Table des matières

Table des figures

Liste des tableaux

Introduction

Les systèmes d'informations sont maintenant accessibles à la plupart des individus et organismes (économiques, sociaux, gouvernementaux, etc.) de notre société. Ceux-ci utilisent largement les possibilités offertes par les systèmes de traitement de l'information et par l'interconnexion de ces systèmes au travers de réseaux informatiques permettant l'échange instantané de données à travers le monde. L'importance prise par les différents outils permettant l'échange et le traitement de l'information engendre une relative dépendance des individus et des organisations à l'égard de ces technologies. Aussi, la défaillance de ces systèmes peut avoir des conséquences graves au niveau économique, sanitaire, social, militaire, etc.

Ces défaillances peuvent être accidentelles ou provoquées par les actions d'individus malveillants. La sécurité informatique s'intéresse à ce second type de défaillances en définissant les méthodes et les outils permettant d'assurer la protection des systèmes et des réseaux d'informations contre les actions de tels individus.

La première étape de sécurisation des systèmes consiste à définir une politique de sécurité. Celle-ci est constituée de règles permettant de s'assurer que des propriétés de sécurité s'appliquent sur les données du système étudié. D'après la définition des critères ITSEC [ITS91] reprise par Yves Deswarte [DM02], ces propriétés sont :

- la confidentialité : les informations ne peuvent être révélées à des utilisateurs non autorisés à les connaître ;
- l'intégrité : les informations ne peuvent être modifiées par des utilisateurs non autorisés ;
- la disponibilité : les informations doivent être accessibles à tout utilisateur autorisé. Cette propriété s'applique également aux éléments du système et du à ceux du réseau qui offrent un service aux utilisateurs du système.

La seconde étape consiste à mettre en œuvre cette politique de sécurité au sein des systèmes. Deux types d'approches, non exclusives, peuvent être envisagées :

- les approches préventives mettent en œuvre des mesures organisationnelles et techniques permettant de s'assurer que les propriétés de la politique seront effectivement vérifiées. Il s'agit par exemple de restreindre l'accès des utilisateurs préalablement authentifiés, de chiffrer les données confidentielles, de vérifier l'absence d'erreur dans les logiciels, etc.

– les approches d'audit et de détection cherchent à déterminer les occurences des intrusions, c'est-à-dire les violations des propriétés de confidentialité, d'intégrité ou de disponibilité définies par la politique de sécurité. Il s'agit donc d'approches *a posteriori*.

Diverses approches préventives sont généralement employées pour la protection des données d'un système, permettant de s'assurer *a priori* du respect de la politique de sécurité. Cependant, l'expérience montre que ces approches sont souvent insuffisantes. Il est donc nécessaire de recourir également à des approches d'audit et de détection.

L'action des utilisateurs malveillants dont le but est de provoquer la défaillance d'un système est définie par le terme d'attaque. Une attaque réussie se traduit par une intrusion qui constitue une violation de la politique de sécurité. Ces intrusions sont possibles en raison de défauts présents sur les systèmes et appelés vulnérabilités. Celles-ci peuvent apparaître à différents moments du cycle de vie d'un système : lors de la conception, lors de l'implémentation, lors du déploiement et de la configuration ou lors de l'exploitation du système.

Nous nous intéressons dans ce travail uniquement aux aspects de confidentialité et d'intégrité des politiques de sécurité, la disponibilité étant considéré en dehors du périmètre de cette étude. La vérification de ces propriétés peut s'avérer relativement complexe. En effet, il n'est pas suffisant de garantir que seuls les utilisateurs légitimes aient accès (en lecture ou en écriture) aux différentes informations. Il faut également s'assurer que les utilisateurs légitimes (ou plus exactement les programmes informatiques s'exécutant pour le compte de ces utilisateurs légitimes) ne transmettent pas d'informations protégées à des utilisateurs non-légitimes. Cela nécessite de suivre les flux d'informations au sein des systèmes.

Les approches et les mécanismes utilisés pour la protection des systèmes d'informations dépendent essentiellement du type de système étudié et de la nature des données exploitées. Nous nous intéressons dans cette thèse aux systèmes d'informations utilisant des applications web. Ce type d'application est aujourd'hui très populaire et tend à remplacer, ou du moins à compléter, les applications traditionnelles du poste client. Une des raisons de ce succès réside dans l'interface unifiée utilisée par ces solutions. En effet, de nombreux services peuvent être accédés à l'aide d'un client web. Ce dernier constitue un type d'application légère et répandue. En raison de cette popularité, le nombre d'attaques ciblant ce type de système est en constante augmentation. La mise en œuvre de politiques de confidentialité et d'intégrité au sein de ces systèmes est une tâche complexe, pour les raisons suivantes :

– ces systèmes comprennent plusieurs composants logiciels qui collaborent à différents niveaux. Il est difficile d'assurer la sécurité «de bout-en-bout» car cela suppose de suivre tous les flux d'informations du système.
– la plupart des composants logiciels, en particulier les serveurs et le système d'exploitation, sont des composants utilisés par plusieurs systèmes et «disponibles sur l'étagère»(*Commercial Off-The-Shelf*). La sécurité informatique n'a pas toujours été prise en compte dès la conception de ces composants qui sont donc affectés par des vulnérabilités. Les organisations

qui déploient ce type de système sont souvent impliquées de façon limitée dans le développement de ces COTS. Afin de résoudre les problèmes de sécurité, ces organisations se contentent généralement de mettre en place une gestion des correctifs de sécurité pour ces composants. La réécriture complète ou le développement d'une nouvelle solution *ad-hoc* n'est souvent pas envisageable pour des raisons économiques.
- quelques composants logiciels peuvent être développés spécialement en interne pour des besoins propres, notamment au niveau des applications métiers. Bien que les organisations aient un contrôle total sur le développement de ces composants, ceux-ci, en pratique, se révèlent bien souvent plus vulnérables que les COTS.
- ces systèmes correspondent en général à des applications commerciales. Dans ce contexte, refuser l'accès à un utilisateur légitime peut induire une perte financière plus importante que celle résultant d'attaques des systèmes. Cette exigence limite l'utilisation de systèmes de prévention apportant un haut niveau de sécurité mais qui sont considérés comme trop restrictifs.

De notre point de vue, les attaques contre ce type de système sont facilitées par la complexité du système et par la complexité des flux d'informations engendrés par la collaboration de plusieurs applications. En outre, des attaques ciblent aujourd'hui la logique applicative. Ces attaques se traduisent par des intrusions qui sont caractérisées par des flux d'informations illégaux internes aux applications. Il est donc parfois nécessaire de suivre les flux d'informations de faible granularité, entre les conteneurs d'informations internes des applications (vue locale). De plus, certains scénarios d'attaques font appel à plusieurs applications et génèrent des flux d'informations entre ces applications du système, via des conteneurs d'informations du système d'exploitation. Il est donc également nécessaire de suivre les flux d'informations entre les différentes applications du système (vue globale). Certaines attaques complexes génèrent ces deux types de flux d'informations. Il est donc nécessaire de combiner les deux niveaux de suivi des flux d'informations.

Afin d'assurer la confidentialité et l'intégrité des données sur ce type de système tout en prenant en compte les différentes contraintes évoquées, nous défendons dans cette thèse la position suivante :
- le contrôle de la politique de sécurité doit s'appuyer sur une approche de suivi des flux d'informations. Cette approche doit prendre en compte les différents niveaux de granularité des flux d'informations. Elle doit en particulier traiter à la fois les flux internes aux applications et les flux entre applications.
- le suivi des flux d'informations doit être réalisé dynamiquement à l'aide d'un mécanisme de détection d'intrusions paramétrée par la politique de flux.
- l'implémentation de la solution retenue sur des systèmes réalistes doit être la moins intrusive possible. Elle doit nécessiter un minimum de modifications permettant d'assurer la compatibilité avec les applications COTS existantes. Elle doit également minimiser le surcoût engendré par le suivi

des flux d'informations.

Pour atteindre ces objectifs, nous proposons la démarche suivante :

- nous définissons un modèle formel de détection d'intrusions paramétrée par la politique de sécurité qui repose sur le suivi des flux d'informations. Ce modèle s'inspire des travaux de Jacob Zimmermann [Zim03], présenté au chapitre 1. Par rapport aux travaux de l'auteur, nous proposons un modèle amélioré et nous déclinons l'approche suivant différents niveaux de suivi des flux d'informations qui collaborent. Ce modèle distingue pour cela les conteneurs d'informations de leur contenu. Il modélise l'évolution de l'état du système lors de la réalisation des flux d'informations ainsi que la politique de flux d'informations. Il valide l'algorithme de détection d'intrusions proposé.
- nous définissons une architecture générique permettant d'implémenter le modèle proposé. Afin de prendre en compte les différents niveaux de granularité, cette architecture définit différents niveaux de suivi et de détection correspondant à différentes implémentations qui collaborent entre elles.
- nous proposons une implémentation de cette architecture générique à partir d'un IDS existant, Blare, qui effectue le suivi des flux d'informations entre les applications et d'un nouveau prototype, JBlare, qui assure le suivi des flux d'informations au sein des applications Java.
- nous validons notre approche en réalisant des expérimentations permettant d'évaluer les capacités de détection de nos prototypes ainsi que le surcoût engendré.

Ce mémoire est organisé de la façon suivante : le chapitre 1 présente les travaux existant dans le domaine de la détection d'intrusions et du contrôle des flux d'informations. Nous proposons dans le chapitre 2 notre modèle de détection d'intrusions paramétrée par la politique de sécurité. Nous présentons ensuite au chapitre 3 l'architecture générique retenue ainsi que les implémentations de deux prototypes. Avant de conclure, nous donnons au chapitre 4 une présentation et une analyse des résultats obtenus lors de nos expérimentations.

Chapitre 1

Etat de l'art

Nous présentons ici les travaux antérieurs relatifs à notre approche. Cette thèse propose un modèle et une implémentation d'un mécanisme de sécurité conjuguant deux domaines orthogonaux de la sécurité informatique :

- la détection des intrusions, c'est-à-dire des violations de la politique de sécurité. Ce domaine, présenté en section 1.1, s'intéresse à la mise en œuvre *a posteriori* des différents types de politique de sécurité, en comparant le comportement observé du système avec un modèle spécifié au préalable.
- le contrôle des flux d'informations. Ce domaine, présenté en section 1.2, s'intéresse aux différentes techniques de mise en œuvre d'un type de politique de sécurité : les politiques de flux d'informations.

1.1 Détection d'intrusions

La détection d'intrusions est née, au début des années 80, de la nécessité d'automatiser les tâches d'audit des systèmes informatiques [And80, Den87, Lun88]. En effet, il est parfois possible, pour un utilisateur malveillant, de contourner les mécanismes de prévention et donc de violer la politique de sécurité que mettent en œuvre ces mécanismes. Une telle violation de politique engendre généralement des effets de bord sur le système. Il est donc du ressort de l'administrateur du système d'analyser régulièrement l'état du système et de vérifier qu'il n'a pas été compromis. Cette tâche d'audit suppose un mécanisme d'enregistrement des événements du système au sein de «journaux» et une phase d'analyse des journaux afin d'identifier une éventuelle violation de la politique.

L'objectif de la détection d'intrusions est d'automatiser la tâche d'audit. Il s'agit bien, théoriquement, de détecter de manière automatique les violations de politique de sécurité, qu'on appelle intrusions. Dans la pratique, les outils actuels ne sont cependant pas configurés directement par la politique. Aussi, s'ils détectent certaines intrusions, détectent-ils aussi les tentatives d'intrusions infructueuses, ce qui n'est pas toujours souhaité. En outre, la relative naïveté des algorithmes de détection conduit à un nombre élevé d'alertes, dont une part

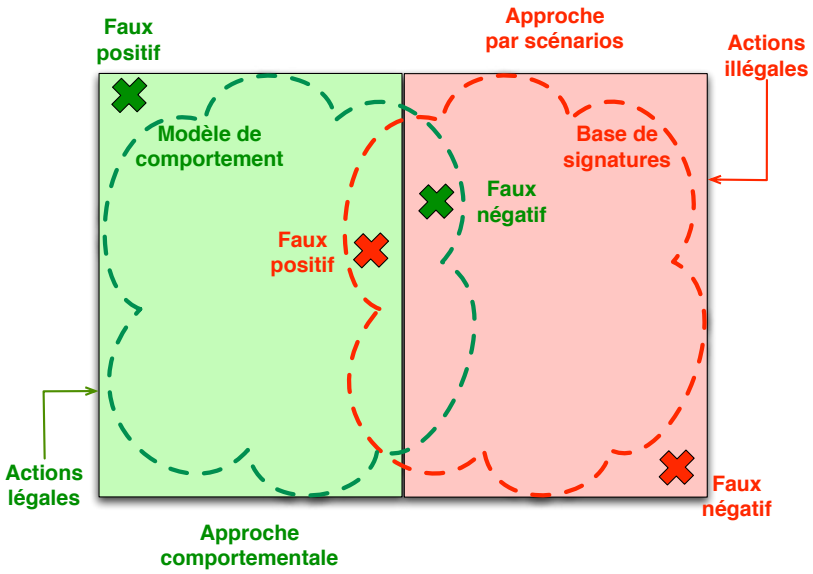

FIG. 1.1 – Erreurs de détection d'un IDS

significative est en fait constituée de fausses alertes ou faux positifs. Enfin, certaines intrusions peuvent ne pas être détectées. On parle alors de faux négatifs. La figure 1.1 illustre la notion de faux positifs et négatifs pour les deux approches de détection d'intrusions qui seront présentées par la suite en section 1.1.1.2.

Afin de qualifier un détecteur d'intrusion, ou IDS pour *Intrusion Detection System*, on s'intéresse à sa fiabilité, qui est sa capacité à émettre une alerte pour toute violation de la politique de sécurité, et à sa pertinence, qui est sa capacité à n'émettre une alerte qu'en cas de violation de la politique de sécurité. Un IDS fiable présente un faible taux de faux négatif (il devrait idéalement être caractérisé par l'absence de faux négatif) ; un IDS pertinent présente un faible taux de faux positif (il devrait idéalement être caractérisé par l'absence de faux positif).

Les IDS actuels ne sont ni fiables ni pertinents. Notre conviction est qu'une des causes essentielles de cet état de fait est l'empirisme qui préside à la conception de ces outils. Notre objectif est donc de proposer une approche formelle, dans laquelle le modèle de détection implanté dans l'IDS présente des capacités que l'on peut prouver.

1.1.1 Architecture classique d'IDS

Nous décrivons dans cette sous-section les trois composants qui constituent classiquement un système de détection d'intrusions [DDW00]. La figure 1.2 illustre les interactions entre ces trois composants. Un capteur est chargé de

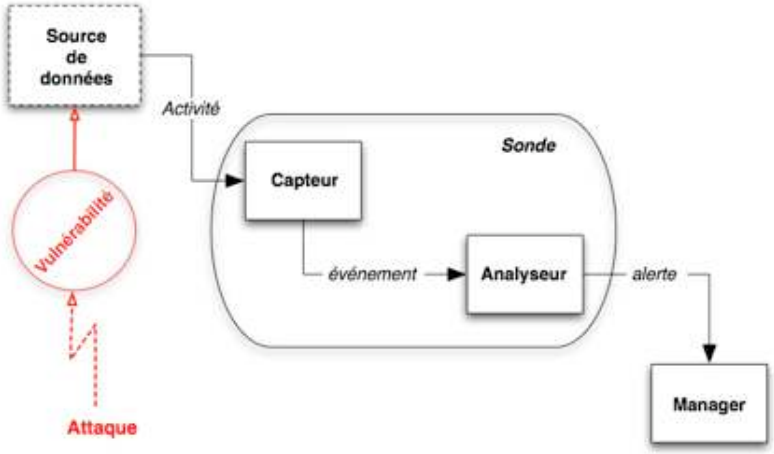

FIG. 1.2 – Architecture classique d'un IDS

collecter des informations sur l'évolution de l'état du système et de fournir une séquence d'événements qui traduit l'évolution de l'état du système. Un analyseur détermine si un sous-ensemble des événements produits par le capteur est caractéristique d'une activité malveillante. Un manager collecte les alertes produites par le capteur, les met en forme et les présente à l'opérateur. Éventuellement, le manager est chargé de la réaction à adopter. Nous détaillons par la suite chacun de ces trois composants.

1.1.1.1 Le capteur

Le capteur observe l'activité du système par le biais d'une source de données et fournit à l'analyseur une séquence d'événements qui informe de l'évolution de l'état du système. Le capteur peut se contenter de transmettre directement ces données brutes, mais en général un prétraitement est effectué. Ce traitement permet, par exemple, de filtrer un certain nombre de données considérées comme non pertinentes afin de limiter la quantité d'information à analyser par la suite. De plus, le capteur réalise généralement une mise en forme des données brutes acquises afin de présenter à l'analyseur des données utilisant un certain format d'événements. Ces fonctions sont, par exemple, réalisées par les modules *Preprocessors* et *Decoder* de l'IDS open-source SNORT[1]. On distingue classiquement trois types de capteurs en fonction des sources de données utilisées pour observer l'activité du système :

– Les capteurs système qui collectent des données produites par les systèmes d'exploitation des machines, notamment par le biais des journaux d'audit système ou par celui des appels système invoqués par les applications.

[1] `http://www.snort.org/`

On désigne les IDS utilisant des capteurs système par l'acronyme HIDS (*Host-based IDS*).
- Les capteurs réseau qui collectent les données en écoutant le trafic réseau entre les machines par le biais d'une interface spécifique. On parle alors de NIDS (*Network-based IDS*).
- Les capteurs applicatifs qui collectent les données produites par une application particulière, avec laquelle des utilisateurs sont susceptibles d'interagir, comme un serveur web ou un serveur de base de données. L'application doit alors être instrumentée à cet effet.

L'avantage principal des capteurs réseau réside dans leur capacité à surveiller un grand ensemble de machines. Cette caractéristique simplifie le déploiement et la maintenance d'une solution de détection visant à garantir une couverture optimale du réseau surveillé. L'approche système est plus complexe à déployer car elle nécessite une multiplication du nombre de capteurs dans le réseau. De plus, le coût engendré par la collecte des données par ces capteurs peut dégrader sensiblement les performances des systèmes sur lesquels ils sont installés.

Cependant, on peut s'interroger sur la pérennité des capteurs réseaux pour trois raisons principales. Premièrement, la montée en débit des réseaux contraint fortement les capacités de collecte de l'intégralité du trafic. Les constructeurs de NIDS ont recours à des capteurs matériels spécifiques pour accélérer la collecte, mais la détection d'intrusions dans le cœur de réseau peut poser problème car seules certaines données peuvent être prises en compte. L'inspection de la totalité des paquets n'étant pas envisageable, les IDS pour les réseaux à haut débit doivent échantillonner les données et l'analyse ne porte souvent que sur l'entête et la détection reste imprécise [GSB06]. Deuxièmement, les capteurs réseau ne peuvent analyser le trafic chiffré. Or, la prise en compte progressive des problèmes de sécurité tend à généraliser l'utilisation du chiffrement dans les protocoles réseau, rendant à terme les capteurs réseau inopérants [ACF+00,LaP99]. Enfin, l'analyse seule du trafic réseau s'avère souvent insuffisante pour assurer une détection fiable et pertinente des violations de politique de sécurité, l'IDS ne disposant que de trop peu d'information sur les systèmes attaqués [ACF+00].

1.1.1.2 L'analyseur

L'objectif de l'analyseur est de déterminer si le flux d'événements fourni par le capteur contient des éléments caractéristiques d'une activité malveillante. Deux grandes approches ont été proposées, l'approche comportementale (*anomaly detection*) et l'approche par scénarios (*misuse detection*) :
- Dans l'approche comportementale, une attaque est qualifiée par la mesure d'une déviation sensible du système surveillé par rapport à un comportement de référence, réputé sain et défini auparavant.
- Dans l'approche par signatures, le système de détection possède une base de signatures qui modélisent les différentes attaques connues. L'analyse consiste à rechercher l'occurrence d'un motif caractéristique d'une attaque dans le flux d'événements.

L'approche par signature est actuellement la plus commune. Elle s'appuie sur une base de signatures d'attaques. Le système de détection consiste alors à reconnaître la présence de signatures parmi les traces d'audit fournies par les observateurs. Plusieurs techniques ont été proposées qui reposent en général sur des mécanismes de reconnaissance de motif ou *pattern matching*.

Le *pattern matching* est une méthode simple à mettre en œuvre. Cependant, la difficulté vient de la définition des motifs. En effet, ceux-ci doivent être suffisamment précis pour pouvoir discriminer les différents types d'attaques, mais également suffisamment génériques pour pouvoir détecter les différentes variantes d'un même type d'attaque. Une signature trop générique conduira à l'augmentation du nombre de faux positifs, diminuant ainsi la fiabilité. La technique de détection par scénarios nécessite en outre une maintenance active du système pour mettre à jour régulièrement la base de signatures. En effet, le système ne peut détecter que des attaques connues *a priori* : il faut donc pouvoir réactualiser cette connaissance. Ceci implique notamment un coût de maintenance important. De plus, se pose le problème du choix du langage de signature pour lequel il n'existe pas, pour l'instant, de réel standard, même si l'on peut considérer, dans la pratique, que les signatures « à la SNORT » constituent un standard *de facto*. En théorie, cette approche devrait produire peu de faux positifs car le système possède des connaissances *a priori* sur les attaques.

Nous pensons que les limites inhérentes à l'approche par signature justifient le recours aux approches comportementales présentées plus en détail en section 1.1.2. Notre modèle de détection, présenté en chapitre 2 correspond en fait à une sous-classe des modèles comportementaux. On peut toutefois noter que les deux approches de détection ne sont pas exclusives, car un même *manager* peut être alimenté par des analyseurs implantant les deux approches, puis combiner les résultats à l'aide d'un mécanisme de corrélation d'alertes [MMM+01]. Cependant, cette possibilité n'est guère utilisée dans la pratique.

1.1.1.3 Le *manager*

Le *manager* est responsable de la présentation des alertes à l'opérateur (fonction de console de management). Il peut également réaliser les fonctions de corrélation d'alertes, dans la mesure de leur disponibilité. Enfin, il peut assurer le traitement de l'incident, par exemple au travers des fonctions suivantes :

- confinement de l'attaque, qui a pour but de limiter les effets de l'attaque ;
- éradication de l'attaque, qui tente d'arrêter l'attaque ;
- recouvrement, qui est l'étape de restauration du système dans un état sain ;
- diagnostic, qui est la phase d'identification du problème, de ses causes et qui peut éventuellement être suivi d'actions contre l'attaquant (fonction de réaction).

Du fait du manque de fiabilité des systèmes de détection d'intrusions actuels, les réactions sont rarement automatisées, car elles peuvent se traduire par un déni de service en cas de réaction à des faux positifs.

1.1.2 Approche comportementale

Cette approche repose sur la modélisation d'un «comportement normal» du système ou de l'entité surveillée, appelé profil, et la détection des écarts par rapport à ce profil. Il en découle deux problèmes majeurs :

- la définition du «comportement normal» et la construction du profil de référence ;
- la définition des critères de déviation et la fixation des seuils associés.

Plusieurs méthodes ont été proposées pour construire le profil de référence : on distinguera ici les méthodes utilisant des mécanismes d'apprentissage (présentées en sous-section 1.1.2.1) de celles qui n'en utilisent pas (approche par spécification, présentée dans la sous-section 1.1.2.2, et approche paramétrée par la politique, présentée dans la sous-section 1.1.2.3).

1.1.2.1 Profil généré par apprentissage

Classiquement, la détection d'une anomalie repose sur un modèle statistique du comportement des utilisateurs. Denning a ainsi identifié trois familles de modèles statistiques [Den87] :

- les modèles simples utilisant des seuils sur des variables. Ces variables peuvent correspondre à la fréquence d'apparition d'un événement. Ce modèle simple est parfois utilisé par d'autres composants logiciels comme les mécanismes d'authentification. Ceux-ci considèrent en effet comme «anormal» un nombre d'échecs successifs donné, lors de la phase d'authentification par identifiant et mot de passe ;
- les modèles utilisant les moments statistiques (moyenne, écart-type, etc.). Ces modèles offrent plus de souplesse et permettent de mieux discriminer les comportements anormaux. Cependant, ils reposent sur l'hypothèse que le comportement «normal» d'un utilisateur peut être modélisé par une loi statistique, ce qui n'est pas toujours le cas ;
- les modèles dérivés du modèle de Markov. Les événements ne sont alors plus considérés indépendamment les uns des autres mais en séquence. Le modèle considère en effet les différents états successifs du système. Lorsque le système change d'état, l'IDS vérifie la probabilité d'occurrence de cette transition. Si cette dernière est suffisamment faible, le comportement observé est considéré comme anormal.

Les deux premières catégories permettent d'établir facilement un modèle comportemental. Néanmoins, leur pouvoir de discrimination est relativement limité et les nouveaux modèles comportementaux d'IDS appartiennent généralement à la dernière catégorie. Forrest propose par exemple de s'intéresser aux séquences d'appels système [FHS97]. L'auteur utilise l'exemple du système immunitaire qui est capable chez les êtres vivants de distinguer les corps étrangers des cellules appartenant à l'individu. Bien que les différentes cellules possèdent des caractéristiques variables du fait de la diversité des fonctions assurées, le système immunitaire dispose d'une définition assez précise du «soi», c'est-à-dire de l'ensemble des caractéristiques définissant les cellules appartenant à un même in-

dividu. Un tel système peut efficacement détecter des corps étrangers, même s'il ne les a jamais vus auparavant. L'auteur a établi expérimentalement que des séquences courtes d'appels système (typiquement 6 appels successifs) constituent de bonnes signatures comportementales d'un processus donné. Cette signature varie d'un processus à un autre, mais reste spécifique à chaque type d'application. Une intrusion se traduit par un comportement anormal d'un programme donné qui voit sa signature évoluer sensiblement.

L'auteur a expérimenté cette approche en développant un prototype d'HIDS. La source d'information est constituée des traces d'appels système qui sont systématiquement analysées. L'IDS implémentant cette technique fonctionne alors en deux temps :

- dans un premier temps, l'IDS constitue une base de données d'apprentissage. Les signatures sont obtenues par une technique de fenêtres glissantes ;
- dans un deuxième temps, l'IDS compare les séquences d'appels systèmes observées avec les signatures de la base et comptabilise le nombre de disparités. En dessous d'un certain seuil de concordance, l'IDS considère le comportement du système comme anormal et lève une alerte.

Cette approche constitue un moyen simple pour établir un profil de référence et mettre en œuvre un mécanisme de détection comportemental. Toutefois, les performances de cette technique restent difficiles à prévoir car beaucoup de paramètres sont fixés de manière empirique. Ce problème est d'ailleurs caractéristique des modèles comportementaux qui établissent le profil de référence par apprentissage. En outre, ces modèles considèrent qu'un comportement anormal au sens statistique du terme caractérise une intrusion. Or ce n'est pas toujours le cas et cela conduit à un nombre important de faux positifs [LJ99, ACF+00].

La phase d'apprentissage s'effectue sur des données enregistrées avant la mise en place de l'IDS : se pose alors le problème de l'apprentissage d'éventuelles intrusions. En effet, cette phase requiert une base de données à la fois saine et exhaustive par rapport au comportement attendu des utilisateurs dans l'environnement réel. Pour éviter la mise en place d'un profil trop rigide et pour pouvoir s'adapter aux changements de comportement des utilisateurs, certains IDS proposent des phases de réapprentissage au cours de l'utilisation de l'IDS [LS98]. Il reste tout de même le risque qu'un attaquant arrive à modifier le profil de référence à son avantage par déviation progressive [LB97, WPS06]. De manière générale, fixer des seuils peut s'avérer délicat et les résultats peuvent être pénalisés par un sur-apprentissage.

L'approche comportementale est cependant intéressante car, ne faisant aucune hypothèse sur les comportements illégaux, elle permet en théorie de détecter de nouvelles formes d'intrusions (attaques dites *zero-day*). La définition du profil de référence par apprentissage restant délicate, il a alors été proposé de définir ce profil en excluant toute forme d'apprentissage : le profil de référence est défini au travers d'une spécification des comportements des applications ou au travers de la définition d'une politique de sécurité.

1.1.2.2 Modèle par spécification du comportement des programmes

La spécification de comportement est une perspective qui nous paraît prometteuse. Cette approche, proposée par Ko [KRL97] initialement, s'appuie sur une spécification du comportement attendu du système (en fait de chaque applicatif) et sur l'analyse de traces d'audit au niveau des appels système. Un travail amont d'analyse et de formalisation de la spécification du comportement des applications est donc nécessaire. L'auteur insiste cependant sur le fait que seuls certains programmes dits «sensibles», c'est-à-dire susceptibles de modifier l'état de sûreté du système, doivent être surveillés. Pour l'environnement UNIX, cela se traduit souvent par des programmes exécutés en mode privilégié (SUID).

La spécification du comportement des programmes correspond à l'expression, pour ces programmes, des contraintes imposées par la politique de sécurité. C'est la politique qui permet de déterminer les traces légales de l'application. La mise au point des spécifications se traduit alors par la définition d'un langage formel, plusieurs modèles ayant été proposés. Ko propose ainsi l'utilisation d'une grammaire adaptée aux environnements distribués (*Parallel Environement Grammars*) [KRL97]. Sekar définit un langage de spécifications à base d'expressions rationnelles (*regular expression*) constitué de règles de la forme : (pattern → action) [SCS98]. Chaque action est ainsi réalisée lorsque le motif *pattern* est vérifié. L'ensemble de ces règles permet alors de spécifier le comportement attendu des programmes mais permet également d'exprimer des signatures d'attaques connues en termes de comportement. L'auteur propose en effet de compléter l'approche par spécification avec des méthodes par scénarios d'attaques.

La spécification de comportement constitue une approche relativement récente qui s'appuie explicitement sur la politique de sécurité. Elle permet la détection d'attaques inconnues et propose une bonne couverture tout en générant peu de faux positifs [US01]. Cette approche, encore jeune, n'a donné lieu qu'à peu d'implémentations dans des outils de détection. Deux implémentations ont été proposées par Ko et Sekar pour valider leurs modèles. Plus récemment, la société NOVELL a proposé Apparmor[2], un système de détection/prévention utilisant un modèle comportemental à base de spécifications du comportement des applications.

Le principal inconvénient de cette approche est qu'elle nécessite un effort lors de la mise en place des spécifications, et ceci pour chaque application à surveiller. Une autre approche a été proposée par Ko [KR02], dans laquelle la politique n'est plus exprimée en termes de spécification explicite pour chaque application : on parle alors d'approche paramétrée par la politique de sécurité (*policy-based IDS*). Nous la présentons dans la section suivante.

1.1.2.3 Approche paramétrée par la politique de sécurité

Un modèle d'IDS paramétré par la politique de sécurité repose sur :
– un modèle du système surveillé ;

[2] `http://www.novell.com/linux/security/apparmor/`

– un modèle de la politique de sécurité en vigueur sur ce système ;
– un modèle de détection, c'est-à-dire de l'algorithme de détection.

Les travaux du domaine cherche à prouver, via l'utilisation de méthodes formelles, que le comportement de l'IDS est cohérent vis-à-vis de la modélisation du système et de la politique. Le premier modèle d'IDS paramétré par la politique de sécurité a été proposé par Ko [KR02, KR03]. Celui-ci s'est intéressé aux problèmes de violations d'une politique d'intégrité par les attaques de type *race-condition*, littéralement «situation de compétition». Cette forme d'attaques correspond en fait à un problème particulier de synchronisation : l'attaquant profite de l'exécution concurrente et non synchronisée d'opérations dont l'une au moins est «sensible» (typiquement une vérification de contrôle d'accès sur une ressource «sensible» suivi d'un accès à cette même ressource) pour lire ou modifier des données auxquelles il n'a pas accès. Ko *et al.* modélisent le système par une machine à état et la politique de sécurité par une propriété de «non-interférence». Ils montrent ensuite que la détection en temps réel des violations de cette politique peut être réalisée par un algorithme prouvé par un théorème de déroulement. Cet algorithme a été implémenté au sein d'un HIDS sous Linux. Les expérimentations réalisées ont montré que l'IDS est alors capable de détecter les violations de politique d'intégrité résultant d'une attaque de type *race-condition*.

Ces travaux sont intéressants et sont parmi les premiers à proposer un modèle théorique prouvé de détection. L'avantage par rapport à la spécification de comportement réside dans le fait qu'il n'est pas nécessaire de spécifier un profil par application, la politique étant spécifiée globalement pour le système. Le principal inconvénient est que les hypothèses de ce modèle sont assez restrictives. Le spectre des attaques prises en compte par le modèle est donc assez restreint : seules les attaques violant les politiques d'intégrité par *race-condition* sont détectées.

Zimmermann a proposé un modèle plus générique de détection paramétrée par la politique : le modèle à flux de références [ZMB02, ZMB03, Zim03]. Ce modèle permet de couvrir un spectre d'attaques plus large que le modèle proposé par Ko et Redmond. L'approche de Zimmermann permet en effet de traiter les violations de l'intégrité et de la confidentialité, et ce quel que soit le scénario d'attaque utilisé. L'auteur désigne par «attaques par délégation» les attaques générant un flux d'informations observable et qui violent une politique de sécurité préalablement spécifiée. La notion de «délégation» renvoie au fait que l'attaquant ne peut réaliser directement l'intrusion (le contrôle d'accès l'en empêchant) et qu'il doit par conséquent «déléguer» une partie des opérations pour générer le flux d'informations interdit. Cette délégation peut s'effectuer de plusieurs manières : exploitation de failles de sécurité dans un programme privilégié, de *race-condition*, etc.

L'auteur propose un modèle théorique du processus de détection ainsi qu'une implémentation au niveau du système d'exploitation. La solution proposée constitue un HIDS pour le noyau Linux appelé Blare. Le modèle, et en particulier la preuve de l'algorithme de détection, reste toutefois incomplet. Nous proposons dans cette thèse un nouveau modèle qui reprend et étend celui proposé par

Zimmermann. Nous établissons la preuve de la cohérence de l'algorithme de détection au regard de la politique de flux spécifiée à l'aide d'un théorème de détection. Une nouvelle implémentation de Blare, présentée dans le chapitre 3, a été réalisée en collaboration avec Zimmermann. Cette nouvelle implémentation respecte le nouveau modèle proposé, minimise la consommation de ressources et permet une collaboration avec une sonde applicative nouvellement développée, également présentée dans le chapitre 3.

L'implémentation initiale réalisée par Zimmermann a été validée par des expérimentations sur un environnement Linux significatif d'une utilisation «serveur» [ZMB03]. Plusieurs vulnérabilités ont été exploitées sur différents services Linux :

- des dépassements de tampons (*buffer-overflow*) et des attaques par traversée illégale de répertoires (*directory traversal attack*) dans un serveur web spécifiquement développé pour des évaluations de sécurité et comportant volontairement de multiples failles ;
- des erreurs de formatage de chaînes de caractères (*format string attack*) dans le serveur de mail Exim et le serveur de fichier NFS ;
- une *race-condition* dans le noyau Linux ;
- etc.

Les résultats de ces expérimentations sont intéressants et démontrent la capacité réelle de Blare à détecter des intrusions résultant de scénarios d'attaques connus ou inconnus lors de la conception et le développement de l'IDS. En effet, l'approche ne s'appuie sur aucune connaissance *a priori* des attaques et ne s'intéresse qu'aux effets de ces attaques en termes de flux d'informations. Cette approche permet donc de détecter un large spectre d'intrusions, à condition que ces dernières vérifient les hypothèses suivantes :

- ces intrusions sont caractérisées par des flux d'informations qui sont observables au niveau du système d'exploitation (plus précisément au niveau de l'interface de médiation que sont les appels système) ;
- le modèle de politique de flux permet de discerner effectivement ces flux ;
- la politique de flux, définie au préalable, considère ces flux comme illégaux.

Comme indiqué précédemment, l'approche ne prend en compte que les effets des attaques en termes d'éventuelles violations d'une politique de flux. Les attaques infructueuses, par exemple les attaques qui sont menées sur un système non vulnérable, ainsi que les attaques qui engendrent des flux par ailleurs autorisés par la politique ne génèrent pas d'alertes. Seules les intrusions, au sens «violations de la politique de flux», sont détectées. Il s'agit donc bien d'une approche de détection d'**intrusions** et non d'une approche de détection d'**attaques**. Par conséquent, le nombre d'alertes émises par Blare reste faible comparé au nombre d'alertes émises par un détecteur de scénarios d'attaques tel que Snort. En pratique, beaucoup d'alertes émises par un détecteur par signatures se révèlent être des faux positifs en termes de détection d'intrusions [Jul01, Axe99, Mor04]. De plus, le modèle ne repose sur aucune donnée empirique ou mécanisme d'apprentissage qui sont inhérents aux approches classiques de détection d'anomalies. Les difficultés rencontrées lors du paramétrage, lors de la fixation des seuils ou lors du processus d'apprentissage, sont ici considérablement amoindries. Le

paramétrage repose en effet uniquement sur la spécification de la politique de flux. Cette politique peut elle-même être spécifiée explicitement et «manuellement» par un administrateur. Elle peut également s'appuyer sur une politique par défaut proposée par une distribution ou utiliser le paramétrage d'un autre dispositif de sécurité tel que le contrôle d'accès. Le modèle de comportement de référence étant directement issu de la spécification de la politique de flux, l'approche offre un taux de faux positifs bien inférieur à celui des approches comportementales «classiques» [ZMB03].

Toutefois, la principale limitation de Blare réside dans la granularité de l'implémentation à la fois en termes de flux observables (et surtout discernables) et en termes de spécification de la politique de flux. En pratique, le comportement interne des applications n'est pas pris en compte. Cela revient à considérer chaque processus comme une «boîte noire» d'un point de vue du système de suivi des flux et donc du mécanisme de détection. Afin de privilégier la fiabilité du système de suivi de flux, les auteurs font l'hypothèse restrictive que toutes les opérations d'accès en écriture d'un processus sont causalement dépendantes des opérations de lecture effectuées auparavant. Autrement dit, les auteurs considèrent un flux élémentaire de tous les conteneurs d'informations lus vers chaque conteneur accédé en écriture. Or il peut s'avérer que, du fait du comportement interne du processus, certaines opérations d'écriture soient décorrélées d'une partie des opérations de lecture. Selon la politique de flux adoptée, cette approximation peut conduire à des faux positifs ou négatifs.

De plus, la notion de flux d'informations est parfois ambiguë. La plupart des travaux portant sur le contrôle de flux, présentés en section 1.2, s'appuient sur la propriété de non-interférence [GM82, GM84] qui garantit qu'un utilisateur non privilégié ne peut inférer aucune information sur les données privées, auxquelles il n'a pas accès, à partir de l'observation des données publiques, auxquelles il a accès. Cette notion permet de garantir l'absence totale de flux d'informations mais se révèle en pratique souvent trop restrictive. Il est parfois nécessaire d'autoriser une certaine forme de «fuite d'information», comme c'est le cas typiquement lors de la vérification d'un mot de passe ou du calcul d'un résumé cryptographique. D'un point de vue pratique, cela revient à considérer des exceptions à la politique de flux. Autoriser de telles exceptions est une décision délicate, le risque étant de ne pas détecter certaines intrusions. En particulier, autoriser une exception pour un processus entier est une décision qui peut être dangereuse. Les exceptions devraient être limitées à des composants logiciels de taille raisonnable, par exemple, une fonction. Cela permet notamment d'envisager un audit du code source afin de garantir que le composant est «sûr». Blare ne gère pas de mécanisme d'exception et, là aussi, le niveau d'implémentation considéré se révèle inapproprié.

Enfin, le niveau de granularité de Blare ne permet pas de prendre en compte un certain nombre d'attaques qui restreignent leur impact au comportement de l'application. Le but de ces attaques n'est pas la prise de contrôle du système. Elles se contentent de modifier la logique applicative afin d'accéder illégalement à de l'information en utilisant des flux d'informations internes à l'application attaquée. Par exemple, les attaque de type *SQL injection* permettent d'accéder

illégalement à de l'information contenue dans un Système de Gestion de Base de Données. Du point de vue de Blare, qui suit les flux d'informations au niveau du système d'exploitation, il est impossible de discerner les flux légaux, résultant d'un accès légitime au SGBD, des flux illégaux résultant d'une attaque de type *SQL injection*.

Cette approche nous paraît très prometteuse malgré ces limitations. Le choix d'une approche comportementale permet de prendre en compte les attaques inconnues lors du déploiement ou du paramétrage de l'IDS (on parle alors d'attaques *zero-day*) ou les attaques polymorphes qu'il est difficile de caractériser à l'aide de signatures. En outre, le paramétrage par la politique permet de limiter le nombre de faux positifs en établissant un profil de référence le plus complet possible. Nous reprenons donc dans cette thèse les travaux de Zimmermann en nous focalisant sur les problèmes suivants :

- la formalisation d'un modèle cohérent de politique de flux et de processus de détection ;
- les problèmes de granularité et de prise en compte des flux internes à l'application.

Nous nous limitons dans notre approche aux politiques de flux d'informations. Nous présentons donc dans la section suivante les travaux antérieurs relatifs au contrôle des flux d'informations.

1.2 Contrôle de flux d'informations

Les politiques de flux d'informations sont des politiques de sécurité qui définissent les propriétés de sécurité en termes de flux d'informations autorisés ou interdits. Une intrusion, c'est-à-dire une violation de la politique de flux, est caractérisée par un ou plusieurs flux d'informations illégaux :

- une violation de la confidentialité est caractérisée par une fuite d'information secrète vers un utilisateur ou une entité publique, qui n'est pas autorisé à accéder à cette information ;
- une violation de l'intégrité est caractérisée par un flux d'informations, généré par un utilisateur ou une entité non privilégié, qui modifie une donnée réputée intègre.

Les différentes méthodes proposées pour contrôler les flux d'informations au sein des programmes informatiques peuvent être séparées en deux catégories :

- les approches **statiques** qui reposent sur l'analyse des propriétés de l'algorithme du programme informatique, sans exécuter le programme ;
- les méthodes **dynamiques** qui reposent sur l'observation, par un moniteur externe ou par une instrumentation du programme, des exécutions du programme surveillé.

1.2.1 Approches statiques

Les méthodes statiques, présentées dans cette sous-section, procèdent à une vérification *a priori* d'une politique de flux, par exemple lors de la compila-

tion. Elle permettent donc de mettre au point et de distribuer des applications réputées «sûres». Nous verrons dans la sous-section suivante qu'il est parfois impossible d'appliquer de telles méthodes à tous les programmes d'un système informatique. Il est alors nécessaire d'utiliser des méthodes dynamiques pour vérifier l'exécution de programmes «non-sûrs».

1.2.1.1 Certification de programme et politique en treillis

Les travaux de Denning [Den76, DD77] sont parmi les premiers à s'intéresser aux flux d'informations dans les programmes informatiques dans le but de garantir des propriétés de sécurité. L'auteur est un des premiers à mentionner le terme de *information flow control*, qui signifie littéralement contrôle de flux d'informations, et qu'il définit comme la technique permettant de réguler la dissémination de l'information.

L'auteur établit tout d'abord la notion de flux d'informations entre conteneurs d'informations. Il considère qu'il existe un flux d'informations d'un conteneur x (par exemple, une variable d'un programme) vers un conteneur y, noté $x \Rightarrow y$, dès lors que l'information initialement présente dans x est transférée vers y ou que les données de y ont été générées à partir de celles de x. La notion de flux d'informations exprime une relation transitive de dépendance entre données. De plus, l'auteur considère deux types de flux : les flux explicites et les flux implicites. Un flux $x \Rightarrow y$ est explicite si l'opération générant ce flux ne dépend pas de la valeur de x. Ce flux permet par exemple de modéliser l'affectation $y := x$ ou $y := x^2$. Un flux $x \Rightarrow y$ est implicite si l'opération générant ce flux est conditionnée par la valeur de x. Typiquement, suivant la valeur de x, l'opération génère un flux explicite $z \Rightarrow y$, z pouvant être une constante. A l'issu de l'exécution de l'opération, la valeur de y dépend donc de la valeur de x et potentiellement de celle de z. Ce type de flux permet de modéliser les structures conditionnelles d'un langage comme dans l'exemple donné dans [DD77] : `y := 1; if x = 0 then y:=0`. La valeur finale de y permet en effet dans ce cas de déterminer si la valeur de x est nulle ou non, d'où un flux d'informations $x \Rightarrow y$.

L'auteur définit ensuite une politique de flux qui spécifie quels sont les flux autorisés. Plutôt que de spécifier tous les flux autorisés entre chaque conteneur d'informations du système, l'auteur propose d'associer une classe de sécurité à chaque conteneur. La classe de sécurité du conteneur x est alors notée $\underline{x}$. Les flux autorisés sont spécifiés à l'aide d'une relation entre classes notée $\rightarrow$. Cette relation définit en fait un ordre partiel entre les différentes classes de sécurité. Le flux d'informations entre les conteneurs x et y, $x \Rightarrow y$ est alors autorisé si et seulement si la relation $\underline{x} \rightarrow \underline{y}$ est vérifiée. L'auteur définit également un opérateur binaire associatif et commutatif de combinaison de classe, noté $\oplus$. Cet opérateur permet de déterminer la classe de sécurité associée aux résultats des fonctions binaires. Par exemple, la classe associée au résultat de l'opération $x + y$ est $\underline{x} \oplus \underline{y}$. L'affectation $z := x + y$ est alors autorisée si et seulement si $\underline{x} \oplus \underline{y} \rightarrow \underline{z}$. Cet opérateur peut être étendu aux opérations d'arité quelconque. La classe de sécurité associée au résultat d'une fonction de n variables $x_1, \ldots, x_n$

est alors notée $\bigoplus(x_1, \ldots, x_n) = x_1 \oplus x_2 \oplus \ldots \oplus x_n$.

Une politique de flux est donc modélisée par un ensemble de classes de sécurité S, une relation entre ces classes définissant les flux autorisés $\rightarrow$ et un opérateur de combinaison de classes $\oplus$. L'auteur considère les cas où $(S, \oplus, \rightarrow)$ forme un treillis ce qui suppose, entre autre, que la relation $\rightarrow$ soit transitive. L'opérateur de combinaison de classe $\oplus$ correspond alors à la borne supérieure des classes combinées. Cette hypothèse de structure en treillis est en général vérifiée et plusieurs travaux la reprennent [San93].

L'auteur propose enfin d'appliquer ce modèle à des systèmes réels afin de garantir la sécurité des flux d'informations au sein de ces systèmes. La principale difficulté réside dans l'identification et le suivi des flux d'informations. Plusieurs solutions sont envisagées par l'auteur :

- des solutions statiques où chaque conteneur d'informations est associé à une classe de sécurité qui ne varie pas ;
- des solutions dynamiques où les classes de sécurité associées aux conteneurs peuvent varier au cours du temps.

Bien qu'il évoque ces deux types de solutions, l'auteur ne propose qu'un mécanisme de vérification statique permettant de certifier les programmes [DD77]. L'idée est de s'assurer dès la conception du programme, ou du moins avant le déploiement de l'application, que tous les flux générés par ce programme vérifient une politique de flux donnée. Cette approche présente plusieurs avantages :

- la vérification étant faite *a priori*, il s'agit d'une méthode préventive qui garantit que toutes les exécutions du programme vérifieront la politique spécifiée ;
- la vérification ne pénalise pas les performances du logiciel puisqu'elle est effectuée au préalable, «hors-ligne» ;
- le suivi des flux d'informations est facilité par l'analyse du code source du programme (il est en particulier possible de suivre les flux d'informations indirects).

Plus précisément, l'auteur s'intéresse à un langage impératif générique, fortement inspiré de Pascal, qui supporte l'assignation de variables, des opérations d'entrée/sortie sur des fichiers, des appels de procédures, des structures conditionnelles (`if...then...else`) et itératives (`while...`), etc. Chaque déclaration de variables ou de constantes est suivie d'une déclaration statique de la classe de sécurité associée à la variable, du type : `i integer security class L`.

Soit une politique de flux spécifiée sous la forme d'un treillis $(S, \oplus, \rightarrow)$, avec S l'ensemble des classes de sécurité associées à chaque variable ou expression, $\rightarrow$ la relation définissant les flux autorisés entre classes et $\oplus$ l'opérateur de combinaison de classes. L'auteur considère que chaque instruction spécifie un flux d'informations. Typiquement, l'instruction d'affectation spécifie un flux d'informations du terme de droite vers le terme de gauche. Dans le cas où le terme de droite est une expression impliquant plusieurs conteneurs d'informations, l'opérateur de combinaison de classe est utilisé pour déterminer la classe du terme de droite, ce qui revient à considérer les différents flux d'informations des variables du terme de droite vers le terme de gauche. Lors des branchements inconditionnels, les flux implicites sont pris en compte à l'aide d'un mécanisme de garde, ce

qui revient à considérer les flux d'informations des conteneurs impliqués dans le test vers chacun des conteneurs modifiés dans la structure conditionnelle.

L'étape de certification consiste à vérifier, par exemple lors de la compilation du programme, que toutes les instructions du programme spécifient des flux qui ne violent pas la politique. Concrètement, du fait de la transitivité de la relation $\rightarrow$, il suffit de vérifier la légalité de chaque flux élémentaire associé à une instruction. La légalité des flux composés, résultant de l'exécution de plusieurs instructions, est alors assurée par transitivité. Si une ou plusieurs instructions spécifient des flux illégaux, c'est-à-dire des flux ne vérifiant pas la relation d'ordre entre les classes associées aux conteneurs source et destination du flux, le programme n'est pas certifié. Les auteurs démontrent par un théorème que, pour tout programme certifié, chaque exécution du programme est telle que pour tout flux d'informations $x \Rightarrow y$, $\underline{x} \rightarrow \underline{y}$ est vérifiée.

Ces travaux précurseurs définissent les principes du contrôle de flux d'informations, principes repris par la plupart des travaux existants dans le domaine. Plusieurs limitations apparaissent cependant :

- l'auteur énonce un principe mais aucune implémentation réelle n'est proposée. En particulier, l'auteur ne présente pas d'expérimentations menées sur le développement d'une application sensible d'un point de vue de la sécurité ;
- la spécification manuelle et statique de classe de sécurité pour toutes les variables d'un programme nous paraît être une hypothèse peu réaliste ;
- l'association statique de classes de sécurité sur les fichiers ne permet pas de prendre en compte les aspects contextuels de la politique de sécurité : une application peut en effet, suivant le contexte d'exécution, manipuler des fichiers de sensibilités différentes ;
- la certification permet de garantir la sécurité de toutes les exécutions d'un programme mais elle peut en revanche rejeter certains programmes dont les exécutions effectives respectent, de par le contexte d'exécution, la politique de flux.

Certaines de ces limitations sont caractéristiques de l'approche statique du contrôle des flux d'informations. D'autres, comme les deux premières, feront l'objet d'amélioration dans les travaux suivants. Ceux-ci peuvent être classés selon deux catégories :

- la première catégorie regroupe des travaux qui se sont attachés à formaliser la notion de flux d'informations ou plutôt la notion d'absence de flux d'informations. Nous présentons ici notamment les travaux de Goguen et Meseguer qui ont défini la notion de non-interférence ;
- la deuxième catégorie regroupe des travaux qui se sont principalement attachés à développer les aspect pratiques d'une telle approche et notamment l'implémentation sur des dérivés de langages utilisés couramment, comme Java.

1.2.1.2 Non-interférence

Les travaux de Denning ne donnent qu'une notion générique des «flux d'informations» : Denning considère qu'il existe un flux d'informations d'un conteneur c_1 vers un conteneur c_2 dès lors que le contenu de c_2 a été généré à partir de celui de c_1. Cette définition «intuitive» des flux d'informations présente un certain nombre de limitations :

- la notion de «dépendance entre contenus» n'est pas précisée explicitement, bien que différents cas soient présentés ;
- la définition ne s'appuie pas sur la notion d'information telle qu'elle est définie dans le domaine de la «théorie de l'information» ou du «traitement du signal» ;

En effet, caractériser l'information produite suppose plusieurs observations du système. Pour un programme informatique traitant des données en entrée et produisant des résultats observables, il convient alors de considérer toutes les traces d'exécutions possibles de ce programme afin de quantifier l'information produite par les observations. Deux approches peuvent être envisagées :

- effectuer plusieurs observations de différentes exécutions du programme ;
- raisonner sur les différentes exécutions possibles à partir du code source du programme.

Les travaux présentés dans cette section reposent sur la deuxième approche.

La notion d'information est étroitement liée à la notion d'observation et donc d'observateur. Plus précisément, c'est l'information que peut inférer un observateur, à partir des observations du système qu'il peut réaliser et de ses connaissances *a priori* sur le système, qui caractérise un flux d'informations. Les travaux de Goguen et Meseguer [GM82, GM84], inspirés de ceux de Feiertag, Levitt et Robinson [FLR77], sont parmi les premiers à s'intéresser aux problèmes d'inférences en sécurité informatique. Les auteurs proposent de définir des politiques de sécurité à partir d'une propriété appelée «non-interférence». Cette propriété définit l'absence de flux d'informations du fait de l'impossibilité pour un observateur d'inférer une information donnée. Plus précisément, les auteurs considèrent que les sources d'informations du système correspondent à des actions d'utilisateurs sur les interfaces d'entrée du système et que les observations correspondent à des actions sur les interfaces de sortie. Dès lors, un groupe d'utilisateurs U_I est considéré comme «non-interférant» avec un second groupe d'utilisateurs U_O si les actions du premier groupe U_I (via l'interface d'entrée du système) n'ont pas d'effet sur les observations du second groupe U_O (réalisées via l'interface de sortie). Par exemple, une politique multi-niveau réduite à deux niveaux, l'un dit *public* et l'autre dit *privé*, spécifie que les actions des utilisateurs privilégiés agissant sur des données *privées* ne doivent par interférer avec les utilisateurs non-privilégiés qui ne sont autorisés qu'à accéder aux données *publiques*. Dès lors, un utilisateur non privilégié ne peut inférer aucune information sur les données *privées* auxquelles il n'a pas accès.

Goguen et Meseguer s'appuient sur un modèle formel du système étudié et de la politique de sécurité. Le système est modélisé par une machine à état

$$M = (S, U, C, Out, s_0, do, out)$$

avec :

- S l'ensemble des états du système ;
- U l'ensemble des utilisateurs, ou sujets, du système ;
- C l'ensemble des commandes accessibles aux utilisateurs (via l'interface d'entrée du système) ;
- Out l'ensemble des observations possibles pour les utilisateurs (via l'interface de sortie du système) ;
- s_0 l'état initial du système ;
- $do : S \times U \times C \to S$ une fonction de transition ou de changement d'état ;
- $out : S \times U \to Out$ une fonction d'observation.

Les systèmes modélisés sont donc supposés déterministes : la succession des états ne dépend que des commandes passées par les utilisateurs et de la fonction do. Le système passe d'un état $s_n \in S$ à un état s_{n+1} suite à l'exécution d'une commande $c_n \in C$ par un utilisateur $u_n \in U$: $s_{n+1} = do(s_n, u_n, c_n)$. Par extension, les auteurs définissent la fonction $sdo : S \times (U \times C)^*$ qui permet d'exprimer l'état atteint après exécution d'une trace de commandes $w = ((u_0, c_0), (u_1, c_1), \ldots, (u_n, c_n)) : s_{n+1} = sdo(s_0, w)$, noté également $s_{n+1} = [[w]]$, l'état initial s_0 étant fixé. Les paires $(u, c), u \in U, c \in C$ forment l'ensemble des entrées du système ou des «sources» des flux d'informations.

La fonction out permet d'exprimer le sous-ensemble de l'état du système observable par chaque utilisateur. L'observation par un utilisateur u de l'état s_n est donc notée $out(s_n, u) = out([[w]], u)$ ou $[[w]]_u$. L'ensemble des observations Out constitue donc les sorties du système ou les «destinations» des flux d'informations.

Les auteurs définissent alors la propriété de non-interférence à l'aide d'une fonction de purge. Pour un groupe d'utilisateurs $G \subset U$ et un ensemble d'actions $A \subset C$, la fonction de purge $p_{G,A} : (U \times C)^* \to (U \times C)^*$ renvoie, pour chaque séquence $w \in (U \times C)^*$ une séquence purgée $w' = p_{G,A}(w)$. La trace purgée w' est constituée de toutes les paires ordonnées $(u, c) \in (U \times C)$ de w sauf celles dont l'utilisateur fait partie du groupe G ($u \in G$) et dont la commande fait partie de A ($c \in A$). La trace purgée contient donc l'ensemble des entrées de w sauf celles réalisées par un utilisateur de G à l'aide d'une commande de A. Soit par exemple $w = ((u_1, c_1), (u_2, c_1), (u_2, c_2), (u_1, c_2))$. Soit $G = \{u_2\}$ et $A = \{c_2\}$, la trace purgée correspondante est alors $w' = p_{G,A}(w) = ((u_1, c_1), (u_2, c_1), (u_1, c_2))$.

Un ensemble d'utilisateurs G_I utilisant des commandes parmi un ensemble A n'interfère pas avec un groupe d'utilisateurs G_O, noté par les auteurs $G_I, A :| G_O$, si et seulement si :

$$\forall w \in (U \times C)^*, \forall u_O \in G_O : [[w]]_{u_O} = [[p_{G_I,A}(w)]]_{u_O}$$

La propriété de non-interférence exprime donc l'impossibilité pour tous les utilisateurs de G_O d'inférer, à partir de leurs observations du système, une quelconque information sur les actions des utilisateurs de G_I. Autrement dit, les effets des actions réalisées par les utilisateurs de G_I sont indiscernables par les utilisateurs de G_O. La non-interférence est donc par nature une propriété de «bout-en-bout» [SRC84] qui assure l'absence de flux d'informations de certaines entrées du système vers certaines sorties.

Les auteurs utilisent les propriétés de non-interférence pour spécifier des politiques de sécurité. Les politiques modélisables de la sorte sont en fait des politiques de flux qui spécifient quels sont les flux d'informations interdits. Les propriétés de non-interférence entre un groupe d'utilisateurs G_I et un groupe d'utilisateurs G_O garantissent en effet que les actions de G_I seront indiscernables par G_O et garantissent donc l'absence de flux d'informations des entrées de G_I vers l'interface de sortie observable par G_O. Une politique de flux est donc un ensemble de propriétés de non-interférence définissant des flux interdits. La plupart des politiques de flux spécifient cependant les flux autorisés, les flux qui ne sont pas déclarés **autorisés** étant considérés comme **interdits**. Ce type de politique peut être modélisé par des exceptions à une propriété générale de non-interférence. Soit $u, A \Rightarrow v$ avec $u, v \in U$ et $A \in C$ une relation définissant les flux autorisés entre utilisateurs ou sujets du système, la politique de flux peut alors s'exprimer de la sorte : $\forall u, v \in U, \forall A \in C\ u, A\ :|\ v$ sauf si $u, A \Rightarrow v$.

Soit par exemple une politique multi-niveau simplifiée comportant trois niveaux : public (P), secret (S) et très secret (TS). Soit $L = \{P, S, TS\}$ l'ensemble des niveaux et $level : U \rightarrow L$ une fonction renvoyant le niveau d'un utilisateur. Soit U_P, U_S et U_{TS} les ensembles modélisant respectivement les utilisateurs de niveau public, secret et très secret :

- $U_P = \{u \in U / level(u) = P\}$
- $U_S = \{u \in U / level(u) = S\}$
- $U_{TS} = \{u \in U / level(u) = TS\}$

La politique de flux peut s'exprimer à l'aide de deux propriétés de non-interférence :

1. U_{TS},C :| $U_P \cup U_S$
2. U_S,C :| U_P

L'approche permet donc de modéliser des politiques de sécurité reposant sur des propriétés de «bout-en-bout» et de haut-niveau. Les propriétés de non-interférence garantissent en effet l'absence de tout flux d'informations entre certaines entrées et sorties du système, et ce quel que soit le chemin effectif «emprunté» par l'information. Aucune restriction n'est imposée sur la capacité d'observation des utilisateurs, aucune hypothèse n'étant faite sur la fonction *out*. La non-interférence permet donc en théorie de prendre en compte tous les canaux d'information, y compris les canaux cachés. Ces derniers désignent des moyens détournés utilisés par un attaquant pour faire transiter de l'information. On distingue généralement deux types de canaux cachés :

- les canaux de stockage (*storage channel*) qui détournent l'utilisation d'un conteneur d'informations du système accessible à un utilisateur externe ;
- les canaux temporels (*time channel*) qui s'appuient sur la modulation dans le temps de l'utilisation d'une ressource du système.

L'utilisation de canaux cachés dans le but de dérober de l'information suppose la collaboration entre un sujet malveillant et un observateur externe. Dans la pratique, il s'agit d'un programme informatique malveillant installé par l'attaquant par un moyen détourné (porte dérobée installée dans la chaîne de distribution de l'application, exploitation d'une vulnérabilité, d'une mauvaise configuration, etc.). Le sujet malveillant et l'attaquant conviennent d'un codage

de l'information et utilisent un moyen accessible publiquement à l'observateur. Dans le cadre des canaux de stockage, il s'agit généralement d'un conteneur d'informations de faible capacité dont la spécification du système n'envisage pas l'utilisation comme moyen de communication. Les drapeaux (*flag*) dans les protocoles réseaux, la valeur de retour d'une fonction (en particulier les codes d'erreurs), etc. peuvent être utilisés. Dans le cadre des canaux temporels, le sujet malveillant module ses accès à une ressource à des intervalles prédéfinis, ces intervalles formant un alphabet de message. L'observateur peut donc, en mesurant les intervalles, retrouver le message. Par exemple, le sujet malveillant peut envoyer des paquets (de connexion, de synchronisation, d'acquittement, etc.) suivant un alphabet à deux intervalles (un intervalle court et un intervalle long) et faire parvenir ainsi à un observateur connecté au réseau un message codé dans l'alphabet binaire.

Si la non-interférence permet d'exprimer facilement des contraintes de sécurité de haut niveau, il est en revanche plus difficile d'établir qu'un système vérifie une propriété de non-interférence. En effet, cette propriété suppose de considérer toutes les traces du système. En pratique, la vérification d'une propriété de non-interférence ne peut être réalisée de manière exhaustive qu'à partir d'une analyse statique. Goguen et Meseguer proposent, afin de vérifier les propriétés de non-interférence sur un système multi-niveaux, une technique de vérification appelée *unwinding*, que l'on peut traduire par «déroulement». Cette technique repose sur un raisonnement par récurrence permettant d'établir, pour chaque action susceptible de faire évoluer le système, un ensemble de conditions nécessaires et suffisantes pour satisfaire la propriété de non-interférence. Il est alors possible de vérifier *a priori*, à partir de la spécification du système, que toutes les actions du système vérifient cet ensemble de conditions et ainsi garantir les propriétés de non-interférence.

Plusieurs travaux [Rus92, HY86] se sont inspirés de ceux de Goguen et Meseguer. Ces travaux se sont notamment attachés à préciser les relations d'équivalence sur les comportements des systèmes et ce afin de rendre les conditions de déroulement les plus explicites possible. Les auteurs établissent notamment des relations d'équivalence entre observations : deux états ou deux traces sont équivalentes pour un observateur si cet observateur ne peut les distinguer (les mêmes entrées observables produisent les mêmes sorties observables). Plusieurs définitions de la relation d'équivalence peuvent être considérées conduisant à différentes formes de propriété de non-interférence. Ainsi, les travaux de Rusby [Rus92] ou de Roscoe [RG99] s'intéressent à la non-interference intransitive. Il s'agit plutôt de politique de flux intransitive donc d'interférences intransitives. Dans une telle politique, la relation d'autorisation des flux n'est pas transitive : il peut exister des cas où les flux directs sont interdits bien que certains flux indirects soient autorisés. Si $\rightarrow$ dénote la relation des flux autorisés et $\nrightarrow$ la relation des flux interdits, le cas suivant peut se produire : $a \rightarrow b$, $b \rightarrow c$ et $a \nrightarrow c$. Ce type de relations intransitives permet notamment de modéliser les politiques faisant appel à la déclassification (*downgrading*) ou les politiques de contrôle de canal (*Channel-Control security policies*). Dans les deux cas, il s'agit d'interdire les flux d'informations entres deux entités sauf si ces flux em-

pruntent un canal de communication sécurisé (par exemple un médium chiffré ou une application dont le code a été audité).

D'autres travaux [McC88, McL94, WJ90] se sont intéressés à généraliser la notion de non-interférence aux modèles de systèmes non-déterministes et à étudier le problème de composition pour différentes propriétés de sécurité inspirées de la non-interférence. Le problème de composition peut être formulé de la façon suivante : soit un système composé de plusieurs modules, la composition pouvant être effectuée suivant différents schémas, si chacun des modules vérifie une propriété de sécurité (par exemple, la non-intérférence), est-ce que le système composé des différents modules vérifie cette propriété ? Certains travaux proposent d'utiliser les algèbres de processus [Ros95, FG95, RS99] ou la théorie des traces [McL92] plutôt que le modèle initial des machines à états.

La non-interférence est *a priori* une propriété intéressante car elle exprime, de manière simple et compréhensible, une exigence de haut niveau essentielle en sécurité informatique et ce sans faire d'hypothèse sur les mécanismes de sécurité utilisés. Cependant, cette formulation élégante de l'absence de flux d'informations présente un certain nombre de limites :

- la première formulation de Goguen et Messeguer a été suivie par de nombreuses autres formulations qui proposent de généraliser le concept aux cas non traités par le modèle initial (non-interférence intransitive, indéterminisme, etc.). Il existe aujourd'hui une grande diversité de définitions de la non-interférence, chaque modèle clamant sa supériorité en termes de généricité ou de précision. Il est difficile de comparer ces différents modèles car ils varient sensiblement [Rya96]. Il n'existe pas aujourd'hui de définition générique et communément adoptée qui reprenne les différents aspects pris en compte par les différents modèles proposés bien que certains travaux se soient attachés à classifier et expliciter ces différents modèles [McL94].
- la non-interférence permet d'exprimer simplement des exigences de haut niveau mais il est en revanche difficile et coûteux de vérifier qu'un système vérifie cette propriété. L'utilisation de théorèmes de déroulement permet de déterminer des conditions ou un ensemble d'équations sur les fonctions de transition du système. Cette formulation moins «intuitive» de la non-interférence facilite la vérification automatisée des propriétés de non-interférence [GM84] mais il reste difficile de vérifier ces conditions sur des systèmes réels, en particulier au niveau de l'implémentation.
- les exigences imposées par la non-interférence, du moins dans sa version «stricte», sont souvent trop fortes. En effet la non-interférence impose l'absence totale de flux d'informations ou plus exactement l'absence totale de variation sur les observations. Dans les systèmes réels, il est impératif d'interdire les flux d'informations permettant de révéler des données secrètes mais il est en revanche souvent nécessaire d'autoriser une certaine forme de fuite d'informations [Zda04]. Plus exactement, il est difficile de garantir l'absence totale de variation sur les données publiques observables. L'utilisation de fonctions de vérification de mot de passe ou de fonctions cryptographiques constituent des exemples classiques de ce cas de figure. Ainsi, lorsqu'un attaquant propose un mot de passe à un système d'au-

thentification, ce dernier, en acceptant ou refusant le mot de passe proposé, renseigne l'attaquant sur le mot de passe réel. Le mot de passe réel, qui constitue la donnée privée à protéger, interfère donc avec l'attaquant car l'observation de l'attaquant dépend du mot de passe stocké. Toutefois, si le mot de passe est judicieusement choisi ou, dans le cas de l'utilisation d'une fonction cryptographique, si la taille de la clef est adaptée, la quantité d'information révélée à chaque essai reste suffisamment faible et empêche l'attaquant de retrouver la donnée secrète, à l'échelle humaine.

Les travaux présentés dans la section suivante se sont intéressés à ces limitations (notamment la seconde). Ils proposent de mettre en oeuvre le contrôle des flux d'informations en vérifiant statiquement des politiques à base de non-interférence sur des programmes informatiques (et non plus sur un système complet), à l'aide d'un mécanisme de typage de sécurité. Ces travaux reposent en effet sur une définition de la non-interférence adaptée aux programmes informatiques : la non-interférence entre variables (et non plus entre utilisateurs). Ils proposent également un mécanisme permettant la vérification automatique du respect d'une politique de sécurité exprimée en termes de propriétés de non-interférence.

1.2.1.3 Contrôle de flux d'informations par système de types

L'utilisation de mécanismes de contrôle de types pour le contrôle statique de flux d'informations est une technique relativement récente qui nous semble particulièrement prometteuse [SM03a, Smi07]. En informatique, la notion de type désigne une catégorie de données vérifiant un ensemble de propriétés. Par extension, le type d'un conteneur d'informations, par exemple, d'une variable, désigne la ou les catégories de données que le conteneur peut recevoir ainsi que les opérations applicables à ce conteneur. Les langages informatiques proposent généralement des types de base (entier, réels, booléens, etc.). A partir de ces types de base, d'autre types peuvent être proposés :

- des types paramétrés (tableaux, pointeurs, etc.) ;
- des types énumérés modélisant les ensembles finis ;
- des types composés (structures, classes, objets, etc.) ;
- etc.

Le contrôle de types consiste alors à vérifier que les valeurs stockées dans les variables ou que les valeurs passées en paramètre aux fonctions respectent les règles de typage. Deux approches peuvent être adoptées :

- le contrôle statique de types associe un type aux conteneurs d'information. Le type doit être spécifié dans le code source mais les mécanismes d'inférence de type permettent de s'affranchir de la spécification explicite par le programmeur. Les vérifications sont effectuées lors de la compilation assurant ainsi une exécution sûre d'un point de vue typage. Le langage objet produit par le compilateur est optimisé et n'a pas besoin de contenir d'informations sur le type des variables, d'où un gain d'occupation mémoire.
- le contrôle dynamique de types associe un type aux contenus (données).

Le type d'une variable est déterminé lors de l'exécution en fonction des données stockées dans cette variable. Le langage objet ou intermédiaire doit contenir, en plus de la valeur, le type de chaque variable ce qui génère un surcoût d'occupation mémoire. De plus, la vérification ayant lieu à l'exécution, elle entraîne une dégradation des performances et peut rendre le débogage des applications plus difficile. Cette approche, plus souple, est souvent adoptée par les langages de script (Python, Php, Ruby, etc.)

Un système de types est constitué de deux parties :

- un environnement de typage, généralement noté Γ, qui associe un type à chaque variable ;
- un ensemble de règles appelées jugements de la forme $\Gamma \vdash p : \tau$, qui permet de déterminer le type d'une assertion du langage.

Certains jugements découlent directement de l'environnement de typage et forment un ensemble d'axiomes. Par exemple, soit un environnement de typage Γ qui associe le type entier, int, au conteneur x. Le jugement suivant découle directement de l'environnement de typage :

$$\Gamma \vdash x : int$$

Ce jugement définit que de Γ, on peut inférer que x est de type int. D'autres jugements peuvent être déduits par des règles d'inférence, à partir de jugements préalablement établis. Par exemple, soit un environnement de typage Γ qui associe le type entier, int, aux conteneurs x et y. La règle d'inférence suivante permet de déduire le type correspondant à l'addition de x et y :

$$\frac{\Gamma \vdash x : int \qquad \Gamma \vdash y : int}{\Gamma \vdash x + y : int}$$

Ces règles se lisent de haut en bas. Les termes situés au dessus de la ligne horizontale constituent les prémisses ou hypothèses que l'on considère comme valides. Les termes situés en dessous de la ligne constituent les conclusions qui sont vérifiées dès lors que les prémisses le sont. Les jugements sont utilisés en série pour constituer un arbre de preuve permettant de vérifier que chacune des assertions d'un programme est correctement typée et donc que le programme vérifie les règles de typage. Par exemple, soit le système de types suivant, X étant un ensemble de variables entières et Y un ensemble de variables booléennes :

$$\forall x_i \in X, \Gamma \vdash x_i : int$$

$$\forall y_i \in Y, \Gamma \vdash y_i : bool$$

$$\frac{\Gamma \vdash x_i : int \qquad \Gamma \vdash x_j : int}{\Gamma \vdash x_i + x_j : int}$$

$$\frac{\Gamma \vdash y_i : bool \qquad \Gamma \vdash y_j : bool}{\Gamma \vdash y_i \wedge y_j : bool}$$

Les expressions $x_1 + x_2$, $x_1, x_2 \in X$ et $y_1 \wedge y_2$, $y_1, y_2 \in Y$ vérifient les règles de typage. En revanche les expressions $x_1 + y_2$ ou $y_1 \wedge x_2$ sont mal typées.

L'utilisation des systèmes de types pour contrôler les flux d'informations repose généralement sur un typage statique. Volpano et Smith [VIS96, VD97] sont les premiers à utiliser les systèmes de types pour vérifier les flux d'informations au sein d'un programme informatique. Ils s'inspirent des travaux de Denning présentés précédemment. Le principe consiste à définir, pour chaque variable ou expression, un type de sécurité, en plus du type «ordinaire» tel que *entier* ou *réel*. Ce type de sécurité spécifie en fait les politiques de flux qui s'appliquent sur les données manipulées et correspond aux «labels de sécurité» proposés par Denning. La nouveauté réside dans l'utilisation d'un système de vérification des types afin de mettre en œuvre la politique de flux et rejeter, à la compilation, les programmes mal typés, c'est-à-dire les programmes spécifiant des flux d'informations interdits par la politique de flux d'informations. L'utilisation de systèmes de types permet également la composition : un système composé de sous-systèmes réputés «sûrs» d'un point de vue typage de sécurité sera lui même réputé «sûr» dès lors que les signatures externes des différents sous-sytèmes vérifient les règles de typage [SM03a]. C'est là un des principaux avantages des systèmes de type de sécurité qui permet de décomposer l'étape de vérification d'un système.

Le principe du contrôle statique de flux d'informations par contrôle de types est de proposer un mécanisme de vérification automatique de propriétés de non-interférence, présentées dans la sous-section précédente. La définition de la non-interférence retenue est adaptée à l'échelle étudiée : intuitivement, la non-interférence est respectée si le contenu des variables de niveau de sécurité *haut* (ou *privé*) n'influence pas les sorties observables de niveau de sécurité *bas* (ou *public*) du système. Supposons que la politique de flux d'informations définisse deux niveaux de sécurité (la définition pouvant être généralisée à N niveaux de sécurité hiérarchisés par une relation d'ordre partielle définissant une structure de treillis). Nous notons par la suite ces deux niveaux *bas* (B) et *haut* (H). Soit $s = (s_B, s_H)$ un état du système, s_B correspondant à l'état des variables de niveau de sécurité *bas* et s_H correspondant à l'état des variables de niveau de sécurité *haut*. L'état initial est appelé état d'entrée du système et, suite à l'exécution du programme, le système peut atteindre un état $s' = (s'_B, s'_H)$, appelé état de sortie, ou diverger. La sémantique d'un programme ou d'une commande C est alors une fonction $[[C]] : S \to S_\perp$ avec $S_\perp = S \cup \{\perp\}$ et $\perp \notin S$. Cette fonction associe à tout état d'entrée $s \in S$ un état de sortie $[[C]](s)$ ou $\perp$ si le programme ne termine pas. Afin de ne prendre en compte que les variations sur les variables de niveau *haut* de l'état d'entrée du système, les auteurs définissent une relation d'équivalence sur les niveaux *bas* des états d'entrée. Deux états d'entrée du système $s_1 = (s_{1_B}, s_{1_H})$ et $s_2 = (s_{2_B}, s_{2_H})$ sont équivalents sur les niveaux *bas*, noté $s_1 =_B s_2$, si et seulement si $s_{1_B} = s_{2_B}$. Le pouvoir d'observation et de distinction des états de sortie d'un attaquant est modélisé par une relation symétrique et réflexive entre les états de sortie notée $\approx_B$. Cette relation détermine la « vue de niveau bas» du système. Deux états de sortie s'_1 et s'_2 vérifiant $s'_1 \approx_B s'_2$ sont indiscernables par un attaquant qui est réputé ne pouvoir observer que les variables de niveau *bas*. Dès lors, un programme C vérifie la non-interférence entre les niveaux de sécurité *haut* et *bas* si et seulement

si :

$$\forall s_1, s_2 \in S, s_1 =_B s_2 \Longrightarrow [[C]](s_1) \approx_B [[C]](s_2)$$

Cette propriété traduit le fait que le contenu en sortie des variables de niveau *bas* ne dépend pas des variables en entrée de niveau *haut*. Il n'y a donc pas de flux d'informations des variables de niveau *haut* vers les variables de niveau *bas*, observables par un attaquant.

Volpano et Smith définissent un système de types de sécurité pour un langage structuré simple manipulant des entiers et supportant les structures de test et de bouclage [VIS96]. Les auteurs montrent que le système de types proposé permet de contrôler les flux d'informations directs et indirects au sein du programme. Leur contribution principale réside dans l'établissement de la preuve de correction du système de types. Celle-ci repose sur deux lemmes de sécurité :

- un lemme démontrant la propriété de **sécurité simple** : si, d'après le système de types, il est possible de donner le type τ à une expression e, alors seuls les contenus des variables de type, donc de niveau de sécurité, τ (ou de niveau de sécurité inférieur) seront lus lors de l'exécution de e ;
- un lemme démontrant la propriété de **confinement** : si, d'après le système de types, il est possible de donner le type τ à une expression e, alors aucune variable de niveau de sécurité inférieur à τ ne sera modifiée lors de l'exécution de e ;

Les auteurs complètent leur approche en considérant un langage impératif procédural [VD97] et la gestion des exceptions [Smi07]. Ils démontrent également que tout programme respectant le système de types de sécurité vérifie une propriété de non-interférence entre variables telle que définie précédemment [VD97, Smi07]. Plusieurs travaux se sont également intéressés aux système de types de sécurité et en ont étendu le domaine d'application. Les travaux de Banerjee et Naumann [BN03] et Myers [Mye99] portent notamment sur les langages à objets. Pottier et Simonet [PS03] se sont intéressés aux langages fonctionnels. D'autres travaux ont porté sur les aspects liés à l'indéterminisme. Les causes mêmes de l'indéterminisme sont multiples et nécessitent parfois des approches différentes :

- le programme peut contenir des instructions dont l'exécution est intrinsèquement aléatoire ou pseudo-aléatoire (par exemple l'utilisation d'une fonction de tirage aléatoire) ;
- le programme peut contenir plusieurs files d'exécution s'exécutant parallèlement sur un ou plusieurs processeurs ;
- le programme peut être distribué sur plusieurs machines communiquant à travers un réseau ;

La non-interférence reposant, dans sa version originale, sur un modèle déterministe du système, n'est pas adaptée d'où le recours à des modèles raisonnant sur l'ensemble des valeurs possibles (modèles possibilistes) [Man00]. Banâtre, Bryce et Le Métayer [BBM94] sont parmi les premiers à s'intéresser au contrôle de flux statique dans un langage où les choix peuvent être aléatoires. Smith et Volpano [SV98, VS99, Smi01], tout comme Boudol et Castellani [BC01] définissent des systèmes de types de sécurité adaptés à l'exécution concurrente de plusieurs

files d'exécution. Sabelfeld et Mantel [SM02] s'intéressent aux programmes distribués et proposent un système de types compatible avec l'exécution concurrente et distribuée via l'échange de messages.

1.2.1.4 Mise en pratique

Le contrôle de flux par analyse statique, et en particulier par contrôle de types de sécurité, est aujourd'hui un domaine de recherche prolifique. Les travaux publiés couvrent différents aspects, présentés succinctement dans le paragraphe précédent. Les articles de Myers [SM03a] et Smith [Smi07] constituent par ailleurs une référence bibliographique intéressante couvrant la plupart des travaux du domaine. Pourtant, malgré plus de trente années de recherche, force est de constater que ces recherches ont eu peu d'impacts en pratique [Smi07] et que les techniques de contrôle statique de flux d'informations sont peu ou pas utilisées dans les systèmes opérationnels [Zda04]. Plusieurs arguments peuvent expliquer la faible adoption de ces techniques :

- le manque de prise en compte des architectures et des problèmes de sécurité «réels» rencontrés dans les systèmes opérationnels. En effet, beaucoup de travaux se sont attachés à définir des propriétés complexes afin d'effectuer une analyse statique précise des fuites d'information possibles. Peu de travaux en revanche se sont intéressés à définir un langage adapté au développement des systèmes opérationnels. En outre, peu de travaux se sont attachés à démontrer les bénéfices réels de ces techniques dans la prévention des attaques courantes, sur des architectures opérationnelles.
- les restrictions trop importantes imposées par la non-interférence. Cette propriété requiert en effet, pour toute modification des variables privées, l'absence totale d'observation donc l'absence de variation sur les variables publiques. A l'extrême, une modification d'un seul bit d'information sur une variable publique, suite à la modification d'une variable privée, suffit à contredire cette propriété. Or dans bien des cas, l'observation d'une variation sur les observations ne permet pas d'inférer une quantité d'information suffisante pour un attaquant. Comme indiqué en section 1.2.1.2, l'utilisation de primitives cryptographiques ou la vérification d'un mot de passe illustre ce problème. De manière générale, la non-interférence est souvent violée dans les systèmes opérationnels et ce même en l'absence d'attaque [RMMG01].
- le manque d'intégration avec les mécanismes de sécurité existants et de prise en compte des politiques complexes de haut niveau. Les politiques de flux d'informations reposant sur des propriétés de «bout-en-bout», elles doivent être appliquées à l'ensemble du système. Les techniques de contrôle statique des flux d'informations reposant sur une réécriture, fut-elle partielle, du code des différents composants logiciels, l'application de ces techniques à l'ensemble des composants du système n'est souvent pas envisageable. Seuls certains composants peuvent être réécrits et analysés. Dès lors, ces techniques doivent coopérer avec des mécanismes s'appliquant à l'ensemble du système. De plus, les techniques de contrôle statique de

flux d'informations reposent sur la spécification de politiques de flux d'informations entre des conteneurs de faible granularité, par exemple des variables d'un programme. Il s'agit donc de politiques de bas-niveau, spécifiées sur chaque conteneur à l'aide de labels ou de types de sécurité. *A contrario*, les politiques de sécurité qui s'appliquent sur les systèmes reposent sur des exigences de haut-niveau, définies entre des conteneurs de granularité plus importante (des fichiers, des interfaces réseau, etc.) ou des entités actives (des utilisateurs, des applications, etc.). Il est donc nécessaire d'établir un lien entre les exigences de haut niveau et la politique de flux de bas niveau, et ce de façon dynamique. En effet, de nombreuses applications pouvant manipuler des données de sensibilités différentes, les labels ou types de sécurité doivent être en partie associés dynamiquement aux conteneurs d'informations de l'application, suivant le contexte d'exécution. Par exemple, un même éditeur de texte peut être utilisé pour lire et modifier des données publiques ou confidentielles, suivant le contexte d'exécution.

Certains travaux récents se sont attachés à résoudre les problèmes évoqués précédemment. Les travaux de Myers [Mye99] sur JIF[3], un langage dérivé de Java proposant un système de types de sécurité, et ceux de Simonet et Pottier [Sim03] sur Flow Caml[4], un système de types de sécurité pour Caml, démontrent que des implémentations «réalistes» de la technique de typage de sécurité sont possibles. Les récents travaux de Li et Zdancewic [LZ05b] ainsi que de Chong *et al.* [CLM+07, CVM07] étendent le domaine d'application de ces approches au contrôle de flux dans les applications web. Zheng et Myers [ZM04] ainsi que Tse et Zdancewic [TZ04] se sont également intéressés aux aspects dynamiques et contextuels des politiques de sécurité. En effet, certaines politiques de flux dépendent du contexte d'exécution et ne peuvent être définies lors de la compilation. Ces cas ne peuvent donc être traités par l'utilisation des labels statiques et justifient le recours à des labels de sécurité dynamiques. Sous JIF, ces labels sont des variables liées à des variables de type classique (booléen, entier, etc.) et qui représentent en fait le label de sécurité de la variable liée. JIF propose par ailleurs des instructions qui permettent de manipuler ces variables de label et notamment de tester leur valeur. Les instructions de test sur les variables de label permettent au système de vérification de type de sécurité de prendre en compte certaines variables dont le type de sécurité ne peut être déterminé de façon statique, lors de la vérification. L'exemple suivant, tiré du manuel de JIF[5] illustre ce principe :

```
void m(int{*lbl} i, label{} lbl) {
    int{Alice:} x;
    x = i;
}
```

[3] http://www.cs.cornell.edu/jif/

[4] http://cristal.inria.fr/~simonet/soft/flowcaml/

[5] http://www.cs.cornell.edu/jif/doc/jif-3.2.0/manual.html

Les types de sécurité sont accolés aux types usuels de Java (int, bool, etc.) et notés entre accolades. Ainsi, à la ligne 2, la variable locale x est déclarée de type entier et de type de sécurité $\{Alice :\}$, ce qui signifie qu'elle correspond à la classe de sécurité de l'utilisateur Alice. L'argument de la fonction m, l'entier i, possède quant à lui un type de sécurité dynamique, inconnu lors de la vérification statique, c'est-à-dire lors de la compilation. Le label de sécurité de ce paramètre est spécifié par la variable *lbl* (de type «label de sécurité», *label*{}). La variable *lbl* est liée à la variable i par la notation $\{*lbl\}$ qui rappelle la notation utilisée en C pour les pointeurs : le type de sécurité de i est la valeur de *lbl*. Le type de sécurité de i ne pouvant être déterminé par l'analyse statique, ce programme sera rejeté lors de la compilation du fait de l'incompatibilité des types des variables x et i. Afin de prendre en compte le type dynamique de i, le développeur doit «décorer» l'opération d'affectation par une structure conditionnelle portant sur *lbl*, comme dans l'exemple suivant :

```
void m(int{*lbl} i, label{} lbl) {
    int{Alice:} x;
    if (lbl <= new label {Alice:}) {
        x = i;
    }
    else {
        x = 0;
    }
}
```

Dans cet exemple, la structure conditionnelle de la ligne 3 permet de prendre en compte le type de sécurité de i. Le test permet en effet de s'assurer que l'affectation de la ligne 4 ne sera effectuée que si le label dynamique de sécurité associé à i est dominé par $\{Alice :\}$. Le système de vérification des types interprète l'information apportée par ce test en faisant évoluer le *contexte de typage*. Ainsi, suite à la structure conditionnelle de la ligne 3, le contexte de typage assume que le type de i est dominé par $\{Alice :\}$. Dès lors, l'affectation de la ligne 4 respecte le système de types. Cette approche permet donc de traiter les labels dynamiques, inconnus lors de la phase de vérification. Néanmoins, la vérification étant statique, elle ne peut tenir compte du contexte d'exécution que si le développeur spécifie explicitement les vérifications sur les variables de labels dynamiques. Dès lors que l'ensemble des labels de sécurité forme un treillis complexe et de taille importante, ces vérifications peuvent devenir complexes et fastidieuses.

Enfin, certains travaux ont porté sur un élément indispensable à l'application des techniques de contrôle de flux aux systèmes opérationnels : la déclassification. Ce principe consiste à abaisser le label de sécurité attaché à une donnée, ou à un conteneur, et ce afin d'autoriser certains flux d'informations qui auraient été interdits par ailleurs : il s'agit donc de traiter des exceptions à la politique de flux. Les raisons qui justifient l'utilisation de la déclassification sont notamment d'ordre pratique. Les politiques de flux basées sur la non-interférence reposent sur une définition stricte de l'absence de flux d'informations. En pratique, il est

souvent nécessaire non pas d'interdire totalement un flux d'informations mais de limiter la quantité d'information échangée. Par exemple, lors de la vérification d'un mot de passe, la politique de flux doit interdire les flux révélant totalement le mot de passe à un attaquant mais doit autoriser la vérification du mot de passe par un test de comparaison. De même, les flux mettant en œuvre des données chiffrées peuvent être autorisés là où les flux de données en clair sont interdits. Dans chacun des cas, l'hypothèse est faite que la quantité d'information transmise est insuffisante pour qu'un attaquant infère (ou modifie) sensiblement les données protégées. Afin de traiter ces différents cas, il serait en théorie nécessaire d'utiliser un modèle probabiliste permettant d'évaluer la quantité d'information transmise lors d'un flux d'informations. Les travaux de McCamant et Ernst [ME08] ainsi que ceux de Lowe [Low02] s'intéressent entre autres à définir un modèle quantitatif des flux d'informations. Cependant, cette approche, orthogonale à la déclassification, est difficile à mettre en œuvre en pratique :

- il est difficile en pratique de définir une politique de flux en fixant des seuils de quantité d'information acceptables ;
- la mesure quantitative des flux d'informations est elle même plus délicate que l'appréciation qualitative de la présence de flux d'informations.

Si la déclassification permet de traiter en pratique un certain nombre de cas, la souplesse qu'elle apporte contredit la propriété de non-interférence. L'utilisation de ce mécanisme entraîne une violation de la politique de flux car la non-interférence ne permet pas de distinguer une fuite d'information volontaire d'un flux résultant d'une intrusion. Les approches qui proposent un mécanisme de déclassification reposent donc sur l'hypothèse que ce mécanisme est utilisé correctement, c'est-à-dire qu'il ne permet pas à un attaquant de découvrir ou de modifier une donnée à laquelle il n'a pas accès. Dans le cadre des techniques d'analyse statique, c'est le développeur (ou la personne responsable de la certification du programme) qui définit les labels de sécurité et peut avoir recours à la déclassification. L'hypothèse est donc faite que cette personne utilise le mécanisme à bon escient. Le langage JIF propose par exemple un mécanisme de déclassification au travers de la primitive `declassify(expression, label)`. Cette primitive permet de définir explicitement le type de sécurité associé à une expression et ce quelles que soient les règles de typage associées à l'expression. Ce mécanisme est donc semblable au mécanisme de conversion de type (ou cœrcition) utilisé habituellement dans les langages comme C ou Java permettant notamment de transformer un entier en réel. Considérons par exemple un mécanisme trivial de vérification de mot de passe. Le mot de passe est un entier et le système de types de sécurité comportent deux labels, *secret* (S) et *public* (P). Seuls les flux de *public* vers *secret* sont autorisés : le niveau de sécurité *public* domine donc le niveau *secret* dans le treillis à deux éléments de cette politique de flux d'informations. Les données manipulées par l'utilisateur désirant s'authentifier sont de niveau de sécurité *public*. En revanche le mot de passe stocké est de niveau *privé* car il ne doit pas être révélé à un attaquant. Le pseudo-code correspondant au système est le suivant :

```
int {S} pwd = 17;
int {P}  guess =  getUserInput();
boolean {S} test = (guess == pwd);
if (declassify(test, P))
        print(``Mot de passe OK'');
else
        print(``Mauvais mot de passe'');
```

Le mot de passe proposé par l'utilisateur est récupéré grâce à la fonction `getUserInput()`. Cette donnée, de niveau *public*, est comparée au mot de passe stocké et le résultat est une donnée de niveau *secret*. Cette opération nécessite une conversion de type de la variable `guess` de *public* vers *secret*. Cette conversion de type peut toutefois être réalisée automatiquement sans violer la politique de flux car *public* domine *secret*. En revanche, il n'est pas possible d'informer l'utilisateur du résultat du test sans violer la politique de flux. En effet, le résultat du test dépend du mot de passe stocké qui est lui-même de niveau *secret*. Le développeur considérant ici que la quantité d'information révélée par le test est raisonnable (il devrait en toute rigueur limiter le nombre d'essais), le type de sécurité de `test` a été modifié explicitement à l'aide de l'instruction `declassify(test, P)` afin de pouvoir informer l'utilisateur du résultat du test et donc du processus d'authentification. Le programme ainsi spécifié est vérifié par le système de types bien que la politique de flux ne soit pas respectée.

Plusieurs travaux se sont intéressés à définir une propriété de sécurité permettant de prendre en compte la déclassification. Il s'agit généralement d'une forme relaxée de la non-interférence. Ainsi Matos et Boudol [MB05] définissent l'absence de divulgation ou *non-disclosure*, comme une forme généralisée de non-interférence. Plus précisément, les auteurs proposent un mécanisme de déclassification reposant sur des politiques locales de flux d'informations. Ainsi, dans chaque état du système, la politique de flux d'informations qui s'applique est constituée par l'union de la politique globale, qui définit les flux autorisés pour tous les états du système, et d'une politique locale qui définit les flux d'informations autorisés seulement dans l'état courant et qui permet par exemple d'autoriser des flux par ailleurs interdits par la politique globale. L'absence de divulgation peut alors être vue comme une forme de non-interférence locale, qui est vérifiée pour chaque état du système à partir de l'union de la politique globale et de la politique locale. Mantel et Sands [MS04] proposent une approche similaire. Dans les deux cas, la propriété généralisant la non-interférence permet de garantir l'absence de flux d'informations en dehors de l'utilisation du mécanisme de déclassification, ce que ne permet pas la non-interférence, mais aucune garantie n'est apportée sur les flux résultant de l'utilisation de la déclassification. D'autres travaux se sont en revanche intéressés aux conditions d'utilisation des mécanismes de déclassification en imposant des restrictions. Ainsi Myers *et al.* définissent la notion de déclassification robuste. Les auteurs proposent de définir **qui** peut effectuer la déclassification. Ils s'appuient en effet sur le modèle de labels décentralisés proposé par Myers et Liskov [ML97] et qui introduit la notion de propriétaire de l'information. Cette notion étend aux contenus un

principe utilisé couramment pour la gestion des conteneurs par les mécanismes de contrôle d'accès discrétionnaire. Dans ce modèle, utilisé notamment dans JIF, seul le propriétaire peut déclassifier l'information qu'il possède. La déclassification robuste impose en outre qu'un attaquant actif, qui possède de l'information et peut donc la déclassifier, ne peut inférer plus d'information qu'un attaquant passif. Li et Zdancewic proposent également une propriété de sécurité adaptée à la déclassification [LZ05a]. Les auteurs définissent une politique de déclassification à l'aide de labels spécifiant **comment** l'information peut être déclassifiée (suite à l'utilisation d'une fonction de hachage ou de cryptographie, suite à une comparaison à une valeur, etc.). Sabelfeld et Myers introduisent la notion de fuite d'information limitée ou *delimited information release* [SM03b]. Les auteurs considèrent que les flux d'informations doivent respecter une politique globale de flux d'informations sauf en certains endroits du programme, les «sas de secours» ou *escape-hatch*. De plus, les «sas de secours» ne doivent pas permettre de faire fuir plus d'information, au sein de l'ensemble du programme, que la quantité d'information transitant uniquement par les «sas de secours». Concrètement cette restriction impose que le programme ne peut modifier les données susceptibles d'influencer la valeur des données déclassifiées. Les auteurs définissent donc **quelle** information peut violer la politique de flux et **où** (dans le programme) peut s'effectuer la déclassification.

Ainsi, plusieurs propriétés de sécurité généralisant la notion de non-interférence ont été proposées afin de prendre en compte la déclassification. Cependant, il n'existe pas aujourd'hui de consensus sur la propriété adéquate. De plus, il convient de s'interroger sur la confiance qu'apportent de telles propriétés dans la sécurité des programmes certifiés. En effet, ces propriétés n'offrent des garanties que sur les flux d'informations non déclassifiés. L'hypothèse est donc faite que les opérations de déclassification ne peuvent être détournées par un attaquant. Dans tous les cas, la confiance dans la sécurité du système dépend de la confiance que l'on place dans l'entité, personne physique ou composant logiciel, responsable de l'opération de déclassification. Dans le cadre de l'analyse statique, il s'agit généralement du développeur ou du certifieur du programme informatique, qui spécifie explicitement les flux déclassifiés. La déclassification automatique peut également être envisagée, notamment dans le cadre des approches dynamiques présentées dans la section suivante, pour des composants logiciels réputés «sûrs». Il s'agit typiquement de briques logicielles élémentaires, de petite taille, dont la spécification et l'implémentation ont fait l'objet d'une étude approfondie garantissant que ces composants ne peuvent être détournés par un attaquant pour révéler ou modifier des données privées, par exemple des fonctions cryptographiques, des primitives d'authentification, etc. Certains travaux se sont intéressés à limiter l'usage de la déclassification suivant plusieurs axes orthogonaux et complémentaires résumés par Sabelfeld et Sands [SS07] :

- **Quelle** information peut être déclassifiée ?
- **Qui** peut déclassifier ?
- **Où** peut-on déclassifier ?
- **Comment** les flux peuvent-ils être déclassifiés ?
- etc.

Restreindre l'utilisation de la déclassification permet en théorie de limiter la confiance dans l'entité réalisant la déclassification et de renforcer la confiance dans le contrôle des flux d'informations. Cependant, aucun des travaux cités précédemment n'évalue en pratique l'efficacité des restrictions proposées, celles-ci pouvant même dans certains cas interdire l'utilisation légitime de la déclassification. Ce mécanisme, nécessaire dans certains cas, doit donc être utilisé avec parcimonie et à bon escient et impose de déléguer une partie de la confiance sur l'entité responsable de la déclassification.

1.2.1.5 Bilan sur le contrôle statique de flux d'informations

L'approche statique du contrôle des flux d'informations a fait l'objet de nombreux travaux, depuis ceux précurseurs de Denning jusqu'aux travaux récents sur la déclassification et l'implémentation pour des langages modernes comme Java. Cette approche a longtemps été préférée à l'approche dynamique, décrite dans la section 1.2.2, car elle présente un certain nombre d'avantages :

- le contrôle des flux est réalisé *a priori*, avant toute exécution. Il s'agit donc d'une méthode préventive qui permet de garantir la sécurité des logiciels ;
- le contrôle est réalisé « hors-ligne », par exemple lors de la compilation ou lors de la certification du programme. Le système n'est donc pas pénalisé à l'exécution, en termes de charge et d'occupation mémoire, par la vérification des flux ;
- l'analyse porte sur le code source des programmes et permet de raisonner sur les différentes exécutions du programme. Il est alors possible de vérifier des propriétés de sécurité fortes telle que la non-interférence.

L'analyse précise du code source est toutefois délicate. Les solutions proposées reposent généralement sur un sous-ensemble des flux autorisés : les programmes validés par la méthode respectent la politique de flux mais il existe des programmes, rejetés par la méthode, qui respectent également cette politique. Le programme suivant illustre l'incomplétude, en termes de programmes acceptés, des méthodes statiques :

```
a := 3;
b := 5;
c := 2;
d := b - (a + c);
```

Il est évident que la valeur prise par d à l'issue de la dernière affectation sera toujours nulle. Dès lors, l'observation de la valeur de d ne révèle aucune information sur la valeur initiale de a, b et c. Il n'existe donc pas de flux d'informations de a, b ou c vers d. Seule une spécification manuelle ou la détermination à partir du code source des propriétés d'invariants [ECGN01, NE02] permet de prendre en compte efficacement ce type de cas lors d'une analyse statique des flux d'informations. La plupart des approches se contentent de raisonner sur les flux d'informations «probables» et considèrent dans l'exemple précédent que l'affectation `d := b - (a + c);` génère un flux d'informations de a, b et c vers d.

En outre, les techniques de contrôle statique restent peu ou pas utilisées en pratique. Certains travaux se sont attachés à développer les aspects pratiques afin de favoriser l'adoption des techniques de contrôle de flux par analyse statique. Le contrôle de types de sécurité, bien que reposant sur une approximation de la sémantique du programme, permet de contrôler les flux d'informations à la compilation du programme. Ce mécanisme, implémenté par exemple dans JIF, permet au développeur de s'assurer que le programme qu'il développe respecte une politique de flux préalablement définie et qui constitue donc une spécification des exigences de sécurité associées au programme. Toutefois, plusieurs limites subsistent :

- le contrôle étant effectué lors de la compilation, il ne peut prendre en compte le contexte de déploiement et d'exécution. Or, pour beaucoup de logiciels, une part importante de la politique de sécurité est définie lors du déploiement, par l'administrateur ou le responsable de la sécurité. De plus, la politique de sécurité dépend en partie du contexte d'exécution. Par exemple, la politique peut dépendre du sujet qui exécute le programme, des conteneurs d'informations accédés, des entités avec lesquelles le programme dialogue via les ports d'entrée/sortie, etc. Certaines approches, comme JIF, permettent en partie de prendre en compte ces aspects dynamiques de la politique de sécurité mais elles nécessitent que le développeur spécifie explicitement les contrôles (par exemple sur l'identité des sujets). Cette solution nécessite donc que le développeur envisage dès la conception du logiciel les différents contextes d'exécution possibles ce qui n'est pas envisageable pour beaucoup de programmes.
- les méthodes statiques permettent de s'assurer que le logiciel est «sûr» et donc que **toutes** les exécutions possibles respectent la politique de flux. Cette propriété *a priori* intéressante peut s'avérer contraignante. La non-interférence étant par nature une propriété très restrictive et les systèmes de contrôle raisonnant sur une sur-approximation des flux d'informations, les méthodes statiques rejettent donc un nombre important de programmes dont certaines exécutions sont susceptibles de violer la politique de sécurité. Le développement de programmes «sûrs», dont toutes les exécutions respectent la politique, est en effet coûteux et parfois difficile. Il est souvent nécessaire d'utiliser des programmes «non sûrs» mais dont les exécutions effectives respectent, de par le contexte d'exécution, la politique de sécurité.
- dans les systèmes complexes, constitués de plusieurs composants logiciels provenant de plusieurs fournisseurs et interagissant entre eux, la détection des intrusions nécessite de contrôler tous les flux d'informations au sein de chacun des logiciels déployés, ainsi qu'entre les différents programmes interagissants. Certains de ces logiciels sont des *Commercial Of-The-Shelves*, qui signifie littéralement «disponible dans le commerce sur l'étagère». L'utilisateur de ces logiciels n'est donc pas ou peu impliqué dans leur développement. Dans certains cas il n'a même pas accès au code source. Il n'est dans ce cas pas envisageable d'appliquer les méthodes statiques à l'ensemble du système et de ce fait certains flux d'informations ne peuvent

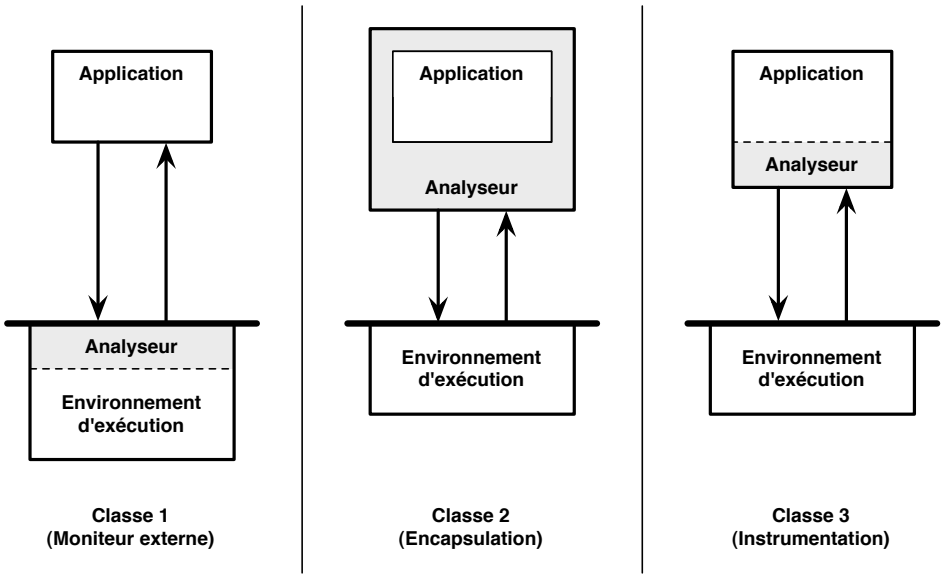

FIG. 1.3 – Différentes classes de solutions pour le contrôle dynamique des flux d'informations

être contrôlés.

Ces limites justifient le recours aux méthodes dynamiques, présentées dans la section suivante, qui effectuent un contrôle lors de l'exécution.

1.2.2 Approches dynamiques

Le contrôle des flux d'informations peut également être réalisé dynamiquement, lors de l'exécution. Afin de suivre les flux d'informations et d'en vérifier la légalité, le système doit comporter une entité supplémentaire, logicielle ou matérielle, afin de suivre et analyser les flux d'informations. Plusieurs solutions ont été proposées pour implémenter un analyseur qui peuvent être regroupées suivant trois grandes classes, illustrées par la figure 1.3[6] :

– dans la première classe, l'analyseur est implémenté sous la forme d'un moniteur externe, situé au niveau de l'environnement d'exécution (système d'exploitation, machine virtuelle, etc.) sur lequel les programmes s'exécutent. L'environnement d'exécution constituant l'interface de médiation entre les applications et le matériel (processeur, mémoire vive, mémoire de masse, etc.), l'implémentation d'un moniteur au sein de cet environnement permet donc d'observer les différents accès aux conteneurs d'informations. Cette approche nécessite de modifier en partie l'environnement d'exécu-

[6]Cette figure est inspirée de celle proposée par Raphaël Khoury dans son mémoire de master [Kho05]

tion, ce qui n'est pas toujours aisé. En revanche, elle ne nécessite aucune modification des différentes applications.

- la deuxième classe est constituée par les approches d'encapsulation ou *wrapping*. L'analyseur est implémenté sous la forme d'un ensemble de fonctions de suivi des flux d'informations. Les appels aux fonctions des bibliothèques du système sont interceptés et des appels aux fonctions de suivi des flux d'informations sont intercalés. L'approche est similaire à celle de la première classe tout en permettant de s'affranchir de modifier le système d'exécution. Toutefois, il est parfois nécessaire de vérifier que l'attaquant ne tente pas d'échapper au contrôle de flux, les fonctions des bibliothèques système ne constituant pas nécessairement une interface de médiation.
- la troisième classe est constituée par les approches d'instrumentation des programmes. Dans cette classe, l'analyseur est implémenté au sein même des applications par l'ajout d'instructions permettant le suivi et le contrôle des flux d'informations. L'instrumentation peut être réalisée à l'échelle du code source, lors de la compilation. Elle peut également être réalisée à partir du code binaire ou du *bytecode* des applications. Cette approche permet de suivre au plus près les flux d'informations internes des applications en contrôlant l'accès aux différentes variables du programme. Afin de suivre les flux entre les différentes applications, cette approche doit être combinée avec une des deux précédentes ou s'appuyer sur un support matériel. Il est également parfois nécessaire de s'assurer qu'un attaquant ne puisse échapper au contrôle [ÚES99], notamment grâce à l'injection de code, par exemple en exploitant un débordement de tampon.

Dans les différents cas de figure, les solutions sont généralement de type logiciel mais peuvent s'appuyer en partie sur des mécanismes matériels. Le support matériel permet de faciliter le suivi des flux d'informations, notamment pour les conteneurs d'informations de faible granularité (par exemple des registres de processeurs). Il permet également d'améliorer les performances et de limiter le risque d'attaques contre le mécanisme de suivi des flux (les solutions matérielles étant généralement réputées plus résistantes que les solutions logicielles). Cependant il nécessite de modifier les architectures matérielles existantes (au niveau des processeurs, de la mémoire, des bus, etc.), ce qui limite considérablement les chances d'adoption de ce type de solution. Des modifications partielles des architectures matérielles à des fins de sécurité peuvent certes être envisageables (l'ajout du bit de non-exécution ou *NX bit* par exemple). Cependant, l'adoption de ces modifications par les fondeurs, pour les architectures classiques (X86, AMD64, ARM, etc.) impose un certain nombre de contraintes :

- la solution proposée doit apporter un gain substantiel en termes de sécurité pour une partie importante des utilisateurs ;
- les modifications doivent être mineures afin de minimiser le surcoût et assurer une compatibilité avec les architectures existantes.

L'architecture que nous proposons, présentée au chapitre 3, combine des solutions appartenant à différentes classes. Elle s'appuie à la fois sur un moniteur externe implémenté au sein du système d'exploitation et sur l'instrumentation

de certaines applications, au niveau du bytecode Java.

Un certain nombre de travaux se sont intéressés à définir un cadre théorique pour le contrôle dynamique des politiques de sécurité [Sch00a, BLW02]. Le but est notamment de définir l'ensemble des politiques de sécurité qu'il est possible de contrôler dynamiquement à partir de l'observation d'une exécution d'un programme. Schneider définit ainsi la classe EM (*Execution monitoring*) des mécanismes de contrôle dynamique qui surveillent les différentes étapes de l'évolution d'un système «cible». Plusieurs hypothèses caractérisent cette classe de mécanisme :

- le mécanisme de sécurité fonde uniquement son jugement sur la trace courante du système surveillé et la spécification de la politique. Cette trace est elle-même constituée de l'ensemble des opérations observées (accès à des conteneurs d'informations, opérations d'entrée/sortie, etc.) depuis l'état initial. L'observation d'une seule exécution permet donc de déterminer si la politique est violée : la politique de sécurité doit être une propriété de trace.
- toute exécution rejetée par le mécanisme doit l'être après l'observation d'une trace finie ;
- si une trace finie viole la politique, alors toute trace préfixée par cette trace finie doit être rejetée par le mécanisme.

Ces hypothèses caractérisent en fait les propriétés de sûreté. L'auteur montre que les politiques de sécurité qui peuvent être mises en application à l'aide d'un mécanisme de classe EM sont des politiques de sûreté. Les hypothèses évoquées précédemment forment donc des conditions nécessaires mais non suffisantes, toutes les propriétés de sûreté ne pouvant pas nécessairement être mise en application à l'aide d'un mécanisme de classe EM. En corollaire, il n'est donc pas possible, d'un point de vue strict, de mettre en pratique à l'aide d'un mécanisme de classe EM une politique de sécurité qui ne serait pas une propriété de sûreté. En particulier, l'auteur mentionne, en s'appuyant sur les travaux de MacLean [McL94], le fait que les politiques de flux reposant sur des propriétés dérivées de la non-interférence, évoquée en section 1.2.1.2, ne sont pas des propriétés de trace. En effet, la non-interférence est une propriété d'ensemble de traces. L'observation d'une seule exécution ne permet pas dans le cas général de déterminer précisément à elle seule l'absence de flux d'informations, telle que la définit la non-interférence. Il est nécessaire, dans le cas général, d'observer plusieurs exécutions du système afin de détecter de manière précise et complète la présence (ou l'absence) de flux d'informations. En particulier, l'absence de modification d'un conteneur d'informations lors d'une exécution ne permet pas d'affirmer l'absence de flux d'informations. Considérons par exemple la fonction suivante :

```
void test(bool a) {
print(``Bonjour'');
if (a)
        print(``a est vraie'');
print(``Au revoir'');
```

```
}
```

Supposons que le paramètre `a` de cette fonction soit une donnée de type *Secret* et que la console de sortie où s'affichent les messages soit visible par un observateur non privilégié, de type *Public*. Si la politique interdit les flux du domaine *Public* au domaine *Secret*, cette fonction viole clairement la politique de flux, et ce quelles que soient les exécutions. En effet deux exécutions sont possibles :

- si la fonction est appelée avec le paramètre `a` valant `true`, l'utilisateur observe l'affichage de la séquence :

  ```
  Bonjour
  a est vraie
  Au revoir
  ```

  Il existe donc un flux d'informations de `a` vers la console d'affichage et ce flux est détectable par un mécanisme dynamique car il est matérialisé par une lecture de la variable `a`, lors du test de la ligne 3, suivi d'une écriture sur l'interface de sortie via la fonction d'affichage.
- si la fonction est appelée avec le paramètre `a` valant `false`, l'utilisateur observe l'affichage de la séquence :

  ```
  Bonjour
  Au revoir
  ```

  Il existe en fait là aussi un flux d'informations de `a` vers la console d'affichage car un attaquant peut inférer la valeur de `a` en observant l'absence du message «`a est vraie`». En revanche la lecture de la valeur de `a` n'étant pas suivie d'une écriture, dans le contexte du branchement conditionnel, un analyseur de la classe EM ne peut détecter ce flux d'informations.

Les moniteurs de la classe EM, utilisés pour suivre les flux d'informations au sein d'un système, peuvent donc présenter un certain nombre de faux négatifs. De plus, ils peuvent présenter en pratique, tout comme les approches statiques, un certain nombre de faux positifs. En effet, l'observation de la succession des accès en lecture puis en écriture ne permet pas de décrire précisément les flux d'informations. Ainsi, l'instruction `b=a-a` sera considérée à tort par un mécanisme de classe EM comme un flux d'informations de `b` vers `a`.

Deux approches complémentaires peuvent être adoptées pour faire face à ce problème :

1. des hypothèses peuvent être faites sur la nature des flux d'informations utilisés par les attaquants (absence de flux indécelable) et par le programme (faible probabilité des cas «pathologiques»). En pratique, des faux positifs ou négatifs peuvent apparaître lorsque ces hypothèses ne sont pas vérifiées.
2. l'analyse de la trace courante peut être augmentée par des connaissances supplémentaires sur le programme, notamment sur les branches non exécutées, les invariants, etc. L'analyse dynamique peut ainsi être couplée à une analyse statique réalisée auparavant. Les analyseurs s'appuyant sur

ces techniques ne font pas partie des analyseurs de la classe EM et peuvent donc en partie s'affranchir de leurs limites.

1.2.2.1 Contrôle par moniteur externe

La plupart des travaux proposant de suivre les flux d'informations à l'aide d'un moniteur externe s'appuient sur un mécanisme implémenté au sein du système d'exploitation [MR92,FGQ96,Fra00,BD03,EKV+05,ZBWKM06,KYB+07, Zim03]. Un système d'exploitation ou *Operating System (OS)* est aujourd'hui présent sur la majorité des systèmes informatiques, parfois sous une forme minimale. Il assure une interface de médiation entre les applications et le matériel, cette interface étant généralement protégée par un dispositif matériel présent sur la majorité des processeurs. L'utilisation d'un moniteur externe au sein du système d'exploitation permet donc de suivre les flux d'informations générés par l'ensemble des applications du système. La capacité d'un attaquant à contourner le système de suivi des flux d'informations est limitée grâce à l'interface de médiation que forment les appels système. L'ensemble des composants de confiance, souvent appelé *Trusted Computing Base* [LABW92], auxquels il est nécessaire de se fier est réduit au matériel et au noyau du système d'exploitation qui comprend alors également le moniteur de suivi des flux d'informations.

Certains de ces systèmes peuvent être assimilés à des systèmes de contrôle d'accès mandataire (MAC) évoqués en introduction. La principale différence avec les système classiques de contrôle mandataire, implémentés par exemple dans SELinux [7], TrustedBSD [8] ou TrustedSolaris [9] réside dans l'approche adoptée pour le suivi dynamique des flux d'informations. Les mécanismes classiques de contrôle d'accès mandataire ont en effet pour but la prévention des flux d'informations illégaux en assignant des labels de sécurité statiques aux conteneurs et aux processus du système. Cette approche permet de mettre en œuvre des politiques très restrictives et offre au système un haut niveau de protection nécessaire, entre autres, pour les applications militaires. Cependant, ces systèmes sont souvent critiqués pour leur manque de souplesse et leur difficulté de mise en œuvre. L'utilisation de labels statiques explique en partie le manque de souplesse de ces systèmes : l'administrateur doit prévoir *a priori* l'ensemble des flux d'informations utilisés légalement par les applications afin de spécifier explicitement l'ensemble des labels de sécurité.

Les travaux de Weissman sur ADEPT-50 [Wei69] sont parmi les premiers à proposer de suivre les flux d'informations à l'aide de labels dynamiques. Cette approche est reprise dans les travaux de McIlroy et Reeds [MR92] ainsi que ceux de Foley, Gong et Qian [FGQ96]. Les auteurs précisent que les labels dynamiques, dont la valeur varie lors de l'exécution, peuvent potentiellement constituer un canal caché entraînant une fuite d'information. Toutefois, ces problèmes peuvent être limités en imposant des restrictions sur l'évolution des labels [FGQ96]. En outre, McIlroy et Reeds affirment que l'utilisation de labels

[7] `http://www.nsa.gov/selinux/`
[8] `http://www.trustedbsd.org/`
[9] `http://www.sun.com/software/solaris/trustedsolaris/index.xml`

dynamiques constitue un canal caché tolérable pour une utilisation non-militaire et qu'en revanche cette approche permet une souplesse indispensable pour ce type d'utilisation [MR92]. Plus récemment, Fraser [Fra00] ainsi que Beres et Dalton [BD03] ont implémenté cette approche sous Linux. Zimmermann [Zim03] propose également une solution de détection d'intrusions paramétrée par la politique de sécurité, présentée en section 1.1.2.3, qu'il implémente sous la forme d'un moniteur de suivi des flux d'informations pour le noyau Linux.

Certains travaux se sont intéressés à développer de nouveaux systèmes d'exploitation permettant le suivi dynamique des flux d'informations tels que Asbestos [EKV+05] et HiStar [ZBWKM06]. Ces travaux bénéficient d'une architecture adéquate et optimisée pour le suivi dynamique des flux d'informations et sont donc plus efficaces que les approches s'appuyant sur des systèmes existants dont l'architecture n'a pas été étudiée pour faciliter ce suivi. Néanmoins, le manque de compatibilité avec les applications existantes constitue un frein à l'adoption de tels systèmes car ils nécessitent la réécriture partielle de l'ensemble des applications. HiStar dispose d'un mode de compatibilité UNIX pour les applications mais propose peu de pilotes de périphériques.

Nous pensons qu'un moniteur externe implémenté sur un OS existant constitue une approche efficace et réaliste pour le contrôle des flux d'informations sur un système. Ce type d'approche permet en effet à moindre coût de surveiller l'ensemble des applications du système tout en minimisant le nombre de composants de la TCB. Néanmoins, ce type d'approche souffre généralement d'un manque de précision dans le suivi des flux. En effet, pour des raisons de facilité d'implémentation, ce type d'approche considère des conteneurs d'informations de forte granularité et adaptés au niveau du système d'exploitation (fichier, page mémoire, etc.). Le système n'a donc pas accès aux flux internes des applications entre conteneurs de faible granularité (variables, champs d'objet, etc.). Il est donc amené à sur-approximer les flux d'informations ce qui peut, dans certains cas, conduire à des faux positifs dans le processus de détection des flux illégaux.

1.2.2.2 Contrôle par instrumentation

Une part importante des travaux sur le contrôle dynamique des flux d'informations repose sur une instrumentation des applications. A la différence des moniteurs externes, cette approche permet facilement de suivre les flux d'informations internes aux applications. En revanche, le suivi des flux entre les différentes applications est plus délicat. Il nécessite l'aide d'un mécanisme supplémentaire, par exemple un support matériel ou un moniteur externe. Il est en outre nécessaire d'instrumenter toutes les applications.

Les travaux publiés jusqu'alors se sont essentiellement intéressés à une classe particulière de suivi des flux d'informations : le contrôle de pollution des données ou *taint analysis*. Cette approche s'apparente au contrôle d'intégrité. Le but est de s'assurer que les paramètres des opérations sensibles pour la sécurité, telles que le branchement, les requêtes SQL, l'émission d'une page html comportant des *script*, etc., soient des données «intègres». En effet, dès lors qu'un paramètre d'une de ces opérations est une donnée non-intègre, produite par exemple à

partir de données brutes générées par un utilisateur non privilégié, il existe un risque d'intrusion. Un attaquant peut donc, en tant qu'utilisateur non privilégié, générer des données spécialement créées pour exploiter une vulnérabilité du système et influencer une opération privilégiée. Les exemples de telles attaques sont courants :

- dans le cadre d'un dépassement de tampon ou *buffer overflow*, l'attaquant génère des données dont la taille dépasse la taille du tampon réservé en mémoire pour accueillir ces données. Ces dernières contiennent un code malveillant, généralement appelé *shellcode* car permettant historiquement d'accéder à une console ou *shell*. Elles sont de plus générées de manière à écraser l'adresse de retour d'une fonction sur la pile (*stack overflow*) ou un pointeur de fonction (*heap overflow*) par l'adresse du code malveillant injecté. Dans tous les cas, le code malveillant est exécuté. Il s'agit donc d'influencer une opération de branchement par des données non intègres.
- dans le cadre d'une attaque de type défaut de formatage de chaîne de caractère ou *format string attack*, l'attaquant peut profiter d'une mauvaise utilisation des fonctions de type `printf()` en C ou C++ pour écraser l'adresse de retour d'une fonction. Comme dans le cas précédent, un code malveillant peut être exécuté suite à un branchement influencé par des données non intègres.
- dans le cadre plus général des attaques par injection de code (*code injection*), l'attaquant peut produire des données qui seront interprétées comme des instructions par un interpréteur (par exemple PHP, Perl, etc.). Une classe particulière de ce type d'attaques concerne le langage SQL (*SQL injection*). L'attaquant peut alors générer des données qui lui permettent de forger une requête SQL quelconque. Dans ces différents cas de figure, il s'agit là aussi d'exécuter une opération sensible à partir de données non intègres.
- dans le cadre d'une attaque de type *Cross Site Scripting* (*XSS*), l'attaquant soumet à un site web une page contenant des scripts malveillants en exploitant généralement la naïveté d'un utilisateur via une URL reçue par mail ou sur un site web. Les sites vulnérables ne filtrent pas ce type de requêtes et répondent à l'utilisateur attaqué par une page web contenant le script malicieux. Ce type d'attaque permet d'augmenter les privilèges du script malveillant. Celui-ci étant reçu par l'utilisateur depuis une tierce partie «de confiance», le site web vulnérable, via l'attaque XSS, il est alors lui-même réputé «de confiance» et exécuté. L'attaque est rendue possible du fait du manque de contrôle par le serveur web sur les données qu'il traite en entrée. Des données non intègres peuvent conduire à l'exécution d'une opération sensible, à savoir l'émission d'une page web contenant des *script*.

Dans les faits, dans le cadre d'un fonctionnement normal et sain du système, un certain nombre d'opérations sensibles dépendent des données générées par les utilisateurs en entrée du système. Il n'est donc pas envisageable d'interdire complètement ce type de flux d'informations. Il est en revanche nécessaire de filtrer et contrôler les données en entrée, qui peuvent potentiellement contenir des

attaques, avant qu'elles n'atteignent les opérations sensibles. Ainsi, les requêtes SQL générées par une application web dépendent essentiellement des données entrées par l'utilisateur du système. Il est donc nécessaire de filtrer ces entrées afin de limiter les possibilités offertes à l'attaquant. Celui-ci doit par exemple pouvoir renseigner des champs permettant de construire une requête prédéterminée. En revanche, il ne doit pas pouvoir forger des requêtes quelconques à l'aide de mots du langage SQL tels que `EXEC`, `SELECT`, `INSERT`, etc.

Le contrôle de pollution vise donc essentiellement à vérifier que les données d'entrée du système, considérées comme non-intègres, soient correctement traitées et validées avant d'être passées en paramètre à des opérations sensibles. Afin de contrôler ces flux d'informations, les solutions de contrôle de pollution doivent effectuer les opérations suivantes :

- identifier les différentes **sources** d'informations. Il s'agit des données présentes initialement au sein du système ou des interfaces d'entrée du système susceptibles de produire des données utilisées lors des opérations sensibles. Ces données sont ensuite marquées à l'aide d'un tag de sécurité suivant leur niveau d'intégrité. La politique est généralement réduite à sa plus simple expression : seules les données réputées non-intègres sont marquées comme étant «polluées» (*tainted*), les données non marquées étant supposées intègres.
- identifier les différents **puits** d'informations. Il s'agit des opérations sensibles à proprement parler. Il peut s'agir par exemple des fonctions effectuant des requêtes SQL ou générant des pages web. Ces fonctions ne peuvent être appelées avec des données marquées ou «polluées» en paramètre.
- définir des règles de propagation des marques, ou tags, associées aux données. Généralement, seuls les flux explicites, résultant d'une copie ou d'une transformation des données, sont pris en compte. Les flux implicites résultant des branchements conditionnels sont ignorés. Par exemple, dans le cadre de la prévention des attaques de type *SQL injection*, toutes les chaînes de caractères obtenues à partir de chaînes de caractères marquées doivent elles-mêmes être marquées. La propagation doit également tenir compte du filtrage des données qui agit comme une opération de déclassification ou, plus exactement, une opération d'approbation car il s'agit ici d'un problème d'intégrité. Ainsi, par exemple, une chaîne de caractères marquée sera considérée comme «dépolluée» si elle est soumise à une fonction de validation préalablement définie.

Une implémentation de ce type de système existe dans Perl[10] [Sch00b] et dans Ruby[11] [TH05]. Plusieurs travaux se sont intéressés à généraliser cette approche. Ainsi Madsen propose d'étendre l'approche au niveau OS [Mad00]. Nguyen-Tuong *et al.* [NTGG+05] ainsi que Haldar, Chandra et Franz [HCF05a] proposent de porter cette approche respectivement sur PHP et Java pour protéger les applications web. Halfond, Orso et Manolios proposent d'utiliser une

[10] `http://search.cpan.org/dist/perl/pod/perlsec.pod`
[11] `http://www.ruby-doc.org/docs/ProgrammingRuby/html/taint.html`

approche symétrique de *positive tainting* [HOM06]. A la différence des autres approches, ce sont les données intègres qui sont marquées, les données non marquées étant considérées comme «polluées». Cette approche permet selon les auteurs de diminuer les faux négatifs. Xu, Bhatkar et Sekar [XBS06] ainsi que Lam et Chiuch [LcC06] proposent d'étendre la technique à l'ensemble des applications du système. Leur approche repose sur la compilation des applications à l'aide d'un compilateur adéquat ce qui permet de limiter la surcharge nécessaire pour le suivi des flux d'informations. Le spectre des attaques couvertes est ainsi étendu mais l'approche nécessite de compiler de nouveau l'ensemble des applications et des bibliothèques du système ce qui n'est pas toujours envisageable, notamment pour les applications commerciales dont le code source n'est pas disponible. Qin *et al.* proposent quant à eux un mécanisme d'instrumentation du code binaire optimisé pour limiter la surcharge due au contrôle des flux d'informations, LIFT [QWL+06]. Plus récemment, Clause, Li et Orso se sont intéressés à définir et implémenter un mécanisme générique, flexible et paramétrable, Dytan [CLO07] qui permet de suivre à la fois les flux directs et indirects. Plusieurs approches proposent également de s'appuyer en partie sur un support matériel [SLZD04, CPG+04, CC04, KZZ06, DKK07]. Comme évoqué précédemment, un tel support permet d'optimiser le contrôle des flux mais limite le développement des solutions proposées qui restent pour l'instant au stade expérimental, utilisant des architectures re-programmables (FPGA) ou des simulateurs.

De manière générale, le contrôle de pollution apparaît comme une solution pragmatique permettant de contrer un certain nombre d'attaques. Toutefois, le spectre de couverture se limite aux attaques à l'intégrité, la fuite de données confidentielles n'étant généralement pas traitée. De plus, les politiques sont en général réduites à deux classes de sécurité (données «polluées» ou «non-polluées»). Plusieurs travaux se sont donc intéressés à généraliser cette technique afin de prendre en compte différentes politiques de flux. Ainsi Franz *et al.* [HCF05b, Fra06] proposent de suivre dynamiquement les flux d'informations au sein des applications Java en instrumentant le *bytecode* Java. Les auteurs proposent de suivre les flux entre les champs des objets Java lors de l'invocation des méthodes et des accès aux champs des objets Java. Cette approche permet de s'affranchir de modifier en profondeur la machine virtuelle Java (*JVM*) et n'impose donc pas le choix de la JVM. L'instrumentation est réalisée «à la volée» lors du chargement du *bytecode* des fichiers de classe par le chargeur de classe ou *classloader*. Plusieurs aspects du *bytecode* Java facilitent le contrôle de flux d'informations par instrumentation :

- il s'agit d'un langage intermédiaire, entre le langage de haut niveau qu'est Java et le code binaire interprété par le processeur. Ce langage est fortement typé et permet d'identifier aisément les différents conteneurs d'informations. De nombreux outils permettent la modification «à la volée» du *bytecode* [BHM07]
- la machine virtuelle Java qui interprète le *bytecode* effectue des vérifications qui limitent les attaques par injection de code. En particulier, les accès mémoire sont protégés par le typage statique fort et le contrôle de la taille

des données avant l'écriture dans les tampons. Les attaques de type *buffer overflow* ou *format string* ne sont donc pas possibles, du moins pour les applications Java (la JVM, généralement écrite en C/C++ pouvant être vulnérable à ce type d'attaque). Il est donc plus difficile à un attaquant d'échapper au contrôle de flux.

Les travaux de Yoshihama *et al.* [YYW+07] suivent la même approche et apportent les améliorations suivantes :

- les auteurs proposent de suivre également les flux d'informations entre les variables locales, à l'intérieur des méthodes ;
- les auteurs proposent un mécanisme permettant le marquage de données provenant de sources extérieures, principalement des Systèmes de Gestion de Base de Données via APM4JDBC[12], un framework permettant d'intercepter les appels aux fonctions de l'API JDBC, elle même utilisée comme interface entre une application Java et un SGBD ;
- les auteurs proposent un mécanisme de déclassification des flux d'informations à travers une API de méthodes (par exemple des méthodes de chiffrement ou de génération de résumés cryptographiques).

Toutefois, la solution proposée présente un certain nombre de limites :

- seules les applications web s'exécutant sur un conteneur de *servlet* J2EE (les auteurs utilisent Apache Tomcat) sont instrumentées. Le code des objets Java de l'environnement d'exécution JRE et du conteneur de *servlet* n'est pas instrumenté et par conséquent le suivi des flux d'informations est limité à l'application web.
- l'interface avec les conteneurs d'informations externes (fichiers, bases de données, etc.) est une fonctionnalité intéressante mais le mécanisme proposé repose sur la définition explicite et statique de la politique de marquage pour ces conteneurs, sous la forme d'un fichier de signatures de méthodes réalisant l'interface. Les auteurs ne proposent pas d'interface avec un système global de suivi des flux d'informations, qui permet de suivre les flux d'informations externes à Java, réalisés par exemple par d'autres applications non-Java s'exécutant en parallèle sur le même système.

Une des raisons expliquant le peu d'engouement jusqu'alors pour les méthodes dynamiques de contrôle des flux d'informations réside dans la difficulté de suivre les flux indirects, issus de l'évaluation des expressions lors des branchements conditionnels. En particulier, les flux d'informations résultant de l'absence de modification dans une des branches sont indécelables par un moniteur de la classe EM, qui ne résonne que sur la trace courante. De tels flux d'informations sont appelés **flux d'informations indirects implicites** par opposition aux **flux d'informations indirects explicites** : les flux indirectes explicites résultent d'une modification d'un conteneur d'information dans une branche d'une structure conditionnelle tandis que les flux indirects implicites naissent paradoxalement de l'absence de modification de conteneurs lors d'un branchement conditionnel. Le pseudo-code suivant illustre ce principe :

```
bool a,b;
```

[12] `http://www.alphaworks.ibm.com/tech/apm4jdbc`

```
...
b=false;
if (a)
        b=true;
else
        {}
```

Lors du test, si la valeur de `a` est «vraie» alors la branche `b=true` est exécutée. L'affectation de `true` à `b` génère un flux indirect explicite de `a` vers `b`. Si la valeur de `a` est «faux» lors du test, la branche qui est choisie n'exécute aucune opération et donc ne modifie aucune variable. Néanmoins, il existe un flux d'informations indirect implicite de `a` vers `b` car un attaquant capable d'observer que la valeur de `b` n'a pas été modifiée peut inférer la valeur de `a`. Ce flux ne résulte pas des opérations exécutées dans la trace mais de celles pouvant l'être dans les branches non-exécutées. Un certain nombre de travaux se sont donc intéressés à contourner cette limitation. Dans tous les cas, le moniteur doit disposer d'informations supplémentaires qui lui permettent de raisonner sur les branches non-exécutées du programme.

Ainsi, Le Guernic *et al.* [GBJS06] proposent un mécanisme de suivi dynamique des flux d'informations qui, en ayant accès au code source du programme, permet de prendre en compte les flux implicites. Les auteurs prouvent que leur mécanisme permet de vérifier une propriété de «non-interférence de trace» [GJ05], garantissant l'absence de flux d'informations dans l'exécution courante. Venkatakrishnan *et al.* [VXDS06] proposent une technique similaire dans laquelle une analyse statique réalisée avant l'exécution permet d'inclure, pour chacune des branches du programme, des informations concernant les branches non-exécutées. De même Shroff, Smith et Thober [SST07] proposent un mécanisme dynamique de suivi des flux d'informations qui s'appuie sur des données définissant les dépendances entre les modifications de variables et les conditions de branchement. Ces dépendances peuvent être spécifiées explicitement *a priori*, à l'aide d'une analyse statique, ou découvertes au fur et à mesure de l'exécution. Ces travaux définissent les principes des méthodes hybrides de contrôle des flux d'informations, utilisant conjointement une analyse statique et dynamique, en démontrant des propriétés dérivées de la non-interférence sur des langages théoriques. Elles ne proposent pas en revanche d'implémentation pour des langages usuels.

Les travaux récents de Chandra et Franz [CF07] ainsi que ceux de Nair *et al.* [NSCT07] proposent quant à eux des implémentations de mécanismes de contrôle de flux sous Java reposant sur l'approche hybride. Chandra et Franz complètent ainsi leurs travaux précédents [HCF05b, HCF05a, Fra06] sur le contrôle dynamique de flux d'informations par instrumentation du *bytecode* Java. Leur nouveau mécanisme permet de suivre les flux d'informations à la fois entre les champs d'objets et les variables locales. Une analyse statique effectuée avant le chargement de chaque classe permet également de traiter les flux indirects. Nair *et al.* suivent une approche similaire mais leur mécanisme est implémenté au niveau de la JVM qu'ils ont en partie modifiée. Ces deux approches

constituent des exemples intéressants d'implémentation de méthodes hybrides. Toutefois, les auteurs de ces deux travaux pointent une limitation pratique importante lors de l'analyse des branches non exécutées. En effet, l'ensemble des objets et des méthodes utilisés dans les branches non-exécutées ne sont pas toujours identifiables lors de l'analyse statique. S'il s'agit de variables locales, la méthode est envisageable en pratique. En revanche, dès lors qu'il s'agit d'instance d'objet créée dynamiquement, l'analyse est plus délicate. L'analyse étant réalisée lors du chargement de la classe ou classe par classe, son périmètre est nécessairement limité. Par conséquent, les auteurs sont amenés à faire un certain nombre d'hypothèses restrictives susceptibles d'engendrer des faux positifs. Nous pensons donc qu'en pratique la prise en compte des flux indirects implicites n'est pas souhaitable car générant potentiellement des faux positifs, la diminution des faux négatifs restant à démontrer en pratique. Les méthodes hybrides peuvent cependant contribuer à l'optimisation du processus de détection en limitant le surcoût de l'analyse dynamique.

1.2.2.3 Bilan sur le contrôle dynamique des flux d'informations

Le contrôle dynamique des flux d'informations repose sur l'observation d'une trace correspondant à l'exécution courante du programme surveillé. Deux techniques principales sont utilisées :

- un moniteur externe, généralement implémenté au nivau du système d'exploitation, peut surveiller les flux entre les différentes applications. Cette solution permet de surveiller l'intégralité du système mais échoue parfois à suivre correctement les flux internes aux applications en raison de l'approximation faite sur le comportement interne des applications.
- une instrumentation du code de l'application peut surveiller en détail les flux internes à l'application. La surveillance de l'ensemble des flux entre les différentes applications est en revanche plus délicate avec cette méthode.

Nous proposons donc de combiner les deux approches. L'instrumentation de certaines applications complexes permet de raffiner la vue globale du moniteur externe implémenté au sein du système d'exploitation. Dans le cadre des langages utilisant un *bytecode* interprété par une machine virtuelle, nous pensons que l'instrumentation de ce langage intermédiaire «à la volée» constitue une solution intéressante. En effet, elle ne nécessite pas de disposer du code source des applications ni de modifier en profondeur la machine virtuelle. Il s'agit donc d'une solution assurant un maximum de compatibilité avec les logiciels existants. L'approche dynamique présente un certain nombre d'avantages sur l'analyse statique :

- elle permet de s'appuyer sur une spécification de la politique de flux définie par l'utilisateur ou le responsable de la sécurité lors du déploiement de l'application et non sur une spécification «figée» lors de la compilation. Pour la plupart des applications, la définition «figée» de la politique de sécurité n'est pas une hypothèse réaliste car celle-ci dépend des aspects contextuels liés à l'exécution.
- elle permet de valider ou rejeter une exécution d'une application. Il n'est

pas nécessaire que toutes les traces possibles du programme surveillé vérifient la politique de flux. Seules celles effectivement exécutées doivent vérifier la politique. Il est donc possible d'utiliser des programmes qui n'ont pu être validés comme étant «sûrs» par une analyse statique, ce qui est généralement le cas.
- elle est parfois plus précise dans la détermination des flux d'informations effectivement réalisés lors de l'exécution. Les approches statiques raisonnent sur une sur-approximation de l'ensemble des flux d'informations possibles et peuvent donc être amenées à rejeter un nombre important de programmes (faux positifs).

Cette approche n'est cependant pas sans inconvénients :
- le suivi dynamique des flux d'informations implique nécessairement un sur-coût lors de l'exécution ;
- le suivi dynamique nécessite l'ajout de composants logiciels supplémentaires, implémentés au niveau de l'environnement d'exécution ou au sein même des applications. Ces composants peuvent eux-mêmes être vulnérables et être contournés ou pire utilisés afin de pénétrer le système surveillé. L'ensemble des composants de confiance de la TCB est donc augmenté.
- l'analyse d'une seule exécution ne permet pas de déterminer complètement l'ensemble des flux d'informations, donc potentiellement des intrusions. Certains faux négatifs peuvent subsister si l'attaquant utilise un flux d'informations indirect implicite. Cependant, il n'existe pas pour l'instant à notre connaissance d'attaques utilisant de tels flux d'informations.

1.2.3 Bilan général sur le contrôle des flux d'informations

Le contrôle des flux d'informations au sein des applications permet de suivre précisément les flux d'informations internes des applications. Il s'agit donc essentiellement de méthodes de «niveau langage» qui peuvent compléter des approches de «niveau OS». Les approches de contrôle statique, à l'aide par exemple d'un système de types de sécurité, et les approches de suivi dynamique des flux d'informations, à l'aide d'une instrumentation des applications, constituent les deux approches majeures et complémentaires du domaine. Ces deux approches, présentées en détail dans cette section, possèdent chacune des avantages et des inconvénients. Comme le fait remarquer Smith [Smi07], la comparaison des systèmes de contrôle des flux d'informations doit tenir compte du but recherché et du scénario d'utilisation. Ainsi, les analyses statiques et dynamiques ne répondent pas aux mêmes objectifs.

L'analyse statique a pour but le déploiement d'applications réputées «sûres», dont toutes les exécutions possibles vérifient une politique de flux donnée. Deux scénarios sont envisageables :

1. Le premier scénario correspond à la certification de programme. Dans ce scénario, initialement imaginé par Denning [DD77], le concepteur du logiciel fournit le code source de son application à une tierce partie chargée de certifier que le programme en question vérifie une politique de flux

donnée, qui constitue une spécification des exigences de sécurité. Plutôt que de se fier à l'analyse manuelle du code de l'application par un expert ou à des techniques de test, nécessairement non exhaustifs, l'organisme certifieur effectue une analyse statique qui lui permet de prouver que le programme vérifie les propriétés de non-interférence qui constituent la politique de flux. L'organisme peut alors délivrer un certificat à l'utilisateur final du logiciel qui lui garantit l'absence de fuite d'information, le programme étant exempt de vulnérabilité permettant de générer des flux d'informations interdits. Si une seule exécution ne vérifie pas la politique, le programme est rejeté. Cette approche est *a priori* intéressante car elle permet de s'assurer avant toute exécution que la politique de flux ne sera pas violée. De plus, le résultat ne dépend que de la politique spécifiée et des propriétés intrinsèques du logiciel. Malheureusement, peu de programmes «fonctionnels» passent ce test avec succès. Beaucoup de programmes sont rejetés, l'approche étant très restrictive comme nous l'avons évoqué dans les sections précédentes. Dès lors, ce scénario ne peut s'appliquer qu'à des cas particuliers de systèmes simples pour lesquels les exigences de sécurité sont fortes et la politique de flux est statique (par exemple de petits systèmes embarqués, des environnements JavaCard, etc.) Pour la majorité des systèmes, comprenant de nombreux composants logiciels interagissant entre-eux, parfois dans un environnement ouvert, l'approche est inapplicable en pratique.

2. Le deuxième scénario vise à aider le développeur dans sa tâche en l'informant des risques de fuites ou de modifications inappropriées d'informations dans l'application qu'il conçoit. Cette approche, inspirée des systèmes de types, suppose que le développeur spécifie les classes de sécurité associées à chaque variable, en plus du code fonctionnel de l'application. Le système repose sur une analyse statique des propriétés du programme mais égalemment sur des annotations permettant de déclassifier l'information : le développeur peut spécifier des exceptions lorsqu'il le juge nécessaire. La confiance dans le jugement apportée par l'analyse repose donc en partie sur le développeur. Celui-ci peut, y compris par inadvertance, spécifier un nombre trop important d'exceptions permettant par la suite à un attaquant d'effectuer des flux d'informations illégaux. Cette approche implique donc une confiance plus faible dans les programmes acceptés, qui ne vérifient pas, au sens strict, de propriété de non-interférence, du fait des exceptions. En revanche, cette approche plus souple accepte un nombre plus important de programmes. Ce scénario contribue à l'augmentation de la sécurité en incitant le développeur à raisonner dès la conception sur les problèmes de sécurité liés aux flux d'informations. Il s'agit d'une méthode amont qui peut être assimilée aux techniques de génie logiciel dont le but est d'augmenter la qualité des logiciels produits. Le spectre d'utilisation est donc plus grand du fait de la diminution des faux positifs. Il n'est cependant applicable qu'aux applications nouvellement développées ou partiellement réécrites. De plus, les aspect contextuels, en particulier

de la politique de flux, doivent faire l'objet de tests explicitement spécifiés par le développeur, ce qui est parfois difficilement envisageable.

L'analyse dynamique, quant à elle, vérifie la légalité d'une seule exécution d'un programme donné. Cette méthode permet donc l'utilisation de programmes dont certaines exécutions violent la politique de flux. Le recours à de tels logiciels est souvent nécessaire :

- la réécriture de l'ensemble des applications du système est une hypothèse trop restrictive. Il est nécessaire d'utiliser, en partie, des logiciels non vérifiés ou qui n'ont pu être développés à l'aide de techniques de vérification de flux.
- un certain nombre de programmes possèdent intrinsèquement des limitations et de ce fait certaines de leurs exécutions peuvent violer la politique de flux. Développer de nouveau de tels logiciels pour éliminer ces limitations n'est pas toujours aisé ou peut se révéler très coûteux.

L'utilisation de programmes non validés par l'analyse statique est donc souvent nécessaire. En outre, l'hypothèse est généralement faite que ces programmes vérifient la politique de flux lors d'une utilisation «normale et saine» mais que des exécutions résultant d'une attaque par des utilisateurs malveillants peuvent conduire à des intrusions, c'est-à-dire à des violations de la politique de flux d'informations. L'analyse dynamique permet alors, en vérifiant chacune des exécutions, de détecter l'occurrence de ces intrusions, d'alerter l'utilisateur ou le responsable de la sécurité et éventuellement de terminer l'exécution en cours. L'analyse portant sur une seule trace, le nombre de programmes acceptés (temporairement car une exécution violant la politique est *a priori* toujours possible) est plus important. Le nombre de faux positifs est limité mais il ne peut en pratique être nul car la détection précise des flux d'informations à partir de l'observation d'une seule trace est impossible. De même, l'approche n'est pas complète du fait des flux d'informations indirects implicites. L'hypothèse peut cependant être faite que le nombre d'intrusions utilisant ce type de flux reste, pour l'instant, négligeable. Ce scénario de détection d'intrusions, présenté en section 1.1, correspond donc au besoin exprimé en introduction. Nous proposons ainsi un modèle et une implémentation de détecteur d'intrusions paramétré par la politique reposant sur le contrôle dynamique des flux d'informations à différents niveaux. L'utilisation de méthodes statiques pour certains logiciels peut toutefois compléter avantageusement cette approche de contrôle dynamique.

1.3 Bilan de l'état de l'art

Nous souhaitons détecter les intrusions sur un système utilisé classiquement pour le déploiement des applications web. Nous avons vu en section 1.1 que les solutions de détection d'intrusions «classiques» présentent un certain nombre de limites :

- l'approche de détection par signatures est nécessairement incomplète car elle dépend d'une connaissance *a priori* des attaques sous la forme d'une base de signatures d'attaques ;

– l'approche comportementale traditionnelle doit faire face à un taux important de faux positifs et à la difficulté de paramétrage du profil de référence.

Une forme particulière de détection comportementale, celle dite paramétrée par la politique, nous paraît la plus prometteuse. Cette approche s'appuie uniquement sur la définition de la politique de sécurité et sur la vérification de conditions logiques découlant de cette politique. Nous nous inspirons en particulier des travaux de Jacob Zimmermann sur Blare, un détecteur d'intrusions paramétré par la politique de sécurité assurant le suivi des flux d'informations au niveau du système d'exploitation.

Les résultats de Blare sont prometteurs mais ce détecteur est limité lorsqu'il s'agit de discerner les flux d'informations internes aux applications. Nous souhaitons donc nous inspirer des méthodes de suivi dynamique des flux d'informations, présentées en section 1.2, pour compléter le suivi des flux d'informations réalisé par Blare.

Nous proposons donc :

- une modélisation formelle d'IDS reposant principalement sur le suivi de flux d'informations ;
- une architecture générique d'IDS permettant d'implémenter une solution réalisant le suivi des flux d'informations à plusieurs niveaux de granularité ;
- un prototype implémentant l'architecture proposée au travers de deux mécanismes :
  - une version modifiée de Blare permettant le suivi global des flux d'informations au sein d'un système d'exploitation ;
  - un mécanisme de suivi dynamique des flux d'informations internes à une application Java, JBlare, qui collabore avec Blare.

Nous exposons notre modèle de détection au chapitre 2. Nous présentons ensuite l'architecture générique correspondante ainsi que l'implémentation d'une solution de détection d'intrusions par contrôle collaboratif des flux d'informations.

Chapitre 2

Proposition d'un modèle de détection d'intrusions

Nous nous intéressons dans nos travaux aux politiques de flux d'informations. Nous entendons par flux d'informations une relation de dépendance dans laquelle un ou plusieurs éléments d'information transitent des sources vers les cibles. Une politique de flux d'informations précise quels sont les flux d'informations autorisés pour un système donné. Définir une telle politique suppose donc d'une part d'identifier les différentes sources d'informations du système et d'autre part d'identifier les destinations légales pour les différents types d'informations. Afin de suivre les flux d'informations au sein d'un système, nous distinguons clairement dans notre modèle les **contenus** des **conteneurs** d'informations. Les premiers représentent l'information à proprement parler, c'est-à-dire les données présentes au sein du système (par exemple, les mots de passe des utilisateurs ou la valeur d'une variable). Les seconds représentent le lieu où ces données résident (par exemple, les fichiers ou les variables). Lors d'une exécution, il est possible d'observer des flux d'informations : les contenus sont propagés entre conteneurs après avoir été éventuellement modifiés et combinés entre eux. Par exemple, lors de l'exécution de la pseudo instruction $c = a + b$, le contenu initial de la variable (conteneur) a est combiné avec celui de la variable b et le résultat de cette combinaison forme le nouveau contenu de la variable c. Cette situation correspond donc à un flux d'informations dont les sources sont a et b et la destination c. Il est possible de définir l'état d'un système par un ensemble de relations entre les contenus et les conteneurs du système. Les flux d'informations générés par l'exécution de commandes modifient ainsi l'état du système, ce qui se traduit par des modifications de l'ensemble des relations contenus/conteneurs.

Nous proposons de suivre dynamiquement les flux d'informations du système et d'en vérifier la légalité, au regard d'une politique de flux d'informations préalablement spécifiée, à l'aide de tags de sécurité attachés à chaque conteneur. Nous montrons finalement que la loi de propagation de ces tags ainsi que la règle de légalité des flux d'informations déterminée à partir de ces tags permet de dé-

tecter dynamiquement les intrusions, c'est-à-dire les violations de la politique de flux d'informations spécifiée.

Nous présentons dans un premier temps plus en détail la notion de conteneurs, de contenus et de flux d'informations dans la section 2.1. Nous définissons ensuite, dans la section 2.2, notre modèle de politique de flux d'informations, qui repose sur un ensemble d'associations contenus/conteneurs autorisées. Nous présentons en section 2.3 l'évolution de l'état du système en termes d'évolution des conteneurs, des contenus et des tags de sécurité associés. Dans la section 2.4, nous montrons finalement à l'aide d'un théorème de détection que l'évolution des tags de sécurité permet de détecter la légalité des flux d'informations observés au regard d'une politique de flux d'informations préalablement spécifiée.

2.1 Contenus, conteneurs et flux d'informations

2.1.1 Conteneurs d'informations

Nous nous intéressons aux flux d'informations entre conteneurs. Nous entendons par conteneur un ensemble structuré et identifiable de données permettant à un utilisateur ou un programme informatique d'accéder aux données à proprement parler. Suivant la granularité de l'observation, les conteneurs d'informations peuvent être des fichiers, des pages mémoires, des variables, etc. Nous supposons que les conteneurs du système étudié forment un ensemble dénombrable. Nous supposons de plus que cet ensemble est fini et nous notons N le nombre maximal de conteneurs du système. Cette hypothèse nous paraît réaliste car les systèmes étudiés sont par nature des systèmes à mémoire finie. Le nombre de conteneurs est donc limité, ne serait-ce que par la quantité de mémoire adressable. Cette limite est parfois exprimée explicitement. Pour le système de fichier *ext3* utilisé par le système d'exploitation *Linux*, le nombre de fichiers par système de fichiers est ainsi limité par le nombre maximal de blocs[1] soit 2^{31} ce qui correspond à environ 2 milliards de fichiers. Cette limite théorique n'est toutefois jamais atteinte car en pratique le nombre de fichiers est limité par la taille du système divisée par la taille moyenne des fichiers. Nous modélisons donc l'ensemble des conteneurs du système susceptibles d'apparaître dans un flux d'informations :

$$\mathbb{C} = \{c_1, \ldots, c_N\}$$

La création et la destruction de conteneurs ne sont pas explicitement prises en compte dans notre modèle. Nous montrons toutefois en section 2.2 comment traiter les cas de création et de destruction de conteneurs. $\mathbb{C}$ peut alors être considéré comme fini s'il représente l'ensemble maximal des conteneurs que le système peut gérer.

[1] La limite est en réalité le minimum entre le nombre maximale de blocs et $(\frac{V}{2})^{13}$, V étant la capacité de stockage du système de fichiers en octet. Pour des partitions de grandes capacités telles que le permettent les disques durs actuels, le nombre maximal de blocs détermine souvent le nombre maximal de fichiers

Les systèmes informatiques étant rarement déconnectés du monde qui les entoure, il est nécessaire de prendre en compte les flux d'informations qui entrent ou sortent du système. Ces flux d'informations peuvent correspondre à des échanges de données entre différents systèmes reliés par un réseau informatique ou à des échanges de données avec des utilisateurs physiques. Afin de modéliser ces flux d'informations, nous considérons des **conteneurs interface** qui modélisent les interfaces d'entrée/sortie du système. Nous considérons que tout conteneur interface c_u appartient à l'ensemble des conteneurs : $c_u \in \mathbb{C}$. A l'échelle du système d'exploitation, les conteneurs interface peuvent ainsi modéliser les terminaux en mode caractère (tty) ou les *socket* réseaux. A l'échelle d'un programme informatique, les conteneurs interface peuvent modéliser les appels aux fonctions de librairies externes ou les appels système. Les flux d'informations d'entrée/sortie sont alors interprétés en termes de lecture/écriture sur les conteneurs interface. Par exemple, un utilisateur utilisant son clavier génère un flux d'informations vers la mémoire d'un processus. Ce flux d'informations est modélisé par la lecture du conteneur interface correspondant au terminal de l'utilisateur connecté. Inversement, l'information transmise à l'utilisateur via son écran est assimilable à une écriture dans son conteneur interface.

2.1.2 Contenus

Afin d'observer les flux d'informations du système, nous proposons de suivre l'évolution des contenus des différents conteneurs du système. Les données du système étant exprimées sous formes binaires, un suivi précis de chaque contenu suppose potentiellement de s'intéresser à l'évolution de chaque bit d'information, ce qui s'avère inconcevable en pratique. Nous proposons en revanche de nous intéresser à l'origine des contenus du système. Nous considérons que les données du système ont été générées à partir de deux types de sources :

1. les contenus initiaux des différents conteneurs du système ;
2. les informations provenant de l'extérieur du système et transitant par les conteneurs interface.

Le premier type de source correspond aux données du système pris dans son état initial. Nous ne faisons aucune hypothèse sur cet état initial, en particulier sur l'origine des différents contenus du système dans cet état. En pratique, l'état initial correspond à l'initialisation du mécanisme de détection, par exemple après l'installation du système ou lors de la séquence de démarrage du système. Nous considérons donc, dans cet état initial, que chaque contenu provient du conteneur qui le contient et nous associons à chaque conteneur c_n une unique **information atomique** initiale i_n. Nous n'exprimons en fait que l'origine des données, à partir de l'état initial, et nous considérons que les contenus n'ont qu'une seule origine dans l'état initial. Notre but est de modéliser les évolutions du système à partir de l'état initial, les flux d'informations antérieurs à cet état ne sont donc pas pris en compte par le modèle du système.

Nous supposons également que deux conteneurs distinct c et c' contiennent, dans l'état initial, des informations atomiques initiales distinctes i et i'. Bien

que l'information présente dans ces deux conteneurs puisse se révéler identique en pratique, l'origine des contenus, pris dans l'état initial, n'en demeure pas moins distinct. Nous considérons que la prise en compte de la teneur effective des données est du ressort de la politique de flux d'informations.

Le deuxième type de source correspond aux flux d'informations externe vers les conteneurs du système. Nous ne faisons là aussi aucune hypothèse sur l'extérieur du système. Nous considérons donc que, pour chaque état du système, l'information transitant par un conteneur interface c_u provient uniquement de ce conteneur. Nous associons donc à chaque conteneur interface c_u une unique **information atomique** i_u. Nous considérons également que deux conteneurs interface distincts i_u et i_v produisent des **informations atomiques** distinctes c_u et c_v. Nous ne prenons pas explicitement en compte la notion de sujet, c'est-à-dire d'utilisateur ou de processus s'exécutant pour le compte d'un utilisateur, dans notre modèle contrairement aux modèles traditionnels de contrôle d'accès. Nous distinguons uniquement les entités actives du système à travers les différents conteneurs qu'ils accèdent. Il est notamment possible, en s'appuyant sur un mécanisme d'authentification, d'attribuer dynamiquement un conteneur interface à chaque utilisateur authentifié. Il est alors possible de distinguer les actions des différents utilisateurs lors des accès aux conteneurs interface.

Les conteneurs du système formant un ensemble fini, il est possible de définir un ensemble fini des **informations atomiques** :

$$\mathbb{I} = \{i_1, \ldots, i_N\}$$

Cet ensemble correspond aux différentes origines possibles des contenus du système.

Lorsque l'état du système évolue suite à l'exécution de commandes générant des flux d'informations, les contenus des différents conteneurs évoluent. Comme nous l'avons précisé précédemment, nous voulons suivre les différents flux d'informations en exprimant l'origine des différents contenus à partir de l'état initial. Nous modélisons donc chaque contenu par un ensemble d'informations atomiques. Cet ensemble représente l'origine des flux d'informations ayant permis de générer le contenu en question. Il dénote également l'ensemble des informations atomiques dont dépend le contenu. Le contenu initial de chaque conteneur c_n est donc un singleton $\{i_n\}$. Par la suite, nous notons $I = \{i_1, \ldots, i_n\} \subset \mathbb{I}$ le contenu courant du conteneur c, ce qui signifie que l'information actuellement mémorisé dans c a été générée à partir de celle initialement présente dans les conteneur $c_1, \ldots c_n$. L'exemple 1 et la figure 2.1 illustrent l'état initial d'un système décrit d'après notre modèle.

Exemple 1. *Considérons par exemple un système simple dans lequel des utilisateurs peuvent accéder aux contenus de fichiers à travers une interface utilisateur (par exemple un terminal texte). Supposons que dans l'état initial, lorsque le processus de détection d'intrusions est lancé, le système soit seulement composé de trois fichiers $c_j, j \in \{1, 2, 3\}$ et d'un seul utilisateur authentifié u. D'après notre modèle, l'état initial du système sera alors caractérisé par :*

- *3 conteneurs $c_{j\in\{1,2,3\}}$ correspondant aux trois fichiers ;*

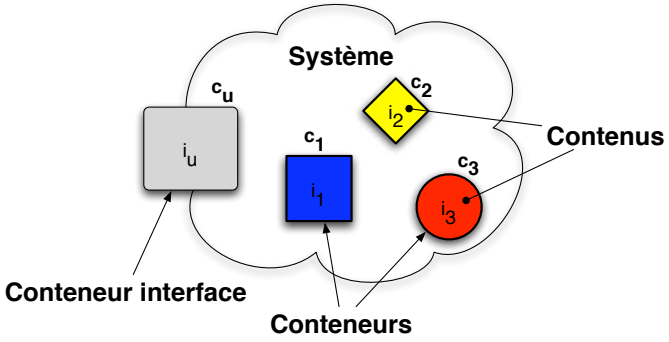

FIG. 2.1 – Exemple de modélisation de l'état initial d'un système

- *3 informations atomiques initiales $i_{j\in\{1,2,3\}}$ correspondant aux trois contenus initiaux des trois fichiers ;*
- *un conteneur interface c_u modélisant l'interface de sortie de l'utilisateur,*
- *une information atomique i_u associée à c_u et modélisant l'information que l'utilisateur u peut générer via son interface.*

2.1.3 Commande, trace et flux d'informations

2.1.3.1 Flux d'informations

Les flux d'informations expriment une relation de dépendance entre des données sources et des conteneurs cibles ou destinations. Denning [DD77] considère par exemple qu'il existe un flux d'informations d'un conteneur x vers un conteneur y dès lors qu'une donnée initialement stockée dans x est transférée dans y ou que cette donnée a été utilisée pour dériver une information elle-même transférée dans y. L'auteur note cette relation $x \Rightarrow y$ où x et y sont des conteneurs. Le modèle proposé par Denning prend à la fois en compte les flux d'informations résultant d'une copie d'information, tel l'assignation d'une variable ($a = b$) et les relations de dépendance plus complexes, résultant par exemple d'un calcul sur les données sources. Considérons par exemple une fonction *sqrt* permettant de calculer la racine carrée, l'exécution de l'instruction $a = sqrt(b)$ génère aussi un flux d'informations $b \Rightarrow a$.

Nous proposons de reprendre la définition donnée par Denning, en distinguant les contenus des conteneurs.

Définition 2.1.1 (flux d'informations)**.** *Nous considérons qu'il existe un flux d'informations d'un contenu $I = \{i_1, \ldots, i_n\}$ vers un conteneur c dès lors que ce contenu I est transféré vers c ou utilisé pour générer une information qui est elle-même transférée dans c. Nous notons cette relation de dépendance $I \Rightarrow c$. Nous appelons un tel flux d'informations, composé d'un seul contenu source et d'un unique conteneur destination, un flux d'informations unitaire. Un flux d'informations peut également mettre en jeu plusieurs sources et plusieurs cibles. Nous proposons de traiter les différents cas de la manière suivante :*

- *lorsque le flux d'informations comporte plusieurs cibles, nous considérons que ce flux d'informations est équivalent à plusieurs flux d'informations unitaires (autant que de conteneurs cibles) réalisés en parallèle. Par exemple, un flux d'informations d'un contenu I vers les conteneurs c_1 et c_2 est équivalent à la réalisation en parallèle des flux d'informations $I \Rightarrow c_1$ et $I \Rightarrow c_2$.*
- *lorsque le flux d'informations comporte plusieurs contenus sources, nous considérons que ce flux d'informations est équivalent à un flux d'informations unitaire de l'union des contenus sources vers le conteneur destination. Par exemple, un flux d'informations des contenus I_1 et I_2 vers c est équivalent au flux d'informations $I_1 \cup I_2 \Rightarrow c$*

Cette dernière interprétation est cohérente avec la notion de contenus présentée précédemment dans la sous-section 2.1.2. Nous considérons en effet qu'un contenu est un ensemble d'informations atomiques exprimant l'origine de l'information. L'origine d'une donnée générée à partir de plusieurs contenus est exprimée à partir de l'origine de tous les contenus ayant servi à générer cette donnée. Par exemple, l'instruction $c_1 = c_2 + c_3$ génère un flux d'informations des contenus de c_2 et c_3 vers c_1. Supposons qu'avant l'exécution de cette instruction c_2 contienne une unique information atomique initiale i_2 et que de même c_3 contienne i_3. L'exécution génère alors un flux d'informations $\{i_2, i_3\} \Rightarrow c_1$. Le nouveau contenu de c_1, $I_1 = \{i_2, i_3\}$ est obtenu à partir de l'union des contenus sources du flux d'informations : $I_1 = \{i_2\} \cup \{i_3\}$.

Nous considérons enfin que le contenu de chaque conteneur forme un tout et que par conséquent toutes les parties de ce contenu ont la même origine. Dans le cas d'un flux d'informations comportant plusieurs sources, c'est-à-dire un flux d'informations impliquant la lecture de plusieurs contenus différents, nous supposons que le contenu final généré par ce flux d'informations dépend de tous les contenus lus. Toute partie de ce contenu est donc vue comme une agrégation de tous les contenus lus. Dans l'exemple précédent, le nouveau contenu de c_1 dépend à la fois de i_2 et de i_3. Supposons que l'information contenue dans c_1 soit un entier codé sur un octet. Considérons maintenant l'exécution successive des instructions suivantes qui permettent d'extraire successivement les quatre bits de poids faible puis les quatre bits de poids fort de c_1 : $c_{1'} = c_1$ AND 15, $c_{1''} = c_1$ AND 240. L'exécution de ces instructions génère successivement les flux d'informations $\{i_2, i_3\} \Rightarrow c_{1'}$ et $\{i_2, i_3\} \Rightarrow c_{1''}$.

Cette hypothèse peut sembler peu réaliste, notamment au niveau du système d'exploitation. Il est par exemple possible d'accéder indépendamment à chaque ligne d'un fichier texte. Nous pouvons cependant noter que notre modèle n'im-

pose aucune granularité quant à la taille des conteneurs. Il est donc possible de considérer plusieurs sous-parties d'un même fichier comme des conteneurs distincts. La taille des conteneurs est en fait déterminée par le niveau de granularité de la politique de flux d'informations. Il convient de considérer séparément les différentes sous-parties d'un contenu seulement si la politique d'intégrité/confidentialité distingue ces contenus en termes d'autorisation. Ce cas peu fréquent correspond par exemple aux fichiers des bases de données. Il est également possible d'envisager plusieurs niveaux de modélisation comme nous le verrons dans le chapitre suivant.

2.1.3.2 Commande et flux d'informations élémentaire

Nous proposons de modéliser le système et son évolution par un automate à états finis déterministe.

Définition 2.1.2 (Automate système)**.**
L'évolution du système est modélisé par un automate :

$$A = (S, \Sigma, t, s_0, Q)$$

avec :
- S *l'ensemble des états du système. Nous précisons par la suite ce que nous entendons par état du système. L'ensemble des conteneurs étant supposé fini, nous considérons ici qu'il existe un nombre fini d'états que le système peut atteindre.*
- Σ *l'ensemble des commandes qui peuvent être exécutées et observées sur le système et qui font transiter le système d'un état à un autre.*
- t *la fonction de transition qui décrit l'évolution du système en fonction des commandes exécutées.*
- s_0 *l'état initial du système.*
- Q *l'ensemble des états terminaux du système.*

L'état initial est l'état du système lors de l'initialisation du mécanisme de détection. Comme décrit précédemment, il est caractérisé par la propriété suivante : le contenu de chaque conteneur c_n est un singleton $\{i_n\}$.

Les commandes désignent les opération élémentaires qui peuvent être exécutées sur le système. Suivant le niveau de la modélisation, les commandes peuvent désigner les appels système au niveau du système d'exploitation, les appels de fonctions ou de méthodes, les instructions sur les variables d'un programme, etc. Aucune hypothèse n'est faite sur la nature exacte des commandes et le traitement effectué sur les données. Nous supposons seulement que les commandes sont des événements observables et que pour toute commande $\sigma \in \Sigma$ susceptible de s'exécuter sur le système, il est possible d'identifier l'ensemble C_σ^R des conteneurs accédés en lecture et l'ensemble C_σ^W des conteneurs accédés en écriture. Nous faisons ensuite l'hypothèse que chaque commande génère un flux d'informations de l'ensemble des contenus lus vers chaque conteneur accédé en écriture. Plus précisément, nous considérons que suite à l'exécution d'une

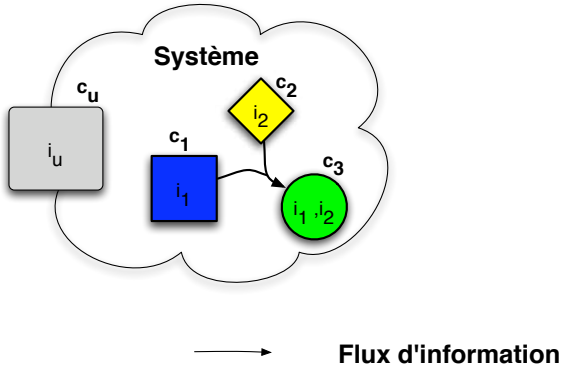

FIG. 2.2 – Exemple d'un flux d'informations comportant plusieurs sources

commande, seul le contenu des conteneurs accédés en écriture est modifié et que le nouveau contenu de chacun des conteneurs de C_σ^W a été généré à partir des contenus de tous les conteneurs de C_σ^R. Nous associons donc à chaque exécution de commande un flux d'informations que nous appelons flux d'informations élémentaire. L'exemple suivant illustre ce concept :

Exemple 2. *Soit une commande σ effectuant un accès en lecture sur le fichier c_1, un accès en lecture sur le fichier c_2 et un accès en écriture sur le fichier c_3. Nous considérons uniquement le résultat de l'exécution de cette commande en termes de flux d'informations. Supposons que cette commande soit exécutée dans l'état initial. Soit $\{i_1\}$ le contenu initial de c_1 et $\{i_2\}$ celui de c_2. Exprimer l'origine du contenu de c_3 dans l'état atteint après l'exécution de la commande permet de prendre en compte le flux d'informations de i_1 et i_2 vers c_3. L'origine des données contenues dans c_3 à l'issu de ce flux d'informations est représentée par les informations atomiques initiales qui ont été lues. Le contenu de c_3 après l'exécution de σ sera donc ici $\{i_1, i_2\}$. Cela signifie que le contenu de c_3 dans l'état final a été obtenu à partir de i_1 et i_2 comme l'illustre la figure 2.2. Cette notion sera formalisée dans la définition 2.3.2 du changement d'état.*

La fonction de transition $t : S \times \Sigma \rightarrow S$ décrit l'évolution du système suite à l'exécution d'une commande. Par exemple, l'évolution du système d'un état s_m à un état s_{m+1} suite à l'exécution d'une commande σ_m est notée :

$$s_{m+1} = t(s_m, \sigma_m)$$

Par souci de clarification, nous adopterons par la suite la notation équivalente :

$$s_m \xrightarrow{\sigma_m} s_{m+1}$$

Nous considérons également que les accès en écriture sont destructeurs et que le contenu présent avant l'exécution de la commande est effacé. Afin de

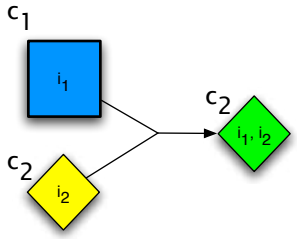

FIG. 2.3 – Exemple d'une modification de conteneur

modéliser les modifications de contenu, telles que l'ajout dans un fichier texte, nous considérons alors que la commande lit le contenu du conteneur modifié avant l'opération d'écriture. Cette notion revient à considérer des flux d'informations comprenant des sources multiples dont l'une correspond également à un conteneur destination. L'exemple suivant illustre ce cas particulier :

Exemple 3. *Soit le système décrit dans l'exemple 1 pris dans l'état initial* s_0 *(figure 2.1). Soit une commande* σ_1 *qui ajoute le contenu de* c_1 *dans* c_2 *:* $s_0 \xrightarrow{\sigma_1} s_1$*. Nous considérons que cette commande génère un flux d'informations des contenus* $\{i_1\}$ *et* $\{i_2\}$ *vers le conteneur* c_2*. Le nouveau contenu de* c_2 *dans l'état* s_1*, après l'exécution de la commande, est donc* $\{i_1, i_2\}$ *: il dépend à la fois de l'information initialement présente dans* c_2 *et de celle présente dans* c_1*. La figure 2.3 illustre ce cas.*

2.1.3.3 Traces d'exécution et flux d'informations composés

Soit $(\sigma_0, \sigma_1, \ldots, \sigma_m) \in \mathcal{P}(\Sigma)$ une trace d'exécution désignant la séquence de commandes menant le système de l'état initial s_0 à l'état s_{m+1}. Nous définissons une fonction de transition $t_{Seq} : P(\Sigma) \rightarrow S$ qui correspond à la composition, à partir de l'état initial, de la fonction de transition t pour chaque commande de la trace :

$$s_{m+1} = t_{Seq}((\sigma_0, \sigma_1, \ldots, \sigma_m))$$

Nous adopterons là aussi la notation équivalente :

$$s_0 \xrightarrow{(\sigma_0, \sigma_1, \ldots, \sigma_m)} s_{m+1}$$

Nous supposons que pour chaque commande σ, les ensembles C_σ^W et C_σ^R sont non vides. Les commandes qui ne vérifient pas cette hypothèse ne génèrent pas de flux d'informations et ne modifient pas l'état du système tel que nous le décrivons par la suite. Nous pouvons donc les éliminer de la trace observée.

Un flux d'informations élémentaire étant associé à chaque commande, il est possible de prendre en compte la composition de ces flux d'informations élémentaires lors de l'exécution d'une trace. Nous appelons de tels flux d'informations,

résultant de l'exécution d'une séquence de commandes, des flux d'informations composés. L'exemple suivant illustre intuitivement ce que nous entendons par flux d'informations composés :

Exemple 4. *Soit le système décrit dans l'exemple 1. Nous considérons ici une séquence de commandes exécutée à partir de l'état initial du système :*

1. *Lors de l'exécution de la première commande σ_0, l'utilisateur u génère de l'information via son interface et la sauvegarde dans le fichier c_1. Le système passe de l'état initial s_0 à l'état s_1 : $s_0 \xrightarrow{\sigma_0} s_1$. Nous considérons ici un flux d'informations élémentaire de i_u vers le conteneur c_1, $\{i_u\} \Rightarrow c_1$, le contenu initial de ce conteneur étant effacé par le nouveau contenu $\{i_u\}$;*
2. *Soit p_1 un processus du système. Supposons que le processus p_1 lise le fichier c_1, sélectionne une partie de son contenu, effectue un traitement sur les données lues et écrive le résultat de ce traitement dans le fichier c_2. Nous considérons également ici que cette deuxième commande $s_1 \xrightarrow{\sigma_1} s_2$ entraîne un flux d'informations du conteneur c_1 vers le conteneur c_2 ;*
3. *Soit p_2 un autre processus du système. Supposons que, consécutivement à l'exécution de p_1, p_2 effectue successivement les opérations suivantes :*
   (a) *il lit le conteneur c_2,*
   (b) *il effectue un test conditionnel sur les données lues,*
   (c) *il réalise, suivant la valeur du test, une écriture sur le fichier c_3.*

   Nous considérons qu'il existe alors un flux d'informations du contenu de c_2 vers c_3 lors de cette dernière commande $s_2 \xrightarrow{\sigma_2} s_3$. Nous considérons également que la composition de ces trois commandes, lors de l'exécution de la trace $s_0 \xrightarrow{(\sigma_0,\sigma_1,\sigma_2)} s_3$, génère un flux d'informations de i_u vers c_3, même si ce flux d'informations ne résulte pas d'une copie directe des données. En effet la première commande génère un flux d'informations de i_u vers c_1, la seconde un flux d'informations de cette même information vers c_2 (depuis c_1) et la dernière commande un flux d'informations vers c_3 (depuis c_2).

Cet exemple, illustré par la figure 2.4, montre que la notion de flux d'informations exprime ici une dépendance causale entre contenus.

Après avoir distingué la notion de contenu de celle de conteneur et défini ce que nous entendions par flux d'informations, nous proposons maintenant de définir la politique de flux d'informations.

2.2 Politique de flux d'informations

Nous nous intéressons aux politiques de sécurité qui déterminent la légalité de flux d'informations entre conteneurs à partir d'un état initial. Notre modèle de **politique de flux** spécifie, pour chaque information atomique, quelles sont

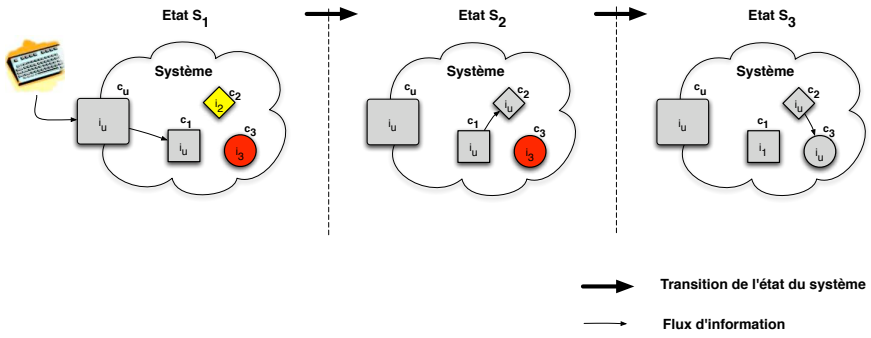

FIG. 2.4 – Exemples de flux d'informations

les destinations autorisées en termes de conteneurs. Nous proposons de ne spécifier que les flux d'informations légaux. Tout flux d'informations qui n'est pas explicitement autorisé par la politique sera donc considéré comme illégal.

2.2.1 Définitions

2.2.1.1 Politique de flux d'informations et CCAL

La définition 2.2.1 explicite formellement notre notion de politique de flux d'informations en termes d'associations légales contenus/conteneurs. Nous appelons ce type d'association un CCAL, pour *Contents - Container Authorized Link*, qui signifie littéralement lien autorisé entre contenus et conteneurs.

Définition 2.2.1 (Politique de flux d'informations)**.**
Soient $\mathbb{C}$ *l'ensemble des conteneurs du système et* $\mathbb{I}$ *l'ensemble des informations atomiques associées. Une politique de flux d'informations est un ensemble* SP *de paires* (I, C) *appelées CCAL, où :*
– $I \in \mathcal{P}(\mathbb{I})$ *est un ensemble d'informations atomiques,*
– $C \in \mathcal{P}(\mathbb{C})$ *est un ensemble de conteneurs.*
Chaque paire (I, C) *signifie que tout flux d'informations d'un sous-ensemble des informations initiales de* I *vers un des conteneurs de* C *est légal.*

2.2.1.2 Violation de la politique de flux d'informations

Etant donnée une politique de flux d'informations SP, nous considérons qu'une violation de cette politique correspond à un état interdit du système. Cet état est caractérisé par un ou plusieurs conteneurs dont le contenu courant n'est pas autorisé par la politique. Il existe donc une ou plusieurs associations contenus/conteneurs interdites, c'est-à-dire non spécifiées par SP.

Définition 2.2.2 (Violation de la politique de flux d'informations)**.**
Une violation de la politique de flux d'informations du système est caractérisée par un état du système dans lequel :
- *Il existe au moins un conteneur c dont le contenu courant est I_c et*
- *$\nexists (I, C) \in SP$ tel que $I_c \subseteq I$ et $c \in C$*

L'exemple 5 illustre la notion de CCAL et de violation de la politique de flux d'informations.

Exemple 5. *Soit le système décrit dans l'exemple 1. Soit Alice un utilisateur du système, c_A et i_A le conteneur et l'information atomique modélisant l'interface utilisateur d'Alice. Soit une politique de flux d'informations autorisant seulement les flux d'informations suivant :*
- *les flux d'informations de l'interface d'Alice (modélisée par l'information atomique i_A) et de i_1 (le contenu initial de c_1) vers les conteneurs c_1 et c_A ;*
- *les flux d'informations de i_1 (le contenu initial de c_1) vers c_1 ou c_2.*

Cette politique peut être modélisée simplement par deux CCAL :
- $CCAL_1 = (\{i_1, i_A\}, \{c_1, c_A\})$
- $CCAL_2 = (\{i_1\}, \{c_1, c_2\})$

La politique de flux d'informations est alors :

$$SP = \{CCAL_1, CCAL_2\}$$

Considérons maintenant l'état initial s_0 du système illustré par la figure 2.1.

Soit une commande $s_0 \xrightarrow{\sigma_0} s_1$ qui accède en lecture à l'interface d'Alice puis écrit dans c_1 : $C^R_{\sigma_0} = \{c_A\}$ et $C^W_{\sigma_0} = \{c_1\}$. L'exécution de cette commande dans l'état initial génère donc un flux d'informations $\{i_A\} \Rightarrow c_1$. Ce flux d'informations est autorisé par la politique SP car i_A et c_1 appartiennent tous les deux à $CCAL_1$.

Soit maintenant une commande $s_1 \xrightarrow{\sigma_1} s_2$ qui accède en lecture à c_1 puis écrit dans c_2 soit $C^R_{\sigma_1} = \{c_1\}$ et $C^W_{\sigma_1} = \{c_2\}$. L'exécution de cette commande dans l'état s_1 génère donc un flux d'informations $\{i_A\} \Rightarrow c_2$ dû à la trace $s_0 \xrightarrow{(\sigma_0, \sigma_1)} s_2$. Ce flux d'informations n'est pas autorisé par la politique SP car il n'existe pas de $CCAL$ dans SP où i_A et c_2 apparaissent à la fois dans ce même $CCAL$.

Considérons une nouvelle exécution à partir de l'état initial s_0. Soit une commande $s_0 \xrightarrow{\sigma_{0'}} s_{1'}$ qui accède en lecture à l'interface d'Alice et modifie le contenu de c_1 : $C^R_{\sigma_{0'}} = \{c_A, c_1\}$ et $C^W_{\sigma_{0'}} = \{c_1\}$ (la modification d'un conteneur est modélisée par une lecture suivie d'une écriture de ce même conteneur). Le flux d'informations élémentaire issu de l'exécution de cette commande dans l'état initial est donc $\{i_A, i_1\} \Rightarrow c_1$. Ce flux d'informations n'est pas autorisé par la politique car i_A, i_1 et c_1 n'apparaissent ensemble dans aucun $CCAL$ de SP.

2.2.2 Création et suppression de conteneurs

La création de nouveaux contenus est limitée dans notre modèle à la création de nouveaux conteneurs interface. Nous considérons en effet que les contenus sont générés à partir de l'ensemble des informations atomiques initiales du système et de l'ensemble des informations atomiques provenant des conteneurs interface. Le premier ensemble est par définition fixé par l'état initial ; le second est en revanche susceptible d'évoluer, par exemple lors de l'ajout de nouveaux services établissant une nouvelle interface ou lors de la connexion de nouveaux utilisateurs sur le système. Il convient alors de modifier la politique de flux d'informations afin de prendre en compte ce nouveau conteneur interface et l'information atomique qui lui est associée. En l'absence de modification de la politique, tout flux d'informations à destination de ce conteneur interface est interdit. De même, tout flux d'informations utilisant l'information atomique associée est interdit.

La création de conteneurs génériques, ne modélisant pas une interface, n'implique pas la création de nouveaux contenus. Par conséquent nous considérons que la création de nouveaux conteneurs vides est toujours autorisée. En toute rigueur, la politique de flux d'informations devrait être modifiée pour autoriser les flux d'informations à destination de ce nouveau conteneur. Nous verrons par la suite qu'il est possible d'autoriser par défaut tous les flux d'informations à destination de ce nouveau conteneur sans violer la politique établie jusqu'alors. Il est en revanche parfois nécessaire de modifier explicitement la politique pour assurer l'intégrité du nouveau conteneur.

La destruction de conteneurs implique en revanche la destruction de contenus. Il y a donc potentiellement atteinte à l'intégrité. Nous considérons pour cela les opérations de destruction de conteneurs comme des opérations d'écriture dans ces mêmes conteneurs. Le flux d'informations élémentaire associé à une action supprimant un conteneur dépend de l'observation : à l'échelle du système d'exploitation, nous considérons par exemple qu'une destruction de fichier à l'aide de l'appel système *unlink* génère un flux d'informations élémentaire de la mémoire du processus générant l'appel système vers le fichier supprimé.

2.2.3 Initialisation de la politique de flux d'informations

Nous ne nous intéressons pas ici à la classe des différentes politiques de sécurité qui peuvent être modélisées à l'aide de notre modèle de politique de flux d'informations. Nous supposons que ce travail de spécification est du ressort de l'administrateur système ou du responsable sécurité. La spécification manuelle de l'ensemble des CCAL peut cependant s'avérer fastidieuse. Nous pensons que l'utilisation de mécanismes d'aide à la spécification de la politique est souhaitable. Plusieurs solutions peuvent être envisagées. Les systèmes que nous souhaitons protéger utilisant généralement un mécanisme de contrôle d'accès discrétionnaire, nous proposons ici d'interpréter les permissions du contrôle d'accès et d'en déduire une politique de flux d'informations.

2.2.4 Interprétation d'une matrice de contrôle d'accès

Les permissions du contrôle d'accès discrétionnaire, que l'on peut exprimer sous la forme d'une matrice de contrôle d'accès, spécifient des droits statiques liés aux conteneurs. Ces droits peuvent être modifiés mais cela relève d'un changement de politique qui, dans le cas d'un contrôle discrétionnaire, est à la «discrétion» des différents utilisateurs «propriétaires», ces propriétaires étant généralement les créateurs des conteneurs d'informations. Il n'est donc pas possible d'établir de manière univoque une politique de flux d'informations à partir de ces permissions. Nous proposons ici une interprétation possible qui peut s'avérer utile lors de la spécification de la politique de flux d'informations en fournissant un premier ensemble de CCAL qui peut ensuite être modifié explicitement. Nous utilisons l'algorithme proposé ici dans une implémentation au niveau du système d'exploitation de notre modèle, présentée dans le chapitre 3.

Afin d'établir un ensemble de flux d'informations légaux, nous faisons l'hypothèse suivante : seuls les flux d'informations directs qui sont autorisés par le contrôle d'accès sont considérés comme légaux. Nous entendons par flux d'informations direct un flux d'informations qui peut être réalisé par une seule entité active (utilisateur ou plus précisément processus s'exécutant pour le compte d'un utilisateur). Par opposition, les flux d'informations nécessitant la collaboration de plusieurs entités sont considérés comme des flux d'informations indirects. L'exemple 6 illustre cette notion.

Exemple 6 (Génération d'une politique de flux d'informations à partir d'une interprétation d'une matrice de contrôle d'accès)**.** *Soit une matrice de contrôle d'accès spécifiant un ensemble de ressources, un ensemble d'utilisateurs et, pour chaque paire (ressource, utilisateur), les opérations (lecture ou écriture) que l'utilisateur est autorisé à effectuer sur la ressource. Considérons par exemple la matrice suivante :*

	c_1	c_2	c_3
Alice	*lecture, écriture*		*lecture*
Bob		*lecture, écriture*	*lecture, écriture*
Charlie	*lecture, écriture*	*lecture, écriture*	

Cette matrice spécifie trois conteneurs c_1, c_2 et c_3. Nous considérons également implicitement les trois contenus initiaux de c_1, c_2 et c_3. Nous notons i_1, et respectivement i_2, i_3, l'information atomique contenue initialement dans c_1, et respectivement c_2, c_3.

Il est également nécessaire de prendre en compte les interfaces entre le système et ses utilisateurs. Nous considérons donc un conteneur interface et son information atomique associée pour chaque utilisateur authentifié. Nous définissons donc trois conteneurs interface c_u, $u \in \{A, B, C\}$ pour chaque utilisateur Alice, Bob ou Charlie. Les informations atomiques associées à ces conteneurs sont notées $i_u, u \in \{A, B, C\}$.

Cette matrice définit des flux d'informations directs autorisés. Par exemple, le flux d'informations de i_3 vers c_1 est un flux d'informations direct autorisé par

le contrôle d'accès car l'utilisateur Alice peut à la fois lire i_3 (le contenu initial de c_3) et écrire dans c_1. Il peut donc réaliser seul ce flux d'informations. De même le flux d'informations de i_B et i_2 vers c_3 est un flux d'informations direct autorisé car Bob peut le réaliser. En revanche, le flux d'informations de i_1 vers c_3 n'est pas un flux d'informations direct autorisé par la politique car aucune entité active du système ne peut le réaliser seule. Ce flux d'informations peut en revanche être réalisé par la composition de deux flux d'informations réalisés par deux entités actives distinctes, ici Charlie et Bob, qui doivent collaborer.

De manière générale, nous définissons un CCAL $(I_u, C_u)_{u\in\{A,B,C\}}$ pour chaque utilisateur afin de prendre en compte l'ensemble des flux d'informations directs autorisés. I_u est l'ensemble des contenus atomiques que l'utilisateur peut lire. C_u est l'ensemble des conteneurs auxquels l'utilisateur est autorisé à accéder en écriture.

Alice, par exemple, est autorisée à lire les contenus atomiques initiaux i_1 de c_1 et i_3 de c_3. Elle peut elle-même générer l'information i_A. De plus, Alice peut écrire dans les conteneurs c_1 et c_A. Tous les flux d'informations de i_1, i_3 et i_A vers les conteneurs c_1 ou c_A sont donc légaux puisqu'Alice peut réaliser des commandes susceptibles de générer de tels flux d'informations. La partie de la politique de flux d'informations relative à l'utilisateur Alice est donc modélisée par le CCAL :

$$CCAL_A = (\{i_1, i_3, i_A, \}, \{c_1, c_A\})$$

En suivant le même raisonnement, nous pouvons définir deux autres CCAL correspondant à Bob et Charlie :

$CCAL_B = (\{i_2, i_3, i_B\}, \{c_2, c_3, c_B\})$
$CCAL_C = (\{i_1, i_2, i_C\}, \{c_1, c_2, c_C\})$

La politique de flux d'informations est finalement modélisée par l'ensemble de ces trois CCAL :

$$SP = \{CCAL_A, CCAL_B, CCALs_C\}$$

Il est donc possible de spécifier, manuellement ou à l'aide d'une interprétation automatique, une politique de flux d'informations sous la forme d'un ensemble de CCAL. Nous proposons par la suite de lier les conteneurs et leur contenu courant aux différents CCAL dans lesquels ils sont impliqués à l'aide de tags de sécurité.

2.2.5 Tags de sécurité

Nous avons vu précédemment qu'il est possible de définir une politique de flux d'informations sous la forme d'un ensemble de relations contenus/conteneurs autorisées. Nous voulons construire un système de détection qui, pour chaque commande s'exécutant sur le système, vérifie la légalité des flux d'informations, qu'ils soient directement engendrés par cette commande ou qu'ils résultent de la composition des flux d'informations issus de l'exécution de la trace complète. Etant donnée une politique définie sous forme d'un ensemble

de CCAL, nous pourrions intuitivement vérifier la légalité des flux d'informations, au regard de cette politique, en suivant l'évolution de tous les contenus du système et en vérifiant, pour chaque étape, la légalité des associations contenus/conteneurs. En pratique, cette technique ne nous semble pas réaliste car elle entraîne une explosion combinatoire, chaque conteneur étant susceptible de recevoir un contenu dérivé de toutes les informations présentes dans le système. De plus, la définition du contenu que nous avons retenue désignant l'origine des données, cet ensemble croît lorsque les flux d'informations modifient les données.

Nous proposons d'associer à chaque conteneur deux méta-données relatives à la sécurité que nous nommons **tags de sécurité**. Le premier, appelé **tag de sécurité en lecture**, est relatif au contenu courant du conteneur. Sa valeur évolue lorsque le contenu évolue, c'est-à-dire lors des accès en écriture sur le conteneur. Le second, appelé **tag de sécurité en écriture**, est relatif au conteneur à proprement parler. Sa valeur n'évolue pas sauf à la création et à la suppression du conteneur.

Chaque tag de sécurité contient un ensemble de CCAL :

- le **tag de sécurité en lecture** contient l'ensemble des CCAL qui sont relatifs au **contenu** courant ;
- le **tag de sécurité en écriture** contient l'ensemble des CCAL qui sont relatifs au **conteneur**.

Un tag de sécurité est donc un «ensemble d'ensembles». D'un point de vue pratique, il est possible d'associer un identifiant unique à chaque CCAL, un tag étant alors un ensemble d'identifiants. Dans l'état initial, les tags reflètent uniquement l'expression de la politique de sécurité. Plus précisément, soit une politique de flux d'informations SP exprimée sous forme d'un ensemble de CCAL. Pour chaque conteneur c_n de l'état initial auquel nous associons l'information atomique initiale i_n, nous définissons un tag en lecture T^R et un tag en écriture T^W tels que :

- T^R contient l'ensemble des CCAL où i_n apparaît ;
- T^W contient l'ensemble des CCAL où c_n apparaît.

Ces tags sont susceptibles d'évoluer lorsque l'état du système change : nous donnons en section 2.3 une définition par récurrence de la valeur des tags de sécurité. Dans un souci de clarté, nous noterons par la suite en indice l'état du système correspondant. Ainsi le tag de sécurité en écriture lié au conteneur c_n sera noté T^W_{n,s_n} dans l'état s_n. Nous verrons par la suite, en section 2.3, qu'il est possible de déterminer la légalité des flux d'informations à partir de la seule observation de l'évolution des tags de sécurité. L'implémentation du détecteur d'intrusions peut donc reposer uniquement sur ces tags. Nous verrons également que la loi d'évolution des tags est telle que le cardinal d'un tag est une fonction décroissante en fonction du temps. De plus, le nombre de CCAL définis pour un système est en pratique très inférieur au nombre d'informations atomiques du système. Cela permet donc de limiter l'occupation mémoire.

L'exemple suivant illustre l'initialisation des tags de sécurité pour le système décrit dans l'exemple 1 et la politique de flux d'informations définie dans l'exemple 6.

Exemple 7 (Initialisation des tags de sécurité à partir d'une politique de flux d'informations)**.** *Soit le conteneur c_3 et son information initiale i_3. Etant donnée la politique de flux d'informations SP définie dans l'exemple 6, nous pouvons remarquer que c_3 apparaît dans $CCAL_B$ et que i_3 apparaît dans $CCAL_A$ et $CCAL_B$. Les tags de sécurité associés à c_3 seront donc, dans l'état initial :*

- $T_3^R = \{CCAL_A, CCAL_B\}$
- $T_3^W = \{CCAL_A\}$

En suivant le même raisonnement, nous pouvons définir les tags initiaux de c_2 et c_1 :

- $T_2^R = \{CCAL_B, CCAL_C\}$
- $T_2^W = \{CCAL_B, CCAL_C\}$
- $T_1^R = \{CCAL_A, CCAL_C\}$
- $T_1^W = \{CCAL_A, CCAL_C\}$

Nous proposons par la suite une loi régissant l'évolution des tags de sécurité et une règle de légalité de flux d'informations calculée à partir de ces mêmes tags. Nous décrivons notre modèle de l'état du système et de son évolution. Nous montrons ensuite que la loi d'évolution et la règle de légalité de flux d'informations permettent de détecter les violations d'une politique de flux d'informations définie en termes de CCAL.

2.3 Modèle de système de détection

Nous voulons modéliser l'évolution de l'état d'un système résultant de flux d'informations entre conteneurs. Nous avons présenté précédemment ce que nous entendions par conteneurs, contenus et flux d'informations. Nous avons vu en particulier qu'il était possible de suivre les flux d'informations en exprimant le contenu courant de chaque conteneur, en termes d'origines des données courantes, à partir de l'état initial du système. Nous définissons ici plus précisément ce que nous entendons par état du système et transition du système d'un état à un autre. Nous nous intéressons à un modèle d'état structuré, aussi nous définissons en premier lieu la notion d'**objet** qui correspond à la partie atomique de l'état structuré du système. Nous définissons ensuite les transitions en termes d'évolutions d'objets.

2.3.1 Objets

Un objet est une relation entre un conteneur c_n, son contenu courant I_n et les deux tags de sécurité qui lui sont associés T_n^R et T_n^W. Le terme d'objet ne désigne pas seulement ici, comme c'est souvent le cas, le conteneur d'informations. Il s'agit de la partie atomique de l'état structuré du système, relative à un conteneur particulier. L'état du système peut donc être décrit comme un ensemble d'objets, ensemble qui évolue lors de l'exécution de commandes générant des flux d'informations. Ces flux d'informations entraînent en effet la modification de certains contenus et des tags de sécurité associés.

Définition 2.3.1 (Objet)**.**
Un objet est un quadruplet (I_n, c_n, T_n^R, T_n^W) *où :*
– $c_n \in \mathbb{C}$ *est un conteneur,*
– $I_n \subseteq \mathbb{I}$ *est le contenu courant du conteneur,*
– $T_n^R \subseteq SP$ *est le tag de sécurité en lecture associé au contenu,*
– $T_n^W \subseteq SP$ *est le tag de sécurité en écriture associé au conteneur.*
Ω *est l'ensemble de tous les objets du système.*

L'ensemble $\mathbb{C}$ des conteneurs étant supposé fini, l'ensemble Ω des objets l'est également.

2.3.2 Flux de transition

Lorsqu'une commande s'exécute sur le système, elle génère un flux d'informations qui modifie l'état du système. Intuitivement, les contenus des conteneurs accédés en écriture sont modifiés et par conséquent les tags en lecture doivent l'être également.

Considérons une commande σ_m, $C_{\sigma_m}^R$ l'ensemble des conteneurs accédés en lecture par σ_m et $C_{\sigma_m}^W$ l'ensemble des conteneurs accédés en écriture par σ_m. Comme indiqué précédemment, nous supposons que cette commande engendre un flux d'informations élémentaire de l'union des informations contenues dans les conteneurs de $C_{\sigma_m}^R$ vers chacun des conteneurs de $C_{\sigma_m}^W$. Nous associons à ce flux d'informations un flux de transition modélisant l'évolution des objets liée à ce flux d'informations.

Nous identifions dans un premier temps l'ensemble des objets TBR, littéralement *to-be-read*. Cet ensemble correspond à l'état des conteneurs accédés en lecture, avant l'exécution de la commande. Nous identifions ensuite l'ensemble des objets TBW, littéralement *to-be-written*. Cet ensemble correspond à l'état des conteneurs accédés en écriture, avant l'exécution de la commande.

Soit s_m l'état du système avant l'exécution de la commande :
– TBR_m est l'ensemble des objets de s_m dont le conteneur appartient à $C_{\sigma_m}^R$;
– TBW_m est l'ensemble des objets de s_m dont le conteneur appartient à $C_{\sigma_m}^W$.

Afin de décrire formellement ces deux ensembles, nous définissons l'application suivante :

$$Cont : \Omega \rightarrow \mathbb{C}$$

telle que :

$$Cont(o = (I, c, T^R, T^W)) = c$$

Soit la transition $s_m \xrightarrow{\sigma_m} s_{m+1}$. Nous avons alors :

$$\mathsf{TBR}_m = \{o \in s_m | Cont(o) \in C_{\sigma_m}^R\}$$

$$\mathsf{TBW}_m = \{o \in s_m | Cont(o) \in C_{\sigma_m}^W\}$$

L'état de chaque conteneur accédé en écriture étant modifié à la suite de l'exécution de la commande, nous définissons un nouvel ensemble d'objets W_m, littéralement *written*, qui correspond au nouvel état des conteneurs accédés en écriture, après l'exécution de la commande. La modification de l'état des conteneurs accédés en écriture se traduit donc à la fois par la création de nouveaux objets (ceux de W_m) et la destruction de certains objets (ceux de TBW_m). Le flux de transition est alors une application de TBR_m et TBW_m vers W_m, cette notion étant explicitée formellement dans la définition 2.3.2. La définition de W est la suivante :

- les conteneurs de W_m sont ceux de TBW_m puisque seuls les objets accédés en écriture voient leur état modifié ;
- le contenu associé à chacun de ces conteneurs est l'union de tous les contenus lus ;
- les tags en écriture des objets de W_m sont ceux des objets de TBW_m puisque ces tags sont liés aux conteneurs ;
- le tag en lecture de chaque objet de W_m est l'intersection des tags en lecture des objets de TBR_m.

Le dernier point désigne en fait la loi de propagation des tags de lecture. Nous verrons par la suite que cette loi, couplée à la règle de légalité de flux d'informations, permet de détecter les flux d'informations interdits par la politique spécifiée dans l'état initial. Le nouveau contenu des objets de W_m dépend causalement de tous les contenus lus. Par conséquent, la politique qui s'applique sur ce contenu doit être au moins aussi restrictive que celle qui s'applique sur chacun des contenus lus.

Définition 2.3.2 (Flux de transition)**.**
Un flux de transition est une application

$$f : \mathcal{P}(\Omega) \times \mathcal{P}(\Omega) \rightarrow \mathcal{P}(\Omega)$$

$$(\mathsf{TBR}, \mathsf{TBW}) \mapsto \mathsf{W}$$

avec :

$$\mathsf{TBR} = \{(I_j, c_j, T_j^R, T_j^W)\}_{j \in J}$$

$$\mathsf{TBW} = \{(I_k, c_k, T_k^R, T_k^W)\}_{k \in K}$$

$$\mathsf{W} = \{(\bigcup_{j \in J} I_j, c_k, \bigcap_{j \in J} T_j^R, T_k^W)\}_{k \in K}$$

2.3.3 Système et transitions

Comme nous l'avons présenté précédemment, l'état d'un système est modélisé par un ensemble dynamique d'objets. Chaque flux d'informations généré par l'exécution d'une commande modifie les associations courantes contenus/-conteneurs et par conséquent modifie l'ensemble des objets définissant l'état du système. La définition 2.3.3 définit plus précisément l'état d'un système par

récurrence. Soit un système et une politique de flux d'informations SP. Nous définissons dans un premier temps l'état initial de la manière suivante :

- nous associons un objet, que nous appelons par la suite *objet initial*, à chaque conteneur. L'état initial est tel que le conteneur de chaque objet est différent ;
- le contenu de chaque objet est un singleton correspondant à l'information atomique initialement présente dans le conteneur de l'objet ;
- le tag de sécurité en lecture est l'ensemble des CCAL de SP où le contenu de l'objet apparaît ;
- le tag de sécurité en écriture est l'ensemble des CCAL de SP où le conteneur de l'objet apparaît.

Nous définissons ensuite les autres états par récurrence. Le système passe d'un état s_m à un état s_{m+1} suite à l'exécution d'une commande σ_m. L'état du système étant défini par un ensemble, les transitions sont modélisées par des opérations sur cet ensemble :

- les objets TBW_m, correspondant à l'état des conteneurs accédés en écriture avant l'exécution du flux d'informations, sont retirés ;
- les objets W_m, correspondant à l'état des conteneurs accédés en écriture après l'exécution du flux d'informations, sont ajoutés ;

Comme indiqué précédemment, les changements d'état résultent à la fois de la destruction d'objets existants (dont les objets initiaux) et la création de nouveaux objets traduisant le changement d'état des conteneurs accédés en écriture.

Définition 2.3.3 (Transition du système)**.**
Soit un système de N conteneurs $c_n \in \mathbb{C}$, auxquels sont associés les N informations atomiques $i_n \in \mathbb{I}$, décrit par un automate $A = (S, \Sigma, t, s_0, Q)$. Soit SP une politique de flux d'informations exprimée sous forme d'un ensemble de CCAL.
L'état initial s_0 du système est défini par :

$$s_0 = \{(\{i_n\}, c_n, T^R_{n,s_0}, T^W_{n,s_0})\}_{n\in\{1,\dots,N\}}$$

où :
– $T^R_{n,s_0} = \{(I, C) \in SP \mid i_n \in I\}$

– $T^W_{n,s_0} = \{(I, C) \in SP \mid c_n \in C\}$

Pour tout état s_m, l'exécution d'une commande σ_m fait transiter le système d'un état s_m à un état s_{m+1}. Les objets accédés par σ_m sont :

$$\mathsf{TBR}_m = \{o \in s_m | Cont(o) \in C^R_{\sigma_m}\}$$

$$\mathsf{TBW}_m = \{o \in s_m | Cont(o) \in C^W_{\sigma_m}\}$$

Le flux de transition associé à l'exécution de σ_m génère les objets

$$\mathsf{W}_m = f(\mathsf{TBR}_m, \mathsf{TBW}_m)$$

et la transition d'état est définie par :

$$s_m \xrightarrow{\sigma}_m s_{m+1}$$

où : $s_{m+1} = t(s_m, \sigma_m) = (s_m \setminus \mathsf{TBW}_m) \cup \mathsf{W}_m$

Exemple 8. *Considérons le système et la politique de flux d'informations SP décrits dans l'exemple 7. Ce système est caractérisé par trois utilisateurs Alice (A), Bob (B) et Charlie (C) auxquels nous associons donc trois conteneurs interface c_A, c_B et c_C. Il est également composé de trois conteneurs c_1, c_2 et c_3. La politique de flux d'informations SP est spécifiée à l'aide de trois CCAL : $SP = \{CCAL_A, CCAL_B, CCAL_C\}$.*

Nous modélisons dans un premier temps l'état initial de ce système par 6 objets correspondants aux 6 conteneurs du système. Chaque objet est de la forme $(\{i_n\}, c_n, T^R_{n,s_0}, T^W_{n,s_0})$. Le premier objet du système est par exemple caractérisé par :

- *son conteneur c_1 ;*
- *le contenu de ce conteneur qui se réduit dans l'état initial au singleton $\{i_1\}$;*
- *le tag de sécurité en lecture associé, qui contient l'ensemble des CCAL de SP dans lesquels i_1 apparaît soit $T^R_{1,s_0} = \{CCAL_A, CCAL_C\}$;*
- *le tag de sécurité en écriture associé, qui contient l'ensemble des CCAL de SP dans lesquels c_1 apparaît soit $T^W_{1,s_0} = \{CCAL_A, CCAL_C\}$.*

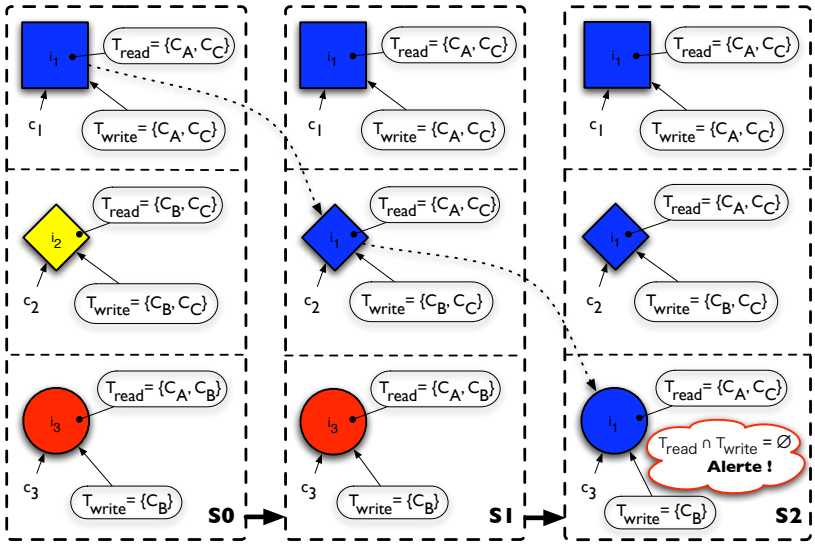

FIG. 2.5 – Evolution de l'état d'un système

Les 5 autres objets initiaux sont définis en suivant le même raisonnement et l'état initial s_0 du système est alors :

$$\begin{aligned} s_0 \quad = \{ \quad & (\{i_1\}, c_1, \{CCAL_A, CCAL_C\}, \{CCAL_A, CCAL_C\}), \\ & (\{i_2\}, c_2, \{CCAL_B, CCAL_C\}, \{CCAL_B, CCAL_C\}), \\ & (\{i_3\}, c_3, \{CCAL_A, CCAL_B\}, \{CCAL_B\}), \\ & (\{i_A\}, c_A, \{CCAL_A\}, \{CCAL_A\}), \\ & (\{i_B\}, c_B, \{CCAL_B\}, \{CCAL_B\}), \\ & (\{i_C\}, c_C, \{CCAL_C\}, \{CCAL_C\}) \ \} \end{aligned}$$

Supposons maintenant que l'exécution d'une commande σ_0 copie l'information du conteneur c_1 dans le conteneur c_2, soit $C^R_{\sigma_0} = \{c_1\}$ et $C^W_{\sigma_0} = \{c_2\}$. Ce flux d'informations implique un flux de transition mettant en œuvre les ensembles d'objets suivants :

- *TBR* $= Cont(C^R_{\sigma_0}) = \{(\{i_1\}, c_1, \{C_A, C_C\}, \{C_A, C_C\})\}$, *l'ensemble des objets dont le conteneur est lu par la commande et*
- *TBW* $= Cont(C^W_{\sigma_0}) = \{(\{i_2\}, c_2, \{C_B, C_C\}, \{C_B, C_C\})\}$, *l'ensemble des objets dont le conteneur est accédé en écriture par la commande.*

Dans cet exemple, seul un conteneur est modifié, l'ensemble W des objets générés par le flux de transition (voir définition 2.3.2) est donc réduit à un singleton. Le nouvel objet est caractérisé par :

- *son conteneur c_2 qui est le seul conteneur accédé en écriture lors de l'exécution de l'opération ;*
- *le contenu i_1 qui est le nouveau contenu de c_2 résultant du flux d'informations ;*

- *le nouveau tag de sécurité en lecture* T^R *qui est l'intersection de tous les tags de sécurité en lecture des objets de* TBR*. Ce dernier ensemble étant lui-même réduit à un singleton, le nouveau tag en lecture est simplement* $\{CCAL_A, CCAK_C\}$ *;*
- *le tag de sécurité en écriture, lié au conteneur et qui ne varie donc pas par rapport à l'objet de* TBW*, soit* $\{C_B, C_C\}$*.*

Le nouvel état $s_1 = (s_0 - \mathsf{TBW}) \cup \mathsf{W}$ *atteint après l'exécution de ce flux d'informations élémentaire est donc :*

$$\begin{aligned} s_1 \quad = \{ \quad & (\{i_1\}, c_1, \{CCAL_A, C_C\}, \{CCAL_A, CCAL_C\}), \\ & (\{\mathbf{i_1}\}, c_2, \{\mathbf{CCAL_A}, \mathbf{CCAL_C}\}, \{CCAL_B, CCAL_C\}), \\ & (\{i_3\}, c_3, \{CCAL_A, CCAL_B\}, \{CCAL_B\}), \\ & (\{i_A\}, c_A, \{CCAL_A\}, \{CCAL_A\}), \\ & (\{i_B\}, c_B, \{CCAL_B\}, \{CCAL_B\}), \\ & (\{i_C\}, c_C, \{CCAL_C\}, \{CCAL_C\}) \ \} \end{aligned}$$

Les éléments de cet ensemble notés en gras correspondent aux modifications apportées par rapport à l'état précédent, ici s_0*.*

2.3.4 Règle de propagation des tags de sécurité

D'après la définition 2.3.3, les transitions de l'état du système ne modifient que les tags de sécurité en lecture associés aux conteneurs accédés en écriture. De nouveaux objets sont créés pour ces conteneurs tels que :

- le tag de sécurité en écriture, lié au conteneur, n'évolue pas par rapport à celui des objets supprimés ;
- le tag de sécurité en lecture, lié au nouveau contenu, est l'intersection des tags de sécurité en lecture de tous les objets dont le conteneur a été accédé en lecture durant le flux d'informations élémentaire.

Nous montrons maintenant que cette règle de propagation des tags de sécurité permet de suivre les flux d'informations et donc de vérifier que l'exécution d'une trace est conforme à une politique de flux d'informations spécifiée en termes de CCAL. Plus précisément, les tags de sécurité en lecture correspondent à la partie de la politique de flux d'informations qui s'applique aux contenus courants. Les tags de sécurité en écriture correspondent à la partie de la politique de flux d'informations qui s'applique aux conteneurs. Le lemme 1 précise les relations qui existent, pour chaque objet, entre le contenu courant et le tag de sécurité en lecture d'une part et entre le conteneur et le tag de sécurité en écriture d'autre part. Ce lemme montre que pour chaque objet de **chaque** état du système, quatre propriétés sont vérifiées :

1. Le conteneur de l'objet est un des conteneurs du système ;
2. Le contenu de l'objet est l'ensemble des informations atomiques de l'état initial qui ont été utilisées pour générer les données actuelles du conteneur ;
3. Le tag de sécurité en écriture est le même que celui de l'objet initial, c'est-à-dire appartenant à l'état initial, dont le conteneur est celui de l'objet en question.

4. Le tag de sécurité en lecture de l'objet est l'intersection des tags de sécurité des objets initiaux associés, dans l'état initial, à toutes les informations atomiques utilisées pour créer le contenu courant de l'objet en question.

La première propriété est triviale car par définition aucun conteneur n'est créé lors d'un flux d'informations. La troisième propriété découle directement de la règle de propagation des tags de sécurité en écriture. Le tag en écriture est lié au conteneur et par conséquent tous les objets associés à un conteneur donné et qui traduisent les différents états de ce conteneur possèdent le même tag de sécurité en écriture. La deuxième propriété illustre le fait que notre modèle exprime, pour chaque flux d'informations élémentaire, l'origine des nouveaux contenus créés à partir de l'origine des contenus lus. Il est donc possible d'exprimer, à chaque état du système, l'origine de chaque contenu à partir de l'état initial. Ainsi, cela revient à considérer le flux d'informations équivalent résultant de la composition des différents flux d'informations élémentaires qui ont amené le système de l'état initial à l'état courant. La quatrième propriété est la plus intéressante. Elle exprime la dualité entre contenu courant, exprimé en termes d'informations atomiques, et tag de sécurité en lecture. Elle montre que ce dernier ne dépend que :

1. des informations atomiques du contenu courant, donc des flux d'informations qui ont permis de générer ce contenu ;
2. de la politique de flux d'informations SP exprimée dans l'état initial.

Lemme 1.
Soit un système de N conteneurs $c_n \in \mathbb{C}$, auxquels sont associés les N informations atomiques $i_n \in \mathbb{I}$, décrit par un automate $A = (S, \Sigma, t, s_0, Q)$. Soit SP une politique de flux d'informations sur ce système exprimée sous la forme d'un ensemble de CCAL. Soit s_0 l'état initial du système.
Si $s_0 = \{(\{i_n\}, c_n, T^R_{n,s_0}, T^W_{s,s_0})\}_{n\in\{1,\dots,N\}}$ *alors :*

$$\forall s_m,\ \forall \Theta = (I_\Theta, c_\Theta, T^R_{\Theta,s_m}, T^W_{\Theta,s_m}) \in s_m$$

1. $\exists c_k, k \in \{1, ..., N\}$ *tel que* $c_\Theta = c_k$
2. $\exists \{i_{j_1}, \dots, i_{j_p}\} \subseteq \{i_1, \dots, i_N\}$ *tels que* $I_\Theta = \{i_{j_1}, ..., i_{j_p}\}$
3. $T^R_{\Theta,s_m} = T^R_{j_1,s_0} \cap \dots \cap T^R_{j_p,s_0}$
4. $T^W_{\Theta,s_m} = T^W_{k,s_0}$

Démonstration. Nous démontrons le lemme 1 à l'aide d'une démonstration par récurrence sur la longueur de la séquence des flux d'informations élémentaires qui a amené le système de l'état initial s_0 à l'état courant s_m.

1. Les quatre propriétés sont vérifiées pour l'état s_0 car elles découlent directement de la définition 2.3.3.
2. Nous supposons maintenant que les quatre propriétés sont vérifiées pour tous les objets dans l'état s_m.

3. Nous devons établir que ces quatre propriétés sont vérifiées dans l'état s_{m+1}.
   Soit $\Theta \in s_{m+1}$
   Nous supposons que l'exécution d'une commande σ_m fait transiter le système de l'état s_m à l'état s_{m+1}. Les ensembles d'objets de s_m $\mathsf{TBR}_m = \{o \in s_m | Cont(o) \in C^R_{\sigma_m}\}$ et $\mathsf{TBW}_m = \{o \in s_m | Cont(o) \in C^W_{\sigma_m}\}$ sont définis à partir de $C^R_{\sigma_m}$, l'ensemble des conteneurs accédés en lecture par σ_m et $C^W_{\sigma_m}$, l'ensemble des conteneurs accédés en écriture par σ_m. Le flux de transition associé à l'exécution de la commande génère l'ensemble des nouveaux objets $\mathsf{W}_m = f(\mathsf{TBR}_m, \mathsf{TBW}_m)$.
   Etant donné que $\Theta \in s_{m+1}$ et $s_{m+1} = (s_m - \mathsf{TBW}_m) \cup \mathsf{W}_m$, alors soit $\Theta \in s_m$, soit $\Theta \in \mathsf{W}_m$.
   – Si $\Theta \in s_m$, alors les quatre propriétés sont vérifiées pour Θ d'après l'hypothèse de récurrence.
   – Dans le cas contraire, nous pouvons établir à partir de la définition du flux de transition :

$$\mathsf{TBR}_m = \{(I_j, c_j, T^R_{j,s_m}, T^W_{j,s_m})\}_{j \in J \subseteq \{1,\ldots,N\}}$$

$$\mathsf{TBW}_m = \{(I_k, c_k, T^R_{k,s_m}, T^W_{k,s_m})\}_{k \in K \subseteq \{1,\ldots,N\}}$$

$$\mathsf{W}_m = \{(\bigcup_{j \in J} I_j, c_k, \bigcap_{j \in J} T^R_{j,s_m}, T^W_{k,s_m})\}_{k \in K}$$

   $\Theta \in \mathsf{W}_m$, donc Θ a été créé par f et :

$$\exists k \in K, \text{ tel que } \Theta = (\bigcup_{j \in J} I_j, c_k, \bigcap_{j \in J} T^R_{j,s_m}, T^W_{k,s_m})$$

   D'après l'hypothèse de récurrence qui s'applique à l'état s_m :
   (a) $\forall j \in J, \exists i_{j_1}, \ldots, i_{j_p}$ tels que $I_j = \{i_{j_1}, \ldots, i_{j_p}\}$
   (b) $\forall j \in J, T^R_{j,s_m} = T^R_{j_1,s_0} \cap \ldots \cap T^R_{j_p,s_0}$.
   (c) $T^W_{k,s_m} = T^W_{k,s_0}$
   Ainsi, $\Theta = (\bigcup_{j \in J}\{i_{j_1}, \ldots, i_{j_p}\}, c_k, \bigcap_{j \in J} T^R_{j_1,s_0} \cap \ldots \cap T^R_{j_p,s_0}, T^W_{k,s_0})$

□

Les propriétés 2 et 3 de ce lemme montrent qu'il est possible de raisonner sur les contenus des objets, en termes de politiques de flux d'informations, à partir des seuls tags de sécurité. Nous montrons dans la section suivante que la loi de propagation des tags de sécurité permet de mettre en œuvre un algorithme de détection d'intrusions. Une intrusion, considérée comme une violation de la politique de flux d'informations, est détectée lorsqu'apparaît, au sein du système, un objet dont les tags de sécurité en lecture et en écriture sont des ensembles disjoints.

2.4 Détection d'intrusions

A partir du modèle proposé dans les sections précédentes, nous décrivons maintenant notre algorithme de détection d'intrusions. Nous apportons ensuite la preuve que cet algorithme permet de détecter les violations de la politique de flux d'informations en vigueur au sein du système. Le théorème de détection démontre la cohérence du processus de détection à partir d'une observation (l'ensemble des flux d'informations détectés) et d'une spécification (la politique de flux d'informations).

L'algorithme proposé permettant uniquement de détecter des flux d'informations violant une politique de flux d'informations préalablement spécifiée, seules les violations d'intégrité ou de confidentialité induites par ces flux d'informations seront détectées. Les problématiques de disponibilité, résultant par exemple d'une attaque en dénis de service, ne sont pas prises en compte. De même, une attaque ne générant que des flux d'informations autorisés par la politique ne sera pas détectée. Par exemple, l'exécution, sur le compte d'un utilisateur du système, d'un malware qui n'effectue que des opérations autorisées pour cet utilisateur, comme l'envoi de mails, ne sera pas détectée. Toutefois, l'installation de ce malware peut être détectée si elle génère des flux d'informations interdits.

2.4.1 Théorème de détection d'intrusions

Nous avons exprimé, dans la définition 2.2.1 , la légalité de flux d'informations entre conteneurs grâce à un ensemble de relations contenus/conteneurs autorisées. Ainsi, d'après la définition 2.2.2, un flux d'informations est illégal au regard de la politique de flux d'informations si et seulement si ce flux d'informations génère un contenu I_c dans un conteneur c et qu'il n'existe pas de CCAL (I, C) tel que $I_c \subseteq I$ et $c \in C$. Nous montrons maintenant à l'aide du théorème de détection qu'un tel état, interdit par la politique, est atteint si et seulement si le flux de transition qui a permis d'atteindre cet état a généré un objet dont les tags de sécurité en lecture T^R et en écriture T^W sont des ensembles disjoints.

Théorème 1 (Détection d'intrusions)**.**
Soit un système de N conteneurs $c_n \in \mathbb{C}$, auxquels sont associés les N informations atomique $i_n \in \mathbb{I}$, décrit par un automate $A = (S, \Sigma, t, s_0, Q)$. Soit SP une politique de flux d'informations sur ce système exprimée sous la forme d'un ensemble de CCAL. Soit s_0 l'état initial du système, de la forme :

$$s_0 = \{\Theta_n = (\{i_n\}, c_n, T^R_{n,s_0}, T^W_{n,s_0})\}_{n\in\{1,\dots,N\}}$$
$$T^R_{n,s_0} = \{(I, C) \in SP | i_n \in I\}$$
$$T^W_{s,s_0} = \{(I, C) \in SP | c_n \in C\}$$

L'exécution d'une opération

$$s_m \xrightarrow{\sigma_m} s_{m+1} = s_m \backslash \mathsf{TBW}_m \cup W_m$$

constitue une violation de la politique de flux d'informations SP si et seulement si

$$\exists \Theta = (I_\Theta, c_\Theta, T^R_{\Theta,s_m}, T^W_{\Theta,s_m}) \in \mathsf{W}_m \text{ tel que } T^R_{\Theta,s_m} \cap T^W_{\Theta,s_m} = \emptyset$$

Démonstration. La démonstration de ce théorème est réalisée en deux étapes :

1. Nous montrons tout d'abord que si un flux d'informations génère un objet $\Theta = (I_\Theta, c_\Theta, T^R_{\Theta,s_m}, T^W_{\Theta,s_m})$ tel que $T^R_{\Theta,s_m} \cap T^W_{\Theta,s_m} = \emptyset$, alors ce flux d'informations viole la politique de flux d'informations SP.

   Considérons un objet $\Theta \in \mathsf{W}_m$ tel que $T^R_{\Theta,s_m} \cap T^W_{\Theta,s_m} = \emptyset$. D'après le lemme 1, il existe, dans l'état initial, un conteneur c_j et p informations atomiques $\{i_{j_1}, \dots, i_{j_p}\}$ tels que :

   $$\Theta = (\{i_{j_1}, \dots, i_{j_p}\}, c_j, T^R_{j_1,s_0} \cap \dots \cap T^R_{j_p,s_0}, T^W_{j,s_0}) \in \mathsf{W}_m$$

   Ainsi, d'après la règle de propagation, nous avons :

   $$T^R_{\Theta,s_m} \cap T^W_{\Theta,s_m} = \emptyset \Leftrightarrow T^R_{j_1,s_0} \cap \dots \cap T^R_{j_p,s_0} \cap T^W_{j,s_0} = \emptyset$$

   D'après la défintion 2.3.3, cela signifie qu'il n'existe pas de CCAL (I, C) dans la politique de flux d'informations SP tel que $\{i_{j_1}, \dots, i_{j_p}\} \subseteq I$ et $c_j = c_\Theta \in C$. D'après la définition de l'état initial du système s_0, si un tel CCAL existait, il devrait appartenir à tous les tags de sécurité en lecture des objets initiaux associés aux contenus $\{i_{j_1}\}, \dots, \{i_{j_p}\}$ et au tag de sécurité en écriture de c_Θ, ce qui contredit (1).

   L'objet Θ résulte donc d'une violation de la politique de flux d'informations SP puisque d'après cette politique, le conteneur c_Θ n'est pas autorisé à contenir un contenu généré à partir de toutes les informations atomiques $\{i_{j_1}, \dots, i_{j_p}\}$.

2. Nous montrons dans un deuxième temps que si l'exécution d'une commande $s_m \xrightarrow{\sigma_m} s_{m+1}$ viole la politique de flux d'informations, alors le flux de transition associé amène le système dans un état interdit où il existe au moins un objet $\Theta = (I_\Theta, c_\Theta, T^R_{\Theta,s_m}, T^W_{\Theta,s_m})$ tel que $T^R_{\Theta,s_m} \cap T^W_{\Theta,s_m} = \emptyset$.

   D'après la définition 2.2.1, si σ_m génère un flux d'informations qui viole la politique de flux d'informations alors le flux de transition $f(\mathsf{TBR}_m, \mathsf{TBW}_m)$ associé à σ_m génère au moins un objet $\Theta = (I_\Theta, c_\Theta, T^R_{\Theta,s_m}, T^W_{\Theta,s_m})$ dont le conteneur c_Θ contient I_Θ et l'association (I_Θ, c_Θ) n'est pas autorisée par la politique de flux d'informations SP. Cela signifie qu'il n'existe pas de CCAL (I, C) tel que $I_\Theta \subseteq I$ et $c_\Theta \in C$. Nous notons (ϵ) cette assertion et nous démontrons maintenant que $T^R_{\Theta,s_m} \cap T^W_{\Theta,s_m} \neq \emptyset$ est nécessairement vérifiée pour cet objet. Nous pouvons déduire du lemme 1 qu'il existe un ensemble d'informations atomiques $\{i_{j_1}, \ldots, i_{j_p}\}$ tel que :

$$I_\Theta = \{i_{j_1}, \ldots, i_{j_p}\} \text{ et } T^R_{\Theta,s_m} = T^R_{j_1,s_0} \cap \ldots \cap T^R_{j_p,s_0}$$

   Supposons que $T^R_{\Theta,s_m} \cap T^W_{\Theta,s_m} \neq \emptyset$. Ceci impliquerait $T^R_{j_1,s_0} \cap \ldots \cap T^R_{j_p,s_0} \cap T^W_{\Theta,s_m} \neq \emptyset$. Cela signifierait qu'il existe au moins un CCAL (I, C) tel que $I_\Theta \subseteq I$ et $c_\Theta \in C$, ce qui contredit l'assertion (ϵ).

□

Ce théorème de détection montre qu'en suivant la règle de propagation des tags de sécurité et en vérifiant, pour chaque flux d'informations élémentaire s'exécutant sur le système, la règle de légalité des flux d'informations, il est possible de détecter toutes les occurrences des violations de la politique de flux d'informations définie dans l'état initial. L'exemple suivant illustre l'évolution du système et le déroulement de l'algorithme de détection.

Exemple 9. *Considérons le système présenté dans les exemples 6 et 8 et illustré par la figure 2.5. D'après le théorème de détection, l'état s_1 est autorisé par la politique de flux d'informations SP si, pour chaque objet Θ de s_1, $T^R_{\Theta,s_1} \cap T^W_{\Theta,s_1} \neq \emptyset$.*

Supposons maintenant qu'une commande $s_1 \xrightarrow{\sigma_1} s_2$ soit exécutée et qu'elle génère un flux d'informations du contenu de c_2 vers le conteneur c_3 soit $C^R_{\sigma_1} = \{c_2\}$ et $C^W_{\sigma_1} = \{c_3\}$.

L'évolution du système qui découle de cette exécution est représentée sur la figure 2.5. L'état du système est maintenant défini par :

$$\begin{aligned} s_2 \quad = \{ \quad & (\{i_1\}, c_1, \{C_A, C_C\}, \{C_A, C_C\}), \\ & (\{i_1\}, c_2, \{C_A, C_C\}, \{C_B, C_C\}), \\ & (\{\mathbf{i_1}\}, c_3, \{\mathbf{C_A}, \mathbf{C_C}\}, \{C_B\}), \\ & (\{i_A\}, c_A, \{C_A\}, \{C_A\}), \\ & (\{i_B\}, c_B, \{C_B\}, \{C_B\}), \\ & (\{i_C\}, c_C, \{C_C\}, \{C_C\}) \ \} \end{aligned}$$

Cette dernière opération est autorisée par le contrôle d'accès puisque l'utilisateur Bob peut lire le fichier c_2 et copier son contenu dans c_3. Toutefois, ce flux d'informations génère une alerte puisque, pour le conteneur c_3, la règle de légalité de flux d'informations n'est pas vérifiée : $\{C_1, C_2\} \cap \{C_3\} = \emptyset$.

Cet état du système est bien caractéristique d'une violation de la politique de flux d'informations SP car la composition successive des deux flux d'informations élémentaires est ici équivalente à un flux d'informations $\{i_1\} \Rightarrow c_3$ qui est interdit par la politique. Dans l'état s_2, c_3 contient des données qui ont été générées à partir de l'information initialement présente dans c_1. Ce flux d'informations est interdit puisque, d'après la matrice de contrôle d'accès, aucun utilisateur ne peut à la fois lire le fichier c_1 et écrire dans le fichier c_3. Cette séquence d'opérations est une attaque à la confidentialité puisque Bob a lu le contenu initial de c_1 suite à sa lecture de c_2. Cette intrusion est détectée par notre algorithme de détection d'intrusions, quel que soit le scénario réel utilisé pour réaliser cette intrusion.

Supposons maintenant que la première commande exécutée à partir de l'état initial s_0 ait été un flux d'informations de l'interface de Charlie vers le conteneur c_2. Nous aurions alors eu une transition $s_0 \xrightarrow{\sigma_{0'}} s_{1'}$ vers un état $s_{1'}$ caractérisé par :

$$\begin{aligned} s'_1 \quad = \{ \quad & (\{i_1\}, c_1, \{C_A, C_C\}, \{C_A, C_C\}), \\ & (\{\mathbf{i_C}\}, c_2, \{\mathbf{C_C}\}, \{C_B, C_C\}), \\ & (\{i_1\}, c_3, \{C_A, C_B\}, \{C_B\}), \\ & (\{i_A\}, c_A, \{C_A\}, \{C_A\}), \\ & (\{i_B\}, c_B, \{C_B\}, \{C_B\}), \\ & (\{i_C\}, c_C, \{C_C\}, \{C_C\}) \ \} \end{aligned}$$

Après l'exécution de la deuxième opération $s_{1'} \xrightarrow{\sigma_1} s_{2'}$ (la copie du contenu de c_2 dans le conteneur c_3), le système aurait été amené dans l'état suivant :

$$\begin{aligned} s'_2 \quad = \{ \quad & (\{i_1\}, c_1, \{C_1, C_2\}, \{C_1\}), \\ & (\{i_C\}, c_2, \{C_C, C_1, C_2\}, \{C_2\}), \\ & (\{\mathbf{i_C}\}, c_3, \{\mathbf{C_C}, \mathbf{C_1}, \mathbf{C_2}\}, \{C_3\}), \\ & (\{i_A\}, c_A, \{C_A, C_1\}, \{C_A\}), \\ & (\{i_B\}, c_B, \{C_B, C_2, C_3\}, \{C_B\}), \\ & (\{i_C\}, c_C, \{C_C, C_1, C_2\}, \{C_C\}) \ \} \end{aligned}$$

Cet état est caractéristique d'une violation de la propriété d'intégrité puisque l'information courante de c_3 a été générée par Charlie et que cette association contenus/conteneurs n'est pas autorisée par la politique de flux d'informations SP. Cette intrusion est également détectée par notre algorithme puisque le conteneur c_3 possède des tags de sécurité associés disjoints : $\{C_C, C_1, C_2\} \cap \{C_3\} = \emptyset$.

2.4.2 Discussion

Supposons qu'un détecteur d'intrusion implémente notre modèle de détection. Ce détecteur propage des tags de sécurité et lève une alerte lorsque les

tags de sécurité en lecture et en écriture associés à un conteneur deviennent des ensembles disjoints, c'est-à-dire $T^R \cap T^W = \emptyset$. Notre théorème de détection peut alors être interprété de la manière suivante : une alerte est levée *si et seulement si* l'exécution d'une commande sur le système a engendré un flux d'informations qui n'est pas autorisé par la politique de flux d'informations spécifiée dans l'état initial. D'un point de vue théorique, ce théorème prouve à la fois la pertinence et la fiabilité de l'algorithme de détection des violations de la politique de flux d'informations spécifiée :

- La pertinence signifie l'absence de faux positif (ou fausse alarme). Si le système émet une alerte, il le fait à raison et cela implique la présence d'un flux d'informations violant la politique ;
- La fiabilité signifie l'absence de faux négatif. Si un flux d'informations viole la politique de sécurité durant l'exécution, alors une alerte sera émise.

Toutefois, une implémentation de notre algorithme ne peut, en pratique, assurer une détection strictement fiable et pertinente des intrusions. En effet, cela supposerait :

1. que tous les aspects de la politique de sécurité puissent être modélisés par un ensemble de CCAL ;
2. que tous les flux d'informations du système soient observables et discernables.

Concernant le premier point, nous n'affirmons pas ici que tous les aspects d'une politique de sécurité donnée puissent être exprimés sous la forme d'un ensemble de CCAL. Par exemple, les aspects concernant la disponibilité des informations ne peuvent être exprimés dans notre modèle de politique de flux d'informations. Nous supposons en outre que la politique de flux est correctement spécifiée par un administrateur, en positionnant les tags de sécurité dans l'état initial.

Le deuxième point est plus délicat car il est en pratique difficile de suivre précisément les flux d'informations au sein d'un système complet. En effet, seuls sont pris en compte les flux d'informations entre les conteneurs identifiés du système. Un suivi précis des flux d'informations suppose donc une identification précise des différents conteneurs. De manière générale, se pose le problème du niveau de granularité des conteneurs et du processus de détection. Par exemple, une implémentation au niveau du système d'exploitation considère typiquement des conteneurs de la taille d'un fichier ou d'une page mémoire. Certains flux d'informations ne sont donc pas pris en compte. Par exemple, les flux d'informations internes aux applications ne sont pas observables à ce niveau. De même, les différents flux d'informations entre les différents champs d'un seul fichier de base de données ne sont pas discernables puisque mettant en jeu un seul et unique conteneur.

Il est possible de résoudre certains de ces problèmes en affinant le modèle ou en changeant le niveau de granularité de l'implémentation. Idéalement chaque variable devrait être prise en compte ce qui suppose une implémentation au niveau langage. Une telle implémentation implique cependant une modification de toutes les applications du système, ce qui n'est pas toujours possible et se

révèle pénalisant en termes de performance. Nous proposons dans le chapitre suivant d'implémenter le modèle de détection à plusieurs niveaux de granularité et de faire collaborer les différents niveaux de détection.

Chapitre 3

Implémentation et résultats expérimentaux

L'un des objectifs de cette thèse est de proposer un mécanisme applicable aux systèmes opérationnels existants permettant de détecter un large spectre d'intrusions caractérisées par des violations de propriétés de confidentialité ou d'intégrité. Nous proposons ici une architecture de détecteur d'intrusions qui repose sur le modèle de détection paramétrée par la politique de sécurité présenté au chapitre 2. Le but est donc de suivre les flux d'informations sur des systèmes existants ce qui impose certaines restrictions que l'on peut regrouper au sein de deux types d'exigences :

- La solution proposée doit être suffisamment générique afin de couvrir un large spectre d'applications et de domaines d'utilisation. Cette exigence impose notamment les restrictions suivantes :
  - la solution proposée doit être compatible avec les systèmes existants et en particulier avec les composants réutilisables (COTS). Elle doit ainsi s'adresser aux OS existants (Linux, Windows, MacOSX, etc.) ainsi qu'aux architectures (applications web, modèle client serveur, etc.) et aux langages de programmation (C/C++, Java, PhP, SQL, etc.) couramment employés pour ce type de composants.
  - la solution proposée ne doit pas se restreindre à un seul type d'application. Les attaques auxquelles doivent faire face les systèmes opérationnels aujourd'hui déployés sont multiples et ciblent tous les différents composants d'un système. Certains scénarios d'attaques complexes peuvent même exploiter différentes vulnérabilités présentes sur différents composants du système.
- La solution proposée doit être la moins intrusive possible afin de limiter l'impact de la surveillance des flux d'informations sur le déploiement et l'utilisation des applications surveillées. Cette exigence impose notamment les restrictions suivantes :
  - la solution proposée doit nécessiter peu de modifications des composants

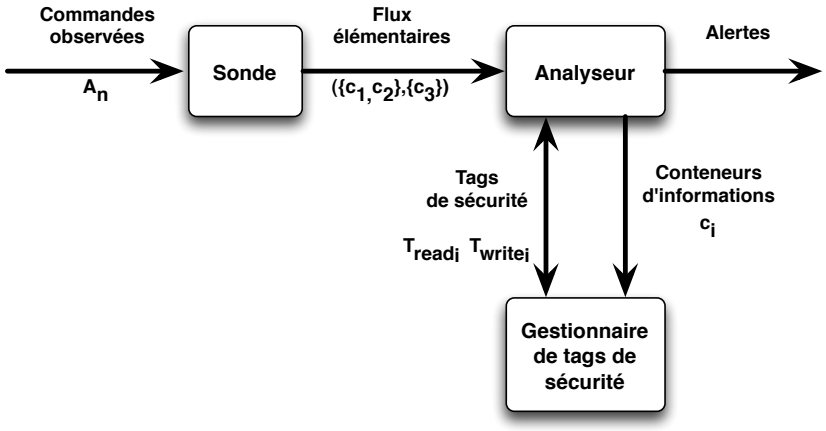

FIG. 3.1 – Spécification minimale d'un mécanisme de détection d'intrusions par contrôle des flux d'informations

utilisés. En particulier, la réécriture complète d'un composant n'est pas envisageable. En outre, la solution retenue doit prendre en compte le fait que le code source de certaines applications n'est pas toujours disponible.

– la solution proposée doit limiter le surcoût engendré par le suivi des flux d'informations afin d'envisager une utilisation en temps réel.

Afin de répondre à ces différentes exigences, nous présentons une architecture générique de détection d'intrusions paramétrée par la politique de sécurité et reposant sur le contrôle collaboratif des flux d'informations (section 3.1). Cette architecture se veut suffisamment générique pour être applicable à différents systèmes. Afin de valider cette architecture et de démontrer la faisabilité d'une telle approche, nous présentons dans les sections suivantes l'implémentation de prototypes de suivi global des flux d'informations pour les programmes s'exécutant sous Linux (section 3.2) et de suivi plus détaillé des flux d'informations pour les programmes Java (section 3.3).

3.1 Architecture générique

Nous proposons ici une architecture générique permettant d'implémenter un mécanisme de détection d'intrusions paramétrée par la politique de sécurité reposant sur le suivi des flux d'informations. Etant donné le but recherché et les contraintes imposées, une solution reposant uniquement sur la vérification statique des flux d'informations n'est pas envisageable pour les raisons évoquées au chapitre 1. Notre approche s'appuie donc essentiellement sur un suivi dynamique des flux d'informations, éventuellement complété par une analyse statique partielle de certains composants. Comme tout mécanisme de détection

d'intrusions comportemental, la solution retenue doit comporter au moins les dispositifs suivants :

- un dispositif permettant d'observer le comportement du système en fournissant un flux d'événements. Notre solution s'appuyant sur le suivi des flux d'informations, ce dispositif, qui constitue la sonde de notre IDS, est donc en charge du suivi des flux d'informations entre les différents conteneurs du système. Il interprète chaque opération du système observée en termes de flux d'informations élémentaire correspondant.
- un dispositif permettant la spécification du comportement attendu, ou autorisé, du système. Il s'agit dans notre cas de l'ensemble des flux d'informations, élémentaires ou composés, autorisés par la politique de flux. Les flux non explicitement spécifiés comme autorisés sont considérés comme interdits. Ces flux sont spécifiés à l'aide de tags de sécurité attachés à chaque conteneur d'information considéré. Ces tags contiennent l'ensemble des CCAL s'appliquant au conteneur et à son contenu courant.
- un dispositif de contrôle des flux d'informations qui, pour chaque flux d'informations élémentaire observé par la sonde, détermine la légalité, au regard de la politique de flux spécifiée à l'aide des CCAL, du flux d'informations élémentaire et des flux d'informations composés qui en découlent. Ce dispositif, qui constitue l'analyseur de notre IDS, propage les tags des conteneurs lus par le flux élémentaire vers les conteneurs dont le contenu est modifié. Cette propagation des tags permet de prendre en compte les flux d'informations composés. Le dispositif de contrôle des flux d'informations vérifie également que les données générées par le flux d'informations peuvent légalement être contenues dans chacun des conteneurs modifiés. Il doit pour cela vérifier que l'intersection des tags de sécurité est non vide, en accord avec l'algorithme de détection présenté au chapitre 2. Il génère une alerte pour chaque flux interdit détecté.

Cette liste n'est pas exhaustive : d'autres dispositifs peuvent être présents, suivant l'implémentation, permettant notamment le contrôle de l'IDS ou la collecte des alertes. Ces dispositifs complémentaires sont détaillés par la suite pour chacune des implémentations proposées. En revanche, les trois dispositifs listés précédemment doivent être présents et constituent donc une spécification minimale du mécanisme de détection, illustrée par la figure 3.1. Nous verrons par la suite qu'il est possible d'envisager la collaboration de plusieurs implémentations différentes de ces dispositifs qui doivent donc impérativement respecter une même spécification.

Nous détaillons dans les sous-sections suivantes chacun de ces dispositifs et l'impact de la granularité des conteneurs d'information considérés sur leur implémentation.

3.1.1 Gestion des tags de sécurité

D'après le modèle de détection d'intrusions présenté au chapitre 2, la politique de flux d'informations peut être spécifiée sous la forme d'un ensemble de CCAL. Chaque CCAL contient lui-même un ensemble d'informations atomiques

et un ensemble de conteneurs. Il représente un ensemble de flux d'informations, élémentaires ou composés, autorisés par la politique de flux. D'après la définition 2.2.1 du chapitre 2, chaque CCAL autorise l'ensemble des flux d'informations d'un contenu généré à partir d'une ou plusieurs informations atomiques présentes dans le CCAL vers chacun des conteneurs présents dans le même CCAL. Les CCAL sont donc les éléments constitutifs de la politique de flux d'informations, chacun d'entre eux spécifiant un ensemble de flux d'informations légaux en termes de sources (les informations atomiques) et de destinations (les conteneurs) possibles. Nous proposons d'implémenter l'ensemble des CCAL de la façon suivante :

- un identifiant unique est associé à chaque CCAL ;
- un tag de sécurité en lecture, noté T^Rdans le modèle du chapitre 2, est associé à chaque conteneur et mémorise l'ensemble des identifiants des CCAL relatifs au contenu courant ;
- un tag de sécurité en écriture, noté T^W dans le modèle 2, est associé à chaque conteneur et mémorise l'ensemble des identifiants des CCAL relatifs au conteneur.

Nous proposons de reprendre les choix de conception de Jacob Zimmermann pour Blare [Zim03] et d'implémenter les tags de sécurité sous la forme d'un tableau de bit (*bitmap*). Cette structure permet, lors de la propagation des tags et de la vérification de la politique, d'optimiser les opérations sur les tags qui se traduisent alors par des opérations booléennes binaires, appliquées à chacun des bits. La taille du tableau binaire (donc de l'ensemble des CCAL possibles) n'est pas imposée. Tous les composants d'un détecteur d'intrusions implémentant notre approche doivent bien évidemment respecter la même taille de tableau et s'accorder sur la signification des identifiants des différents CCAL.

Il n'est pas nécessaire à ce stade d'imposer de mécanisme permettant d'associer contenus et conteneurs à leur tag respectif, car ce mécanisme dépend fortement du type d'implémentation envisagée. Dans la plupart des cas, seul le conteneur est identifié : il peut s'agir d'un identifiant de noeud (*i-node*) pour les conteneurs du système d'exploitation, du nom ou de l'adresse d'une variable, etc. Le dispositif de gestion de tags de sécurité doit donc comprendre une fonction permettant, à partir de l'identifiant du conteneur, de retrouver le tag de sécurité associé à ce conteneur ainsi que celui associé à son contenu courant. Le contenu étant lui-même susceptible d'évoluer, le tag de lecture doit donc être mis à jour après chaque modification du conteneur.

La granularité des conteneurs d'information considérés n'a pas d'influence directe sur l'implémentation du mécanisme de gestion de tags. En revanche, elle influe sur la granularité des politiques de flux d'informations qui peuvent être surveillées à l'aide de ce mécanisme. Nous classons les conteneurs d'information en trois classes, suivant leur granularité :

- les conteneurs de forte granularité (fichiers, interfaces de type *socket*, pages mémoire, etc.) ;
- les conteneurs de faible granularité (variables, champs d'objet, lignes de fichiers, etc.) ;
- les conteneurs de très faible granularité (registres processeur, éléments de

la pile d'une machine virtuelle, etc.).
En associant un seul tag en lecture et un seul tag en écriture à chaque conteneur d'information, notre approche considère donc que toutes les données présentes dans ce conteneur doivent obéir aux mêmes règles du point de vue de la politique de flux d'informations. Ceci peut se révéler d'autant plus contraignant que la granularité des conteneurs d'information considérés est grande. Les politiques de flux nécessitant de prendre en compte différemment les données d'un même conteneur ne peuvent être gérées. Ainsi, plus la granularité du conteneur est importante, plus le nombre de politiques qu'il est possible de mettre en œuvre est restreint. A l'inverse, considérer des conteneurs de faible granularité permet de discriminer plus précisément les différentes données manipulées. En revanche, cela conduit à gérer un nombre plus important de tags de sécurité ce qui génère un surcoût plus important en occupation mémoire et une surcharge plus importante lors du suivi des flux d'informations et de la manipulation des tags de sécurité. A l'extrême, la distinction de chacun des octets de la mémoire (ou pire, de chacun des bits) conduit à un surcoût d'occupation mémoire considérable. Ce surcoût est d'ailleurs d'autant plus important que la taille des tags, donc de l'ensemble des CCAL, est importante. Or le nombre de CCAL gérés influe également sur les politiques de flux. Ainsi, on peut remarquer que, dans les travaux de l'état de l'art qui s'intéressent à des conteneurs de très faible granularité, les tags de sécurité sont souvent codés sur un seul bit, ce qui revient dans notre approche à ne considérer uniquement que deux CCAL. L'ensemble des politiques de flux est alors restreint aux politiques qui ne considèrent que deux classes d'information, par exemple $\{Secret, Public\}$.

Un compromis doit donc être trouvé entre la granularité des conteneurs considérés, le surcoût acceptable et le nombre de CCAL. Ce compromis n'est pas toujours facile à trouver pour un système complexe comportant de multiples programmes et dont la politique de flux globale nécessite de considérer des flux d'informations de granularités différentes. Plutôt que d'imposer une granularité à l'ensemble du système, il est possible d'envisager plusieurs niveaux de granularités, suivant les besoins de la politique de flux, ce qui permet d'adresser un nombre important de politiques de flux d'informations tout en minimisant le surcoût global du mécanisme de suivi des flux d'informations. Il est souhaitable de considérer un premier niveau de la politique de flux qui fixe des règles pour l'ensemble des conteneurs de forte granularité du système (fichiers, *socket*, pages mémoire des processus, etc.). Puis, pour certains conteneurs seulement, (par exemple, les bases de données ou la mémoire de certaines applications complexes) un second niveau de granularité, plus fin, est pris en compte. Cette solution est au cœur de l'approche de contrôle collaboratif des flux d'information que nous détaillons par la suite. Nous présentons dans les sections suivantes deux implémentations de notre approche permettant d'adresser différents niveaux de granularité de conteneurs :

- l'implémentation présentée en section 3.2 gère des conteneurs de forte granularité pour les programmes sous Linux ;
- l'implémentation présentée en section 3.3 gère des conteneurs de faible granularité pour les programmes Java ;

Nous ne présentons pas d'implémentation gérant des conteneurs de très faible granularité. Il est parfaitement envisageable d'appliquer notre approche pour de tels conteneurs mais cela nécessiterait, pour des raisons d'efficacité, un support matériel (l'implémentation serait alors en partie réalisée de façon matérielle). En outre, la taille des tags doit être restreinte pour les raisons évoquées précédemment.

De même, il est possible de décliner notre approche sur d'autres systèmes d'exploitation (Windows, BSD, MacOSX, etc.) ou sur d'autres langages (PHP, Perl, Python, Ruby, etc.) couramment utilisés dans les systèmes que nous étudions dans cette étude. Les différentes implémentations peuvent cependant varier suivant l'architecture du système surveillé et l'accès au code source et aux API du système.

3.1.2 Observation des flux d'informations

Le contrôle dynamique des flux d'informations suppose la détection des flux d'informations à l'aide d'une sonde. D'après le modèle présenté au chapitre 2, l'état du système évolue suite à l'exécution successive de commandes qui constituent la trace. Le rôle de la sonde est donc, à partir de l'observation de la trace du système, de fournir à l'analyseur une séquence de flux d'informations élémentaires correspondant à l'exécution de chaque commande. Le modèle n'impose aucune restriction sur la nature des commandes et sur l'interprétation en termes de flux d'informationss élémentaires. Ces deux points dépendent fortement de l'implémentation et de la granularité des conteneurs d'informations considérés. Dans le cas de conteneurs de forte granularité, les commandes peuvent être constituées par l'ensemble des appels système ou des fonctions internes du noyau permettant d'accéder aux conteneurs du système d'exploitation. Dans le cas de conteneurs de plus faible granularité, il s'agira des instructions du langage de programmation permettant d'accéder aux champs des objets, aux variables locales, etc.

Le choix de l'ensemble des commandes à observer constitue le premier élément à prendre en compte lors de l'implémentation d'une sonde pour notre approche de détection. Le second élément réside dans l'interprétation des commandes en termes de flux d'informations élémentaires. Il s'agit, pour chaque commande observable, de définir les éléments suivants :

- l'ensemble des conteneurs accédés en lecture, dont les contenus courants constituent les sources du flux d'informations élémentaire correspondant à la commande ;
- l'ensemble des conteneurs accédés en écriture qui constituent la destination du flux d'informations élémentaire correspondant à la commande.

La détermination de ces ensembles est fixée lors de l'implémentation à partir de l'analyse des commandes observées. Dans la plupart des cas, ce choix est trivial. Ainsi, l'observation de l'accès à un champ d'objet `b=a.c` correspond à un flux élémentaire du champ vers le conteneur affecté $a.c \rightarrow b$. Cependant, dans certains cas, le flux d'informations élémentaire ne peut être déterminé précisément, notamment lorsque les conteneurs considérés sont de forte granularité. Plusieurs

solutions peuvent être alors adoptées :

- considérer le cas le plus restrictif en sur-approximant le flux d'informations, au risque de générer des faux positifs ;
- considérer le cas le plus probable, au risque de générer des faux négatifs.

La détermination de l'ensemble des commandes observables et l'interprétation des commandes en termes de flux d'informations élémentaire influent sur la précision et la complétude de la sonde dans le suivi des flux d'informations. En outre, la précision et la complétude de la sonde influencent la précision et la complétude du processus de détection des intrusions. Il est parfois nécessaire de faire un compromis entre précision et complétude, notamment lorsque la granularité des conteneurs considérés est importante.

3.1.3 Contrôle des flux d'informations

Le dispositif de contrôle des flux d'informations joue ici le rôle d'analyseur. A partir de la séquence des flux d'informationss élémentaires fournie par la sonde, il effectue les opérations suivantes :

- il identifie, à l'aide du système de gestion de tags, les tags de sécurité des conteneurs et des contenus impliqués dans le flux élémentaire observé ;
- il détermine le tag de sécurité du nouveau contenu généré par le flux d'informations élémentaire observé et il le propage pour chacun des conteneurs modifiés ;
- il vérifie la légalité du flux d'informations élémentaire observé et de tous les flux composés qui en découlent à partir de la règle issue du théorème de détection proposé au chapitre 2.

Il s'agit du seul dispositif dont l'implémentation ne dépend pas du niveau de granularité des conteneurs. Un même système de détection implémentant notre approche ne devrait donc en toute rigueur compter qu'un unique analyseur alimenté éventuellement par différentes sondes. Pour des raisons d'efficacité, il est parfois souhaitable de dupliquer ce mécanisme. Il est alors nécessaire de recourir à un mécanisme de collaboration entre les différentes sondes et analyseurs afin de garantir la cohérence du processus de détection et des éventuelles alertes émises. Nous détaillons le principe d'un tel mécanisme dans la sous-section suivante.

3.1.4 Principe de contrôle collaboratif des flux d'informations

Il est parfois judicieux, au sein d'une même solution de détection d'intrusions par contrôle des flux d'informations, de recourir à différentes implémentations des dispositifs présentés dans les sections précédentes. En effet, les choix d'implémentation dépendent fortement de la granularité des conteneurs d'information que l'on souhaite observer. De plus, la diversité des politiques de flux qu'il est nécessaire de mettre en œuvre pour protéger un système complexe impose de surveiller les flux d'informations à différents niveaux de granularité. Dans ce contexte, nous proposons une solution comportant plusieurs systèmes de gestion de tags, travaillant à des niveaux de granularité différents. Ce choix impose

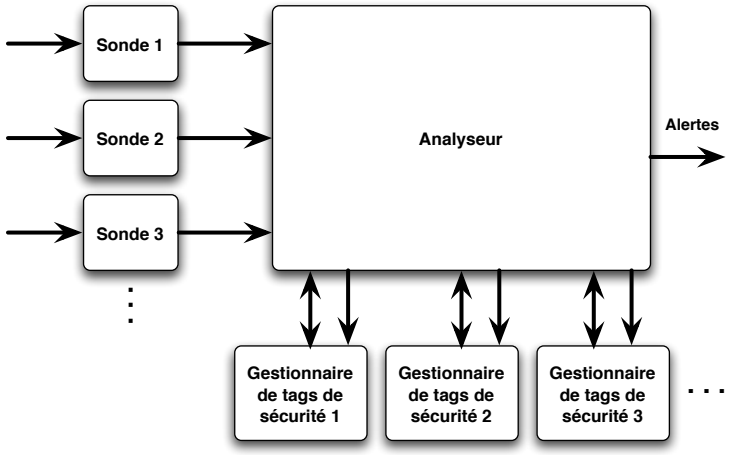

FIG. 3.2 – Architecture centralisée

naturellement de recourir à plusieurs sondes travaillant elles aussi à des niveaux de granularité différents, l'ensemble des commandes observées par une sonde étant lié au niveau de granularité des conteneurs d'information du système de gestion de tags associé à la sonde. L'analyseur étant le seul dispositif qui ne dépende pas de la granularité des conteneurs d'information observés, il est en théorie possible d'utiliser une architecture centralisée, illustrée par la figure 3.2, regroupant plusieurs sondes et gestionnaires de tags qui alimentent un unique analyseur.

L'inconvénient majeur d'un système reposant sur l'architecture centralisée réside dans la communication entre les différentes sondes et l'analyseur. En effet, cette architecture requiert que l'implémentation de l'analyseur soit découplée des implémentations des différentes sondes, qui peuvent se situer dans différentes couches logicielles ou matérielles, et communique par un système d'échange de messages. Cette solution, élégante d'un point de vue purement architectural, peut s'avérer en pratique pénalisante pour les performances. En effet, les sondes de suivi des flux d'informations sont caractérisées par le débit important des événements qu'elles doivent produire. Ce débit est d'autant plus important que la granularité des conteneurs surveillés est faible. Le surcoût généré par les communications entre les sondes et l'analyseur est important et justifie le recours à une deuxième architecture, illustrée par la figure 3.3, où l'analyseur est dupliqué. Ce type d'architecture permet d'implémenter l'analyseur au plus près des sondes situées à un même niveau logiciel ou matériel. Ainsi, les communications entre les sondes et les analyseurs peuvent être simplifiées ce qui permet de limiter l'impact sur les ressources. Par exemple, les différentes sondes implémentées dans l'espace noyau d'un système d'exploitation peuvent être regroupées avec un analyseur, lui aussi implémenté dans l'espace noyau, tandis que les différentes sondes applicatives, implémentées dans l'espace utilisateur, sont

regroupées autour d'un ou plusieurs analyseurs également implémentés dans l'espace utilisateur. La communication entre les sondes et l'analyseur associé peut alors être réalisée par des appels de fonction au sein d'un même espace (utilisateur ou noyau) et évite ainsi le surcoût généré par les changements de contexte entre espace utilisateur et espace noyau.

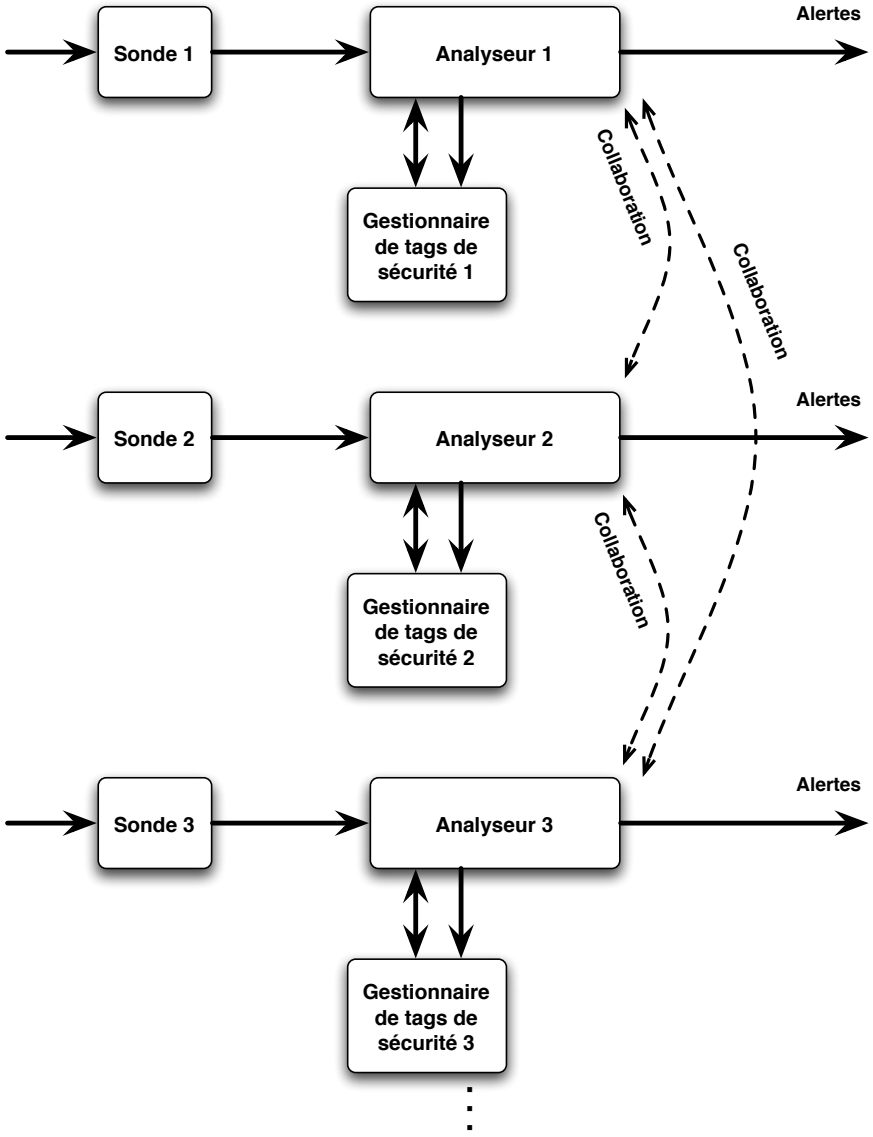

FIG. 3.3 – Architecture de collaboration

Le choix d'une architecture comportant plusieurs analyseurs nécessite un mécanisme de collaboration entre les différents analyseurs, afin de garantir la cohérence du processus de détection. Plusieurs solutions peuvent être envisagées :

- utilisation d'un mécanisme de communication entre analyseurs. Lorsqu'un ou plusieurs analyseurs souhaitent émettre une alerte, ce mécanisme permet, via un protocole d'accord, d'assurer la cohérence des réponses de tous les analyseurs. Il est par exemple possible de désigner un analyseur «maître», responsable de l'émission des alertes. Ce mécanisme nécessite l'envoie de messages entre les différents analyseurs mais seulement en cas d'alerte. L'hypothèse est faite que la plupart des flux d'informations observés sont légaux et par conséquent que le débit d'émission des alertes est beaucoup plus faible que celui des flux d'informations observés. Cette solution nous paraît cependant peut réaliste en raison de la complexité et du surcoût lié au protocole d'accord, bien qu'il soit limité par rapport à celui généré par la solution de l'architecture centralisée.
- ajout d'un dispositif supplémentaire de collecte et d'analyse des alertes. Ce dispositif joue le rôle du *manager* que l'on retrouve dans certaines architectures d'IDS utilisant plusieurs sondes. Il collecte les alertes émises par les différents analyseurs sur lesquels il effectue un pré-traitement avant de fournir un rapport synthétique à l'opérateur. Il peut notamment repérer et éliminer les doublons, tenter de corréler différentes alertes, échantillonner les alertes, etc.
- utilisation d'un mécanisme de pré-négociation. Ce mécanisme permet, grâce à une communication entre les analyseurs, de définir différentes zones ou domaines de surveillance répartis entre les différents analyseurs. Chaque analyseur est responsable d'un domaine de surveillance et n'émet que des alertes relatives à ce domaine de surveillance. Ce mécanisme doit s'assurer que les différents domaines de surveillance sont disjoints, ce qui permet de s'affranchir de post-traitement lors de l'émission des alertes. Chaque analyseur étant en charge d'un domaine de surveillance particulier, il n'émet que des alertes relatives à ce domaine : l'ensemble des alertes concernant le système surveillé correspond à l'union des alertes émises par les différents analyseurs. Ce mécanisme nécessite une communication entre analyseurs mais celle-ci ne s'effectue que lors de la phase d'initialisation, ce qui limite considérablement le surcoût nécessaire à la collaboration.

Ces différentes solutions ne sont pas exclusives et peuvent être combinées entre elles. Cependant, la troisième solution nous paraît convenir le mieux à notre approche. En effet, l'utilisation de différentes sondes est ici justifiée par la surveillance de conteneurs d'information de granularités différentes. Chaque sonde étant en charge de la surveillance d'un type de conteneur, le découpage en domaines de surveillance est naturellement déterminé par la granularité des conteneurs. De plus, cette solution permet de limiter le surcoût de la collaboration et ne nécessite pas de post-traitement sur les alertes émises par les différents analyseurs. Cette solution n'exclut pas l'utilisation d'un *manager*, qui pourrait, par exemple, corréler dans le temps les différentes alertes mais nous

nous sommes restreints dans cette thèse à l'étude et la conception des sondes et des analyseurs.

Nous proposons le mécanisme de collaboration suivant :

- Un analyseur maître, initialisé en premier, se charge de la répartition des domaines de surveillance. Lors de son initialisation, il est déclaré en charge de la surveillance de l'ensemble du système.
- Lorsqu'un analyseur esclave s'initialise, il effectue une demande de délégation pour un domaine de surveillance auprès de l'analyseur maître. L'analyseur maître restreint alors son domaine de surveillance et délègue la surveillance d'une partie de son domaine initial à l'analyseur esclave.
- Une fois l'ensemble des analyseurs initialisés, chaque analyseur surveille le domaine qui lui a été attribué et émet des alertes lorsqu'il détecte un flux d'informations interdit par la politique de sécurité.

La collaboration entre différents analyseurs assure une séparation des flux d'informations surveillés. En revanche, certains conteneurs, situés à l'interface entre les différents domaines de surveillance, doivent être gérés par plusieurs analyseurs. Un même gestionnaire de tags peut donc être amené à communiquer avec plusieurs analyseurs. Par exemple, un flux d'informations entre un conteneur de faible granularité et un conteneur de forte granularité nécessite qu'un même analyseur, responsable de la surveillance du flux d'informations observé, accède à la fois au tag de sécurité du conteneur de forte granularité et à celui du conteneur de faible granularité.

Nous décrivons par la suite un prototype de détection d'intrusions paramétrée par la politique de sécurité implémentant l'architecture proposée et reposant sur deux sous-systèmes de suivi des flux d'informations collaborant entre eux :

- Blare, un mécanisme de détection pour le noyau Linux qui assure le suivi des flux d'informations entre conteneurs de forte granularité (fichiers, *socket*, pages mémoire, etc.)
- JBlare, un mécanisme de détection pour les programmes Java qui assure le suivi des flux d'informations à l'intérieur des applications Java, entre conteneurs de faible granularité (champs d'objet, méthodes, etc.)

L'architecture de notre prototype, illustrée par la figure 3.4, correspond donc à un cas particulier de l'architecture générique présentée précédemment et illustrée par la figure 3.3.

3.2 Blare

Blare est un détecteur d'intrusions paramétré par la politique de sécurité développé initialement par Jacob Zimmermann [Zim03]. Cet IDS contrôle les flux d'informations des différentes applications s'exécutant sur un environnement Linux afin de détecter des intrusions. Celles-ci sont caractérisées par des violations d'une politique de flux définie au préalable à l'aide de tags de sécurité positionnés sur des conteneurs d'information de forte granularité : des fichiers et des *socket* réseau. Plusieurs expérimentations ont montré que Blare pouvait détecter un large spectre d'intrusions caractérisées par la lecture ou la

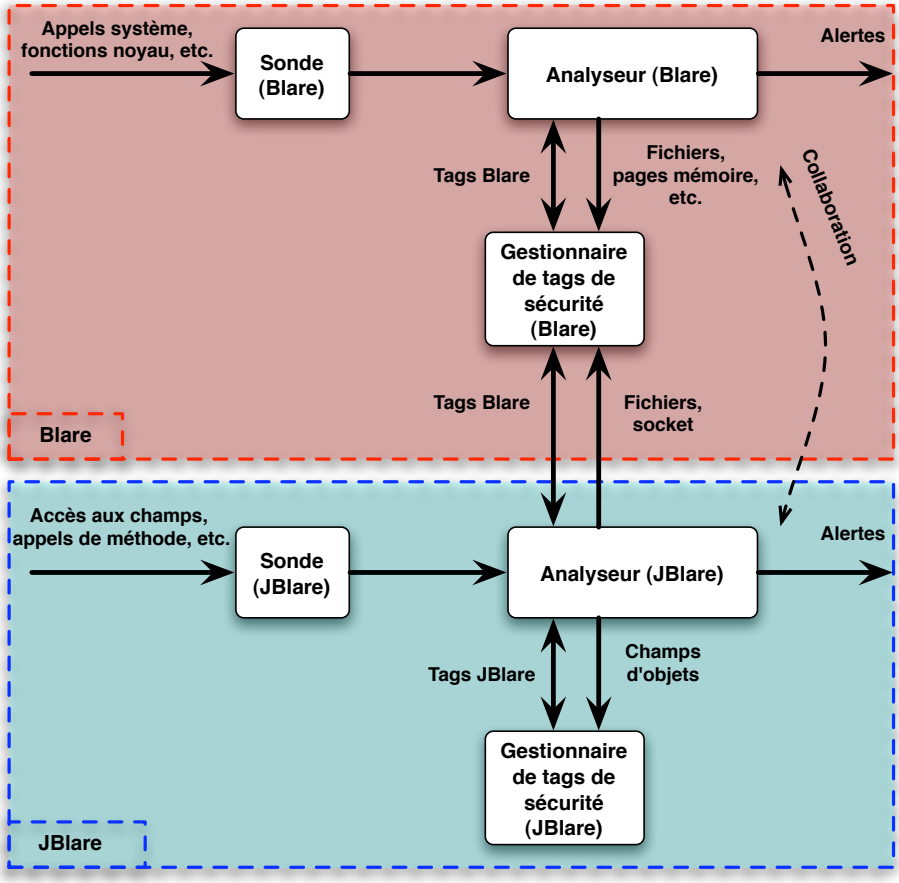

FIG. 3.4 – Architecture du prototype Blare/JBlare

modification illégale d'informations contenues dans des fichiers [ZMB03].

Le prototype présenté ici correspond à une nouvelle version de Blare, initialement conçu et développé par Jacob Zimmermann. Nous avons également participé à son développement, en collaboration avec l'auteur, en l'adaptant au modèle proposé. Comparé aux versions précédentes présentées dans les articles [Zim03, ZMB03], la conception a été radicalement modifiée afin de limiter le surcoût engendré par le suivi des flux d'informations entre conteneurs de niveau OS. Cette nouvelle version est donc principalement implémentée dans l'espace noyau. Elle comporte également quelques utilitaires et une bibliothèque de fonctions permettant aux utilitaires d'accéder facilement aux fonctionnalités de Blare à travers une interface de programmation (API). Blare est distribué sous licence GPL sous la forme de *patch* du noyau Linux (la dernière version supportée à la date de la rédaction de ce mémoire est la version 2.6.25 du noyau

officiel “vanilla”).

3.2.1 Architecture

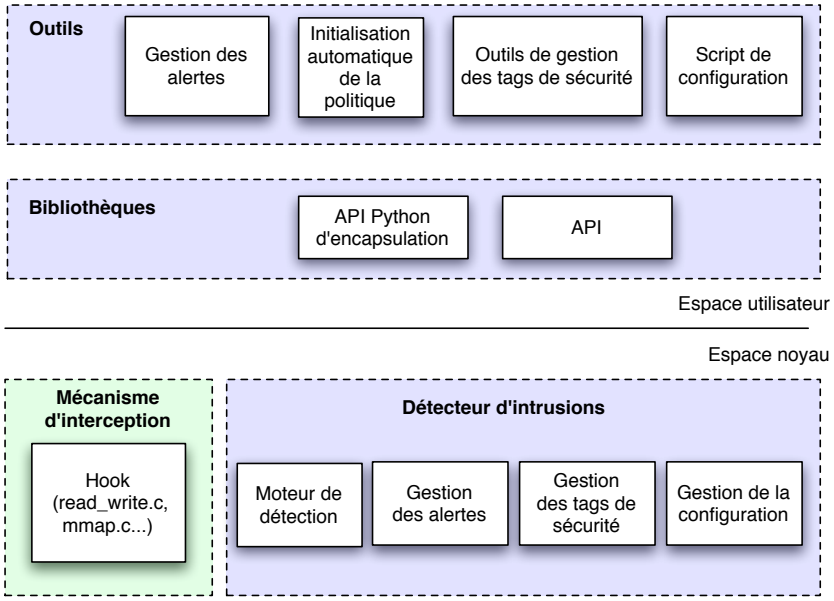

Fig. 3.5 – Architecture de Blare

La figure 3.5 illustre l’architecture de Blare. Cet IDS se présente sous la forme d’un moniteur interne au noyau Linux permettant le suivi et le contrôle dynamique des flux d’informations. Nous décrivons par la suite les différents éléments de Blare en regroupant la description des éléments appartenant aux trois dispositifs de notre architecture générique : la sonde, le gestionnaire de tags et l’analyseur.

3.2.1.1 Sonde

Le suivi des flux d’informations est assuré par deux sous-ensembles :

– les points d’accroches (*hook*) du mécanisme d’interception constituent le sous-ensemble permettant l’observation de la trace des commandes du système. Il s’agit en fait principalement de fonctions de la couche VFS (*Virtual File System*) du noyau Linux qui gèrent les accès à la plupart des conteneurs de niveau OS via la notion de fichier. Ces fonctions ont été instrumentées en modifiant le code source du noyau Linux. Certains conteneurs (*socket*, terminaux TTY,etc.) nécessitent l’utilisation d’autres fonctions en dehors de la couche VFS qui sont également instrumentées.

- chacun des points d'accroche évoqué précédemment effectue un appel aux fonctions du moteur de détection qui permettent de traduire la commande observée en termes de flux d'informations élémentaires entre conteneurs, suivant les paramètres passés à ces fonctions. L'étude des fonctions interceptées, qui correspondent ici aux commandes du modèle, montrent que celles-ci peuvent être interprétées selon deux types de flux :
  - la plupart des commandes génèrent un flux élémentaire simple d'un unique conteneur source vers un unique conteneur destination. Il s'agit par exemple de la fonction `write` permettant d'écrire dans un fichier et qui se traduit donc par un flux d'informations de la mémoire du processus vers le fichier.
  - certaines commandes génèrent un flux élémentaire comportant deux conteneurs sources et un conteneur destination. Ce type de flux permet par exemple de modéliser la modification d'un fichier en mode d'ajout (*append mode*) ou l'accès à la mémoire du processus. Nous considérons alors que la modification n'entraîne pas la suppression totale du contenu initialement présent dans le conteneur modifié. Par conséquent, la modification est interprétée par un flux d'informations dont les sources sont le contenu initial du conteneur modifié et le contenu utilisé pour effectuer la modification.

Lorsqu'un programme de l'espace utilisateur accède à un conteneur du système, il effectue un appel système qui exécute une fonction du noyau, elle-même interceptée par le mécanisme de suivi des flux d'informations. Ainsi, toutes les commandes générant des flux d'informations entre objets du système sont observées par la sonde qui les interprète en termes de flux d'informations élémentaires qu'elle fournit à l'analyseur.

3.2.1.2 Analyseur

L'analyseur est implémenté par des fonctions ajoutées au noyau Linux qui constituent le moteur de détection. On retrouve au sein de ce sous-système les fonctions qui assurent la propagation des tags de sécurité pour chaque flux élémentaire fourni par la sonde. L'analyseur implémente également l'algorithme de détection issu du modèle présenté au chapitre 2. Dès qu'une violation de la politique de flux est détectée, ce qui se caractérise par une intersection entre un tag de lecture et un tag en écriture réduite à l'ensemble vide, une alerte est émise via le sous-système de gestion des alertes.

3.2.1.3 Gestionnaire de tags de sécurité

Conformément à la spécification de l'architecture générique présentée en section 3.1, les tags de sécurité sont implémentés sous la forme d'un tableau de bit. Dans Blare, ce tableau est codé sur quatre entiers, ce qui permet d'envisager 128 CCAL différents sur une architecture 32 bit. Les tags sont liés aux conteneurs du système via l'ajout de pointeurs dans les structures du noyau Linux correspondant à ces conteneurs. Blare supporte les différents types de fichiers,

les *socket* TCP, certains mécanismes de communication entre processus (IPC), etc. La mémoire de chaque processus est modélisée par un conteneur unique, ce qui peut conduire à des imprécisions dans la composition des flux d'informations au sein d'un même processus. En effet, pour Blare, chaque processus constitue une boîte noire du point de vue de son espace mémoire. Tous les flux d'informations «sortants», ayant pour conteneur source la mémoire du processus, sont donc composés avec tous les flux «entrants» précédents qui ont pour destination la mémoire du processus. Cette imprécision a motivé l'implémentation d'un second prototype, JBlare, présenté en section 3.3 et qui permet de suivre plus précisément les flux d'informations au sein des applications Java.

Les tags de sécurité sont représentés par des structures situées en mémoire, dans l'espace noyau. Blare propose également un ensemble d'utilitaires permettant, depuis l'espace utilisateur, d'accéder aux tags de sécurité d'un certain nombre de conteneurs, afin notamment de spécifier la politique de flux d'informations. En toute rigueur, l'accès à ces utilitaires devrait être restreint à un mode particulier d'administration mais cette restriction n'est pour l'instant par intégrée au prototype. L'accès aux tags depuis l'espace utilisateur est assuré par un pseudo-sytème de fichiers permettant d'exporter les tags des fichiers et des *socket* dans l'espace utilisateur. L'utilisation de ce système de fichiers est facilité par une bibliothèque de fonctions (API). Les utilitaires proposés comportent notamment une application en mode console permettant de lister les tags de sécurité d'un conteneur, appelé `lsref`, dont le rôle est similaire à celui de la commande `ls` sous Unix. Un autre utilitaire, `setref`, permet de positionner des tags sur un conteneur et joue donc un rôle similaire aux commandes `chown` et `chmod` sous Unix. L'initialisation automatique de la politique à partir de l'interprétation des droits du contrôle d'accès discrétionnaire, dont le principe est présenté au chapitre 2, est assurée par un script Python. Cette fonctionnalité est détaillée dans la sous-section 3.2.3.

3.2.2 Autres services

Blare comprend également d'autres fonctionnalités non listées dans le schéma d'architecture générique et que nous présentons ici :

- La gestion des alertes est assurée par un ensemble de fonctions ajoutées au noyau Linux et par des utilitaires en mode utilisateur. Les alertes peuvent être émises sur une console de surveillance ou redirigées vers un système de journaux. Un mode d'audit permet d'enregistrer la trace des flux élémentaires observés dans un fichier d'audit.
- Le paramétrage de l'IDS est réalisé par un ensemble de fonctions ajoutées au noyau Linux qui forment le sous-système de gestion de la configuration. Le paramétrage est réalisé depuis l'espace utilisateur à l'aide d'utilitaire dialoguant avec le sous-système du noyau via le pseudo-système de fichiers `/proc`.
- La collaboration avec d'autres modules de détection est assurée par un mécanisme de délégation de la surveillance des flux d'informations. Un domaine de surveillance est ici caractérisé par un ensemble de processus

ou *thread*. Blare agit en tant que système «maître» : il surveille par défaut l'ensemble des flux d'informations entres les différents conteneurs du système. Il peut en revanche déléguer la surveillance des flux d'informations relatifs à un domaine : il n'effectue alors aucune propagation des tags de sécurité vers et depuis les conteneurs représentant l'espace mémoire des différents processus et thread du domaine délégué. La délégation s'effectue à la demande des systèmes «esclaves», via un nouvel appel système. JBlare, présenté en section 3.3, utilise ce mécanisme pour obtenir la délégation de la surveillance des processus Java. Le recours à la délégation suppose une relation de confiance entre le système «maître» et les système «esclaves». Afin d'éviter qu'un logiciel malveillant demande la délégation sur un domaine dans le but de supprimer la surveillance des flux d'informations de ce domaine, un mécanisme d'authentification et de contrôle d'intégrité devrait être mis en place. Toutefois, cette fonctionnalité n'est pas implémentée dans cette version du prototype.

3.2.3 Initialisation de la politique

Afin d'aider l'administrateur système dans sa tâche de spécification de la politique de flux, nous proposons un mécanisme d'initialisation automatique de la politique de flux à partir d'une interprétation des droits du contrôle d'accès. Ce mécanisme est implémenté par un script `Python` qui parcourt l'ensemble des fichiers et positionne les tags de chacun d'entre eux selon le principe présenté au chapitre 2. Afin d'optimiser le processus de génération de la politique, la librairie fournissant l'API de Blare a été encapsulée afin de permettre l'appel natif aux fonctions de l'API depuis le script Python. A titre d'exemple, l'initialisation de la politique pour un système Linux Debian de 180 000 fichiers prend une minute sur une machine virtuelle VMware hébergée sur une station Intel Pentium D, le système hôte disposant de 2 Go de mémoire vive et le système invité de 256 Mo. La politique de flux ainsi générée constitue une interprétation possible de la politique de contrôle d'accès en termes de politique de flux d'informations. D'autres mécanismes d'interprétation, pour l'instant non-implémentés, peuvent être envisagés afin de générer automatiquement une politique initiale. Cette politique peut ensuite être affinée ou modifiée manuellement par un administrateur, par exemple à l'aide de la commande `lsref`.

3.3 JBlare

3.3.1 Motivations

Les résultats obtenus à l'aide de Blare [Zim03, ZMB03] montrent que l'approche de détection d'intrusions paramétrée par la politique de sécurité permet effectivement de détecter un large spectre d'intrusions sur un système complet tout en minimisant l'impact sur le système surveillé. L'implémentation proposée impose un nombre limité de modifications sur le système (en l'occurence,

quelques modifications du noyau de l'OS), elle est compatible avec les applications existantes et le surcoût engendré par la surveillance reste limité (environ 5%). Toutefois, le principal inconvénient de Blare, que l'on retrouve dans la plupart des solutions de contrôle de flux implémentées au niveau du système d'exploitation, réside dans la granularité des conteneurs d'informations considérés et donc dans la granularité des flux d'informations qui peuvent être pris en compte. En effet, Blare ne considère que des conteneurs que nous qualifions de forte granularité comme un fichier complet. En particulier, Blare considère la mémoire d'un processus comme un unique conteneur d'informations. Bien qu'il soit possible de raffiner cette vue en distinguant par exemple les différentes pages de l'espace mémoire d'un processus, ce type d'implémentation montre ici ses limites dès lors qu'il s'agit de suivre les flux d'informations internes à une application. Ainsi, Blare considère chaque processus comme une «boîte noire» du point de vue de son espace mémoire. Par conséquent, tous les flux sortant du processus (par exemple, l'écriture dans un fichier ou l'émission d'un paquet réseau) sont considérés comme dépendant causalement de tous les flux entrants (par exemple la lecture d'un fichier ou la réception d'un paquet réseau) observés au préalable. Considérons par exemple un processus qui effectue successivement les appels système suivants :

1. `read(fd1, buf1, nbyte1)`
2. `read(fd2, buf2, nbyte2)`
3. `write(fd3, buf3, nbyte3)`
4. `write(fd4, buf4, nbyte4)`

Dans le cas général, les différents paramètres (descripteurs de fichiers, adresses de tampon mémoire et nombres d'octets transmis) sont tous différents. Soit c_1, c_2, c_3 et c_4 les conteneurs d'informations correspondant aux fichiers désignés respectivement par les descripteurs de fichiers $fd1$, $fd2$,$fd3$ et $fd4$. Soit c_m le conteneur d'informations correspondant à l'espace mémoire du processus. Considérons, pour simplifier notre exemple, que la trace observée (c'est-à-dire la séquence des appels système) débute dans l'état initial, lors de l'initialisation de l'IDS. Avant l'exécution de la trace, le contenu de chaque conteneur c_n se réduit donc à une unique information atomique i_n. Blare interprète alors chaque commande de la trace en termes de flux d'informations élémentaires et considère donc la séquence suivante de flux d'informations :

1. $\{i_1, i_m\} \rightarrow \{c_m\}$
2. $\{i_2, i_1, i_m\} \rightarrow \{c_m\}$
3. $\{i_2, i_1, i_m\} \rightarrow \{c_3\}$
4. $\{i_2, i_1, i_m\} \rightarrow \{c_4\}$

Cette interprétation amène les remarques suivantes :

– Blare considère ici que les flux entrants modifient le contenu précédent de l'espace mémoire du processus mais ne suppriment pas totalement ce contenu. En termes de flux d'informations, Blare interprète donc la première et la seconde commande comme un flux d'informations possédant

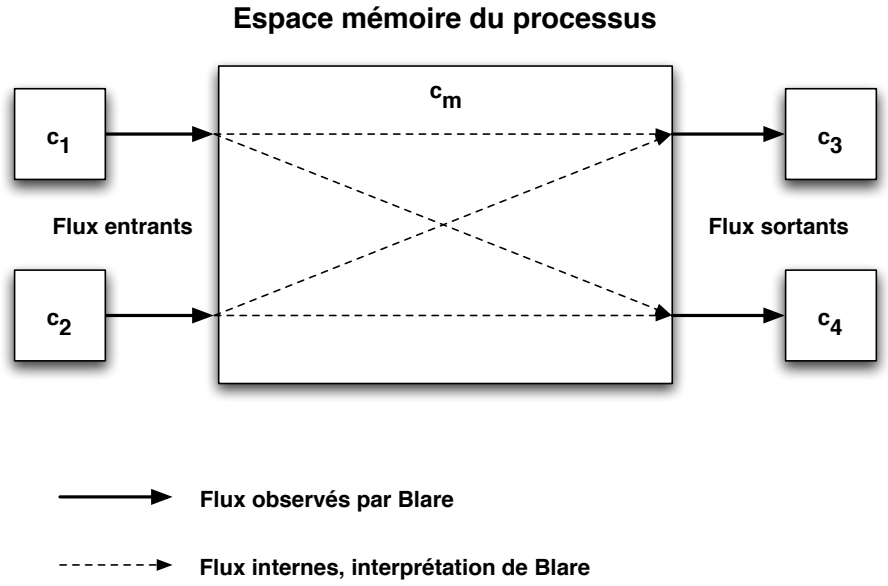

FIG. 3.6 – Interprétation des flux internes par Blare

deux conteneurs sources : le conteneur effectivement lu et le conteneur de destination (ici l'espace mémoire du processus). Ainsi, le contenu source du premier flux d'informations comprend i_1, le contenu du fichier accédé en lecture, mais également c_m, le contenu de l'espace mémoire du processus qui est modifié par le flux d'informations.

– Du fait de la composition des flux d'informations élémentaires et de la granularité des conteneurs surveillés, Blare considère qu'à l'issue de l'exécution de la trace, les contenus des fichiers c_3 et c_4 ont été générés à partir des données contenues initialement à la fois dans c_1, c_2 et c_m. Cette vision restrictive est illustrée par la figure 3.6.

Pour la plupart des processus surveillés, ce niveau de granularité est adapté : les processus manipulent des fichiers de même sensibilité et les hypothèses restrictives sur les flux internes des applications sont vérifiées. En revanche, ces hypothèses ne sont pas toujours vérifiées pour certaines applications complexes dont le comportement n'est pas pris en compte correctement par Blare. Ces applications peuvent accéder à des conteneurs de sensibilités différentes et doivent par ailleurs assurer un cloisonnement entre certaines données qu'elles manipulent. Ainsi, en reprenant l'exemple précédent, il se peut que les flux internes à l'application respectent un cloisonnement entre les données des fichiers c_1 et c_2. Les flux d'informations effectivement réalisés sont alors différents de l'interprétation faite par Blare et correspondent par exemple à ceux illustrés par la figure 3.7.

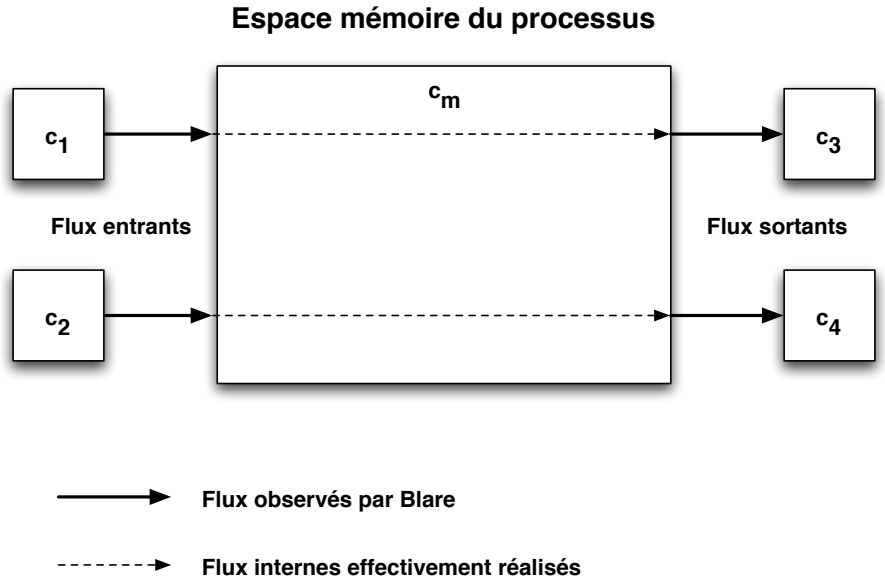

FIG. 3.7 – flux d'informations internes effectivement réalisés

La non-prise en compte des flux internes des processus, entre conteneurs de plus faible granularité, limite la portée de Blare :

- en sur-approximant les flux d'informations effectivement réalisés, Blare peut émettre des fausses alertes, par exemple lorsqu'un même processus accède à des conteneurs de sensibilités différentes tout en assurant une séparation entre les flux d'informations internes.
- Blare ne peut surveiller des politiques de flux définissant la légalité des flux internes des applications. Ces politiques de flux nécessitent en effet de distinguer les différents flux d'informations au sein d'un même processus. Ainsi, Blare ne peut distinguer les différents flux d'informations au sein d'une application web accédant à une base de données stockée dans un fichier unique.
- Enfin, la forte granularité des flux surveillés par Blare constitue un frein à l'utilisation des règles d'exception ou de déclassification. Un processus peut en effet recourir à un mécanisme de contrôle de mot de passe ou à des primitives de chiffrement, de signature, etc. Ces différents cas de figure, évoqués au chapitre 1, nécessitent de spécifier des exceptions à la règle de propagation des tags de sécurité. Bien qu'il existe une dépendance causale entre conteneurs, la quantité d'informations transmise peut être considérée comme suffisamment faible pour éviter qu'un attaquant ne retrouve les données protégées. Il est donc parfois nécessaire d'ignorer certains flux d'informations. L'utilisation de mécanismes d'exception dans Blare nécessiterait de définir des règles par processus, les flux internes n'étant pas

surveillés. Or, l'utilisation de règles d'exception peut conduire à des faux négatifs si un attaquant profite d'une règle d'exception trop permissive. Il est donc nécessaire de les utiliser avec parcimonie et d'en limiter la portée, par exemple pour une fonction particulière. Cela impose donc de pouvoir suivre et discriminer les différents flux internes des processus.

Afin de couvrir un spectre plus large de politiques de flux et d'augmenter la précision de Blare, nous proposons donc de compléter ce prototype par un deuxième détecteur d'intrusions, JBlare, qui permet de suivre les flux d'informations internes des applications lorsque cela s'avère nécessaire. En raison du surcoût engendré par le suivi des flux internes, l'utilisation de JBlare est restreinte à certaines applications complexes, par exemple, un serveur web ou un serveur d'applications. Afin de garantir un suivi des flux d'informations pour l'ensemble des applications du système, JBlare collabore avec Blare. Ces deux prototypes constituent alors une solution de détection paramétrée par la politique de sécurité utilisant un contrôle collaboratif des flux d'informations, dont l'architecture générique a été présentée en section 3.1. La section suivante présente l'architecture et les choix d'implémentation de JBlare.

3.3.2 Choix d'implémentation

Le suivi des flux d'informations internes aux applications nécessite de prendre en compte les conteneurs que nous qualifions de faible granularité (typiquement, des variables ou des champs d'objet et de classe dans le cadre de la programmation impérative ou orientée objet). La solution retenue pour le suivi des flux internes doit donc s'intéresser aux conteneurs définis dans les langages de programmation. Classiquement, les différents langages de programmation sont regroupés suivant trois niveaux :

- les langages de haut-niveau, utilisés lors du développement des applications. Ils peuvent reposer sur différents paradigmes (impératif, objet, fonctionnel, etc.) et permettent au programmeur de s'abstraire des détails inhérents au fonctionnement de la machine. Le code source des applications est généralement exprimé dans un langage de haut-niveau. Ce type de langage est ensuite compilé vers un langage de plus bas niveau, langage machine (C/C++, Ada, Fortran, etc.) ou langage intermédiaire (Java, Python, C#, etc.). Il peut également être directement interprété via un programme adéquat (BASIC, PHP, Javascript, etc.)
- les langages de bas-niveau, proches de l'architecture. Il s'agit du code objet ou du langage machine directement interprétable par le microprocesseur. Ce type de langage englobe également le langage d'assemblage ou assembleur qui représente le langage machine sous une forme lisible par un humain.
- les langages de niveau intermédiaire, issus de la compilation de langages de haut niveau comme Java ou C#. Ces langages constituent une forme intermédiaire, notamment en termes de complexité, entre les langages de haut-niveau et le langage machine. Ce type de langage peut être utilisé à des fins de compatibilité ou d'optimisation. Il est ensuite compilé, généra-

lement dynamiquement lors de l'exécution de l'application, vers le langage machine ou interprété. Le langage de la machine virtuelle Java (*Java Virtual Machine Language* ou *bytecode Java*) ainsi que le langage défini par Microsoft pour son *framework* `.NET` (*Common Intermediate Language*) constituent des exemples de l'utilisation d'un tel langage.

Le code source des applications n'étant pas toujours disponible, nous avons écarté les solutions reposant sur le code source exprimé dans un langage de haut-niveau. Le suivi des flux d'informations au niveau du langage machine est possible, notamment en instrumentant le code machine des applications. Toutefois, une telle approche peut s'avérer difficile à mettre en œuvre en raison de la complexité du code machine et du fossé sémantique entre les concepts exprimés dans le langage de haut-niveau et le code machine. Les langages intermédiaires ou *bytecode* constituent donc, lorsqu'ils sont utilisés, un niveau d'implémentation idéal pour le contrôle des flux d'informations :

- le *bytecode* est toujours disponible, l'application étant distribuée directement sous cette forme ou sous la forme de code source de haut-niveau qu'il est possible de compiler vers le *bytecode* ;
- la complexité du *bytecode* est réduite par rapport au langage machine et le fossé sémantique qui le sépare du langage de haut niveau utilisé lors du développement de l'application est moins important.

La plupart des langages utilisés dans les applications web (Java, C#, Python, etc.) ayant recours à une forme de langage intermédiaire, nous avons décidé d'implémenter un prototype reposant sur un langage de niveau intermédiaire. Plus précisément, nous avons choisi de nous intéresser au *bytecode* des applications écrites pour la plateforme Java. Ce langage est en effet couramment utilisé dans les applications web et repose sur un *bytecode* dont la spécification est très bien documentée. L'approche utilisée peut cependant être reprise pour implémenter d'autres détecteurs d'intrusions, par exemple pour le *Common Intermediate Language* ou le *bytecode* Python. Les applications Java sont souvent considérées comme plus «sûres» car la plateforme Java effectue des contrôles stricts sur le code source et sur le *bytecode*, notamment en ce qui concerne le type des données manipulées et le contrôle du flot d'exécution. Ces applications sont donc moins sensibles aux attaques de type *buffer overflow* ou *format string attack*. Seule la machine virtuelle Java (*Java Virtual Machine*) et les fonctions natives peuvent être éventuellement vulnérables à ce type d'attaque. En revanche, les applications web écrites en Java sont vulnérables aux attaques par injection, notamment aux attaques de type *SQL injection*, ainsi qu'aux différentes attaques visant la logique applicative, qui sont aujourd'hui courantes[1]. Il est donc nécessaire de vérifier la légalité des flux d'informations internes aux applications Java afin de détecter de telles intrusions.

Deux types de solutions, évoquées au chapitre 1, peuvent être adoptées pour suivre les flux d'informations au niveau du *bytecode* Java des applications :

- modifier une machine virtuelle Java (JVM) existante ou proposer une nouvelle implémentation d'une JVM permettant de suivre les flux d'informa-

[1] `http://www.owasp.org/index.php/Top_10_2007`

tions à l'aide d'un moniteur d'exécution ;
- instrumenter le *bytecode* Java afin d'y inclure les mécanismes de suivi et de vérification des flux d'informations.

La première solution permet d'envisager un suivi précis des flux d'informations car elle permet d'avoir potentiellement accès aux différentes structures de données de la JVM. Toutefois, cette solution impose un certain nombre de contraintes :

- la solution devient dépendante d'une implémentation particulière de la JVM ce qui limite la portabilité ;
- le code d'une JVM peut s'avérer complexe, en particulier lorsque les différents moyens de génération (interpréteur, compilateur à la volée (*Just In Time compiler*) doivent être supportés.

Nous avons donc préféré utiliser principalement l'intrumentation de *bytecode* pour JBlare, ce qui permet d'assurer une compatibilité accrue et de limiter la complexité d'implémentation. Toutefois, notre solution nécessite quelques modifications du code de la JVM et des bibliothèques système fournies dans le *Java Runtime Environment*. Nous précisons par la suite les modifications à apporter, dont nous nous sommes efforcés de minimiser le nombre afin de faciliter l'implémentation sur diverses JVM. Notre prototype supporte actuellement la version 6 de OpenJDK[2], la JVM *open-source* de SUN.

JBlare repose donc principalement sur l'instrumentation de *bytecode* ou *ByteCode Instrumentation*. Cette technique consiste sous Java à modifier le contenu des différents fichiers de classe issus de la compilation d'un programme écrit en Java. Chaque fichier de classe contient les données relatives à une classe Java ainsi que le *bytecode* correspondant à chaque méthode définie explicitement dans la classe. La structure d'un fichier de classe, définie par la spécification Java [LY99] comprend notamment les éléments suivants :

- un ensemble de constantes (*constant pool*) comprenant notamment les chaînes de caractères désignant les différents éléments de la classe ;
- les drapeaux d'accès à la classe (*public*, *final*, etc.)
- le nom de la classe ;
- le nom de la classe mère dont hérite la classe ;
- un tableau des noms des différentes interfaces implémentées par la classe ;
- une liste des différents champs, de classe ou d'instance de classe, définis dans la classe ;
- un liste des différentes méthodes, de classe ou d'instance de classe, définies dans la classe ;
- une liste des attributs de la classe.

Chaque classe est chargée en mémoire dynamiquement par un chargeur de classes, plusieurs chargeurs de classes pouvant cohabiter au sein d'une même JVM. Les différents chargeurs de classes sont organisés suivant une hiérarchie, chaque chargeur de classes étant responsable d'un domaine particulier. Les chargeurs de classes doivent obéir à des règles définies dans la spécification de Java [LY99], notamment en ce qui concerne la délégation. Le système est

[2] `http://download.java.net/openjdk/jdk6/`

complexe et constitue une source d'erreurs pour les programmeurs d'applications Java qui souhaitent mettre en place leur propre chargeur de classes. Au lancement de la JVM, trois chargeurs de classes différents sont utilisés :

- le chargeur de classes initial ou *bootstrap class loader*, écrit en langage natif et qui est responsable du chargement des premières classes constituant le cœur du langage Java comme `java.lang.Object`.
- le chargeur de bibliothèques externes responsable du chargement des classes contenues dans les bibliothèques d'extension ;
- le chargeur système responsable du chargement de l'ensemble des classes restantes, en l'absence de chargeurs de classes définis par les applications.

Seul le chargeur système peut être remplacé par un ou plusieurs chargeurs de classes définis par l'application. Celle-ci peut ainsi assurer une séparation entre différents espaces de noms de classes. Cette technique permet également de modifier les classes à la volée, lors de leur chargement.

Plusieurs stratégies peuvent donc être adoptées pour instrumenter les classes Java :

- les classes peuvent être instrumentées au préalable, en modifiant le contenu de chaque fichier de classe contenu dans les différents répertoires et fichiers d'archives JAR ;
- les classes peuvent être instrumentées dynamiquement, «à la volée», lorsque les chargeurs de classes effectuent le chargement du contenu des fichiers de classe dans la mémoire de la JVM.

L'instrumentation dynamique peut elle-même être réalisée de différentes manières :

- le chargeur système peut être remplacé par un chargeur permettant d'instrumenter les classes de l'application ;
- un agent d'intrumentation de classes peut être déclaré lors du chargement de la JVM. Le contenu des fichiers de classe est alors passé en paramètre à la fonction `transform` de cet agent dès que la classe est chargée en mémoire par un chargeur de classes autre que le chargeur initial.

Ces différentes solutions ont chacune leurs avantages et leurs inconvénients. Ainsi, l'instrumentation statique des fichiers de classe permet de réaliser l'instrumentation «hors-ligne», lors du déploiement de l'application, ce qui limite le surcoût lié au suivi des flux d'informations. En effet, l'instrumentation est une étape relativement longue qui pénalise l'application lors du chargement de la classe, si elle est effectuée dynamiquement. De plus, cette technique est la seule applicable pour certaines classes chargées par le *bootstrap classloader*. En revanche, elle nécessite d'identifier tous les différents fichiers de classe. Lorsqu'une nouvelle application est déployée, l'administrateur doit identifier ces différents fichiers et les instrumenter. Il existe donc un risque que certaines classes échappent à l'instrumentation, suite à un oubli involontaire ou consécutivement à une attaque permettant à un attaquant de provoquer le chargement de classes non-instrumentées.

L'instrumentation dynamique permet quant à elle de s'assurer que toutes les classes chargées (sauf pour celles chargées par le *bootstrap classloader*) ont été instrumentées. Elle présente cependant l'inconvénient majeur d'engendrer un

surcoût lors du chargement des classes. De plus, le remplacement du chargeur de classes système permet de supporter différentes versions de Java (ce mécanisme étant intégré depuis la version 1.0 de Java). Toutefois, cette solution, simple lorsque seul le chargeur système est utilisé, devient relativement compliquée dès lors que l'application définit elle-même un ou plusieurs chargeurs de classes. L'utilisation d'un agent d'instrumentation évite d'interférer avec le mécanisme de délégation des chargeurs de classes puisque la JVM fournit le code de chaque classe chargée, quel que soit le chargeur de classes utilisé. Ce mécanisme n'est cependant disponible que sur les JVM qui implémentent cette fonctionnalité introduite dans la version 1.5 de Java[3].

Compte tenu des avantages et des inconvénients de ces différentes solutions, nous avons choisi d'utiliser une solution hybride :

- certaines classes, notamment celles chargées par le *bootstrap classloader*, sont instrumentées statiquement et marquées à l'aide du mécanisme d'annotation introduit dans Java 1.5. Les classes des applications déployées peuvent également être instrumentées de la sorte pour optimiser le temps de chargement.
- un agent d'instrumentation est utilisé pour instrumenter les classes qui n'ont pas été instrumentées statiquement. L'agent détecte les classes pré-instrumentées grâce à l'annotation qui leur a été ajoutée lors de la phase d'instrumentation statique. Cette vérification permet de s'assurer qu'une classe n'est instrumentée qu'une seule fois et accélère le temps de chargement des classes.

Cette solution permet d'instrumenter l'intégralité des classes chargées par la JVM tout en minimisant le temps de chargement.

Il existe différents *framework* permettant d'instrumenter le *bytecode* des classes Java, les plus utilisés étant Javassist[4], BCEL[5] et ASM[6]. Les premières versions de JBlare utilisaient Javassist car cette solution permet de définir facilement les modifications à apporter. Celles-ci étant spécifiées directement en Java, il n'est donc pas nécessaire de maîtriser le *bytecode* produit. Ce choix a permis de valider l'approche mais nous avons décidé d'utiliser un autre *framework* en raison des limites intrinsèques de Javassist. En effet, Javassist présente des restrictions sur les modifications qu'il est possible d'effectuer sur les classes instrumentées. Nous avons donc choisi de réécrire JBlare en utilisant ASM. Ce *framework* est connu pour sa souplesse et son efficacité en ce qui concerne les performances des classes instrumentées. ASM possède deux modes d'instrumentation de classes :

- Le premier mode correspond au patron de conception *Visitor*. Les différents champs de la structure qui représentent une classe Java ou les différentes instructions du *bytecode* des méthodes sont parcourus en séquence. Un module «visiteur» génère des événements pour chaque champ

[3] `http://java.sun.com/j2se/1.5.0/docs/api/java/lang/instrument/package-summary.html`

[4] `http://www.jboss.org/javassist/`

[5] `http://jakarta.apache.org/bcel/`

[6] `http://asm.objectweb.org/`

de la structure de la classe ou chaque instruction de la méthode. Ces événements peuvent être transmis à plusieurs modules qui peuvent ainsi modifier la structure de la classe ou le *bytecode* de ses méthodes. Il est nécessaire d'utiliser au moins deux modules «visiteur» particuliers :
- le `ClassReader`, un module permettant de lire le *bytecode* des classes, fourni sous la forme d'un fichier ou d'un tableau d'octets, et permettant de générer les événements correspondant aux différents champs et opérations présents dans la structure de la classe.
- le `ClassWriter`, un module permettant d'écrire le *bytecode* des classes modifiées, en fonction des événements fournis par les autres modules.

Afin de modifier la classe, il est nécessaire d'intercaler des modules de modification, appelés `ClassAdapter`, entre un module `ClassReader` et un module `ClassWriter`. De même, les modules permettent la modification des méthodes de chaque classe. Ces modules peuvent être chaînés, suivant une structure série ou parallèle, afin de simplifier la spécification des modifications à apporter. Concrètement, lors de la réception d'un événement, un module de modification peut :

1. transmettre directement l'événement au module suivant, sans modifier la classe ou la méthode instrumentée ;
2. prendre en compte l'événement et apporter des modifications sur la partie de la classe concernée par l'événement ;
3. ne pas transmettre l'événement, ce qui permet de supprimer certains éléments de la classe ou certaines instructions des méthodes de la classe.

Le module peut également générer de lui-même de nouveaux événements, ce qui permet d'ajouter des éléments à la classe. Nous détaillons par la suite le fonctionnement de la classe `JBlareClassAdaptor`, le module de transformation de JBlare.

- Le second mode construit une représentation de la structure de la classe sous la forme d'un d'arbre. Il est ensuite possible de parcourir les différents éléments de l'arbre, de les modifier ou de les supprimer. Ce mode permet de spécifier des modifications complexes, qu'il est difficile ou impossible de spécifier à l'aide du premier mode. En revanche, ce mode d'instrumentation est plus lent. Les modifications que nous voulions apporter étant réalisables à l'aide du premier mode, nous n'utilisons pas ce second mode d'instrumentation dans la version actuelle de JBlare. Il peut toutefois s'avérer nécessaire d'utiliser ce mode pour réaliser une analyse statique des classes instrumentées.

JBlare est donc constitué de deux éléments :
- un *package* de classes écrites en Java utilisant le *framework* ASM pour instrumenter les classes Java ;
- un *patch* de la JVM (openJDK version 6) apportant un petit nombre de modifications nécessaires pour instrumenter les classes système.

Le *package* jBlare, illustré par le diagramme de classes de la figure 3.8, comprend six classes regroupées au sein d'une archive JAR :

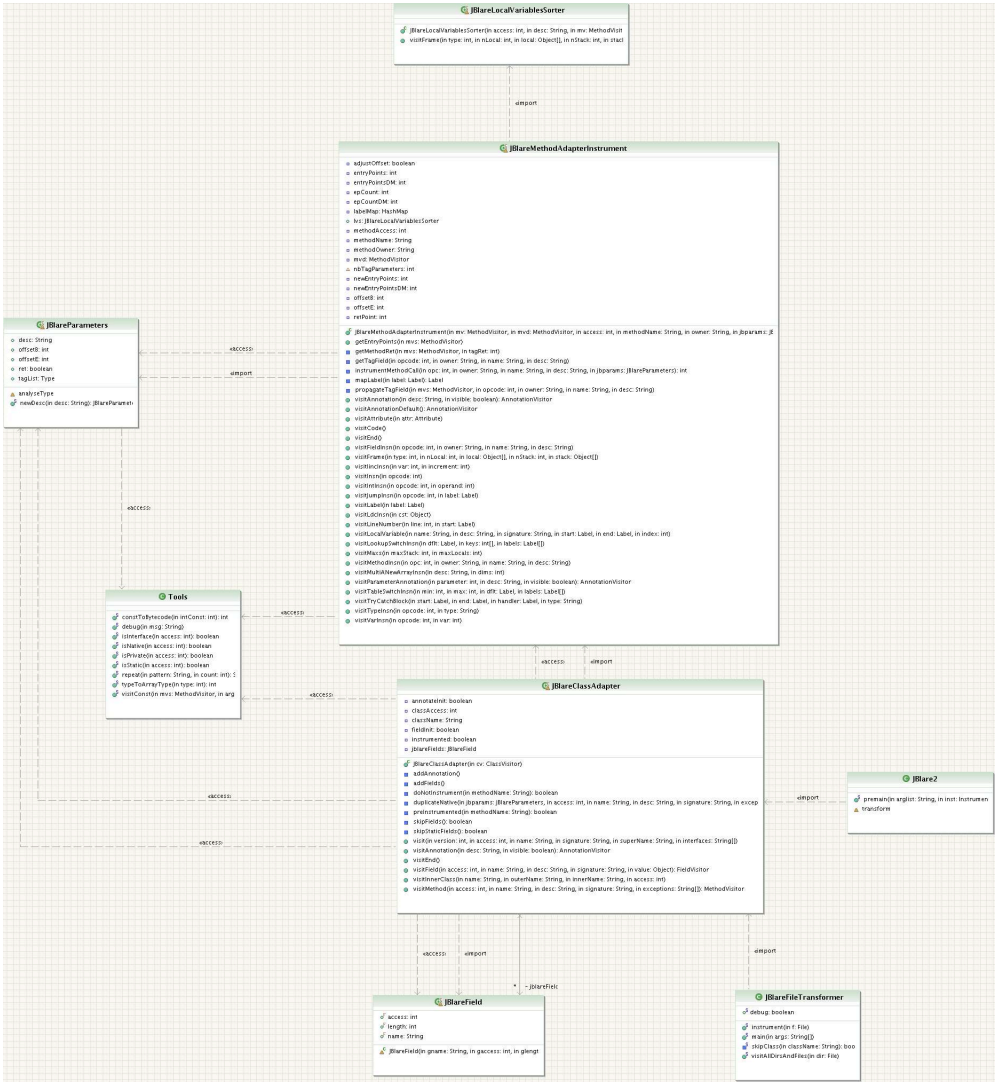

FIG. 3.8 – Diagramme de classes de JBlare

– la classe `JBlare` implémente l'interface `ClassFileTransformer` et constitue l'agent d'instrumentation permettant d'instrumenter dynamiquement les classes lors de leur chargement dans la JVM.
– la classe `JBlareFileTransformer` permet d'instrumenter statiquement les classes Java. Cette classe contient une fonction *main* permettant d'exécuter JBlare depuis un terminal. Le nom des fichiers de classe à instrumenter ou du répertoire contenant les fichiers de classe à instrumenter est passé

en paramètre.
- la classe `JBlareClassAdapter` étend la classe `ClassAdapter` de ASM. Elle permet de modifier les différents éléments d'un fichier de classe en fonction des événements qui lui sont envoyés par un objet `ClassReader`, lui même instancié dans la classe `JBlare` ou `JBlareFileTransformer`.
- la classe `JBlareMethodAdapter` étend la classe `MethodAdapter` de ASM. Elle permet de modifier le *bytecode* de chaque méthode de la classe instrumentée en fonction des événements qui lui sont envoyés par un objet `MethodVisitor` instancié dans la classe `JBlareClassAdapter`.
- les classes `JBlareFields` et `JBlareParameter` sont utilisées par les autres classes pour représenter les champs d'objet et de classe ainsi que les paramètres des méthodes qui sont instrumentées.

Nous détaillons par la suite l'architecture de la solution de détection d'intrusions mise en place par l'instrumentation, notamment à l'aide des classes `JBlareClassAdapter` et `JBlareMethodAdapter`. Nous précisons également les modifications apportées à la JVM.

3.3.3 Architecture

Le mécanisme de suivi et de contrôle des flux d'informations mis en place par le module JBlare repose sur l'insertion, au sein du code des applications et des classes système Java, d'instructions *bytecode* permettant d'assurer les fonctions des trois dispositifs de notre architecture générique évoquée en section 3.1 :
- l'observation et l'interprétation en termes de flux d'informations des commandes du système, assurées par la sonde ;
- la gestion des tags de sécurité, assurée par le gestionnaire de tags de sécurité ;
- la propagation des tags de sécurité et la détection des flux d'informations illégaux, assurées par l'analyseur.

Nous présentons par la suite les différents conteneurs d'informations pris en compte par JBlare ainsi que les mécanismes de gestion des tags de sécurité associés. Nous précisons ensuite comment les différentes fonctions évoquées sont implémentées lors de l'instrumentation des champs des classes et des objets ainsi que lors de l'instrumentation des méthodes de classe et d'objet.

3.3.3.1 Conteneurs d'informations pris en compte et gestion des tags de sécurité

Java supporte deux types de données, donc de conteneurs d'informations :
- les données de type primitif, `int`, `char`, etc.
- les données de type référence, qui correspondent à l'adresse d'une structure contenant plusieurs données de type primitif. Ces données correspondent sous Java aux classes, aux objets (ou instances de classes) et aux tableaux.

Nous avons choisi d'interpréter les conteneurs d'informations de la manière suivante :

- Nous appelons conteneurs d'informations atomiques les conteneurs d'informations susceptibles de recevoir des données de type primitif et nous souhaitons associer un tag de sécurité à chacun de ces conteneurs.
- Nous appelons conteneurs d'informations structurés les conteneurs d'informations susceptibles de recevoir des données de type référence. Nous ne souhaitons pas, à proprement parler, associer des tags de sécurité à chacun de ces conteneurs. En effet, nous les considérons comme des collections de conteneurs atomiques, chaque conteneur atomique d'un conteneur structuré possédant son propre tag de sécurité. Par exemple, chaque champ de type primitif d'un objet Java constitue un conteneur atomique, porteur d'informations, les références à l'objet n'étant qu'un moyen d'accéder à la structure (l'objet) englobant les différents conteneurs atomiques (les champs de type primitif).

Les différents conteneurs d'informations manipulés par le *bytecode* Java peuvent être regroupés en trois catégories :

1. les variables globales constituées par les champs des classes et des objets.
2. les variables locales des méthodes. Cette catégorie de conteneurs d'informations comprend également les paramètres et la valeur de retour des méthodes.
3. les éléments de la pile d'exécution de la JVM relative à une méthode Java.

Nous proposons dans JBlare d'associer un tag de sécurité à chacun des conteneurs d'informations atomiques de la première catégorie et nous précisons le traitement de ces conteneurs en section 3.3.3.2. Les conteneurs d'informations de la seconde catégorie sont pris en compte via les notions de point d'entrée et de point de sortie lors de l'instrumentation des méthodes explicitée dans la section 3.3.3.3. Les conteneurs d'informations de la troisième catégorie ne sont en revanche pas traités explicitement par l'instrumentation dynamique de JBlare. La prise en compte de ces conteneurs pourrait intervenir dans une phase d'analyse statique des méthodes, non implémentée à l'heure actuelle.

Les tags de sécurité sont quant à eux implémentés sous la forme d'un tableau binaire (*bitmap*) de quatre entiers longs. Ce type de structure permet d'assurer la compatibilité avec les tags de sécurité de Blare, notamment lors des échanges avec le gestionnaire de tags de sécurité de Blare évoqués en section 3.3.3.4. La propagation des tags de sécurité repose sur un mécanisme de copie à la modification (*Copy On Write*). Lors de la propagation des tags de sécurité, seule la référence pointant sur le tag de sécurité est dupliquée, ce qui limite le surcoût lié à la copie des différents éléments du tableau. Ce dernier doit en revanche être dupliqué lorsque la valeur du tag de sécurité doit être modifiée.

Nous souhaitons spécifier la politique de sécurité à l'extérieur des applications ou à l'interface entre les conteneurs de forte granularité et les conteneurs de faible granularité. Nous pensons en effet que, dans le contexte de cette étude, la spécification de la politique de flux d'informations pour chaque conteneur interne des applications n'est pas une hypothèse réaliste. Les tags de sécurité manipulés par JBlare sont donc issus de la propagation des tags de sécurité gérés

par Blare, définis à l'extérieur ou à l'interface des applications Java instrumentées. Seuls les tags de sécurité en lecture sont propagés aussi nous n'avons donc pas implémenté de tags de sécurité en écriture pour les conteneurs internes des applications Java dans la version actuelle de JBlare. Toutefois il est parfaitement envisageable d'associer un deuxième tag de sécurité à chaque conteneur afin de spécifier des politiques de flux internes aux applications.

3.3.3.2 Instrumentation des champs d'objet et de classe

Les différents champs atomiques des classes et des objets (ou instances de classes) de Java constituent l'ensemble des conteneurs de première catégorie. Pour chaque conteneur atomique de la classe ou de l'objet considéré (de type `int`, `char`, `bool`, etc.), nous associons un champ contenant le tag de sécurité atomique associé à ce conteneur. Le nom de ce champ est constitué du nom du champ correspondant au conteneur atomique préfixé par une chaîne de caractères permettant d'identifier les tags de sécurité.

L'ajout de champs de classe et d'objet est une opération simple à réaliser à l'aide de l'instrumentation statique ou dynamique des classes Java. Cependant, nous souhaitions instrumenter l'intégralité des classes Java manipulées par la JVM or certaines classes chargées par le *bootstrap classloader* lors de l'initialisation de la JVM ne peuvent être modifiées sans apporter de changement à la JVM. En effet, pour ces classes (par exemple, `java.lang.String` , `java.lang.Integer`, etc.), les indices des tableaux des différents champs et méthodes de la classe sont implémentés «en dur» dans le code de la JVM. L'ajout d'un champ nécessite donc de décaler ces indices dans le code source de la JVM et de la recompiler. Ces modifications sont distribuées sous la forme d'un *patch* pour la JVM du projet open-JDK.

L'implémentation des tags de sécurité relatifs aux champs d'objet ou de classe sous la forme d'un champ situé au sein de la même classe ou du même objet que le conteneur auquel il se rapporte simplifie le mécanisme de gestion des tags de sécurité. Ainsi, lorsque la sonde détecte une commande impliquant un flux d'informations vers ou depuis ce type de conteneur, par exemple, une instruction `GETFIELD` ou `PUTFIELD` du *bytecode* JAVA, le mécanisme d'instrumentation dispose de tous les éléments pour retrouver simplement le tag de sécurité du conteneur d'informations concerné :

- le nom de la classe et le nom du conteneur font partie des opérandes de l'instruction ;
- la référence de l'objet (dans le cas d'un champ d'objet) est présente sur la pile d'exécution.

Le tag peut alors être retrouvé à l'aide d'une instruction `GETFIELD` ou `GETSTATIC`, le nom du tag étant obtenu en préfixant le nom du conteneur surveillé par la chaîne de caractères de reconnaissance des tags. Il est en outre parfois nécessaire de dupliquer ou de permuter les différents éléments de la pile à l'aide des instructions adéquates.

Les champs d'objet ou de classe constitués par des tableaux de type primitif constituent un cas particulier de conteneur de première catégorie. Les différents

éléments de ces tableaux sont en effet des conteneurs atomiques et nous devrions en toute rigueur associer un tag de sécurité à chacun de leurs éléments. Néanmoins, il n'est pas possible d'associer conteneurs atomiques et tags de sécurité au sein du même tableau en raison de l'incompatibilité intrinsèque des différents types de données. De plus, en pratique, l'ensemble des données d'un même tableau relève généralement de la même politique de flux. Nous associons donc dans JBlare un tag de sécurité à chaque champ d'objet ou de méthodes constitué d'un tableau de conteneurs atomiques. Cette solution n'est pas optimale, notamment en raison de la possibilité de passage par référence du tableau ou de l'un de ses sous-éléments dans le cas d'un tableau à plusieurs dimensions. Toutefois, elle permet en pratique de détecter correctement un certain nombre de flux d'informations et de détecter les intrusions qui en découlent. Le traitement approprié des tableaux nécessiterait soit une modification de la JVM soit l'utilisation d'un mécanisme natif afin d'associer un tableau de tags de sécurité à chaque tableau de conteneur atomique. Ce mécanisme devrait notamment permettre de retrouver le tableau de tags associé à un tableau de conteneurs atomiques à partir de la référence du tableau de conteneurs atomiques. L'implémentation d'un tel mécanisme constitue un axe de recherche pour de futurs travaux.

3.3.3.3 Instrumentation des méthodes d'objet et de classe

Les différentes variables locales constituent la deuxième catégorie de conteneurs d'informations prise en compte par JBlare. Les méthodes contiennent l'intégralité du code exécuté par les applications Java. L'instrumentation des méthodes constitue donc l'étape la plus importante de l'instrumentation réalisée par JBlare. Le suivi précis des flux d'informations au sein des méthodes Java nécessite en toute rigueur de prendre en compte les différents éléments stockés sur la pile d'exécution. Il est alors nécessaire de modifier la JVM ou d'implémenter un mécanisme natif de pile permettant d'associer un tag de sécurité à chaque élément de la pile d'exécution. Il est ensuite nécessaire de modifier la pile de tags pour chaque instruction modifiant la pile d'exécution. En raison de la complexité et du surcoût générés par ce type de solution, nous avons préféré utiliser un modèle qui sur-approxime les flux d'informations au sein de chaque méthode. Nous identifions, pour chaque méthode, les éléments suivants :

- les différents conteneurs destinations des flux d'informations entrants, que nous appelons les points d'entrée de la méthode ;
- les différents conteneurs sources des flux d'information sortants, que nous appelons les points de sortie de la méthode.

Le comportement interne de la méthode est modélisé par la relation de dépendance entre les points de sortie et les points d'entrée de la méthode. Nous prenons en compte les différents points d'entrée suivants :

- les paramètres de type primitif (ou constitués par un tableau de type primitif) de la méthode, stockés dans les premières variables lors de l'appel de la méthode instrumentée ;

- les valeurs de retour des méthodes appelées au sein de la méthode instrumentée ;
- les valeurs des champs atomiques d'objet ou de classe accédés en lecture au sein de la méthode instrumentée.

De même, nous prenons en compte les points de sortie suivants :

- les paramètres de type primitif (ou constitués par un tableau de type primitif) des méthodes appelées au sein de la méthode instrumentée ;
- la valeur de retour de la méthode instrumentée, si elle correspond à un conteneur atomique ;
- les valeurs des champs atomiques d'objet ou de classe modifiées au sein de la méthode instrumentée.

La levée et l'interception des exceptions constituent également des points de sortie et d'entrée de la méthode instrumentée mais nous ne prenons pas en compte ce type de flux d'informations dans la version actuelle de JBlare.

Le comportement interne de la méthode est modélisé par la relation de dépendance entre les points de sortie et les points d'entrée de la méthode. Plusieurs solutions peuvent être envisagées pour déterminer cette relation :

- utiliser une sur-approximation des flux internes aux méthodes. Cette sur-approximation consiste, comme le fait Blare pour les applications, à considérer que chaque point de sortie dépend de tous les points d'entrée atteints au préalable.
- utiliser des signatures établies au préalable par une analyse de code manuelle ou à l'aide de méthodes d'analyse statique ;
- utiliser conjointement un mécanisme d'analyse statique à l'instrumentation de la classe.

JBlare utilise par défaut la première solution. Il peut également avoir recours à des signatures pour spécifier les flux internes de certaines méthodes. Les signatures permettent ainsi de prendre en compte les méthodes nécessitant une opération de déclassification, par exemple, des méthodes effectuant la vérification d'un mot de passe ou des opérations de cryptographie. L'utilisation conjointe d'un mécanisme d'analyse statique n'est pas implémentée dans la version actuelle de JBlare mais l'architecture de notre solution de détection a été pensée afin d'inclure facilement cette fonctionnalité. Cependant, l'utilisation conjointe d'un mécanisme d'analyse statique avec l'instrumentation dynamique des classes laisse supposer un surcoût important lors du chargement des classes. Un compromis doit donc être trouvé entre la précision du suivi des flux d'informations et le surcoût de la solution.

L'instrumentation des méthodes consiste donc à effectuer les opérations suivantes :

1. identifier les différents points d'entrée et de sortie ;
2. associer un tag de sécurité à chaque point d'entrée et de sortie ;
3. propager les tags de sécurité entre les points d'entrée et de sortie ;
4. vérifier la légalité des flux d'informations en s'assurant que les tags de sécurité ne contiennent pas des valeurs nulles, ce qui correspond à un ensemble de CCAL vide.

L'identification des points d'entrée et de sortie et l'association des tags de sécurité correspondent à la fonction de sonde de notre architecture d'IDS. Elles sont réalisées par des méthodes de la classe `JBlareMethodAdapter` qui détectent les occurrences des instructions du *bytecode* Java correspondant aux différents points d'entrée et de sortie.

La propagation des tags de sécurité et la détection des flux illégaux sont réalisées par un ensemble de méthodes statiques regroupées au sein de la classe `JBlareTag`. Nous avons ajouté cette classe dans l'archive `rt.jar` des classes système. Cette technique permet de faire appel aux méthodes de la classe `JBlareTag` depuis le code instrumenté. Une partie importante des modifications à apporter est ainsi factorisée et le processus d'instrumentation est simplifié.

La propagation des tags de sécurité correspondant aux champs d'objet et de classe est triviale, ces tags étant eux-mêmes stockés sous la forme de champs. La propagation des tags correspondant aux paramètres et valeurs de retour des méthodes nécessite quant à elle un mécanisme supplémentaire permettant de partager les tags de sécurité entre plusieurs méthodes. Nous avons décidé d'implémenter ce mécanisme en modifiant la signature des méthodes instrumentées afin de passer les tags en paramètres. Ainsi, pour chaque méthode comportant des paramètres ou une valeur de retour atomique, nous ajoutons des paramètres pour les tags de sécurité correspondant aux différents paramètres ou valeurs atomiques de la fonction instrumentée. Le tag éventuel correspondant à la valeur de retour est passé en paramètre à l'aide d'un mécanisme d'indirection qui permet à la méthode appelante de récupérer la valeur spécifiée dans la méthode appelée. Les méthodes natives ne peuvent être modifiées de la sorte qu'à condition que le code natif correspondant, écrit en C ou C++, soit également modifié. Par défaut, JBlare ne modifie pas la signature des méthodes natives et crée des méthodes enveloppantes (technique de *wrapping*).

3.3.3.4 Collaboration avec Blare

La collaboration avec Blare comporte deux aspects :

- la demande de délégation pour les processus et *thread* correspondant à l'application Java surveillée par JBlare ;
- la propagation des tags de sécurité de JBlare vers et depuis les tags de sécurité de Blare, lorsque l'application surveillée par JBlare accède aux conteneurs de forte granularité gérés par Blare.

Afin d'effectuer une demande de délégation, nous avons modifié le processus d'initialisation de la JVM pour qu'il exécute l'appel système d'enregistrement et de demande de délégation auprès de Blare. Une fois l'appel effectué, Blare ne propage plus les tags de sécurité vers et depuis l'espace mémoire du processus de la JVM et de ces différents *thread*. Il incombe alors à JBlare de propager les tags de sécurité au sein de l'application Java.

JBlare est également responsable de la propagation des tags de sécurité depuis ou vers les conteneurs du système d'exploitation accédés par l'application Java surveillée. Afin d'assurer cette propagation, nous avons instrumenté les fonctions natives, écrites en C, permettant aux applications Java d'accéder aux

ressources du système sur lequel s'exécute la JVM, par exemple, les fichiers ou les *socket* réseau. Le code instrumenté fait appel à des fonctions de l'API de Blare permettant de lire ou de modifier les tags de sécurité des fichiers et des *socket*. Les tags de JBlare sont donc propagés vers les tags de Blare ou *vice-versa*.

L'instrumentation statique et manuelle des fonctions natives permet de propager efficacement les tags de sécurité entre les différents niveaux de suivi des flux d'informations. Toutefois, se pose le problème de l'exhaustivité de l'instrumentation des fonctions natives. Notre approche suppose en effet que toutes les différentes méthodes natives permettant d'accéder aux conteneurs du système d'exploitation soient identifiées et instrumentées. Pour identifier ces différentes méthodes, nous nous sommes inspirés de l'approche utilisée par le mécanisme de contrôle d'accès de Java. En termes de suivi des flux d'informations depuis ou vers les conteneurs du système d'exploitation, l'exhaustivité de notre mécanisme correspond donc à celle du mécanisme de contrôle d'accès de Java. Si une nouvelle application qui utilise ses propres méthodes natives permettant l'accès aux ressources du système d'exploitation est déployée, il est nécessaire d'identifier et d'instrumenter ces nouvelles méthodes natives. En toute rigueur, il serait souhaitable de recourir à des méthodes statiques ou dynamiques pour identifier toutes les méthodes natives permettant d'accéder aux conteneurs de l'OS.

Chapitre 4

Résultats expérimentaux

De nombreuses expérimentations ont été réalisées afin de valider notre approche de détection d'intrusions paramétrée par la politique de sécurité utilisant le contrôle collaboratif des flux d'informations. Nous avons utilisé pour cela les deux prototypes présentés au chapitre 3 :

- Blare, un moniteur de flux d'informations implémenté au niveau du noyau Linux ;
- JBlare, une solution d'instrumentation des classes des applications Java.

Notre objectif est de démontrer que notre approche permet effectivement de détecter des intrusions résultant d'attaques «réelles» sur des applications couramment utilisées dans les systèmes que nous souhaitons étudier. Ces expérimentations permettent aussi d'évaluer le comportement de nos deux prototypes qui, bien qu'imparfaits, illustrent d'un point de vue pratique les capacités et les limites du suivi des flux d'informations, au niveau OS comme au niveau langage, et ce sur des applications «réalistes». Nous souhaitons notamment :

1. vérifier la capacité de détection de la nouvelle version de Blare, conforme au modèle présenté au chapitre 2 ainsi que celle de JBlare ;
2. démontrer l'utilité de la précision du suivi des flux d'informations apportée par JBlare ;
3. démontrer l'utilité du mécanisme de coopération entre Blare et JBlare ;
4. évaluer le surcoût engendré par ces différents mécanismes.

Afin d'atteindre ces objectifs, nous avons mis en place une maquette comprenant un système Linux caractéristique d'une installation de type «serveur», différentes applications vulnérables et nos deux prototypes de détecteurs. Cette maquette se présente sous la forme d'un fichier image VMware permettant d'exécuter le système invité sur différentes machines hôtes tout en facilitant le transport et l'échange. Les résultats présentés ici on été obtenus à l'aide d'un système hôte utilisant un processeur Intel Pentium D de 2,4 GHz, une mémoire vive de 2 Go et un disque dur de 150 Go. La taille de l'archive du système invité est de 7 Go. La mémoire allouée sur le système hôte pour le système invité a été fixée à 256 Mo. Le système invité comprend notamment les éléments suivants :

- un système Linux Debian, configuré pour une utilisation de type «serveur». Les outils classiques d'administration en mode console ont été installés.
- différents services comprenant, en plus des services installés par défaut (cron, syslog, etc.) les applications suivantes, installées depuis les *package* de la distribution : le serveur de mails Exim, le serveur web Apache, l'interpréteur PHP, le système de gestion de bases de données MySQL.
- un serveur web spécialement développé à des fins pédagogiques et de démonstration, comportant plusieurs vulnérabilités «classiques» (*buffer overflow*, mauvais traitement des URL, etc.) ;
- un serveur web écrit en Java, Jetty [1], qui fait également office de conteneur de *servlet* et de JSP Java. Nous avons utilisé la version 4.2.19 de Jetty en raison des multiples vulnérabilités que cette version comporte.
- une application web de librairie en ligne, Bookstore, téléchargée à partir du site de GotoCode [2],
- une application web de Wiki écrite en PHP, phpwiki [3]. Nous avons utilisé la version 1.3.10 de phpwiki, celle-ci possédant une vulnérabilité permettant de l'attaquer à l'aide d'une attaque de type *code injection*.
- le système de détection Blare, comprenant un noyau Linux (2.6.24) modifié ainsi que différents utilitaires et bibliothèques de fonctions correspondants ;
- le système de détection JBlare, comprenant une distribution Java (JDK) modifiée obtenue à partir de la version 6 du projet OpenJDK [4] ainsi qu'une archive JAR du *package* jBlare que nous avons développé. Les classes de ce *package* permettent d'instrumenter statiquement ou dynamiquement les différentes classes Java du système.

Nous présentons par la suite les différentes expérimentations réalisées. Le plan de ce chapitre est le suivant : la section 4.1 illustre les capacités de détection de Blare et de JBlare sur des scénarios d'attaques générant des flux d'informations illégaux de forte granularité. La section 4.2 présente l'utilité de JBlare pour la distinction des flux internes des applications. La section 4.3 illustre l'intérêt de la collaboration entre Blare et JBlare. Enfin la section 4.4 compare les différents surcoûts induits par les deux prototypes.

4.1 Détection d'instrusions

Les versions antérieures de Blare ont été validées expérimentalement [Zim03, ZMB03] et les résultats démontraient la capacité de Blare à détecter un large spectre d'attaques sur des services «classiques» (serveurs web, serveurs mail, etc.). Les résultats présentés ici sont relatifs à la nouvelle version de Blare présentée au chapitre 3, qui est conforme au modèle présenté au chapitre 2. Nous voulions nous assurer que cette nouvelle version de Blare permet de détecter les

[1] `http://www.mortbay.org/jetty/`
[2] `http://www.gotocode.com/index.asp`
[3] `http://phpwiki.sourceforge.net/`
[4] `http://openjdk.java.net/`

Nom de fichier	Propriétaire	Groupe propriétaire	Droits
`/etc/shadow`	root	shadow	-rw-r----
`/etc/passwd`	root	root	-rw-r--r-
`/var/www/*`	www-data	www-data	-rw-r--r-

TAB. 4.1 – Configuration des droits d'accès DAC de l'environnement de test

flux d'informations illégaux résultant d'attaques classiques. Nous voulions également vérifier que JBlare permet également de détecter ces flux interdits. D'autre part, les résultats illustrent la fonctionnalité d'initialisation automatique de la politique à l'aide de *script* Python.

Nous avons utilisé lors de ces expérimentations différents serveurs web :

- un serveur web «jouet» spécialement développé pour des besoins de démonstration. Ce serveur délivre des pages web «statiques» situées dans le répertoire `/var/www/`.
- le serveur Apache utilisant l'interpréteur PHP et une base de données MySQL. Ce serveur délivre des pages web dynamiques issues de l'application PHP phpwiki et situées dans le répertoire `/var/www2/`.
- le serveur web Jetty. Ce serveur délivre dans cette expérimentation les pages web statiques du répertoire `/var/www/`.

4.1.1 Initialisation automatique de la politique et attaques sur le serveur «jouet»

L'objectif de ces expérimentations était de vérifier les capacités de détection de Blare à partir d'une politique générée automatiquement. Le tableau 4.1 résume les principaux fichiers utilisés durant nos expérimentations ainsi que leurs permissions d'accès associées.

Nous avons conduit deux séries de tests s'appuyant sur deux politiques de sécurité. Dans le cas 1, nous avons utilisé une politique de sécurité générée automatiquement à partir de l'interprétation des droits d'accès. Cependant, il convient parfois de préciser certains aspects de la politique, qui doit donc être modifiée explicitement. De notre point de vue et comme le montre le cas 2 ci-dessous, de telles modifications ou précisions sont relativement simples à mettre en oeuvre et ne constituent pas un frein important à l'utilisation de notre système.

1. La politique de sécurité est ici générée automatiquement à partir de l'interprétation des droits d'accès en termes de CCAL, comme expliqué dans l'exemple de la section 2.2.4 du chapitre 2. Les groupes ont été considérés comme des utilisateurs, le super utilisateur *root* a été considéré comme un utilisateur non privilégié, c'est-à-dire sans prendre en compte les droits particuliers qu'il possède sur les fichiers dont il n'est pas propriétaire. Nous avons également considéré un utilisateur virtuel *everybody* afin de prendre en compte les droits d'accès UNIX qui s'appliquent aux utilisateurs non

propriétaires et ne faisant pas partie du groupe propriétaire (*others*). Les tags de sécurité ont été générés en parcourant l'ensemble des utilisateurs du système (66 dans notre cas, en incluant les groupes) ainsi que l'ensemble des conteneurs du système, c'est-à-dire les fichiers (environ 180 000 fichiers lors de nos expérimentations).

2. Dans un deuxième temps, nous avons précisé la politique de sécurité générée automatiquement afin de traiter les aspects distants de la politique ou de modifier l'interprétation de la politique DAC. Il existe, par exemple, des fichiers qui sont lisibles par tous, c'est-à-dire des fichiers possédant un droit de lecture pour *others*. Il est cependant communément admis et souhaité qu'un utilisateur distant ne puisse pas accéder, via le serveur web, à des données situées en dehors de l'espace web, c'est-à-dire en dehors de */var/www/html*, même si, localement, les droits UNIX permettent à tout utilisateur local de lire certaines données. Habituellement, l'application de cette politique suppose de faire confiance au serveur web ou d'utiliser des fonctionnalités *ad hoc* afin de cacher aux clients web la partie du système de fichiers située en dehors de l'espace web (par exemple en utilisant des techniques de cages *chroot* ou des mécanismes de contrôle d'accès mandataire *SELinux*). En fait, ce type de contraintes n'est pas exprimé directement par les droits d'accès UNIX : nous avons donc dû les exprimer manuellement via des CCAL adéquats. Il est, par exemple, possible de spécifier un CCAL pour la *socket* d'écoute du serveur web, de sorte que les seuls flux d'informations légaux partant ou arrivant vers la *socket* soient ceux qui arrivent ou partent de l'espace web.

Après avoir spécifié l'une ou l'autre de ces deux politiques sur le système, nous avons évalué notre IDS en termes de pertinence et de fiabilité. Nous avons réalisé dans un premier temps un certain nombre d'actions légales (comme la consultation de pages web depuis un navigateur) puis nous avons réalisé quelques scénarios d'attaques exploitant les vulnérabilités de notre serveur web. Ces scénarios sont les suivants :

- accéder à des données situées en dehors de l'espace web en utilisant une URL illégale du style */../../etc/shadow*. Cette attaque est possible car le serveur web que nous utilisons ne vérifie pas la validité de l'URL spécifiée lors de la requête HTTP. Nous avons à la fois tenté d'accéder à des données *publiques*, c'est-à-dire des données lisibles par tous (*everybody*) et à des données *privées*, par exemple des données lisibles seulement par le super utilisateur *root* ;
- obtenir un accès à un interpréteur de commandes ou à une autre application en utilisant une URL illégale vers l'espace des *script* CGI du type */cgi-bin/../../bin/sh*. Nous avons ensuite tenté de lire et de modifier certaines données publiques ou privées, à l'intérieur et à l'extérieur de l'espace des pages web ;
- obtenir un interpréteur de commandes via l'exploitation d'un *buffer overflow* sur le serveur web.

Les résultats de ces expérimentations sont présentés dans le tableau 4.2. Le

Scénario	Type d'action	Alertes (politique 1)	Alertes (politique 2)
m_1	`GET index.html`		
m_{21}	`GET /../../../etc/shadow`	x	x
m_{22}	`GET /../../../etc/passwd`		x
m_3	`GET /cgi-bin/../../../bin/sh`		
m_{31}	`cat /var/www/html/index.html`		
m_{32}	`cat /etc/shadow`	x	x
m_{33}	`cat /etc/passwd`		x
m_{34}	`cp /etc/passwd /var/www/html`		
m_4	Exploitation de *buffer overflow*		
$m_{41} - m_{44}$	Actions identiques à $m_{31} - m_{34}$ via *buffer overflow*	résultats identiques à $m_{31} - m_{34}$	résultats identiques à $m_{31} - m_{34}$

TAB. 4.2 – Alertes observées pour les politiques 1 et 2

scénario m_1 correspond à l'utilisation normale. L'utilisateur accède aux données accessibles depuis le serveur web sans générer d'attaque. Aucune alerte n'est émise, les flux résultants respectant les politiques 1 et 2. Les scénarios m_{2x} correspondent aux scénarios d'attaques utilisant le défaut de vérification sur les URL. L'attaquant peut donc accéder à des données situées en dehors de l'espace web. L'accès aux données confidentielles (le fichier `/etc/shadow` dans le scénario m_{21}) génère un flux d'informations interdit par les politiques 1 et 2. Ce type d'attaque est détecté dans les deux cas. En revanche, l'accès à des fichiers lisibles par tous au sein du système (le fichier `/etc/passwd` dans le scénario m_{22}) n'est détecté que pour la politique 2. Les scénarios m_{3x} correspondent à l'attaque utilisant le défaut de vérification sur les URL pour obtenir un interpréteur de commande ou *shell* (scénario m_3). Dans le scénario m_{31}, l'attaquant utilise un moyen détourné (une commande `cat` dans un *shell* obtenu par l'attaque m_3) pour accéder à une page de l'espace web. Cette attaque ne génère pas d'alerte car le flux d'informations induit est autorisé par les politiques 1 et 2. Les scénarios m_{32} et m_{33} permettent d'accéder à des données en dehors de l'espace web, comme dans le cas des scénarios m_{21} et m_{22}. Le scénario m_{34} correspond à un flux d'informations interne au système autorisé par les politiques 1 et 2 (aucune information n'est révélée à l'attaquant à l'issue de ce flux). L'accès ultérieur au fichier copié dans l'espace web entraînera l'émission d'une alerte si la politique 2 est utilisée car cette politique interdit l'accès au contenu initial de `/etc/passwd` via le serveur web. Les scénarios m_{4x} correspondent aux mêmes flux d'informations que ceux générés par les scénarios m_{3x} mais le *shell* est obtenu en exploitant le *buffer-overflow*. Les résultats sont identiques quel que soit le vecteur d'attaque utilisé. Le résultat dépend uniquement des flux d'informations engendrés par le scénario et de la politique de flux spécifiée.

Nous pouvons remarquer que les actions légales ne génèrent pas d'alerte. En

cas d'attaques, nous observons qu'une alarme est émise uniquement lorsqu'une violation de la politique de sécurité a lieu :

- les accès aux données privées impliquent toujours l'émission d'une alerte, quelle que soit la méthode utilisée pour lire ou modifier cette donnée ;
- les accès aux données publiques sont considérés comme légaux par la première politique, étant donné que ces données sont lisibles par *everybody* d'après DAC. Aussi, dans ce cas, aucune alerte n'est émise. En ce qui concerne la deuxième politique, tout accès aux données publiques situées en dehors de l'espace des pages web provoque l'émission d'une alerte ;
- l'obtention de privilèges *root* ou d'un interpréteur de commandes n'implique pas directement l'émission d'une alerte. Par contre, certaines des actions exécutées à partir de l'interpréteur génèrent des flux illégaux et par conséquent provoquent l'émission d'alertes.

4.1.2 Attaque sur phpwiki

Les résultats présentés précédemment montrent que la réponse de Blare dépend uniquement des flux d'informations générés par le scénario d'attaque et de la politique de flux spécifiée. Cette réponse ne dépend ni des moyens offerts pour accéder à l'information ou la modifier, ni des types et techniques d'attaques utilisés. Ce résultat est confirmé par l'expérience sur phpwiki. Nous avons utilisé dans cette expérimentation le serveur web Apache pour accéder à un site web dynamique de type Wiki. L'application utilisée, phpwiki, est vulnérable à une attaque de type *PHP script injection*. Plus exactement, l'application utilise une version vulnérable de la bibliothèque `xmlrpc` qui ne filtre pas correctement les données entrées par les utilisateurs. L'attaquant peut soumettre un fichier XML via une requête HTTP POST. Ce fichier contient une entrée `name` qui est évaluée directement par la fonction `eval()` de PHP sans être filtrée au préalable. Il est possible d'exécuter une commande arbitraire de PHP en spécifiant une valeur appropriée pour ce champ, par exemple : `',''); system('cat /etc/shadow',$xx); exit;/*`.

Nous avons utilisé une politique de sécurité similaire à la politique 2 présentée en section 4.1.1 : la politique de flux est déduite de l'interprétation des droits d'accès et complétée de manière à interdire les accès depuis le serveur web en dehors de l'espace web de l'application. Les résultats obtenus sont similaires à ceux présentés dans la section 4.1.1 :

- les scénarios permettant d'accéder aux informations situées en dehors de l'espace web (par exemple via la commande `system`) donnent lieu à l'émission d'une alerte ;
- les scénarios permettant d'accéder aux informations situées au sein de l'espace web ne génèrent pas d'alerte. En effet, les flux d'informations induits par ces scénarios sont autorisés par la politique.

Si le Wiki est en mode «fermé», c'est-à-dire si la modification des pages n'est possible que pour les utilisateurs authentifiés, Blare ne détecte pas les attaques permettant à un attaquant non-authentifié de modifier l'ensemble des pages du Wiki. En effet, les flux engendrés par cette attaque sont autorisés par la politique

de flux spécifiée. Blare ne pouvant distinguer les différents utilisateurs du serveur web, il ne peut distinguer les différents flux correspondant à chaque utilisateur. Il n'est donc pas en mesure de gérer une politique de flux qui différencierait les flux d'informations générés par les différents utilisateurs du Wiki. Afin de suivre et de distinguer précisément ces différents flux d'informations, il serait nécessaire d'implémenter les mécanismes suivants :

- un mécanisme de suivi des flux d'informations internes aux applications web ;
- un mécanisme permettant de distinguer les différents conteneurs d'une base de données.

JBlare correspond au premier type de mécanisme, pour les applications Java. L'implémentation d'un mécanisme de gestion de tags de sécurité au sein d'une application de bases de données reste à effectuer pour permettre une détection précise de ce type d'intrusions.

4.1.3 Attaque sur Jetty

Nous souhaitons également valider la capacité de détection de JBlare lorsque ce dernier collabore avec Blare. Nous avons pour cela instrumenté une application Java «réaliste». Jetty est un conteneur de *servlet* et de JSP Java. Il peut également être utilisé en tant que serveur web «statique» comme Apache. Cette application peut être utilisée seule. Elle peut également être intégrée comme module par exemple dans un serveur d'applications web comme JOnAS [5] ou JBoss [6]. Ce type d'application constitue, avec les serveurs d'applications et les serveurs de bases de données, l'architecture trois tiers communément utilisée pour les applications web. Jetty est également une application complexe, comportant de multiples *thread*, des classes internes, un *security manager* et des chargeurs de classes spécifiques.

Les différentes versions de Jetty comportent des vulnérabilités : le site SecurityFocus [7] en recense 15. Nous avons exploité au cours de cette expérimentation une faille de type *directory traversal*. Comme dans l'application web utilisée dans l'expérimentation présentée en section 4.1.1, il s'agit d'un défaut de vérification des URL indiquées par l'utilisateur. Un attaquant peut ainsi accéder à des données en dehors de l'espace web.

Nous avons également utilisé dans cette expérimentation une politique de flux d'informations inspirée de la politique 2 de la section 4.1.1 interdisant notamment les flux vers ou en provenance de l'extérieur de l'espace web. Les résultats obtenus sont identiques à ceux obtenus avec Blare pour le même type de politique, bien que le suivi des flux d'informations soit en partie réalisé par le code d'instrumentation. Ce résultat montre que le suivi collaboratif des flux d'informations permet effectivement de détecter des flux d'informations illégaux

[5] `http://jonas.objectweb.org/`

[6] `http://www.jboss.org/`

[7] `http://search.securityfocus.com/swsearch?sbm=\%2F&metaname=alldoc&query=jetty&x=0&y=0`

résultant d'attaques «réelles» sur des applications complexes «réalistes». L'expérimentation présentée dans la section suivante illustre l'intérêt de la précision apportée par le suivi des flux d'informations au niveau langage, lors de la déclassification sélective des flux d'informations.

4.2 Déclassification

Les expérimentations présentées précédemment démontrent la capacité de JBlare et de Blare à détecter des flux d'informations observables au niveau OS. Nous souhaitons également démontrer l'utilité du suivi des flux d'informations internes apporté par JBlare. Nous nous sommes intéressés à une vulnérabilité de Jetty permettant d'accéder au code source des pages JSP [8]. Ces pages web dynamiques contiennent en effet en partie le code source de l'application web. Elles doivent êtres compilées en *servlet* Java, elles-mêmes exécutées par le conteneur de *servlet*. La politique généralement mise en place au sein d'un conteneur de *servlet* comme Jetty consiste à interdire la lecture directe des *servlet* et des JSP mais à autoriser leur interprétation. En effet, le code source des JSP ou des *servlet* peut contenir diverses informations sensibles ou confidentielles, notamment les mots de passe permettant l'accès à la base de données utilisée par l'application web. L'exploitation de la vulnérabilité consiste à générer des caractères spéciaux (en l'occurence des caractères exprimés par leur code ASCII comme `\%5C`). Le but est de générer une URL valide du point de vue du système de fichiers mais dont la syntaxe vise à leurrer le système de filtres utilisé par le conteneur de *servlet* pour identifier les ressources à interpréter. La vulnérabilité choisie n'étant exploitable que sur les systèmes Windows, nous avons modifié le code source de Jetty pour étendre ce type de vulnérabilité au système Linux. Cette modification constitue une injection de vulnérabilités mais celle-ci ne modifie en rien le principe de la vulnérabilité déjà présente.

La détection de ce type de vulnérabilités suppose de distinguer les flux résultant de l'interprétation de la JSP de ceux résultant de sa lecture directe. Toutefois en termes de flux d'informations tels que nous les définissons dans notre modèle, ces différents cas ne sont pas discernables. Le contenu envoyé à l'utilisateur via la *socket* du serveur provient dans les deux cas du même conteneur d'informations. Pour différencier ces différents cas de figure, il est donc nécessaire de recourir à des règles d'exception ou de déclassification sélective. Nous entendons ici par déclassification une exception à la politique de flux d'information qui se traduit par la non propagation de tags de sécurité. La politique qui s'applique sur le contenu déclassifié est donc moins restrictive que celle qui devrait s'appliquer en l'absence de déclassification. La politique de flux d'informations adoptée interdit l'accès direct aux JSP, depuis la *socket* du serveur web. Afin d'autoriser l'interprétation de ces pages dynamiques, il faut en effet autoriser des exceptions à la politique de flux. La définition de ces exceptions au niveau du système d'exploitation, par exemple pour l'application entière, n'est pas pertinente car ce type de règle ne permet pas de détecter les intrusions

[8] `http://secunia.com/advisories/17659/`

résultant de flux internes à l'application. Dans notre cas, la déclassification au niveau de l'application Jetty ne permettrait pas de distinguer une interprétation de la JSP d'une attaque révélant son code source. L'intérêt du contrôle des flux internes des applications est notamment de pouvoir appliquer sélectivement les règles de déclassification, en fonction des flux d'informations internes à l'application. Dans notre cas, il s'agit d'empêcher la propagation des tags de sécurité au sein de la fonction de compilation des JSP. Nous avons introduit cette exception dans le code d'instrumentation de JBlare mais il est possible d'envisager l'utilisation de fichiers définissant des règles d'exception à la propagation des tags de sécurité pour un certain nombre de méthodes «de confiance».

La politique de flux retenue utilise deux CCAL :

- le premier CCAL est associé à la *socket* d'écoute de Jetty ainsi qu'aux contenus statiques (pages html).
- le deuxième CCAL est associé à l'ensemble des JSP.

Cette politique autorise les flux d'informations des contenus statiques vers la *socket* mais interdit l'accès direct aux pages JSP. Néanmoins, lors de l'interprétation des JSP, l'accès aux contenus dynamiques ne génère pas d'alerte en raison de la déclassification. Nous avons pu vérifier en revanche que l'accès direct aux JSP, en exploitant la vulnérabilité de type *source disclosure*, entraînait l'émission d'une alerte, ce type de flux d'informations internes n'étant pas déclassifié. Cette expérience illustre donc la nécessité de recourir au suivi des flux d'informations internes aux applications pour élargir le spectre des intrusions détectables. L'expérimentation décrite dans la section suivante constitue un deuxième exemple de déclassification sélective. Elle démontre également l'intérêt du principe de suivi collaboratif des flux d'informations.

4.3 Détection collaborative

Nous avons vu dans les précédentes expérimentations l'intérêt de JBlare dans le suivi des flux d'informations et la détection des intrusions. Toutefois, le surcoût de cette solution, évoqué en section 4.4 ne permet pas d'envisager de l'appliquer à toutes les applications du système. En outre, JBlare ne dispose que d'une vue «locale» des flux d'informations : il détecte les flux internes de l'application surveillée ainsi que les flux entre l'application surveillée et les conteneurs du système d'information. Blare dispose en revanche d'une vue «globale» lui permettant de suivre tous les flux entre les différentes applications. Afin d'illustrer l'intérêt d'une solution de suivi collaboratif des flux d'informations, permettant de combiner une vue «globale» et une vue «locale» des flux d'informations, nous avons implémenté le système suivant :

- Le système possède deux interfaces, implémentées sur notre maquette sous la forme de deux *socket* d'écoute. L'une est connectée à un réseau de confiance et l'autre à un réseau public.
- Les utilisateurs du réseau de confiance peuvent accéder librement aux données accessibles depuis l'interface du réseau de confiance. Nous avons implémenté ce service d'accès à l'aide d'un serveur web Apache et d'une

petite application PHP permettant de déposer des fichiers dans l'espace web. L'utilisateur du réseau de confiance peut ainsi générer des flux d'informations vers ou depuis les conteneurs d'informations du domaine web privé (`/var/www/`)

– les utilisateurs du réseau public ont accès aux données du domaine web public (`/var/www2`). En revanche, l'accès direct aux données du domaine web privé (`/var/www`) leur est interdit. Cependant, certains utilisateurs privilégiés étant susceptibles d'accéder au système via le réseau public, les flux d'informations depuis le domaine web privé vers l'interface publique sont autorisés à condition que le flux d'informations soit chiffré. Ceci garantit que seuls les utilisateurs privilégiés pourront effectivement accéder à l'information issue du réseau de confiance. Nous avons implémenté ce service d'accès à l'aide d'une *servlet* s'exécutant sous Jetty et utilisant la bibliothèque de chiffrement Jasypt [9]. Cette *servlet* chiffre le contenu des pages du domaine privé accédé depuis l'interface publique.

Afin de détecter les intrusions lorsqu'un attaquant accède, depuis l'interface publique, aux contenus provenant du domaine de confiance, il est nécessaire de mettre en place une politique de flux. En effet, un attaquant peut, en exploitant par exemple la faille de type *directory traversal* de Jetty, contourner le filtrage mis en place pas la *servlet* et accéder directement au contenu du répertoire `/var/www/`. Afin de séparer le domaine public du domaine privé, nous avons défini deux CCAL correspondant à chacun de ces domaines. Les tags de sécurité de chacune des *socket* d'écoute contiennent uniquement le CCAL associé à leur domaine. Les tags de sécurité des fichiers du répertoire du domaine public `/var/www2/` contiennent les deux CCAL, leur contenu étant public et accessible dans les deux domaines. Une telle politique de flux interdit tout flux d'informations entre le domaine de confiance et le domaine public. Afin d'autoriser les flux d'informations chiffrés par la *servlet*, nous avons défini comme dans l'expérimentation de la section 4.2 des règles de déclassification pour la méthode de chiffrement. Ainsi, les flux d'informations des conteneurs du domaine privé vers l'interface publique sont interdits par la politique mais ceux transitant via la méthode de chiffrement ne génèrent pas d'alerte en raison de la règle d'exception qui ne propage pas les tags de sécurité au sein de la méthode de chiffrement. En revanche, l'accès direct aux conteneurs d'informations du domaine privé depuis l'interface publique, en exploitant la faille de type *directory traversal* de Jetty, donne bien lieu à l'émission d'alertes.

La coopération entre les deux mécanismes, Blare et JBlare, permet de discerner les différents flux d'informations entre les différentes applications. En particulier, Blare propage les tags de sécurité de la *socket* d'écoute du serveur Apache sur chacun des conteneurs d'informations du domaine privé. JBlare propage ensuite ces tags à l'intérieur du code instrumenté de Jetty. La précision de JBlare et l'utilisation d'un mécanisme de déclassification permet ensuite de distinguer les différents flux d'informations. Le principe de détection collaboratif permet donc de profiter à la fois de la vue globale de Blare et de la vue locale,

[9] `http://www.jasypt.org/`

plus précise, de JBlare. Cette précision n'est toutefois pas sans conséquence et la section suivante présente et compare le surcoût engendré par les différents prototypes.

4.4 Evaluation du surcoût engendré

Les expérimentations présentées précédemment mettent en évidence les avantages apportés par JBlare dans la précision du suivi des flux d'informations. Se pose alors la question du surcoût engendré par cette solution. Les objectifs de nos expérimentations sont aussi d'évaluer les différents surcoûts liés au processus de suivi des flux d'informations et de détection des intrusions. Nous avons donc évalué les surcoûts des deux prototypes. L'analyse des résultats obtenus nous permet de définir les limites de la solution et les cas d'utilisation appropriés. Nous présentons enfin quelques pistes d'améliorations possibles.

4.4.1 Impacts de Blare

Blare est un prototype implémenté au sein du noyau du système d'exploitation. En apportant de nouvelles fonctionnalités et en ajoutant de nouvelles structures de données au noyau Linux, il impacte donc sur les performances de ce dernier. Le noyau offrant des services aux applications (appels système, ordonnancement des processus, etc.), Blare impacte donc également les performances des applications. Nous pouvons déterminer deux types de surcoût :

- le stockage des tags de sécurité des différents conteneurs du système, réalisé entièrement en mémoire vive afin de limiter le surcoût à l'exécution lors de l'accès aux tags de sécurité, génère une surconsommation de l'espace mémoire ;
- le suivi des flux d'informations, implémenté sous la forme d'appels de fonctions intercallés dans le processus de traitement des appels système, ralentit le système en allongeant les temps de réponse des appels système.

Pour évaluer ces différents effets, nous avons effectué les mesures suivantes :

- nous avons mesuré l'occupation de la mémoire sur un système lancé avec une version non modifiée du noyau Linux, les processus chargés étant ceux lancés automatiquement par le système. Nous avons ensuite mesuré l'occupation mémoire sur un système lancé avec une version modifiée par le *patch* Blare, suite à l'initialisation automatique de la politique de flux. Nous avons réitéré ces mesures après arrêt et redémarrage de la machine virtuelle VMware et nous présentons la moyenne des résultats obtenus. Cette mesure n'est qu'indicative du surcoût d'occupation mémoire car celui-ci dépend du nombre de tags de sécurité présents sur le système. Nous avons évalué le cas où tous les conteneurs de type fichiers possèdent un tag de sécurité, ce qui correspond au cas d'initialisation automatique de la politique. Le nombre de tags réellement présents peut être inférieur si l'administrateur définit manuellement les tags de sécurité sur un sous-ensemble des fichiers du système. Il peut être supérieur lorsque différents

processus sont exécutés et que les tags sont propagés, notamment vers leur espace mémoire.
- nous avons mesuré le surcoût engendré par le système de suivi des flux d'informations lors de la compilation d'un noyau Linux. Ce type d'opération génère en effet de nombreuses opérations d'entrée/sortie et permet donc d'évaluer le surcoût lié notamment à la propagation des tags de sécurité. Comme dans le cas précédent, nous avons mesuré le temps nécessaire à la compilation sur un système utilisant un noyau Linux non modifié puis sur un autre système utilisant Blare et dont la politique de flux a été initialisée automatiquement.

Les résultats montrent qu'en moyenne Blare génère un surcoût d'environ 5% sur le temps d'exécution et un surcoût d'occupation mémoire de 3 Mo. Il s'agit donc d'une application peu intrusive sur le comportement du système.

4.4.2 Impacts de JBlare

JBlare, en instrumentant les différentes classes des applications Java, impacte également sur les performances des applications Java. Nous distinguons deux types de surcoût :
- le surcoût mémoire engendré à la fois par les tags de sécurité et par l'augmentation de la taille des classes instrumentées ;
- le surcoût sur le temps d'exécution.

Le surcoût sur le temps d'exécution dépend de plusieurs facteurs :
- L'instrumentation dynamique de classes allonge le temps de chargement des classes. Le temps de chargement et d'initialisation des applications est ainsi allongé.
- L'exécution du code instrumenté est ralenti en raison des instructions ajoutées pour le suivi des flux d'informations.
- La coopération avec Blare nécessite d'obtenir les tags de sécurité des conteneurs gérés par Blare et de les propager au niveau des tags de sécurité gérés par JBlare. Cette opération est coûteuse en raison du changement de contexte qu'elle nécessite (appel système, recopie de la mémoire du noyau vers la mémoire utilisateur, etc.).

Nous avons évalué le surcoût engendré au chargement de l'application en modifiant Jetty pour qu'il effectue une mesure de temps au début et à la fin du chargement de l'application. Nous avons comparé les temps de chargement entre une version non instrumentée de Jetty et une version instrumentée par JBlare. Nous avons constaté qu'en moyenne le temps de chargement est multiplié par 14.

Nous avons également évalué le surcoût lié à l'instrumentation et à la propagation des tags de sécurité. Pour cela, depuis un client web, nous avons mesuré le temps de réponse suite à l'envoi d'une requête HTTP GET. Nous avons effectué des mesures sur une version non instrumentée de Jetty puis sur une version instrumentée avec JBlare. Nous avons éliminé le temps de réponse de la première requête puisque celle-ci provoque le chargement d'un certain nombre de classes. Nous avons constaté qu'en moyenne le temps de réponse est multiplié par 3.

Enfin, à l'aide du test SciMark [10], nous avons tenté d'évaluer le surcoût global lié à l'exécution. Nous avons exécuté ce test sans instrumentation puis avec instrumentation. Nous avons constaté une baisse du score global de 40% suite à l'utilisation de JBlare.

Ces différents résultats montrent que l'impact de l'instrumentation dynamique de classes et du suivi des flux d'informations est conséquent sur le temps d'exécution des applications. Pour les applications de type serveur, le temps de réponse est prépondérant devant le temps de chargement. En effet, une fois lancée, l'application s'exécute pendant un temps relativement long puisqu'elle doit être à l'écoute des requêtes des utilisateurs. Le temps de réponse est important car c'est celui qui est perçu par l'utilisateur final. Cependant, ce temps reste également conséquent dans le cas de l'utilisation de JBlare. Nous notons néanmoins que ces résultats sont du même ordre de grandeur que ceux présentés dans les travaux qui utilisent des techniques similaires aux nôtres [CF07, YYW+07]. Ils représentent donc en partie le surcoût intrinsèque lié à ce type d'instrumentation.

La coopération entre Blare et JBlare montre ici tout son intérêt dans la limitation du surcoût global du processus de détection. En effet, la plupart des applications qui ne nécessitent pas de suivi des flux d'informations internes peuvent être surveillées par Blare tandis que seules les applications nécessitant un suivi plus précis sont surveillées par JBlare.

Plusieurs pistes d'améliorations peuvent être envisagées :

- le recours massif à l'instrumentation statique pour toutes les classes qui peuvent être identifiées lors du déploiement de l'application ;
- l'utilisation d'un mécanisme de cache lors de la propagation des tags de sécurité depuis Blare ;
- l'utilisation de méthodes statiques permettant d'éliminer certains points d'instrumentation lorsqu'aucun flux ne peut être réalisé ;
- l'utilisation d'un support partiel au niveau de la JVM, ce support nécessitant cependant des modifications profondes du code de la machine virtuelle.

4.5 Bilan

L'objectif de ces expérimentations était de démontrer les capacités de détection de Blare et de JBlare. Les différents cas de figure présentés ici montrent que la détection d'intrusions paramétrée par la politique de sécurité utilisant le contrôle collaboratif de flux d'informations permet effectivement de détecter des intrusions sur des systèmes «réalistes». En particulier, nous avons pu vérifier que la nouvelle version de Blare était conforme au modèle proposé. Ce prototype permet de détecter des intrusions vérifiant les hypothèses suivantes :

- l'intrusion est caractérisée par un flux d'informations observable et discernable des autres flux par Blare ;

[10] `http://math.nist.gov/scimark2/`

– ce flux d'informations est interdit par la politique de flux d'informations spécifiée à l'aide de tags de sécurité.

Le processus de détection repose donc sur une condition logique issue de la spécification de la politique de sécurité, ce que nous souhaitions. De plus, ce processus ne dépend pas du scénario ou du vecteur d'attaque utilisé. Il suppose néanmoins que la politique soit correctement spécifiée. Cette tâche peut s'avérer fastidieuse. Nous proposons un mécanisme permettant de générer automatiquement une première version de la politique qui peut ensuite être modifiée avec un minimum d'efforts. Les résultats montrent que cette politique permet de détecter correctement un certain nombre d'intrusions.

Les résultats illustrent également la limite de Blare dans la précision du suivi des flux d'informations. Ils montrent que JBlare peut compléter Blare pour la surveillance de certaines applications nécessitant de discerner les différents flux d'informations internes aux applications, notamment dans le cadre de la déclassification sélective des flux d'informations. Notre approche de détection par instrumentation de *bytecode* a ainsi été validée sur une application «réaliste», JBlare. Toutefois, les résultats présentés font apparaître le surcoût important apporté par JBlare.

Enfin, ces résultats illustrent l'intérêt de la collaboration entre les différentes solutions de détection d'intrusions fonctionnant à niveaux différents. Elle permet de suivre des flux d'informations complexes entre diverses applications. Elle permet également de limiter l'utilisation de JBlare aux applications nécessitant un suivi précis de leur flux d'informations internes.

Conclusion

Nous avons présenté dans cette thèse une approche de détection d'intrusions s'appuyant sur le suivi collaboratif des flux d'informations. Cette approche a pour objectif la détection des violations d'une politique de flux d'informations définie au préalable au sein des systèmes utilisant des applications web.

Bilan

Le contexte de l'étude, notamment en termes de type de système surveillé, imposait certaines contraintes :

- la complexité du système et la diversité des applications utilisées nécessitaient de suivre tous les flux d'informations, à plusieurs niveaux de granularité ;
- l'utilisation de composants COTS au sein du système étudié nécessitait de réutiliser au maximum les composants logiciels existants disponibles dans le commerce. Cette contrainte excluait toute réécriture complète des logiciels du système. La solution proposée devait également être compatible avec les standards existants.
- la solution retenue devait être la moins intrusive possible. En particulier elle ne devait pas interdire l'accès au système à des utilisateurs légitimes.

Dans ce contexte, l'utilisation d'une approche de détection permet de compléter les mécanismes préventifs existants, par exemple les mécanismes de contrôle d'accès. Il s'agit d'une approche *a posteriori* qui présente des alertes en cas d'intrusions. Ces alertes sont interprétées par un administrateur de sécurité qui peut ensuite prendre les mesures adéquates. En cas de fausse alarme, l'accès des utilisateurs légitimes n'est pas perturbé. Cependant, une approche de détection doit, pour être applicable en pratique, limiter le taux de faux négatifs et de faux positifs. Nous souhaitions donc définir une architecture permettant d'implémenter une solution la plus complète et la plus pertinente possible.

Nous avons écarté les solutions reposant sur une connaissance *a priori* des attaques, par exemple, les approches utilisant des bases de signatures d'attaques. Ces approches sont en effet incomplètes par nature. Les intrusions reposant sur des attaques inconnues lors de la spécification des signatures d'attaques ne peuvent être détectées. De plus, les IDS qui reposent sur de telles approches sont en pratique souvent leurrés par des attaquants expérimentés qui modifient les

scénarios d'attaques connues afin d'empêcher leur détection. Ces IDS nécessitent en outre une mise à jour régulière de la base de signatures.

Afin de détecter les attaques connues lors du déploiement de l'IDS et celles découvertes par la suite, nous nous basons sur une approche comportementale. Ce type d'approche s'appuie en effet seulement sur le comportement attendu et légal du système surveillé. Les approches classiques utilisent des modèles statistiques ou nécessitent une phase d'apprentissage pour modéliser le comportement du système attendu. En pratique, de tels systèmes s'avèrent relativement peu pertinents et leur taux de faux positifs important constitue une des limites majeures à leur déploiement. De plus, le paramétrage de ces IDS repose sur des données empiriques (seuils de détection, durée d'apprentissage, etc.). La modification des paramètres, afin par exemple de modifier le comportement de référence pour éliminer certains faux positifs, est complexe en raison de l'empirisme de l'approche. Nous pensons également que cet empirisme est une des causes du manque de pertinence des IDS actuels.

Nous avons donc choisi de nous appuyer sur une approche de détection comportementale paramétrée par la politique de sécurité. Cette approche repose sur une définition du comportement attendu qui est directement déduite de la politique de sécurité. Les anomalies détectées sont donc des violations de la politique de flux spécifiée, ce qui correspond réellement à la définition des intrusions. Une telle approche repose sur un mécanisme déterministe qui s'appuie sur un modèle formel de détection. Ce modèle comprend trois éléments :

- une modélisation du système surveillé en termes de conteneurs d'informations et de contenus. L'évolution de l'état du système suite à l'exécution de commandes est exprimée à travers l'évolution de la relation entre les contenus et les conteneurs.
- une modélisation de la politique de sécurité. Nous nous sommes restreints aux politiques de flux d'informations permettant d'assurer la confidentialité et l'intégrité des données. Cette politique est exprimée sous la forme d'un ensemble de relations autorisées entre les conteneurs du système et les différents contenus possibles.
- un théorème de détection exprimant la condition logique qui permet de déterminer la légalité des flux d'informations.

Ce modèle est suffisamment générique pour s'appliquer à différents systèmes manipulant des données comprises dans des conteneurs d'informations. En particulier, notre modèle n'impose aucune granularité sur les conteneurs d'informations du système et les flux d'informations réalisés.

Nous avons proposé une architecture générique d'IDS permettant d'implémenter des solutions conformes à notre modèle de détection. Cette architecture permet de prendre en compte les différents niveaux de granularité des conteneurs d'informations du système et des flux d'informations. Nous nous sommes appuyés sur plusieurs mécanismes de suivi des flux d'informations, chaque mécanisme assurant le suivi de flux d'informations à des niveaux de granularité différents. Une contribution majeure de cette thèse réside dans la définition d'une approche collaborative du suivi des flux d'informations. Les différents mécanismes de détection de l'architecture collaborent afin de prendre en compte

les flux d'informations entre des conteneurs d'informations de granularité différentes (par exemple, entre un fichier du système d'exploitation et une variable d'une application).

Nous avons proposé une implémentation de cette architecture générique s'appuyant sur deux prototypes :

- nous réutilisons en l'adaptant un détecteur d'intrusions OS, Blare, qui permet le suivi au niveau des conteneurs de forte granularité, gérés par le système d'exploitation.
- nous proposons une implémentation d'un nouveau détecteur, JBlare, qui permet le suivi des flux d'informations entre conteneurs de faible granularité, à l'intérieur des applications Java.
- nous proposons également un mécanisme de coopération entre ces deux prototypes qui peuvent ainsi être utilisés conjointement.

Notre implémentation permet de suivre les flux d'informations entre les applications ainsi qu'au sein même de certaines applications. Les applications considérées sont des applications réalistes comme le serveur web Apache ou le conteneur de *servlet* Jetty. Notre approche limite le nombre de composants qu'il est nécessaire de recompiler : seules quelques modifications, fournies sous la forme de *patch*, doivent être apportées au noyau du système d'exploitation et à la JVM. Notre implémentation est compatible avec les composants existants, COTS ou applications spécialement développées pour des besoins spécifiques. Ainsi Blare est compatible avec toutes les applications Linux et ne nécessite pas de modifier ces applications. De même JBlare est compatible avec toutes les applications Java et il ne nécessite pas de modifier explicitement le code source de ces applications, l'instrumentation étant réalisée automatiquement au niveau du *bytecode* des fichiers de classes.

Nous avons enfin réalisé un certain nombre d'expérimentations afin de valider notre approche et notre architecture de détection collaborative. Ces expérimentations ont été conduites sur un système représentatif du type de système que nous souhaitons surveiller, comprenant un OS, diverses applications COTS, ainsi que des applications web. Nous avons soumis les différents éléments de ce système à des attaques «classiques» générant des intrusions en exploitant des vulnérabilités présentes sur les différents composants du système. Les résultats de ces expérimentations amènent les commentaires suivants :

- la détection d'intrusions paramétrée par la politique de sécurité à l'aide du contrôle des flux d'informations permet effectivement de détecter des intrusions résultant de différents scénarios d'attaques. Le processus de détection est déterministe et ne dépend que des éléments suivants :
  - la capacité du détecteur, Blare ou JBlare, à observer et à discerner les flux d'informations nécessaires à la réalisation de l'intrusion ;
  - la non-caractérisation par la politique de ces flux d'informations en tant que flux d'informations autorisés.
- les différents scénarios d'attaques générant des flux d'informations identiques produisent des résultats identiques en termes de détection.
- certaines intrusions nécessitent de discerner les différents flux internes des applications et ne peuvent être détectées à l'aide de Blare. L'utilisation de

JBlare et de mécanismes de déclassification sélective permet en revanche de les détecter.
- les intrusions mettant en jeu des flux d'informations complexes, entre applications et à l'intérieur de certaines applications, sont détectées grâce à la collaboration entre Blare et JBlare. Celle-ci permet la propagation des tags de sécurité gérés par Blare vers le code instrumenté par JBlare (par exemple, des tags de sécurité des fichiers vers ceux des variables Java).
- la précision du suivi des flux d'informations réalisé par JBlare génère cependant un surcoût conséquent. Cette solution doit donc être restreinte à la surveillance d'applications complexes nécessitant un suivi précis de leurs flux d'informations internes. La collaboration entre Blare et JBlare permet de limiter le surcoût global du processus de suivi des flux d'informations.

Perspectives

Les résultat obtenus nous paraissent prometteurs et nous laissent envisager un certain nombre de perspectives. Nous avons regroupé ces dernières suivant trois domaines :
- les perspectives concernant la phase «amont» du processus de détection, à savoir la définition de la politique de flux d'informations ;
- les perspectives concernant le suivi des flux d'informations ;
- les perspectives concernant la phase «aval» du processus de détection, à savoir la gestion des alertes ;

Initialisation de la politique de flux d'informations

Notre modèle et notre architecture de détection des intrusions s'appuient sur une définition de la politique de sécurité. Plus précisément, nous nous sommes intéressés au cours de cette thèse aux politiques de flux d'informations permettant d'assurer la confidentialité et l'intégrité des données. Nous n'avons fait aucune hypothèse sur le type de politique que notre approche permet de traiter. Nous avons simplement supposé que la politique est exprimable à l'aide d'un ensemble de CCAL.

Plusieurs perspectives peuvent être envisagées concernant cette phase «amont» du processus de détection. D'un point de vue théorique, il serait intéressant de définir la classe des politiques de sécurité qui peuvent être prises en compte dans notre modèle et de la confronter aux différents modèles de politique de sécurité couramment utilisés. Une seconde piste consiste à définir les modifications à apporter à notre modèle afin de prendre en compte des politiques non traitées actuellement. En particulier, les aspects suivants nous paraissent importants :
- la collaboration de plusieurs entités qui souhaitent partager une partie de leurs informations tout en garantissant une séparation entre les données privées de chacune des entités : il pourrait être intéressant de s'inspirer du modèle de gestion décentralisée des labels de sécurité (*Decentralized Label*

Model) proposé par Liskov et Myers [ML97].
- la déclassification : nous la traitons ici «en dure» au niveau du mécanisme de suivi ; elle pourrait faire l'objet d'une politique définissant des critères autorisant la déclassification.
- la modification de la politique en général : par exemple suite à l'ajout d'un utilisateur ou d'un conteneur d'informations.

D'un point de vue pratique, notre IDS s'appuie sur une définition de «bas niveau» de la politique de flux définie entre les différents conteneurs du système d'exploitation. Définir une telle politique suppose de spécifier les tags de sécurité pour chacun des conteneurs d'informations. Nous proposons ici une méthode d'initialisation automatique de la politique de flux d'informations à partir d'une interprétation des droits d'accès. Une perspective intéressante consisterait à définir d'autres moyens de génération de la politique, utilisant éventuellement des interprétation différentes. Il serait par exemple intéressant d'utiliser des outils de spécification et de déploiement de politiques de «haut niveau» par exemple les politiques à base de rôles (RBAC, ORBAC).

Suivi des flux d'informations

Cette thèse propose une amélioration du processus de suivi des flux d'informations en s'appuyant sur l'approche de détection collaborative des flux d'informations suivant deux niveaux de granularité. Plusieurs améliorations de ce processus de détection peuvent être envisagées :
- d'autres niveaux de granularité ou type de conteneurs peuvent être pris en compte à l'aide de nouvelles implémentations collaborant avec Blare et JBlare. En particulier, il nous paraît essentiel de suivre les flux d'informations entre les différents champs des tables d'une base de données. Ce type de suivi permettrait par exemple de détecter les attaques de type *sql-injection*.
- le suivi des flux d'informations entre plusieurs machines communiquant sur un réseau constitue également une piste intéressante. En effet, les différents composants d'une architecture d'application web sont parfois distribués sur différentes machines hôtes ou sur différents systèmes invités s'exécutant sur un même système hôte à l'aide d'une solution de virtualisation. Les différents services s'échangent alors des données à travers le réseau. Afin de suivre tous les flux d'informations lié sau fonctionnement de l'application web, il convient donc d'associer des tags ou des labels de sécurité aux paquets réseau, en s'inspirant, par exemple, des solutions de type *labeled IPSEC* ou *NetLabel/CIPSO*.
- les approches et implémentations actuellement utilisées peuvent elles-mêmes faire l'objet d'améliorations :
  - certains conteneurs d'informations sont gérés partiellement par JBlare (notamment les tableaux), d'autres ne le sont pas (par exemple, les exceptions). L'implémentation doit donc être poursuivie afin de permettre un suivi complet de tous les flux d'informations.
  - le surcoût lié à JBlare est important. L'optimisation du suivi des flux

d'informations internes aux applications constitue donc un axe d'amélioration prioritaire bien que ce surcoût soit en partie inhérent à l'approche. D'autres techniques permettant le suivi des flux internes peuvent être utilisées. Il est par exemple possible d'envisager d'instrumenter plus en profondeur la JVM et de profiter d'un support partiel ou total d'un mécanisme interne à la JVM pour le suivi des flux d'informations dans les applications Java. Enfin, l'obtention depuis JBlare des tags manipulés par Blare génère un surcoût conséquent. Il convient donc d'optimiser ce mécanisme, par exemple en utilisant un cache de tags de sécurité au niveau du code Java instrumenté.

– Enfin, mêler analyse statique et dynamique serait intéressant. Certaines classes pourraient ainsi être instrumentées «hors-ligne» en fonction d'une analyse statique déterminant précisément les tags qu'il est nécessaire de propager lors du suivi dynamique. L'utilisation combinée de ces deux formes d'analyses permettrait en théorie d'améliorer la précision du suivi en prenant en compte les flux d'informations indirects. Elle devrait également optimiser le processus de suivi dynamique et réduire ainsi le surcoût engendré.

Gestion des alertes

Nous employons dans notre approche ainsi que dans les prototypes implémentés un mécanisme simple de gestion des alertes qui se contente d'émettre une alerte dès qu'un conteneur d'informations contient une information interdite par la politique, ce qui se traduit par une intersection vide entre son tag de sécurité en écriture et son tag de sécurité en lecture. Blare et JBlare réagissent donc à tous les flux d'informations détectés comme illégaux. Il se peut cependant qu'une même attaque génère plusieurs flux d'informations illégaux ou que les différents flux d'informations élémentaires constituant un flux d'informations composé soient tous interdits par la politique. Dans ce cas de figure, particulièrement fréquent pour le suivi des flux internes aux applications, plusieurs alertes sont émises pour une même attaque. Il peut donc s'avérer nécessaire de regrouper ces différentes alertes, par exemple à l'aide d'un mécanisme d'agrégation ou de corrélation des alertes.

Un deuxième type de perspectives concerne les actions à mener en cas d'alertes. Nous envisageons notamment le diagnostic des alertes. En effet, en cas d'alertes, l'administrateur ne dispose à l'heure actuelle que des identifiants des derniers conteneurs accédés qui ont généré le flux d'informations illégal. Il pourrait être intéressant de retrouver les contenus initiaux qui ont été utilisés et éventuellement de retrouver le scénario d'attaque.

Bibliographie

[ACF+00] Julia Allen, Alan Christie, William Fithen, John McHugh, Jed Pickel, and Ed Stoner. State of the practice of intrusion detection technologies. Technical Report CMU/SEI-99-TR-028, Software Engineering Institute, Carnegie Mellon University, January 2000.

[And80] James P. Anderson. Computer Security Threat Monitoring and Surveillance. Technical report, James P. Anderson Company, Fort Washington, Pennsylvania, April 1980.

[Axe99] Stefan Axelsson. The base-rate fallacy and its implications for the difficulty of intrusion detection. In *Proceedings of the 6th ACM Conference on Computer and Communications Security*, pages 1–7, November 1999.

[BBM94] J. Banatre, C. Bryce, and D. Le Métayer. Compile-time detection of information flow in sequential programs. In *3rd European Symposium on Research in Computer Security*, volume 875 of *Notes in Computer Science*, pages 55–73, November 1994.

[BC01] Gérard Boudol and Ilaria Castellani. Noninterference for concurrent programs. In *ICALP '01 : Proceedings of the 28th International Colloquium on Automata, Languages and Programming,*, pages 382–395, London, UK, 2001. Springer-Verlag.

[BD03] Yolanta Beres and Chris I. Dalton. Dynamic label binding at run-time. In *NSPW '03 : Proceedings of the 2003 workshop on New security paradigms*, pages 39–46, New York, NY, USA, 2003. ACM Press.

[BHM07] Walter Binder, Jarle Hulaas, and Philippe Moret. Advanced java bytecode instrumentation. In *PPPJ '07 : Proceedings of the 5th international symposium on Principles and practice of programming in Java*, pages 135–144, New York, NY, USA, 2007. ACM.

[BLW02] Lujo Bauer, Jarred Ligatti, and David Walker. More enforceable security policies. In Iliano Cervesato, editor, *Foundations of Computer Security : proceedings of the FLoC'02 workshop on Foundations of Computer Security*, pages 95–104, Copenhagen, Denmark, 25–26 July 2002. DIKU Technical Report.

[BN03] Anindya Banerjee and David A. Naumann. Using access control for secure information flow in a java-like language. In *IEEE Computer Security Foundations Workshop (CSFW)*, pages pages 155–169, 2003.

[CC04] Jedidiah R. Crandall and Frederic T. Chong. Minos : Control data attack prevention orthogonal to memory model. In *Proceeding of the 37th International Symposium on Microarchitecture*, pages 221–232, December 2004.

[CF07] Deepak Chandra and Michael Franz. Fine-grained information flow analysis and enforcement in a java virtual machine. *acsac*, 0 :463–475, 2007.

[CLM+07] Stephen Chong, Jed Liu, Andrew C. Myers, Xin Qi, K. Vikram, Lantian Zheng, and Xin Zheng. Secure web application via automatic partitioning. *SIGOPS Oper. Syst. Rev.*, 41(6) :31–44, 2007.

[CLO07] James Clause, Wanchun Li, and Alessandro Orso. 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, 2007. ACM Press.

[CPG+04] Jim Chow, Ben Pfaff, Tal Garfinkel, Kevin Christopher, and Mendel Rosenblum. Understanding data lifetime via whole system simulation. In *SSYM'04 : Proceedings of the 13th conference on USENIX Security Symposium*, pages 22–22, Berkeley, CA, USA, 2004. USENIX Association.

[CVM07] Stephen Chong, K. Vikram, and Andrew C. Myers. Sif : Enforcing confidentiality and integrity in web applications. In *Proceedings of the 16th USENIX Security Symposium (Security 07)*, 2007.

[DD77] Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7) :504–513, July 1977.

[DDW00] Hervé Debar, Marc Dacier, and Andreas Wespi. A Revised Taxonomy for Intrusion-Detection Systems. *Annales des Télécommunications*, 55(7-8), 2000.

[Den76] Dorothy E. Denning. A lattice model of secure information flow. *Commun. ACM*, 19(5) :236–243, 1976.

[Den87] Dorothy E. Denning. An Intrusion-Detection Model. *IEEE transaction on Software Engineering*, 13(2) :222–232, 1987.

[DKK07] Michael Dalton, Hari Kannan, and Christos Kozyrakis. 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, 2007. ACM Press.

[DM02] Yves Deswarte and Ludovic Mé. *Sécurité des réseaux et systèmes répartis.* Traité IC2, série Réseaux et Télécoms. Hermes, 2002.

[ECGN01] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2) :99–123, February 2001.

[EKV+05] Petros Efstathopoulos, Maxwell Krohn, Steve VanDeBogart, Cliff Frey, David Ziegler, Eddie Kohler, David Mazières, Frans Kaashoek, and Robert Morris. Labels and event processes in the asbestos operating system. In *SOSP '05 : Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 17–30, New York, NY, USA, 2005. ACM.

[FG95] R. Focardi and R. Gorrieri. A classification of security properties for process algebras. *JCS*, 3(1) :5–33, 1995.

[FGQ96] Simon Foley, Li Gong, and Xiaolei Qian. A security model of dynamic labeling providing a tiered approach to verification. *sp*, 00 :0142, 1996.

[FHS97] S. Forrest, S.A. Hofmeyr, and A. Somayaji. Computer immunology. *Communications of the ACM*, 40(10) :88–96, October 1997.

[FLR77] R. J. Feiertag, K. N. Levitt, and L. Robinson. Proving multilevel security of a system design. *SIGOPS Oper. Syst. Rev.*, 11(5) :57–65, 1977.

[Fra00] Timothy Fraser. Lomac : Low water-mark integrity protection for cots environments. In *SP '00 : Proceedings of the 2000 IEEE Symposium on Security and Privacy*, page 230, Washington, DC, USA, 2000. IEEE Computer Society.

[Fra06] Michael Franz. Moving trust out of application programs :. Technical report, Donald Bren School of Information and Computer Science, University of California, 2006.

[GBJS06] Gurvan Le Guernic, Anindya Banerjee, Thomas Jensen, and David A. Schmidt. Automata-based confidentiality monitoring. In *ASIAN*, pages 75–89, 2006.

[GJ05] Gurvan Le Guernic and Thomas Jensen. Monitoring information flow. In Andrei Sabelfeld, editor, *Proceedings of the Workshop on Foundations of Computer Security*, pages 19–30. DePaul University, JUN 2005.

[GM82] J. Goguen and J. Meseguer. Security policies and security models. In *IEEE Symposium on Research in Security and Privacy*, 1982.

[GM84] J. Goguen and J. Meseguer. Unwinding and inference control. In *IEEE Symposium on Security and Privacy*, 1984.

[GSB06] Thomas Gamer, Marcus Schöller, and Roland Bless. A Granularity-adaptive System for in-Network Attack Detection. In *Proceedings of the IEEE / IST Workshop on Monitoring, Attack Detection and Mitigation 2006*, Computer Networking and Internet CNI 2006-09-1, pages 47–50, Tuebingen, Germany, Sep 2006. Diadem Firewall Project (FP6 IST-2002-002154).

[HCF05a] V. Haldar, D. Chandra, and M. Franz. Dynamic taint propagation for java. In *Annual Computer Security Applications Conference (ACSAC)*, 2005.

[HCF05b] V. Haldar, D. Chandra, and M. Franz. Practical, dynamic information-flow for virtual machines. In *Programming Language Interference and Dependence (PLID'05)*, 2005.

[HOM06] William G. J. Halfond, Alessandro Orso, and Panagiotis Manolios. Using positive tainting and syntax-aware evaluation to counter sql injection attacks. In *SIGSOFT '06/FSE-14 : Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 175–185, New York, NY, USA, 2006. ACM.

[HY86] J. Thomas Haigh and William D. Young. Extending the noninterference version of mls for sat. *sp*, 0 :231, 1986.

[ITS91] ITSEC. Evaluation criteria of the information system security. Technical report, Office des publications officielles des Communautés européennes, 1991.

[Jul01] Klaus Julisch. Mining alarm clusters to improve alarm handling efficiency. In *Proceedings of the 17th Annual Computer Security Applications Conference (ACSAC), December 2001*, 2001.

[Kho05] Raphaël Khoury. Détection du code malicieux : système de type à effet. Master's thesis, FACULTE DES SCIENCES ET DE GENIE UNIVERSITE DE LAVAL, 2005.

[KR02] Calvin Ko and Timotyy Redmond. Noninterference and intrusion detection. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2002.

[KR03] Calvin Ko and Timothy Redmond. Detecting race-condition attacks using noninterference. *Network Associate Advanced Security Research Journal*, 5(1), 2003.

[KRL97] Calvin Ko, Manfred Ruschitzka, and Karl N. Levitt. Execution monitoring of security-critical programs in a distributed system : A specification-based approach. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, pages 175–187, Oakland, CA, May 1997.

[KYB+07] Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M. Frans Kaashoek, Eddie Kohler, and Robert Morris. Information flow control for standard os abstractions. In *Proceedings of the 21st Symposium on Operating Systems Principles*, Stevenson, WA, October 2007.

[KZZ06] Jingfei Kong, Cliff C. Zou, and Huiyang Zhou. Improving software security via runtime instruction-level taint checking. In *ASID '06 : Proceedings of the 1st workshop on Architectural and system support for improving software dependability*, pages 18–24, New York, NY, USA, 2006. ACM.

[LABW92] Butler Lampson, Martin Abadi, Michael Burrows, and Edward Wobber. Authentication in distributed systems : theory and practice. *ACM Trans. Comput. Syst.*, 10(4) :265–310, 1992.

[LaP99] Leonard J. LaPadula. State of the art in anomaly detection and reaction. Technical report, Center for Integrated Intelligence Systems - The MITRE Corporation, 1999.

[LB97] T. Lane and C. Brodley. An application of machine learning to anomaly detection. In *Proc. of the 20th National Information Systems Security Conference*, pages 366–380, October 1997.

[LcC06] Lap Chung Lam and Tzi cker Chiueh. A general dynamic information flow tracking framework for security applications. In *ACSAC '06 : Proceedings of the 22nd Annual Computer Security Applications Conference on Annual Computer Security Applications Conference*, pages 463–472, Washington, DC, USA, 2006. IEEE Computer Society.

[LJ99] Emilie Lundin and Erland Jonsson. Some practical and fundamental problems with anomaly detection. In *Proceedings of the fourth Nordic Workshop on Secure IT systems (NORDSEC'99)*, Kista, Sweden, November 1999.

[Low02] G. Lowe. Quantifying information flow. In *IEEE Computer Security Foundations Workshop*, 2002.

[LS98] W. Lee and S.J. Stolfo. Data mining approaches for intrusion detection. In *Proc. of the 7th Usenix Security Symposium*, January 1998.

[Lun88] Teresa F. Lunt. Automated audit trail analysis and intrusion detection : a survey. In *Proceedings of the 11th National Computer Security Conference*, pages 65–73, Washington, D.C., October 1988.

[LY99] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1999.

[LZ05a] Peng Li and Steve Zdancewic. Downgrading policies and relaxed noninterference. In *POPL '05 : Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 158–170, New York, NY, USA, 2005. ACM Press.

[LZ05b] Peng Li and Steve Zdancewic. Practical information-flow control in web-based information systems. In *CSFW '05 : Proceedings of the 18th IEEE Computer Security Foundations Workshop (CSFW'05)*, pages 2–15, Washington, DC, USA, 2005. IEEE Computer Society.

[Mad00] Dana Madsen. An operating system analog to the perl data tainting functionality. In *in Proceedings of the 23rd National Information Systems Security Conference*, 2000.

[Man00] Heiko Mantel. Possibilistic definitions of security - an assembly kit. In *CSFW '00 : Proceedings of the 13th IEEE workshop on Computer Security Foundations*, page 185, Washington, DC, USA, 2000. IEEE Computer Society.

[MB05] Ana Almeida Matos and Gerard Boudol. On declassification and the non-disclosure policy. In *CSFW '05 : Proceedings of the 18th IEEE Computer Security Foundations Workshop (CSFW'05)*, pages 226–240, Washington, DC, USA, 2005. IEEE Computer Society.

[McC88] Daryl McCullough. Noninterference and the composability of security properties. *sp*, 00 :177, 1988.

[McL92] John McLean. Proving noninterference and functional correctness using traces. *Journal of Computer Security*, 1(1) :37–58, 1992.

[McL94] John McLean. A general theory of composition for trace sets closed under selective interleaving functions. *sp*, 00 :79, 1994.

[ME08] Stephen McCamant and Michael D. Ernst. Quantitative information flow as network flow capacity. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation*, Tucson, AZ, USA, June 9–11, 2008.

[ML97] Andrew C. Myers and Barbara Liskov. A decentralized model for information flow control. *SIGOPS Oper. Syst. Rev.*, 31(5) :129–142, 1997.

[MMM+01] Ludovic Mé, Zakia Marrakchi, Cédric Michel, Hervé Debar, and Frédéric Cuppens. La détection d'intrusions : les outils doivent coopérer. *Revue de l'Electricité et de l'Electronique*, 5 :50–55, May 2001.

[Mor04] Benjamin Morin. *Corrélation d'alertes issues d'outils de détection d'intrusions avec prise en compte d'informations sur le système surveillé*. PhD thesis, INSA de Rennes, 2004.

[MR92] M. D. McIlroy and J. A. Reeds. Multilevel security in the unix tradition. *Softw. Pract. Exper.*, 22(8) :673–694, 1992.

[MS04] H. Mantel and D. Sands. Controlled declassification based on intransitive noninterference. In *Proc. Asian Symp. on Programming Languages and Systems*, volume 3302 of *LNCS*, pages 129–145. Springer-Verlag, NOV 2004.

[Mye99] Andrew C. Myers. Jflow : Practical mostly-static information flow control. In *Proceedings of the 26th ACM on Principles of Programming Languages*, 1999.

[NE02] Jeremy W. Nimmer and Michael D. Ernst. Invariant inference for static checking : An empirical evaluation. In *Proceedings of the 10th ACM SIGSOFT symposium on Foundations of software engineering (SIGSOFT '02)*, pages 11–20, New York, NY, USA, 2002. ACM Press.

[NSCT07] Srijith K. Nair, Patrick N. D. Simpson, Bruno Crispo, and Andrew S. Tanenbaum. A virtual machine based information flow control system for policy enforcement. In *First International Workshop on Run Time Enforcement for Mobile and Distributed Systems (REM 2007)*, pages 1–11, Dresden, Germany, 2007.

[NTGG+05] Anh Nguyen-Tuong, Salvatore Guarnieri, Doug Green, , Jeffrey Shirley, and David Evans. Automatically hardening web applications using precise tainting. In *IFIP Security Conference*, 2005.

[PS03] François Pottier and Vincent Simonet. Information flow inference for ml. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2003.

[QWL+06] Feng Qin, Cheng Wang, Zhenmin Li, Ho seop Kim, Yuanyuan Zhou, and Youfeng Wu. Lift : A low-overhead practical information flow tracking system for detecting security attacks. *39th Annual IEEE/ACM International Symposium on Microarchitecture*, 2006.

[RG99] A.W. Roscoe and M.H. Goldsmith. What is intransitive noninterference ? *csfw*, 00 :228, 1999.

[RMMG01] P. Ryan, J. McLean, J. Millen, and V. Gligor. Non-interference : Who needs it ? *csfw*, 0 :0237, 2001.

[Ros95] A.W. Roscoe. Csp and determinism in security modelling. *sp*, 00 :0114, 1995.

[RS99] P Y A Ryan and S A Schneider. Process algebra and noninterference. *csfw*, 00 :214, 1999.

[Rus92] John Rushby. Noninterference, transitivity and channel-control security policies. Technical report, SRI, 1992.

[Rya96] Peter Ryan. Panel : A genealogy of non-interference. In *CSFW '96 : Proceedings of the 9th IEEE workshop on Computer Security Foundations*, page 158, Washington, DC, USA, 1996. IEEE Computer Society.

[San93] Ravi S. Sandhu. Lattice-based access control models. *Computer*, 26(11) :9–19, 1993.

[Sch00a] Fred B. Schneider. Enforceable security policies. *Information and System Security*, 2000.

[Sch00b] Randal L. Schwartz. Taint so easy, is it ? *Sys Admin*, 9(8) :53–54, 2000.

[SCS98] R. Sekar, Yong Cai, and Mark Segal. A specification-based approach for building survivable systems. In *Proceedings of the 21st National Information Systems Security Conference (NISSC'98)*, pages 338–347, Crystal City, VI, October 1998.

[Sim03] Vincent Simonet. Flow caml in a nutshell. In *Proceedings of the first APPSEM-II workshop*, 2003.

[SLZD04] G. Edward Suh, Jae W. Lee, David Zhang, and Srinivas Devadas. Secure program execution via dynamic information flow tracking. *SIGARCH Comput. Archit. News*, 32(5) :85–96, 2004.

[SM02] Andrei Sabelfeld and Heiko Mantel. Static Confidentiality Enforcement for Distributed Programs. In *Proceedings of the 9th International Static Analysis Symposium, SAS'02*, LNCS 2477, pages 376–394, Madrid, Spain, September 17–20 2002. Springer-Verlag.

[SM03a] A Sabelfeld and A C Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1), 2003.

[SM03b] Andrei Sabelfeld and Andrew C. Myers. A model for delimited information release. In *ISSS*, pages 174–191, 2003.

[Smi01] Geoffrey Smith. A new type system for secure information flow. *csfw*, 00 :0115, 2001.

[Smi07] Geoffrey Smith. Principles of secure information flow analysis. In *Malware Detection*, volume 27. Springer-Verlag, 2007.

[SRC84] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM Trans. Comput. Syst.*, 2(4) :277–288, 1984.

[SS07] A. Sabelfeld and D. Sands. Declassification : Dimensions and principles. *Journal of Computer Security*, 2007. To appear.

[SST07] Paritosh Shroff, Scott F. Smith, and Mark Thober. Dynamic dependency monitoring to secure information flow. *csf*, 00 :203–217, 2007.

[SV98] Geoffrey Smith and Dennis Volpano. Secure information flow in a multi-threaded imperative language. In *POPL '98 : Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 355–364, New York, NY, USA, 1998. ACM.

[TH05] David Thomas and Andrew Hunt. *Programming Ruby : the pragmatic programmer's guide*. The Pragmatic Programmers, LLC., Raleigh, NC, USA, 2 edition, August 2005.

[TZ04] Stephen Tse and Steve Zdancewic. Run-time principals in information-flow type systems. In *IEEE Symposium on Security and Privacy*, 2004.

[US01] Prem Uppuluri and R. Sekar. Experiences with specification-based intrusion detection. In W. Lee, L. Mé, and A. Wespi, editors, *Proceedings of the Fourth International Symposium on the Recent Advances in Intrusion Detection (RAID'2001)*, number 2212 in LNCS, pages 172–189, Davis, CA, October 2001.

[VD97] Smith G. Volpano D. A type-based approach to program security. In *Theory and Practice of Software Development*, 1997.

[VIS96] Dennis Volpano, Cynthia Irvine, and Geoffrey Smith. A sound type system for secure flow analysis. *Journal in Computer Security*, 4(2-3) :167–187, 1996.

[VS99] Dennis Volpano and Geoffrey Smith. Probabilistic noninterference in a concurrent language. *J. Comput. Secur.*, 7(2-3) :231–253, 1999.

[VXDS06] V. N. Venkatakrishnan, Wei Xu, Daniel C. DuVarney, and R. Sekar. Provably correct runtime enforcement of non-interference properties. In Peng Ning, Sihan Qing, and Ninghui Li, editors, *ICICS*, volume 4307 of *Lecture Notes in Computer Science*, pages 332–351. Springer, 2006.

[Wei69] Clark Weissman. Security controls in the adept-50 timesharing system. In *In AFIPS Conference Proceedings*, pages 119–133, 1969.

[WJ90] J. Todd Wittbold and Dale M. Johnson. Information flow in nondeterministic systems. *sp*, 00 :144, 1990.

[WPS06] Ke Wang, Janak J. Parekh, and Salvatore J. Stolfo. Anagram : A Content Anomaly Detector Resistant to Mimicry Attack. In Diego Zamboni and Christopher Kruegel, editors, *Recent Advances in Intrusion Detection*, volume 4219 of *Lecture Notes in Computer Science*, pages 226–248. Springer, 2006.

[XBS06] Wei Xu, Sandeep Bhatkar, and R. Sekar. Taint-enhanced policy enforcement : a practical approach to defeat a wide range of attacks. In *USENIX-SS'06 : Proceedings of the 15th conference on USENIX Security Symposium*, pages 9–9, Berkeley, CA, USA, 2006. USENIX Association.

[YYW+07] Sachiko Yoshihama, Takeo Yoshizawa, Yuji Watanabe, Michiharu Kudo, and Kazuko Oyanagi. Dynamic information flow control architecture for web applications. In Joachim Biskup and Javier Lopez, editors, *ESORICS*, volume 4734 of *Lecture Notes in Computer Science*, pages 267–282. Springer, 2007.

[ZBWKM06] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making information flow explicit in histar. In *OSDI '06 : Proceedings of the 7th symposium on Operating systems design and implementation*, pages 263–278, Berkeley, CA, USA, 2006. USENIX Association.

[Zda04] Steve Zdancewic. Challenges for information-flow security. In *Proceedings of the 1st International Workshop on Programming Language Interference and Dependence*, 2004.

[Zim03] Jacob Zimmermann. *Détection d'intrusions paramétrée par la politique par contrôle de flux de références*. PhD thesis, Université de Rennes 1, 2003.

[ZM04] Lantian Zheng and Andrew C. Myers. Dynamic security labels and noninterference (extended abstract). In *Formal Aspects in Security and Trust*, pages 27–40, 2004.

[ZMB02] Jacob Zimmermann, Ludovic Mé, and Christophe Bidan. Introducing reference flow control for detecting intrusion symptoms at the os level. In Andreas Wespi, Giovanni Vigna, and Luca Deri, editors, *Proceedings of the 5th International Symposium on Recent Advances in Intrusion Detection (RAID'2002)*, volume 2516 of *Lecture Notes in Computer Science*, pages 292–306. Springer, 2002.

[ZMB03] Jacob Zimmermann, Ludovic Mé, and Christophe Bidan. Experimenting with a policy-based hids based on an information flow control model. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, December 2003.

[ÚES99] Úlfar Erlingsson and Fred B. Schneider. Sasi enforcement of security policies : a retrospective. In ACM Press, editor, *Proceedings of the 1999 workshop on New security paradigms*, pages 87–95, 1999.