sujet provoquant l'exécution. En effet, l'exécution d'un binaire malicieux peut mener à la violation de l'intégrité du sujet.

Nous introduisons la propriété d'**Exécutable de Confiance Général** (*General Trusted Path Execution*) afin de traiter aussi bien les cas d'exécution directe qu'indirecte.

Définition 4.3.12 (General TPE) *Suivant une trace système T, un sujet css et un ensemble d'objets sûrs OC, la propriété Exécutable de Confiance Général (GTPE) pour css et OC est respectée si et seulement si css exécute uniquement, de manière directe ou indirecte, des objets (binaires) provenant de OC :*

$$GTPE(T, css, OC) \stackrel{def}{\equiv} \left(\begin{array}{c} \forall cso \in \mathcal{SC}, css \in \mathcal{SSC}, \\ (css \overset{T}{\underset{x}{\Rrightarrow}} cso) \Rightarrow (cso \in OC) \end{array} \right).$$

La mise en oeuvre de cette propriété peut reposer sur la contraposée qui est équivalente à :

$$\neg(cso \in OC) \Rightarrow \neg(css \overset{T}{\underset{x}{\Rrightarrow}} cso)$$

Ainsi, si cso n'est pas dans l'ensemble OC alors css ne peut pas l'exécuter.

Par exemple, nous souhaitons obliger l'ensemble des contextes $user\_u : * : *$, à n'utiliser que des exécutables qui proviennent des répertoires `/bin` et `/usr/bin` ayant respectivement les contextes $system\_u : object\_r : bin\_t$ et $system\_u : object\_r : usr\_bin\_t$. Dans la figure 4.7, $user\_u : user\_r : user\_t \overset{T}{\underset{x}{\blacktriangleright}} system\_u : object\_r : bin\_t$ est autorisée. Par contre, $user\_u : user\_r : user\_t \overset{T}{\underset{x}{\blacktriangleright}} user\_u : user\_r : user\_home\_t$ est refusée car le contexte destination ne fait pas parti des objets sûrs. La troisième et la quatrième interactions forment une exécution indirecte $user\_u : user\_r : user\_t \overset{T}{\underset{x}{\Rrightarrow}} system\_u : object\_r : usr\_bin\_t$ qui est légitime. Par contre, la troisième et la cinquième interaction correspondant à l'exécution indirecte $root\_u : root\_r : root\_t \overset{T}{\underset{x}{\blacktriangleright}} user\_u : user\_r : user\_home\_t$ est illégale. En conséquence, la cinquième interaction est refusée.

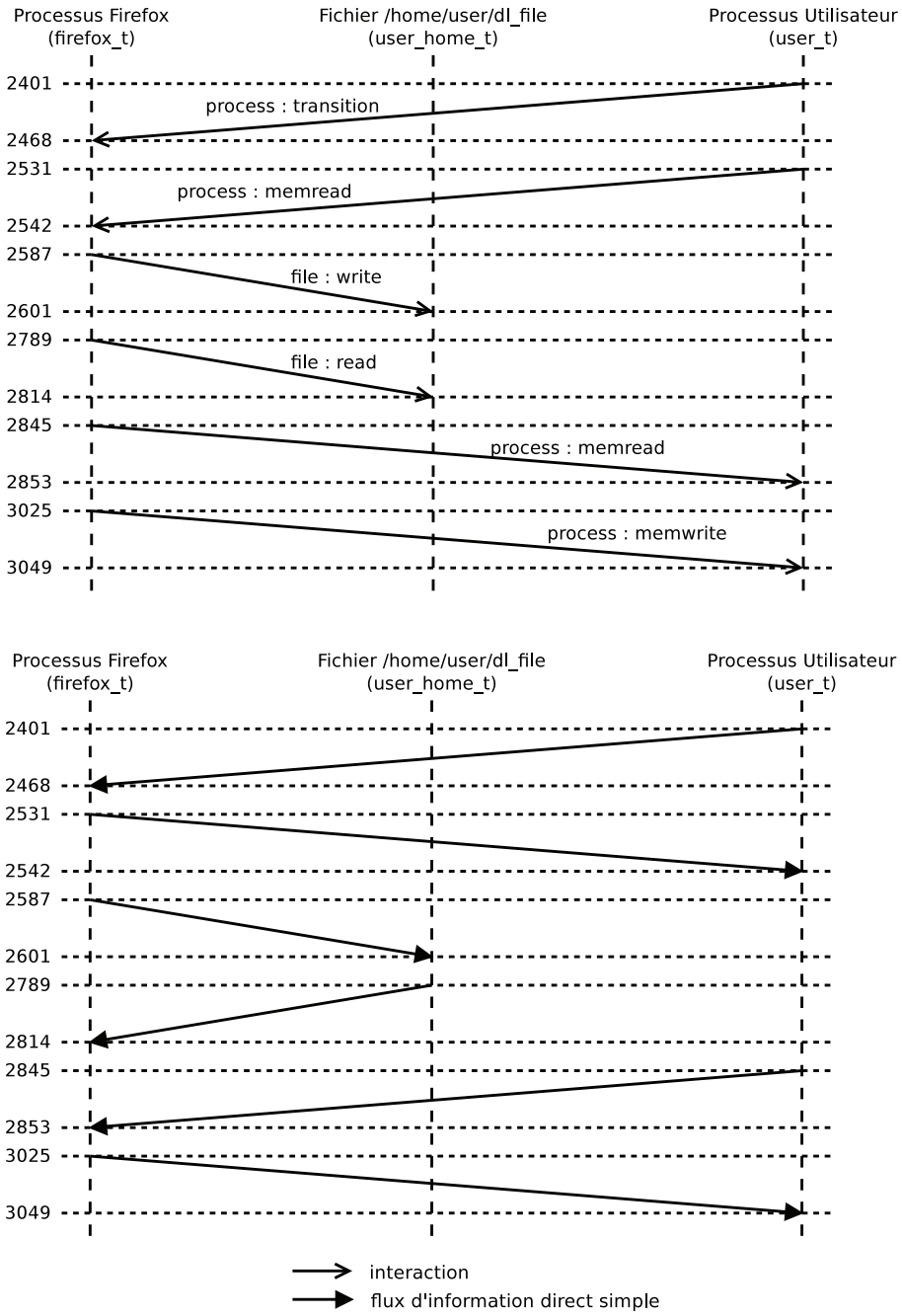

FIGURE 4.6 – Représentation d'une trace système avec rupture du domaine

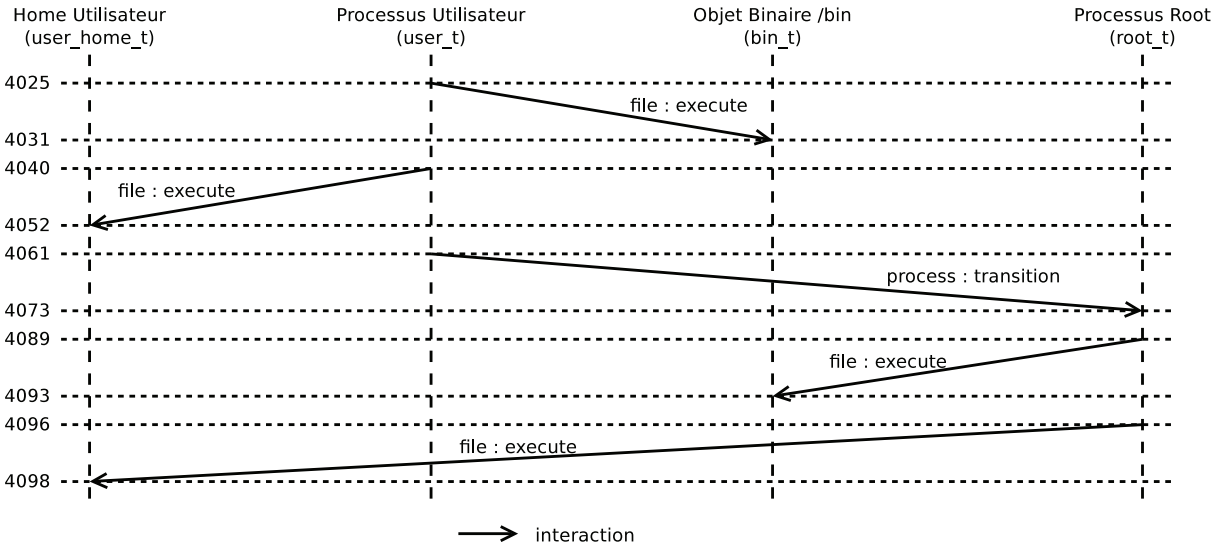

FIGURE 4.7 – Représentation d'une trace système avec rupture du TPE

4.4 Confidentialité

Les propriétés de sécurité de *confidentialité* ont pour objectif de contrôler les accès de type lecture. Dans ce but, nous introduisons la propriété de confidentialité des données du système. D'autres propriétés relèvent également de la confidentialité mais seront abordées dans d'autres parties de ce chapitre.

4.4.1 Confidentialité des Données

Une définition générale de la confidentialité est de prévenir contre l'accès par une entité (c'est-à-dire un sujet) à une information non autorisée (c'est-à-dire un objet). Elle est formalisable comme suit :

Définition 4.4.1 (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.*

Dans notre modélisation, les ensembles X et I correspondent à des ensembles de contextes, respectivement sujets et objets. En partant du prédicat que dans un système, une entité peut accéder à de l'information détenue par une seconde entité si la première peut lire directement de l'information sur la seconde ou si un flux d'information existe allant de la seconde entité vers la première.

Le but de la propriété de confidentialité des données consiste typiquement à contrôler les flux d'informations, directs ou indirects, allant des objets vers les sujets.

Définition 4.4.2 (Confidentialité Générale des Données) *Un objet $cso \in \mathcal{CSS}$ est dit intègre vis-à-vis d'un sujet $css \in \mathcal{CSS}$ pour une trace T si et seulement si il n'existe pas de flux d'information générale de cso à css.*

$$CGD(T, css, cso) \stackrel{def}{=} \neg(cso \overset{T}{\lhd\!\lhd\!\lhd} css)$$

Par exemple, pour le système représenté dans la figure 4.8, nous voulons empêcher un utilisateur ($user\_t$) de lire le fichier `/etc/shadow` ($shadow\_t$). Donc, $shadow\_t \underset{[1101,1109]}{\overset{T}{\rhd}} user\_t$ est interdit. Le flux d'information indirect $shadow\_t \underset{[1206,1276]}{\overset{T}{\lhd\!\lhd}} user\_t$ composé des deux flux directs $shadow\_t \underset{[[1206,1221],r]}{\overset{T}{\rhd}} oot\_t$ et $root\_t \underset{[[1256,1276],u]}{\overset{T}{\rhd}} ser\_t$ correspond à une tentative de rupture de la propriété. En conséquence, le flux de $root\_t$ vers $user\_t$ est interdit.

Ainsi, nous sommes donc en mesure de protéger la confidentialité de `/etc/shadow` vis-à-vis d'un processus utilisateur, et cela, même si ce dernier utilise des canaux cachés (observable par le noyau).

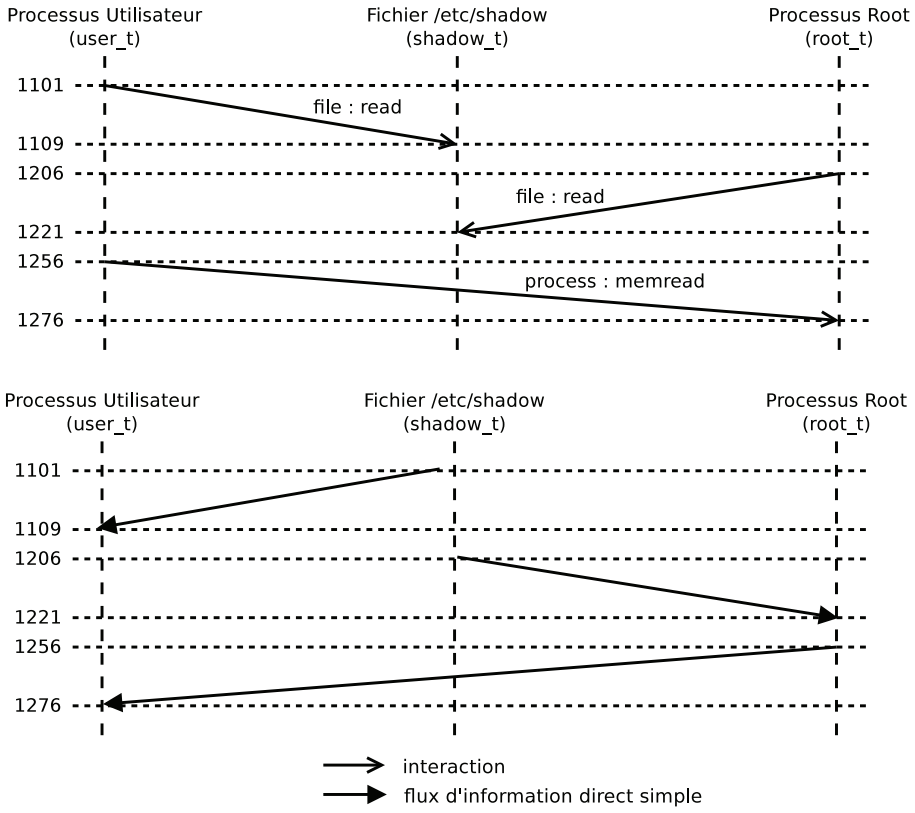

FIGURE 4.8 – Représentation d'une trace système avec rupture de confidentialité directe et indirecte

4.5 Multi Level Security (MLS)

Il existe deux principaux modèles de sécurité pour les systèmes multi-niveaux (MLS) : Biba et BLP. Nous considérons ici que la propriété **Bell&LaPadula restrictif avec plage**. En effet, les mises en oeuvre réelles de BLPR repose majoritairement sur ce modèle. Nous proposons, en annexe 8.1, un ensemble plus large de propriétés MLS qui ont toutes été formalisées au moyen de notre langage.

De plus, comme établi dans [Rushby, 1984], il est suffisant de prendre en compte les flux directs. En effet, cet article établit que les flux indirects sont pris en compte implicitement.

4.5.1 Bell&LaPadula Restrictif avec Plage

Cette propriété de sécurité BLPRP est une version avec plage de niveau d'habilitation (et de sensibilité) de la propriété **BLP restrictif**. Le modèle de confidentialité dit **BLP restrictif** (BLPR) permet de garantir la confidentialité du système via le respect de trois règles de contrôle sur les opérations élémentaires. Ces trois règles sont les suivantes :

blpr-1 Un sujet peut accéder en lecture à un objet *ssi* son niveau d'habilitation est supérieur ou égal au niveau de sensibilité de l'objet.

blpr-2 Un sujet peut ajouter des données (c'est-à-dire modification sans lecture) à un objet *ssi* son niveau d'habilitation est inférieur ou égal au niveau de sensibilité de l'objet.

blpr-3 Un sujet peut modifier un objet *ssi* son niveau d'habilitation est supérieur ou égal au niveau de sensibilité de l'objet.

Contrairement à la version restrictive classique de BLPR, une plage de niveaux d'habilitation est donnée à chaque contexte de sécurité plutôt qu'un seul et unique niveau. Nous définissons ainsi la fonction $hab2 : SC \rightarrow \mathbb{N} \times \mathbb{N}$ qui associe à un contexte sa plage de niveau d'habilitation ainsi que deux fonctions subsidiaires : $habmax : SC \rightarrow \mathbb{N}$ qui retourne la borne supérieur de la plage de niveau d'habilitation d'un contexte et $habmin : SC \rightarrow \mathbb{N}$ qui retourne la borne inférieure.

Nous donnons une définition pour chacune des trois règles.

Définition 4.5.1 (Règle BLPRP-1) *Un objet cso est dit confidentiel vis-à-vis d'un sujet css_1 suivant blpr-1 pour une trace T ssi il n'existe que des flux d'information directs généraux de cso vers css_1 avec $css_1 \in \mathcal{CSS}$, $cso \in \mathcal{CSO}$ et $habmin(css_1) \geqslant habmax(cso)$.*

$$BLPRP1(T, css_1, cso) \stackrel{def}{\equiv} (cso \overset{T}{\rhd} css_1) \wedge (habmin(css_1) \geqslant habmax(cso))$$

Définition 4.5.2 (Règle BLPRP-2) *Un objet cso est dit confidentiel vis-à-vis d'un sujet css_1 suivant blpr-2 pour une trace T ssi toutes les interactions ayant pour opération élémentaire eo un ajout de données ($eo \in \mathcal{AEO}$) ayant pour source $css_1 \in \mathcal{CSS}$ et pour destination $cso \in \mathcal{CSO}$ respectent $habmax(css_1) \leqslant habmin(cso)$.*

$$BLPRP2(T, css_1, cso) \stackrel{def}{\equiv} (css_1 \xrightarrow[eo]{T} cso) \wedge (habmax(css_1) \leqslant habmin(cso)) \wedge (eo \in \mathcal{AEO})$$

Définition 4.5.3 (Règle BLPRP-3) *Un objet cso est dit confidentiel vis-à-vis d'un sujet css_1 suivant blpr-3 pour une trace T ssi il n'existe que des flux d'information direct généraux de css_1 vers cso avec $css_1 \in \mathcal{CSS}$, $cso \in \mathcal{CSO}$, $habmax(css_1) = habmax(cso)$ et $habmin(css_1) = habmin(cso)$.*

$$BLPR3(T, css_1, cso) \stackrel{def}{\equiv} (css_1 \overset{T}{\rhd} cso) \wedge (habmin(css_1) = habmin(cso) \wedge habmax(css_1) = habmax(cso))$$

La propriété de confidentialité dite BLPRP est respectée si et seulement si ces trois règles 4.5.1, 4.5.2 et 4.5.3 sont satisfaites.

Propriété 4.5.1 (Bell&LaPadula Restrictif avec Plages) *Soit $S = (css_1, cso_1), ..., (css_m, cso_m)$, un ensemble de couples de contextes. Un **système**, représenté par une trace T, est dit **confidentiel** selon* BLPRP *si et seulement si les règles 4.5.1, 4.5.2 et 4.5.3 sont respectées pour tous les couples de contextes du système (css_i, cso_i) avec $css_i \in \mathcal{CSS}$ et $cso_i \in \mathcal{CSO}$.*

$$BLPRP(T,S) \stackrel{def}{\equiv} \forall i = 1..m, BLPRP1(T, css_i, cso_i) \wedge BLPRP2(T, css_i, cso_i) \wedge BLPRP3(T, css_i, cso_i)$$

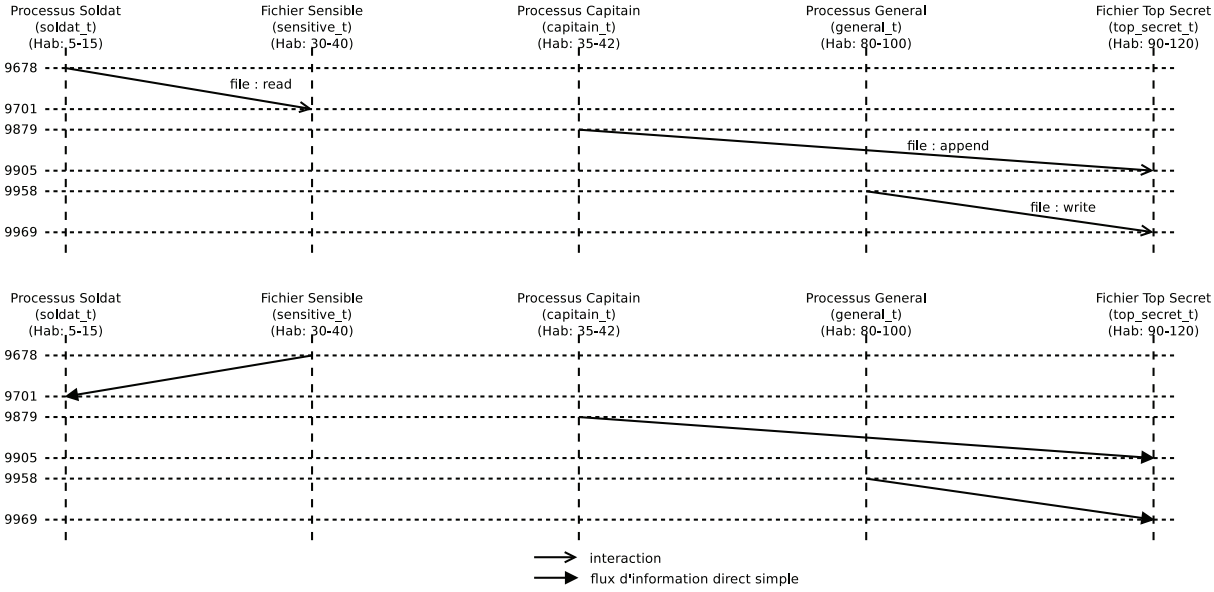

FIGURE 4.9 – Représentation d'une trace système avec rupture de BLPRP

Nous appliquons la propriété de sécurité BLPR sur un système d'exploitation représenté par la trace T, voir figure 4.9.

Le flux d'information $sensitive\_t \overset{T}{\underset{[9687,9701]}{\triangleright}} soldat\_t$ correspond à une lecture et c'est donc la *première règle* qui s'applique. Comme $5 \leq 40$, la règle n'est pas respectée et le flux est interdit.

La deuxième interaction, $capitain\_t \overset{T}{\underset{file:append}{\multimap}} [9879, 9905]\ top\_secret\_t$, correspond à un ajout de donnée, c'est donc la *deuxième règle* qui s'applique. Comme $42 \leq 90$, l'interaction est interdite.

Le flux d'information $general\_t \overset{T}{\underset{[9958,9996]}{\triangleright}} top\_secret\_t$ correspond à une écriture, c'est donc la *troisième règle* de BLPR qui s'applique. Comme $80 \neq 90$ et $100 \neq 120$, le flux est interdit.

4.5.2 Extensions

Dans la pratique, le bon fonctionnement des modèles MLS nécessitent des procédures de déclassification d'information. Ce sont ces procédures que nous abordons dans cette section. Ces extensions peuvent permettre de définir des exceptions à BLP restrictif.

Sujets Sûrs (Trusted Subjects)

Un sujet sûr (*trusted*) peut violer la politique. Cette violation peut avoir comme objectif de classifier ou de déclassifier de l'information.

Définition 4.5.4 (Trusted Subjects) *Soit $S = css_1, ..., css_m$, l'ensemble des sujets sûrs. Un système, représenté par la trace T, implémente la définition des sujets sûrs si et seulement si, tous les flux d'information ayant pour source un sujet sûr sont autorisés.*

$$TrustedSubject(T, S) \stackrel{def}{\equiv} \forall it \in T, src(it) \in S$$

Objets Sûrs (Trusted Objects)

Les objects sûrs (**trusted objects**) sont des objets déclassifiés que tout le monde peut utiliser librement.

Définition 4.5.5 (Trusted Object) *Soit $O = cso_1, ..., cso_m$, l'ensemble des objets sûrs. Un système, représenté par la trace T, supporte le concept d'objets sûrs si et seulement si les flux d'information sont autorisées vers les objets sûrs.*

$$TrustedObject(T, O) \stackrel{def}{\equiv} \forall it \in T, tgt(it) \in O$$

4.6 Abus de privilèges

Cette classe de propriété de sécurité a pour but de prévenir les changements de privilèges anormaux. Nous considérons trois propriétés d'abus de privilèges. Les abus de privilège concernent aussi bien l'intégrité que la confidentialité.

4.6.1 Séparation de Privilèges

Nous nous focalisons sur la séparation de privilèges entre écriture et exécution même si d'autres versions peuvent être formaliser dans notre modélisation. Il s'agit d'une version dynamique de la séparation de privilèges car un sujet n'est pas lié statiquement à un droit d'écriture ou d'exécution.

Définition 4.6.1 (Séparation de privilèges Dynamique) *Un système, représenté par une trace T, respecte la propriété de sécurité dite de Séparation de Privilèges ssi pour chaque opération élémentaire eo de type exécution, la source de l'opération élémentaire c'est-à-dire un sujet css n'est pas également la source d'un flux d'information général allant vers l'objet cso cible de l'opération élémentaire.*

$$SdP(T) \stackrel{def}{\equiv} (css \underset{x}{\overset{T}{\Rrightarrow}} cso, \neg(css \overset{T}{\Rrightarrow} cso)) \vee (css \overset{T}{\Rrightarrow} cso, \neg(css \underset{x}{\overset{T}{\Rrightarrow}} cso))$$

Par exemple, nous souhaitons empêcher le processus `firefox` de télécharger puis d'exécuter un programme. Une des capacités de cette propriété est d'empêcher des attaques de type "drive-by download" qui sont actuellement l'un des vecteurs les plus courants de contamination sur Internet. En pratique, nous souhaitons garantir la propriété pour le contexte $firefox\_u : * : *$ vis-à-vis de tous les objets sur le système c'est-à-dire $* : object\_r : *$.

La première interaction, dans la figure 4.10, $firefox\_t \underset{file:write}{\overset{T}{\rightharpoonup}} [3502, 3512]\ user\_home\_t$, est autorisée car il n'y a pas de conflit avec des exécutions.

La seconde interaction, $firefox\_t \underset{file:execute}{\overset{T}{\rightharpoonup}} [3545, 3562]\ user\_home\_t$ est interdite puisqu'il y a conflit avec la première interaction.

La troisième et la quatrième interaction correspondent à l'exécution indirecte $firefox\_t \underset{[3581,4012]_x}{\overset{T}{\Rrightarrow}} user\_home\_t$. C'est pourquoi la quatrième interaction est interdite.

Ici, ce sont les exécutions qui sont annulées parce que l'opération d'écriture a lieu en premier. A l'inverse, si l'opération d'exécution avait lieu en premier, les opérations d'écriture seraient interdites. On voit bien que la décision est dynamique et dépend de l'ordre d'apparition des opérations.

4.6.2 Absence de Changement de Contexte

Nous introduisons la propriété 4.6.2 qui prévient contre tous les scénarios de changement de contexte qu'ils soient directs ou indirects.

Définition 4.6.2 (Absence de changement de contexte) *Soit $S = \{(css_1, css_2), ..., (css_m, css_{m+1})\}$, un ensemble de couples de contextes sujets. Un système représenté par une trace T respecte la propriété d'absence de changement de contexte* ***ssi*** *pour tout couple (css_i, css_{i+1}), css_i ne peut pas transiter vers css_{i+1}.*

$$ACC(T, S) \stackrel{def}{\equiv} \forall i = 1..m, \neg(css_i \overset{T}{\Rrightarrow}_t css_{i+1})$$

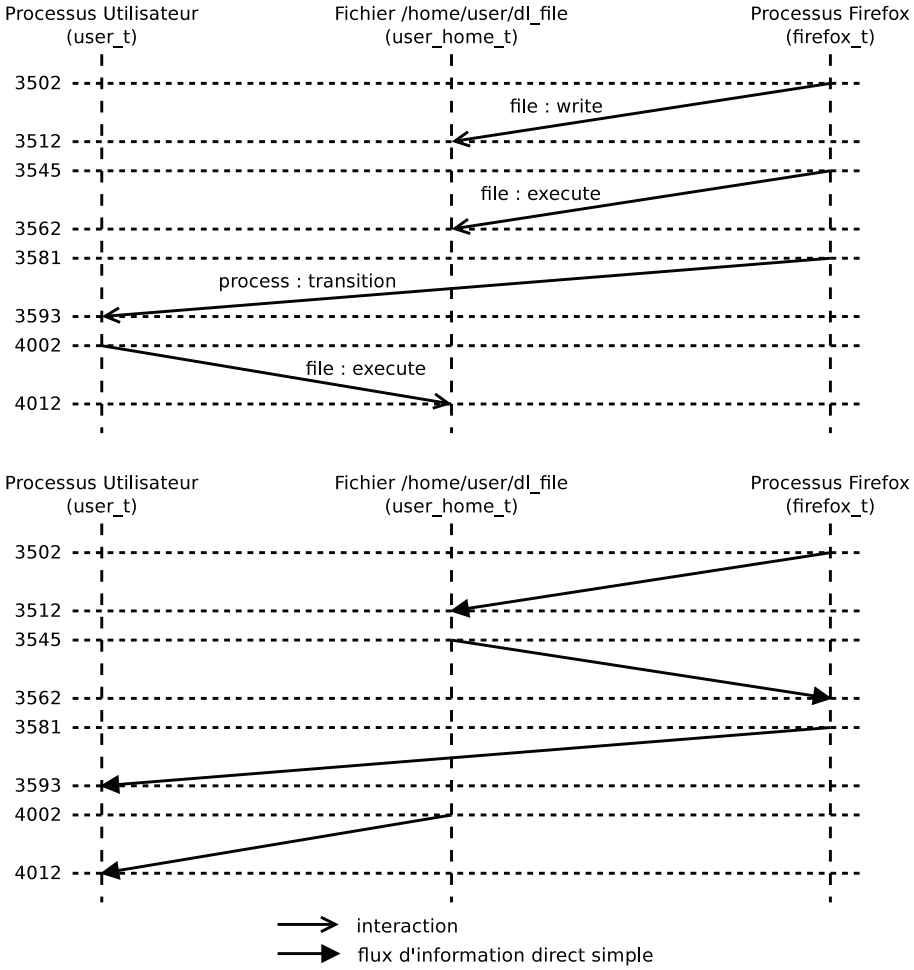

FIGURE 4.10 – Représentation d'une trace système avec rupture de la séparation de privilège

Par exemple, nous souhaitons que le processus `firefox` ne puisse pas transiter vers le domaine de l'utilisateur. Cela nous permet d'empêcher un code malveillant d'atteindre les privilèges de l'utilisateur pour l'espionner/lui voler des données sensibles.

Sur le système visible dans la figure 4.11, la transition $user\_t \overset{T}{\underset{[2563,2578]}{\lll\!\!\rightarrow_t}} firefox\_t$ est autorisée car l'utilisateur doit pouvoir lancer le programme `firefox`. Par contre, la transition $firefox\_t \overset{T}{\underset{[3581,3593]}{\lll\!\!\rightarrow_t}} user\_t$, est interdite car elle permettrait à `firefox` d'atteindre les privilèges de l'utilisateur.

4.6.3 Respect d'une Politique

Une politique de contrôle d'accès $\mathcal{POL}$ correspond un ensemble d'interactions $\{it_1, \ldots, it_n\}$ légales. Il y a violation de la politique lorsqu'il apparait une interaction dans la trace qui n'est pas élément de $\mathcal{POL}$.

Propriété 4.6.1 (Respect d'une politique) *Soit $\mathcal{POL} = \{it_1, \ldots, it_n\}$ une politique de contrôle d'accès et T une trace représentant le système. Le système respecte une politique de contrôle*

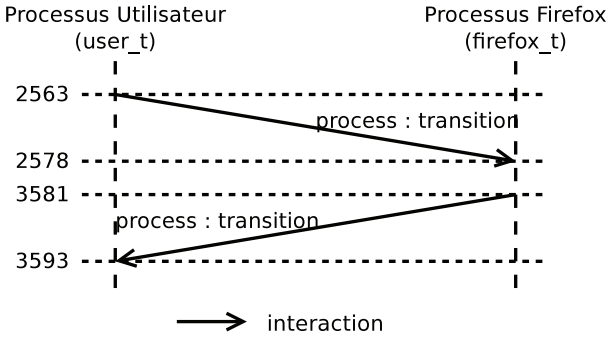

FIGURE 4.11 – Représentation d'une trace système avec rupture de la propriété de changement de contexte

d'accès si toute interaction appartient à la politique.

$$RPol(T, \mathcal{POL}) \stackrel{def}{\equiv} T \subset \mathcal{POL}$$

Cette propriété permet de vérifier la sûreté d'un mécanisme de contrôle d'accès. En pratique, ce mécanisme n'est pas assez précis. En effet, il ne permet pas de distinguer, parmi les violations, les actions indirectes qui sont légitimes de celles qui ne le sont pas. Ainsi parmi les actions, il peut y avoir des flux indirects qui sont légitimes et d'autres illégitimes.

4.7 Propriétés Dynamique

4.7.1 Muraille de Chine

Introduction

Dans cette section, nous démontrons la flexibilité de notre modélisation en transposant les définitions de la propriété de la muraille de chine (voir section 2.2.2) dans notre formalisme. La particularité de la Muraille de Chine (Chinese Wall – CW) est d'utiliser une fonction PR correspondant à l'historique des accès en lecture. C'est sur la base de cet historique qu'est calculé dynamiquement l'autorisation d'accès. Dans notre cas, la fonction PR est représentée par la trace.

Entreprises et Conflits d'Intérêts

Les auteurs de [Brewer and Nash, 1989a] découpent le système en deux groupes d'entités :
- L'ensemble des objets, représenté par O qui est équivalent à $\mathcal{CSO}$;
- L'ensemble des sujets, représenté par S qui est équivalent à $\mathcal{CSS}$.

De plus, chaque objet se voit affecté deux étiquettes :
- Company Dataset (CD) qui regroupe l'ensemble des données appartenant à une société.
- Conflict Of Interest (COI) qui regroupe l'ensemble des CD d'entreprises en concurrences.

Nous étendons notre modélisation pour introduire trois fonctions :
- $cd : \mathcal{CSO} \rightarrow \mathcal{CD}$ qui retourne le CD d'un contexte objet.
- $coi : \mathcal{CSO} \rightarrow \mathcal{COI}$ qui retourne le COI d'un contexte objet.
- $sane : \mathcal{CSO} \rightarrow \mathcal{CSO}_{sane}$ qui retourne $vrai$ si un objet contient uniquement des données publiques. Cela permet de prendre en charge l'échange de données publiques au sein d'un même groupe d'entreprises concurrences. Par exemple, un communiqué de presse.

Le premier axiome ACW_1 du modèle de la Muraille de Chine [Brewer and Nash, 1989a] exprime la condition suivante : si deux objets ont le même CD, ils ont la même COI. L'axiome correspond à l'hypothèse que la labélisation des objets est faite correctement.

$$ACW_1 \stackrel{def}{\equiv} \forall cso_1, \forall cso_2 \in \mathcal{CSO}, cd(cso_1) = cd(cso_2) \Rightarrow coi(cso_1) = coi(cso_2)$$

Le lemme LCW_1 représentant la contraposée de l'axiome ACW_1 est la suivante :

$$LCW_1 \stackrel{def}{\equiv} \forall cso_1 \in \mathcal{CSO}, \forall cso_2 \in \mathcal{CSO}, coi(cso_1) \neq coi(cso_2) \Rightarrow cd(cso_1) \neq cd(cso_2)$$

Chinese Wall Simple Rule

La définition 2.2.3 associée à **Chinese Wall-Simple Security Condition Preliminary Version** est traduite dans notre formalisme par la propriété suivante.

Propriété 4.7.1 (CWSR) *Soit un sujet $css \in \mathcal{CSS}$, il peut lire un objet $cso_1 \in \mathcal{CSO}$ ssi*

$$\forall cso_2 \in \mathcal{CSO}, cso_2 \stackrel{T}{\triangleright} css : \begin{array}{l} coi(cso_2) \neq coi(cso_1) \\ \vee \\ cd(cso_2) = cd(cso_1) \end{array}$$

Nous avons vu dans l'état de l'art que cette propriété ne prend pas en compte les flux d'information transitif, nous présenterons ensuite la propriété **Chinese Wall-* Property**.

État Initial

La propriété de la muraille de chine nécessite la gestion de l'état initial du système. Cet état initial correspond à $T = \varnothing$ et est trivialement défini par l'axiome ACW_2.

$$ACW_2(T) \stackrel{def}{\equiv} T = \varnothing$$

De plus, il faut autoriser le premier accès de chaque sujet. Ceci est formalisé dans la règle RCW_1.

$$RCW_1(T, css) \stackrel{def}{\equiv} \forall it \in T, src(it) \neq css$$

Chinese Wall *-Property

La propriété **CW *-Property** est définie dans notre formalisme par $CWSP$.

Propriété 4.7.2 (CWSP) *Soit un sujet $css \in \mathcal{CSS}$, il peut écrire dans un objet $cso_1 \in \mathcal{CSO}$, c'est-à-dire $css \overset{T}{\triangleright} cso_1$, si et seulement si il respecte les deux conditions suivantes :*

- *$CWSR$ ou ACW_2 ou RCW_1 est vrai ou cso_1 est publique ($sane(cso_1) = VRAI$).*
- *Tous les objets non nettoyés accédés par css font parti du même CD que cso_1.*

$$\begin{aligned} CWSP(css, cso_1) \stackrel{def}{\equiv} (&CWSR(T, css, cso_1) \vee ACW_2(T) \vee RCW_1(T, css) \\ &\vee sane(cso_1) = VRAI) \wedge \\ (&\forall it \in T, src(it) = css, sane(tgt(it)) = FAUX, cd(tgt(it)) = cd(cso_1)) \end{aligned}$$

Application de Muraille de Chine

Nous appliquons la Muraille de Chine à quatre entreprises (Renault, Peugeot, Orange, SFR) séparées en deux groupes de concurrence (automobile et telecom), plus un groupe contenant les données publiques dites saines. Le découpage est visible sur la figure 4.12.

La première interaction (#1), $user1\_t \overset{T}{\underset{file:read}{\rightharpoonup}} [1025, 1036]\ renault\_t$, est autorisée car le sujet $user1\_t$ n'a pas encore interagi avec le système.

La seconde interaction (#2), $user1\_t \overset{T}{\underset{file:read}{\rightharpoonup}} [1038, 1041]\ peugeot\_t$, est refusée car $CWSR$ n'est pas respectée.

La troisième interaction (#3), $user2\_t \overset{T}{\underset{file:read}{\rightharpoonup}} [1045, 1049]\ orange\_t$, est autorisée car le sujet $user2\_t$ n'a pas encore interagi avec le système.

La quatrième interaction (#4), $user1\_t \overset{T}{\underset{file:read}{\rightharpoonup}} [1055, 1063]\ peugeot\_public\_t$, est autorisée car l'objet fait partie de l'ensemble d'informations qualifiées de publiques.

La cinquième interaction (#5), $user1\_t \overset{T}{\underset{file:read}{\rightharpoonup}} [1068, 1072]\ orange\_t$, est autorisée car le sujet $user1\_t$ n'a pas encore initié d'interaction avec l'ensemble de concurrence $telecom$.

La sixième interaction (#6), $user1\_t \overset{T}{\underset{file:write}{\rightharpoonup}} [1078, 1081]\ orange\_t$, est refusée car elle ne respecte pas la deuxième condition de $CWSP$ qui impose, en pratique, la lecture dans seul CD.

La septième interaction (#7), $user2\_t \overset{T}{\underset{file:read}{\rightharpoonup}} [1085, 1092]\ orange\_t$, est autorisée car $user2\_t$ a déjà lu des données dans ce CD.

La huitième interaction (#8), $user2\_t \xrightarrow[file{:}write]{T} [1095, 1099]\, peugeot\_t$, est interdite car $user2\_t$ a déjà lu dans un autre CD (condition 2 de $CWSP$).

Si les interactions #6 et #8 avaient été autorisées, elles auraient formé un flux d'information indirect permettant de transférer de l'information de $renault\_t$ vers $peugeot\_t$. On voit que l'application de la muraille de chine permet de prévenir les flux indirects.

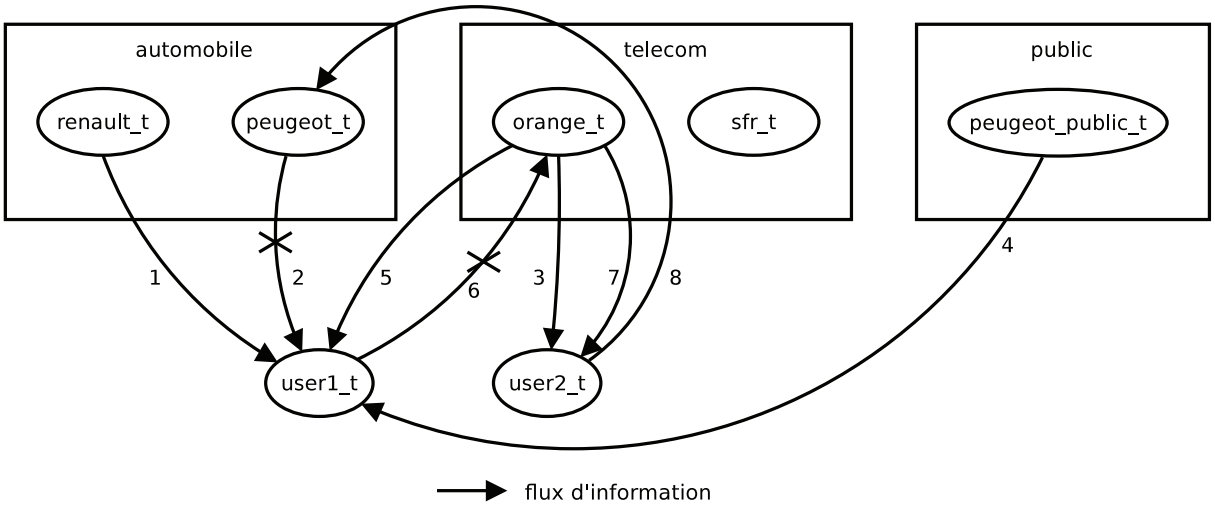

FIGURE 4.12 – Représentation d'une trace système sans rupture de la muraille de chine

4.7.2 Confinement des Données à la Volée : Un nouveau modèle de protection dynamique

Introduction

Jusqu'ici nous avons formalisé des propriétés existantes, nous allons maintenant montrer que notre formalisme permet de proposer de nouveaux modèles de sécurité permettant de résoudre des problèmes complexes de protection. La majorité des difficultés dans un système complet est de fournir un modèle de protection qui empêche les flux d'information indésirables, tout en rendant le système utilisable. Les résultats obtenus par le projet SPACLik [Briffaut et al., 2009b] dans le Défi Sécurité de l'ANR SEC&SI [ANR, 2008] montrent clairement qu'il y a nécessité d'une protection en profondeur prenant bien en compte la dynamique du système. En effet, un internaute peut, par exemple, être amené à faire des usages très variés comme accéder aux courriers électroniques, accéder à ses comptes banquaires, interagir avec des services sociaux, utiliser des progiciels en ligne (ERP), etc. Le grand nombre d'usages introduit obligatoirement une dynamicité du système et donc de la protection qui doit être appliquée. L'approche définie par le projet SPACLik propose de prendre en compte cette dynamicité. Cependant, elle se fait au moyen d'une complexité plus importante. Afin de limiter cette complexité, il est souvent nécessaire de placer le système dans un état puit qui limite les usages ultérieurs. C'est pourquoi nous proposons un modèle dynamique qui réduit la complexité de la protection et permet des usages plus flexibles.

Modèle Nous proposons un nouveau modèle de protection dynamique pour le composant central qu'est le navigateur Internet. Il s'agit de contrôler les flux d'informations en appliquant le confinement de données à la volée dans un environnement de traitement d'informations parallèles. Nous prenons comme principe d'environnement de traitement parallèle un navigateur Internet qui

accéde parallèlement (via plusieurs onglets et/ou processus) à un ensemble de services web. Pour autant, ce modèle s'applique à n'importe quels systèmes de traitement parallèles comme un serveur web ou une base de données. Le but de notre propriété est de pouvoir, de manière simple et dynamique, isoler les différents types d'information traités par le navigateur pour contrôler les flux d'information tout en gardant un fonctionnement acceptable pour l'utilisateur.

L'objectif est que cette propriété soit simple à utiliser, même par un utilisateur final sans connaissance particulière d'un système d'information. La protection s'applique à l'ensemble du système intéragisant avec le navigateur. Par exemple, on veut notament protéger le reste du système des données provenant du navigateur.

Domaines Les données traitées par le navigateur peuvent être découpées en un ensemble de domaines qui ne doivent en aucun cas communiquer entre eux. Un exemple simple est que l'ensemble des services web de eBanking peuvent communiquer entre eux pour, par exemple, permettre de transférer de l'argent sur son compte PayPal depuis le site web de sa banque. Par contre, les services de eBanking ne doivent jamais communiquer avec les services "sociaux", tel que Facebook ou Twitter, pour éviter toutes fuites d'informations bancaires personnelles. Contrairement à la Muraille de Chine, les domaines sont des ensembles de services coopérants et pas des concurrents.

Nous proposons donc de classifier par domaine les sites accédés par le navigateur. Par exemple, on peut associer au domaine $eBanking$ l'ensemble des sites suivants : $\{http://www.credit-financier.fr, http://www.banque-epargne.fr\}$. De manière similaire, on peut définir un domaine $social = \{http://www.facebook.com, http://www.twitter.com\}$.

Domaine particulier : Bac à sable Les sites n'étant pas classifiés dans un domaine sont automatiquement placés dans un domaine particulier *sandbox* (ou *bac à sable*). Nous définissons un objet spécial $unknown\_url\_t$ qui regroupe l'ensemble de ces sites non classifiés. Quand une entité du système tente d'accéder à un site qui n'est pas classifié, elle accède implicitement à cet objet. Quand une entité accède à $unknown\_url\_t$, elle est étiquetée avec le domaine *sandbox*.

De plus, le domaine *sandbox* est découpé en plusieurs **sous-domaines**. De cette façon, chaque site non classifié est associé à un sous domaine du type bac à sable. Cela permet d'isoler les différentes sites non classifiés et d'éviter les interférences entre eux.

Finalement, notre modèle régit l'accès à l'objet $unknown\_url\_t$ en autorisant uniquement une seule lecture. Cela permet d'éviter les flux d'information indirects entre des sites non classifiés.

Domaine particulier : Public De manière similaire au modèle de la muraille de chine, nous définissons un second domaine particulier, appelé public, qui contient l'ensemble des sites et objets comportant des données publiques qui peuvent être librement accédées par n'importe quel autre domaine. Volontairement dans notre modèle, nous n'introduisons pas de possibilité d'écrire dans ce domaine public. En effet, cette action revient à une opération de déclassification d'informations et doit donc être encadrée très strictement et effectuée par une entité de confiance.

Utilisabilité Dans le modèle de la muraille de chine, tous les objets du système doivent être correctement étiquetés. Ceci est une tâche fastidieuse qui demande, en plus, une connaissance très pointue du système et de sa sécurité.

A l'inverse, notre approche requiert seulement l'étiquetage d'un petit sous ensemble d'objets (et d'aucun sujet). Plus précisément, notre modèle prend en compte automatiquement la création et la gestion des domaines. Cela revient à un mécanisme d'étiquettage (labelisation) automatique des entités du système.

Le principe retenu pour l'étiquetage automatique consiste à contaminer les entités en fonction des flux d'information. Les objets à la source d'un flux vont contaminer les entités destinataires

du flux. Cette contamination est positive car elle permet de suivre et donc de contrôler les flux d'information entre domaines.

Définition Informelle De manière informelle, nous définissons la propriété de sécurité comme suit :

Définition 4.7.1 *Soit un système et un ensemble d'objets étiquetés, tout flux d'information apparaissant sur le système doit respecter l'une des règles suivantes :*

A. Se dérouler au sein d'un même domaine ;
B. Avoir pour source d'information un domaine public ;
C. Viser une entité n'ayant pas encore de domaine ;
D. Se dérouler au sein d'un sous domaine de sandbox ;

De plus, tout flux d'information est étiqueté par le domaine du contexte source et, transitivement, il étiquette l'ensemble des entités cibles auquelles il accède. De plus, chaque entité ne peut avoir qu'un seul et unique domaine.

Notre propriété est dynamique suivant deux axes :
- toute décision d'autorisation ou de refus ne dépend pas d'une politique statique mais des accès ayant eu lieu précédemment.
- la propriété ne requiert que la classification d'un sous ensemble d'objets. Elle utilise ensuite l'étiquetage dynamique des autres entités en utilisant le principe de contamination.

Formalisation

Nous définissons l'ensemble $\mathcal{DOM}$ qui contient l'ensemble des domaines utilisés sur le système. La fonction $dom()$ retourne le domaine d'une entité (ou $\varnothing$ quand elle n'en a pas) : dom : $\mathcal{CS} \rightarrow \mathcal{DOM}$. Par exemple, $dom(paypal\_t)$ retourne $eBanking$.

Définition 4.7.2 (Règle A) *Soit un flux d'information fi, il respecte la règle A de la propriété* ***ssi*** *fi a pour source une entité étiquetée et pour destination une seconde entité étiquetée avec le même domaine sauf si le domaine est $sandbox$.*

$$A(fi) \stackrel{def}{\equiv} dom(src(fi)) = dom(tgt(fi)) \wedge dom(src(fi)) \neq sandbox$$

Définition 4.7.3 (Règle B) *Soit un flux d'information fi, il respecte la règle B de la propriété* ***ssi*** *fi a pour source une entité étiquetée avec le domaine $public$.*

$$B(fi) \stackrel{def}{\equiv} dom(src(fi)) = public$$

Définition 4.7.4 (Règle C) *Soit un flux d'information fi, il respecte la règle C de la propriété* ***ssi*** *fi a pour destination une entité pas encore étiquetée.*

$$C(fi) \stackrel{def}{\equiv} dom(tgt(fi)) = \varnothing$$

Si la règle C est vraie, il faut affecter des domaines aux entités n'en disposant pas encore. La régle suivante régit l'étiquetage automatique par domaine :

Définition 4.7.5 (Règle d'Étiquetage) *Soit un flux d'information fi, si la règle C est vraie pour fi alors :*
- *si la destination est un sujet non étiqueté alors il prend le domaine de l'objet lu.*
- *si la destination est un objet non étiqueté alors il prend le domaine du sujet écrivain.*

$$LabelDyn(fi) \stackrel{def}{\equiv} C(fi), dom(tgt(fi)) \leftarrow dom(src(fi)).$$

Sandbox : un domaine particulier Afin de définir la règle D, nous avons besoin de définir les sous domaines de *sandbox*. Nous définissons deux ensembles appelés respectivement "ensemble des sous domaines affectés" ($\mathcal{SDOM}$) et "ensemble des sous domaines affectables" ($\mathcal{SDOM}_{pool}$). $\mathcal{SDOM}$ contient l'ensemble des sous domaines qui sont actuellement utilisés sur le système. $\mathcal{SDOM}_{pool}$ contient l'ensemble des sous domaines potentiellement utilisables par le système. La fonction $sdom$ retourne le sous-domaine d'un contexte : $sdom : \mathcal{CS} \rightarrow \mathcal{SDOM}$.

La règle suivante régit l'étiquetage automatique par sous-domaines.

Définition 4.7.6 (Règle d'Étiquetage par sous domaine) *Soit un flux d'information fi, si il a pour destination un sujet n'ayant pas de domaine et pour source le contexte $unknown\_url\_t$, alors le sujet se voit affecter le domaine $sandbox$ et un sous domaine disponible.*

$$\begin{aligned} LabelSubDyn(fi) \stackrel{def}{\equiv}\ & src(fi) = unknown\_url\_t \wedge dom(tgt(fi)) = \varnothing, \\ & dom(tgt(fi)) \leftarrow sandbox \wedge sdom(tgt(fi)) \leftarrow pull(\mathcal{SDOM}_{pool}) \end{aligned}$$

Comme nous l'avons vu précédemment, il est nécessaire de contrôler la lecture du contexte $unknown\_url\_t$ par un sujet ayant déjà un domaine. En pratique, cela permet d'éviter les flux d'information indirects entre des sites non classifiés.

Définition 4.7.7 (Règle d'accès aux URLs non classifiées) *Soit un flux d'information fi, si la source de fi est le contexte $unknown\_url\_t$ et la destination fi est un sujet non étiqueté alors fi est autorisé.*

$$ACUnknownURL(fi) \stackrel{def}{\equiv} src(fi) = unknown\_url\_t \wedge dom(tgt(fi)) \neq \varnothing$$

Définition 4.7.8 (Règle D) *Soit un flux d'information fi, si fi a pour source une entité étiquetée $sandbox$ et a un sous domaine alors la destination de fi ne peut être qu'une entité non étiquetée ou une entité étiquetée avec le couple domaine, sous domaine de la source de fi.*

$$\begin{aligned} D(fi) \stackrel{def}{\equiv}\ & dom(src(fi)) = sandbox, sdom(src(fi)) \neq \varnothing, \\ & (dom(tgt(fi)) = \varnothing \vee sdom(tgt(fi)) = sdom(src(fi))) \end{aligned}$$

Le nouveau modèle de confinement dynamique que nous proposons est défini par la propriété suivante :

Définition 4.7.9 (Propriété de Confinement Dynamique d'un Système) *Soit une trace T représentant un système, une des quatre règles de la propriété de Confinement Dynamique doit être respectée pour chaque flux d'information. De plus, si la règle C est vraie, la règle d'étiquetage $LabelDyn$ doit être appliquée. Si la règle D est vraie alors les deux règles d'isolation par sous domaine ($LabelSubDyn$ et $ACUnknownURL$) doivent être appliquées.*

$$\begin{aligned} ConfDynSys(T) \stackrel{def}{\equiv}\ & \forall fi \in \mathcal{FIDSE}, A(fi) \vee B(fi) \vee (C(fi) \Rightarrow LabelDyn(fi)) \\ & \vee (D(fi) \Rightarrow (LabelSubDyn(fi) \wedge ACUnknownURL(fi))) \end{aligned}$$

Il est à noter que nous n'avons pas besoin de traiter explicitement les flux d'information indirects dans ce modèle car ceux-ci sont pris en compte de manière implicite comme établi par le théorème suivant :

Théorème 4.7.1 *Soit css_1 et css_2 deux sujets ayant des domaines différents alors il n'est pas possible d'avoir $css_1 \overset{T}{\rhd\!\!\rhd} css_2$ ou $css_2 \overset{T}{\rhd\!\!\rhd} css_1$.*

Preuve 4.7.1 *(Par l'absurde)*

Soit css_1 et css_2 deux sujets ayant respectivement le domaine A et B. Supposons qu'un flux indirect existe de css_1 vers css_2. Pour qu'il existe un flux indirect, il y a au moins un contexte partagé cso. Il existe donc un flux $css_1 \overset{T}{\rhd} cso$ et un flux $cso \overset{T}{\rhd} css_2$ qui transitivement correspondent au flux indirect $css_1 \overset{T}{\rhd\!\!\rhd} css_2$. Or, l'existance de $css_1 \overset{T}{\rhd} cso$ implique le respect du modèle donc que cso a pour domaine A. De manière similaire, l'existance de $cso \overset{T}{\rhd} css_2$ implique le respect du modèle donc que cso ait pour domaine B. Hors, ceci est conflictuel avec la définition du modèle puisqu'une entité ne peut pas avoir deux domaines. □

Exemples

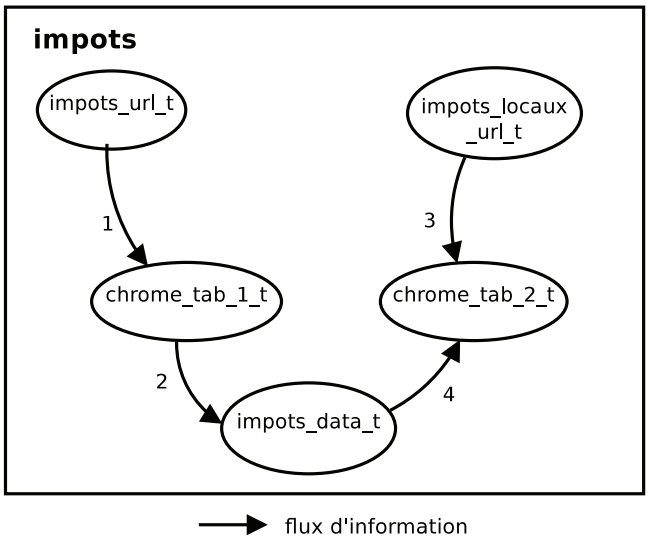

FIGURE 4.13 – Représentation d'une trace système respectant le confinement dynamique

L'exemple exposé dans la figure 4.13 utilise l'étiquetage initial des contextes décrit dans la première ligne du tableau 4.1. Ce tableau fait apparaître le domaine classifié $impot$ et le domaine non classifié $\varnothing$. Au départ, le domaine $impot$ contient deux URL. Cela correspond à l'étiquetage minimal que requiert notre modèle. Les trois contextes n'ayant pas de domaine correspondent respectivement à deux onglets du navigateur Google Chrome et un objet contenant des données ($data\_t$).

Un premier flux d'information a lieu de $impots\_url\_t$ vers $chrome\_tab1\_t$. $chrome\_tab1\_t$ n'a pas encore de domaine, le flux d'information est autorisé et $chrome\_tab1\_t$ est placé dans le domaine $impots$. Le tableau 4.1 qui montre l'évolution des domaines présente un nouvelle état (ligne 2) correspondant à la modification du domaine $impot$.

Le second flux d'information allant de $chrome\_tab1\_t$ à $data\_t$ est autorisé puisque $data\_t$ n'a pas de domaine. L'objet $data\_t$ est ajouté au domaine $impots$ (troisième ligne du tableau 4.1).

Un troisième flux d'information allant de $impots\_locaux\_url\_t$ vers $chrome\_tab2\_t$ est autorisé car le sujet $chrome\_tab2\_t$ n'a pas encore de domaine. Le sujet se voit affecté au domaine $impots$ (ligne 4 du tableau 4.1).

État	Domaine : ∅	Domaine : impôts
0	$chrome\_tab1\_t$, $chrome\_tab2\_t$, $impots\_data\_t$	$impots\_url\_t$ $impots\_locaux\_url\_t$
1	$chrome\_tab2\_t$, $impots\_data\_t$	$impots\_url\_t$ $impots\_locaux\_url\_t$, $chrome\_tab1\_t$
2	$chrome\_tab2\_t$	$impots\_url\_t$ $impots\_locaux\_url\_t$, $chrome\_tab1\_t$, $impots\_data\_t$
3		$impots\_url\_t$ $impots\_locaux\_url\_t$, $chrome\_tab1\_t$, $impots\_data\_t$, $chrome\_tab2\_t$
4		$impots\_url\_t$ $impots\_locaux\_url\_t$, $chrome\_tab1\_t$, $impots\_data\_t$, $chrome\_tab2\_t$

TABLE 4.1 – Tableau des différents domaines relatifs à la figure 4.13

Le quatrième flux allant de $impots\_data\_t$ vers $chrome\_tab2\_t$ est autorisé car il a lieu entre deux contextes ayant le même domaine.

Ce scénario illustre la règle C associée à l'étiquettage automatique réalisé par $LabelDyn$ et la règle A autorisant les flux au sein d'un domaine.

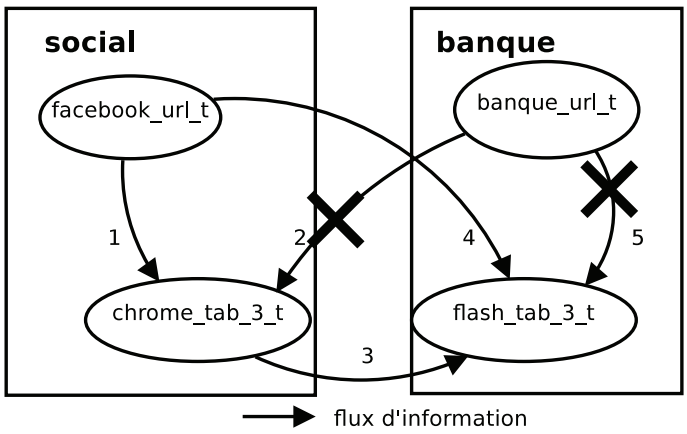

FIGURE 4.14 – Représentation d'une trace système avec rupture du confinement dynamique

Dans ce second exemple exposé, la figure 4.14 est associée à l'état initial exposé dans la ligne 1 du tableau 4.2.

Un premier flux va de $facebook\_url\_t$ vers $chrome\_tab3\_t$ ce qui amène l'étiquetage de $chrome\_tab3\_t$ par le domaine $social$ (ligne du tableau 4.2).

Un second allant de $banque\_url\_t$ vers $chrome\_tab3\_t$ est refusée car les deux contextes ont des domaines différents.

Le troisième flux d'information de $chrome\_tab3\_t$ vers $flash\_tab3\_t$ est autorisé et conduit l'entrée de $flash\_tab3\_t$ dans le domaine $social$ (ligne 4 du tableau 4.2)

Un quatrième flux est autorisé car les deux contextes partagent le domaine $social$.

Ce scénario illustre les mêmes règles que le premier exemple. Il montre l'isolation fournie entre les domaines $social$ et $banque$ et illustre l'absence de flux indirect entre ces deux domaines.

Un troisième exemple illustré par la figure 4.15 et par le tableau 4.3 montre l'usage du domaine $sandbox$.

État	Domaine : ∅	Domaine : ebanking	Domaine : social
0	$chrome\_tab3\_t$, $flash\_tab3\_t$	$banque\_url\_t$	$facebook\_url\_t$
1	$flash\_tab3\_t$	$banque\_url\_t$	$facebook\_url\_t$, $chrome\_tab3\_t$
2	$flash\_tab3\_t$	$banque\_url\_t$	$facebook\_url\_t$, $chrome\_tab3\_t$
3		$banque\_url\_t$	$facebook\_url\_t$, $chrome\_tab3\_t$, $flash\_tab3\_t$
4		$banque\_url\_t$	$facebook\_url\_t$, $chrome\_tab3\_t$, $flash\_tab3\_t$

TABLE 4.2 – Tableau des différents domaines relatifs à la figure 4.14

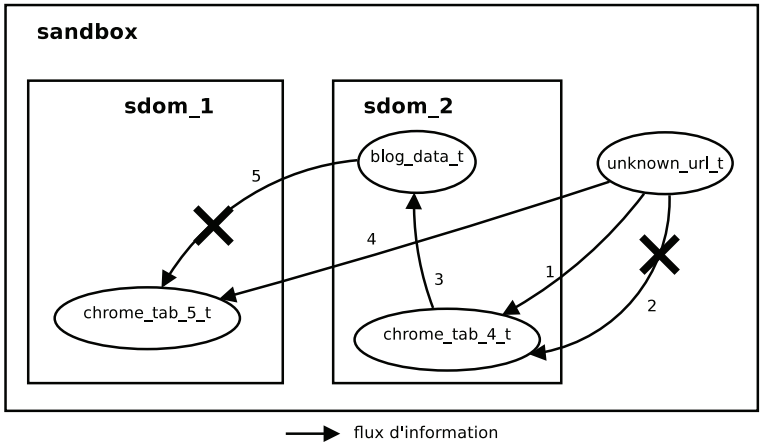

FIGURE 4.15 – Représentation d'une trace système avec rupture du confinement dynamique

Un premier flux allant de $unknown\_url\_t$ vers $chrome\_tab4\_t$ est autorisée. $chrome\_tab4\_t$ rentre dans le domaine $sandbox$ et dans le sous domaine $sandbox\_1$ (ligne 2 du tableau 4.3).

Un deuxième flux similaire de $unknown\_url\_t$ vers $chrome\_tab4\_t$ est refusé car $chrome\_tab4\_t$ a déjà un sous-domaine.

Un troisième flux allant de $chrome\_tab4\_t$ vers $blog\_data\_t$ est autorisé car $blog\_data\_t$ n'a pas de domaine. $blog\_data\_t$ rentre lui aussi dans le sous domaine $sandbox\_1$ (ligne 3 du tableau 4.3).

Un quatrième flux allant de $unknown\_url\_t$ vers $chrome\_tab5\_t$ est autorisé, $chrome\_tab5\_t$ entre dans le sous domaine $sandbox$ (ligne 4 du tableau 4.3).

Un cinquième flux allant de $blog\_data\_t$ vers $chrome\_tab5\_t$ est interdit car il se déroule entre deux sous domaines différents.

Cette exemple illustre la règle D ainsi que l'utilisation des labélisations par sous-domaine ($LabelSubDyn$) et du contrôle d'accès sur l'objet $unknown\_url\_t$ ($ACDefaultURL$). On voit l'intérêt du domaine $sandbox$ qui permet d'éviter les flux indirects entre différentes URL non classifiées.

Originalité

Notre modèle offre une protection dynamique originale. Cette protection ne peut pas être obtenue par le modèle BLP qui est statique. Dans notre cas, les règles de protection sont calculées dynamiquement en fonction des activités comme nous l'avons illustré dans les exemples précédents.

De même, le modèle **Chinese Wall** ne permet pas de réaliser la même protection que notre modèle. En effet, **Chinese Wall** ne dispose pas de fonction d'étiquettage automatique.

État	Domaine : ∅	Domaine : sandbox	$sandbox\_1$	$sandbox\_2$
0	$chrome\_tab4\_t$, $chrome\_tab5\_t$, $blog\_data\_t$	$unknown\_url\_t$		
1	$chrome\_tab5\_t$, $blog\_data\_t$	$unknown\_url\_t$, $chrome\_tab4\_t$	$chrome\_tab4\_t$	
2	$chrome\_tab5\_t$, $blog\_data\_t$	$unknown\_url\_t$, $chrome\_tab4\_t$	$chrome\_tab4\_t$	
3	$chrome\_tab5\_t$	$unknown\_url\_t$, $chrome\_tab4\_t$, $blog\_data\_t$	$chrome\_tab4\_t$, $blog\_data\_t$	
4		$unknown\_url\_t$, $chrome\_tab4\_t$ $blog\_data\_t$	$chrome\_tab4\_t$, $blog\_data\_t$	$chrome\_tab5\_t$
5		$unknown\_url\_t$, $chrome\_tab4\_t$ $blog\_data\_t$	$chrome\_tab4\_t$, $blog\_data\_t$	$chrome\_tab5\_t$

TABLE 4.3 – Tableau des différents domaines (et sous domaines) relatifs à la figure 4.15

Ce modèle est issu de l'expérience que nous avons mené dans le cadre du défi sécurité. En effet, dans le projet SPACLik [Briffaut et al., 2009b], une protection dynamique a du être mise en place pour garantir la protection voulue. Cependant dans l'approche du défi [ANR, 2008], l'ensemble des contextes et des domaines est statique. L'expression de la dynamicité des règles doit être défini au moyen d'un automate d'états. Si cette méthode est relativement efficace, elle reste complexe. De plus, elle ne propose pas de domaines de $sandbox$ et ne dispose que de trois domaines statiques. A l'inverse, ce modèle est plus simple et permet plus facilement de prendre en compte un grand nombre de domaines et de sous domaines. Finalement, il ne demande pas l'étiquetage de l'ensemble des entités du système.

4.8 Conclusion

Grâce à la formalisation des activités systèmes donnée dans le chapitre 3, nous avons, dans ce chapitre, pu formaliser un certain nombre de propriétés.

Nous avons tout d'abord repris des propriétés de la littérature pour les formaliser.

Ainsi, nous avons donné une définition de l'exécution indirecte. Cette propriété est utile pour empêcher des exécutions reposant sur des canaux cachés afin de violer soit l'intégrité, soit la confidentialité.

Nous avons aussi formalisé un certain nombre de propriétés d'intégrité. Ainsi, nous avons donné une définition formelle de la propriété d'intégrité des objets considérée essentiellement dans [ITSEC, 1991]. Cette propriété est essentielle puisqu'elle permet de contrôler les modifications des objets. L'intégrité des sujets, souvent appellée non-interférence, permet d'éviter la contamination des sujets. Une propriété est définie pour éviter les attaques reposant sur les situations de concurrence. Cette propriété permet de protéger les ressources partagées. L'intégrité des domaines est utile pour réaliser ce qui correspond à un bac à sable. Cette propriété permet d'isoler un ensemble de contextes du reste du système. Une propriété permet de définir des exécutables de confiance. En pratique, elle permet aux sujets d'exécuter des objets sûrs.

Nous définissons une seule propriété de confidentialité. Cette propriété est générale puisqu'elle permet de contrôler aussi bien les flux d'information directs qu'indirects.

Ensuite, nous avons développé une propriété associée aux modèles multi-niveaux. Nous proposons une formalisation d'un des modèles de Bell&LaPadula. Ce modèle est celui qui est le plus adapté à un système d'exploitation. Bien qu'il s'agisse d'un modèle de confidentialité, il se classe parmi les approches MLS.

Plusieurs propriétés permettent d'éviter les abus de privilèges. Tout d'abord, la séparation dynamique de privilèges oblige plusieurs sujets à collaborer pour réaliser une tâche. L'intérêt de cette propriété est son caractère dynamique puisque les rôles des sujets ne sont pas définis à l'avance.

L'absence de changement de contexte est une propriété assez élémentaire. Cette propriété est intéressante puisqu'elle permet d'empêcher les changements de contextes successifs pouvant conduire à usurper des privilèges. Finalement, le respect d'une politique peut être une première approche pour détecter certaines violations. En pratique, elle peut être utilisée pour mettre en oeuvre un système mandataire contrôlant les activités directes.

Nous avons formalisé une propriété dynamique classique qui est la Muraille de Chine. Elle permet d'éviter les délis d'initiés.

Afin de montrer que notre formalisme permet de modéliser des nouvelles propriétés, nous avons proposé un nouveau modèle de propriété dynamique. Celle-ci a l'avantage de pouvoir à la fois constituer des domaines de protections et étiqueter dynamiquement les contextes. Le caractère dynamique est donc à deux niveaux. D'une part, l'étiquetage est dynamique. D'autre part, la politique de protection est calculée dynamiquement en fonction des activités du système. Il s'agit d'une propriété qui permet une protection en profondeur tout en étant simple d'utilisation.

Il est clair que nous pouvons aisément définir d'autres propriétés qui permettent de prendre des besoins spécifiques. Un certain nombre d'autres propriétés ont été formalisées au moyen de ce langage. L'intérêt de cette approche est que la formalisation d'une nouvelle propriété reste simple puisque nous proposons un jeu d'opérateurs permettant de prendre en compte aisement à la fois les flux directs et indirects.

En pratique, cette formalisation est directement utilisable pour contrôler les activités systèmes. En effet, elle repose sur l'analyse d'une trace d'exécution. Une telle trace peut être facilement réalisée soit en capturant les appels systèmes, soit en utilisant des traces existantes. Nous verrons dans le chapitre suivant que l'application à la protection d'un système se réduit alors à calculer au moyen de cette trace un graphe représentant les flux que nous souhaitons contrôler. L'avantage de la formalisation est qu'elle propose un moyen via le regroupement d'interactions de pouvoir minimiser la taille de ce graphe. C'est cette partie là que nous allons développer dans le chapitre suivant.

Chapitre 5

Implantation du langage

5.1 Introduction

Dans ce chapitre, nous allons montrer comment mettre en oeuvre notre langage. En pratique, il s'agit d'offrir un moyen de compiler ce langage. Pour cela, nous proposons un graphe de flux d'information qui contient tous les flux nécessaires aux opérateurs de notre langage. Nous montrons comment analyser ce graphe pour fournir des fonctions supportant nos opérateurs. Grâce à ces fonctions, nous proposons une méthode de compilation. La complexité obtenue est polynomiale avec le nombre de contextes. Ainsi, l'ensemble des propriétés formalisables avec notre langage peut aisément être analysé avec des complexités qui, en pratique, restent faible.

5.2 Graphe d'Interactions

Une trace d'interactions estampillées peut être représentée sous forme d'un graphe. Dans cette représentation, chaque interaction (composée d'un évènement d'entrée et de sortie, voir la section 3.2.2) contenue dans la trace est représentée sous la forme d'un arc orienté contenant :

– l'opération élémentaire de l'interaction ;
– sa date d'occurrence.

Par exemple, d'après la trace contenue dans le listing 5.1, nous construisons le graphe d'interactions présenté dans la figure 5.1. La première interaction estampillée est représentée dans le graphe par l'arc #1. Cet arc contient deux arguments, 2758 et 2602, respectivement la date de début et de fin de l'interaction.

Dans ce graphe, le nombre d'arc est potentiellement infini car il y a une interaction estampillée par appel système. Ce graphe n'est donc pas exploitable en pratique. C'est pourquoi nous proposons de construire le graphe de flux d'information.

5.3 Graphes de Flux d'Information

Comme nous l'avons développé dans le chapitre 4, notre langage utilise uniquement la notion de flux d'information qui sont, de plus, déclinés sous différents cas particuliers notamment les transitions. En pratique, nous maintenons donc un seul graphe de flux d'information représentant différents sous graphes que nous détaillons dans cette partie. Ce graphe représente les flux d'information directs. Les flux indirects sont calculés par recherche de chemin. Ce graphe est optimisé grâce aux flux d'information directs multiples estampillés. Contrairement au graphe d'interactions, la complexité de notre graphe de flux est bornée.

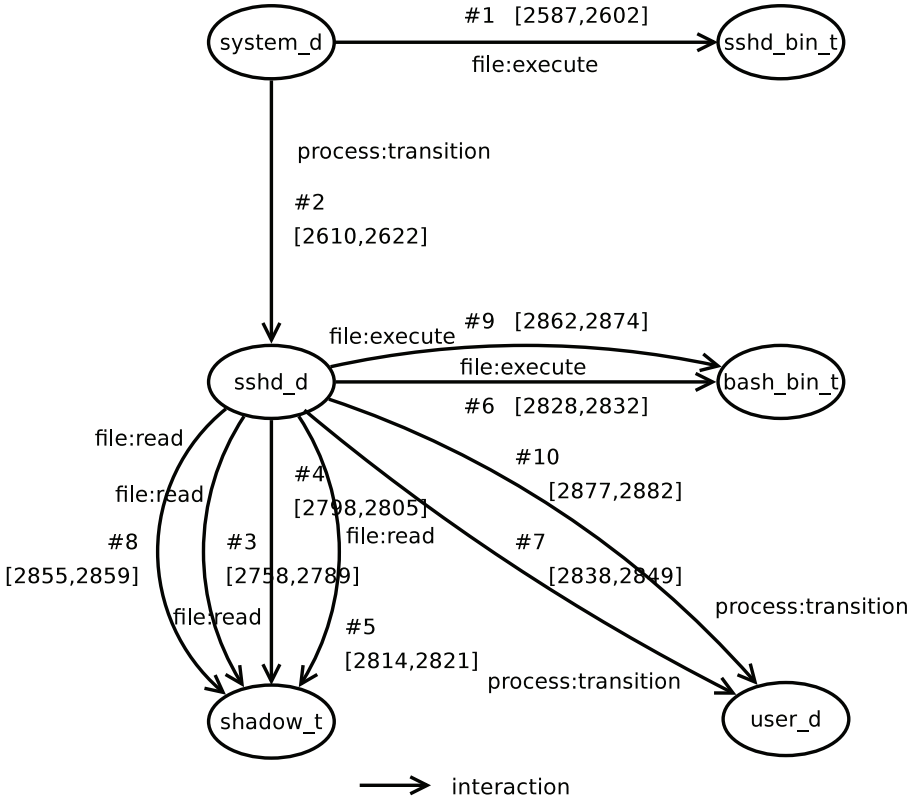

FIGURE 5.1 – Graphe d'interactions

5.3.1 Graphe de Flux d'Information

Listing 5.1 – "Trace représentant les actions passées d'un système"

```
system_d −file:execute−> [2587,2602] sshd_bin_t
system_d −process:transition−> [2610,2622] sshd_d
sshd_d −file:read−> [2758,2789] shadow_t
sshd_d −file:read−> [2798,2805] shadow_t
sshd_d −file:read−> [2814,2821] shadow_t
sshd_d −file:execute−> [2828,2832] bash_bin_t
sshd_d −process:transition−> [2838,2849] user_d
sshd_d −file:read−> [2855,2859] shadow_t
sshd_d −file:execute−> [2862,2874] bash_bin_t
sshd_d −process:transition−> [2877,2882] user_d
```

En utilisant le concept de flux d'information direct multiple (cf. section 3.2.6), nous proposons une représentation compacte des flux d'information directs.

Un arc du graphe de flux d'information (GFI) représente un flux d'information multiple. Cette approche permet d'avoir un nombre borné d'arcs et donc une complexité satisfaisante. Dans ce cas, le graphe comporte, au plus, deux arcs entre chaque couple de contexte de sécurité. Etant donné que l'un d'eux est un sujet, le nombre d'arcs potentiels a est donc en $O(|\mathcal{CSS}| \times |\mathcal{CS}| \times 2)$.

A titre d'exemple, le GFI correspondant au listing 5.1, est présenté dans la figure 5.2. Par exemple, les arcs #3, #4, #5 et #8 du graphe d'interactions (figure 5.1) sont compactés en un seul et unique arc #3 dans le GFI.

Calcul des flux indirects La recherche des flux indirects sur GFI peut être réalisée au moyen d'un algorithme de recherche de chemins en largeur. Cet algorithme a une complexité de $O(a+n)$ où a le nombre d'arcs et n le nombre de nœuds. La complexité d'une recherche de flux indirects, dans GFI, est donc en $O(|\mathcal{CS}| \times ((|\mathcal{CSS}| \times 2) + 1))$.

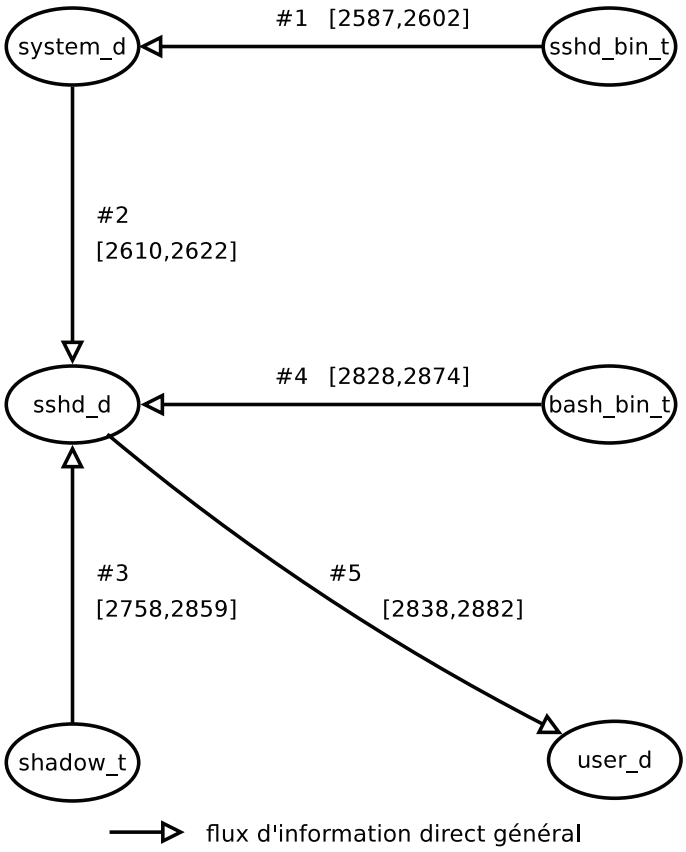

FIGURE 5.2 – GFI du listing 5.1

Concrètement, sur un système d'exploitation GNU/Linux avec la distribution Gentoo Hardened, il y a 878 contextes de sécurité dont 264 sujets. La complexité du calcul des flux indirects est donc $O(878 \times ((264 \times 2) + 1))$ soit $O(464462)$. En pratique, après avoir observé le système pendant plus de 32.10^6 d'interactions, le nombre de flux d'information différents (au sens des couples sources et cibles de contextes de sécurité) s'est stabilisé autour de 80000 ce qui correspond à une complexité pour les flux indirects en $O(80000+878)$ soit $O(80878)$. On voit donc qu'en pratique, on a une complexité beaucoup plus faible que dans le pire des cas.

5.3.2 Graphe de Flux d'Information Augmenté

Le GFI doit être augmenté pour prendre en compte le cas particulier des transitions. Nous appellons ce graphe : Graphe de Flux d'Information Augmenté (GFIA).

A titre d'exemple, utilisons à nouveau la trace contenue dans le listing 5.1, pour construire le GFIA équivalent représenté en figure 5.3. Il correspond à celui de la figure 5.2, augmenté des arcs de transitions présentes dans le listing. Par exemple, la deuxième interaction : $system\_d \overset{process:transition}{\underset{[2610,2622]}{\rightharpoonup}} sshd\_d$, est représentée par l'arc #2, sous la forme d'un flux d'information, et par l'arc #T1, sous la forme d'une transition.

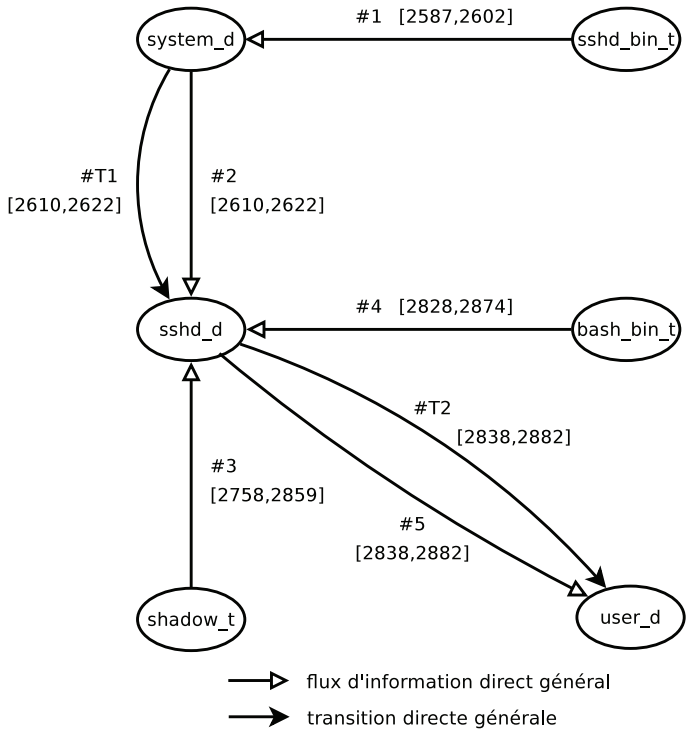

FIGURE 5.3 – GFIA du listing 5.1

Dans le GFIA, il peut y avoir, au plus, deux transitions entre chaque couple de sujets, soit $|\mathcal{CSS}| \times (|\mathcal{CSS}|-1) \times 2$. Le nombre d'arcs total du GFIA est donc celui du GFI plus ce nombre, soit, au plus $a = (|\mathcal{CSS}| \times |\mathcal{CS}| \times 2) + (|\mathcal{CSS}| \times (|\mathcal{CSS}|-1) \times 2)$, soit $(2 \times |\mathcal{CSS}| \times (|\mathcal{CS}| + (|\mathcal{CSS}|-1)))$.

La recherche des transitions indirectes revient à parcourir uniquement les arcs de transition. Donc, la complexité de recherche des transitions indirectes dans le GFIA est en $O((2 \times |\mathcal{CSS}| \times (|\mathcal{CSS}| - 1)) + |\mathcal{CSS}|)$.

Avec le système d'exploitation Gentoo Hardened contenant 878 contextes de sécurité dont 264 sujets, la complexité du GFIA est en $O(602448)$ soit une augmentation de 30% par rapport au GFI. Par contre, la recherche de transitions indirectes est uniquement en $O(69696)$ soit environ six fois moins que la recherche de flux d'information indirects.

Dans la réalité, la taille du GFIA est nettement plus faible. Nous avons observé au sein des 80 000 flux d'information différents, au plus 660 transitions. La complexité du graphe est donc réduite à $O((80000 + 660) + 878)$ soit $O(81538)$ soit une augmentation réelle de 0.81% par rapport au GFI.

5.3.3 Complexité des différents graphes

Le tableau 5.1 résume la complexité du GFIA qui est utilisée en pratique. Il présente le surcoût théorique et pratique du GFIA par rapport au GFI. Il compare aussi la complexité du GFIA avec celle du graphe d'interactions. On voit clairement que notre approche est très effiace.

Type de graphe	**Nb actions différentes**	**Nb actions réelles**		**Nb arcs stockés** a
Graphe d'interactions	$\|CSS\| \times \|CS\| \times \|eo\|$	32.10^6 IT		32.10^6
		$+\infty$		$+\infty$
GFI	$\|CSS\| \times \|CS\| \times 2$	80.10^3 Flux	$\leq 80.10^3$	
		$+\infty$		$\leq 2 \times \|CSS\| \times \|CS\|$
GFIA	$2\times(\|CSS\|\times(\|CS\|+(\|CSS\|-1))$	80.10^3 Flux	660 Trans	≤ 80660
		$+\infty$		$\leq 2 \times \|CSS\| \times (\|CS\|+(\|CSS\|-1))$

TABLE 5.1 – Complexité des différents graphes

Fonction	Type d'action	
$chercheInteraction$	$cs \stackrel{eq}{\rightarrow} cs$	$O(1)$
$chercheExecDirect$	$cs \blacktriangleright_x cs$	$O(1)$
$chercheFluxDirect$	$cs \blacktriangleright cs$	$O(1)$
$chercheTransDirecte$	$cs \blacktriangleright_t cs$	$O(1)$
$chercheUnFluxIndirect$	$cs \underset{[d_{ee_1}, d_{es_k}]}{\overset{T}{\rhd\!\!\rhd}} cs$	$O((((\lvert\mathcal{CSS}\rvert \times (\lvert\mathcal{CS}\rvert) \times 2) + \lvert\mathcal{CS}\rvert) \times 2)$
$chercheUneTransIndirecte$	$cs \underset{[d_{ee_1}, d_{es_k}]}{\overset{T}{\rhd\!\!\rhd_t}} cs$	$O((\lvert\mathcal{CSS}\rvert \times (\lvert\mathcal{CSS}\rvert - 1) \times 2) + \lvert\mathcal{CSS}\rvert)$
$chercheTousFluxIndirects$	$cs \underset{[d_{ee_1}, d_{es_k}]}{\overset{T}{\rhd\!\!\rhd}} cs$	$O((((\lvert\mathcal{CSS}\rvert \times (\lvert\mathcal{CS}\rvert) \times 2) + \lvert\mathcal{CS}\rvert) \times 2)$
$chercheToutesTransIndirectes$	$cs \underset{[d_{ee_1}, d_{es_k}]}{\overset{T}{\rhd\!\!\rhd_t}} cs$	$O((\lvert\mathcal{CSS}\rvert \times (\lvert\mathcal{CSS}\rvert - 1) \times 2) + \lvert\mathcal{CSS}\rvert)$
$chercheUnFluxGeneral$	$cs \underset{[d_{ee}, d_{es}]}{\overset{T}{\rhd\!\!\rhd\!\!\rhd}} cs$	$O((((\lvert\mathcal{CSS}\rvert \times (\lvert\mathcal{CS}\rvert) \times 2) + \lvert\mathcal{CS}\rvert) \times 2)$
$chercheUneTransGenerale$	$cs \underset{[d_{ee}, d_{es}]}{\overset{T}{\rhd\!\!\rhd\!\!\rhd_t}} cs$	$O((\lvert\mathcal{CSS}\rvert \times (\lvert\mathcal{CSS}\rvert - 1) \times 2) + \lvert\mathcal{CSS}\rvert)$
$chercheTousFluxGeneraux$	$cs \underset{[d_{ee}, d_{es}]}{\overset{T}{\rhd\!\!\rhd\!\!\rhd}} cs$	$O((((\lvert\mathcal{CSS}\rvert \times (\lvert\mathcal{CS}\rvert) \times 2) + \lvert\mathcal{CS}\rvert) \times 2)$
$chercheToutesTransGenerales$	$cs \underset{[d_{ee}, d_{es}]}{\overset{T}{\rhd\!\!\rhd\!\!\rhd_t}} cs$	$O((\lvert\mathcal{CSS}\rvert \times (\lvert\mathcal{CSS}\rvert - 1) \times 2) + \lvert\mathcal{CSS}\rvert)$

TABLE 5.2 – Fonctions implantées et ce qu'elles détectent

5.4 Mise en oeuvre des opérateurs

Nous avons introduit et expliqué la construction du Graphe de Flux d'Information Augmenté (GFIA) permettant de garder en mémoire les flux et transitions directs.

Le tableau 5.2 fournit une fonction pour chaque opérateur défini dans le chapitre 3. Ce tableau distingue deux méthodes pour chaque opérateur : une qui vérifie la présence du flux décrit par l'opérateur et une qui cherche tous les flux décrits par l'opérateur. Pour chacune de ces fonctions, nous donnons sa complexité.

Nous différencions deux types d'opérateurs directs : ceux dont les flux sont contenus dans le GFIA (flux d'information et transition) et les autres (interaction et exécution). Pour le second type, les fonctions $chercheExecDirect$ et $chercheInteraction$ permettent de comparer l'interaction courante avec le flux recherché.

5.4.1 Flux d'Information Direct

Etant donné que pour le premier type d'opérateur direct, les flux sont représentés dans le GFIA, il est nécessaire de faire une recherche de flux dans le graphe. Cependant pour des raisons d'optimisation, nous n'ajoutons pas à priori le flux correspondant à l'interaction courante dans le graphe. Le flux correspondant est ajouté à posteriori au graphe uniquement si l'interaction est autorisée.

La fonction $chercheFluxDirect$ vérifie d'abord si l'interaction estampillée courante ite_1 correspond au flux d'information direct recherché $fi_1 = cs_1 \overset{T}{\rhd} cs_2$. Si c'est le cas, elle retourne VRAI. Sinon elle cherche dans $GFIA$ si un arc existe allant de cs_1 à cs_2. Si c'est le cas, elle retourne VRAI sinon FAUX.

L'algorithme a une complexité en $O(1)$. En effet, le graphe peut être représenté sous forme d'une matrice $|\mathcal{CS}| \times |\mathcal{CS}|$. La recherche d'un arc revient à lire une cellule dans cette matrice.

De manière similaire, la fonction $chercheTransDirecte$ permet de vérifier la présence d'une

transition directe dans l'interaction courante et le GFIA. Sa complexité est similaire à celle de $chercheFluxDirect$.

5.4.2 Cherche un Flux d'Information Indirect

Nous rappellons que pour chaque opérateur (de flux et de transition) nous avons besoin d'une fonction qui vérifie l'existence d'un flux correspondant à l'activité décrite au moyen de l'opérateur. Comme précédemment, notre solution optimise la recherche du flux en ajoutant le flux dans le graphe, à posteriori, c'est-à-dire, si l'interaction correspondante est autorisée. Notre solution nécessite une fonction de recherche dans GFIA que nous détaillons d'abord. Cette fonction est ensuite utilisée pour implanter l'opérateur de flux indirect.

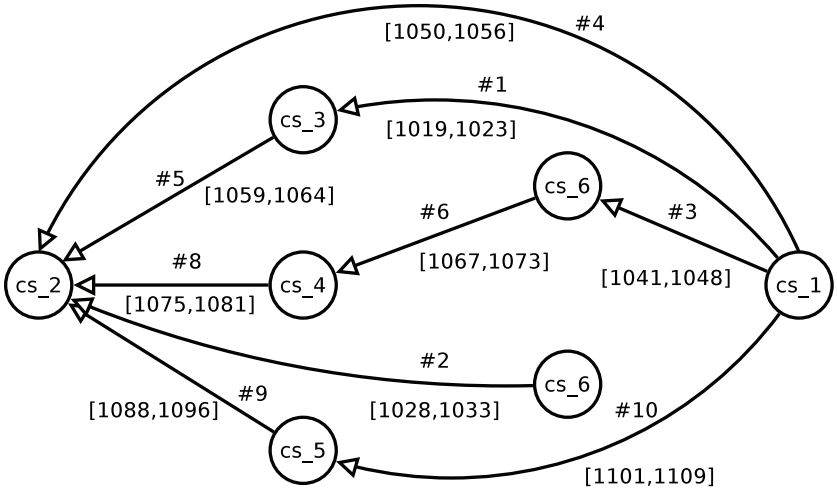

FIGURE 5.4 – Recherche d'un flux dans le Graphe de Flux d'Information Augmenté

Recherche dans le GFIA Tout d'abord, nous présentons $chercheUnFluxIndirectGraph$ qui permet de chercher un flux indirect dans le GFIA. Il correspond à une recherche en largeur dans notre GFIA. De plus, cette fonction vérifie le respect de la causalité qui est nécessaire pour les flux d'information indirects.

Bien que la fonction $chercheUnFluxIndirectGraph$ vise à chercher les flux indirects, elle est également capable de tester l'existance d'un flux direct dans le graphe. Ceci est rendu possible grâce au paramètre en entrée, le booléen $direct$. Cette fonction évite la gestion des cycles en ne parcourant qu'une fois chaque arc (file $arcDejaVu$). De plus, elle traite uniquement les flux ayant une relation de causalité, en ajoutant dans la file $fileCausale$ les arcs qui respectent la relation causale. Cette file évite de parcourir les chemins qui ne respectent pas notre relation de causalité.

Considérons le GFIA présenté dans la figure 5.4. La fonction $chercheUnFluxIndirectGraph(cs_1, cs_2, GFIA, FAUX)$ permet d'y rechercher un flux indirect $fii_1 = cs_1 \overset{T}{\Rrightarrow} cs_2$. L'algorithme 1 enfile tout d'abord dans la file $fileCausale$ (liste des arcs futurs à parcourir) tous les arcs de flux finissant par cs_2 : #2, #4, #5, #8, #9 puis la défile. L'arc #4 forme un flux d'information **direct** de cs_1 à cs_2 et n'est donc pas traité.

Pour les autres, tous les arcs précédents, qui respectent la relation de causalité, sont ajoutés dans $fileCausale$. Ainsi, pour #5, l'arc #1 est ajouté puisque #1 commence avant que #5 finisse ($1019 < 1064$). De la même façon, #6 est ajouté. Par contre, pour #9, #10 n'est pas ajouté puisqu'il ne respecte pas la relation de causalité.

L'algorithme continue à dépiler les arcs. Un chemin est trouvé #1→#5. L'algorithme retourne vrai.

De plus, il existe un autre chemin : #3→#6→#8 qui aurait pu être retenu si #1→#5 ne l'avait pas été.

$chercheUnFluxIndirectGraph$ est une application de recherche en largeur dans un graphe qui contient $\mathcal{CS}$ nœuds et $|\mathcal{CSS}| \times (|\mathcal{CS}|) \times 2$ arcs de flux. Sa complexité est donc en $O((|\mathcal{CSS}| \times (|\mathcal{CS}|) \times 2) + |\mathcal{CS}|)$. $chercheUnFluxIndirectGraph$ est donc applicable dans un temps polynomial.

Algorithm 1 Algorithme de détection d'un flux d'information indirect allant de cs_1 vers cs_2 dans un graphe $GFIA$:$chercheUnFluxIndirectGraph(cs_1, cs_2, GFIA, direct)$

Pré-conditions : $cs_1 \in \mathcal{CS}, cs_2 \in \mathcal{CS}$
  $fileCausale \Leftarrow listArcFinissantPar(cs_2)$
  $arcDejaVu \Leftarrow listArcFinissantPar(cs_2)$
  $arc_1 \Leftarrow$ Defile $fileCausale$
  $dernierArcDirect \Leftarrow$ dernier arc dans $listArcFinissantPar(cs_2)$
  $tailleSuperieureAUn \Leftarrow$ FAUX
  tant que $arc_1 \neq$ NULLE **faire**
    si $(src(arc_1) = cs_1)$**et**$((direct)$**ou**$(tailleSuperieureAUn))$ **alors**
      return VRAI
    sinon
      $listArcTemp \Leftarrow listArcFinissantPar(src(arc_1))$
      pour tout $arc_{tmp} \in listArcTemp$ **faire**
        si $(deb(arc_{tmp}) \leq fin(arc_1))$**et**($arc_{tmp}$ pas dans $arcDejaVu$) **alors**
          Enfile arc_{tmp} dans $fileCausale$
          Enfile arc_{tmp} dans $arcDejaVu$
        fin si
      fin pour
    fin si
    si $arc_1 = dernierArcDirect$ **alors**
      $tailleSuperieureAUn =$VRAI
    fin si
    $arc_1 \Leftarrow$ Defile $fileCausale$
  fin tant que
  return FAUX

Implantation de l'opérateur Nous proposons une fonction $chercheUnFluxIndirect(fii_1, ite_1, GFIA)$ qui correspond à l'implantation de l'opérateur $fii_1 = cs_1 \overset{T}{\Rrightarrow} cs_2$. Tout d'abord, la fonction vérifie si la destination du flux, $dest(fie_1)$, issu de l'interaction ite_1 est égale à la destination du flux recherché, $dest(fii_1)$. Si c'est le cas, la fonction utilise $chercheUnFluxIndirectGraph$ pour rechercher un flux direct ou indirect allant de la source de fii_1 vers la source de fie_1. Si un tel flux est trouvé alors il existe un flux indirect fii_1. Dans le cas contraire, la fonction cherche un chemin allant de source de fii_1 vers sa destination au moyen de $chercheUnFluxIndirectGraphe$.

La complexité de la fonction $chercheUnFluxIndirect$ est égale à deux fois la complexité de $chercheUnFluxIndirectGraph$. Elle est donc en $O(((|\mathcal{CSS}| \times (|\mathcal{CS}|) \times 2) + |\mathcal{CS}|) \times 2)$. En pratique, selon la propriété visée, une seule des deux recherches est effectuée. Il est rare que

les deux recherches soient nécessaires. De plus, il est possible de faire des optimisations pour certaines propriétés.

Algorithm 2 Algorithme de détection de flux d'information indirect fii_1 dans une interaction estampillée ite_1 et d'un graphe $GFIA$: $chercheUnFluxIndirect(fii_1, ite_1, GFIA)$

Pré-conditions : $ite_1 \in \mathcal{ITE}$, $GFIA$, $fii_1 \in \mathcal{FII}$
  si $op(ite_1) \in \mathcal{REO}$ **ou** $op(ite_1) \in \mathcal{WEO}$ **alors**
    $fie_1 \Leftarrow ite_1$
    si $dest(fie_1) = src(fii_1)$ **alors**
      si $chercheFluxIndirectGraph(src(fii_1), src(fie_1), GFIA, VRAI)$ **alors**
        return VRAI
      fin si
    fin si
  fin si
  si $chercheFluxIndirectGraph(src(fii_1), tgt(fii_1), GFIA, FAUX)$ **alors**
    return VRAI
  sinon
    return FAUX
  fin si

Par exemple, pour le GFIA de la figure 5.3 et l'interaction estampillée courante $ite_1 = user\_d \overset{process:transition}{\underset{[3620,3628]}{\rightharpoonup}} sysadm\_d$, considérons la recherche du flux d'information $fi_1 = sshd\_d \blacktriangleright sysadm\_d$. La destination de l'interaction courante est la même que celle du flux recherché. Il reste donc à détecter s'il existe un flux, direct ou indirect, entre la source du flux recherché et la source du flux associée à ite_1, c'est-à-dire, entre $sshd\_d$ et $user\_d$. Or, un tel arc existe, $chercheUnFluxIndirecte$ retourne VRAI.

Pour les transitions indirectes, la fonction $chercheUneTransIndirect(trans_1, ite_1, GFIA)$ permet de chercher une transition indirecte $trans_1$ dans une interaction courante ite_1 et un graphe $GFIA$. Son fonctionnement est similaire à $chercheUnFluxIndirect$. La complexité de $chercheUneTransIndirecte$ est en $O((|\mathcal{CSS}| \times (|\mathcal{CSS}| - 1) \times 2) + |\mathcal{CSS}|)$.

5.4.3 Cherche Tous les Flux d'Information Indirects

La fonction $chercheTousFluxIndirectGraph$ (algorithme 3) retourne tous les flux indirects présents dans GFIA correspondant au flux recherché. Comme nous le verrons par la suite, cette fonction est nécessaire pour permettre la vérification de conditions temporelles lorsque nous définissons des propriétés reposant sur plusieurs flux et des conditions temporelles entre ces flux.

La première différence avec la fonction $chercheUnFluxIndirectGraph$ repose sur l'utilisation d'un tableau $tabChemins$ énumérant les chemins qui permettent d'atteindre la cible. La fonction fournit en résultat pour chaque arc (par exemple, #1 sur la figure 5.5) partant de la source les arcs (#4,#5) qui terminent les chemins. Ainsi, non seulement nous avons une énumération des chemins (#1→#4, #1→#5) mais pour chacun d'eux, nous pouvons connaître leurs dates de début et de fin. Les dates de début correspondent aux dates de début du premier arc composant le chemin ($deb(\#1)$). Les dates de fin correspondent aux dates de fin du dernier arc composant le chemin ($deb(\#4)$, $deb(\#5)$).

La seconde différence est que la fonction évalue systèmatiquement les arcs (même si ils ont déjà été évalués). C'est essentiel puisque un même arc peut prendre part à plusieurs chemins et donc avoir plusieurs arcs de fin différents. Cela se traduit pour chaque élément du tableau par une liste de ces arcs de fin. Par exemple, prenons le GFIA de la figure 5.5. Il existe deux flux

indirects ($tab(\#1) = \{\#4, \#5\}$) #1→#2→#4 et #1→#3→#5 qui passent tous les deux par #1. Si $chercheTousFluxIndirectGraph$ ne prenait pas en compte la possibilité d'évaluer un arc dans plusieurs flux indirects, un des deux flux ne serait pas trouvé. Cela est d'autant plus important que les deux flux indirects trouvés n'ont pas la même date de fin (1010 et 1150). Cet aspect est essentiel, par la suite, pour mettre en oeuvre correctement les propriétés de sécurité en prenant bien en compte tous les flux avec leurs dates respectives.

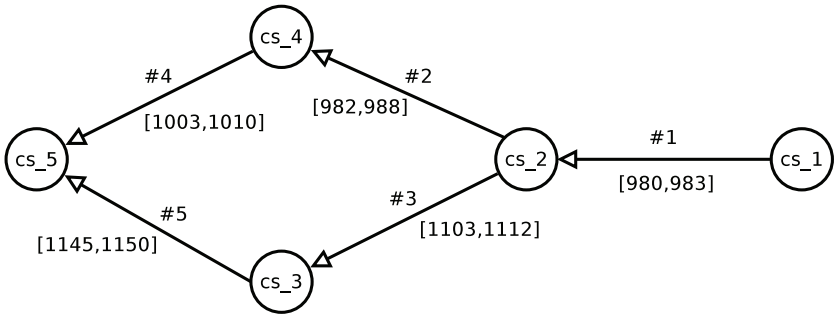

FIGURE 5.5 – Recherche de tous les flux dans le Graphe de Flux d'Information Augmenté

$chercheTousFluxIndirectGraph(cs_1, cs_2, GFIA, direct)$ a une complexité similaire a celle de $chercheUnFluxIndirectGraph$.

$chercheTousFluxIndirect(fi_1, ite_1, GFIA)$ est similaire à $chercheUnFluxIndirect$ mais retourne tous les flux indirects estampillés représentant le flux indirect fii_1. Sa complexité est donc similaire.

Pour les transitions indirectes, la fonction $chercheToutesTransIndirect(trans_1, ite_1, GFIA)$ permet de chercher toutes les transitions indirectes correspondant à $trans_1$ pour l'interaction courante ite_1 et le graphe $GFIA$. Son fonctionnement est similaire à $chercheTousFluxIndirect$.

5.4.4 Flux Général

Pour les flux généraux qu'ils s'agissent des flux d'information ou des transitions, les opérateurs sont implantés en appellant d'abord l'opérateur direct et éventuellement l'opérateur indirect. Par exemple, $chercheUnFluxGeneral$ appelle $chercheFluxDirect$ et si ce fonction ne retourne pas de flux, $chercheUnFluxGeneral$ appelle $chercheUnFluxIndirect$. Les complexités des opérateurs généraux sont donc celles des opérateurs indirects.

5.5 Compilation du langage

Dans cette section, nous présentons comment avec l'interaction estampillée courante, le graphe de flux d'information augmenté (voir la section 5.3) et la mise en oeuvre des opérateurs (voir la section 5.4), nous sommes capable de fournir une méthode générique pour produire l'algorithme qui permet de garantir les propriétés de sécurité qui peuvent être exprimées dans notre langage.

5.5.1 Méthode générique pour la compilation de notre langage

Nous avons vu que la formalisation de nos propriétés se fait au moyen des opérateurs que nous avons définis. Chaque opérateur du langage est implanté par une fonction, comme synthétisé dans le tableau 5.2. Par conséquent, la modélisation de la propriété peut être traduite directement en un algorithme. En effet, les opérateurs sont traduits en appel de fonction. Lorsqu'une propriété

Algorithm 3 Algorithme de recherche des chemins représentant un flux d'information indirecte allant de cs_1 vers cs_2 dans un graphe $GFIA$

Pré-conditions : $GFIA, cs_1 \in \mathcal{CS}, cs_2 \in \mathcal{CS}, fi_1 \in FIDSE, tailleUn$
$listCausale \Leftarrow listArcFin(cs_2)$
$ArcDejaVu \Leftarrow listArcFin(cs_2)$
$arc_1 \Leftarrow$ Dépile $queue$
$taille \Leftarrow 1$
$arcTailleChemin \Leftarrow$ Dernier arc dans $listArcFin(cs_2)$
$tableauOrigine \Leftarrow$ Tableau de 2 colonnes et $|\mathcal{CS}|$ lignes où chaque cellule contient une liste de couple d'arcs et dates et est initialisée à NULLE
tant que arc_1 n'est pas NULLE **faire**
  $tmpListArc \Leftarrow listArcFin(src(arc_1))$
  pour tout $arc_{tmp} \in tmpListArc$ **faire**
    si $(deb(arc_{tmp}) \leq fin(arc_1))$**et**$(arc_{tmp} n'est pas incluedans ArcDejaVu)$ **alors**
      Ajoute arc_{tmp} dans $queue$
      Ajoute arc_{tmp} dans $ArcDejaVu$
    fin si
    {**/*** Evaluation de l'arc même si il a déjà été vu. ***/**}
    si $deb(arc_{tmp}) \leq fin(arc_1)$ **alors**
      si $tableauOrigine[arc_1] ==$ NULLE **alors**
        $tableauOrigine[arc_{tmp}] = (arc_1, deb(arc_1))$
      sinon si $tableauOrigine[arc_1]$ n'est pas dans $tableauOrigine[arc_{tmp}]$ **alors**
        Ajoute $tableauOrigine[arc_1]$ à $tableauOrigine[arc_{tmp}]$
      fin si
    fin si
  fin pour
  si $arc_1 = arcTailleChemin$ **alors**
    $taille = 2$
  fin si
  $arc_1 \Leftarrow$ Dépile $queue$
fin tant que
{**/*** Appel de la fonction extraireFluxIndirect qui, à partir de la source cs_1, extrait pour chaque arc sortant de cs_1 l'élément du tableau ***/**}
return $extraireFluxIndirect(cs_1, tabChemins)$

utilise un opérateur sans les dates, cela signifie que le but est de tester l'existence d'un flux. Pour cela, nous disposons de la fonction $chercheUn$ qui vérifie l'existance d'un flux dans le graphe. Par contre, lorsqu'une propriété utilise à la fois l'opérateur et les dates, cela signifie qu'il faut extraire tous les flux et les dates correspondantes. La fonction $chercheTous$ permet de répondre à ce besoin. Ainsi, nous voyons que nous pouvons traduire automatiquement les différents usages des opérateurs en appel de fonction.

Par exemple, pour la définition de l'intégrité des données dans la section 4.3.1, le flux d'information est utilisé pour détecter la présence du flux : $css \overset{T}{\lll} cso$. Il n'y a pas besoin de vérifier une quelconque relation temporelle, la fonction $chercheUn$ est donc suffisante. Pour l'exécution indirecte qui utilise à la fois les opérateurs de transition indirecte et d'exécution directe ainsi qu'une relation temporelle entre ces flux, cette propriété fait appel à la fonction $chercheTous$ qui permettra la vérification de la relation temporelle pour tous les flux. Nous voyons, par cet exemple, qu'il est possible de traduire automatiquement cette propriété en un algorithme.

En suivant cette méthode, nous pouvons passer directement de la modélisation d'une propriété à un algorithme. Nous sommes donc capable de fournir un compilateur qui traduit automatiquement notre formalisme en algorithme exécutable sur des machines.

De plus, grâce au tableau 5.2 qui donne la complexité de chaque opérateur, nous sommes en mesure de calculer automatiquement la complexité d'une propriété. Par exemple, pour l'exécution indirecte, la complexité est la somme des complexités des fonctions $chercheToutesTransGenerales$ et $chercheDirecteExec$ ainsi que celle de la vérification des conditions temporelles. Dans ce cas, la complexité de la condition temporelle est $1 \times |\mathcal{CSS}| \times |\mathcal{CSS}|$. En effet, la fonction $chercheToutesTransGenerales$ retourne, au maximum, $|\mathcal{CSS}| \times |\mathcal{CSS}|$ chemins possibles car il existe au plus $|\mathcal{CSS}| \times |\mathcal{CSS}|$ chemins commençant par un noeud et finissant par un autre. Par ailleurs, la fonction $chercheExecDirecte$ retourne un seul chemin. La complexité de la propriété de sécurité d'exécution indirecte est donc en $O(1 + (((|\mathcal{CSS}| \times (|\mathcal{CSS}| - 1) \times 2) + |\mathcal{CS}|) \times 2) + ((|\mathcal{CSS}|) \times (|\mathcal{CSS}|)))$.

5.5.2 Exemples de Propriétés de Sécurité

Nous présentons maintenant l'usage effectif de cette méthode générique pour le cas de deux propriétés : l'intégrité des données et la situation de concurrence (Race Condition).

Intégrité des données

La propriété d'intégrité des données – telle que présentée dans la section 4.3.1 – est présentée comme la protection contre les flux d'information généraux.

En utilisant la fonction $chercheUnFluxGeneral$ précédemment décrite, nous proposons l'algorithme 4 pour l'application de la propriété de sécurité exprimée dans la définition 4.3.4. Si un

Algorithm 4 Algorithme d'application de la propriété d'intégrité générale des objets de css envers cso : $intObjet(css, cso, ite_1, GFIA)$

Pré-conditions : $css, cso, ite_1, GFIA$,
$fig_1 = (css, cso)$
return NOT($chercheUnFluxGeneral(fig_1, ite_1, GFIA)$)

flux est détecté dans GFIA $\cup\, ite_1$, l'interaction doit être refusée, l'algorithme retourne alors FAUX. Dans le cas contraire, ite_1 peut être autorisée, l'algorithme retourne VRAI.

L'application de la propriété d'intégrité générale des objets a une complexité égale à celle de la fonction de recherche d'un flux d'information général dans un GFIA soit $O((((|\mathcal{CSS}| \times (|\mathcal{CS}|) \times$

$2) + |\mathcal{CS}|) \times 2)$. Elle est donc polynomiale avec le nombre d'entités sujets et objets présents sur le système.

Race Condition

L'algorithme 5 permet d'implanter la race condition pour une entité lcs contre une seconde entité mcs. Cet algorithme utilise la fonction $chercheTousFluxGenerals$ car il existe une relation temporelle à vérifier entre les trois flux. La propriété considérée est une variante de notre définition générale de la situation de concurrence avec la limitation suivante : le troisième flux cso vers lcs doit être un flux direct, ce qui est le cas dans nombre de situations pratiques. De plus, cette propriété va déjà plus loin que les solutions classiques qui ne prennent pas en compte les flux indirects. En effet, notre propriété prend en compte les flux indirects dans deux cas sur trois. Grâce à simplification, la propriété conserve une complexité acceptable correspondant à deux recherches de chemins ainsi que la condition temporelle. Sa complexité est en $O((((|\mathcal{CSS}| \times (|\mathcal{CS}|) \times 2) + |\mathcal{CS}|) \times 2) \times 2 + |\mathcal{CS}|^4)$.

Algorithm 5 Algorithme optimisé d'application de la propriété d'absence de concurrences d'accès lcs par mcs : $rapideNoConcurrenceAcces(lcs, mcs, ite_1, GFIA)$

Pré-conditions : $ite_1, GFIA, lcs, mcs$
  si $op(ite_1) \in \mathcal{WEO}$ **ou** $op(ite_1) \in \mathcal{REO}$ **alors**
    $fig_1 = lcs \lll dest(ite_1)$
    $fig_2 = mcs \lll dest(ite_1)$
    $liste\_flux_1 = chercheTousFluxGeneraux(fig_1, GFIA)$
    $liste\_flux_2 = chercheTousFluxGeneraux(fig_2, GFIA)$
    si $liste\_flux_1 \neq$ NULLE **et** $liste\_flux_2 \neq$ NULLE **alors**
      pour tout $fi_1 \in liste\_flux_1$ **faire**
        pour tout $fi_2 \in liste\_flux_2$ **faire**
          si $deb(fi_1) \leq fin(fi_2)$ **alors**
            return FAUX
          fin si
        fin pour
      fin pour
    fin si
  fin si
  return VRAI

5.6 Conclusion

Dans ce chapitre, nous avons proposé le graphe de flux d'information augmenté (GFIA) qui contient tous les flux d'information nécessaires à notre langage. Nous avons montré que sa complexité est polynomiale avec le nombre de contextes. En pratique, le nombre de contextes est relativement faible ce qui permet d'implanter le graphe en mémoire.

Nous avons proposé une mise en oeuvre de chacun de nos opérateurs sous forme de fonctions qui réalisent des opérations de recherche de flux dans ce graphe. Nous avons établi la complexité de chacune de ces fonctions qui reste polynomiale avec le nombre de contextes. Nous voyons, là aussi, que ces fonctions présentent une complexité tout à fait acceptable qui permet de faire fonctionner les algorithmes en pratique sur une machine.

Ces différentes fonctions nous permettent de proposer une méthode de compilation de notre langage. De plus, nous pouvons fournir la complexité des algorithmes ainsi générés. Nous montrons que cette complexité reste polynomiale avec le nombre de noeuds. Ceci démontre que notre langage reste compilable et exécutable. Nous présentons des exemples d'algorithmes ainsi générés pour deux propriétés de sécurité. Le fait que ce langage soit compilable permet aisément le support de nouvelles propriétés.

Grâce aux différentes propriétés de sécurité exprimables par notre langage, nous pouvons définir une politique de sécurité complète qui peut être compilée avec la méthode proposée. Le code généré par cette compilation peut être utilisé de différentes façons. Il peut servir à analyser, vérifier des comportements, détecter des activités malicieuses et surtout protéger un système d'exploitation. La protection revient à permettre au système d'exploitation de garantir nos propriétés de sécurité. C'est cet aspect qui est détaillé dans le chapitre suivant.

Chapitre 6

Implantation et Expérimentation

6.1 Introduction

Au chapitre précédent, nous avons décrit comment construire un graphe de flux qui permet d'analyser les activités systèmes et de garantir des propriétés de sécurité. Dans ce chapitre, nous allons décrire comment nous avons intégré cette approche pour la protection d'un système d'exploitation. Cette intégration repose sur deux composants. Le composant PIGA-DYN-Protect s'exécute en espace noyau et génère les interactions estampillées. Le composant PIGA-DYN s'exécute en espace utilisateur et utilise ces interactions pour maintenir le graphe et prendre les décisions d'autorisation.

Nous montrons au travers de ce chapitre que le traitement des canaux cachés est complet d'un point de vue système. Les canaux cachés résiduels reposent sur des failles logicielles internes au noyau ou matérielles et sortent du cadre de cette étude. Bien que nous ne détaillons pas les aspects performances, les mesures que nous avons effectuées, montrent un surcoût très faible au regard des propriétés fournies.

Valider l'efficacité d'une méthode de protection est quelque chose de difficile. L'approche que nous avons retenu est de réaliser une expérimentation réelle et à large échelle. Afin de traiter de réelles vulnérabilités, nous avons mis en place deux types de pots-de-miel (en anglais, HoneyPot) à haute interaction. Les pots-de-miel à haute interaction sont des machines réelles, c'est-à-dire constituées d'un système d'exploitation complet et d'applications qui utilisent les services du système. Le premier type de pot-de-miel vise à tester l'efficacité de notre approche pour la navigation sur le Web. Il s'agit d'un pot-de-miel client (HoneyPot Client) qui parcourt un ensemble de ressources pour s'exposer au plus grand nombre d'exploits possibles. Le second type de pot-de-miel est un HoneyPot serveur fournissant des vulnérabilités connues publiquement. Le nombre de tentatives de violation des propriétés est suffisament élevé pour valider notre solution. Nous verrons que les propriétés fournies permettent réellement de protéger les systèmes contre ces scénarios d'attaques.

6.2 Implantation

6.2.1 Architecture Globale

La figure 6.1 décrit l'architecture logicielle de notre méthode de protection. Cette architecture présente un ensemble de composants. Nous choisissons d'illuster ces composants par la séquence d'exécution associée à un appel de fonction par une application. Dans l'exemple choisi, l'application s'exécute au sein d'un processus système qui appelle une fonction `fread` afin de lire un fichier. Cette fonction est fournie par une librairie en l'occurence la librairie C (`libc`). Cette li-

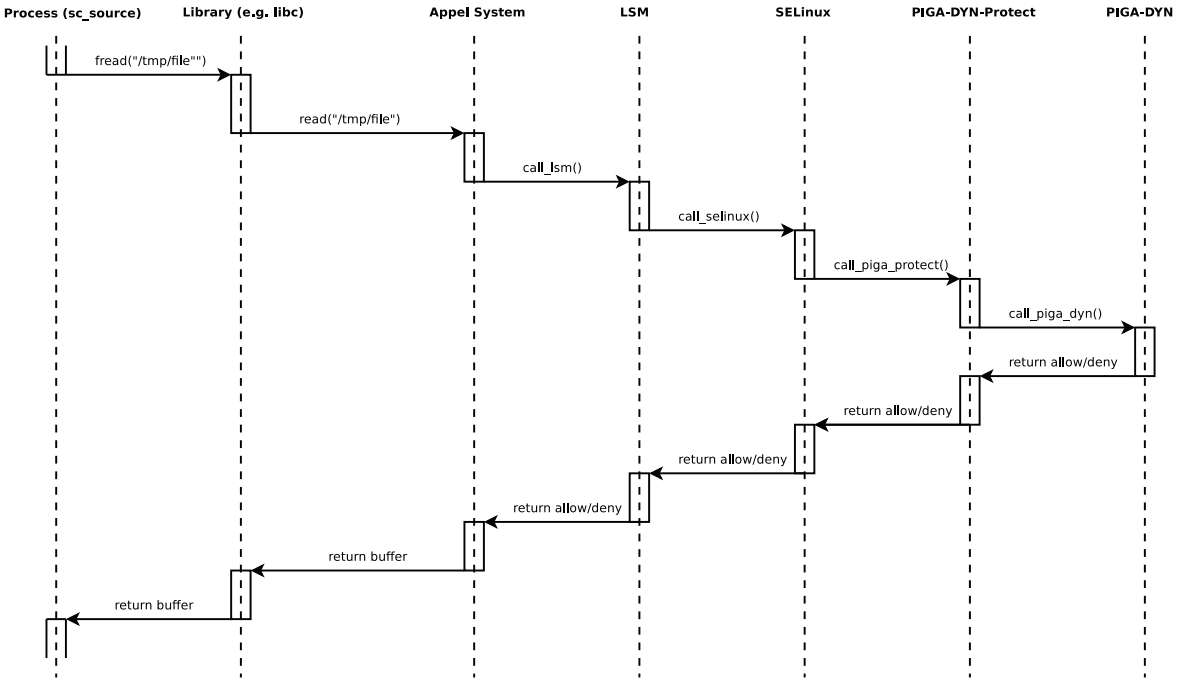

FIGURE 6.1 – Architecture globale de notre système de protection

brairie utilise l'appel système `read` pour communiquer avec le noyau du système d'exploitation. Cet appel système permet de réaliser l'exécution de la lecture en mode noyau. Ce mode est le seul à pouvoir réaliser les entrées/sorties afin de transférer les données entre l'espace utilisateur et les ressources physiques (disque, mémoire, réseau, etc). Avant de réaliser l'opération d'entrée/sortie, le noyau vérifie les protections discrétionnaires classiques d'Unix à savoir les droits des processus du type lecture, écriture, exécution. Si l'opération est permise par le système DAC, celui-ci invoque LSM. Les versions actuelles de GNU/Linux utilisent Linux Security Module (LSM) pour introduire de nouveaux mécanismes de protection. Le principe de LSM est d'introduire des modules qui permettent de détourner les appels systèmes. LSM est utilisé pour appeller le module SELinux. Ce module applique le Type Enforcement, c'est-à-dire, il affecte un contexte à chaque entité du système. PIGA-DYN-Protect est appellé par SELinux et récupére toutes les informations nécessaires pour générer une trace de l'appel système. Cette trace est écrite dans l'espace utilisateur de PIGA-DYN. Ce dernier utilise la trace reçue pour calculer sur le graphe existant si celle-ci viole une des propriétés de sécurité requises. Dans le cas contraire, la trace est utilisée pour mettre à jour le graphe. La décision est renvoyée dans l'espace noyau à PIGA-DYN-Protect. Ce dernier la transmet à SELinux puis à LSM qui prend la décision finale d'autoriser (ou non) l'appel système. Cette décision provoque l'exécution de l'appel système et lorsque celui-ci termine, le noyau rend la main à la librarie C en lui renvoyant les résultats de l'appel. Finalement, la librarie C rend la main au processus qui a réalisé l'opération d'entrée/sortie.

6.2.2 Complétude

Grâce à notre architecture, nous sommes capable de contrôler chaque appel système en lui associant une trace qui est utilisée pour vérifier les propriétés de sécurité requises. Nous allons expliquer dans ce paragraphe pourquoi la capture des appels systèmes nous permet de contrôler complètement les interactions entre les différents contextes.

Etant donné qu'un flux entre un processus et un objet passe forcément par un appel système,

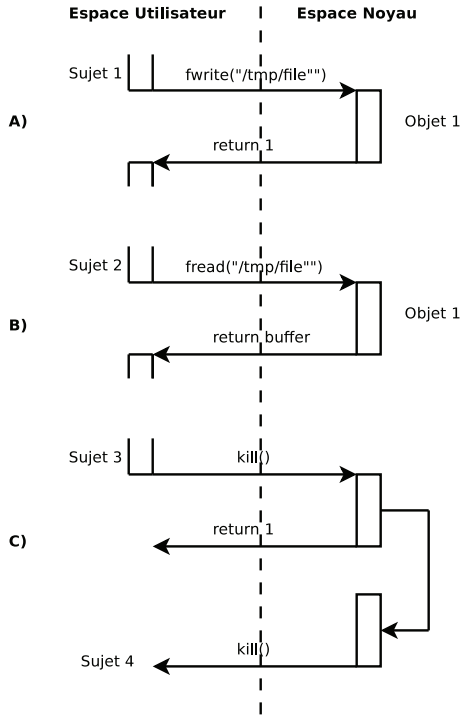

FIGURE 6.2 – Complétude des détournements d'appel système

nous sommes donc en mesure de contrôler tous les flux entre un sujet et un objet. La figure 6.2.A illustre le sujet 1 qui fait un appel système de type écriture sur l'objet 1.

A l'exception des signaux, il n'existe pas de flux direct entre deux sujets. En effet, les flux entre sujets passent par un objet intermédiaire. Il y a donc deux appels systèmes nécessaires pour réaliser un flux entre ces sujets. Dans la figure 6.2.A et 6.2.B, à la suite de l'appel système (de type écriture) réalisé par le sujet 1, le sujet 2 réalise un appel système de type lecture sur l'objet 1, ce qui permet de réaliser le transfert d'information entre les deux sujets.

Concernant les signaux entre sujets, ceux-ci passent aussi par un appel système et nous sommes donc en mesure de les contrôler. Par exemple, dans la figure 6.2.C présente le sujet 3 qui envoie un signal au sujet 4 grâce à l'appel système `kill`. Comme nous capturons cet appel système avant que le noyau envoie le signal au destinataire, nous sommes en mesure d'autoriser ou d'interdire l'envoi de ce signal.

Il existe une classe particulière de signaux émis par le noyau sans appel système. C'est le cas, par exemple, d'un processus qui ferait une erreur d'accès à la mémoire. Le noyau décide alors de lui envoyer un signal du type `segmentation fault`. Nous n'interceptons pas ces signaux. En effet, ceux-ci servent à la sûreté de fonctionnement et proviennent d'une source sûre qui est le noyau. Il n'est donc pas nécessaire de les contrôler.

Enfin, il reste le cas des canaux de contrôle. Ces canaux correspondent à des canaux cachés dans le système, par exemple, les métadonnées d'un fichier. Nous traitons correctement les canaux de contrôle. En effet, ces canaux de contrôle sont associés à des fonctions du noyau (ex. `getattr`) que nous détournons de la même façon que les autres appels systèmes.

Nous voyons que nous sommes donc en mesure d'instrumenter complétement le système. En effet, nous capturons bien l'ensemble des interactions aussi bien entre un sujet et un objet qu'entre deux sujets. Il reste cependant des canaux cachés qui peuvent être soit dans le noyau lui-même, soit dans le matériel. Cependant, ces canaux cachés sortent du contexte de cette étude. Il s'agirait

de vérifier la sûreté d'un noyau système ou la sûreté d'un matériel.

6.2.3 SELinux

Le module SELinux permet d'appliquer le principe du Type Enforcement (TE) à GNU/Linux. Ainsi, chaque processus se trouve associé à un contexte de sécurité sujet. Chaque appel système concerne un objet. Etant donné que chaque objet est associé statiquement à un contexte de sécurité, SELinux est capable de définir l'interaction pour le contexte sujet demandant une permission (associée à l'appel système) sur le contexte objet. Ainsi, SELinux nous fournit directement les informations nécessaires à la trace. Le listing 6.1 donne un exemple d'une telle trace. Elle représente l'accès en écriture par le sujet $mozilla\_t$ à un fichier ayant le contexte tmp_t.

Listing 6.1 – "Trace d'audit SELinux"

```
Oct 06 19:42:45 AR1DEV kernel: audit(1107747765.871:7550947): avc: denied { write } for pid=2479 exe=/
    usr/bin/firefox dev=sda2 ino=921135 scontext=user_u:mozilla_r:mozilla_t tcontext=system_u:
    object_r:tmp_t tclass=file
```

Dans notre implantation, nous n'utilisons pas la protection mandataire fournie par SELinux. En effet, notre but est de contrôler dynamiquement les appels systèmes pour respecter les propriétés de sécurité requises. Donc nous n'avons pas besoin d'un contrôle préalable par SELinux. En pratique, nous avons remplacé la procédure de prise de décision de SELinux par notre propre mécanisme.

En revanche, SELinux nous sert à labéliser l'ensemble des entités du système aussi bien les objets que les sujets. Cette labélisation est statique. Lorsque nous avons besoin d'une labélisation dynamique, celle-ci est réalisée par notre solution.

6.2.4 PIGA-DYN-Protect

Le composant PIGA-DYN-Protect fait parti de notre solution. C'est le module noyau qui remplace la fonction de prise de décision de SELinux. En fait, cette fonction réalise une interface entre la partie noyau et un processus en espace utilisateur qui contrôle la non-violation des propriétés de sécurité. Cette fonction d'interface récupère la décision prise par le processus utilisateur et la traduit en autorisation ou refus de l'appel système. Le rôle de PIGA-DYN-Protect est notamment d'associer les dates à la trace de l'appel système afin de calculer une interaction estampilllée.

La figure 6.3 présente le fonctionnement de PIGA-DYN-Protect. Le schéma présente les interactions entre le module noyau PIGA-DYN-Protect et le processus utilisateur qui effectue un appel système. Il présente aussi les interactions entre PIGA-DYN-Protect et le processus utilisateur PIGA-DYN chargé de calculer la décision d'autorisation. L'application réalise une opération d'écriture qui se traduit par une activité noyau chargée de la réalisation de l'appel système. L'activité appelle la fonction d'interface PIGA-DYN-Protect. PIGA-DYN-Protect récupère les informations relatives à l'interaction (sujet, objet, permission) et l'estampille. PIGA-DYN-Protect écrit l'interaction estampillée dans une liste de messages `requête` interne au noyau. Ensuite, PIGA-DYN-Protect se met en attente de lecture sur une seconde liste de messages `réponse`. Lorsque PIGA-DYN-Protect lit la réponse correspondant à la requête dans la liste `réponse`, elle rend la main à l'activité qui peut ainsi retourner la décision d'autoriser ou d'interdire l'appel système. En pratique, la gestion de la liste de message `réponse` est réalisée afin que la lecture retourne la réponse associée à la requête. Pour cela, la réponse transporte un numéro associé à la requête.

6.2.5 PIGA-DYN

PIGA-DYN est le composant qui récupère les interactions estampillées, maintient le graphe tel que présenté dans le chapitre 5 et calcule les violations potentielles des propriétés de sécu-

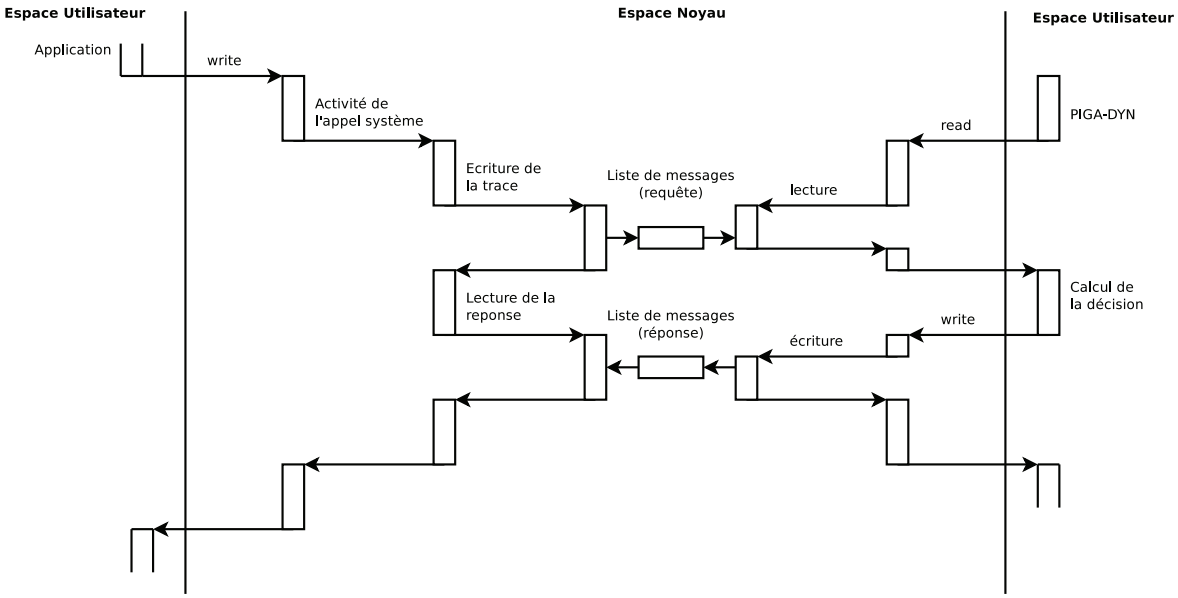

FIGURE 6.3 – Fonctionnement du module noyau PIGA-DYN-Protect

rité requises. Plus précisément, PIGA-DYN a, en entrée, les interactions estampillées ainsi que les propriétés requises. Lorsqu'il reçoit une nouvelle interaction estampillée, il vérifie l'absence de violations et met à jour le graphe si besoin. En sortie, PIGA-DYN fournit la décision pour l'interaction estampillée.

Comme présenté dans la figure 6.1, le processus utilisateur PIGA-DYN réalise un appel système de lecture afin d'obtenir une interaction estampillée correspondant à une requête issue de PIGA-DYN-Protect. Cet appel système de lecture génère une activité noyau chargée du traitement. Cette activité appelle une fonction de lecture de la liste `requête`. Cette fonction de lecture est bloquante, elle rend la main lorsqu'une requête est disponible dans la liste. L'activité de l'appel système de lecture écrit la requête dans les paramêtres de retour de la lecture et réveille PIGA-DYN. Ce processus récupère ainsi une requête contenant une interaction estampillée. PIGA-DYN calcule la décision selon les principes algorithmiques que nous avons décrit dans le chapitre 5. Il utilise un appel système d'écriture pour transférer cette décision à la partie noyau. L'appel système d'écriture génère une nouvelle activité noyau qui appelle une fonction d'écriture sur la liste de messages `réponse`.

6.3 Expérimentation

6.3.1 HoneyPot Client

Introduction

Un pot de miel client [Provos et al., 2007] (HoneyPot Client) est une architecture permettant de parcourir un ensemble de ressources de manière active afin de détecter des malwares s'installant lors de la consultation d'une ressource. En général, les HoneyPots Clients sont dédiés au parcours de pages webs afin de détecter des attaques du coté client. Dans le cadre des pages webs, ces attaques sont le plus souvent du type drive-by download.

La différence majeure avec un pot de miel classique (HoneyPot Serveur) est que le HoneyPot Client n'attend pas passivement les attaquants mais va activement accéder aux ressources pour

tenter de déclencher une attaque. De plus, contrairement aux HoneyPots Serveur, toutes les ressources parcourues ne sont pas forcément malicieuses. Il faut donc que le HoneyPot Client soit capable de faire la différence entre une ressource malicieuse et une ressource légitime.

Tout comme pour les HoneyPots Serveur, il existe deux types de HoneyPots Client : les basse interaction et les haute interaction. Les basses interactions se limitent à émuler un client et récupérer la reponse envoyée par la ressource interrogée. Cette réponse est ensuite analysée pour déterminer si elle est légitime ou malicieuse. Les HoneyPots Client basse interaction ont deux inconvénients. Tout d'abord, ils se limitent à une émulation qui est souvent partielle et qui résulte par une non-détection de certaines attaques dûe à l'absence de certains composants. De plus, ils sont connus pour générer plus de faux positifs que les solutions hautes interactions car ils ne réagissent pas comme un client normal et sont limités par la qualité de l'émulation.

Les HoneyPots Client haute interaction sont basés sur un système d'exploitation complet où est installé le client. Le client est contrôlé, de manière plus ou moins automatique, pour accéder aux ressources. Différents outils vérifient l'état du système pour voir si des modifications malicieuses ont eu lieu. L'intérêt de cette approche est qu'elle ne se limite pas à analyser des réponses, bien souvent pour la classification mais permet également de détecter des attaques inconnues en regardant leur effet sur le système. Par contre, elle nécessite plus de puissance que l'approche basse interaction, car elle demande de faire tourner un système complet. De plus, elle requiert une bonne compréhension du système afin de l'instrumenter correctement (pour l'analyse) mais aussi de protéger le reste de l'environement contre le client qui pourrait potentiellement être détourné à des fins malicieuses.

Plateforme

Pour pouvoir tester notre solution de protection PIGA-DYN contre des attaques clients, nous souhaitions utiliser un HoneyPot Client Haute Interaction pour GNU/Linux. A notre connaissance, aucune solution de ce type existe. Par conséquent, nous avons implanté notre propre solution décrite dans la figure 6.4.

Le HoneyPot Client fonctionne en se basant sur des tâches qui sont composées d'un site web source et d'une profondeur de recherche. La profondeur correspond au suivi récursif des liens hypertextes. Notre HoneyPot Client est construit à base de composants Python interconnectés par des interfaces XML-RPC facilitant leur répartition et le passage à l'échelle. Le HoneyPot Client est administrable par une interface Web. L'interface Web utilise ensuite les composants Controler et DB pour consulter les tâches en cours ou fini et pour en ajouter des nouvelles. L'interface Web permet également de consulter les violations pour chaque URL. Deux composants centraux (DB et Controler) sont en charge de l'orchestration de l'ensemble de l'infrastructure. Le composant Spider permet à partir d'un site source et d'une profondeur de retourner une liste d'URL. Les Agents tournent sur des systèmes d'exploitation GNU/Linux avec Mozilla Firefox et notre solution de protection. Ils permettent d'afficher automatiquement chaque URL trouvé par les Spiders. Les Agents retournent ensuite la liste des violations de propriétés de sécurité qui ont eu lieu pour chaque URL.

Résultats

Le tableau 6.1 présente les résultats d'expérimentations de notre pot de miel client. Sur ce type de pot de miel, il est impossible d'utiliser notre solution sous la forme de protection. En effet, dans le cas d'une protection, aucune attaque ne peut avoir lieu et il n'y a donc pas de possibilité d'analyse des scnéarios. C'est pourquoi les résultats fournis par PIGA-DYN correspondent à une analyse d'activités malicieuses. Au total, il s'agit de 588602 pages Web qui ont ainsi été visitées par notre pot de miel à partir de 39 URL sources.

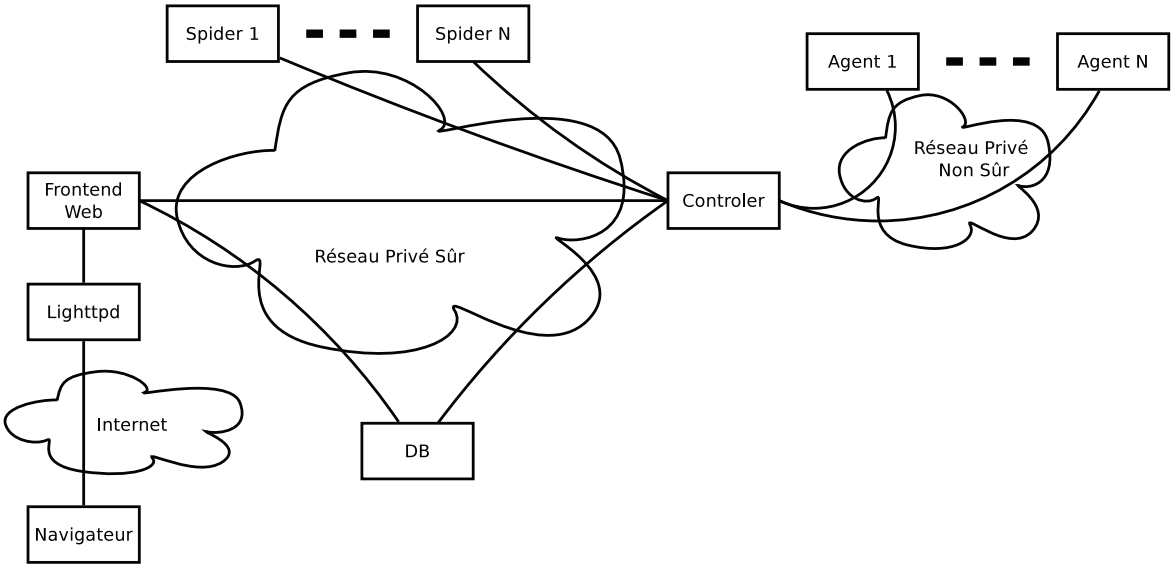

FIGURE 6.4 – Architecture de notre HoneyPot Client Haute Interaction sous GNU/Linux et Mozilla Firefox

La première propriété qui est présentée correspond à l'intégrité des données des mots de passe ($dataint(shadow\_t)$). Afin de tester cette propriété, la machine a été configurée de façon permissive et elle autorise les écritures sur le fichier `/etc/shadow`. En utilisation normale, Firefox n'a aucune légitimité à pouvoir modifier le fichier de mot de passe. On voit qu'il existe 4 violations de l'intégrité du fichier de mots de passe. Ces quatres violations proviennent d'un site Flash. Le composant Flash hérite des droits de Firefox et donc de l'utilisateur. Cette propriété met donc en évidence l'intérêt de mettre le composant Flash dans un bac à sable. En pratique, ceci est possible. Pour cela, il faut faire transiter le processus Firefox lorsqu'il lit le composant Flash dans un domaine particulier, par exemple, $mozilla\_flash\_t$ et mettre en place une propriété $vchroot(mozilla\_flash\_t)$. Ainsi, notre système de protection pourrait prévenir ces attaques tout en autorisant l'exécution de Flash. On voit que, non seulement, cette propriété permet ainsi de classifier les composants devant être mis dans un bac à sable. Mais surtout qu'elle permet de déduire les propriétés à mettre en place pour confiner les attaques tout en permettant le fonctionnement normal de l'application.

La seconde propriété permet de garantir l'intégrité des binaires. De façon similaire à l'exemple précédent, la machine a été configurée de manière permissive autorisant la modification de binaires. De plus, le navigateur est configuré pour autoriser le maximum de scripts, notamment offrir le plus de permissions à Javascript. Il existe au total 68 violations de cette propriété. La majorité des attaques correspond à des sites Flash. Dans ce cas, le même principe que précédemment peut être appliqué pour protéger contre ces attaques. Une minorité d'attaque est basée sur l'utilisation de Javascript. Il existe deux solutions pour contrer ces attaques. La plus simple consiste à mettre en oeuvre un parefeu applicatif qui détecte le code Javascript embarqué et fait transiter le processus Firefox dans un contexte spécifique, par exemple, $mozilla\_javascript\_t$. Il est alors possible de mettre en place une propriété $vchroot(mozilla\_javascript\_t)$. L'inconvénient de cette méthode est qu'elle ne permet pas de prévenir contre des codes Javascript chiffrés, par exemple, dans une connexion SSL. La deuxième solution permet de traiter le cas des données chiffrées. On peut ainsi imaginer développer un plugin Firefox qui pourra récupérer l'information en clair en cas de chiffrement et donc faire transiter le processus Firefox dans le domaine $mozilla\_javascript\_t$. On

	Analyse
$dataint(mozilla\_t, shadow\_t)$	4
$dataint(mozilla\_t, bin\_t, usr\_bin\_t)$	68
$dataint(mozilla\_t, lib\_t)$	97
$dataconf(mozilla\_t, shadow\_t)$	12
$trans(mozilla\_t, *)$	25658
$vchroot(mozilla\_d)$	11248963
$tpe(mozilla\_t, bin\_t, usr\_bin\_t)$	178
$dataint(mozilla\_conf\_t)$	2941

TABLE 6.1 – HoneyPot Client GNU/Linux avec Mozilla Firefox

voit que si la méthode de détection des codes Javascript peut varier, PIGA-DYN peut, en tout cas, empêcher ces violations au moyen d'une propriété dédiée.

La troisième propriété permet de garantir l'intégrité des librairies. Firefox a été configuré pour accepter l'installation de tous les plugins ce qui permet d'installer et/ou de modifier des librairies. On trouve 97 violations de cette propriété. Ces violations peuvent être empêchées en mettant en place une propriété de séparation de priviléges pour Firefox ($SdP(mozilla\_t)$). Cette propriété laisserait Firefox télécharger les plugins mais empêcherait leurs exécutions. Il peut exister d'autres propriétés pour prévenir un tel comportement.

La propriété $dataconf(shadow\_t)$ permet d'analyser les violations de confidentialité des fichiers de mots de passe. Pour autoriser ces violations, nous avons donné les droits en lecture du fichier `/etc/shadow` à l'utilisateur. 11 violations correspondent à des sites Flash et une seule provient d'un plugin. Les mêmes méthodes de protection pourront être appliquées grâce aux propriétés $vchroot(mozilla\_flash\_t)$ et $SdP(mozilla\_t)$.

La propriété $trans(mozilla\_t, *)$ permet d'analyser les changements de contextes. La majorité des 25658 violations correspond à des accès de Firefox à l'espace utilisateur. Ceci explique le grand nombre de violations. Cela met en évidence la nécessité d'avoir un bac à sable dédié à Firefox afin que celui ne puisse pas interférer avec l'espace utilisateur.

La propriété $vchroot(mozilla\_d)$ permet d'isoler le domaine de Firefox du reste du système. Etant donné qu'il n'y a pas de protection, toute interaction entre le domaine Firefox et le système génère une violation. Cette propriété nous sert à l'analyse comportementale des interactions de Firefox avec le système. Elle permet d'avoir des données précises sur les actions de Firefox sur le système. En termes de protection, cette propriété permet d'isoler Firefox du reste du système. Nous avons vu que cette propriété est essentielle pour prévenir un certain nombre de violations aussi bien d'intégrité que de confidentialité.

La propriété $tpe(mozilla\_t, bin\_t, usr\_bin\_t)$ permet de limiter les exécutions issues de Firefox à des binaires de confiance. Les 178 violations constatées correspondent à des téléchargements puis exécutions de binaires. Une propriété de séparation de priviléges ($SdP(mozilla\_t)$) permettrait d'éviter ces exécutions. De plus, la propriété TPE appliquée en terme de protection combinée avec la seconde propriété ($dataint(bin\_t, usr\_bin\_t)$) offre une seconde méthode pour prévenir ces attaques.

La dernière propriété $dataint(mozilla\_conf\_t)$ permet de détecter les modifications du fichier de configuration de Firefox. L'essentielle des violations correspond à des plugins qui modifient la configuration de Firefox. Cette propriété activée en terme de protection empêche de tels comportements. En effet, en usage normal, c'est l'utilisateur qui doit pouvoir modifier la configuration et non pas Firefox. Cette propriété n'empêche pas l'utilisateur de modifier la configuration de Firefox.

	Protection	**Analyse**
$dataint(user\_t, shadow\_t)$	0	2580
$dataint(user\_t, bin\_t, usr\_bin\_t)$	0	6354
$dataint(user\_t, lib\_t)$	0	14369
$dataconf(user\_t, shadow\_t)$	0	854
$trans(user\_t, *)$	0	19325
$tpe(user\_t, bin\_t, usr\_bin\_t, tmp\_t)$	0	36843
$vchroot(user\_t)$	98562	2695457
$racecondition(user\_t, root\_t)$	3526	4129

TABLE 6.2 – SSH sans mot de passe et faille dans `login`

6.3.2 HoneyPot Serveur

Introduction

Les HoneyPot Serveur sont des machines (ou des émulations de machines) qui fournissent un service contenant volontairement une ou plusieurs vulnérabilités exploitables. Tout comme pour les HoneyPot Client, il existe deux types de HoneyPot Serveur. Les versions hautes et basses interactions ont les mêmes avantages et inconvénients que l'approche Client.

Plateforme

Pour tester notre solution contre les attaques coté service, nous avons mise en place une solution de type HoneyPot Serveur Haute Interaction. L'architecture est composée d'un ensemble de machines virtuelles fournissant chacune un service. Chaque service est déployé deux fois : une avec une protection strict et une pour l'analyse des scénarios d'attaque. Périodiquement, nous retournons à un état sain (en utilisant des fonctions de checkpointing) pour l'analyse de scénarios. Cela nous permet d'éviter que des attaquants utilisent nos machines pour lancer d'autres attaques. De plus, toujours dans le but d'éviter que notre HoneyPot soit une source d'attaque, nous avons installé un parefeu très strict limitant le nombre de packets en sortie.

Résultats

Nous présentons maintenant les résultats obtenus sur le HoneyPot Serveur. Trois types de services ont été évalués. Le premier service est une application de connexion à distance sécurisée (SSH). Le deuxième est constitué d'applications web écrites en PHP à savoir une application de forum (`phpBB`) et un service web pour lire le courrier éléctronique (`RoundCube`). Le troisième service est un serveur de fichiers (`Samba`) qui autorise le partage entre plusieurs systèmes d'exploitation (par exemple, Unix et MS-Windows). Pour chaque service, nous proposons deux modes de contrôle des activités. Le premier mode correspond à un service protégé au moyen de propriétés de sécurité. Le second ne présente pas de protection au moyen de ces propriétés mais permet à l'attaquant de développer des scénarios. Le but de la seconde approche est l'analyse de ces scénarios.

Service SSH Le service de connexion à distance SSH a été volontairement fragilisé en autorisant systématiquement la connexion d'un utilisateur sans vérifier son mot de passe. De plus, l'application `login` présente une situation de concurrence permettant une élévation de privilèges vers le compte administrateur ($root$).

Pour le mode protection (voir le tableau 6.2), une fois l'attaquant connecté sur la machine, il commence par télécharger un exploit pour la vulnérabilité `login`. Ce téléchargement est suivi d'une exécution de l'exploit et se traduit par une violation de la propriété tpe puisque l'exploit n'est pas dans les binaires de confiance. Il peut avoir d'autres téléchargements d'exploit et au total, cela correspond à 8567 tentatives de violations. L'attaquant essaie aussi diverses interactions en dehors de son espace utilisateur ce qui correspond à 98562 tentatives de violations de la propriété $vchroot$. Etant donné que l'attaquant a la possibilité d'installer l'exploit dans le répertoire `/tmp` qui est un répertoire de confiance (voir propriété tpe), il a donc la possibilité d'exécuter cet exploit qui est bloqué par la propriété $racecondition$. L'attaque n'aboutissant pas, on voit qu'il n'y a pas de tentatives de violations pour les autres propriétés. Dans le cas inverse, si l'attaquant avait réussi à passer administrateur ($root$), il aurait pu tenter d'autres compromissions mais se serait vu bloqué par les propriétés d'intégrité et de confidentialité que nous avons mises en place.

Pour le mode analyse (voir le tableau 6.2), les attaques sont similaires jusqu'à l'utilisation de l'exploit de situation de concurrence ($racecondition$) sur `login`. On voit donc un nombre important de violations des trois propriétés tpe, $vchroot$, $racecondition$. A l'inverse du mode protection, ces attaques aboutissent et permettent de développer des scénarios. On voit que ces scénarios permettent notamment de transiter dans différents domaines ($root$, $netstat$, etc), de violer la confidentialité des mots de passe, de violer l'intégrité des libraries, des binaires ainsi que des mots de passe. Pour toutes les violations de confidentialité et d'intégrité, on voit que les activités sont indirectes. En effet, ce n'est qu'en passant par le domaine $root\_t$ que l'utilisateur peut violer ces propriétés. Donc il y a forcément au moins une transition de $user\_t$ vers $root\_t$ avant de pouvoir provoquer les violations.

Services Web Nous avons délibérement installé des versions de `phpBB` et `RoundCube` qui présentaient des vulnérabilités de type *exécution d'un interpréteur de commandes à distance* (Remote Shell Execution). De plus, des exploits publics existent pour ces vulnérabilités et nous avons même communiqué au sein du forum pour permettre aux attaquants d'accéder à ces exploits.

Pour le mode protection (voir le tableau 6.3), la propriété $NoExec$ empêche l'exécution de l'interpréteur ($shell\_exec\_t$) par Apache ($apache\_t$). Donc l'attaquant voit les exploits échouer. Cette propriété protége même contre les activités indirectes qui partiraient d'Apache pour tenter d'exécuter ensuite un interpréteur. C'est pourquoi on voit qu'il n'y a aucune autre tentative de violation sur cette machine.

Pour l'analyse (voir le tableau 6.3), on voit que les attaques commencent aussi par l'exécution d'un interpréteur. Donc, nous voyons un grand nombre de violations de la propriété $NoExec$. Ensuite, des scénarios peuvent se développer puisqu'il n'y a pas de protection. Par la suite, on voit des scénarios d'attaque similaire à ceux de SSH. Ces scénarios conduisent à des violations des propriétés $vchroot$, tpe, $trans$, $dataconf$ et $dataint$.

Samba Nous avons installé une version de `Samba` contenant une vulnérabilité du type *exécution d'interpréteur de commandes à distance* (Remote Shell Execution). Cette vulnérabilité est utilisable par un exploit disponible publiquement.

Aussi bien pour la protection que l'analyse (voir le tableau 6.4), on voit que les attaques commencent toutes par une exécution d'un interpréteur et la tentative ou la violation pour la propriété $NoExec$. Ensuite, le déroulement est similaire à l'exécution distante vu précédemment. On voit donc le même type de scénario se dérouler. La différence est un nombre d'attaques sur `Samba` que sur les services web dû à la popularité des attaques web.

	Protection	**Analyse**
$dataint(apache\_t, shadow\_t)$	0	158
$dataint(apache\_t, bin\_t, usr\_bin\_t)$	0	985
$dataint(apache\_t, lib\_t)$	0	1463
$dataconf(apache\_t, shadow\_t)$	0	29
$trans(apache\_t, *)$	0	2053
$vchroot(apache\_t)$	0	569841
$tpe(apache\_t, bin\_t, usr\_bin\_t)$	0	9651
$NoExec(apache\_t, shell\_exec\_t)$	302584	295349

TABLE 6.3 – phpBB et RoundCube avec une faille de Remote Shell Execution

	Protection	**Analyse**
$dataint(samba\_t, shadow\_t)$	0	9
$dataint(samba\_t, bin\_t, usr\_bin\_t)$	0	24
$dataint(samba\_t, lib\_t)$	0	31
$dataconf(samba\_t, shadow\_t)$	0	14
$trans(samba\_t, *)$	0	105
$vchroot(samba\_d)$	0	1879
$tpe(samba\_t, bin\_t, usr\_bin\_t)$	0	368
$NoExec(samba\_t, shell\_exec\_t)$	124	106

TABLE 6.4 – Samba avec une faille de Remote Shell Execution

6.4 Conclusion

Nous avons présenté dans ce chapitre la solution PIGA-DYN qui protége le système en garantissant les propriétés de sécurité formalisées au moyen de notre langage. Nous avons vu que cette implantation traite complètement les différents canaux cachés existants au niveau des applications. Ces canaux cachés reposent sur des ressources intermédiaires ainsi que sur l'utilisation d'informations de contrôle. Nous avons montré que le traitement des appels systèmes permet de protéger contre ces canaux. Les canaux cachés restant sont internes au noyau ou matériel, ils sortent du cadre des protections que peut garantir un noyau et relèvent essentiellement de la sûreté de fonctionnement.

L'implantation repose sur deux composants. Une partie s'exécute en espace noyau et l'autre en espace utilisateur. On pourrait penser que l'existance d'un processus en espace utilisateur pour contrôler les appels systèmes peut poser des problèmes de performance. Cependant, les analyses de performance, que nous avons menées, montrent qu'il n'en est rien. En effet, nous avons observé en moyenne un surcoût en terme de temps de traitement de l'ordre de 10%. Ces performances s'expliquent par une implantation concurrente au niveau du noyau du contrôle des appels systèmes. D'une part, cette concurrence est indispensable dès lors qu'une partie se déroule en espace utilisateur. D'autre part, cette concurrence est efficace puisqu'elle minimise en pratique les surcoûts observés. Cet ordre de grandeur est comparable à celui de SELinux. Il est très largement inférieur à ce que nous pourrions obtenir par coloration. De plus, les propriétés de sécurité garanties vont plus loin que SELinux. Donc globalement, le rapport protection/performance est très bon.

L'expérimentation menée à partir de pots de miel à haute interaction permet de valider, par la pratique, l'efficacité de nos méthodes de protection. Le HoneyPot Client permet d'analyser

un très grand nombre de vulnérabilités issues du web et ciblant le poste de travail client. Si les résultats donnés sont fournis en terme d'analyse afin de capturer un grand nombre de vulnérabilités et de scénarios, il est clair que les propriétés qui permettent d'analyser ces vulnérabilités sont entièrement capable de bloquer les attaques correspondantes. De plus, le modèle de protection dynamique que nous avons défini, permet de fournir une méthode de protection qui est utilisable et configurable facilement. Ce modèle s'adapte mal à un pot de miel puisqu'il est adapté au contrôle des interactions de l'utilisateur. Hors, ici, ce sont des outils automatiques qui parcourent le web et qui ne peuvent pas interagir comme un utilisateur final.

Le HoneyPot Serveur montre clairement l'efficacité de la méthode de protection. En effet, on voit que quelques propriétés de sécurité suffisent à bloquer les scénarios d'attaque malgré des vulnérabilités importantes mises en place sur ces systèmes. On voit que le fait de prendre en compte les flux indirects empêche quasiment tous les scénarios transitifs puisque nous ne constatons aucune tentative de violation de l'intégrité ou de la confidentialité. Même si de telles tentatives avaient pu aboutir, les propriétés, mises en place pour garantir l'intégrité et la confidentialité, auraient bloquées ces attaques.

Chapitre 7

Conclusion

Dans cette thèse, nous avons motivé les besoins de systèmes de protection qui permettent de garantir dynamiquement des propriétés de sécurité pour différents domaines d'usage. Nous avons justifié que seule la garantie dynamique de propriétés de sécurité permet de réaliser des systèmes protégeant efficacement différents domaines.

Notre état de l'art a montré les limitations des approches existantes. D'une part, il manque un moyen pour formaliser simplement les propriétés de sécurité observables sur un système d'exploitation. D'autre part, les solutions de la littérature traitent seulement d'un sous ensemble des propriétés requises. Ceci est clairement lié à l'absence de langage puisque sans moyen d'expression adapté, il n'est pas possible de formaliser et donc de garantir l'ensemble des propriétés de sécurité dont peut avoir besoin un administrateur. Les approches traitant les flux d'information indirects reposent le plus souvent sur une coloration des données ou sur une analyse à priori des flux. Ces méthodes ne sont pas applicables à l'échelle d'un système d'exploitation car elles sont trop complexes d'une part, et ne prennent pas en compte l'ensemble des propriétés de sécurité requises d'autre part. Les résultats du Défi Sécurité [ANR, 2008] ont montré la complexité des méthodes statiques pour réaliser une protection en profondeur. Dans ce cas, il faut calculer statiquement les politiques nécessaires à tous les niveaux du système et coordonner ces politiques en fonction des domaines d'usage.

Nous proposons un langage de description d'activités qui offre un ensemble d'opérateurs qui permettent d'analyser les traces d'exécution d'un système d'exploitation. Ces opérateurs traitent de tous les types de flux directs et indirects entre processus utilisateurs. Ils offrent un moyen de protéger contre les canaux cachés, c'est-à-dire, tous les flux indirects. Un des aspects essentiels de ces opérateurs est qu'ils traitent correctement les relations de causalité entre les flux directs. En effet, un flux indirect existe en pratique si il existe une relation de causalité entre les flux directs le composant. Nous proposons une sur-estimation de ces relations de causalité. En pratique, c'est la seule approche viable pour un système d'exploitation car elle revient à mesurer des dates. Elle n'oblige donc pas une coloration coûteuse des flux.

A l'aide de ce langage, nous avons formalisé l'essentiel des propriétés de sécurité de la littérature. Nous avons d'abord formalisé des propriétés classiques comme l'intégrité des données. Pour montrer l'adéquation de notre langage, nous avons essayé de donner un aperçu le plus large possible des propriétés de sécurité qui peuvent être formalisées. Nous avons défini d'autres propriétés que celles présentées dans ce document. Nous ne les avons pas exposées par souci de concision. Cependant, une formalisation du modèle dynamique de la muraille de Chine est proposée. Elle montre la capacité de notre langage à supporter des propriétés dynamiques. De plus, nous avons défini un nouveau modèle de protection dynamique. Celui-ci est adapté à la protection de domaines de navigation Web. Il permet d'éviter les flux indirects entre domaines de navigation différents, tout en labélisant dynamiquement ces domaines.

Nous définissons une méthode pour compiler la définition formelle des propriétés de sécurité. Cette méthode repose sur trois points. Tout d'abord, nous calculons un graphe de flux d'information. Ce graphe contient tous les flux d'information et les transitions des processus. Il a une complexité polynomiale avec le nombre de contextes de sécurité. Ce nombre étant réduit en pratique, l'implantation en mémoire de ce graphe ne pose pas de difficulté. Ensuite, nous proposons des fonctions pour chacun des opérateurs. Ces fonctions parcourent le graphe pour vérifier la présence des flux recherchés. Leur complexité est également polynomiale avec le nombre de contexte de sécurité. Finalement, nous proposons une méthode pour traduire automatiquement notre syntaxe formelle de définition de propriétés en un algorithme qui utilise les fonctions associées aux opérateurs. Les algorithmes, même si ils sont abstraits, sont tout à fait traduisibles dans n'importe quel langage de programmation impératif. Nous avons montré l'usage de cette compilation pour deux propriétés de sécurité. La méthode, que nous proposons, permet de calculer la complexité de la propriété. En pratique, les complexités restent polynomiales et tout à fait exécutable en un temps raisonnable.

Une application de notre langage à la protection des systèmes est présentée. La solution repose sur deux composants. Le composant PIGA-DYN-Protect intercepte et contrôle les appels systèmes. Il dialogue avec le composant PIGA-DYN en espace utilisateur afin de connaître les décisions d'autorisation. PIGA-DYN propose une mise en oeuvre dans le langage Java des algorithmes abstraits générés par le compilateur. Ainsi, nous avons un moyen pour analyser dynamiquement notre graphe de flux d'information et autoriser en conséquence les appels systèmes. Le surcoût de notre méthode de protection est de l'ordre de 10%. Il est légérement supérieur à ce que propose SELinux. Cependant, les propriétés de sécurité que nous garantissons, sont beaucoup plus avancées. Nous traitons notamment tous les flux indirects ce qui peut correspondre à plusieurs millions de possibilités de violations sur un système SELinux. De plus, la définition de la politique de sécurité est beaucoup plus simple avec notre langage. Il s'agit d'une dizaine de règles de protection contre plusieurs milliers de règles SELinux. Cela montre la simplicité d'usage de notre méthode.

Afin de valider l'efficacité de notre approche, nous avons conduit deux expérimentations à large échelle. Un pot de miel client nous a permis d'analyser des scénarios d'attaque basés sur les navigateurs Web. Ces scénarios montrent la nécessité de protéger non seulement l'intérieur d'un domaine de navigation (par exemple, les impôts) mais aussi les interactions entre domaines afin d'éviter les flux d'information illégaux (par exemple, un flux partant d'impôts vers les réseaux sociaux). L'analyse sur le pot de miel client a été réalisée au moyen de huit propriétés de sécurité. L'application de ces propriétés en terme de protection permet de protéger un internaute contre ces scénarios d'attaque. Cependant, l'intégration dans un système constitue une défense en profondeur qui peut nécessiter soit la modification du navigateur soit le contrôle de son exécution par un mécanisme externe. Une telle intégration a été réalisée dans le cadre du Défi Sécurité et est donc envisageable pour la protection d'un navigateur Web.

La protection d'un pot de miel serveur a été réalisée au moyen de huit propriétés. En pratique, une seule de ces propriétés permet d'empêcher l'élévation de privilèges qui est le point d'entrée des scénarios d'attaque. Cependant, les autres propriétés ont aussi un intérêt. En effet, dans certaines configurations, il est nécessaire de permettre les élévations de privilèges. Dans ce cas, les autres propriétés permettent d'éviter que ces élévations de privilèges autorisent la compromission du système.

Nous avons ainsi proposé une méthode très ouverte et efficace pour formaliser des propriétés de sécurité qui permettent une défense en profondeur. L'approche traite tous les niveaux nécessaires à l'automatisation et à la mise en oeuvre de l'approche. En effet, nous proposons un langage pour formaliser des propriétés de sécurité, une méthode pour compiler ce langage en algorithme abstrait, un support de ces algorithmes abstrait par les langages impératifs et une application de ce langage à la protection d'un système d'exploitation. Ainsi le système d'exploitation devient

capable de garantir des propriétés de sécurité exprimées dans notre langage. L'ensemble des propriétés définies ainsi que les expérimentations effectuées à large échelle montrent l'efficacité et l'extensibilité de l'approche.

Il existe de nombreuses perspectives d'utilisation de nos travaux.

Bien évidemment, les résultats sont applicables à la protection de différents systèmes. Nous avons proposé une mise en oeuvre sur GNU/Linux. Mais il est tout à fait possible d'appliquer cette méthode à d'autres systèmes, notamment MS-Windows. Des travaux sont en cours dans ce sens. L'essentiel du travail consiste à étiquetter les ressources et à intercepter les appels systèmes. Ensuite, le processus en espace utilisateur est tout à fait capable de réaliser les traitements dans le cas d'autres systèmes.

Notre approche a montré son efficacité pour formaliser des modèles de protection dynamique. De plus, elle est très adaptée puisque le contrôle des activités est entièrement dynamique. Il est donc envisageable de définir différents modèles de protection dynamique adaptée à des contextes particuliers. Nous avons proposé un nouveau modèle. Mais il est tout à fait possible d'en définir de nouveaux pour supporter d'autres usages.

Notre approche est applicable à l'analyse de programmes ou d'activités malicieuses. C'est ce que nous avons montré dans le cadre du pot de miel client. Il semble possible d'élargir cette approche pour qu'elle puisse être intégrée dans le cadre de laboratoires d'analyse virologique. Dans ce cadre, nous pourrions fournir un environnement d'exécution contrôlé pour les virus, qui permette à la fois d'empêcher la propagation des virus tout en les caractérisant par leur impact sur le système au moyen de nos propriétés de sécurité. Ainsi, non seulement, nous offririons un moyen d'analyse virale mais aussi le moyen de protection associé. De plus, la caractérisation par propriétés de sécurité permet de traiter des virus inconnus ayant des scénarios d'attaques similaires.

Bibliographie

[Ahn and Sandhu, 2000] Ahn, G.-J. and Sandhu, R. (2000). Role-based authorization constraints specification. *ACM Trans. Inf. Syst. Secur.*, 3(4) :207–226.

[Ammann and Sandhu, 1992] Ammann, P. and Sandhu, R. S. (1992). The extended schematic protection model. 1(3-4) :335–384.

[Amoroso and Merritt, 1994] Amoroso, E. and Merritt, M. (1994). Composing system integrity using i/o automata. In *Computer Security Applications Conference, 1994. Proceedings., 10th Annual*, pages 34 –43.

[Anderson, 1980] Anderson, J. (1980). Computer security threat monitoring and surveillance. Technical report, James P. Anderson Company, Fort Washington, Pennsylvania.

[ANR, 2008] ANR (2008). Défi de sécurité ANR SEC&SI. `http://goo.gl/obXf`.

[Badger et al., 1995] Badger, L., Sterne, D. F., Sherman, D. L., and 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.

[Basin et al., 2009] Basin, D., Burri, S. J., and Karjoth, G. (2009). Dynamic enforcement of abstract separation of duty constraints. In *ESORICS'09 : Proceedings of the 14th European conference on Research in computer security*, pages 250–267, Berlin, Heidelberg. Springer-Verlag.

[Bauer et al., 2002] Bauer, L., Ligatti, J., and Walker, D. (2002). More enforceable security policies. In *Foundations of Computer Security*, Copenhagen, Denmark.

[Bell and La Padula, 1973] Bell, D. E. and La Padula, L. J. (1973). Secure computer systems : Mathematical foundations and model. Technical Report M74-244, The MITRE Corporation, Bedford, MA.

[Biba, 1975] Biba, K. J. (1975). Integrity considerations for secure computer systems. Technical Report MTR-3153, The MITRE Corporation.

[Bishop, 1995] Bishop, M. (1995). Race conditions, files, and security flaws : or, the tortoise and the hare redux. *Technical Report*, 95(8).

[Bishop, 2003] Bishop, M. (2003). *Computer Security Art and Science*. Number ISBN 0201440997. Addison-Wesley Professional.

[Bishop and Dilger, 1996] Bishop, M. and Dilger, M. (1996). Checking for race conditions in file accesses. *Computing Systems*, 9 :131–152.

[Boebert and Kain, 1985] Boebert, W. E. and 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 and Nash, 1989a] Brewer, D. and Nash, M. (1989a). The chinese wall security policy. In *Security and Privacy, 1989. Proceedings., 1989 IEEE Symposium on*, pages 206–214.

[Brewer and Nash, 1989b] Brewer, D. F. C. and Nash, M. J. (1989b). The chinese wall security policy. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 206–214, Oakland, CA, USA. IEEE.

[Briffaut, 2007] Briffaut, J. (2007). *Formalisation et garantie de propriétés de sécurité système : application à la détection d'intrusions*. PhD thesis, Université d'Orléans. SDS.

[Briffaut et al., 2009a] Briffaut, J., Lalande, J.-F., and Toinard, C. (2009a). Formalization of security properties : enforcement for MAC operating systems and verification of dynamic MAC policies. *International journal on advances in security*, 2(4) :325–343. ISSN : 1942-2636.

[Briffaut et al., 2009b] Briffaut, J., Rouzaud Cornabas, J., and Toinard, C. (2009b). SEC&SI : Un défi pour la réalisation d'un système d'exploitation cloisonné et sécurisé pour l'Internaute : Le Projet SPACLik. In *Symposium sur la Sécurité des Technologies de l'Information et de la Communication 2009*, Rennes France.

[Chaudhuri et al., 2009] Chaudhuri, A., Naldurg, P., and Rajamani, S. (2009). A type system for data-flow integrity on Windows Vista. *ACM SIGPLAN Notices*, 43(12) :9.

[Chen and Wagner, 2002] Chen, H. and Wagner, D. (2002). Mops : an infrastructure for examining security properties of software. In *CCS '02 : Proceedings of the 9th ACM conference on Computer and communications security*, pages 235–244, New York, NY, USA. ACM.

[Chess, 2002] Chess, B. (2002). Improving computer security using extended static checking. In *Security and Privacy, 2002. Proceedings. 2002 IEEE Symposium on*, pages 160 – 173.

[Clark and Wilson, 1987] Clark, D. D. and Wilson, D. R. (1987). A Comparison of Commercial and Military Computer Security Policies. *Proc. IEEE Symp. Computer Security and Privacy, IEEE CS Press*, pages 184–194.

[Clarkson and Schneider, 2008] Clarkson, M. R. and Schneider, F. B. (2008). Hyperproperties. *Computer Security Foundations Symposium, IEEE*, 0 :51–65.

[Clause et al., 2007] Clause, J., Li, W., and Orso, A. (2007). Dytan : a generic dynamic taint analysis framework. In *ISSTA '07 : Proceedings of the 2007 international symposium on Software testing and analysis*, pages 196–206, New York, NY, USA. ACM.

[Clemente et al., 2010] Clemente, P., Rouzaud Cornabas, J., and Toinard, C. (2010). From a Generic Framework for Expressing Integrity Properties to a Dynamic MAC Enforcement for Operating Systems. *Transactions on Computational Sciences Journal*, page 0. 27 pages, to appear.

[Cova et al., 2010] Cova, M., Kruegel, C., and Vigna, G. (2010). Detection and analysis of drive-by-download attacks and malicious javascript code. In *WWW '10 : Proceedings of the 19th international conference on World wide web*, pages 281–290, New York, NY, USA. ACM.

[Cowan et al., 2001] Cowan, C., Beattie, S., Wright, C., and Kroah-Hartman, G. (2001). Raceguard : kernel protection from temporary file race vulnerabilities. In *SSYM'01 : Proceedings of the 10th conference on USENIX Security Symposium*, pages 13–13, Berkeley, CA, USA. USENIX Association.

[Dalton et al., 2007] Dalton, M., Kannan, H., and Kozyrakis, C. (2007). Raksha : a flexible information flow architecture for software security. In *ISCA '07 : Proceedings of the 34th annual international symposium on Computer architecture*, pages 482–493, New York, NY, USA. ACM.

[Dalton et al., 2008] Dalton, M., Kannan, H., and Kozyrakis, C. (2008). Real-world buffer overflow protection for userspace & kernelspace. In *SS'08 : Proceedings of the 17th conference on Security symposium*, pages 395–410, Berkeley, CA, USA. USENIX Association.

[Efstathopoulos and Kohler, 2008] Efstathopoulos, P. and Kohler, E. (2008). Manageable fine-grained information flow. *SIGOPS Oper. Syst. Rev.*, 42(4) :301–313.

[Ferraiolo and Kuhn, 1992] Ferraiolo, D. F. and Kuhn, D. R. (1992). Role-based access controls. In *15th National Computer Security Conference*, pages 554–563, Baltimore, MD, USA.

[Focardi and Gorrieri, 2001] Focardi, R. and Gorrieri, R. (2001). Classification of security properties (part i : Information flow).

[Foley, 2003] Foley, S. (2003). A nonfunctional approach to system integrity. *Selected Areas in Communications, IEEE Journal on*, 21(1) :36 – 43.

[Foley et al., 1996] Foley, S., Gong, L., and Qian, X. (1996). A security model of dynamic labeling providing a tiered approach to verification. In *SP '96 : Proceedings of the 1996 IEEE Symposium on Security and Privacy*, page 142, Washington, DC, USA. IEEE Computer Society.

[Forrest et al., 1997] Forrest, S., Hofmeyr, S. A., and Somayaji, A. (1997). Computer immunology. *Communications of the ACM*, 40(10) :88–96.

[Fraser, 2000] Fraser, T. (2000). LOMAC : Low Water-Mark integrity protection for COTS environments. *Proceeding 2000 IEEE Symposium on Security and Privacy. S&P 2000*, pages 230–245.

[Fraser and Badger, 1998] Fraser, T. and Badger, L. (1998). Ensuring continuity during dynamic security policy reconfiguration in dte. pages 15 –26.

[Goguen and Meseguer, 1982] Goguen, J. and Meseguer, J. (1982). Security policies and security models. In *Proc. 1982 IEEE symp. Security and Privacy*, pages 11–20, Oakland, CA. IEEE.

[Guttman et al., 2005] Guttman, J. D., Herzog, A. L., Ramsdell, J. D., and Skorupka, C. W. (2005). Verifying information flow goals in security-enhanced linux. *J. Comput. Secur.*, 13(1) :115–134.

[Harrison et al., 1976] Harrison, M. A., Ruzzo, W. L., and Ullman, J. D. (1976). Protection in operating systems. *Communications of the ACM*, 19(8) :461–471.

[Hicks et al., 2006] Hicks, B., King, D., McDaniel, P., and Hicks, M. (2006). Trusted declassification : : high-level policy for a security-typed language. In *PLAS '06 : Proceedings of the 2006 workshop on Programming languages and analysis for security*, pages 65–74, New York, NY, USA. ACM.

[Hicks et al., 2007] Hicks, B., Rueda, S., Jaeger, T., and McDaniel, P. (2007). From trusted to secure : building and executing applications that enforce system security. In *ATC'07 : 2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference*, pages 1–14, Berkeley, CA, USA. USENIX Association.

[Iglio, 1999] Iglio, P. (1999). Trustedbox : a kernel-level integrity checker. In *Computer Security Applications Conference, 1999. (ACSAC '99) Proceedings. 15th Annual*, pages 189 –198.

[ISO/IEC, 2009] ISO/IEC (2009). Common criteria for information technology security evaluation. In *Part 1 : Introduction and general model*, number Version 3.1, Revision 3.

[ITSEC, 1991] ITSEC (1991). Information Technology Security Evaluation Criteria (ITSEC) v1.2. Technical report.

[Jacob, 1991] Jacob, J. (1991). The basic integrity theorem. In *Computer Security Foundations Workshop IV, 1991. Proceedings*, pages 89 –97.

[Jaeger and Tidswell, 2001] Jaeger, T. and Tidswell, J. E. (2001). Practical safety in flexible access control models. *ACM Trans. Inf. Syst. Secur.*, 4(2) :158–190.

[Joshi et al., 2005] Joshi, A., King, S. T., Dunlap, G. W., and Chen, P. M. (2005). Detecting past and present intrusions through vulnerability-specific predicates. In *SOSP '05 : Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 91–104, New York, NY, USA. ACM.

[Knorr, 2000] Knorr, K. (2000). Dynamic access control through petri net workflows. In *ACSAC '00 : Proceedings of the 16th Annual Computer Security Applications Conference*, page 159, Washington, DC, USA. IEEE Computer Society.

[Ko et al., 1994] Ko, C., Fink, G., and 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 and Redmond, 2002] Ko, C. and Redmond, T. (2002). Noninterference and intrusion detection. In *Security and Privacy, 2002. Proceedings. 2002 IEEE Symposium on*, pages 177–187.

[Ko et al., 1997] Ko, C., Ruschitzka, M., and Levitt, K. (1997). Execution monitoring of security-critical programs in distributed systems : A specification-based approach. pages 175–187.

[Krohn and Tromer, 2009] Krohn, M. and Tromer, E. (2009). Noninterference for a practical difc-based operating system. In *SP '09 : Proceedings of the 2009 30th IEEE Symposium on Security and Privacy*, pages 61–76, Washington, DC, USA. IEEE Computer Society.

[Kuhn, 1997] Kuhn, D. R. (1997). Mutual exclusion of roles as a means of implementing separation of duty in role-based access control systems. In *RBAC '97 : Proceedings of the second ACM workshop on Role-based access control*, pages 23–30, New York, NY, USA. ACM.

[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, 1988] Lee, T. M. P. (1988). Using mandatory integrity to enforce "commercial" security. *In Proc. IEEE Symposium on Security and Privacy*, pages :140–146.

[Lhee and Chapin, 2005] Lhee, K.-s. and Chapin, S. J. (2005). Detection of file-based race conditions. *International Journal of Information Security*, 4 :105–119. 10.1007/s10207-004-0068-2.

[Li et al., 2007a] Li, N., Mao, Z., and Chen, H. (2007a). Usable mandatory integrity protection for operating systems. In *SP '07 : Proceedings of the 2007 IEEE Symposium on Security and Privacy*, pages 164–178, Washington, DC, USA. IEEE Computer Society.

[Li et al., 2007b] Li, N., Tripunitara, M. V., and Bizri, Z. (2007b). On mutually exclusive roles and separation-of-duty. *ACM Trans. Inf. Syst. Secur.*, 10(2) :5.

[Liang and Sun, 2001] Liang, H. and Sun, Y. (2001). Enforcing Mandatory Integrity Protection in Operating System. In *ICCNMC '01 : Proceedings of the 2001 International Conference on Computer Networks and Mobile Computing (ICCNMC'01)*, page 435, Washington, DC, USA. IEEE Computer Society.

[Lin, 1989] Lin, T. (1989). Chinese wall security policy-an aggressive model. pages 282 –289.

[Lin, 2000] Lin, T. Y. (2000). Chinese wall security model and conflict analysis. pages 122 –127.

[Lin, 2007] Lin, T. Y. (2007). Chinese wall security policy-revisited a short proof. pages 3027 –3028.

[Lin and Pan, 2009] Lin, T. Y. and Pan, J. (2009). Granular computing and flow analysis on discretionary access control : Solving the propagation problem. pages 2965 –2971.

[Loscocco et al., 1998] Loscocco, P. A., Smalley, S. D., Muckelbauer, P. A., Taylor, R. C., Turner, S. J., and 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.

[Mao et al., 2009] Mao, Z., Li, N., Chen, H., and Jiang, X. (2009). Trojan horse resistant discretionary access control. In *SACMAT '09 : Proceedings of the 14th ACM symposium on Access control models and technologies*, pages 237–246, New York, NY, USA. ACM.

[Mazieres and Kaashoek, 1997] Mazieres, D. and Kaashoek, M. (1997). Secure applications need flexible operating systems. In *Operating Systems, 1997., The Sixth Workshop on Hot Topics in*, pages 56 –61.

[McPhee, 1974] McPhee, W. S. (1974). Operating system integrity in os/vs2. *IBM Syst. J.*, 13(3) :230–252.

[Mohay and Zellers, 1997] Mohay, G. and Zellers, J. (1997). Kernel and shell based applications integrity assurance. In *ACSAC '97 : Proceedings of the 13th Annual Computer Security Applications Conference*, page 34, Washington, DC, USA. IEEE Computer Society.

[Netzer and Miller, 1990] Netzer, R. H. and Miller, B. P. (1990). On the complexity of event ordering for shared-memory parallel program executions. In *In Proceedings of the 1990 International Conference on Parallel Processing*, pages 93–97.

[Netzer and Miller, 1992] Netzer, R. H. B. and Miller, B. P. (1992). What are race conditions ? some issues and formalizations. *LOPLAS*, 1(1) :74–88.

[Newsome and Song, 2005] Newsome, J. and Song, D. (2005). Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the Network and Distributed System Security Symposium (NDSS 2005)*.

[Nuansri et al., 1999] Nuansri, N., Singh, S., and Dillon, T. S. (1999). A process state-transition analysis and its application to intrusion detection. In *ACSAC '99 : Proceedings of the 15th Annual Computer Security Applications Conference*, page 378, Washington, DC, USA. IEEE Computer Society.

[Nyanchama and Osborn, 1999] Nyanchama, M. and Osborn, S. (1999). The role graph model and conflict of interest. *ACM Trans. Inf. Syst. Secur.*, 2(1) :3–33.

[Provos et al., 2007] Provos, N., McNamee, D., Mavrommatis, P., Wang, K., and Modadugu, N. (2007). The ghost in the browser analysis of web-based malware. In *HotBots'07 : Proceedings of the first conference on First Workshop on Hot Topics in Understanding Botnets*, pages 4–4, Berkeley, CA, USA. USENIX Association.

[Qin et al., 2006] Qin, F., Wang, C., Li, Z., Kim, H.-s., Zhou, Y., and Wu, Y. (2006). Lift : A low-overhead practical information flow tracking system for detecting security attacks. In *MICRO 39 : Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 135–148, Washington, DC, USA. IEEE Computer Society.

[Radhakrishnan and Solworth, 2006] Radhakrishnan, M. and Solworth, J. A. (2006). Application security support in the operating system kernel. In *ASIACCS '06 : Proceedings of the 2006 ACM Symposium on Information, computer and communications security*, pages 201–211, New York, NY, USA. ACM.

[Rahimi, 2004] Rahimi, N. A. (2004). Trusted path execution for the linux 2.6 kernel as a linux security module. In *ATEC '04 : Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 34–34, Berkeley, CA, USA. USENIX Association.

[Rouzaud Cornabas et al., 2010] Rouzaud Cornabas, J., Clemente, P., and Toinard, C. (2010). An Information Flow Approach for Preventing Race Conditions : Dynamic Protection of the Linux OS. In *Fourth International Conference on Emerging Security Information, Systems and Technologies SECURWARE'10*, pages 11–16, Venise Italie.

[Rushby, 1984] Rushby, J. (1984). The bell and la padula security model. Technical report, SRI.

[Sabelfeld and Myers, 2003] Sabelfeld, A. and Myers, A. (2003). Language-based information-flow security. *Selected Areas in Communications, IEEE Journal on*, 21(1) :5 – 19.

[Saltzer and Schroeder, 1975] Saltzer, J. and Schroeder, M. (1975). The protection of information in computer systems. *Proceedings of the 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, 1992a] Sandhu, R. (1992a). A lattice interpretation of the Chinese Wall policy. In *Proceedings of the 15th NIST-NCSC National Computer Security Conference*, pages 329–339. NIST.

[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, 1992b] Sandhu, R. S. (1992b). The Typed Access Matrix Model. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, pages 122–136, Oakland, CA, USA. IEEE.

[Sandhu et al., 1996] Sandhu, R. S., Coyne, E. J., Feinstein, H. L., and Youman, C. E. (1996). Role-based access control models. *Computer*, 29 :38–47.

[Schmuck and Wylie, 1991] Schmuck, F. and Wylie, J. (1991). Experience with transactions in quicksilver. In *SOSP '91 : Proceedings of the thirteenth ACM symposium on Operating systems principles*, pages 239–253, New York, NY, USA. ACM.

[Schneider, 2000] Schneider, F. B. (2000). Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3(1) :30–50.

[Schwarz et al., 2005] Schwarz, B., Chen, H., Wagner, D., Lin, J., Tu, W., Morrison, G., and West, J. (2005). Model checking an entire linux distribution for security violations. In *ACSAC '05 : Proceedings of the 21st Annual Computer Security Applications Conference*, pages 13–22, Washington, DC, USA. IEEE Computer Society.

[Sekar et al., 1999] Sekar, R., Bowen, T. F., and 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., and Segal, M. (1998). A specification-based approach for building survivable systems. In *Proc. 21st NIST-NCSC National Information Systems Security Conference*, pages 338–347.

[Simon and Zurko, 1997] Simon, R. and Zurko, M. E. (1997). Separation of duty in role-based environments. In *CSFW '97 : Proceedings of the 10th IEEE workshop on Computer Security Foundations*, page 183, Washington, DC, USA. IEEE Computer Society.

[Tahara et al., 2008] Tahara, T., Gondow, K., and Ohsuga, S. (2008). Dracula : Detector of data races in signals handlers. pages 17–24, Beijing, China. IEEE Computer Society.

[TCSEC, 1985] TCSEC (1985). Trusted Computer System Evaluation Criteria. Technical Report DoD 5200.28-STD, Department of Defense.

[Triem Tong et al., 2010] Triem Tong, V., Clark, A., and Me, L. (2010). Specifying and Enforcing a Fine-Grained Information Flow Policy : Model and Experiments. *Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications*, 1(1) :56.

[Tsafrir et al., 2008] Tsafrir, D., Hertz, T., Wagner, D., and Da Silva, D. (2008). Portably solving file tocttou races with hardness amplification. In *FAST'08 : Proceedings of the 6th USENIX Conference on File and Storage Technologies*, pages 1–18, Berkeley, CA, USA. USENIX Association.

[Tsyrklevich and Yee, 2003] Tsyrklevich, E. and Yee, B. (2003). Dynamic detection and prevention of race conditions in file accesses. In *SSYM'03 : Proceedings of the 12th conference on USENIX Security Symposium*, pages 17–17, Berkeley, CA, USA. USENIX Association.

[Uppuluri et al., 2005] Uppuluri, P., Joshi, U., and Ray, A. (2005). Preventing race condition attacks on file-systems. In *SAC '05 : Proceedings of the 2005 ACM symposium on Applied computing*, pages 346–353, New York, NY, USA. ACM.

[Uppuluri and Sekar, 2001] Uppuluri, P. and Sekar, R. (2001). Experiences with specification-based intrusion detection. In Lee, W., Mé, L., and Wespi, A., editors, *Recent Advances in Intrusion Detection, 4th International Symposium, RAID 2001 Davis, CA, USA, October 10-12, 2001, Proceedings*, pages 172–189. Springer.

[Van Hentenryck, 1989] Van Hentenryck, P. (1989). *Constraint satisfaction in logic programming*. MIT Press, Cambridge, MA, USA.

[Vandebogart et al., 2007] Vandebogart, S., Efstathopoulos, P., Kohler, E., Krohn, M., Frey, C., Ziegler, D., Kaashoek, F., Morris, R., and Mazières, D. (2007). Labels and event processes in the asbestos operating system. *ACM Trans. Comput. Syst.*, 25(4) :11.

[Wang et al., 2008] Wang, X., Li, Z., Li, N., and Choi, J. Y. (2008). Precip : Towards practical and retrofittable confidential information protection. In *NDSS*. The Internet Society.

[Wray, 1991] Wray, J. (1991). An analysis of covert timing channels. pages 2 –7.

[Wright et al., 2007] Wright, C. P., Spillane, R., Sivathanu, G., and Zadok, E. (2007). Extending acid semantics to the file system. *Trans. Storage*, 3(2) :4.

[Yin et al., 2007] Yin, H., Song, D., Egele, M., Kruegel, C., and Kirda, E. (2007). Panorama : capturing system-wide information flow for malware detection and analysis. In *CCS '07 : Proceedings of the 14th ACM conference on Computer and communications security*, pages 116–127, New York, NY, USA. ACM.

[Yu et al., 2005] Yu, Y., Rodeheffer, T., and Chen, W. (2005). Racetrack : efficient detection of data race conditions via adaptive tracking. *SIGOPS Oper. Syst. Rev.*, 39(5) :221–234.

[Zeldovich et al., 2006] Zeldovich, N., Boyd-Wickizer, S., Kohler, E., and Mazières, D. (2006). Making information flow explicit in histar. In *OSDI '06 : Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, pages 19–19, Berkeley, CA, USA. USENIX Association.

Chapitre 8

Annexes

8.1 MLS

8.1.1 BIBA

Dans cette section, nous modélisons une propriété extraite de la littérature : BIBA. Le but de cette propriété est le respect de trois régles distinctes permettant de garantir l'intégrité du système :

biba-1 Pour qu'un sujet puisse accéder en lecture à un objet, son niveau d'intégrité doit être inférieur ou égal à celui de l'objet.

biba-2 Pour qu'un sujet puisse accéder en écriture à un objet, son niveau d'intégrité doit être supérieur ou égal à celui de l'objet.

biba-3 Pour qu'un sujet puisse invoquer un second sujet , son niveau d'intégrité doit être supérieur ou égal à celui du second sujet.

Il y aura rupture de la propriété, et donc de l'intégrité, si l'une de ces régles n'est pas respectée.

Un niveau d'intégrité est défini pour chaque contexte. Dans notre modélisation, le niveau d'intégrité a pour forme un label supplémentaire sur chaque contexte. Afin de faciliter l'utilisation des niveaux d'intégrité, nous introduisons une fonction $int(cs)$ qui retourne le niveau d'intégrité du contexte : $int : \mathcal{CS} \rightarrow \mathbb{N}$. Les niveaux d'intégrité sont fixés par l'administrateur.

Nous utilisons la fonction $int()$ ainsi que les trois régles de Biba pour modéliser la propriété. La définition 8.1.1 modélise la régle **biba-1**. En pratique, la régle **biba-1** est respectée si quand un sujet css_1 accéde en lecture à un objet cso, son niveau d'intégrité est inférieur ou égal à celui de l'objet i.e. $int(css_1) \leq int(cso)$.

Définition 8.1.1 (Régle Biba-1) *Un objet cso est dit intègre vis-à-vis d'un sujet css_1 suivant biba-1 pour une trace T ssi il n'existe que des flux d'information direct généraux de cso vers css_1 avec $css_1 \in \mathcal{CSS}$, $cso \in \mathcal{CSO}$ et $int(css_1) \leqslant int(cso)$.*

$$Biba1(T, css_1, cso) \stackrel{def}{\equiv} (cso \overset{T}{\triangleright} css_1) \wedge (int(css_1) \leqslant int(cso)) \tag{8.1}$$

La définition 8.1.2 modélise la régle *biba-2*. En pratique, la régle **biba-2** est respectée si quand un sujet css_1 accéde en écriture à un objet cso, son niveau d'intégrité $int(css_1)$ est supérieur ou égal à celui de l'objet $int(cso)$.

Définition 8.1.2 (Régle Biba-2) *Un objet cso est dit intègre vis-à-vis d'un sujet css suivant biba-2 pour une trace T ssi il n'existe que des flux d'information direct généraux de css vers cso avec $css \in \mathcal{CSS}$, $cso \in \mathcal{CSO}$ et $int(css) \geqslant int(cso)$.*

$$Biba2(T, css, cso) \stackrel{def}{\equiv} (css \overset{T}{\triangleright} cso) \wedge (int(css) \geqslant int(cso)) \tag{8.2}$$

La définition 8.1.3 modélise la régle *biba-3*. En pratique, la régle **biba-3** est respectée si quand un sujet css_1 invoque un second sujet css_2, son niveau d'intégrité $int(css_1)$ est supérieur ou égal à celui de l'autre sujet $int(css_2)$.

Définition 8.1.3 (Régle Biba-3) *Un sujet css_2 est dit intègre vis-à-vis d'un second sujet css_1 suivant biba-3 pour une trace T ssi il n'existe que des transitions directes généraux de css_1 vers css_2 avec $css_1, css_2 \in \mathcal{CSS}$ et $int(css_1) \geqslant int(css_2)$.*

$$Biba3(T, css_1, css_2) \stackrel{def}{\equiv} (css_1 \overset{T}{\underset{t}{\rhd}} css_2) \wedge (int(css_1) \geqslant int(css_2)) \tag{8.3}$$

La propriété Biba est respectée si et seulement si les trois régles sont respectées. Nous formalisons donc la propriété d'intégrité de Biba tel que suit :

Définition 8.1.4 (Biba) *Soit $S1 = (css_1, cso_1), ..., (css_m, cso_m)$, un ensemble de couples de contextes et $S2 = (css_1, css_2), ..., (css_n, css_{n+1})$, un ensemble de couples de sujets. Un **système**, représenté par une trace T, est dit intégre selon Biba si et seulement si les définitions 8.1.1 et 8.1.2 sont respectées pour tous les couples de contextes du système (css_i, cso_i) avec $css_i \in \mathcal{CSS}$ et $cso_i \in \mathcal{CSO}$ et la définition 8.1.3 est respectée pour tous les couples de sujets du système (css_j, css_{j+1}) avec $css_j, css_{j+1} \in \mathcal{CSS}$.*

$$Biba(T, S1, S2) \stackrel{def}{\equiv} \forall i = 1..m, Biba1(T, css_i, cso_i) \wedge Biba2(T, css_i, cso_i), \\ \forall j = 1..n+1 Biba3(T, css_j, cs_j)$$

Par exemple, un des objectifs de cette propriété peut être d'empêcher un processus, i.e. un sujet, qualifié de non sûr, avec un faible niveau d'intégrité, de modifier un objet sûr avec un fort niveau d'intégrité. Nous utilisons le label $untrusted\_user\_t$ pour le processus ayant un faible niveau d'intégrité, $root$ pour le processus ayant un fort niveau d'intégrité et $shadow\_t$, i.e. `/etc/shadow`, pour l'objet sûr ayant un fort niveau d'intégrité. Ces niveaux sont décrits dans le tableau 8.1.

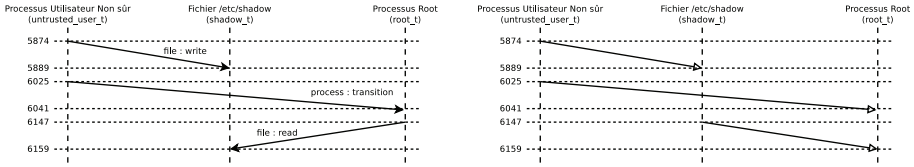

FIGURE 8.1 – Représentation d'une trace système avec rupture de Biba

La trace T, voir à gauche de la figure 8.1, représente le système et les flux d'information sont représentés dans la figure 8.1 à gauche. La première interaction, $untrusted\_user\_t \overset{T}{\underset{file:write}{\rightharpoonup}} [5874, 5889]\ shadow\_t$, peut être représentée sous la forme du flux d'information $untrusted\_user\_t \overset{T}{\underset{[5874,5889]}{\rhd_t}} shadow\_t$. La deuxième régle de Biba s'applique à ce flux i.e. un sujet écrit dans un objet. Il faut donc, pour que la propriété soit respectée, que le niveau du sujet soit supérieur ou égal à celui de l'objet. Hors, $0 \leq 7$, la régle n'est pas respectée et l'interaction estampillée correspond à une action interdite. En conséquence, la rupture est détectée et dans le cas de la protection, l'interaction est annulée. Le graphe représentant la trace est donc modifié pour devenir celui représentée dans la figure 8.2.

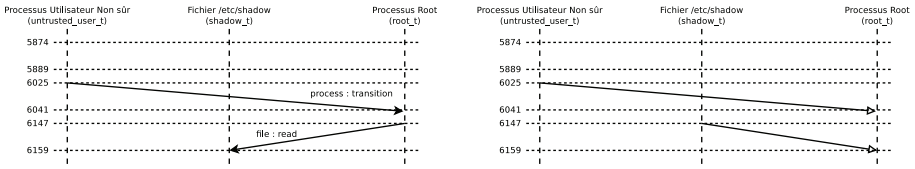

FIGURE 8.2 – Représentation d'une trace système sans rupture de la 2ème régle de Biba

La seconde interaction, $untrusted\_user\_t \overset{process:transition}{\underset{[6025,6041]}{\searrow}} root\_t$, peut être représentée sous la forme d'une transition $untrusted\_user\_t \overset{T}{\underset{[6025,6041]}{\rhd_t}} root\_t$. La troisième régle de Biba s'applique à cette transition i.e. un sujet invoque un second sujet. Il faut que le niveau du sujet appellant soit supérieur ou égale à celui invoqué. Hors, $0 \leq 10$, la régle n'est pas respectée. En conséquence, la rupture est détectée et dans le cas de la protection, l'interaction est annulée. Le graphe représentant la trace est donc modifié pour devenir celui représentée dans la figure 8.3.

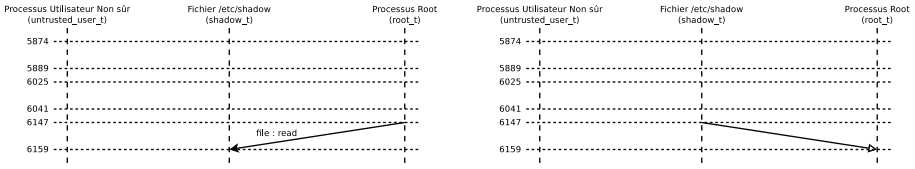

FIGURE 8.3 – Représentation d'une trace système sans rupture de la 3ème et 2ème régle de Biba

La troisième interaction, $untrusted\_user\_t \xrightarrow[file:read]{T} [6147, 6159]\ shadow\_t$, peut être représentée sous la forme du flux d'information $shadow\_t \underset{[6147,6159]}{\overset{T}{\triangleright_t}} untrusted\_user\_t$. La première régle de Biba s'applique à ce flux i.e. un sujet lit un objet. Il faut pour que la propriété soit respectée que le niveau du sujet soit inférieur ou égal à celui de l'objet. Hors, $0 \leq 7$, la régle est respectée et l'interaction estampillée est donc légitime et acceptée.

entité	**Niveau d'Intégrité**
untrusted_user_t	0
root	10
`/etc/shadow`	7

TABLE 8.1 – Tableau des niveaux d'intégrité

8.1.2 BIBA avec Plage de niveau

Cette propriété dite **Biba avec plages de niveau** repose sur le respect des trois régles propres à la propriété classique Biba mais en les étendant à non plus un niveau d'intégrité par contexte de sécurité mais à une plage de niveaux. En conséquence, les régles doivent être modifiées pour supporter ces plages :

bibap-1 Pour qu'un sujet puisse accéder en lecture à un objet, la borne supérieure de sa plage de niveau d'intégrité doit être inférieure ou égale à la borne inférieure de la plage de niveau d'intégrité de l'objet.

bibap-2 Pour qu'un sujet puisse accéder en écriture à un objet, la borne inférieure de sa plage de niveau d'intégrité doit être supérieure ou égale à la borne supérieure de la plage de niveau d'intégrité de l'objet.

bibap-3 Pour qu'un sujet puisse invoquer un second sujet, la borne inférieure de sa plage de niveau d'intégrité doit être supérieure ou égale à la borne supérieure de la plage de niveau d'intégrité du second sujet.

Le niveau d'intégrité étant représenté par une plage de niveaux d'intégrité, nous introduisons trois nouvelles fonctions permettant de connaitre les bornes de cette plage. La fonction $int2$: $\mathcal{CS} \rightarrow \mathbb{N} \times \mathbb{N}$ associe à un contexte sa plage de niveaux d'intégrité. Nous définissons également $intmax : \mathcal{CS} \rightarrow \mathbb{N}$ qui retourne la borne supérieure de la plage de niveau d'intégrité et $intmin$: $\mathcal{CS} \rightarrow \mathbb{N}$ qui retourne la borne inférieure de la plage.

Nous utilisons les fonctions $intmin()$ et $intmax()$ ainsi que les trois régles pour exprimer la propriété Biba avec plages dans notre modèle formel.

La définition 8.1.5 modélise la régle **bibap-1**. La régle **bibap-1** est respectée si quand un sujet css_1 accéde en lecture à un objet *cso*, la plage de niveau d'intégrité du sujet $int(css_1)$ est inférieure ou égal à la plage de niveau d'intégrité de l'objet $int(cso)$ i.e. la borne supérieure de la plage de niveau d'intégrité du sujet $intmax(css_1)$ est inférieure ou égale à la borne inférieure de la plage de niveau d'intégrité de l'objet $intmin(cso)$.

Définition 8.1.5 (Régle BibaP-1) *Un objet cso est dit intègre vis-à-vis d'un sujet css_1 suivant bibap-1 pour une trace T ssi il n'existe que des flux d'information directs généraux de cso vers css_1 avec $css_1 \in \mathcal{CSS}$, $cso \in \mathcal{CSO}$ et $intmax(css_1) \leqslant intmin(cso)$.*

$$BibaP1(T, css_1, cso) \overset{def}{\equiv} (cso \overset{T}{\triangleright} css_1) \wedge (intmax(css_1) \leqslant intmin(cso)) \tag{8.4}$$

La définition 8.1.6 modélise la régle *bibap-2*. La régle **bibap-2** est respectée si quand un sujet css_1 accéde en écriture à un objet cso, le niveau d'intégrité du sujet $int(css_1)$ est supérieur ou égal au niveau d'intégrité de l'objet $int(cso)$ i.e. la borne inférieure de la plage de niveau d'intégrité du sujet $intmin(css_1)$ est supérieure ou égale à la borne supérieure de la plage de niveau d'intégrité de l'objet $intmax(cso)$.

Définition 8.1.6 (Régle BibaP-2) *Un objet cso est dit intègre vis-à-vis d'un sujet css_1 suivant bibap-2 pour une trace T ssi il n'existe que des flux d'information directs généraux de css_1 vers cso avec $css_1 \in \mathcal{CSS}$, $cso \in \mathcal{CSO}$ et $intmin(css_1) \geqslant intmax(cso)$.*

$$BibaP2(T, css_1, cso) \stackrel{def}{\equiv} (css_1 \overset{T}{\rhd} cso) \wedge (intmin(css_1) \geqslant intmax(cso)) \tag{8.5}$$

La définition 8.1.7 modélise la régle *bibap-3*. La régle **bibap-3** est respectée si quand un sujet css_1 invoque un second sujet css_2, le niveau d'intégrité du sujet invoquant $int(css_1)$ est supérieur ou égal au niveau d'intégrité du sujet invoqué $int(css_2)$ i.e. la borne inférieure de sa plage de niveau d'intégrité $intmin(css_1)$ est supérieure ou égale à la borne supérieure de la plage de niveau d'intégrité du sujet invoké $intmax(css_2)$.

Définition 8.1.7 (Régle BibaP-3) *Un sujet css_2 est dit intègre vis-à-vis d'un second sujet css_1 suivant biba-3 pour une trace T ssi il n'existe que des transitions directes généraux de css_1 vers css_2 avec $css, css_2 \in \mathcal{CSS}$ et $int(css_1) \geqslant int(css_2)$.*

$$Biba3P(T, css_1, css_2) \stackrel{def}{\equiv} (css_1 \overset{T}{\underset{t}{\rhd}} css_2) \wedge (intmin(css_1) \geqslant intmax(css_2)) \tag{8.6}$$

La propriété d'intégrité de BIBA est donc respectée si les définitions (8.1.5), (8.1.6) et (8.1.7) sont respectées.

Nous définissons ainsi la propriété d'intégrité de Biba avec Plage (*BibaP*) :

Définition 8.1.8 (BibaP) *Soit $S = (css_1, cso_1), ..., (css_m, cso_m)$, un ensemble de couples de contextes et $S2 = (css_1, css_2), ..., (css_n, css_{n+1})$, un ensemble de couples de sujets. Un **système**, représenté par une trace T, est dit intégre selon BibaP si et seulement si les définitions 8.1.5 et 8.1.6 sont respectées pour tous les couples de contextes du système (css_i, cso_i) avec $css_i \in \mathcal{CSS}$ et $cso_i \in \mathcal{CSO}$ et la définition 8.1.7 est respectée pour tous les couples de sujets du système (css_j, css_{j+1}) avec $css_j, css_{j+1} \in \mathcal{CSS}$.*

$$BibaP(T, S1, S2) \stackrel{def}{\equiv} \forall i = 1..m, BibaP1(T, css_i, cso_i) \wedge BibaP2(T, css_i, cso_i), \\ \forall j = 1..n+1 BibaP3(T, css_j, css_j)$$

Par exemple, cette propriété permet d'empêcher un processus, i.e. un sujet, qualifié de non sûr, avec un faible niveau d'intégrité, de modifier un objet sûr, avec un fort niveau d'intégrité. Nous utilisons le label $untrusted\_user\_t$ pour le processus ayant un faible niveau d'intégrité, $root$ pour le processus ayant un fort niveau d'intégrité et $shadow\_t$, i.e. `/etc/shadow`, pour l'objet sûr avec un fort niveau d'intégrité. Les plages de niveaux sont décrits dans le tableau 8.2.

La trace T, voir à gauche de la figure 8.1, représente le système et les flux d'information présents dans cette trace sont représentés dans la figure 8.1 à droite. La première interaction, $untrusted\_user\_t \underset{file:write}{\overset{T}{\rightarrow}} [5874, 5889]\ shadow\_t$, peut être représentée sous la forme du flux d'information $untrusted\_user\_t \underset{[5874,5889]}{\overset{T}{\rhd_t}} shadow\_t$. La deuxième régle de Biba s'applique à ce flux i.e. un sujet écrit dans un objet. La borne inférieure de la plage de niveau du sujet doit être

supérieur ou égal à la borne supérieur de celle de l'objet, hors, $0 \leq 7$. La régle n'est donc pas respectée et l'interaction estampillée correspond à une rupture de la propriété. En conséquence, dans le cas de la protection, l'interaction est annulée. Le graphe représentant la trace est donc modifié pour devenir celui représenté dans la figure 8.2.

La seconde interaction, $untrusted\_user\_t \underset{process:transition}{\overset{T}{\rightharpoonup}} [6025, 6041]\ root\_t$, peut être représentée sous la forme d'une transition $untrusted\_user\_t \underset{[6025,6041]}{\overset{T}{\triangleright_t}} root\_t$. La troisième régle de Biba s'applique à cette transition i.e. un sujet invoque un second sujet. La borne inférieure de la plage de niveau du sujet appellant doit être supérieur ou égale à la borne supérieur de la plage de celui qu'il invoque, hors, $0 \leq 13$. La régle n'est donc pas respectée et l'interaction estampillée correspond à une rupture de la propriété. En conséquence, dans le cas de la protection, l'interaction est annulée. Le graphe représentant la trace est donc modifié pour devenir celui représentée dans la figure 8.3.

La troisième interaction, $untrusted\_user\_t \underset{file:read}{\overset{T}{\rightharpoonup}} [6147, 6159]\ shadow\_t$, peut être représentée sous la forme du flux d'information $shadow\_t \underset{[6147,6159]}{\overset{T}{\triangleright_t}} untrusted\_user\_t$. La première régle de Biba s'applique à ce flux i.e. un sujet lit un objet. La propriété est respectée si la borne supérieure de la plage de niveau du sujet est inférieure ou égale à la borne inférieure de celle de l'objet. Comme $5 \leq 6$, la régle est respectée et l'interaction estampillée est légitime et autorisée.

entity	**Plage de Niveaux d'Integrité**
untrusted_user_t	[0 - 5]
`/etc/shadow`	[6 - 7]
root	[7 - 13]

TABLE 8.2 – Tableau des plages de niveaux d'intégrité

8.1.3 Bell&LaPadula

Le modèle de sécurité appellé Bell&LaPadula (BLP) a pour but de garantir la confidentialité d'un système en respectant deux régles de contrôle des opérations élémentaires. Chaque entité du système se voit attribuer un niveau (tout comme dans Biba). Pour les contextes sujets, ce niveau est appellé niveau d'habilitation et pour les contextes objets, un niveau de sensibilité. Les deux régles qui régissent BLP sont les suivantes :

blp-1 Pourqu'un sujet puisse accéder en lecture à un objet, son niveau d'habilitation doit être supérieur ou égal au niveau de sensibilité de l'objet. Cette règle permet d'assurer la confidentialité de l'information.

blp-2 Si un flux d'information allant d'un objet vers un second objet apparait dans le système, il est uniquement autorisé si le niveau de sensibilité du contexte cible est supérieur ou égal à celui du contexte source. En pratique, les écritures "vers le haut" sont les seules autorisées d'un point de vue des niveau de sensibilité. Cette règle permet d'assurer la non divulgation d'information.

Pour que la propriété soit respectée, il faut que ces deux régles le soient. Tout comme Biba, nous définissons une fonction $hab : \mathcal{CS} \rightarrow \mathbb{N}$ qui associe un contexte à son niveau d'habilitation/-sensibilité.

Nous utilisons les régles relatives à BLP ainsi que la fonction $hab()$ pour modéliser la propriété. La définition 8.1.9 définit la régle **blp-1** dans notre formalisme. La régle **blp-1** est respectée si

quand un sujet css_1 accéde en lecture à un objet cso, son niveau d'habilitation $hab(css_1)$ est supérieur ou égal au niveau de sensibilité de l'objet $hab(cso)$.

Définition 8.1.9 (Régle BLP-1) *Un objet cso est dit confidentiel vis-à-vis d'un sujet css_1 suivant blp-1 pour une trace T ssi il n'existe que des flux d'information directs généraux de cso vers css_1 avec $css_1 \in \mathcal{CSS}$, $cso \in \mathcal{CSO}$ et $hab(css_1) \geqslant hab(cso)$.*

$$BLP1(T, css_1, cso) \stackrel{def}{\equiv} (css_1 \overset{T}{\rhd} cso) \wedge (hab(css_1) \geqslant hab(cso)) \tag{8.7}$$

La définition 8.1.10 modélise la régle *blp-2*. La régle **blp-2** est respectée si quand un flux d'information entre un objet source cso_1 et un objet cible cso_2 apparait sur le système, le niveau de sensibilité du contexte cible $hab(cso_2)$ est supérieur ou égal à celui du contexte source $hab(cso_1)$.

Définition 8.1.10 (Régle BLP-2) *Un objet cso_1 est dit confidentiel vis-à-vis d'un second objet cso_2 suivant BLP-2 pour une trace T ssi il n'existe que des flux d'information généraux de cso_1 vers cso_2 avec $cso_1, cso_2 \in \mathcal{CSO}$ et $hab(cso_2) \geqslant hab(cso_1)$.*

$$BLP2(T, cso_1, cso_2) \stackrel{def}{\equiv} (cso_1 \overset{T}{\rhd\!\!\rhd} cso_2) \wedge (hab(cso_2) \geqslant hab(cso_1)) \tag{8.8}$$

La propriété de confidentialité dite BLP est respectée si et seulement si les deux définitions 8.1.9 et 8.1.10 sont respectées. Nous formalisons avec la propriété 8.1.1 le respect de BLP sur un système.

Propriété 8.1.1 (Bell&LaPadula) *Soit $S1 = (css_1, cso_1), ..., (css_m, cso_m)$ et $S2 = (cso_1, cso_2), ..., (cso_n, cso_m)$, deux ensemble de couples de contextes. Un **système**, représenté par une trace T, est dit **confidentiel** selon* BLP *si et seulement si la définition 8.1.9 est respectée pour tous les couples de contextes du système. (css_i, cso_i) avec $css_i \in \mathcal{CSS}$ et $cso_i \in \mathcal{CSO}$ et la définition 8.1.10 est respectée pour tous les couples de contextes (cso_j, cso_{j+1}) avec $cso_j, cso_{j+1} \in \mathcal{CSO}$.*

$$BLP(T, S1, S2) \stackrel{def}{\equiv} \forall i = 1..m, BLP1(T, css_i, cso_i), \forall j = 1..m, BLP2(T, cso_j, cso_{j+1})$$

8.1.4 Bell&LaPadula Restrictif

Le modèle de confidentialité dit BLP restrictif (BLPR) permet de garantir la confidentialité du système via le respect de trois règles de contrôle sur les opérations élémentaires. Ces trois règles sont les suivantes :

blpr-1 Un sujet peut accéder en lecture à un objet si et seulement si son niveau d'habilitation est supérieur ou égal au niveau de sensibilité de l'objet.

blpr-2 Un sujet peut ajouter des données (i.e. modification sans lecture) à un objet si et seulement si son niveau d'habilitation est inférieur ou égal au niveau de sensibilité de l'objet.

blpr-3 Un sujet peut modifier un objet si et seulement si son niveau d'habilitation est supérieur ou égal au niveau de sensibilité de l'objet.

Les trois régles doivent être respectées pour que la confidentialité suivant BLPR soit respectée sur le système. Nous réutilisons la fonction introduite pour la propriété BLP, i.e. $hab : \mathcal{CS} \rightarrow \mathbb{N}$, qui associe un contexte à son niveau d'habilitation/sensibilité.

Nous modélisons les trois régles dans notre formalisme ce qui nous permet avec la fonction $hab()$ d'exprimer la propriété BLPR.

La définition 8.1.11 translate la régle **blpr-1**. La régle **blpr-1** est respectée si quand un sujet css_1 accéde en lecture à un objet cso, son niveau d'habilitation $hab(css_1)$ est supérieur ou égal au niveau de sensibilité de l'objet $hab(cso)$.

Définition 8.1.11 (Régle BLPR-1) *Un objet cso est dit confidentiel vis-à-vis d'un sujet css_1 suivant blpr-1 pour une trace T ssi il n'existe que des flux d'information directs généraux de cso vers css_1 avec $css_1 \in \mathcal{CSS}$, $cso \in \mathcal{CSO}$ et $hab(css_1) \geqslant hab(cso)$.*

$$BLPR1(T, css_1, cso) \stackrel{def}{\equiv} (cso \overset{T}{\triangleright} css_1) \wedge (hab(css_1) \geqslant hab(cso)) \tag{8.9}$$

La définition 8.1.12 modélise la régle **blpr-2**. La régle **blpr-2** est respectée si quand un sujet css_1 ajoute des données (i.e. modification sans lecture) à un objet cso, son niveau d'habilitation $hab(css_1)$ est inférieur ou égal au niveau de sensibilité de l'objet $hab(cso)$.

Définition 8.1.12 (Régle BLPR-2) *Un objet cso est dit confidentiel vis-à-vis d'un sujet css_1 suivant blpr-2 pour une trace T ssi toutes les opérations élémentaires eo de type ajout de données ont pour source css_1 et pour destination cso avec $css_1 \in \mathcal{CSS}$, $cso \in \mathcal{CSO}$, $eo \in \mathcal{AEO}$ et $hab(css_1) \geqslant hab(cso)$.*

$$BLPR2(T, css_1, cso) \stackrel{def}{\equiv} (css_1 \underset{eo}{\overset{T}{\rightharpoonup}} cso) \wedge (hab(css_1) \leqslant hab(cso)) \wedge (eo \in \mathcal{AEO}) \tag{8.10}$$

La définition 8.1.13 modélise la régle **blpr-3**. La régle **blpr-3** est respectée si quand un sujet css_1 modifie un objet cso, son niveau d'habilitation $hab(css_1)$ est supérieur ou égal au niveau de sensibilité de l'objet $hab(cso)$.

Définition 8.1.13 (Régle BLPR-3) *Un objet cso est dit confidentiel vis-à-vis d'un sujet css_1 suivant blpr-2 pour une trace T ssi il n'existe que des flux d'information directs généraux de css_1 vers cso avec $css_1 \in \mathcal{CSS}$, $cso \in \mathcal{CSO}$ et $hab(css_1) \geqslant hab(cso)$.*

$$BLPR3(T, css_1, cso) \stackrel{def}{\equiv} (css_1 \overset{T}{\triangleright} cso) \wedge (hab(css_1) \geqslant hab(cso)) \tag{8.11}$$

La propriété de confidentialité dite BLPR est respectée si et seulement si les trois définitions 8.1.11, 8.1.12 et 8.1.13 sont respectées. Nous formalisons avec la propriété 8.1.2 le respect de BLP restrictif sur un système.

Propriété 8.1.2 (Bell&LaPadula Restrictif) *Soit $S = (css_1, cso_1), ..., (css_m, cso_m)$, un ensemble de couples de contextes. Un **système**, représenté par une trace T, est dit **confidentiel** selon* BLPR *si et seulement si les définitions 8.1.11, 8.1.12 et 8.1.13 sont respectées pour tous les couples de contextes du système (css_i, cso_i) avec $css_i \in \mathcal{CSS}$ et $cso_i \in \mathcal{CSO}$.*

$$BLPR(T, S) \stackrel{def}{\equiv} \forall i = 1..m, BLPR1(T, css_i, cso_i) \wedge BLPR2(T, css_i, cso_i) \\ \wedge BLPR3(T, css_i, cso_i)$$

entity	Niveau d'Habilitation
general_t	100
top_secret_t	100
capitain_t	40
sensitive_t	35
soldat_t	10
public_t	5

TABLE 8.3 – Tableau de niveau de sensibilité/habilitation

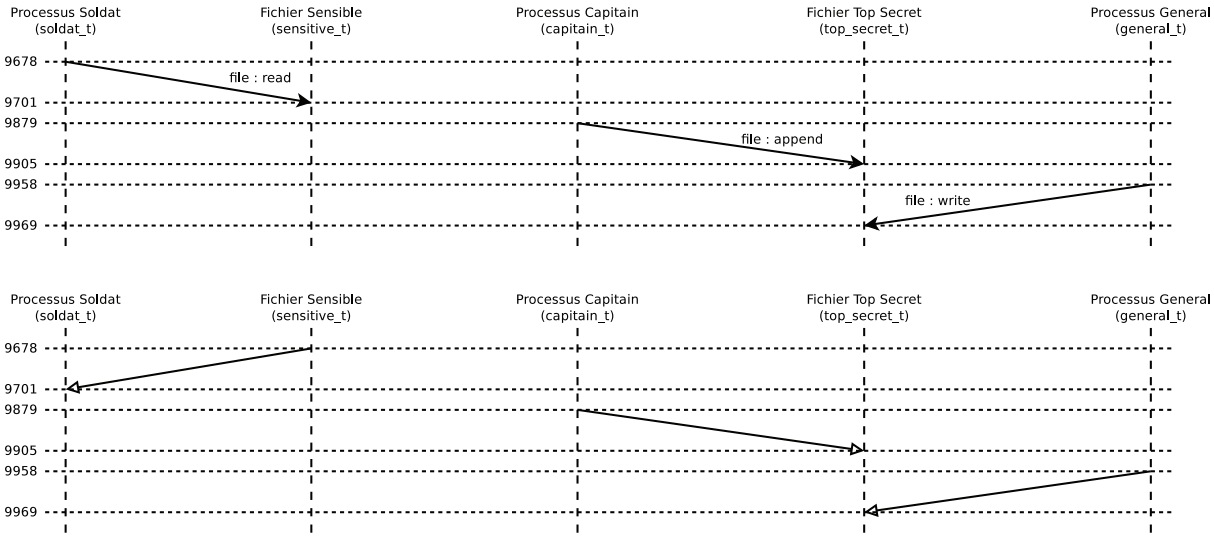

FIGURE 8.4 – Représentation d'une trace système avec rupture de BLPR

Par exemple, nous souhaitons appliquer la propriété de sécurité BLPR sur un système d'exploitation représenté par la trace T, visualisé sous la forme de la figure 8.4 en haut. Cette trace est également représentée sous la forme de flux d'information dans la figure 8.4 en bas. De plus, nous introduisons, dans le tableau 8.3, une matrice de sensibilité / habilitation permettant de connaitre le niveau de chaque contexte sur le système.

La première interaction, $soldat\_t \underset{file:read}{\overset{T}{\leftarrow}} [9687, 9701]\ sensitive\_t$, est également le flux d'information $sensitive\_t \underset{[9687,9701]}{\overset{T}{\rhd}} soldat\_t$. Cette interaction correspond à une lecture et c'est donc la première régle de BLPR qui s'applique. Dans ce cas, le niveau d'habilitation du sujet doit être supérieur ou égal à celui de l'objet. En pratique, le niveau d'habilitation de $soldat\_t$ doit être supérieur ou égal à celui de $sensitive\_t$. Hors $10 \leq 35$, la régle n'est donc pas respectée et l'interaction correspond à une action interdite. Elle est donc détectée et, dans le cas de la protection, annulée. Le graphe représentant la trace se voit donc modifier comme celui visible dans la figure 8.5. Ici, nous démontrons qu'un utilisateur, ayant un niveau de sensibilité faible, ne peut pas lire des données confidentielles.

La deuxième interaction de la trace, $capitain\_t \underset{file:append}{\overset{T}{\leftarrow}} [9879, 9905] top\_secret\_t$ correspond à un ajout de donnée dans un fichier, c'est donc la deuxième régle de BLPR qui s'applique. Dans ce cas, le niveau d'habilitation du sujet doit être inférieur ou égal à celui de l'objet. En pratique, le niveau d'habilitation de $capitain\_t$ doit être inférieur ou égal à celui de $sensitive\_t$. Hors, $40 \leq 35$, la régle n'est donc pas respectée et l'interaction correspond à une action interdite. Elle est donc détectée et, dans le cas de la protection, annulée. Le graphe représentant la trace se voit donc modifier comme celui visible dans la figure 8.6. Dans ce cas, nous avons montré qu'un utilisateur ne peut pas ajouter des données dans un fichier ayant un niveau inférieur à lui ce qui correspondrait à une déclassification d'information sauvage.

La troisième interaction de la trace, $general\_t \underset{file:write}{\overset{T}{\leftarrow}} [9958, 9996]\ top\_secret\_t$ peut être représentée sous la forme d'un flux d'information $general\_t \underset{[9958,9996]}{\overset{T}{\rhd}} top\_secret\_t$. Cette interaction correspond à un écriture de donnée dans un fichier, c'est donc la troisième régle de BLPR qui s'applique. Dans ce cas, le niveau d'habilitation du sujet doit être supérieur ou égale à celui

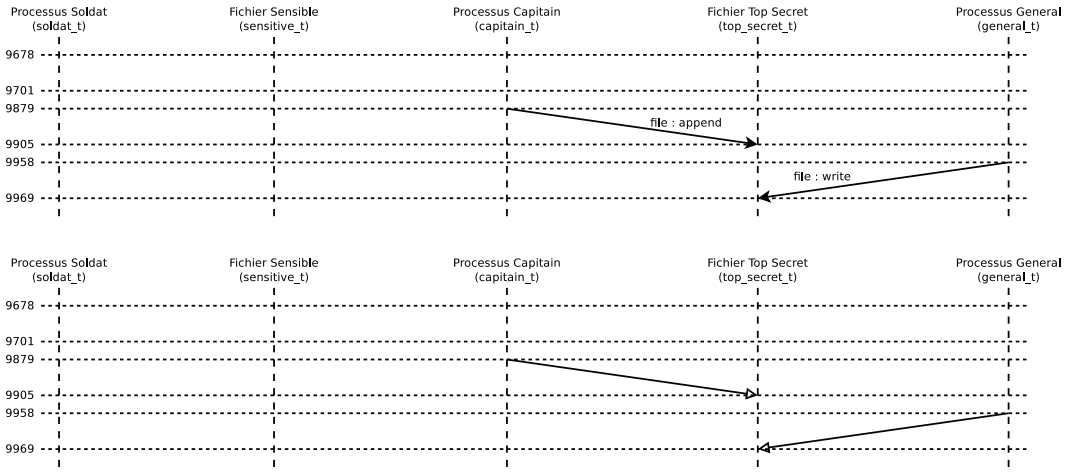

FIGURE 8.5 – Représentation d'une trace système sans rupture de la 1ère régle de BLPR

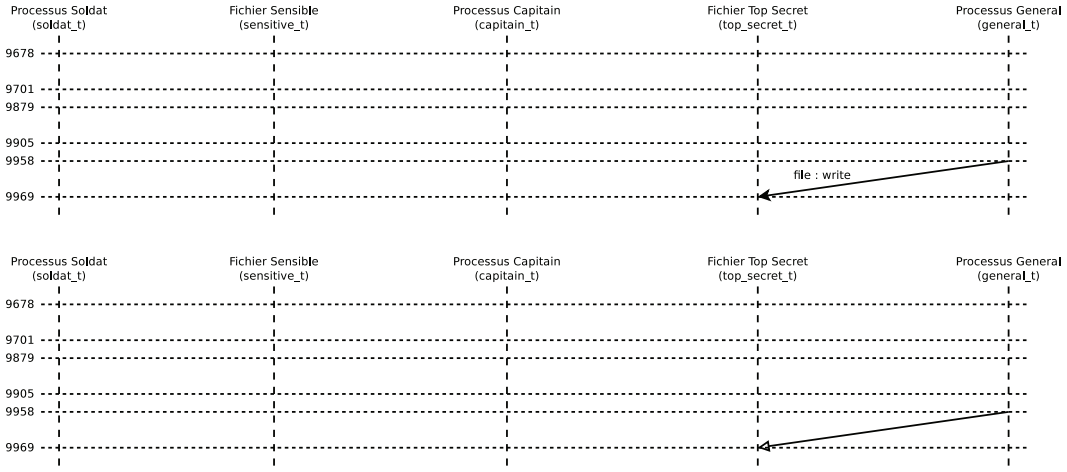

FIGURE 8.6 – Représentation d'une trace système sans rupture de la 1ère et 2ème régle de BLPR

de l'objet. En pratique, le niveau d'habilitation de $general\_t$ doit être inférieur ou égal à celui de top_secret_t. Comme $100 \geq 100$, la régle est respectée et la propriété l'est donc également aussi. L'interaction est donc autorisée dans le cadre de la protection. En effet, un utilisateur ayant un niveau fort d'habilitation peut écrire des données ayant elle aussi un fort niveau d'habilitation. C'est écrire ces données sensibles dans des objets ayant un faible niveau d'habilitation qui est interdit car elle est assimillée à de la déclassification d'information sauvage.

Jonathan Rouzaud-Cornabas

Formalisation de propriétés de sécurité pour la protection des systèmes d'exploitation

Résumé :
Cette thèse traite du problème d'une protection en profondeur qui puisse être assurée par un système d'exploitation. Elle établit la faiblesse des solutions existantes pour l'expression des besoins de sécurité. Les approches supportent en général une seule propriété de sécurité.
Nous proposons donc un langage qui permet de formaliser un large ensemble de propriétés de sécurité. Ce langage exprime les activités système directes et transitives. Il permet de formaliser la majorité des propriétés de confidentialité et d'intégrité de la littérature. Il est adapté à l'expression de modèles de protection dynamique. A titre d'exemple, un nouveau modèle dynamique est proposé pour la protection des différents domaines d'usage d'un navigateur Web.
Nous définissons une méthode de compilation du langage pour analyser les appels systèmes réalisés par les processus utilisateurs. La compilation repose sur la construction et l'analyse d'un graphe de flux d'information. Nous montrons qu'en pratique la complexité reste faible.
Une implantation de ce langage est proposée sous la forme d'un contrôle d'accès mandataire dynamique pour Linux. Une expérimentation à large échelle a été réalisée sur des pots-de-miel à haute interaction. Notre protection a montré son efficacité aussi bien pour les serveurs que les postes client. Il présente des perspectives intéressantes aussi bien pour la protection des systèmes que pour l'analyse de vulnérabilités.
Ce travail a contribué au projet SPACLik vainqueur du défi sécurité de l'ANR SEC&SI.
Mots clés : sécurité, système d'exploitation, politique de sécurité, contrôle d'accès, protection, mandataire, propriétés de sécurité

Security properties formalization for operating system protection

Résumé :
The subject of this thesis is to propose an in-depth protection that can be enforced by the operating system. First, we present that current security solutions are weak in the expression of security. Indeed, most of them support only one security properties.
We introduce a language that allows to formalize a large set of security properties. This language expresses directs and transitives system activities. It allows to formalize the majority of integrity and confidentiality security properties introduced in the litterature. Moreover, the language can also expresses dynamic security properties. We introduces a new dynamic security model for the protection of multiple security domains managed by a web browser.
We define a method to compil our language. The purpose is to analyze the system call done by the users processes. The compilation process build and analyze an information flow graph. Futhermore, we show that the complexity of our protection solution is low.
We propose an implementation of this language as a dynamic mandatory access control for Linux. We experiment it on large scale high interaction honeypots. Our protection shows its efficiency both for clients and servers. Moreover, it presents interesting perspectives for the protection of other systems and for the vulnerability analysis.
This work has contributed to the SPACLik project that wins the security contest of the French National Research Agency : ANR SEC&SI.
Keywords : security, operating system, security policy, access control, protection, mandatory, security properties

LIFO, 88bld Lahitolle, 18020 BOURGES Cedex