Proposition et implémentation d'une coopération entre deux moniteurs de flux d'information

Thomas Letan

Master Recherche Informatique

Rapport de stage

Proposition et implémentation d'une coopération entre deux moniteurs de flux d'information

Auteur :
Thomas Letan

Superviseur :
Guillaume Hiet
CIDre

INSA RENNES

Remerciements

Je remercie Guillaume Hiet qui en plus d'écrire un sujet de stage rien que pour moi m'a été d'une grande aide pour préparer ma future vie professionnelle. Je remercie Christophe Hauser et Guillaume Brogi pour avoir supporté vaillamment mes très nombreuses questions. Je remercie enfin toute l'équipe CIDre pour son accueil chaleureux.

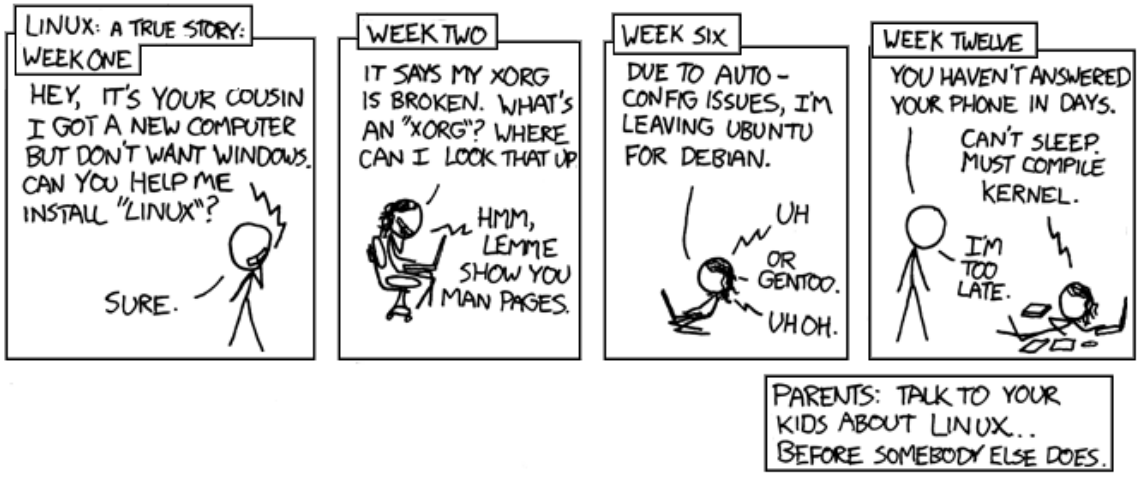


Table des matières

Résumé

Blare est un projet de recherche de l'équipe CIDre (Supelec) visant à faire de la détection d'intrusion grâce à du suivi de flux. Contrairement à la plupart modèle du domaine qui visent à connaître le degré de confidentialité d'un conteneur en fonction des informations qu'il contient, le modèle de Blare permet de savoir quelles sont les informations d'un conteneur et leur provenance. Blare a été implémenté, entre autre, au niveau OS (kBlare pour le noyau Linux) et au niveau langage (jBlare pour le Java). Le sujet de mon stage était de proposer puis d'implémenter un mécanisme de coopération entre ces deux moniteurs.

Après avoir étudié l'état de l'art du domaine, je me suis intéressé plus particulièrement aux travaux abordant la problématique de la coopération entre moniteurs de flux d'information. Si de nombreux travaux proposent des approches permettant le suivi dynamique des flux d'information au niveau langage [HCF05, Ass11, MZZ+01] ou au niveau OS [EKV+05, VTTCM10], Laminar est, à ma connaissance, le seul à proposer un mécanisme de coopération entre différent niveau de suivi. En outre, l'approche proposée repose sur des hypothèses qui se sont avérées incompatibles avec l'approche utilisée dans Blare.

Ma première préoccupation a donc été de proposer et de définir formellement un mécanisme de coopération générique qui ne dépende pas d'une implémentation particulière au niveau langage particulière. Ensuite, j'ai dû étudier en détail le fonctionnement d'une machine virtuelle Java pour définir plus précisément l'utilisation de ce mécanisme pour une coopération entre kBlare et jBlare. L'implémentation que j'ai finalement réalisée a donné les résultats attendus sur des petits programmes de tests et constitue en un *proof of concept*.

Mots clefs : Suivi de flux, détection d'intrusion, noyau Linux, JVM

Première partie

Introduction

Pour assurer la sécurité d'un système informatique, il convient en premier lieu de définir puis de mettre en œuvre une politique de sécurité. Celle-ci peut s'exprimer en termes d'exigences concernant la confidentialité (seules les personnes autorisées peuvent lire les données), l'intégrité (seules les personnes autorisées peuvent modifier les données) et la disponibilité (les personnes autorisées peuvent effectivement accéder aux données). La mise en œuvre de la politique passe par l'utilisation de méthodes préventives (par exemple, Linux implémente un contrôle d'accès DAC [Lam74]). Ces méthodes sont nécessaires mais souvent insuffisantes et il convient de s'assurer, *a posteriori*, du respect de la politique. La détection d'intrusion consiste à surveiller un système informatique dans le but de détecter les tentatives d'intrusion, c'est-à-dire un contournement de la politique de sécurité. Une attaque contre un système est la tentative par un acteur (logiciel ou humain) de réaliser une intrusion. Une intrusion repose sur l'exploitation de vulnérabilités du système, par exemple des bogues logiciels.

Il existe deux approches pour détecter les intrusions. Les IDS (*Intrusion Detection System*) *signature-based* [DDW99] s'attachent à repérer dans les données du système (par exemple, les journaux fournis par le système d'exploitation pour un *host-based* IDS ou les paquets réseaux pour un *network-based* IDS) qu'ils observent les « signatures » d'attaques ou d'intrusions connues. Le grand défaut de cette approche est de ne pouvoir détecter que les exploitations de vulnérabilités connues. L'autre approche utilisée dans la détection d'intrusion est l'approche comportementale [DDW99] : plutôt que de rechercher une attaque, on modélise le comportement normal du système et on vérifie que ce dernier ne s'en éloigne pas. La difficulté est justement dans la définition de ce comportement. Les IDS auxquels nous allons nous intéresser sont des IDS dits *policy-based* [KR02]. Dans cette approche, le comportement normal du système est modélisé par l'expression d'une politique de sécurité.

KBlare [VTTCM10] et jBlare [Ass11] sont deux IDS comportementaux *policy-based* et basés sur les flux d'information qui ont été développés par l'équipe CIDre. Ils implémentent le même modèle à deux niveaux différents : le premier au niveau du noyau Linux et le second au niveau de la JVM. L'équipe CIDre cherche à les faire coopérer. En effet, kBlare surveille tous les flux du système mais il est obligé de sur-approximer l'action des processus. Si un processus accède à deux fichiers, kBlare estime qu'il a mélangé leurs contenus même si ce n'est pas le cas. JBlare, quant à lui, surveille les flux au sein des applications Java mais son utilisation nécessite de spécifier explicitement les labels de sécurité. Une coopération entre kBlare et jBlare permettrait à kBlare de ne plus sur-approximer les flux d'information des applications Java et dispenserait le développeur de devoir spécifier les labels de son code Java, les niveaux de sécurité des variables étant hérités de kBlare.

Le présent rapport est organisé comme suit : dans une première partie, nous présentons un état de l'art du contrôle d'accès et de la détection d'intrusion. Nous développons les raisons qui justifient une coopération entre deux moniteurs de flux d'informations et nous nous intéressons plus particulièrement à Laminar [RPB+09], qui propose un mécanisme similaire à ce que nous voulons développer. Ensuite, nous proposons un modèle de sémantique formelle qui nous permet de décrire les modèles de kBlare et jBlare, ainsi que le modèle de coopération que nous proposons. Après avoir discuté de l'implémentation de ce

mécanisme de coopération, nous terminons en discutant de nos expérimentations sur son efficacité, en terme de précision des suivis de flux comme de performances

Deuxième partie
État de l'art

Notre objectif est de proposer un mécanisme de coopération entre deux moniteurs de flux d'information. Ces derniers peuvent être utilisés dans les systèmes d'information pour s'assurer du respect de la politique de sécurité. Après avoir présenté les mécanismes permettant de s'assurer du respect *a priori* et *a posteriori* de la politique de sécurité, nous nous intéressons plus particulièrement aux moniteurs de flux d'information avant d'expliquer les raisons qui nous motivent à envisager une coopération entre deux moniteurs de niveaux différents.

1 Sécurité d'un système d'information

L'administrateur sécurité définit, pour un système d'information, une politique de sécurité qui fixe des objectifs en terme de confidentialité, d'intégrité et de disponibilité. Une fois cette politique définie, il convient d'implémenter des mécanismes permettant d'assurer son respect.

1.1 Contrôle d'accès

Le contrôle d'accès vise à assurer que chaque accès à une ressource se fait dans le respect de la politique de sécurité établie. Il existe plusieurs modèles proposés dans la littérature, nous n'en présenterons que deux : le DAC (implémenté par défaut dans Linux ou Windows) et le MAC [SS94] (implémenté dans les extensions de sécurité comme SELinux [SVS01] ou AppArmor [Bau06]).

1.1.1 DAC

Le DAC (pour *Discretionary Access Control*) est le modèle de contrôle d'accès utilisé par défaut dans la plupart des systèmes d'exploitation. Il garantit que tous les accès directs à une ressource sont explicitement autorisés par la politique de sécurité. Ainsi, les différents utilisateurs possèdent des permissions de réaliser des actions (typiquement lecture, écriture, exécution) sur des objets. Il est dit discrétionnaire car la délégation (le transfert de certaines permissions) est à la discrétion des propriétaires des objets, qui peuvent de ce fait donner leurs permissions à d'autres utilisateurs. Dans le DAC de Linux, cet aspect de délégation transparaît dans les commandes `chown` et `chmod`.

1.1.2 MAC

À l'inverse d'un DAC traditionnel, un modèle MAC (pour *Mandatory Access Control*) ne permet pas la délégation sans contrôle. Cela signifie que la politique de sécurité est entièrement définie par un administrateur et les utilisateurs doivent obtenir l'accord d'une autorité centrale administrative pour déléguer un droit.

Le modèle de Bell et LaPadula [BL73] est le premier modèle à considérer une approche dite « multi-niveaux, » communément appelé MLS pour *multi-level security*. Il vise à assurer des propriétés de confidentialité. Les modèles MLS sont une approche possible permettant de réaliser une politique d'accès MAC. Dans le modèle de Bell et LaPadula, on distingue deux types d'acteurs : les sujets (actifs) et les conteneurs (passifs). Le modèle définit aussi un ensemble de niveaux d'information (par exemple, $Public$, $Secret$, $Confidentiel$) et un ordre partiel de cet ensemble. À chaque conteneur, on associe un niveau en lien avec l'information qu'il contient. Ainsi, le fichier `/etc/passwd` pourra être de niveau confidentiel, du fait qu'on ne désire pas que son contenu puisse être lu par n'importe qui. Chaque sujet u reçoit un niveau d'habilitation s_u. Lorsqu'il décide d'accéder au système d'information, l'utilisateur ouvre une session avec un niveau d'habilitation s_c, avec $s_c \leq s_u$. Par la suite, le système s'assure du respect de deux règles fondamentales : le *no read up* et le *no write down*. Ces règles ont été pensées pour que la session d'un utilisateur de confiance ne soit pas détournée par un programme malicieux.

No read up Si un sujet active une session de niveau s_c, alors il ne peut lire des conteneurs que de niveaux s_r, si $s_r \leq s_c$. Cela signifie tout simplement qu'un sujet ne peut lire une information d'un niveau de confidentialité plus haut que le sien.

No write down Bell et LaPadula ont imaginé un modèle qui assure des propriétés de confidentialités sur un système d'information. La règle du *no write down* s'assure qu'une information d'un niveau s ne fuit pas dans un conteneur de niveau s' si $s' \leq s$. Ainsi, un sujet ayant activé une session avec un niveau s_c ne pourra écrire que dans des conteneurs de niveaux s_w, avec $s_c \leq s_w$. Cette propriété permet d'écraser des données de hauts niveaux de confidentialités avec des données de niveaux plus faibles, ce qui n'est pas un problème car le modèle ne se soucie pas de l'intégrité des données.

Il existe un modèle analogue à Bell et LaPadula s'assurant quant à lui de l'intégrité des données : le modèle de Biba [Bib77] se place dans un contexte similaire et énonce des règles analogues : *no write up* (des données de basses intégrités ne peuvent être écrites dans des données de hauts intégrités) et *no read down* (un sujet ne peut récupérer des données de basses intégrités pour les écrire dans des fichiers de hautes intégrités).

1.2 Détection d'intrusion

Les mécanismes précédemment décrits sont censés assurer le respect de la politique de sécurité définie par l'administrateur, mais ces derniers ne sont pas à l'abri de vulnérabilités ou d'erreurs rendant possible leur contournement. Cette possibilité justifie la surveillance du système afin de détecter *a posteriori* les dits contournements. On parle d'IDS, pour *Intrusion Detection System* [DDW99].

Types de détecteurs Les IDS *Host-based* (HIDS) ont à leur disposition les informations fournies par le systèmes d'exploitation. Elles peuvent être de plusieurs natures : journaux du système, traces des appels système, etc.

Les IDS *Network-based* (NIDS) sont des sondes qui analysent les paquets transitant sur le réseau. Les NIDS analysent les paquets et plus particulièrement les *payloads* (leur taille, leur contenu, etc.).

Les IDS *Application-based* (AIDS) analysent les données d'une application en particulier. On peut par exemple citer WAF (*Web Application Firewall*) qui vise à assurer la protection des applications web en filtrant en amont les requêtes qui lui sont soumises (par exemple, *ModSecurity* pour le serveur web Apache). WebSTAT [VRKK03] est un autre projet de recherche qui analyse les requêtes faites à des serveurs web à la recherche de comportements malicieux. Les auteurs ont défini un langage de description précis pour décrire des attaques en plusieurs étapes. Ces deux exemples ne sont pas des NIDS dans le sens où ils sont intégrés à des applications (serveurs) et ne s'intéressent qu'à un protocole en particulier, clairement identifié. Un NIDS s'intéresse aux *payloads* bruts, il surveille tout le trafic et doit décoder les différentes couches protocolaires. Un AIDS est beaucoup plus robuste mais est très spécifique.

Approche de détections Il existe deux approches pour faire de la détection d'intrusion : l'approche dite « par signature » et l'approche dite « comportementale ».

La détection par signature vise à repérer des motifs connus dans, par exemple, des paquets IP. Parmi les IDS par signatures, Snort [R+99] est sans doute l'un des plus populaire. Il s'agit d'un NIDS dont le moteur s'appuie sur le *pattern matching* [KS94]. Il existe aussi des méthodes plus complexes nécessaire afin de lutter contre l'obfuscation. Il est possible de modéliser une intrusion comme une suite d'états du système. C'est l'approche retenue par USTAT [Ilg93] ou par EMERALD [PN97], ce dernier utilisant un système expert.

La plupart des antivirus du marché utilisent l'approche par signatures qui a cependant le défaut de nécessiter des bases de signatures continuellement mises à jour. Un détecteur par signature sera de plus impuissant face aux vulnérabilités dites *0-day*, c'est à dire qui n'ont pas encore été publiées et qui sont donc inconnues.

Au contraire, les détecteurs comportementaux cherchent à modéliser le comportement « normal » du système qu'ils surveillent [Den87]. Ils se servent ensuite de ce modèle pour détecter quand le système s'en éloigne : dans ce cas, ils supposent que le système a été compromis.

Il existe plusieurs façons de faire de la détection comportementale. La première problématique est de construire le modèle, la seconde est de savoir quand le système s'éloigne effectivement de son comportement normal. Plusieurs modèles utilisent l'apprentissage automatique (*machine learning*) [Axe00, HSB+12], la démarche est alors souvent la même. La première étape est de sélectionner un jeu de données, dont la première moitié servira à construire le modèle et la seconde à le valider. La phase d'apprentissage peut en effet avoir lieu en conditions réelles, avec le risque d'observer des attaques et de les intégrer au comportement normal. L'autre solution est d'utiliser un jeu de données artificiel, dont on est sûr qu'il ne contient aucune attaque. Toutefois, il est difficile de s'assurer que ce jeu de donnée soit réaliste et couvre l'intégralité des cas légaux. Néanmoins, les IDS basés sur le *machine learning*, s'ils sont fortement développés dans le monde de la recherche, ne rencontrent pas ou peu de succès dans les entreprises, en utilisation « réelle ». L'explication avancée est que les cas d'utilisation où les *machine learning* sont efficaces sont assez éloignés de la détection d'intrusion [SP10].

Ce manque de résultats a poussé la mise au point d'autres approches comportementales. En particulier, dans l'approche paramétrée par la politique de sécurité, l'IDS s'appuie sur une spécification de la politique de sécurité et vérifie que le comportement du système est explicitement autorisé par la politique. L'implémentation kBlare du projet Blare de

l'équipe CIDre est un IDS *policy-based* qui s'appuie sur le contrôle des flux d'information. Nous présentons donc par la suite les travaux relatifs au contrôle de flux et leur application à la détection d'intrusions.

2 Contrôle de flux

Le contrôle d'accès vise à réguler les interactions des sujets avec les objet ou conteneurs d'information. Toutefois, lorsqu'un accès est autorisé (par exemple, en lecture), ce type d'approche ne permet pas, en général, de contrôler la dissémination des informations. Les modèle de contrôle d'accès multi niveaux permettent certes de contrôler les flux d'information. Toutefois, ces approches sont très restrictives car elles nécessitent d'associer un niveau de sécurité à chacun des objets et restreint les accès en fonction de ce niveau. Ce niveau de sécurité est associé au conteneur et n'évolue pas (sauf en cas de changement de politique). Les mécanismes de suivi et de contrôle des flux d'information permettent quand à eux d'exprimer et de vérifier des propriétés de sécurité « de bout en bout ». Il s'agit donc d'un mécanisme complémentaire au contrôle d'accès. Le principe consiste à associer une étiquette (aussi appelée *tag* ou label) à chaque conteneur d'information et décrivant leur contenu. Les travaux précédant du domaine peuvent être classés en deux catégorie : ceux implémentant un mécanisme de contrôle au niveau du système d'exploitation (OS) ou au niveau langage.

2.1 Au niveau OS

2.1.1 Principe

Un moniteur de flux au niveau d'un OS infère les flux en fonction des différents appels système qu'il intercepte. Cette approche permet d'avoir une vue globale du système et de connaître toutes les interactions entre ses différents composants. Malheureusement, pour ce type d'approche les les processus sont considérés comme des boîtes noires dans lesquelles les flux internes à l'application sont totalement inconnus. De ce fait, les moniteurs au niveau OS se placent toujours « dans le pire cas », c'est-à-dire dans l'hypothèse où les informations sont toujours mélangées : cela entraîne une sur approximation des flux.

2.1.2 Projets de recherche

Il existent plusieurs modèles de suivi de flux au niveau OS. Nous avons choisi d'en présenter deux. HiStar, dans un premier temps, est un prototype de système d'exploitation intégrant nativement un suivi de flux. Il possède une approche qui est relativement proche de la plupart des autres moniteurs OS, basé sur la classification de l'information en différents niveaux. KBlare, quant à lui, adopte une démarche différente, car il ne classe pas les informations entre elles.

HiStar Histar [ZBWKM06] est un prototype de système d'exploitation implémentant un système de DIFC [ML00](*Decentralized Information Flow Control*). Contrairement à un MAC centralisé autour d'une entité centrale, un DIFC privilégiera un modèle décentralisé où les applications peuvent d'elle-même (selon certaines conditions) déclassifier les informations. Le but est d'obtenir un modèle plus flexible et facile à administrer qu'un MLS strict.

HiStar est la suite directe d'Asbestos [EKV+05] mais cherchant à réduire au maximum la quantité de code de confiance.

Le modèle de HiStar repose sur les labels d'Asbestos. À chaque sujet du système, on associe un label de réception L_R symbolisant le label maximum que peut atteindre ce conteneur et un label d'envoi L_E symbolisant le niveau courant ; on parle aussi de contamination courante. Pour qu'un flux de p vers q soit possible, il faut que $L_E(p) \sqsubseteq L_R(p)$. Si le flux est possible, alors on met à jour le label d'envoi de p, avec $L_E(p) = L_S(q) \sqcap L_E(p)$.

Les labels d'Asbestos ne sont pas de simples niveaux d'habilitation comme dans un Bell et LaPadula classique. Ils sont étroitement liés à la notion de DIFC. En effet, les labels reposent sur la notion de catégories d'information : chaque processus peut déclarer de nouvelles catégories d'information en fonction de ses besoins. Un label est alors une fonction qui donne pour chaque catégorie d'information un niveau. Un label s'écrira sous la forme $L = \{w0, r3, 1\}$, avec $L(w) = 0$, $L(r) = 3$ et $L(c) = 1$ si $c \neq r$ et $c \neq w$. Les auteurs d'Asbestos introduisent la relation d'ordre $L_1 \sqsubseteq L_2$ si $\forall c, L_1(c) \leq L_2(c)$.

L'ensemble des niveaux est quant à lui limité : $\{\star, 0, 1, 2, 3\}$ avec $\star$ un niveau particulier signifiant que le flux est autorisé quoiqu'il arrive, sans contamination. C'est ce niveau qui rend la déclassification d'une information possible.

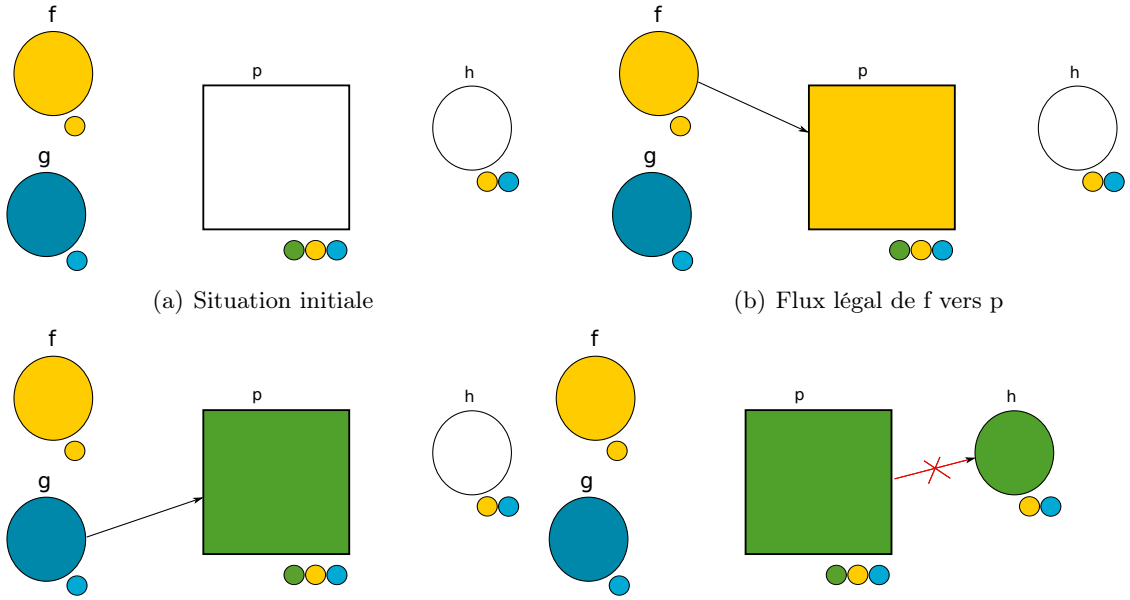
Le parti pris de HiStar est d'implémenter un prototype d'OS totalement nouveau, avec des abstractions différentes du monde Unix conçues pour réaliser du contrôle dynamique des flux d'informations. Le désavantage de cette approche est qu'avec l'état du marché actuel, HiStar a peu de chance de se faire une place dans l'écosystème des systèmes d'exploitation existants et utilisés.

kBlare kBlare [VTTCM10] est un autre moniteur de flux d'information implémenté au niveau OS. Si son modèle utilise des concepts similaires à celui de HiStar, son utilisation est quant à elle totalement différente. Là où HiStar fait du contrôle de flux, c'est à dire qu'il permet ou non à un processus d'engendrer des flux vers des ressources selon une politique de flux d'information, kBlare est quant à lui utilisé comme un IDS. Plus particulièrement, il est un IDS *Host-based* et *Policy-based*.

L'originalité de kBlare par rapport à d'autres solutions comme HiStar est de permettre de garder un historique des informations présentes dans un conteneur (processus, fichiers, pages mémoires, etc).

La Figure 1 présente une simplification du modèle de kBlare, dans laquelle f, g et h sont des fichiers et p un processus. Dans kBlare, le contenu des fichiers est teinté (ici, par une couleur). Ainsi, au départ (a) f contient l'information jaune et g l'information bleue, tandis que p et h sont vu comme des conteneurs vides. Après un flux de f vers p (b) (qui peut se traduire concrètement par « p lit le contenu de f »), l'information contenue par p est teintée p. Quand un second flux de g vers p a lieu (c), l'information contenue par p change de teinte pour devenir verte : on sait ainsi qu'il s'agit d'un mélange de bleu et de jaune, d'où la notion d'historique.

La politique de sécurité du système est exprimée dans kBlare grâce à des *tags* que l'on attache aux conteneurs et qui permettent de spécifier quelles sont les informations autorisées. Ainsi, dans la Figure 1, on a attaché un *tag* jaune à f, ce qui signifie que ce fichier ne peut contenir que l'information f. Le processus p, quant à lui, peut contenir l'information bleue, l'information jaune et l'information verte ce qui fait des flux déjà décrits



(a) Situation initiale

(b) Flux légal de f vers p

(c) Flux légal de g vers p

(d) Flux illégal de p vers h

FIGURE 1 – Blare simplifié au maximum

des flux légaux. À l'inverse, le fichier h ne peut contenir qu'une information jaune ou bleue mais pas verte, donc un flux de p vers h (d) est illégal.

Le plus grand avantage d'un moniteur de flux d'informations au niveau OS est d'avoir connaissance des flux engendrés par toutes les applications du système. L'inconvénient majeur qui se retrouve dans tous les modèles est que le processus est vu comme une boîte noire et les flux internes aux applications inconnues : comme il ne peut savoir quelles sont les informations véritablement utilisées, le moniteur se placera toujours dans le « pire cas », à savoir dans l'hypothèse où le processus mélange toutes les informations qu'il possède.

2.2 Au niveau langage

Le suivi de flux au niveau langage [SM03, Zda04] se fait principalement selon deux écoles. La première s'appuie sur une analyse statique de code qui permet de s'assurer du respect d'une politique pour toutes les exécutions possibles. La seconde repose sur une analyse dynamique qui vérifie à l'exécution du programme le respect de la politique dans cette exécution en particulier. Certains moniteurs prennent quant à eux le parti de ne pas exclure une analyse au profit d'une autre : on parle d'analyses hybrides.

Nous présentons trois moniteurs illustrant chacun un type d'analyse. Le dénominateur commun de ces moniteurs est qu'ils visent le langage Java, particulièrement utilisé ces dernières années.

Approche statique : Jif [MZZ+01] dote le langage Java d'un support du contrôle de flux. Une analyse statique des flux d'information permet de s'assurer du respect de la confidentialité et de l'intégrité des données. Jif étend le Java en ajoutant la possibilité d'attacher des labels aux variables. Ces labels sont en fait des types de sécurité. Par exemple,

`int { Alice` $\rightarrow$ `Bob } x;` signifie que la variable x est possédée par le sujet *Alice* et que le sujet *Bob* peut le voir.

Le compilateur Jif prend en entrée les programmes ainsi étendus. Après avoir fait une vérification de typage (est-ce que les types de sécurité sont compatibles entre eux ?), il les compile en Java pur qu'un compilateur classique se charge de transformer en un `.class`. Le programme obtenu peut donc être exécuté par une JVM non modifiée mais possède des propriétés de sécurité assurées.

Il existe des solutions similaires à Jif implémentées dans d'autres langages comme par Flow Caml [Sim03] qui est une sur-couche d'OCaml.

Approche dynamique : Vivek Haldar et al. ont proposé une propagation de teintes dynamiques pour Java [HCF05]. Quand Jif s'assure que la politique de flux est respecté *pour toutes les exécutions possibles* en effectuant une analyse statique à la compilation, cette approche vérifie *lors de chaque exécution* que la politique est respectée. C'est une démarche moins efficace en terme de performance, car il faut faire des contrôles à chaque fois, mais elle est plus flexible, car elle ne s'intéresse qu'aux exécutions qui ont véritablement lieu. L'analyse de Jif pourrait en effet échouer dans des cas peu probables voir jamais rencontré à l'exécution.

Les données teintées sont les entrées utilisateurs et, grâce à la propagation des teintes, il est possible d'éviter d'utiliser des entrées de l'utilisateur dans des contextes où la sécurité est importante. L'implémentation faite est indépendante de la JVM et se base sur une instrumentation du *bytecode*.

Approche hybride : jBlare [Ass11] est un moniteur de flux d'information de *bytecode*. Il s'agit d'une JVM (JamVM) modifiée qui base sa propagation de *tags* sur une double analyse statique et dynamique, la première permettant d'optimiser la seconde. Ainsi, l'analyse statique permet de limiter les opérations de propagation de *tags* au strict minimum. Elle permet aussi de prendre en compte les flux implicites, liés par exemple au branchement conditionnel. Pour comprendre ce qu'est un flux implicite, le plus simple est d'observer le code suivant :

Le modèle de jBlare est le même que celui de kBlare pour ce qui concerne le suivi de flux, à savoir qu'on teinte les informations qui circulent dans le programme et qu'on suit leur parcours dans les différentes variables qui les contiennent.

```
if (a < 0) {
  x = 1;
} else {
  y = 2;
}
```

En fonction de la valeur de a, x prendra une valeur différente. C'est exactement ce que l'on entend par flux implicite de a vers x. De tels flux sont invisibles pour un moniteur purement dynamique.

3 Coopération de moniteurs

3.1 Pourquoi coopérer ?

Un moniteur de flux au niveau OS possède une vision complète du système qu'il surveille mais sera incapable de suivre les flux à l'intérieur des processus. Cela signifie qu'il doit toujours envisager « le pire cas » et partir du principe que les processus utilisent toujours les informations qu'ils possèdent lorsqu'ils modifient la valeur d'un conteneur. Ainsi, si un processus lit un fichier f_1 puis un fichier f_2 avant d'écrire dans un fichier f_3, il supposera que ce dernier récupérera toutes les informations contenues dans f_1 et f_2. Cette hypothèse est obligatoire pour ne pas faire de sous approximation des flux (ce qui conduit à avoir des faux négatifs, des flux jugés légaux alors qu'ils ne le sont pas) mais elle entraîne l'effet inverse, à savoir une sur approximation des flux (avec l'apparition de faux positifs, des flux qui sont jugés illégaux alors qu'ils ne le sont pas). Ce problème se rencontre dans HiStar comme dans kBlare ou n'importe quelle autre implémentation d'un moniteur au niveau OS.

Au contraire, un moniteur de flux d'information au niveau langage suit les flux d'information à l'intérieur des programmes en permettant une granularité de l'ordre de la variable. Cette approche comporte néanmoins des inconvénients majeurs. Il est par exemple difficile d'étiqueter les informations du programme. De nombreuses questions se posent : faut-il utiliser une étiquette étiqueter toutes les constantes du programme ? Comment gérer les entrées utilisateur, faut-il utiliser une étiquette commune ? Ou faut-il se baser sur le contexte ? Une entrée utilisateur correspondant à la recopie d'un *captcha* n'a pas à recevoir le même niveau de confidentialité que celle qui correspond à la saisie d'un mot de passe. La difficulté d'établir une politique de flux ne se limite pas à cela. Quand le conteneur type est la variable, la granularité est très fine ; un programme peut contenir plusieurs milliers de variables et il serait très long de leur spécifier à chacune un label. Il est aussi parfois difficile de pouvoir prédire *a priori* le type d'information que pourra contenir une variable et donc de leur associer un label adéquat. En effet, certaines variable peuvent contenir, durant l'exécution du programme, des données de différentes natures.

Nous souhaitons nous servir de la connaissance d'un moniteur de flux au niveau applicatif pour améliorer la précision d'un moniteur de flux d'information au niveau OS. L'avantage d'une telle démarche est que le développeur n'a plus à spécifier explicitement les labels associé à chaque variable.

Une communication entre les niveaux OS et langage implique une communication entre l'espace noyau et l'espace utilisateur. Plusieurs moyens existent [Lov10] et tous possèdent leurs avantages et leurs inconvénients. Les appels système, par exemple, sont simples à implémenter mais les contributeurs du noyau essayent d'en ajouter le moins possible. Les communications basées sur les systèmes de fichiers (`/proc`, par exemple) ne permettent pas à l'espace noyau de notifier facilement à l'espace utilisateur un changement. Une autre façon de communiquer reposant sur les *sockets* permet ce genre de notification [NAGL10]. Savoir par quels moyens nos deux moniteurs communiqueront n'est qu'une partie de notre problème. Il faut aussi déterminer à quel moment cette communication aura lieu et quelles seront les informations échangées.

3.2 kBlare et jBlare

Nous avons avons choisi de considérer notre mécanisme de coopération avec kBlare comme moniteur de flux d'information au niveau OS. Notre choix pour le niveau applicatif s'est porté sur jBlare, car il implémente le même modèle de suivi de flux. Une coopération avec jBlare permettrait ainsi de diminuer le taux de faux positif de kBlare, en tout cas dans le cadre des applications Java.

De plus, kBlare et jBlare s'inscrivent dans un projet plus large de l'équipe CIDre (le projet Blare). On peut citer AndroBlare [ATM12] qui est un portage de kBlare sur Android. De la même façon, un deuxième moniteur au niveau applicatif est en cours d'étude dans l'équipe, cette fois intégré à un navigateur internet. En pensant notre mécanisme de coopération de façon générique (sans faire d'hypothèse sur le moniteur applicatif), alors il permet d'envisager de l'utiliser dans le cadre des autres travaux du projet Blare.

3.3 Laminar

Le projet Laminar est, à notre connaissance, le seul qui ce soit intéressé à la coopération entre un moniteur de flux d'information de niveau OS et un moniteur d'information de niveau langage. Nous n'avons trouvé que Laminar. Les auteurs de Laminar avaient un objectif proche du notre, à savoir améliorer la précision de leur moniteur au niveau OS en établissant une coopération avec un moniteur (pour Java) au niveau application.

3.3.1 Modèle

Les auteurs de Laminar considèrent deux entités : les *principals* (entités actives) et les objets (conteneurs d'informations). On s'intéresse à la légalité d'un flux entre un *principal* p et un objet o. À chaque entité est associé un *label*, qui est un ensemble de *tags* de confidentialité et un ensemble de *tags* d'intégrité. Les *tags* sont des étiquettes sans aucune signification particulière mais dotés d'un ordre partiel qui permet de les comparer. Pour que p puisse lire o, il faut respecter la *règle de confidentialité*, à savoir $S_o \subseteq S_p$, avec, pour toute entité u, S_u l'ensemble de ses *tags* de confidentialité. Pour que p puisse écrire dans o, il faut respecter la *règle d'intégrité*, à savoir que $I_o \subseteq I_p$, avec, pour toute entité u, I_u l'ensemble de ses *tags* d'intégrité. Ces règles suivent les principes de Bell et LaPadula, respectivement le *no read up* et le *no write down*.

Dans certains cas, il peut être nécessaire de vouloir déclassifier (retirer un *tag*) ou de classifier (ajouter un *tag*) une information. Pour ce faire, un *principal* p a besoin de *capabilities*. Pour classifier (respectivement déclassifier) un objet o avec le *tag* a, p a besoin de la *capabilities* a^+ (respectivement a^-). L'ensemble des *capabilities* est noté $C_p = C_p^+ \cup C_p^-$ (l'union des *capabilities* de classification et de déclassification). Pour que p puisse changer le *label* L_1 d'un objet x en L_2, il faut que $L_2 \setminus L_1 \subseteq C_p^+$ et que $L_1 - L_2 \subseteq C_p^-$.

Ainsi, si un objet o possède le *tag* de confidentialité s et le *tag* d'intégrité i pour *label*, alors le *principal* p doit posséder le *tag* s ou supérieur (règle de confidentialité) pour lire et le *tag* i ou inférieur (règle d'intégrité) pour écrire dans o.

3.3.2 Design et implémentation

Design Dans Laminar, les *principals* sont les *threads* gérés par le noyau et les objets sont les éléments du système de fichiers (fichier, *socket*, etc.). Tous les *threads* possèdent

donc leur propre *label*. À chaque interaction avec un objet du système, LAMINAR vérifie le respect du DIFC et des règles présentées dans la section précédente. Même si le *thread* est bien l'acteur principal du système, la notion de processus n'est pas absente du *design* de LAMINAR. En effet, LAMINAR vérifie que tous les *threads* d'un même processus possèdent tous le même *label*, sinon il lève une erreur. La seule exception à cette règle concerne le *processus* de la JVM de LAMINAR. Il s'agit du seul processus dont les *threads* peuvent avoir des *labels* différents.

À sa création, un processus hérite des *tags* et *capabilities* de son père. Il peut en fait hériter d'un sous-ensemble de ces derniers. Il existe deux autres moyens d'acquérir de nouvelles *capabilities*. Lorsqu'un processus crée un nouveau *tag*, il obtient les *capabilities* associées [1]. Deux processus peuvent s'échanger des *capabilities* grâce à des mécanismes de connexion *ad hoc*. Dans LAMINAR, seules les *capabilities* sont persistantes [2]. Un *thread* peut changer son *label* en fonction de ses *capabilities*, créer un fichier ou un dossier avec un *label* donné, etc.

Par défaut, le *thread* principal de l'application Java possède un *label* vide. La seule façon qu'il ait d'interagir avec des données associées à un *label* non vide est d'entrer dans une « région sécurisée ». Il s'agit d'une portion de code à laquelle le développeur Java assigne un *label* et des *capabilities* à l'aide d'un nouveau mot clef (`secure`). Quand la JVM entre dans une région sécurisée, elle demande au système de mettre à jour les *labels* et *capabilities* de son *thread* courant. Quand elle sort de la région sécurisée, elle retrouve son état précédent [3].

LAMINAR assigne aussi des *labels* aux objets Java [4]. Par défaut, un objet est créé avec un *label* vide. Quand il est créé dans une zone sécurisée, il possède le *label* de la zone. Il est aussi possible, en usant des *capabilities* de la zone, d'augmenter ou de diminuer son *label*.

Implémentation OS La partie OS de LAMINAR est implémentée sous la forme d'un module de sécurité Linux. Une partie de cette implémentation est réalisée grâce à l'interface LSM [WCS+02], comme le fait Blare. Cela permet à LAMINAR d'intercepter les accès de type *file* ou *inode* pour vérifier que le *label* du *thread* courant est compatible avec le *label* de la ressource demandée. Le *label* d'un *inode* protège le contenu et les meta-données d'un objet à l'exception de son nom et du *label* lui-même, qui sont eux protégés par le *label* du dossier parent. Les *labels* d'un fichier sont par ailleurs stockés dans les attributs étendus des fichiers, tout comme dans Blare.

L'autre partie de LAMINAR liée au noyau est la modification de ce dernier afin de lui ajouter un jeu d'appels système permettant par exemple à un *thread* de créer un nouveau *tag*, d'ajouter un *label* (confidentialité ou intégrité) au *thread* courant, d'abandonner une *capabilities*, etc. Ces appels système sont utilisés par les processus pour mettre à jour leur *label* courant, par exemple s'ils veulent écrire dans un fichier avec une intégrité plus faible. Bien entendu, pour pouvoir faire cela, ils ont besoin de posséder la bonne *capabilitie* et d'être modifiés pour tirer parti de toutes les fonctionnalités de LAMINAR.

1. Pour un *tag* t, on obtient t^+ et t^-
2. Elles sont stockées dans un fichier de configuration propre à chaque utilisateur
3. Les zones sécurisées sont réentrantes.
4. Et pas aux types primitifs ou aux adresses mémoires des objets, seulement aux objets dans le tas.

3.3.3 Implémentation Java

Le deuxième composant de LAMINAR est une modification de Jikes RVM [AAB+05]. LAMINAR rajoute une API particulière qui comprend, entre autres, des classes pour les *labels*, *tags* et *capabilities*, mais aussi des fonctions qui permettent de modifier les *capabilities* (abandon uniquement) du *thread* courant. Une modification du *label* passe forcément par une zone sécurisée. LAMINAR exécute une compilation à la volée du *bytecode* et c'est lors de cette dernière qu'elle va ajouter des barrières. Ces dernières sont des vérifications qui sont donc insérées automatiquement à l'exécution.

Le premier rôle de LAMINAR au niveau Java est de mettre à jour le *label* du *thread* courant de l'application Java exécutée, à l'entrée et la sortie d'une zone sécurisée. Ces dernières ne sont en effet pas visible par l'OS et c'est donc à la JVM de lui notifier ses changements de *label*. Pour savoir si un *thread* peut entrer dans une zone sécurisée de *label* de confidentialité S_R, de *label* d'intégrité I_R et de *capabilities* C_P, LAMINAR vérifie si $S_R \subseteq (C_P^+ \cup S_P)$ et si $I_R \subseteq (C_P^+ \cup I_P)$, avec S_P et I_P les *labels* parents. À l'entrée et à la sortie d'une zone sécurisée, le compilateur va donc ajouter une barrière pour mettre à jour le *label*.

Le second rôle de LAMINAR au niveau Java est d'assurer le DIFC au sein des objets Java. Pour cela, LAMINAR impose quelques règles pour les variables locales (de types primitifs, entre autres) et les variables statiques. Ainsi, une variable locale qui subit une écriture dans une zone sécurisée n'est plus lisible une fois sortie de celle-ci si elle possède des *tags* de confidentialité. De même, une variable locale qui a subit une écriture avant une zone sécurisée n'est pas lisible à l'intérieur de celle-ci si elle possède des *tags* d'intégrité. Pour ce qui est des variables statiques, une région sécurisée avec *tags* d'intégrité (respectivement confidentialité) empêche l'écriture (respectivement la lecture) de variable statique. Ces vérifications sont faites par une analyse statique (lors de la compilation JIT [5]) de flux de données qui ont lieu en plus de la vérification du DIFC.

À cela s'ajoute la vérification portant sur les objets en eux-mêmes. Dans une zone sécurisée, les *labels* d'un objet sont assignés à son allocation via une barrière. Lors d'un accès en lecture ou en écriture, LAMINAR s'assure du respect des règles DIFC en insérant des barrière. De la même façon, à l'extérieur d'une zone sécurisée, une barrière vérifie lors d'un accès que la variable possède un *label* vide.

3.3.4 Évaluations

On évalue habituellement un IDS selon deux critères majeurs : son *overhead* et son efficacité. L'efficacité d'un IDS est évaluée en fonction de sa fiabilité (pas de faux négatif) et de sa pertinence (pas de faux positif). Les auteurs de LAMINAR se concentrent sur une évaluation de son *overhead* seulement.

Sans aucune région sécurisée, les performances Java sont tout de même dégradées (de 17% en moyenne) à cause des barrières servant à vérifier les accès en lecture et écriture d'un objet, par exemple. Du côté de l'OS, on observe un *overhead* moyen de 8% avec `lmbench` [6]. C'est équivalent aux *overheads* constatés dans d'autres modules de sécurité Linux.

Pour calculer un second *overhead* intégrant des régions sécurisées, quatre applications Java ont été modifiées pour implémenter une politique de sécurité cohérente avec leurs

5. *Just In Time.*
6. `lmbench` est une suite d'outil portable réalisant des micro-*benchmarks* pour UNIX/POSIX.

besoins. L'importance des modifications nécessaires n'est pas mise en valeur et les auteurs se concentrent sur l'impact sur la performance et ce dernier est grandement variable, allant de moins de 1% dans le meilleur des cas à 56% dans le pire.

3.3.5 Conclusion

Par plusieurs aspects, LAMINAR se rapproche beaucoup de kBlare et jBlare. Il est basé sur du contrôle de flux et est composé d'un moniteur implémenté au niveau OS et d'un moniteur implémenté au niveau JVM. Il assure la coopération entre ces deux niveaux. La principale différence est que, dans LAMINAR, les *labels* ne se propagent pas entre conteneurs d'information. Dans le cas de Blare, la difficulté est de déterminer précisément quelle variable va recevoir l'information ou, inversement, de quelle variable vient l'information afin de réaliser la propagation des *tags*. Ces problématiques sont absentes de LAMINAR, qui par ailleurs impose plusieurs contraintes au développeur Java désirant l'utiliser (contraintes sur les variables locales et statiques, obligation de modifier son code, etc.). À l'inverse, nous souhaitons faire en sorte de limiter au maximum, dans le cadre de la coopération entre kBlare et jBlare, les modifications à effectuer sur les programmes Java et, si possible, les éviter totalement. Nous ne souhaitons pas non plus implémenter de nouveaux appels système pour réaliser notre mécanisme de coopération.

Troisième partie
Formalisation

Blare est un projet de recherche de l'équipe CIDre. Il s'agit d'un modèle expliquant comment faire de la détection d'intrusion avec du suivi de flux. Il reste très générique et doit être précisé pour être adapté aux différents niveaux de détections. Pour l'heure, Blare a été adapté au niveau OS (kBlare et AndroBlare) et au niveau langage (jBlare). Nous présentons d'abord les modèles préexistants que nous décrivons sous la forme d'une sémantique formelle. Ce travail constitue une première contribution.

4 Modèles existants

Nous présentons dans un premier temps Blare, puis ses adaptations au niveau OS (kBlare) et applicatif (jBlare).

4.1 Le modèle générique Blare

Blare considère des conteneurs, qui peuvent être de différents types selon l'implémentation considérée. Plus particulièrement, Blare s'intéresse aux informations que contiennent ces conteneurs. Ces informations sont reconnues grâce à des étiquettes qui n'ont aucun sens particulier à l'extérieur du modèle. Par exemple, si on considère le fichier `/etc/passwd` comme un conteneur, on peut étiqueter son contenu (les mots de passe des utilisateur) avec `1`. On peut aussi étiqueter le contenu d'un autre fichier `/home/tim-cook/iphone6` avec `2`.

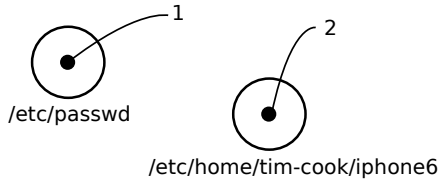


FIGURE 2 – Informations des conteneurs

Pour assurer le suivi de flux d'information, les auteurs de Blare ont attaché à chaque conteneur un *tag* d'information (noté *itag*), qui décrira pour un instant t son contenu, sous la forme de l'ensemble des étiquettes des informations qu'il contient.

Définition 1. *On note $\mathcal{C}$ l'ensemble des conteneurs du modèle et $\mathcal{E}$ l'ensemble de ses étiquettes. Alors, $itag : \mathcal{C} \to \mathcal{P}(\mathcal{E})$ est la fonction qui associe pour chaque conteneur l'ensemble des étiquettes décrivant son contenu.*

Ainsi, dans l'exemple précédent, on peut exprimer le contenu des différents fichiers grâce à la fonction *itag* : $itag(\texttt{/etc/passwd}) = \{\texttt{1}\}$ et $itag(\texttt{/home/tim-cook/iphone6}) = \{\texttt{2}\}$. Ce système est résumé dans la Figure 2.

Pour tout élément c de $\mathcal{C}$, $itag(c)$ évolue en fonction des flux observés. Pour les besoins de ce stage, nous avons décidé de réécrire formellement le modèle Blare en utilisant une grammaire pour décrire les flux interceptés par un moniteur et une sémantique pour décrire son fonctionnement.

Les grammaires que nous proposons pour décrire les flux observés par les différents moniteurs sont de la forme suivante :

$$\langle \mathit{Trace} \rangle \ ::= \ \langle \mathit{Flow} \rangle \, ; \, \langle \mathit{Trace} \rangle \mid \epsilon$$

$$\begin{aligned} \langle \mathit{Flow} \rangle \ ::= \ & \langle \mathit{Flow1} \rangle \\ \mid \ & \langle \mathit{Flow2} \rangle \\ \mid \ & \ldots \end{aligned}$$

Une *Trace* est une succession de *Flow* et un *Flow* peut être de plusieurs types (soit de type *Flow1*, soit de type *Flow2*, etc.). Cette grammaire permet de décrire les flux observés par le moniteur. À cette grammaire, on associe une sémantique, qui sera constituée de règles de la forme suivante :

$$\mathit{Flow}, \Gamma \Rightarrow \Gamma'$$

Γ est l'état du système avant que le *Flow* n'ait lieu et Γ' son état après. La définition de l'état du système dépendra de l'implémentation considérée mais contiendra forcément la fonction *itag*. Peu importe la sémantique définie, elle possède une règle de transitivité de la forme :

$$\frac{\mathit{Flow}, \Gamma \Rightarrow \Gamma' \qquad \mathit{Trace}, \Gamma' \Rightarrow \Gamma''}{\mathit{Flow} \ ; \ \mathit{Trace}, \Gamma \Rightarrow \Gamma''} \ \text{Trans}$$

4.1.1 Exemple

Il est possible d'illustrer ce formalisme avec un système très simple, dans lequel $\mathcal{C} = \{c_1, c_2\}$, $\mathcal{E} = \mathbb{N}$[7] et Γ se limite à la fonction *itag*. Dans ce système, on ne peut observer qu'un seul flux, d'un élément de $\mathcal{C}$ à un autre, le flux `concat` qui ajoute le contenu d'un premier conteneur à celui d'un second. La grammaire décrivant les traces observables peut être écrite comme suit :

$$\begin{aligned}
\langle \mathit{Trace} \rangle \ &::= \ \langle \mathit{Flow} \rangle \, ; \, \langle \mathit{Trace} \rangle \\
&\mid \ \epsilon \\
\langle \mathit{Flow} \rangle \ &::= \ \langle \mathit{Concat} \rangle \\
\langle \mathit{Concat} \rangle \ &::= \ c_i \xrightarrow{\mathit{concat}} c_j
\end{aligned}$$

Grâce à cela, plutôt que d'écrire « trois flux surviennent, un premier flux `concat` de f vers g, un second de g vers h et un troisième de h vers i », on écrira :

$$f \xrightarrow{\mathit{concat}} g \ ; \ g \xrightarrow{\mathit{concat}} h \ ; \ h \xrightarrow{\mathit{concat}} i$$

Et pour décrire l'action du flux `concat`, on lui associe la règle de sémantique suivante :

$$\frac{}{c_i \xrightarrow{\mathit{concat}} c_j, \mathit{itag} \Rightarrow \mathit{itag}[c_j \rightarrow \mathit{itag}(c_i) \cup \mathit{itag}(c_j)]} \text{ Concat}$$

Cette règle exprime sans ambiguïté que lorsque l'on observe un flux `concat` d'un conteneur c_i vers un conteneur c_j, alors on met à jour la fonction *itag* avec l'union de $\mathit{itag}(c_j)$ et de $\mathit{itag}(c_i)$. Par exemple :

$$\frac{}{c_1 \xrightarrow{\mathit{concat}} c_2, \{c_1 \rightarrow \{1\}, c_2 \rightarrow \{2\}\} \Rightarrow \{c_1 \rightarrow \{1\}, c_2 \rightarrow \mathbf{\{1,2\}}\}} \text{ Concat}$$

On peut aussi chercher à prouver des propriétés de notre modèle grâce à cette sémantique. Par exemple, on peut vouloir prouver qu'après un flux concat de p vers q suivi d'un flux concat de q vers p, les deux conteneurs partagent le même *tag* d'information.

$$\dfrac{\text{Concat}\ \dfrac{}{p \xrightarrow{\mathit{concat}} q, \mathit{it} \Rightarrow \mathit{it}'} \qquad \dfrac{}{q \xrightarrow{\mathit{concat}} p, \mathit{it}' \Rightarrow \mathit{it}_f}\ \text{Concat}}{p \xrightarrow{\mathit{concat}} q \ ; \ q \xrightarrow{\mathit{concat}} p, \mathit{it} \Rightarrow \mathit{it}_f}\ \text{Trans}$$

Avec :
- $\mathit{it} = \{p \rightarrow \mathit{it}_p, q \rightarrow \mathit{it}_q\}$;
- $\mathit{it}' = \{p \rightarrow \mathit{it}_p, q \rightarrow \mathit{it}_p \cup \mathit{it}_q\}$;
- $\mathit{it}_f = \{p \rightarrow \mathit{it}_p \cup \mathit{it}_p, q \rightarrow \mathit{it}_p \cup \mathit{it}_q\}$.

On prouve bien que $\mathit{it}_f(p) = \mathit{it}_f(q)$.

7. Dans toutes les implémentations de Blare, on est dans ce cas où $\mathcal{E} = \mathbb{N}$.

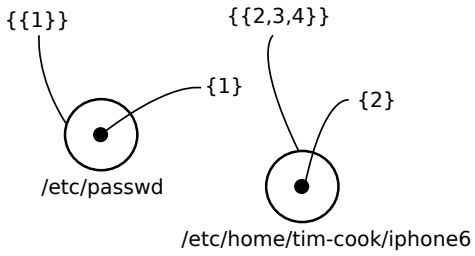


FIGURE 3 – Politique de sécurité

4.2 Politique de sécurité

Pour l'heure, seul le suivi de flux d'information a été évoqué, or les auteurs de Blare proposent d'utiliser ce suivi pour faire de la détection d'intrusion. Pour ce faire, ils attachent un deuxième *tag* aux conteneurs : le *tag* de politique de sécurité. Ce *tag* permet de spécifier quelles sont les informations pouvant cohabiter à l'intérieur du conteneur et se présente comme un ensemble d'ensemble d'étiquettes. La Figure 3 est une représentation graphique du système précédent constitué de deux fichiers auxquels sont attachés un *itag* et un *ptag*.

Définition 2. *On définit* $ptag : \mathcal{C} \to \mathcal{P}(\mathcal{P}(\mathcal{E}))$ *la fonction qui à chaque conteneur* c *associe son tag de politique de sécurité. Pour un conteneur* c*, si* $ptag(c) = \emptyset$*, alors* c *peut contenir n'importe quelle formation légalement.* A contrario*, si* $ptag(c) = \Sigma$*, alors il faut vérifier qu'il existe* $e \in \Sigma$ *tel que* $itag(c) \subseteq e$*.*

Concrètement, si $ptag(c) = \{\{1, 2\}, \{2, 3\}\}$, alors par exemple (et de manière non exhaustive) :

- $itag(c) = \{1\}$ est légal, car $\{1, 2\} \in ptag(c)$;
- $itag(c) = \{2\}$ est légal, car $\{1, 2\} \in ptag(c)$ et $\{2, 3\} \in ptag(c)$;
- $itag(c) = \{3\}$ est légal, car $\{2, 3\} \in ptag(c)$;
- $itag(c) = \{1, 3\}$ est illégal, car il n'existe pas de sur ensemble de $itag(c)$ dans $ptag(c)$.

Dans l'exemple des fichiers déjà présentés, si les informations concernant le futur *iPhone 6* sont toutes étiquetées `2`, `3` ou `4`, alors un administrateur pourra définir $ptag(p) = \{\{2, 3, 4\}\}$ comme c'est le cas sur la Figure 3.

4.3 Implémentation historique

Le premier raffinement de Blare a été étudié pour permettre un suivi de flux et une détection d'intrusion au niveau de l'OS. kBlare est un HIDS, pour *Host IDS*, un détecteur qui se base sur des données fournies par le système d'exploitation.

Dans kBlare, il existe plusieurs types de conteneurs : les conteneurs persistants comme les fichiers, les conteneurs volatiles comme les pages mémoires et les processus qui sont des conteneurs volatiles un peu particulier étant donné qu'ils exécutent du code.

Définition 3. *On note* $\mathcal{CP}$ *l'ensemble des conteneurs persistants,* $\mathcal{CV}$ *l'ensemble des conteurs volatiles et* $\mathcal{P}$ *l'ensemble des processus. Dans kBlare,* $\mathcal{C} = \mathcal{CP} \cup \mathcal{CV} \cup \mathcal{P}$*.*

Les flux que va traquer kBlare sont les flux engendrés par les processus. La Figure 4 montre l'action du moniteur. Le système seul a connaissance des conteneurs et des flux.

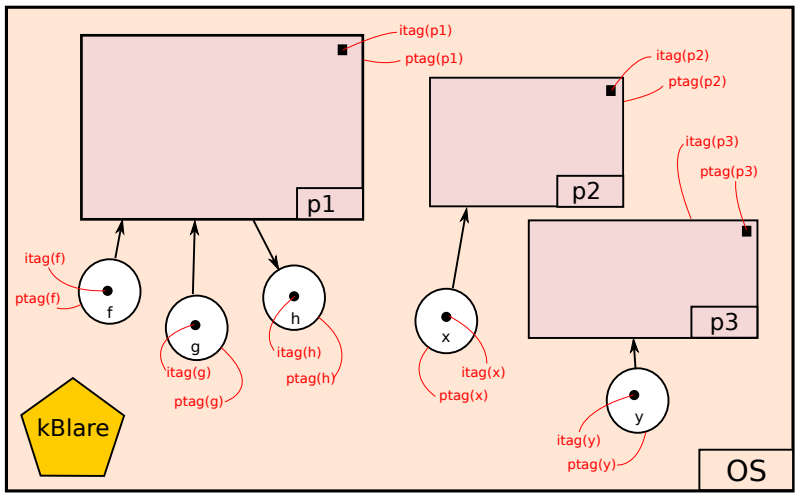


FIGURE 4 – Apports de kBlare (en rouge)

KBlare, quant à lui, attachent des méta-informations aux conteneurs pour faire du suivi de flux.

En réalité, kBlare n'attache pas deux mais trois *tags* aux différents conteneurs. Le *tag* de politique d'exécution permet de spécifier la politique de sécurité des processus créés en exécutant le code présents dans les conteneurs.

Définition 4. *On note* $xptag : \mathcal{C} \to \mathcal{P}(\mathcal{E})$ *la fonction qui associe pour chaque conteneur son* tag *de politique d'exécution.*

Pour connaître l'origine du code exécuté par un processus, kBlare différencie les données des codes exécutés. Les données sont les contenus des fichiers et dans le cadre d'un fichier binaire elles deviennent le code exécuté quand le fichier est chargé en mémoire par un processus comme code exécutable.

Définition 5. *Soit* $\mathcal{I}$ *l'ensemble des étiquettes associées aux données et* $\mathcal{X}$ *l'ensemble des étiquettes associées aux codes exécutés. Dans kBlare,* $\mathcal{E} = \mathcal{I} \cup \mathcal{X}$*, avec* $\mathcal{I} = \mathbb{N}^+$ *et* $\mathcal{X} = \mathbb{N}^-$*.*

On dispose désormais de tous les éléments nécessaires pour décrire formellement le modèle de kBlare.

4.3.1 Grammaire des flux

Il existe beaucoup de flux différents dans le noyau Linux, mais on peut les regrouper en cinq grandes familles. Nous simplifions donc le modèle pour ne prendre en compte que ces cinq types de flux. De plus, nous nous limitons à deux types de conteneurs : les fichiers et les processus. Ces simplifications permettent d'expliquer le modèle de kBlare plus facilement. Toutefois, le principe peut facilement être étendu pour décrire kBlare de manière exhaustive. La grammaire suivante décrit les principaux flux que kBlare peut observer :

$$\langle Trace \rangle \ ::= \ \langle Flow \rangle \ \langle Trace \rangle \mid \epsilon$$

$$\langle Flow \rangle \ ::= \ \langle Fork \rangle \mid \langle Exec \rangle \mid \langle Read \rangle \mid \langle Write \rangle \mid \langle Append \rangle$$

$$\langle Fork \rangle \ ::= \ \langle Process \rangle \xrightarrow{fork} \langle Process \rangle$$

$$\langle Exec \rangle \ ::= \ \langle File \rangle \xrightarrow{exec} \langle Process \rangle$$

$$\langle Read \rangle \ ::= \ \langle File \rangle \xrightarrow{read} \langle Process \rangle$$

$$\langle Write \rangle \ ::= \ \langle Process \rangle \xrightarrow{write} \langle File \rangle$$

$$\langle Append \rangle \ ::= \ \langle Process \rangle \xrightarrow{append} \langle File \rangle$$

Les non-terminaux *File* et *Process* se dérivent en les identifiants que l'on décide de donner respectivement aux fichiers et aux processus.

Un flux *Fork* d'un processus p vers un processus q décrit la création d'un processus fils q identique à son processus père p. Un flux *Exec* d'un fichier f vers un processus p est le chargement d'un fichier f par un processus p en vue de l'exécuter. Un flux *Read* d'un fichier f vers un processus p est la lecture (totale ou partielle) de f par p. Un flux *Write* d'un processus p vers un fichier f est l'écrasement du contenu de f par p. Enfin, un flux *Append* d'un processus p vers un fichier f est la concaténation de données connues de f à la fin du fichier f.

4.3.2 Sémantique de kBlare

Définition 6. *L'état* Γ *du système est un triplet formé des trois fonctions décrites précédemment et décrivant l'état du système.* $(itag, ptag, xptag)$ *est un état valide du système.*

Définition 7. *Soit* $f' = f[c \to v]$*, alors* $f'(x) = \begin{cases} v & si\ x = c \\ f(x) & sinon \end{cases}$.

Il est désormais possible d'écrire les règles de la sémantique décrivant très précisément la façon dont kBlare propage les différents *tags*.

Fork Le processus fils est la copie exacte de son père et il hérite de ses *tags*.

$$\frac{it' = it[q \to it(p)] \qquad pt' = pt[q \to pt(p)] \qquad xpt' = xpt[q \to xpt(p)]}{p \xrightarrow{fork} q, (itag, ptag, xptag) \Rightarrow (it', pt', xpt')} \text{ Fork}$$

Exec Lors d'un `exec`, le processus voit sa mémoire réinitialisée et son nouveau code chargé. Le modèle de kBlare prévoit de garder une trace du code exécuté par un processus et c'est pour cela que les auteurs introduisent un ensemble d'étiquettes particulières pour le décrire.

Définition 8. *On définit* $Run : \mathcal{I} \to \mathcal{X}$ *une bijection qui pour tout étiquette d'une donnée associe une étiquette d'un code exécuté. Dans kBlare,* $Run(x) = -x$.

Si un fichier exécutable est étiqueté avec $s \in \mathcal{I}$, alors le processus l'exécutant sera étiqueté avec $Run(s) \in \mathcal{X}$. Lorsque ce processus fera un accès en écriture sur un fichier, $Run(s)$ sera propagé ce qui à kBlare de connaître la provenance des informations.

$$\frac{itag' = itag[p \rightarrow \bigcup_{k \in itag(f) \setminus \mathcal{X}} Run(k)] \qquad ptag' = ptag[p \rightarrow xptag(f)]}{p \xrightarrow{exec} q, (itag, ptag, xptag) \Rightarrow (itag', ptag', xptag)} \text{ Exec}$$

Cette particularité du modèle de kBlare est un premier pas vers la possibilité de faire de la déclassification. En effet, on fera la différence entre un *tag* $\{Run(p1), f1, f2\}$ et $\{Run(p2), f2, f2\}$. Si $p1$ correspond au code d'un algorithme de hachage et $p2$ un algorithme de compression, il est normal que les deux situations ne soient pas traitée de la même manière (on autorisera le haché mais pas le compressé, etc.).

Read, write et append Lors d'un flux de lecture d'un conteneur f vers un processus p, kBlare initialise $itag(p)$ avec la valeur de $itag(f)$ dont on retire les éléments de $\mathcal{X}$. En effet, les éléments de $\mathcal{X}$ permettent de savoir quel code est à l'origine de l'écriture des informations (pour un fichier) et quel code est exécuté (pour un processus). Garder en mémoire les éléments de $\mathcal{X}$ d'un fichier quand il est lu par un processus corromprai leur signification : on ne saurait plus quel est le code exécuté par le processus. Lors d'un flux en écriture (écrasement) d'un processus p vers un conteneur f, kBlare efface l'ancienne valeur de $itag(f)$ et la remplace par $itag(p)$. Dans le cas d'un flux de concaténation (*append*), on effectue la même opération que pour un flux de lecture, mais cette fois en mettant à jour $itag(f)$ et non $itag(p)$.

$$\frac{itag' = itag[p \rightarrow (itag(f) \setminus \mathcal{X}) \cup itag(p)]}{f \xrightarrow{read} p, (itag, ptag, xptag) \Rightarrow (itag', ptag, xptag)} \text{ Read}$$

$$\frac{itag' = itag[f \rightarrow itag(p)]}{p \xrightarrow{write} f, (itag, ptag, xptag) \Rightarrow (itag', ptag, xptag)} \text{ Write}$$

$$\frac{itag' = itag[f \rightarrow itag(f) \cup itag(p)]}{p \xrightarrow{append} f, (itag, ptag, xptag) \Rightarrow (itag', ptag, xptag)} \text{ Append}$$

Ces deux dernières règles sont la cause de la sur approximation de kBlare. En effet, rien ne garantit, quand un processus effectue une écriture, que les données réellement écrites dépendent de toutes les données lues.

5 Coopération

Nous voulons réduire ou supprimer la sur approximation de kBlare. Pour cela, nous voulons établir un mécanisme de coopération entre lui et un moniteur au niveau langage.

5.1 Notion d'applications tierces

Blare est un modèle abstrait qui peut être appliqué concrètement dans plusieurs situations. Si kBlare, l'implémentation principale et historique pour Linux, et AndroBlare, son portage sur Android, se placent au niveau système d'exploitation et surveillent les flux engendrés par les appels systèmes, il est aussi possible d'adapter le modèle pour obtenir un moniteur de flux *Application-based*, c'est à dire qui surveille une application en particulier.

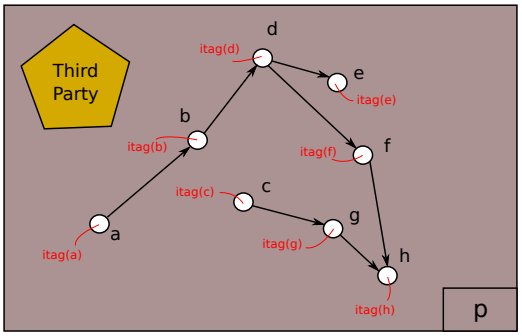


FIGURE 5 – Apport d'une application tierce (en rouge)

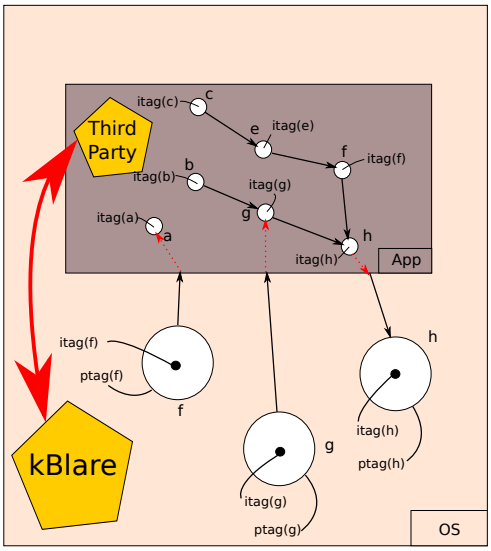


FIGURE 6 – Apport de la coopération (en rouge)

La Figure 5 montre l'apport d'une implémentation de Blare sur un processus p, à savoir attacher des *tags* aux variables pour effectuer finalement le suivi de flux.

Les raisons qui nous poussent à envisager une coopération entre kBlare et jBlare ont déjà été explicitées : elle permet d'améliorer le suivi de flux de kBlare et d'affecter automatiquement les étiquettes aux variables surveillées par jBlare. La Figure 6 résume le fonctionnement de la coopération entre kBlare et une application tierce. On parle de kBlare comme d'une application maîtresse et de l'implémentation de Blare au niveau application comme d'une application tierce parce que la détection d'intrusion (but principal du projet Blare) reste du ressort de kBlare, qui utilise un autre composant pour améliorer ses résultats.

5.2 Politique de coopération

En terme de sécurité, on parle de TCB [8] pour l'ensemble des composants, matériels et logiciels, qui sont réputés de confiance. Les applications présentes dans la TCB ne sont pas surveillées et participent à la protection du système. Une TCB se doit d'être la plus petite possible, car en cas de compromission, la politique de sécurité peut être contournée (totalement ou en partie selon les cas). Une application tierce (en mesure de coopérer avec kBlare) fera parti de la TCB : elle aura des droits supplémentaires et sa compromission entraînerait un possible contournement de la politique de sécurité (elle pourrait « mentir »). Tous les processus ne peuvent donc pas coopérer avec kBlare.

Nous avons décidé que la coopération n'avait pas à être établie par défaut (kBlare reconnaît l'application et lui fait confiance) mais sur demande (l'application demande à coopérer et kBlare décide d'accepter ou non). Nous voulions un mécanisme de coopération le plus générique possible et nous avons remarqué trois choses :

1. Rien ne dit qu'une application ait à coopérer à chaque fois, elle doit pouvoir initier la coopération ;
2. Devoir choisir pour tous les processus si oui ou non la coopération est possible apporterait très certainement un fort *overhead*.

Il faut donc déterminer dans quelles conditions une coopération est envisageable. Nous voulons un mécanisme générique, c'est à dire qui ne soit pas spécifique à jBlare. Notre problématique est de déterminer quel est le code de confiance. Le modèle de kBlare permet de déterminer quel est le code exécuté par un processus (grâce aux étiquettes appartenant à $\mathcal{X}$, soit concrètement les entiers négatifs des *tags* d'information).

Définition 9. *On définit $\mathbb{P}_{co} \subseteq \mathcal{P}(\mathcal{P}(\mathcal{X}))$ la politique de coopération. Un processus p peut coopérer avec kBlare si $\exists co \in \mathbb{P}_{co}$ tel que $co \subseteq itag(p)$.*

La politique de coopération permet d'indiquer quel est le code minimum à exécuter pour être considéré comme « de confiance ».

L'exemple suivant illustre le fonctionnement de la politique de coopération $\mathbb{P}_{co}$, le plus simple est de l'illustrer avec un exemple. Imaginons que nous ayons développé deux applications tierces implémentant le modèle de Blare au niveau langage : un interpréteur Ruby modifié (un exécutable `ruby`) et une JVM modifiée (en fait un exécutable `java` qui charge

8. Trusting Computing Base.

ensuite la bibliothèque partagée `libjvm.so` pour charger une JVM). L'administrateur définit les *tags* d'information des différents fichiers de la façon suivante : $itag(\texttt{ruby}) = \{r\}$, $itag(\texttt{java}) = \{j\}$ et $itag(\texttt{libjvm.so}) = \{s\}$.

En définissant $\mathbb{P}_{co} = \{\{Run(r)\}, \{Run(j), Run(s)\}\}$, on met en place notre politique de coopération. En effet, lorsque l'exécutable `ruby` sera chargé par un processus p (via un flux de type *exec*), on aura $itag(p) = \{Run(r)\}$ et dans ce cas, la coopération sera possible. Toutefois, elle ne sera effective que si l'application en fait la demande explicite. De la même façon, pour qu'une application java q puisse coopérer avec kBlare, il faut qu'elle ait été lancée avec l'exécutable `java` et qu'elle utilise la bonne JVM (le bon fichier `libjvm.so`). Dans ce cas là, on aura $itag(q) = \{Run(j), Run(s)\}$ et le processus pourra coopérer en accord avec $\mathbb{P}_{co}$.

5.3 *Tag* de coopération

Pour permettre la coopération entre kBlare et une application tierce, nous avons ajouté un nouveau *tag* au modèle : le *tag* de coopération. Le but du *tag* de coopération est d'être le point de communication entre les deux moniteurs. En plus de faire évoluer le *tag* d'information de l'application tierce, kBlare va aussi modifier son *tag* de coopération selon les mêmes règles. L'application tierce, grâce à un accès en lecture et en écriture sur son *tag* de coopération, va pouvoir d'une part le modifier avant ses appels système en écriture (pour qu'il reflète réellement les informations qui vont être écrites et uniquement elles) et d'autre part lire son cotenu après ses appels systèmes en lecture (pour connaître très précisément ce qui a été lu). Le *tag* de coopération remplace donc le *tag* d'information lorsque la coopération est mise en place.

Définition 10. *On rappelle que $\mathcal{P}$ est l'ensemble des processus. On définit $ctag : \mathcal{P} \to \mathcal{P}(\mathcal{E})$ la fonction qui associe à chaque processus p son* tag *de coopération.*

Définition 11. *On note $coop : \mathcal{P} \to \{true, false\}$ la fonction qui à tout processus p associe son état (en coopération avec kBlare ou non).*

La coopération entre kBlare et une application tierce rajoute un flux à prendre en compte : la demande initiale de coopération. À noter que nous n'avons pas prévu « d'arrêt » de la coopération, cette dernière prenant fin *de facto* lorsque le processus se termine et meurt. La grammaire décrivant les flux de kBlare devient :

$$\begin{aligned}
\langle Trace\rangle &::= \langle Flow\rangle\ \langle Trace\rangle \\
&\mid \epsilon \\
\\
\langle Flow\rangle &::= \langle Fork\rangle \\
&\mid \langle Exec\rangle \\
&\mid \langle Read\rangle \\
&\mid \langle Write\rangle \\
&\mid \langle Append\rangle \\
&\mid \langle Cooperation\rangle \\
\\
\langle Fork\rangle &::= \langle Process\rangle \xrightarrow{fork} \langle Process\rangle
\end{aligned}$$

$$\begin{aligned}
\langle Exec\rangle &::= \langle File\rangle \xrightarrow{exec} \langle Process\rangle \\
\\
\langle Read\rangle &::= \langle File\rangle \xrightarrow{read} \langle Process\rangle \\
\\
\langle Write\rangle &::= \langle Process\rangle \xrightarrow{write} \langle File\rangle \\
\\
\langle Append\rangle &::= \langle Process\rangle \xrightarrow{append} \langle File\rangle \\
\\
\langle Cooperation\rangle &::= \downarrow \langle Processus\rangle
\end{aligned}$$

Il faut désormais adapter notre sémantique à cette nouvelle grammaire et définir un nouveau modèle pour l'état du système prenant en compte les fonctions *ctag* et *coop*. À

noter que pour une meilleure lisibilité, nous écrirons par la suite *it* pour *itag*, *ct* pour *ctag*, *pt* pour *ptag*, *xpt* pour *xptag* et *co* pour *coop*.

Fork Le processus fils se trouve être dans le même état de coopération que le processus père.

$$\frac{\forall f \in \Gamma, \exists f' \in \Gamma' \text{ tel que } f' = f[q \to f(p)]}{p \xrightarrow{fork} q, \Gamma \Rightarrow \Gamma'} \text{ Fork}$$

Exec Après un `exec`, le processus n'est pas dans un état de coopération avec kBlare, même si son processus père l'était. En effet, ils ne partagent plus le même code.

$$\frac{it' = it[p \to \bigcup_{k \in itag(f) \backslash \mathcal{X}} Run(k)] \qquad ct' = ct[p \to it'(f)] \qquad co' = pt[p \to false]}{p \xrightarrow{exec} q, (it, ct, pt, xpt, co) \Rightarrow (it', ct, pt', xpt, co)} \text{ Exec}$$

Read Lors d'un accès en écriture, on enrichie le *tag* de coopération du processus en plus de son *tag* d'information.

$$\frac{it' = it[p \to (it(f) \backslash \mathcal{X}) \cup it(p)] \qquad ct' = ct[p \to (it(f) \backslash \mathcal{X}) \cup ct(p)]}{f \xrightarrow{read} p, (it, ct, pt, xpt, co) \Rightarrow (it', ct, pt, xpt, co)} \text{ Read}$$

Write et append La propagation lors d'un flux d'écriture dépend de l'état de coopération : si l'application tierce coopère, alors c'est le *tag* de coopération qui est pris en compte par kBlare. Sinon, le comportement normal prévaut et c'est le *tag* d'information qui est utilisé.

$$\frac{co(p) = false \qquad it' = it[f \to it(p)]}{p \xrightarrow{write} f, (it, ct, pt, xpt, co) \Rightarrow (it', ctpt, xpt, co)} \text{ Write}$$

$$\frac{co(p) = false \qquad it' = it[f \to it(f) \cup it(p)]}{p \xrightarrow{append} f, (it, ct, pt, xpt, co) \Rightarrow (it', ct, pt, xpt, co)} \text{ Append}$$

$$\frac{co(p) = true \qquad it' = it[f \to ct(p)]}{p \xrightarrow{write} f, (it, ct, pt, xpt, co) \Rightarrow (it', ct, pt, xpt, co)} \text{ WriteCoop}$$

$$\frac{co(p) = true \qquad it' = it[f \to it(f) \cup ct(p)]}{p \xrightarrow{append} f, (it, ct, pt, xpt, co) \Rightarrow (it', ct, pt, xpt, co)} \text{ AppendCoop}$$

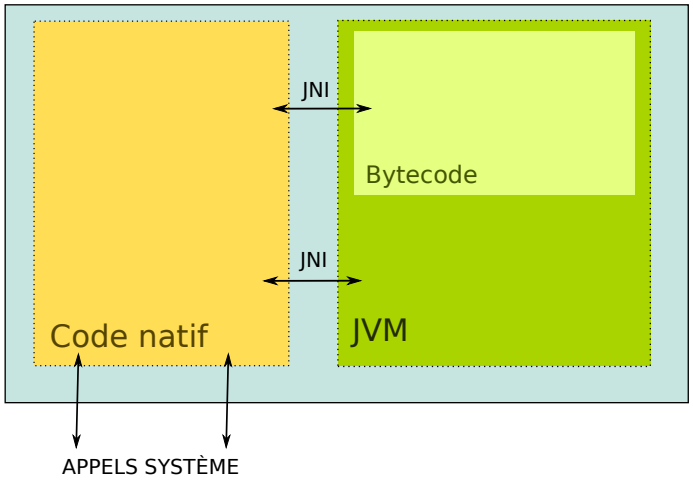


FIGURE 7 – Architecture d'un processus Java

Coopération Pour coopérer, il faut respecter la politique de sécurité (voir Section 4.2).

$$\frac{co' = co[p \to true] \qquad \exists c \in \mathbb{P}_{co} \text{ tel que } c \subseteq it(p)}{\downarrow p, (it, ct, pt, xpt, co) \Rightarrow (it, ct, pt, xpt, co')} \text{ Coop}$$

$$\frac{co' = co[p \to false] \qquad \nexists c \in \mathbb{P}_{co} \text{ tel que } c \subseteq it(p)}{\downarrow p, (it, ct, pt, xpt, co) \Rightarrow (it, ct, pt, xpt, co')} \text{ FailCoop}$$

Avec toutes ces règles, kBlare est prêt à coopérer avec une application tierce. Il faut encore exprimer comment se déroule précisément cette coopération. En effet, actuellement, on a toujours $itag(p) = ctag(p)$.

6 Coopération avec jBlare

En 2011, Mounir ASSAF a travaillé à l'élaboration d'une implémentation de Blare pour le langage Java, nommée jBlare. jBlare n'implémente que le mécanisme de suivi de flux, sans se préoccuper à la politique de sécurité. Plus précisément, jBlare effectue un suivi de flux au niveau du *bytecode*.

6.1 Fonctionnement des applications Java

La Figure 7 résument l'architecture d'un processus exécutant une application Java. La partie code est divisée en deux parties : le code de la JVM, chargé au lancement de l'application et qui contient plus particulièrement l'interpréteur de bytecode, et le code natif, qui provient de librairies partagées codées en C. Une application Java peut en effet utiliser du code compilé grâce à une interface particulière : JNI (pour *Java Native Interface*). C'est par cette même interface que le code compilé peut interagir avec la JVM afin de modifier, par exemple, des objets Java.

Définition 12. *Comme dans kBlare,* $\mathcal{E} = \mathbb{N}$ *dans jBlare.*

La JVM associe à chaque variable et objet du tas du programme un *tag* d'information, nommé aussi label. L'objectif de la coopération est de définir le label d'une variable avec les *tags* d'information des fichiers dont elle possède le contenu. Ainsi, si une application Java lit une ligne du fichier `/etc/passwd` et met son contenu dans la variable `pwd`, on veut $itag($`/etc/passwd`$) = itag($`pwd`$)$.

En pratique, le lien entre la lecture du fichier `/etc/passwd` et `pwd` ne peut pas être fait de façon simple, à cause du fonctionnement du bytecode. Ce dernier ne permet qu'un nombre restreint d'opération (opérations arithmétiques, comparaisons, appels de fonction, etc.) mais en aucun cas des interactions avec le système. Pour réaliser ces actions, l'application utilise des fonctions dites « natives » grâce à l'interface JNI. Or, comme jBlare ne surveille que le bytecode, il n'a aucune idée des flux ayant cours lors de l'exécution de ces fonctions natives. Une suppression totale de la sur approximation est donc hors de portée.

6.2 Stratégie de coopération

Si nous ne pouvons pas prétendre à une coopération parfaite (dans le sens *sans sur approximation*) nous pouvons chercher à nous en approcher en la limitant au maximum. Ainsi, au lieu de $itag($`/etc/passwd`$) = itag($`pwd`$)$, nous allons chercher à avoir $itag($`/etc/passwd`$) \subseteq itag($`pwd`$)$. Pour parvenir à ce résultat, nous avons décidé de limiter la sur approximation à un appel de fonction plutôt qu'au programme en lui-même. C'est à dire que si le processus initie deux flux de lecture lors d'un appel de fonction native, le label du retour de cette fonction sera le mélange des *tags* d'information des deux conteneurs plus.

L'hypothèse forte sur laquelle nous nous basons pour construire notre stratégie de coopération est qu'une fonction native est souvent une opération atomique (obtenir l'inode d'un fichier, écrire dans un fichier, etc.). Avec cette hypothèse, nous pensons que la sur approximation moyenne sera relativement faible. Nous n'avons pas évalué quantitativement cette sur approximation.

Nous avons donc décidé de considérer les *frames* de fonction native comme des conteneurs et de leur associer un *tag* d'information, ainsi que le prévoit le modèle Blare. La particularité est que le *tag* de la frame native la plus haute dans la pile est le *tag* de coopération du programme. Ainsi, lorsque la fonction native effectue un appel système (et donc un flux, du point de vue de kBlare), c'est le *tag* de coopération qui est propagé (comme spécifié dans le mécanisme de coopération), à savoir le *tag* de la *frame*. Les autres *tags*, correspondant aux autres *frames*, sont stockés dans une pile interne à la JVM.

Lorsque jBlare empile une nouvelle *frame* de fonction native dans la pile de *frames*, il lit la valeur de son *ctag* et l'empile afin de le garder en mémoire. Quand jBlare dépile une *frame* native, il dépile en même temps un *tag* d'information de sa pile et rétablit la valeur du *tag* de coopération à sa valeur précédente.

6.3 Formalisation

Les flux observés par kBlare ne sont plus les seuls à devoir être pris en compte par notre modèle : il faut aussi rajouter ceux internes à la JVM.

Définition 13. *On définit $\mathcal{V}$ l'ensemble des variables de tous les programmes Java. Alors $\mathcal{C} = \mathcal{P} \cup \mathcal{CP} \cup \mathcal{CV} \cup \mathcal{V}$.*

$\langle Trace \rangle ::= \langle Flow \rangle \langle Trace \rangle$
$\mid \epsilon$

$\langle Flow \rangle ::= \langle Fork \rangle$
$\mid \langle Exec \rangle$
$\mid \langle Read \rangle$
$\mid \langle Write \rangle$
$\mid \langle Append \rangle$
$\mid \langle Cooperation \rangle$
$\mid \langle Get \rangle$
$\mid \langle Set \rangle$
$\mid \langle CallJava \rangle$
$\mid \langle JavaFlow \rangle$
$\mid \langle RetJava \rangle$
$\mid \langle CallJNI \rangle$
$\mid \langle RetJNI \rangle$

$\langle Fork \rangle ::= \langle Process \rangle \xrightarrow{fork} \langle Process \rangle$

$\langle Exec \rangle ::= \langle File \rangle \xrightarrow{exec} \langle Process \rangle$

$\langle Read \rangle ::= \langle File \rangle \xrightarrow{read} \langle Process \rangle$

$\langle Write \rangle ::= \langle Process \rangle \xrightarrow{write} \langle File \rangle$

$\langle Append \rangle ::= \langle Process \rangle \xrightarrow{append} \langle File \rangle$

$\langle Cooperation \rangle ::= \downarrow \langle Process \rangle$

$\langle CallJava \rangle ::= \langle Process \rangle \xrightarrow{callJava} \langle JavaVar \rangle *$

$\langle RetJava \rangle ::= \langle JavaVar \rangle \xrightarrow{retJava} \langle Process \rangle$

$\langle JavaFlow \rangle ::= \langle JavaVar \rangle \xrightarrow{java} \langle Process \rangle \langle JavaVar \rangle$

$\langle CallJNI \rangle ::= \langle JavaVar \rangle * \xrightarrow{callJNI} \langle Process \rangle$

$\langle RetJNI \rangle ::= \langle Process \rangle \xrightarrow{retJNI} \langle JavaVar \rangle$

$\langle Get \rangle ::= \langle JavaVar \rangle \xrightarrow{get} \langle Process \rangle$

$\langle Set \rangle ::= \langle Process \rangle \xrightarrow{set} \langle JavaVar \rangle$

Cette grammaire décrit tous les flux pouvant impliquer un processus en prenant en compte les possibilités du JNI. Ainsi, le non terminal *CallJava* décrit la possibilité d'appeler du code Java depuis du code natif. Cela se traduit par un flux du processus vers des variables du monde Java passées en argument. Le non terminal *RetJava* décrit un flux ayant lieu à la fin de l'exécution d'une méthode Java appelée depuis du code natif, il traduit un flux d'une variable Java (celle retournée par la méthode) vers le processus. *CallJNI*et *RetJNI* sont les opérations dans le sens inverse (code natif vers code Java). *JavaFlow* sont les différents flux internes à Java, simplifiés au maximum (pas de distinction entre les flux explicites ou implicites, etc.) ; ils sont invisibles pour kBlare mais sont traqués par jBlare. En dernier lieu, *Get* et *Set* traduisent les flux engendrés par la possibilité offerte par JNI à un code natif de récupérer une référence sur un objet Java et de le modifier.

Cette grammaire est très permissive : elle ne fait aucune hypothèse sur l'ordre des flux et accepte des traces illogiques (on peut observer un *RetJava* sans avoir vu un *CallJava*). On ne s'intéressera qu'à des traces globales cohérentes.

Définition 14. *Une trace est cohérente si toutes ses sous-traces sont cohérentes.*

Définition 15. *Une sous-trace t_p d'une trace t est l'ensemble des flux impliquant le processus p.*

Définition 16. *Une sous-trace t_p est cohérente si elle vérifie la grammaire suivante :*

$\langle Trace \rangle ::= \langle Flow \rangle \langle Trace \rangle$
$\mid \epsilon$

$\langle Flow \rangle ::= \langle Fork \rangle$
$\mid \langle Exec \rangle$
$\mid \langle Read \rangle$
$\mid \langle Write \rangle$
$\mid \langle Append \rangle$
$\mid \langle Cooperation \rangle$
$\mid \langle Get \rangle$
$\mid \langle UseJava \rangle$

$\langle Fork \rangle ::= \langle Process \rangle \xrightarrow{fork} \langle Process \rangle$

$\langle Exec \rangle ::= \langle File \rangle \xrightarrow{exec} \langle Process \rangle$

$\langle Read \rangle ::= \langle File \rangle \xrightarrow{read} \langle Process \rangle$

$\langle Write \rangle ::= \langle Process \rangle \xrightarrow{write} \langle File \rangle$

$\langle Append \rangle ::= \langle Process \rangle \xrightarrow{append} \langle File \rangle$

$\langle Cooperation \rangle ::= \downarrow \langle Process \rangle$

$\langle UseJava \rangle ::= \langle CallJava \rangle \quad \langle JavaFlows \rangle \quad \langle RetJava \rangle$

$\langle JavaFlows \rangle ::= \langle JavaFlow \rangle \langle JavaFlows \rangle$
$\mid \epsilon$

$\langle CallJava \rangle ::= \langle Process \rangle \xrightarrow{callJava} \langle JavaVar \rangle *$

$\langle RetJava \rangle ::= \langle JavaVar \rangle \xrightarrow{retJava} \langle Process \rangle$

$\langle JavaFlow \rangle ::= \langle JavaVar \rangle \xrightarrow{java} \langle Process \rangle \quad \langle JavaVar \rangle$

$\langle UseJNI \rangle ::= \langle CallJNI \rangle \langle Flows \rangle \langle RetJNI \rangle$

$\langle CallJNI \rangle ::= \langle JavaVar \rangle * \xrightarrow{callJNI} \langle Process \rangle$

$\langle RetJNI \rangle ::= \langle Process \rangle \xrightarrow{retJNI} \langle JavaVar \rangle$

$\langle Get \rangle ::= \langle JavaVar \rangle \xrightarrow{get} \langle Process \rangle$

$\langle Set \rangle ::= \langle Process \rangle \xrightarrow{set} \langle JavaVar \rangle$

Avec ces grammaires, nous pouvons désormais définir nos règles sémantiques pour exprimer le modèle de cette coopération entre kBlare et jBlare.

Définition 17. *$[A]$ désigne une pile d'éléments de l'ensemble A. Par définition, $a :: R_a \in [A]$ si $a \in A$ et $R_a \in [A]$ et $\epsilon \in [A]$. Par abus de notation, on peut écrire $a_1 :: a_2 :: a_3$ au lieu d'écrire $a_1 :: a_2 :: a_3 :: \epsilon$*

La sémantique que nous présentons n'est pas complète car nous n'introduisons que les règles s'appliquant dans le cas où la coopération avec le processus est effective. Par souci de simplification de l'écriture, nous avons omis d'ajouté la précondition $co(p) = true$. Dans le cas inverse où $co(p) = false$, il n'y a en réalité rien à faire.

Définition 18. *On définit $\mathcal{P}_j$ l'ensemble des processus Java exécuté par jBlare, $\mathcal{P}_j \subseteq \mathcal{P}$*

Définition 19. *On définit $jtag : \mathcal{P}_j \to [\mathcal{P}(\mathcal{E})]$ la fonction qui à tout processus Java exécuté par jBlare associe sa pile de* tag *d'information.*

On rajoute *jtag* dans les fonctions appartenant à l'état Γ du système. De cette façon, on peut écrire précisément les règles d'une sémantique pour la grammaire vue précédemment et donc décrire la coopération de manière formelle.

Règles de kBlare Les règles de la sémantique liée au fonctionnement de kBlare ne changent pas, si ce n'est la nouvelle définition de Γ ($(it, ct, pt, xpt, jt, coop)$ et non plus seulement $(it, ct, pt, xpt, coop)$). La pile des *tags* d'information des processus exécutant jBlare n'intervient que dans le cadre des règles pour JNI.

Appel d'une méthode Java Lorsqu'un code natif fait appel à une méthode Java, les *itag* des arguments sont initialisés avec le *ctag* courant. Lorsqu'une méthode Java appelée depuis du code natif se termine, l'*information tag* de la variable passée en retour est ajoutée au *tag* de coopération du processus.

$$\frac{it' = it[v_1 \rightarrow ct(p), v_2 \rightarrow ct(p), ...]}{p \xrightarrow{CallJava} v_1\ v_2..., (it, ct, pt, xpt, jt, co) \Rightarrow (it', ct, pt, xpt, jt, co)}\ \text{CallJava}$$

$$\frac{ct' = ct[p \rightarrow it(v) \cup ct(p)]}{v \xrightarrow{retJava} p, (it, ct, pt, xpt, jt, co) \Rightarrow (it, ct', pt, xpt, jt, co)}\ \text{RetJava}$$

Flux Java Dans ce modèle, les flux Java sont simplifiés au maximum pour ne garder que des flux directs d'une variable vers une autre, sans se soucier des flux indirects, des exceptions, etc. jBlare gère ces flux et ils seront donc pris en compte dans l'implémentation. Le processus p n'apparaît dans cette règle que pour permettre de savoir à quelle sous-trace il appartient.

$$\frac{it' = it[v_2 \rightarrow it(v_1)]}{v_1 \xrightarrow{java}_p v_2, (it, ct, pt, xpt, jt, co) \Rightarrow (it', ct, pt, xpt, jt, co)}\ \text{Java}$$

Appel de code natif Lorsqu'une méthode Java effectue un appel natif, le *tag* de coopération est sauvegardé dans la pile des $jtags(p)$ avant d'être effacé. Sa nouvelle valeur est l'union de tous les *information tags* passés en paramètres de la fonction. Lorsqu'une fonction native appelée depuis une méthode Java termine son exécution, on associe le *tag* de coopération du processus à la variable de retour.

$$\frac{ct' = ct[p \rightarrow it(v_1) \cup it(v_2) \cup ...] \qquad jt' = jt[p \rightarrow ct(p) :: jt(p)]}{v_1, v_2, ... \xrightarrow{callJNI} p, (it, ct, pt, xpt, jt, co) \Rightarrow (it, ct', pt, xpt, jt', co)}\ \text{CallJNI}$$

$$\frac{jt(p) = ct_p :: jt_p \qquad ct' = ct[p \rightarrow ct_p] \qquad jt' = jt[p \rightarrow jt_p]}{p \xrightarrow{retJNI} v, (it, ct, pt, xpt, jt, co) \Rightarrow (it, ct', pt, xptjt', co)}\ \text{RetJNI}$$

Récupérer une référence Une fonction native peut, à tout moment, récupérer une référence ou une variable du monde Java. Dans ces cas là, on enrichie le *ctag* du processus avec l'*itag* de la variable.

$$\frac{ct' = ct[p \rightarrow ct(p) \cup it(v)]}{v \xrightarrow{get} p, (it, ct, pt, xpt, jt, co) \Rightarrow (it, ct', pt, xpt, jt, co)}\ \text{Get}$$

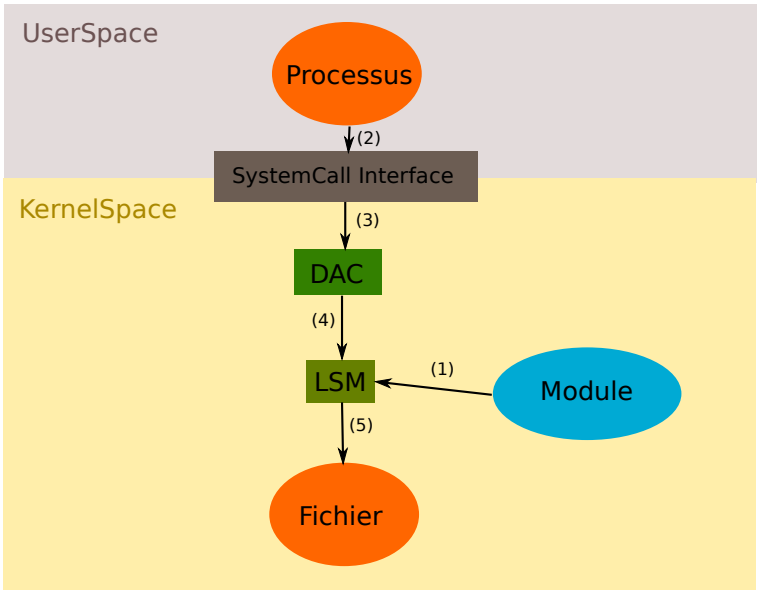


FIGURE 8 – Fonctionnement du *framework* LSM

Modifier un objet du tas Une fonction native peut aussi modifier une référence ou une variable Java grâce à des appels JNI. Dans ce cas, l'*itag* de la variable est remplacé par le *ctag* du processus.

$$\frac{it' = it[v \rightarrow ct(p)]}{p \xrightarrow{set} v, (it, ct, pt, xpt, jt, co) \Rightarrow (it', ct, pt, xpt, jt, co)}\ \text{Set}$$

Quatrième partie
Implémentation

7 kBlare

7.1 Implémentation de CIDre

L'équipe CIDre travaille sur Blare depuis 2001. Son implémentation dans Linux, nommée kBlare, a été réécrite plusieurs fois pour suivre les évolutions du noyau. La dernière version de kBlare se base sur LSM, pour *Linux Security Module*, qui est un *framework* qui permet d'implémenter des modules de contrôles d'accès pour le noyau en plus du DAC utilisé par défaut. LSM repose sur un système de *hooks* (appels de fonctions du module) placés aux endroits stratégiques du code pour limiter au maximum les modifications apportées au kernel. Si kBlare ne fait pas à strictement parler du contrôle d'accès, les *hooks* sont placés avant les appels systèmes pour faire du suivi de flux.

La Figure 8 résume le fonctionnement de LSM. À l'amorçage du noyau, un module de sécurité est chargé (1). Ensuite, à chaque fois qu'un processus fera un appel système (2), le DAC fera une première vérification du droit d'accès (3) puis laissera la main au LSM

(4). Si un module a été chargé, la fonction idoine sera appelée et en fonction du résultat, le processus accèdera au fichier (5). Le module est donc appelé après le DAC et seulement si celui-ci autorise l'accès.

Au total, le code du noyau n'est presque pas modifié par kBlare qui rajoute seulement un *hook* au *framework* LSM. Le reste du code représente moins de 7 000 lignes de code C (pour 15 000 000 lignes de codes dans le Kernel en entier).

7.2 Modification du suivi de flux

La première étape de l'implémentation d'un mécanisme de coopération dans kBlare a été d'introduire le *tag* de coopération. La structure `blare_task_struct` présente dans le fichier `security/blare/blare.h` a donc été enrichie d'un nouveau champ `cinfo_list`, de même type que `info_list` (le champ représentant le *tag* d'information).

Il a ensuite fallu modifier les différents *hooks* LSM, afin de coller au formalisme exprimée dans le chapitre III. Plutôt que d'utiliser un booléen pour savoir si un processus est actuellement en train de coopérer, nous avons préférer définir un pointeur vers `info_list` (pas de coopération) ou `cinfo_list` (coopération) suivant le cas. Cela permet de ne pas avoir de test à faire lors des flux en écriture, le pointeur contient toute l'information dont on a besoin. Et si, dans certains cas, on a besoin de connaître si oui ou non le processus est en train de coopérer, il suffit de comparer la valeur du pointeur avec la valeur des adresses de `info_list` ou `cinfo_list`.

7.3 API pour dialogue avec l'espace utilisateur

Modifier le suivi de flux n'avait aucun intérêt si nous ne fournissions pas un moyen aux développeurs d'intégrer les mécanismes de coopérations à leurs programmes. Ces derniers se résument à (1) demander à coopérer, (2) lire la valeur de son *tag* de coopération de l'application et (3) modifier ce *tag*. Ces trois opérations impliquent des communications entre espace noyau et espace utilisateur et comme souvent, il existe de nombreuses façons de les réaliser.

Notre choix s'est porté sur l'utilisation de Netlink, qui est une famille de *socket* qui permet justement des communications entre l'espace noyau et l'espace utilisateur (mais aussi entre différents modules du noyau et entre différentes applications utilisateurs). Le fonctionnement de Netlink repose sur des familles de protocoles. Chaque famille correspondant à un module du noyau, par exemple `NETLINK_SELINUX` est utilisé pour les notifications de SELinux. Le nombre de familles maximum étant limité, les développeurs du noyau ont implémenté une famille dite « générique. »

L'API que nous avons développé est constituée de trois fichiers d'en-têtes : `ctag.h`, `coop.h` et `policy.h`. Le fichier `ctag.h` fournit les abstractions nécessaires pour manipuler facilement les *tags* de coopération (une structure `struct ctag_t`, des fonctions d'allocations et de destruction, des fonctions de manipulation pour fusionner, séparer, etc.). Le fichier `coop.h` fournit les fonctions pour réaliser la coopération avec le noyau : initialiser la coopération, lire le *tag* de coopération, le modifier. Enfin, le fichier `policy.h` fournit la définition d'une fonction permettant de modifier la politique de sécurité (voir 4.2).

L'API peut ainsi être utilisée par n'importe quelle application de l'espace utilisateur et communique avec le module kBlare du noyau. Ce dernier déclare deux familles Netlink basées sur la famille générique, une pour la coopération et une pour la politique.

(a) Dans JamVM

(b) Dans jBlare

FIGURE 9 – Organisation d'une *frame* de fonction

8 jBlare

Comme pour kBlare, jBlare a été plusieurs fois implémentée. La dernière version date de 2011 et a été implémentée par Mounir ASSAF lors de son stage de Master 2 Recherche.

8.1 Analyse dynamique et analyse hybride

Son travail s'est décliné en deux JVM distinctes. La première implémente un suivi de flux purement dynamique, la seconde est plus aboutie et offre un suivi de flux dynamique se basant sur une analyse statique qui rajoute des annotations dans le *bytecode* (de manière totalement portable) et qui permet un suivi plus efficace et précis. Nous avons choisi de modifier le jBlare purement dynamique, qui est plus facilement utilisable car ne nécessitant pas une compilation particulière.

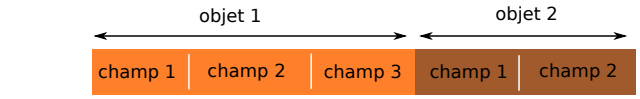
D'un point de vue purement implémentation, les labels de sécurité sont de simples tableaux d'entiers. Ils ont une taille fixe afin de ne pas avoir à faire d'allocation dynamique, opération désastreuse en terme de temps d'exécution.

Pour comprendre comment fonctionne jBlare, il faut donc s'intéresser au fonctionnement interne de la JVM, qui repose sur une pile : la pile des *frames* de fonctions. Une *frame* de fonction, d'un point de vue JVM, est une structure contenant toutes les informations et données relatives à une fonction précise. Si sa composition exacte n'a pas beaucoup d'intérêt ici, elle possède cependant deux éléments incontournables : les locales et la pile interne de la fonction. Les locales sont des registres correspondant à une référence vers `this` (objet appelant), les arguments de la fonction et les variables locales. La *frame* contient aussi une pile car le bytecode est en effet un langage à pile, comme peut l'être l'assembleur.

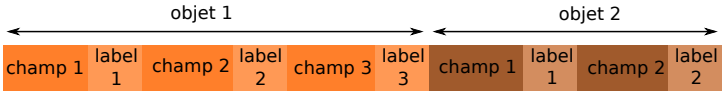
Une donnée peut donc se trouver à trois endroits distincts : dans les locales, dans la pile interne ou dans le tas (objet alloué). Dans ces trois cas, les labels ne sont pas stockés de la même façon.

Dans la *frame* La Figure 9 montre l'organisation d'un élément d'une *frame* de fonctions dans JamVM (a) et dans jBlare (b). Le bloc « structure *frame* » correspond à des pointeurs qui permettent d'accéder facilement aux autres blocs. Le nombre de variables locales et la taille maximale de la pile des opérandes sont données par le *bytecode* de la fonction. Dans jBlare, deux blocs sont rajoutés pour stocker les labels et sont des miroirs. Ainsi, le premier élément du bloc labels locales est le label de la première variable locale, etc.

Dans le tas La Figure 10 montre l'organisation du tas dans JamVM (a) et dans jBlare (b). Dans JamVM, les objets sont rangés les uns après les autres et sont organisés comment



(a) Dans JamVM



(b) Dans jBlare

FIGURE 10 – Organisation du tas

étant une succession de champs de différents tailles. Dans jBlare, on ajoute à la suite de chaque champ le label correspondant.

8.2 Modification de l'interpréteur

Dans jBlare, l'interpréteur de JamVM est déjà modifié et prend en charge le suivi de flux interne aux applications Java. Cependant, ce suivi n'était pas complètement implémenté : il manquait l'opération d'union entre deux labels. Dans jBlare, les labels ont une taille fixes. Nous avons gardé ce choix d'implémentation car il permet de limiter le surcout à l'exécution en limitant le nombre d'allocations dynamiques. Cela permet aussi d'avoir un code beaucoup plus simple. Une modification en profondeur de jBlare pour donner une taille dynamique aux labels aurait entraîné deux problèmes : une baisse importante de performance (beaucoup d'allocations dynamiques) et un travail conséquent de mise à jour. Dans un premier temps, nous avons donc décidé de laisser cette mise à jour de côté. En cas de dépassement de la capacité des labels, jBlare imprime un message d'erreur.

La modification principale que nous avons apportée à l'interpréteur vise à adapter son comportement lorsqu'une application Java fait appel à du code natif, pour implémenter la pile de *tags*. Il faut récupérer les labels des variables passés en argument pour définir le nouveau *tag* de coopération de l'application et sauvegarder l'ancien dans la fameuse pile. C'est lors de cette étape que nous avons repéré et corrigé un bug assez gênant de jBlare, qui n'avait pas tous ses labels initialisés correctement. En l'occurrence, il s'agissait des labels des variables statiques. Ce bug montre bien la difficulté de reprendre un projet aussi complexe et complet que JamVM pour l'adapter à de nouveaux besoins.

L'API développée pour la coopération avec kBlare fournit déjà une structure pour faciliter la manipulation des *tags* de coopération mais la structure de pile était à faire. Une fois cette dernière implémentée et les *tags* des variables correctement initialisés, la modification de l'interpréteur n'était pas difficile.

8.3 Modification de JNI

Modifier l'interpréteur de jBlare ne permet d'implémenter que les règles *CallJNI* et *RetJNI*. Les autres règles (*CallJava*, *CallJNI*, *Get* et *Set*) sont implémentées en modifiant l'interface JNI de jBlare, directement héritée de JamVM sans aucune modification. En réalité, comme pour kBlare qui possédait une gamme d'opération plus large que les simples *Read*, *Write* et *Append*, JNI ne fournit pas que quatre opérations mais beaucoup plus :

l'interface offre plus de 110 fonctions ! Heureusement, toutes n'ont pas à être modifiées, mais *CallJava* par exemple se dérive concrètement en 18 fonctions englobant tous les cas de figures (fonctions statiques ou non, fonctions rendant une variable ou non, etc.).

Les modifications à apporter à chaque fonction sont globalement triviales et courtes, mais la difficulté réside plus dans le nombre important de fonctions à lire et à comprendre. De plus, comme JNI permet d'interagir avec tous les éléments de Java (types primitifs, références, objets, mais aussi tableaux) cela oblige à comprendre en profondeur comment jBlare gère les labels pour chacun de ces cas de figures.

Cinquième partie
Expérimentation

Pour les expérimentations qui vont suivre, nous avons exécuté la nouvelle version de jBlare sur un système Linux (ArchLinux à jour). Nous avons utilisé le système de virtualisation QEMU pour faciliter la mise en œuvre de nos tests.

9 Discussion sur l'efficacité

Nous cherchons dans un premier temps à savoir si la coopération telle qu'elle a été implémentée fonctionne correctement ou non.

9.1 Programme jouet

Pour valider le mécanisme de coopération proposé, la première étape consiste à prendre un programme « jouet, » c'est-à-dire très simple et dont on possède l'entière maitrise. Dans ces conditions, il est préférable d'écrire son propre code JNI plutôt que d'utiliser l'API Java. Nous définissons la classe `SimpleMethod`, proposant deux fonctions natives. La fonction `char getFirstChar(String file)` lit et renvoie le premier caractère du fichier `file`. La fonction `void writeChar(String file, char c)` écrit le caractère `c` dans le fichier `file`. Une fonction statique `main` est ajoutée à la classe `SimpleMethod`, ce qui rend le fichier `.class` exécutable.

L'implémentation Java est succincte, les deux fonctions natives n'étant que déclarées et pas définies. On remarque l'utilisation de la directive `System.loadLibrary` qui permet de charger la bibliothèque partagée regroupant les implémentations des fonctions en C.

```
class SimpleMethod
{
  static {

    System.loadLibrary("simplemethod");
  }

  public native char getFirstChar(String file);
  public native void writeChar(String file, char c);

  public static void main (String [] args)
```

```
  {
    SimpleMethod sm = new SimpleMethod();
    char c = sm.getFirstChar("f");
    char c2 = sm.getFirstChar("h");
    sm.writeChar("g", c);
    sm.writeChar("i", c2);
  }
}
```

L'implémentation en C n'est guère plus longue mais peut-être moins évidente à lire étant donné qu'elle utilise l'interface JNI et utilise donc des constructions inhabituelles. `JNIEnv` symbolise l'environnement Java et propose un ensemble de fonction permettant l'interaction avec, par exemple `GetStringUTFChars` qui permet de convertir un objet `String` en `char *`.

```
#include <stdlib.h>
#include <stdio.h>

#include "SimpleMethod.h"

JNIEXPORT jchar JNICALL Java_SimpleMethod_getFirstChar
  (JNIEnv *env, jobject this, jstring file)
{
  const char *filename = (*env)->GetStringUTFChars(env, file, 0);
  FILE* f = fopen(filename, "r");
  char res = fgetc(f);

  (*env)->ReleaseStringUTFChars(env, file, filename);

  fclose(f);

  return res;
}


JNIEXPORT void JNICALL Java_SimpleMethod_writeChar
  (JNIEnv *env, jobject this, jstring file, jchar c)
{
  const char *filename = (*env)->GetStringUTFChars(env, file, 0);
  FILE* f = fopen(filename, "w");
  char res = fgetc(f);

  (*env)->ReleaseStringUTFChars(env, file, filename);
  fputc(c, f);

  fclose(f);
}
```

(a) Avant

Fichier	Contenu	Tags
f	"Coucou"	{1}
h	"Bonjour"	{2}
g	""	{}
i	""	{}

(b) Après

Fichier	Contenu	Tags
f	"Coucou"	{1}
h	"Bonjour"	{2}
g	"C"	{1}
i	"B"	{2}

TABLE 1 – Évolution des *tags* des fichiers

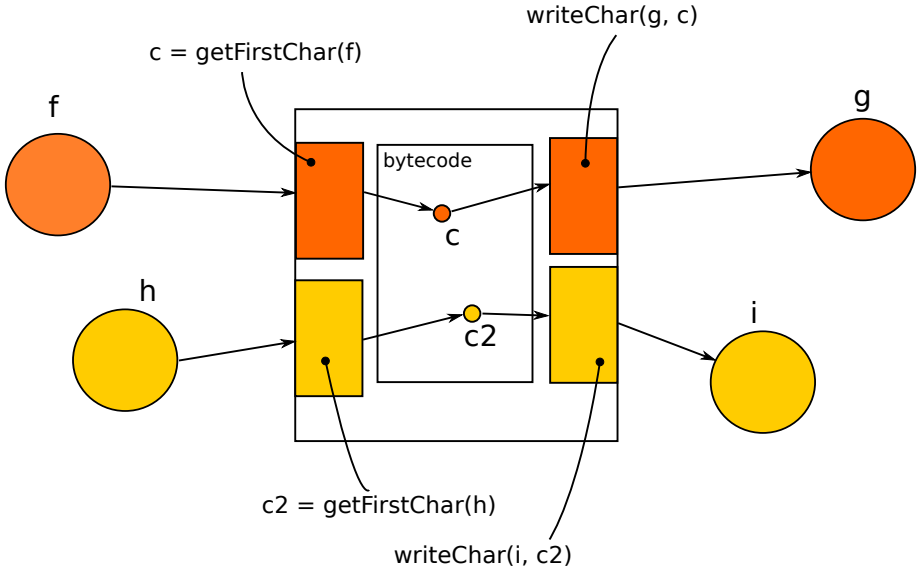


FIGURE 11 – Scénario du programme joué

Le programme va lire le premier caractère de deux fichiers distincts et écrire ces deux caractères dans deux autres fichiers. Les tableaux 1 résume l'évolution des *tags* des fichiers et la Figure 11 illustre ce qu'il se passe. On remarque que la sur approximation de kBlare est supprimée grâce à la coopération établie avec jBlare.

9.2 Avec l'API Java

Dans la pratique, un développeur Java ne va pas écrire son propre code JNI à chaque fois qu'il veut interagir avec le système. L'API Java fournit les abstractions (classes) nécessaires. Il existe plusieurs implémentations de la librairie standard, nous utilisons OpenJDK version 6. Pour tester jBlare « en conditions réelles » nous avons implémenté un programme effectuant exactement les mêmes actions que précédemment, mais avec les classes `FileReader` et `FileWriter`.

```
import java.io.*;

class JavaOnly
{
```

```
  public static void main (String [] args)
  {
    try {
      FileReader f = new FileReader("f");
      FileReader h = new FileReader("h");

      char[] c = new char[1], c2 = new char[1];

      f.read(c);
      h.read(c2);

      FileWriter g = new FileWriter("g");
      FileWriter i = new FileWriter("i");

      g.append(c[0]);
      i.append(c2[0]);

      f.close();
      g.close();
      h.close();
      i.close();

    } catch (IOException e) {}
  }
}
```

De façon très satisfaisante, nous obtenons avec ce programme Java les mêmes résultats qu'avec notre programme jouet, à savoir que nous supprimons la sur approximation de kBlare.

9.3 Faiblesses

Le fonctionnement de la coopération entre kBlare et jBlare repose sur une hypothèse forte qui n'a pas encore été explicitée jusqu'ici : les fonctions natives ne peuvent pas s'échanger d'informations autrement qu'en utilisant l'interface JNI ou les appels systèmes. Ces derniers étant surveillés respectivement par jBlare et kBlare, sous cette hypothèse nous pouvons affirmer que la coopération ne fait pas de sous approximation (une sous approximation entraîne l'existence possible de faux négatifs).

Cette hypothèse est néanmoins fausse. En effet, le code natif possède les mêmes possibilités qu'un programme C classique, en l'occurrence il peut utiliser le tas *via* l'allocation dynamique ou utiliser des variables statiques. On peut ainsi imaginer la fonction JNI suivante :

```
JNIEXPORT jchar JNICALL Java_SomeClass_readFromFileF
  (JNIEnv *env, jobject this, jstring file, jchar c)
{
  static char c = 0;
```

```
  if (!c) {
    FILE* f = fopen(f, "r");
    c = fgetc(f);
    fclose(f);
  }

  return c;
}
```

Au premier appel de cette fonction, `c` vaudra 0. L'utilisation de la fonction `fgetc` engendrera un appel système *read-like* et le *tag* de coopération sera donc mis à jour en conséquence, ce qui fait que le retour de la fonction `readFromFileF` sera effectivement teinté par le fichier `f`. À l'appel suivant, cependant, comme `c` aura été initialisé, la fonction `fgetc` ne sera pas appelée et le résultat ne sera pas teintée comme il le devrait, amenant à une sous approximation des *tags*. Ce comportement reste néanmoins inhabituel et les fonctions natives ont surtout pour but d'être atomique.

10 Performances

Nous nous sommes intéressés à l'impact de nos modifications sur les performances de kBlare et jBlare. En effet, notre mécanisme de coopération intervient dans toutes les fonctions du module de sécurité et dans l'interpréteur *bytecode*. Nous n'avons pas eu le temps de faire des *benchmarks* pertinents et révélateurs. Nous décrivons cependant les conditions dans lesquels nous comptons les faire. L'étude est en cours et nous n'avons pas encore pu obtenir des résultats satisfaisants.

10.1 *Overhead* kBlare

Nous souhaitons savoir quel est l'impact de nos modifications sur kBlare en terme de performance. Dans un premier temps, nous avons choisi de nous limiter aux interactions avec le système de fichiers. L'essentiel de nos essais sur la coopération ont en effet été réalisés sur le système de fichier, comme le montre la section précédente.

Pour mesurer l'*overhead* de kBlare, nous téléchargerons sur `kernel.org` les sources de noyau Linux, que nous teintons de *tags* d'information tous uniques. Par la suite, nous créons une archive vide et rajoutons un à un tous les fichiers précédemment préparés. Nous comparerons les temps d'exécution de l'archivage. Nous comparerons les temps d'exécution obtenu avec un noyau Linux non modifié, un noyau utilisant kBlare sans mécanisme de coopération et un noyau utilisant kBlare avec mécanisme de coopération.

L'overhead moyen annoncé par les auteurs de kBlare est de l'ordre de 10%. Nous nous attendons à une valeur supérieure dans nos propres expérimentations. La raison est que nous testons les interactions avec le système de fichiers, qui est la situation la plus défavorable pour le module de sécurité. En effet, kBlare n'implémente encore aucun mécanisme de cache ou d'optimisations.

10.2 *Overhead* jBlare

L'*overhead* de jBlare est son grand point faible : il faut compter sur un facteur multiplicatif allant de 4 à 10 fois le temps d'exécution d'une JamVM non modifiée. Ce chiffre n'est

pas forcément représentatif, car il concerne une version de jBlare qui n'implémentait pas une coopération véritable. Comme pour calculer l'*overhead* de kBlare, nous nous concentrerons ici sur les interactions avec le système de fichiers. Les tests seront réalisés sur un noyau kBlare modifié pour implémenter notre mécanisme de coopération.

Sixième partie
Conclusion

Notre objectif était d'améliorer la précision de kBlare, un IDS *Host-based* et *Policy-based* utilisant le suivi de flux d'information et plus particulièrement le modèle Blare, décrit par l'équipe CIDre. Pour ce faire, nous devions proposer un mécanisme de coopération entre kBlare et des applications tierces de confiance implémentant elles aussi le modèle Blare. La validation de cette proposition devait se faire grâce à son implémentation dans kBlare et dans jBlare, une JVM modifiée pour faire du suivi de flux d'information au niveau *bytecode*.

Le mécanisme de coopération que nous proposons repose sur un *tag* de coopération attaché aux processus. Il évolue en parallèle du *tag* d'information et est utilisé pour la propagation des *tags* en écriture quand l'application exécutée par le processus a demandé et obtenu de coopérer. Pour déterminer si une application peut ou non coopérer avec kBlare, nous avons introduit la notion de politique de coopération. Une application coopérant avec kBlare peut modifier son *tag* de coopération, pour rendre la propagation de *tag* plus précise grâce à sa connaissance de ses flux internes.

Nous avons ensuite adapté jBlare pour qu'il puisse réaliser une coopération avec kBlare. À cause du fonctionnement des JVM, il est impossible d'effectuer une coopération parfaite qui élimine complètement la sur approximation de kBlare. En effet, les interactions entre une application Java et le système se font *via* l'interface JNI, qui permet d'utiliser du code natif qui n'est pas surveillé par jBlare. Nous avons néanmoins proposé et implémenté un modèle qui permettait de réduire fortement la sur approximation de kBlare. Cette implémentation obtient de bons résultats avec des applications Java simples.

Références

[AAB+05] Bowen Alpern, Steve Augart, Stephen M Blackburn, Maria Butrico, Anthony Cocchi, Perry Cheng, Julian Dolby, Stephen Fink, David Grove, Michael Hind, et al. The jikes research virtual machine project : building an open-source research community. *IBM Systems Journal*, 44(2) :399–417, 2005.

[Ass11] Mounir Assaf. Utilisation de méthodes hybrides pour la détection d'intrusion paramétrée par la politique de sécurité reposant sur le suivi des flux d'information. 2011.

[ATM12] Radoniaina Andriatsimandefitra, Valérie Viet Triem Tong, and Ludovic Mé. User data on android smartphone must be protected. *Cybercrime*, page 18, 2012.

[Axe00] Stefan Axelsson. Intrusion detection systems : A survey and taxonomy. Technical report, Technical report, 2000.

[Bau06] Mick Bauer. Paranoid penguin : an introduction to novell apparmor. *Linux Journal*, 2006(148) :13, 2006.

[Bib77] Kenneth J Biba. Integrity considerations for secure computer systems. Technical report, DTIC Document, 1977.

[BL73] D Elliott Bell and Leonard J LaPadula. Secure computer systems : Mathematical foundations. Technical report, DTIC Document, 1973.

[DDW99] Hervé Debar, Marc Dacier, and Andreas Wespi. Towards a taxonomy of intrusion-detection systems. *Computer Networks*, 31(8) :805–822, 1999.

[Den87] Dorothy E. Denning. An intrusion-detection model. *Software Engineering, IEEE Transactions on*, (2) :222–232, 1987.

[EKV+05] Petros Efstathopoulos, Maxwell Krohn, Steve VanDeBogart, Cliff Frey, David Ziegler, Eddie Kohler, David Mazieres, Frans Kaashoek, and Robert Morris. Labels and event processes in the asbestos operating system. *ACM SIGOPS Operating Systems Review*, 39(5) :17–30, 2005.

[HCF05] Vivek Haldar, Deepak Chandra, and Michael Franz. Dynamic taint propagation for java. In *Computer Security Applications Conference, 21st Annual*, pages 9–pp. IEEE, 2005.

[HSB+12] Dina Hadžiosmanović, Lorenzo Simionato, Damiano Bolzoni, Emmanuele Zambon, and Sandro Etalle. N-gram against the machine : on the feasibility of the n-gram network analysis for binary protocols. In *Research in Attacks, Intrusions, and Defenses*, pages 354–373. Springer, 2012.

[Ilg93] Koral Ilgun. Ustat : A real-time intrusion detection system for unix. In *Research in Security and Privacy, 1993. Proceedings., 1993 IEEE Computer Society Symposium on*, pages 16–28. IEEE, 1993.

[KR02] Calvin Ko and Timothy Redmond. Noninterference and intrusion detection. In *Security and Privacy, 2002. Proceedings. 2002 IEEE Symposium on*, pages 177–187. IEEE, 2002.

[KS94] Sandeep Kumar and Eugene H Spafford. A pattern matching model for misuse intrusion detection. 1994.

[Lam74] Butler W Lampson. Protection. *ACM SIGOPS Operating Systems Review*, 8(1) :18–24, 1974.

[Lov10] Robert Love. *Linux kernel development*. Addison-Wesley Professional, 2010.

[ML00] Andrew C Myers and Barbara Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 9(4) :410–442, 2000.

[MZZ+01] Andrew C Myers, Lantian Zheng, Steve Zdancewic, Stephen Chong, and Nathaniel Nystrom. Jif : Java information flow. *Software release. Located at http ://www. cs. cornell. edu/jif*, 2005, 2001.

[NAGL10] Pablo Neira-Ayuso, Rafael M Gasca, and Laurent Lefevre. Communicating between the kernel and user-space in linux using netlink sockets. *Software : Practice and Experience*, 40(9) :797–810, 2010.

[PN97] Phillip A Porras and Peter G Neumann. Emerald : Event monitoring enabling response to anomalous live disturbances. In *Proceedings of the 20th national information systems security conference*, pages 353–365, 1997.

[R+99] Martin Roesch et al. Snort-lightweight intrusion detection for networks. In *Proceedings of the 13th USENIX conference on System administration*, pages 229–238. Seattle, Washington, 1999.

[RPB+09] Indrajit Roy, Donald E Porter, Michael D Bond, Kathryn S McKinley, and Emmett Witchel. *Laminar : Practical fine-grained decentralized information flow control*, volume 44. ACM, 2009.

[Sim03] Vincent Simonet. The flow caml system. *Software release. Located at http ://-cristal. inria. fr/˜ simonet/soft/flowcaml*, 218, 2003.

[SM03] Andrei Sabelfeld and Andrew C Myers. Language-based information-flow security. *Selected Areas in Communications, IEEE Journal on*, 21(1) :5–19, 2003.

[SP10] Robin Sommer and Vern Paxson. Outside the closed world : On using machine learning for network intrusion detection. In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 305–316. IEEE, 2010.

[SS94] Ravi S Sandhu and Pierangela Samarati. Access control : principle and practice. *Communications Magazine, IEEE*, 32(9) :40–48, 1994.

[SVS01] Stephen Smalley, Chris Vance, and Wayne Salamon. Implementing selinux as a linux security module. Technical report, NAI Labs Report, 2001.

[VRKK03] Giovanni Vigna, William Robertson, Vishal Kher, and Richard A Kemmerer. A stateful intrusion detection system for world-wide web servers. In *Computer Security Applications Conference, 2003. Proceedings. 19th Annual*, pages 34–43. IEEE, 2003.

[VTTCM10] Valérie Viet Triem Tong, Andrew J Clark, and Ludovic Mé. Specifying and enforcing a fine-grained information flow policy : Model and experiments. *Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications*, 1(1) :56–71, 2010.

[WCS+02] Chris Wright, Crispin Cowan, Stephen Smalley, James Morris, and Greg Kroah-Hartman. Linux security modules : General security support for the linux kernel. In *Proceedings of the 11th USENIX Security Symposium*, volume 2, page 44. San Francisco, CA, 2002.

[ZBWKM06] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making information flow explicit in HiStar. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 263–278. USENIX Association, 2006.

[Zda04] Steve Zdancewic. Challenges for information-flow security. In *Proceedings of the 1st International Workshop on the Programming Language Interference and Dependence (PLID'04)*, 2004.