Utilisation de méthodes hybrides pour la détection d'intrusion paramétrée par la politique de sécurité reposant sur le suivi des flux d'information

Mounir Assaf

Master Recherche en Informatique - Rennes 1
Encadrant : Guillaume Hiet (équipe SSIR, Supélec)

Utilisation de méthodes hybrides pour la détection d'intrusion paramétrée par la politique de sécurité reposant sur le suivi des flux d'information

Mounir ASSAF

Rennes, Juin 2011

Table des matières

Chapitre 1

Introduction

Les systèmes d'information sont de nos jours de plus en plus confrontés à de nombreux risques menaçant leur sécurité. La protection de ces systèmes suppose dans un premier temps de définir une politique de sécurité exprimant les besoins en terme de confidentialité, intégrité et disponibilité des biens et services, puis dans un second temps de mettre en œuvre cette politique.

Généralement, des mécanismes d'authentification et de contrôles d'accès sont déployés pour la mise en œuvre de la politique de sécurité. Ils vérifient d'abord l'identité des utilisateurs, puis s'assurent que ces derniers ont bien les autorisations nécessaires pour accéder aux ressources du système. Cependant, il est important de noter que ces approches préventives ne sont pas toujours suffisantes pour garantir les propriétés de sécurité. Elles nécessitent la vérification et le contrôle des actions des utilisateurs afin de s'assurer à postériori que la politique est bien respectée. Intervient alors la détection d'intrusion qui n'est autre que la détection des violations de la politique de sécurité [And80].

Plusieurs modèles de systèmes de détection d'intrusion (IDS) ont été proposés depuis les travaux séminaux d'Anderson [FHS97, SCS98, KR02, ZMB03, HMZ+07]. Certaines approches, dites par scénarios, se basent sur une connaissance à priori des attaques contre le système. D'autres, dites comportementales, définissent plutôt le comportement de référence du système. Ces approches comportementales ont l'avantage de pouvoir détecter même des attaques nouvelles contrairement aux approches par scénarios.

Dans le cadre de ce stage, nous nous intéressons en particulier à un modèle comportemental de détection d'intrusion, paramétré par la politique de sécurité et reposant sur le suivi du flux d'information. Ce modèle proposé par le laboratoire SSIR, a été implémenté au niveau du système d'exploitation Linux. Une deuxième implémentation a ensuite été réalisée au niveau de la machine virtuelle Java (JVM). Cette seconde implémentation (JBlare) s'intègre avec la première (Blare) et permet d'assurer un suivi dynamique du flux d'information à différents niveaux selon les applications. L'objectif de ce stage est d'étudier les techniques d'analyse statique qui pourraient être mises en oeuvre pour le suivi des flux d'information au niveau des programmes Java afin de diminuer l'impact de l'analyse dynamique et améliorer la précision du suivi.

Ce rapport est organisé comme suit : Nous dresserons dans la première partie 2 un état de l'art de la détection d'intrusion et du contrôle de flux d'information. Nous montrerons les limites du contrôle d'accès, puis nous présenterons les différents modèles de détection d'intrusion, dont un modèle paramétré par la politique de sécurité et reposant sur le suivi des flux d'information. Nous exposerons ensuite deux modèle de politique de flux et les différentes approches pour le contrôle de

flux d'information. Nous introduisons ensuite dans la partie 3 le contexte de ce stage. Nous fixerons nos objectifs, puis nous présenterons les outils sur lesquels nous fondons notre approche. La partie 4 présentera ensuite en détail la démarche que nous proposons. Enfin, nous concluerons et discuterons des travaux futurs envisagés dans la dernière partie 5.

Chapitre 2

État de l'art

Nous présentons dans cette partie deux modèles de contrôle d'accès et leurs limites. Nous montrons que ces mécanismes sont soit très restrictifs, soit insuffisants pour garantir l'application de la politique de sécurité. Ensuite, différents modèles de détection d'intrusion sont exposés, dont un modèle paramétré par la politique de sécurité et reposant sur le suivi du flux d'information. Nous présontons aussi dans cette partie deux modèles de politiques de flux ainsi que les propriétés qu'ils cherchent à garantir, puis nous faisons un état de l'art concernant les mécanismes de contrôle de flux.

2.1 Contrôle d'accès

Les mécanismes de contrôle d'accès définissent en général deux types d'entités différentes. Les sujets sont les entités actives du système et peuvent représenter des utilisateurs ou des programmes. Les objets par contre sont les entités passives du système et représentent l'ensemble des ressources accessibles d'un système, des fichiers par exemple. Il est intéressant de noter qu'un objet peut très bien être considéré comme un sujet, dès lors qu'un processus actif peut créer un nouveau processus fils, le terminer ou le suspendre selon les autorisations qu'il possède.

2.1.1 DAC

L'un des modèles classiques de contrôle d'accès est le contrôle d'accès discrétionnaire (DAC). Ce modèle est largement implémenté dans les systèmes d'exploitation. Il définit pour les sujets un ensemble d'actions qu'ils peuvent réaliser sur des objets. La définition de cet ensemble de permissions sur un objet particulier est laissée à la discrétion du sujet auquel cet objet appartient. Dans [HRU76], les auteurs proposent de représenter l'ensemble des permissions par une matrice d'accès telle que l'illustre la figure 2.1. Les lignes de cette matrice correspondent aux sujets et les colonnes correspondent aux objets (qui peuvent eux-mêmes être des sujets). L'objet O par exemple appartient au sujet S1 qui possède sur cet objet les droits de lecture, écriture et exécution. Le sujet S1 n'autorise par contre que la lecture et l'exécution de l'objet O pour le sujet S2.

Bien que ce modèle ait connu un grand succès, il est implémenté dans Unix, Linux et Windows, grâce à la flexibilité qu'il introduit, il souffre néanmoins d'un inconvénient majeur permettant de contourner la politique de sécurité. Prenons l'exemple du sujet S3 dans la figure 2.1. N'ayant pas été autorisé à lire le contenu de l'objet O par le sujet S1, S3 ne devrait pas pouvoir y accéder. S2 qui a

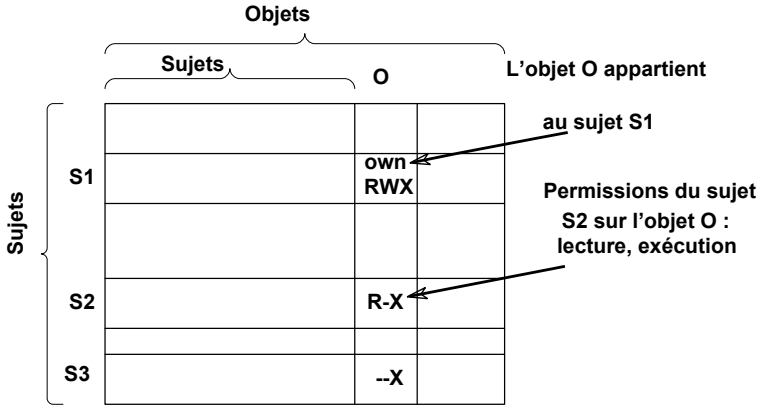

FIGURE 2.1 – Matrice de contrôle d'accès

les droits de lecture sur O, pourrait toutefois copier ce contenu dans un second objet qu'il mettrait à la disposition de S3. Ce problème est inhérent au fait que ce modèle ne tient pas compte des flux d'informations engendrés par la suite d'accès auxquels a été autorisé le sujet S2. En effet, une fois l'accès à l'information autorisé, ce modèle n'impose pas de contraintes quant à la dissémination de l'information contrairement au modèle introduit par Bell et Lapadula [LB73], présenté par la suite.

2.1.2 MAC

Le contrôle d'accès obligatoire (MAC) impose que les droits d'accès aux objets ne soient pas seulement définis par les propriétaires des objets concernés. Bell et Lapadula [LB73] ont introduit un modèle de contrôle d'accès obligatoire qui associe à tous les sujets et objets un niveau de sécurité. Les sujets reçoivent un niveau d'habilitation ainsi qu'un ensemble de domaines auxquels ils sont confinés. Les objets reçoivent un niveau de classification et un ensemble de domaines. Ce modèle définit de nouvelles règles d'accès afin d'assurer le confinement de l'information en se basant sur les niveaux de sensibilité affectés aux sujets (habilitation) ou objets (classification) ainsi que sur les domaines auxquels ils appartiennent. Une relation d'ordre total $\leq$ est définie sur l'ensemble des niveaux de sensibilité, alors qu'une relation d'ordre partiel correspondant à la relation $\subseteq$ de la théorie des ensembles est définie pour les domaines.

Les règles du contrôle d'accès obligatoire dépendent de la propriété de sécurité à laquelle on s'intéresse, confidentialité ou intégrité. En effet, le modèle proposé par Bell et Lapadula [LB73] assure la confidentialité alors que celui proposé par Biba [Bib77] assure l'intégrité. Comme l'illustre la figure 2.2, le modèle mettant en œuvre la confidentialité assure qu'un sujet S ne pourra lire l'information contenue dans un objet O que dans le cas où le niveau de classification de O est inférieur ou égal au niveau d'habilitation de S pour la relation $\leq$, et que le domaine de S domine le domaine de O pour la relation $\subseteq$. Ce modèle confine aussi le flux d'information puisque le sujet S ne peut déclassifier l'information. Pour que S puisse modifier un objet O, il faut que le niveau d'habilitation de S soit inférieur au niveau de classification de O et que le domaine de S domine le domaine de O. Ce modèle impose donc bien de ce fait la confidentialité de l'information pouvant être lue. De la même manière, dans le cas du modèle mettant en œuvre l'intégrité, un sujet ne peut modifier un objet s'il n'a pas un niveau d'habilitation supérieur au niveau de classification de l'objet et si son domaine ne domine pas le domaine de l'objet pour la relation $\subseteq$.

Type de politique / Types d'accès	**Confidentialité**	**Intégrité**
Accès en lecture	*no read up* $H_{sujet} \geq C_{objet}$ $D_{objet} \subseteq D_{sujet}$	*no read down* $H_{sujet} \leq C_{objet}$ $D_{objet} \subseteq D_{sujet}$
Accès en écriture	*no write down* $H_{sujet} \leq C_{objet}$ $D_{objet} \subseteq D_{sujet}$	*no write up* $H_{sujet} \geq C_{objet}$ $D_{objet} \subseteq D_{sujet}$

FIGURE 2.2 – Règles d'autorisation d'accès des modèles de contrôle d'accès obligatoire

Cette approche permet un certain contrôle du flux d'information, mais elle est très restrictive. Elle pose le problème de la contamination de labels (*label creep*) puisque le modèle de Bell et Lapadula impose qu'à chaque fois qu'un sujet habilité modifie un objet, le niveau de classification de l'information contenue dans cet objet augmente.

Les mécanismes classiques de mise en œuvre des politiques de sécurité, dits préventifs, s'avèrent souvent insuffisants ou trop restrictif. Le contrôle d'accès discrétionnaire n'est pas adapté pour suivre de manière fiable et précise, le flot d'information afin de garantir les propriétés d'intégrité et de confidentialité. Le contrôle d'accès obligatoire par contre est trop restrictif pour être utilisé dans des systèmes grand public ou professionnels. Il est donc nécessaire de s'en remettre à des mécanismes d'audit et de contrôle à postériori automatisés.

2.2 Détection d'intrusion

Les systèmes de détection d'intrusions servent en théorie à détecter toutes violations de la politique de sécurité [And80]. Ces systèmes se composent classiquement de trois éléments distincts. Un capteur collecte des données qui peuvent être des paquets transitant sur le réseau ou des journaux d'audit système par exemple. Ces données sont ensuite fournies à l'analyseur qui détectera des intrusions éventuelles dans le système. Enfin, le manager collecte les alertes produites par l'analyseur, les corrèle puis réagit ou les présente à l'opérateur. Nous allons nous intéresser dans le cadre de nos travaux aux méthodes d'analyses utilisées par ces systèmes.

Il existe deux écoles du point de vue des différentes méthodes d'analyse pour la détection. **L'approche par scénarios** définit un ensemble de comportements du système qui violent la politique de sécurité et cherche des motifs ou signatures correspondant à ces scénarios d'attaques pendant l'analyse des données collectées. **L'approche comportementale** par contre s'appuie sur la définition d'un ensemble de comportements de référence pour le système. Elle compare ensuite l'activité du système à ces profils « normaux » définis au préalable afin de détecter des déviations considérées comme étant des intrusions.

Nous allons présenter succinctement ces différentes approches en exposant leurs limites.

2.2.1 Approches par scénarios

Les approches par scénarios se basent sur une connaissance à priori des attaques contre les systèmes. La plupart des outils commerciaux tels que *Snort*[1] sont basés sur ces approches. Elles

1. http ://www.snort.org

définissent des signatures correspondant à l'exploitation de vulnérabilités connues et remontent une alerte dès qu'une tentative d'intrusion est détectée. Faciles à configurer, les systèmes de détection d'intrusion basés sur ce type d'approche, permettent aussi de fournir des rapports très précis concernant l'intrusion détectée, puisque chaque signature est en général propre à une vulnérabilité particulière, présente sur une version particulière du programme attaqué. Cependant, ces systèmes ne détectent pas les intrusions effectives, mais les attaques contre le système. Ils sont aussi très limités par le fait qu'ils ne peuvent détecter des attaques résultant de l'exploitation de vulnérabilités non publiques (*zero days vulnerabilities*). De plus, garder une base de connaissance des signatures d'attaques à jour peut s'avérer être une tâche très laborieuse au vue du nombre de nouvelles vulnérabilités découvertes chaque jour.

2.2.2 Approches comportementales

Les approches comportementales ou approches par détection d'anomalies s'attachent à définir un comportement « normal » du système. Toute déviation par rapport à ce comportement de référence est considérée comme étant une intrusion. Ces approches permettent en théorie de détecter les attaques inconnues, contrairement aux approches par scénarios. Mais elles posent alors le problème de la définition de ces comportements de référence.

Forrest [FHS97] propose notamment une approche qui s'inspire du système immunitaire du corps humain. Elle fait l'analogie entre la détection d'intrusion et la détection de corps étrangers par les cellules immunitaires et les anticorps. Cette approche construit dans une première phase une base de données des séquences d'appels systèmes observés durant une exécution normale d'un programme donné. Durant la seconde phase qui correspond à la phase d'analyse, la trace de l'exécution de ce programme est comparée aux séquences d'appels système de référence définis précédemment. À chaque fois que la trace d'exécution diverge suffisamment des séquences de référence, l'analyse lève une alerte d'intrusion. Les limites de cette approche dite par **apprentissage** sont intrinsèques à son mode de fonctionnement. En effet, cette approche suppose qu'un comportement anormal implique une intrusion. Cette hypothèse n'est pas nécessairement vraie, et peut induire la détection de plusieurs faux-positifs. Il se peut aussi que la phase d'apprentissage intègre dans le comportement de référence une intrusion. Ceci est d'autant plus probable s'il est possible de réapprendre et de compléter la trace de référence afin de tenir compte des évolutions possibles du système et du comportement des utilisateurs.

Une seconde approche proposée par Sekar [SCS98] consiste à utiliser une **spécification explicite du comportement**. Cette approche, contrairement à la précédente, spécifie le comportement normal au lieu de l'apprendre. L'auteur a donc été amené à définir un langage formel de haut niveau, *ASL* (*Auditing Specification Language*), afin de définir les comportements possibles d'un programme. Ces comportements sont spécifiés en terme de séquences d'évènements possibles et de conditions que ces événements doivent vérifier. Chaque événement défini dans le langage *ASL* correspond à un ensemble d'appels systèmes. Par exemple, la spécification de la figure 2.3 autorise le programme concerné à n'ouvrir que le fichier /etc/passwd et seulement en lecture. Ce langage permet aussi de

$$open(file, mode)|(realpath(file) \notin \{'/etc/passwd'\}) \vee (mode \neq RDONLY) \rightarrow fail$$

FIGURE 2.3 – Exemple de spécification de comportement en ASL

définir les mesures à prendre dans le cas où la compromission d'un programme est identifiée. Ces mesures permettent ainsi d'isoler le programme compromis pour s'assurer qu'il ne pourra menacer

la sécurité du système. Ce modèle a été validé par une implémentation proposée par Sekar. Néanmoins, il souffre d'un inconvénient majeur puisque la mise en place de spécifications pour chaque programme du système et chacune des mises à jour de ces programmes s'avére laborieuse.

Une troisième approche consiste à s'affranchir de la spécification des comportements légaux de tous les programmes et d'induire ce comportement à partir de la politique de sécurité. Cette approche dite **paramétrée par la politique de sécurité** a été initialement proposée par Ko et Redmond [KR02] afin de détecter les violations d'intégrité par des programmes concurrents (des conditions de course). Les travaux du laboratoire SSIR proposent un modèle plus général qui permet de prendre en compte à la fois les problématiques de confidentialité et d'intégrité [ZMB03, HMMT07] et ne sont pas cantonnés aux attaques dites de conditions de course. Ce modèle infère l'ensemble des comportements sains des applications en termes de flux d'informations autorisés en se basant sur la politique de sécurité définie au niveau de l'OS, la politique DAC notamment. Notons que ce modèle ne s'intéresse qu'à la détection des violations de propriétés de confidentialité et d'intégrité. Un programme malveillant présent sur le système ne sera détecté qu'au moment où il violera l'une de ces propriétés de sécurité par exemple. Une implémentation de ce modèle a été proposée dans [ZMB03, HMMT07]. Elle permet le suivi et le contrôle du flux d'informations au niveau OS (Blare) ainsi qu'au niveau de la JVM (JBlare).

JBlare réalise un suivi dynamique de flux d'information en instrumentant le bytecode des programmes Java. Ce suivi surapproxime cependant les flux engendrés au sein des méthodes Java, considérées comme des boîtes noires. Les parties suivantes s'intéresseront au contrôle de flux d'information et notamment aux mécanismes permettant d'améliorer les performances et la précision du suivi réalisé par JBlare.

2.3 Contrôle de flux d'information

Le contrôle de flux s'intéresse au suivi des flux d'information afin de s'assurer que ces flux sont légaux vis-à-vis de la politique de sécurité. Cette section présente formellement deux modèles de politique de flux. Le premier est classiquement utilisé pour le contrôle de flux, notamment en conjonction avec la propriété de non-interférence pour certifier la sécurité des programmes, propriété que nous présentons dans la deuxième sous-section. Le second modèle est plus adapté au contrôle dynamique. Il interprète la politique de sécurité en termes de flux autorisés entre objets du système.

2.3.1 Politique de flux en treillis

Un modèle de politique de flux, initialement proposé par [Den76] est classiquement utilisé dans les mécanismes de contrôle de flux. Ce modèle définit les flux possibles entre objets, en leur assignant à chacun un niveau de sécurité. Les flux d'information autorisés entre objets sont alors spécifiés en définissant une relation $\rightarrow$ sur l'ensemble des niveaux de sécurité. Le flux d'information d'un objet A vers B est alors autorisé si et seulement si leurs niveaux de sécurité $\underline{A}$ et $\underline{B}$ respectifs vérifient la relation : $\underline{A} \rightarrow \underline{B}$.

Denning [Den76] propose de modéliser toute politique de flux d'information par un modèle en treillis. Une relation $\oplus : SC \times SC \mapsto SC$ est introduite, où SC représente l'ensemble des labels de sécurite. Cette relation spécifie par exemple le label de sécurité de l'objet défini par l'application d'une opération binaire sur deux objets A et B : $\underline{(A+B)} = \underline{A} \oplus \underline{B}$.

Définition 1 (Modèle de politique de flux d'information en treillis)**.** Un modèle de flux d'information $< SC, \rightarrow, \oplus >$ forme un treillis si :

1. la relation $\rightarrow$ définit une relation d'ordre partielle sur l'ensemble SC ;
2. l'ensemble SC est fini ;
3. SC admet une borne inférieure L telle que $\forall \underline{A} \in SC, L \rightarrow \underline{A}$;
4. $\oplus$ définit une borne supérieure minimale sur $SC \times SC$.

L'hypothèse 1 suppose que la relation $\rightarrow$ définisse une relation d'ordre partielle sur SC. Elle est donc réflexive, antisymétrique et transitive. La réflexivité et l'antisymétrie ne posent pas de problèmes particuliers. En effet, il est cohérent que toute politique autorise les flux « intra-classes », un exemple typique étant l'affectation d'une variable à elle-même. De plus, si l'antisymétrie n'est pas vérifiée alors $\exists(\underline{A}, \underline{B}) \in SC \times SC$ tels que $\underline{A} \rightarrow \underline{B}$ et $\underline{B} \rightarrow \underline{A}$ avec $\underline{A} \neq \underline{B}$. Dans ce cas, rien ne justifie de différencier ces deux classes de sécurité puisqu'un flux d'une classe vers l'autre est considéré légal. La condition de transitivité implique que si $\underline{A} \rightarrow \underline{B}$ et $\underline{B} \rightarrow \underline{C}$ alors $\underline{A} \rightarrow \underline{C}$, c'est à dire que si le flux indirect entre A et C en passant par B est permis alors le flux entre A et C est permis. Cette condition est raisonnable, bien qu'il soit nécessaire parfois de définir des politiques de flux intransitives ; dans le cas de la déclassification par exemple, secret $\rightarrow$ déclassifié et déclassifié $\rightarrow$ publique alors que secret $\nrightarrow$ publique.

Les hypothèses 2 et 3 considèrent que l'ensemble SC est fini et qu'il admet une borne inférieure. En vue d'une implémentation sur un système informatique, l'ensemble SC doit être fini. L'hypothèse de l'existence d'une borne inférieure ne pose pas de problèmes particuliers non plus puisque dans tous systèmes informatiques, il existe des informations publiques si ce n'est les constantes utilisées dans les programmes informatiques. Pour l'hypothèse 4, par définition de $\oplus$, $\underline{A} \rightarrow \underline{A} \oplus \underline{B}$ et $\underline{B} \rightarrow \underline{A} \oplus \underline{B}$. Or si $\exists C \in SC$ tel que $\underline{C} \rightarrow \underline{A} \oplus \underline{B}$ et $\underline{C} \neq \underline{A} \oplus \underline{B}$, alors il suffit de poser $\underline{A} \oplus \underline{B} = \underline{C}$.

2.3.2 La propriété de non-interférence

Goguen et Meseguer [GM82] ont été les premiers à définir le notion de non-interférence. Cette propriété spécifie qu'un ensemble d'instructions pouvant être exécuté par un groupe d'utilisateurs est non-interférent avec un second groupe d'utilisateurs, si ces instructions n'ont aucun effet sur ce que pourrait observer ce second groupe. Considérons par exemple le cas simple de deux utilisateurs dont l'un (noté utilisateur S) est privilégié et peut avoir accès à de l'information classée secrète, alors que le second (noté utilisateur P) est non-privilégié et n'a accès qu'à de l'information publique. Cet utilisateur S a le droit d'exécuter des opérations O de modification, de création et de suppression de fichiers secrets F. Ces opérations sont alors non-interférentes avec le second utilisateur P si elles ne modifient nullement l'information publique concernant les fichiers F et observable par P. Ceci veut dire que P n'a aucun moyen d'inférer quelles opérations ont été exécutées en observant les fichiers F par exemple.

Pour définir formellement la propriété de non-interférence pour un programme p, un ensemble de labels de sécurité SC associés aux variables de p est considéré. Cet ensemble est muni d'une relation d'ordre partiel $\rightarrow$. Intuitivement, si les niveaux de sécurité des variables x et y, notés respectivement $\underline{x}$ et $\underline{y}$ vérifient la relation $\underline{x} \rightarrow \underline{y}$, alors le flux d'information de x vers y est autorisé. Une relation d'équivalence $\sim_l$, où $l \in SC$, est aussi définie pour deux mémoires μ et ν d'un programme, telle que $\mu \sim_l \nu$ si et seulement si pour toute variable x du programme considéré, $\underline{x} \rightarrow l \Longrightarrow \mu(x) = \nu(x)$.

La non-interférence peut alors être défini comme suit :

Définition 2. (non-interférence) Un programme p est non-interférent si
$\forall l \in SC, \mu \sim_l \nu \wedge (\langle p, \mu \rangle \Rightarrow \mu') \wedge (\langle p, \nu \rangle \Rightarrow \nu') \Longrightarrow \mu' \sim_l \nu'$

Cette définition inspirée de [VSI96] et [RS10], spécifie que les variables de niveau bas ne dépendent pas des variables de niveau haut. En effet, si les valeurs de toutes les variables de niveau supérieur à l sont changées entre deux exécutions successives du programme p, celà n'impactera pas du tout les valeurs des variables de niveau inférieur ou égale à l. Cette définition ne tient toutefois pas compte de la non-terminaison du programme qui pourrait causer une fuite d'information. Elle est dite insensible à la terminaison du programme.

2.3.3 Politique de flux en termes de CCAL

L'approche présentée par les travaux de l'équipe SSIR [HMMT07] [ZMB03] introduit un autre modèle de politique de flux. Ce modèle s'appuie sur la notion de contenus et de conteneurs. Les **conteneurs** sont les objets du système qui contiennent des **contenus**, c'est-à-dire l'information elle-même. Pour un fichier F, le modèle distingue donc l'information f qu'il contient de l'objet F en tant que réceptacle d'information. Par la suite, l'ensemble des contenus du système est noté $I = \{I_1, \ldots, I_n\}$ et l'ensemble des conteneurs est noté $O = \{O_1, \ldots, O_n\}$.

Le modèle proposé interprète la politique de sécurité définie sur le système afin d'en déduire les flux d'informations légaux. Il repose ainsi sur le concept de *CCAL* (*Contents - Container Authorized Link*) qui associe des contenus aux conteneurs vers lesquels ces contenus peuvent s'écouler sans enfreindre la politique de sécurité. La politique de flux est alors définie comme étant l'ensemble de ces *CCAL* :

Définition 3 (Politique de flux)**.** Une politique de flux est un ensemble des paires $(I_i, \Omega_i)_{i \in I}$ appelées *CCAL* où :
- $I_i \in \mathbb{P}(I)$ est un ensemble de contenus
- $\Omega_i \in \mathbb{P}(O)$ est un ensemble de conteneurs

Une violation de la politique de sécurité est alors définie par l'association d'un contenu I à un conteneur O alors que cette association n'est autorisée par aucun CCAL définissant la politique de flux :

Définition 4 (Violation de la politique de sécurité)**.** Une violation de la politique de sécurité est caractérisée par une situation où :
$\exists O, I$ *tels que* O *contient* I *et* $\nexists (I', \Omega)$ *tel que* $I \subseteq I'$ *et* $O \in \Omega$

2.4 Les mécanismes de contrôle de flux

Les mécanismes de contrôle de flux s'intéressent aux flux d'information engendrés par un programme informatique afin de garantir les propriétés de confidentialité et d'intégrité. Certains reposent sur une analyse statique afin de certifier toutes les exécutions possibles d'un programme, alors que d'autres dits dynamiques ou hybrides ne s'intéressent qu'à l'exécution courante d'un programme afin de s'assurer que cette exécution ne viole pas les propriétés de sécurité.

Deux types de flux d'information peuvent être distingués. Le premier est dit explicite, lorsque la valeur d'une variable x dépend explicitement de la valeur d'une variable y. Dans le cas de l'affectation $x := y$ par exemple, il y a un flux explicite de y vers x. Le second type de flux est dit

implicite lorsque la valeur d'une variable dépend d'un branchement conditionnel. Dans l'exemple **if** (*secret*) **then** *public* := 1 **else** *public* := 0, il y a un flux d'information implicite de la variable *secret* contrôlant le branchement vers la variable *public*.

2.4.1 L'analyse statique

Volpano et Smith [VSI96] ont été les premiers à proposer un système de type pour le contrôle de flux d'information en analyse statique. Ce système de type a été proposé au préalable pour un langage impératif simple présenté figure 2.5. La fiabilité de l'approche proposée a été prouvée en montrant que tout programme typable respecte la propriété de non-interférence insensible à la terminaison du programme (définition 2). Ce système de type a ensuite été étendu à un langage procédural simple [VS97]. Cette approche considère un modèle de flux d'information en treillis $< SC, \rightarrow, \oplus >$.

Un système de type introduit un ensemble de règles d'inférences et d'axiomes permettant la dérivation de jugements de type. Le jugement de type $\gamma \vdash x : \tau$ par exemple associe le type $\tau \in SC$ à la variable x, γ étant un environnement de typage faisant correspondre les variables x d'un programme à leurs types respectifs τ. L'environnement γ est initialisé en assignant à chaque variable globale x d'un programme un label de sécurité noté $\underline{x}$.

Le langage impératif (figure 2.5) utilisé par Volpano est constitué de **phrases** p, c'est-à-dire d'expressions e ou de commandes c. Les expressions x, l et n sont des métavariables qui représentent respectivement des identifiants de variables, des pointeurs et des entiers. La sémantique à grand pas de ce langage est présentée à la figure 2.7. Cette sémantique introduit une mémoire μ, définie comme une fonction évaluant la valeur des pointeurs du programme. Le domaine de définition de cette fonction est noté $dom(\mu)$. Aussi, le contenu d'un pointeur l est noté $\mu(l)$, alors que l'affectation de la valeur entière n au pointeur l est noté $\mu[l := n]$. Cette mémoire μ permet de définir des jugements d'évaluation tels $\mu \vdash e \Rightarrow n$ (l'évaluation d'une expression e fournit une valeur entière) ou $\mu \vdash c \Rightarrow \mu'$ (l'évaluation d'une commande c fournit une nouvelle mémoire μ').

L'auteur introduit aussi deux niveaux de typage, le typage de données notés τ qui représentent les classes de sécurité de l'ensemble SC, puis le typage des phrases noté ρ qui regroupent les types de données τ, les types d'expressions τ *var* et les types de commandes τ *cmd*. Si $\gamma \vdash x : \tau$ *var*, alors la variable x contient des données dont le niveau de sécurité est inférieur ou égale à τ selon la relation $\rightarrow$ définie sur SC. Si $\gamma \vdash c : \tau$ *cmd*, alors les variables affectées dans la commande c sont de niveau supérieur ou égale à τ. Une relation de sous-typage sur les types ρ est aussi définie, et notée $\subseteq$; Si $\tau \rightarrow \tau'$ où $(\tau, \tau') \in SC \times SC$, alors $\tau \subseteq \tau'$ et τ' *cmd* $\subseteq \tau$ *cmd*. Intuitivement, les variables de niveau de sécurité "bas" peuvent être forcées vers un niveau de sécurité "plus haut", alors que les commandes d'un niveau de sécurité "haut" peuvent être forcées vers un niveau de sécurité "plus bas".

Les règles de typages définies par Volpano sont présentées en figure 2.6. Ces règles garantissent que les flux explicites et implicites d'un programme respectent bien la politique de flux définie par $< SC, \rightarrow, \oplus >$. Les trois règles (INT), (VAR), (VARLOC) définissent les types de sécurité associés aux expressions de base ; entiers, variables et pointeurs. La règle (ARITH) stipule que la combinaison binaire des expressions e_1 et e_2 n'est possible que si ces expressions ont le même niveau de sécurité. Cette règle combinée à une des règles de sous-typage s'assure que le label de sécurité associé à l'expression $e_1 + e_2$ est au moins $\underline{e_1} \oplus \underline{e_2}$ puisque si $\underline{e_1} \neq \underline{e_2}$, alors il est possible de forcer les deux labels vers un niveau commun "plus haut" τ'' tel que $\tau' \rightarrow \tau''$ et $\tau \rightarrow \tau''$. La règle (R-VAL) permet la conversion du type τ *var* en τ. La règle (ASSIGN) garantit que les flux explicites

		(phrases)	p : :=	$e \mid c$
(types des données)	τ	(expressions)	e : :=	$x \mid l \mid n \mid e + e' \mid e - e' \mid e = e' \mid e \leq e'$
(types des phrases)	$\tau \mid \tau\ var \mid \tau\ cmd$	(commands)	c : :=	$e := e' \mid c; c' \mid if\ e\ then\ c\ else\ c' \mid$
				$while\ e\ do\ c \mid letvar\ x := e\ in\ c$

FIGURE 2.4 – Les différents types

FIGURE 2.5 – Langage impératif introduit par Volpano

(INT) $\lambda; \gamma \vdash n : \tau$

(VAR) $\lambda; \gamma \vdash x : \tau\ var$ if $\gamma(x) = \tau\ var$

(VARLOC) $\lambda; \gamma \vdash l : \tau\ var$ if $\lambda(l) = \tau$

(ARITH) $$\frac{\lambda; \gamma \vdash e : \tau, \lambda; \gamma \vdash e' : \tau}{\lambda; \gamma \vdash e + e' : \tau}$$

(R-VAL) $$\frac{\lambda; \gamma \vdash e : \tau\ var}{\lambda; \gamma \vdash e : \tau}$$

(ASSIGN) $$\frac{\lambda; \gamma \vdash e' : \tau}{\lambda; \gamma \vdash e := e' : \tau\ cmd}$$

(COMPOSE) $$\frac{\lambda; \gamma \vdash c : \tau\ cmd, \lambda; \gamma \vdash c' : \tau\ cmd}{\lambda; \gamma \vdash c; c' : \tau\ cmd}$$

(IF) $$\frac{\lambda; \gamma \vdash e : \tau, \lambda; \gamma \vdash c : \tau\ cmd, \lambda; \gamma \vdash c' : \tau\ cmd}{\mu \vdash e + e' \Rightarrow n + n'}$$

(WHILE) $$\frac{\lambda; \gamma \vdash e : \tau, \lambda; \gamma \vdash c : \tau\ cmd}{\lambda; \gamma \vdash while\ e\ do\ c : \tau\ cmd}$$

(LETVAR) $$\frac{\lambda; \gamma \vdash e : \tau, \lambda; \gamma[x : \tau\ var] \vdash c : \tau'\ cmd}{\mu \vdash e + e' \Rightarrow n + n'}$$

FIGURE 2.6 – Les règles du système de type de Volpano et Smith

$\mu \vdash n \Rightarrow n$ (BASE)

$\mu \vdash l \Rightarrow \mu(l)$ (CONTENTS)

$$\frac{\mu \vdash e \Rightarrow n, \mu \vdash e' \Rightarrow n'}{\mu \vdash e + e' \Rightarrow n + n'}\text{(ADD)}$$

$$\frac{\mu \vdash e \Rightarrow n, l \in dom(\mu)}{\mu \vdash l := e \Rightarrow \mu[l := n]}\text{(UPDATE)}$$

$$\frac{\mu \vdash c \Rightarrow \mu', \mu' \vdash c' \Rightarrow \mu''}{\mu \vdash c; c' \Rightarrow \mu''}\text{(SEQUENCE)}$$

$$\frac{\mu \vdash e \Rightarrow 1, \mu \vdash c \Rightarrow \mu'}{\mu \vdash if\ e\ then\ c\ else\ c' \Rightarrow \mu'}\text{(BRANCH)}$$

$$\frac{\mu \vdash e \Rightarrow 0, \mu \vdash c' \Rightarrow \mu'}{\mu \vdash if\ e\ then\ c\ else\ c' \Rightarrow \mu'}$$

$$\frac{\mu \vdash e \Rightarrow 0}{\mu \vdash while\ e\ do\ c \Rightarrow \mu}\text{(LOOP)}$$

$$\frac{\mu \vdash e \Rightarrow 1, \mu \vdash c \Rightarrow \mu', \mu' \vdash while\ e\ do\ c \Rightarrow \mu''}{\mu \vdash while\ e\ do\ c \Rightarrow \mu''}$$

$$\frac{\mu \vdash e \Rightarrow n, l \notin dom(\mu), \mu[l := n] \vdash [l/x]c \Rightarrow \mu'}{\mu \vdash letvar\ x := e\ in\ c\ \Rightarrow \mu' - l}\text{(BINDVAR)}$$

FIGURE 2.7 – Sémantique naturelle du language impératif

respectent la politique de flux. Elle spécifie que l'affectation de la valeur d'une variable e' à e n'est possible que si $\underline{e} = \underline{e'}$, tout en permettant cette affectation dans le cas général où $\underline{e'} \rightarrow \underline{e}$ grâce au sous-typage. La règle (COMPOSE) combinée au sous-typage de commandes, s'assure que le type associé à la composition de deux commandes c et c' est au plus égal à $c \otimes c'$. Les règles (IF) et (WHILE) garantissent que les flux implicites respectent la politique de flux. Elles s'assurent que les affectations de variables globales à l'intérieur de branchements ne concernent que des variables dont le niveau de sécurité est plus haut que celui de la variable de contrôle. Enfin, la règle (LETVAR) garantit que le flux d'information explicite dû à l'initialisation des variables locales ne viole pas la politique de flux. En effet, si une variable x est initialiée avec la valeur de l'expression e, la mémoire μ est mise à jour pour associer à x le label de e.

Jif [Mye99], une implémentation de ce système de type a été réalisée. Elle propose d'associer statiquement des labels de sécurité aux variables, pour le contrôle de flux au sein des programmes Java. Cependant, le caractère figé des labels de sécurité associés aux variables fait de ce système de type une analyse trop restrictive. Le programme $secret = public; public' = secret$ n'est pas typable par exemple, car cette analyse considère qu'il y a une fuite d'information suite à l'affectation de la variable *secret* à *public'*, alors même que *secret* contient une valeur publique. Cette analyse est dite

insensible au flux car elle ne permet pas la propagation des labels de sécurité. Une analyse statique plus permissive a été proposée dans [HS06]. Cette analyse permet le typage du programme ci-dessus car un label de sécurité bas est associé à la variable *secret* suite à l'affectation $secret = public$. Cette analyse est dite sensible au flux car elle permet la propagation des labels de sécurité. Cependant, ces deux analyses statiques cherchent à prouver que toutes les exécutions possibles d'un programme sont compatibles avec la politique de sécurité. Or il est parfois désirable de s'intéresser à la sécurité de l'exécution courante d'un programme seulement, sans prendre en compte toutes les exécutions possibles.

2.4.2 L'analyse dynamique

Une approche dynamique de suivi de flux d'information pour la détection d'intrusion a été présentée dans [HMZ+07] et [ZMB03]. Cette approche initialise la politique de flux présentée dans la section 2.3.3, en interprétant la politique de sécurité définie au niveau du système d'exploitation. Elle définit ainsi un ensemble de CCAL (définition 3) spécifiant les flux autorisés du système.

Cette approche se base sur la notion de contenu et conteneur telle qu'introduite dans la section 2.3.3. Chaque objet se voit alors associé à un ensemble de TAG de sécurité en lecture et en écriture. Le **TAG en lecture** T^R d'un objet O contenant initialement l'information o, est l'ensemble des CCAL où le contenu o apparait. Son **TAG en écriture** T^W est donc l'ensemble des CCAL où le conteneur O apparait. Un flux d'information d'un contenu source vers un conteneur cible est alors traduit par la propagation des restrictions du TAG en lecture du contenu vers les restrictions du TAG en lecture du conteneur.

L'exemple de la figure 2.8b illustre l'ensemble de CCAL interprétant la matrice d'accès discrétionnaire de la figure 2.8a. Les conteneurs des fichiers F_1 et F_2 sont notés C_1, C_2. Leurs contenus sont notés f_1, f_2. Il est à noter que l'interface de chaque utilisateur a été modélisée par un contenu (f_A, f_B), représentant l'information générée par un utilisateur et un conteneur (C_A, C_B), représentant l'information lue. Chaque conteneur est associé à ses TAG en lecture et écriture.

	F_1	F_2
Alice	lecture, écriture	lecture, écriture
Bob		lecture, écriture

(a) Exemple de matrice d'accès discrétionnaire

$$CCAL_A \quad (\{f_1, f_2, f_A\}, \{C_1, C_2, C_A\})$$
$$CCAL_B \quad (\{f_2, f_B\}, \{C_2, C_B\})$$

(b) Ensemble des CCAL

FIGURE 2.8

L'interprétation en termes de CCAL de la figure 2.8b interdit à l'utilisateur B d'accéder en lecture au contenu f_1. Les figures 2.9b et 2.9c illustrent l'évolution des objets suite à la copie du contenu du fichier F_1 dans F_2 puis la lecture de F_2 par l'utilisateur B. La copie du fichier F_1 vers F_2 génére un flux d'information du contenu f_1 vers le conteneur F_2. Ce flux d'information se traduit alors par la propagation des restrictions du TAG en lecture $T^R_{F_1} = \{CCAL_A\}$ du fichier F_1 vers les restrictions du TAG en lecture du fichier $F_2 = \{CCAL_A, CCAL_B\}$. Cette propagation est induite par l'intersection de ces deux tags : $T^R_{F_2} \cap T^R_{F_1} = \{CCAL_A\}$ (figure 2.9b). Ce flux d'information peut donc être noté :

$$((F_1, f_1, T^R_{F_1}, T^W_{F_1}),\ (F_2, f_2, T^R_{F_2}, T^W_{F_2})) \mapsto (F_2, f_2 \cup f_1, T^R_{F_2} \cap T^R_{F_1}, T^W_{F_2})$$

La pertinence et la fiabilité de ce modèle ont été prouvées grâce au théorème de détection :

Théorème 1. *(Théorème de détection) Une violation de la politique de sécurité est caracétérisée par :* $T^R \cap T^W = \varnothing$

Ainsi lors de la lecture du fichier F_2 par l'utilisateur Bob, un flux est généré du contenu f_2 vers le conteneur C_b (2.9c). Il est donc possible de détecter une violation de la politique de sécurité suite à la génération de ce flux.

	T^R	T^W
C_1	$CCAL_A$	$CCAL_A$
C_2	$CCAL_A, CCAL_B$	$CCAL_A, CCAL_B$
C_A	$CCAL_A$	$CCAL_A$
C_B	$CCAL_B$	$CCAL_B$

(a) Associations initiales

	T^R	T^W
C_1	$CCAL_A$	$CCAL_A$
C_2	$CCAL_A$	$CCAL_A, CCAL_B$
C_A	$CCAL_A$	$CCAL_A$
C_B	$CCAL_B$	$CCAL_B$

(b) L'évolution dûe au flux $f_1 \rightarrow C_2$

	C_L	C_E
C_1	$CCAL_A$	$CCAL_A$
C_2	$CCAL_A$	$CCAL_A, CCAL_B$
C_A	$CCAL_A$	$CCAL_A$
C_B	$\varnothing$	$CCAL_B$

(c) L'évolution dûe au flux $f_2 \rightarrow C_B$

FIGURE 2.9 – L'association conteneur, TAG en lecture et TAG en écriture

Cette approche a fait l'objet d'une implémentation [ZMB03, HMZ+07] sous Linux pour valider un prototype (Blare [2]) de système de détection d'intrusion reposant sur le suivi de flux d'information. L'inconvénient majeur de Blare est la sur-approximation des flux d'information générés par les applications, considérées comme des boîtes noires. Un prototype (JBlare) a été alors proposé [HMMT07] afin d'assurer le suivi d'information au niveau des applications Java en instrumant le *bytecode*. JBlare est basé sur le même concept de propagation des CCAL. Il coopère avec Blare afin d'assurer un suivi de flux d'information plus précis et limiter ainsi les faux-positifs.

2.4.3 L'analyse hybride

Russo et Sabelfeld [RS10] ont proposé un contrôleur sensible au flux d'information, combinant analyse statique et dynamique. Ce contrôleur hybride assure le suivi de flux d'information pour un langage impératif simple, étendu avec une instruction gérant des sorties au niveau de deux interfaces (affichage sur un écran, écriture sur le disque ...). La première interface est associée à un niveau de sécurité secret (noté H) et correspond à l'information pouvant être lue par des utilisateurs privilégiés, alors qu'un niveau de sécurité publique (noté L) est associé à la seconde. En effet, la sémantique du contrôleur est exprimée pour deux labels de sécurité tels que $L \rightarrow H$ (n'est autorisé qu'un flux d'information de *Low* vers $High$) pour simplifier. L'approche proposée constitue cependant un *framework* hybride de contrôle de flux, pouvant être généralisé à un modèle de politique en treillis fini, de cardinalité supérieure à deux. Par la suite, le modèle $< \{L, H\}, \rightarrow, \oplus >$ est considéré.

La sémantique du contrôleur de flux d'information 2.10 introduit deux environnements de typages Γ et Γ_s. L'environnement Γ associe à chaque variable du programme un label de sécurité.

2. http ://www.rennes.supelec.fr/blare/

L'analyse proposée étant une analyse sensible au flux, la valeur qu'associe Γ à chaque variable peut varier au cours de l'exécution. Le second environnement Γ_s est utilisé pour tenir compte des flux implicites. Il est initialisé à $[\epsilon]$ au début de l'exécution. À chaque fois qu'un branchement sur une variable de contrôle de niveau H est rencontré, les variables modifiées dans les branches non exécutée sont ajoutées à Γ_s, en y associant le label H. La génération de cet environnement empilé dans Γ_s, repose en général sur des annotations adjointes au programme durant l'analyse statique. Γ_s est aussi souvent appelé **contexte d'exécution**. En effet si $\Gamma_s \neq \epsilon$, alors l'exécution courante du programme génère des flux implicites qu'il faut prendre en compte. Il est à noté que si $\Gamma_s = [\epsilon]$, le niveau de sécurité associé au contexte d'exécution est L ($\underline{\Gamma_s} = L$). Si $\Gamma_s \neq [\epsilon]$, alors $\underline{\Gamma_s} = H$.

La sémantique du langage impératif utilisé est présentée figure 2.11. Les instructions de ce langage déclenche des événements internes notés α. Le contrôleur réagit alors via des événements γ. Ces événements synchronisent en fait le programme et le contrôleur de flux.

$$\text{(STOP)} \quad \langle \Gamma, \Gamma_s \rangle \xrightarrow{s} \langle \Gamma, \Gamma_s \rangle$$

$$\text{(LEVEL)} \quad \langle \Gamma, \Gamma_s \rangle \xrightarrow{a(x,e)} \langle \Gamma[x \mapsto lev(e,\Gamma) \oplus lev(\Gamma_s)], \Gamma_s \rangle$$

$$\text{(B-LOW)} \quad \frac{lev(e,\Gamma) = L}{\langle \Gamma, \epsilon \rangle \xrightarrow{b(e,c)} \langle \Gamma, \epsilon \rangle}$$

$$\text{(B-HIGH)} \quad \frac{lev(e,\Gamma) = H \vee \Gamma_s \neq \epsilon}{\langle \Gamma, \Gamma_s \rangle \xrightarrow{b(e,c)} \langle \Gamma, update_h(c) : \Gamma \rangle}$$

$$\text{(J-HIGH)} \quad \langle \Gamma, \Gamma' : \Gamma_s \rangle \xrightarrow{f} \langle \Gamma \oplus \Gamma', \Gamma_s \rangle$$

$$\text{(J-LOW)} \quad \langle \Gamma, \epsilon \rangle \xrightarrow{f} \langle \Gamma, \epsilon \rangle$$

FIGURE 2.10 – Sémantique du contrôleur

La figure 2.12 représente un programme P divulgant la valeur des variables *secret* et *secret'*. Ce programme est initialisée avec les valeurs $secret = 1$ et $secret' = 1$. Les variables secrètes reçoivent le label de sécurité H. Les variables x et y reçoivent le niveau L. L'évolution du contrôleur est représentée figure 2.13.

L'exécution du programme P suit le schéma suivant :

- 0 : Initialiement, l'environnement Γ associe chaque variable à son niveau de sécurité. Γ_s est initialisée à ϵ.
- 1 : l'instruction if déclenche l'événement interne $b(secret, skip)$. La variable *secret* étant une variable de niveau H, la règle $B\text{-}HIGH$ applique la fonction $update_H(skip)$. Γ_s empile alors l'ensemble vide $\{\}$ puisqu'aucune variable n'est modifiée par la branche non exécutée, c'est-à-dire *skip*.
- 2 : la seconde instruction if déclenche l'événement $b(secret', skip)$ puisque $\Gamma_s \neq \epsilon$. Le contrôleur exécute alors la fonction $update_H(y := 0)$ et donc Γ_s empile l'ensemble $\{y \mapsto H\}$. Il est à noter que même si la variable de contrôle de ce branchement est associée à un niveau L, la règle $B\text{-}HIGH$ aurait forcé l'exécution de la fonction $update_H$ puisque Γ_s associe au contexte d'exécution un niveau H pour empêcher tout flux implicite découlant du branchement de l'instruction 1.
- 3 : l'affectation de la variable x déclenche l'événement $a(x, 1)$. Le label $\Gamma(x)$ est alors mis à jour avec la valeur $lev(x,\Gamma) \oplus lev(\Gamma_s) = H$.
- 6 : l'événement interne f est déclenché puisque les deux branches de l'instruction 2 se rejoignent.

$$\text{(SKIP)} \quad \langle skip, m\rangle \xrightarrow{s} \langle stop, m\rangle$$

$$\text{(ASSIGN)} \quad \frac{m(e) = v}{\langle x := e, m\rangle \xrightarrow{a(x,e)} \langle stop, m[x \mapsto v]\rangle}$$

$$\text{(COMP)} \quad \frac{\langle c_1, m\rangle \xrightarrow{\alpha} \langle stop, m'\rangle}{\langle c_1; c_2, m\rangle \xrightarrow{\alpha} \langle c_2, m'\rangle}$$

$$(COMP') \quad \frac{\langle c_1, m\rangle \xrightarrow{\alpha} \langle c_1', m'\rangle, c_1' \neq stop}{\langle c_1; c_2, m\rangle \xrightarrow{\alpha} \langle c_1'; c_2, m'\rangle}$$

$$\text{(IF)} \quad \frac{m(e) \neq 0}{\langle if\ e\ then\ c_1\ else\ c_2, m\rangle \xrightarrow{b(e,c_2)} \langle c_1; end, m\rangle} \qquad \frac{m(e) = 0}{\langle if\ e\ then\ c_1\ else\ c_2, m\rangle \xrightarrow{b(e,c_1)} \langle c_2; end, m\rangle}$$

$$\text{(WHILE)} \quad \frac{m(e) \neq 0}{\langle while\ e\ do\ c\ , m\rangle \xrightarrow{b(e,skip)} \langle c; end; while\ e\ do\ c, m\rangle} \qquad \frac{m(e) = 0}{\langle while\ e\ do\ c\ , m\rangle \xrightarrow{b(e,skip)} \langle c; end; while\ e\ do\ c, m\rangle}$$

$$\text{(STOP)} \quad \langle end, m\rangle \xrightarrow{f} \langle stop, m\rangle$$

$$\text{(OUTPUT)} \quad \frac{m(e) = v}{\langle output_l(e), m\rangle \xrightarrow{o_l(e,v)} \langle stop, m\rangle}$$

FIGURE 2.11 – Sémantique du langage impératif et événements internes

```
1  if secret
2          if secret'
3                  x = 1
4          else
5                  y = 0
6          endif
7  else
8          skip
9  endif
10 output_L(x)
```

FIGURE 2.12 – Exemple de fuite d'information dans un programme

	α	Γ	Γ_s
0		$secret \mapsto H,\ secret' \mapsto H,\ x \mapsto L,\ y \mapsto L$	$[\epsilon]$
1	b($secret$, skip)	$secret \mapsto H,\ secret' \mapsto H,\ x \mapsto L,\ y \mapsto L$	$[\{\} : \epsilon]$
2	b($secret'$, y :=0)	$secret \mapsto H,\ secret' \mapsto H,\ x \mapsto L,\ y \mapsto L$	$[\{y \mapsto H\} : \{\} : \epsilon]$
3	a(x, 0)	$secret \mapsto H,\ secret' \mapsto H,\ x \mapsto H,\ y \mapsto L$	$[\{y \mapsto H\} : \{\} : \epsilon]$
6	f	$secret \mapsto H,\ secret' \mapsto H,\ x \mapsto H,\ y \mapsto H$	$[\{\} :\ \epsilon]$
9	f	$secret \mapsto H,\ secret' \mapsto H,\ x \mapsto H,\ y \mapsto H$	$[\epsilon]$

FIGURE 2.13 – Évolution de Γ et Γ_s suite à l'exécution de P

L'ensemble $\{y \mapsto H\}$ est dépilé, puis $\Gamma(y)$ est mise à jour ($J\text{-}HIGH$).

– 9 : l'événement interne f dépile simplement l'ensemble vide.

L'instruction 10 représente une violation de la confidentialié puisque la valeur de x peut divulguer la valeur des variables *secret* et *secret'*. En effet, le *framework* proposé considère que seules les interfaces de sortie sont responsables de la fuite d'information.

Russo et Sabelfeld ont démontré que le *framework* d'analyse hybride proposé garantissait la propriété de non-interférence insensible à la terminaison du programme 2. Cepedant, il s'avère que le suivi de flux d'information n'est pas une tâche aisée en pratique. Plusieurs difficultés se posent, notamment pour le suivi des flux implicites et la gestion des programmes concurrents.

Plusieurs travaux se sont intéressés à l'implémentation de mécanismes de contrôles de flux hybrides pour les programmes Java [NSCT07, CF07]. Les attaques contre ce langage qui est très souvent utilisé dans les applications Web, ont connu une nette hausse récemment. Elles représentaient 7% de l'ensemble des attaques en Septembre 2010 (contre 5% en Juillet 2010), ce qui en fait l'un des premiers vecteur d'attaques Web [3].

Ces mécanismes s'appuient sur une analyse statique du *bytecode* Java pour construire le graphe de flot de contrôle du programme (GFC) contrôlé. Un GFC est un graphe orienté, dont les noeuds représentent des blocs d'instructions et les arcs représentent les différents chemins que l'exécution peut suivre. Cette étape est nécessaire pour définir les variables dont les labels de sécurité doivent être mis à jour pour tenir compte des flux implicites. L'analyse statique est effectuée préalablement à l'exécution pour des raisons de performance. L'analyse dynamique propage ensuite les labels de sécurité des variables au cours de l'exécution.

L'une des difficultés de cette approche est de construire le GFC pour suivre de manière précise les flux implicites. Cependant, il ne suffit pas de prendre en compte seulement les structures conditionnelles de Java telles les instructions if et les boucles *while*. En effet, le transfert de flot de contrôle peut aussi être induit par des exceptions. L'exemple suivant $try\ \{int\ z = 1/(x-y);\}$ $catch\ (Exception\ e)\{print(\text{"}secret\ is\ true\text{"});\}$ illustre un exemple de programme pouvant divulguer de l'information par le biais d'une exception. Un attaquant peut déduire la valeur de la variable *secret* en ayant accès à la variable *public*. Cette exception est générée suite à l'évaluation d'une instruction faisant intervenir la variable y. Or le niveau de sécurité de y est égal à H puisqu'elle dépend de la variable *secret*. On peut donc considérer qu'il existe un flux implicite dont il faut tenir compte, dû à la génération d'une exception suite à l'évaluation d'une expression faisant intervenir la variable y. Les exceptions doivent alors être gérées de la même manière qu'une instruction conditionnelle.

3. http ://www.cisco.com/en/US/prod/collateral/vpndevc/3q10_cisco_threat.pdf

Cette exception étant déclarée et gérée explicitement par une instruction *try*{...} *catch*{...}, Il est possible de tenir compte d'une manière précise du transfert de flot de contrôle grâce à l'analyse statique du *bytecode*. *Nair et al.* [NSCT07] ainsi que Chandra et Franz [CF07] préconisent dans ce cas d'ajouter au GFC une arête reliant le début du bloc d'instructions *try*{...} au bloc d'instructions *catch*{...}.

Les exceptions peuvent être déclarées mais non gérées explicitement par la méthode qui les déclare. Il est alors beaucoup plus difficile, de déterminer quel bloc d'instructions est exécuté pour la gestion de l'erreur levée, au moment de l'analyse statique. Chandra et Franz ignorent ce type d'exceptions. Nair et al. adoptent par contre une approche différente. L'analyse statique essaie de déterminer quel type d'exception est levé, pour tenter de trouver l'instruction qui place dans la pile la référence à ce type d'exception. Cette référence correspond au bloc chargé de gérer cette exception. Une arête reliant l'instruction *throw* à ce bloc est ajoutée au GFC. En cas d'échec, une arête relie chaque instruction *throw* au bloc *exit* de la méthode dans laquelle elle se trouve. Il est à noter que les deux approches ignorent les exceptions non déclarées.

Chapitre 3

Contexte

Dans cette partie, nous allons tout d'abord rappeler les objectifs de ce stage. Nous présenterons ensuite l'outil Soot d'analyse statique de Bytecode, la représentation intermédiaire Jimple sur laquelle nous fondons notre analyse statique, puis deux exemples d'analyse de flux de données statique.

3.1 Objectifs

L'objectif de ce stage est de proposer un mécanisme hybride de suivi de flux d'information au sein des programmes Java, pour la détection d'intrusion paramétrée par la politique de sécurité. Ce mécanisme inclue une première analyse, statique, qui opère sur le Bytecode, puis une seconde analyse, dynamique, qui déduit les flux d'information générés par une exécution d'un programme grâce aux résultats de l'analyse statique.

L'analyse statique doit réaliser un profil pour chaque classe Java analysée puis le sauvegarder sous forme d'annotations dans le fichier contenant la classe correspondante. Ceci permettra à l'analyse dynamique de réutiliser ce profil lors de chaque exécution, sans avoir à relancer l'analyse statique pour le regénérer. Contrairement à Nair et al. [NSCT07] ainsi que Chandra et Franz [CF07] qui s'appuient sur l'analyse statique pour tenir compte des flux implicites seulement, nous souhaitons que notre analyse puisse aussi tenir compte des flux explicites pour améliorer les performances du suivi de flux d'information lors de l'analyse dynamique.

L'analyse dynamique doit s'appuyer sur les annotations générées par l'analyse statique pour déduire les flux d'information générés par le programme contrôlé, puis mettre à jour les labels de sécurité associés aux variables manipulées. Comme Nair et al [NSCT07], nous prenons le parti de modifier une JVM plutôt que d'instrumenter le Bytecode Java. Ceci permettra de limiter l'impact de l'analyse dynamique et faciliter son intégration avec le mécanisme Blare de suivi des flux d'information au niveau OS.

La première partie de ce stage, comptant pour le Master Recherche en Informatique de l'université de Rennes 1, doit déboucher sur le développement d'un modèle d'analyse statique pour le suivi de flux d'information. La conception et l'implémentation de l'analyse dynamique seront réalisées lors de la seconde partie de ce stage, comptant comme stage de fin d'étude pour l'école Supélec. Les travaux effectués concernant l'analyse statique sont présentés dans la partie 4. Les trois sections suivantes exposent les outils ainsi que le contexte de l'analyse statique développée.

3.2 Soot : analyse statique de Bytecode

Soot est un framework d'optimisation et d'analyse statique de Bytecode. Il a été réalisé afin de simplifier l'analyse et la transformation de Bytecode Java grâce à l'utilisation d'une représentation intermédiaire sous forme de code 3 adresses. Il a bénéficié d'un développement intensif depuis une dizaine d'année, ce qui a permis de l'enrichir de plusieurs analyses statiques intra-procédurales ou inter-procédurales, ainsi que différentes optimisations telles que l'élimination de code inutile ou la compaction de variables. Il intègre aussi différentes représentations intermédiaires du Bytecode telles que Jimple que nous présentons dans la section suivante. Enfin, ce framework intègre un plugin *eclipse* permettant la visualisation pas à pas de l'analyse statique implémentée, ce qui s'est avérée être d'une grande utilité lors de la validation de l'analyse statique développée et du débogage.

3.3 Jimple : représentation intermédiaire de Bytecode

L'analyse statique de Bytecode et plus généralement des langages de pile s'avérant très complexe, Vallée-Rai et Hendren [VRH98] ont eu recours au développement d'une représentation intermédiaire de Bytecode faciliant cette tâche. Cette forme intermédiaire, appelée Jimple, a les caractéristiques suivantes :

- elle est sous la forme d'un code 3 adresses ;
- elle n'utilise pas de pile ;
- ses variables sont explicitement déclarées et typées.

L'exemple suivant de la figure 3.1, tiré de [VRH98], est une séquence d'instructions Bytecode stockant dans la variable locale $l0$ la valeur de l'expression $l1/(5*(l2+l3))$. Cette exemple illustre la difficulté de l'analyse statique de Bytecode. En effet, avant de pouvoir déterminer quelle expression est utilisée à la ligne 10 par exemple, il faut d'abord examiner toutes les instructions précédentes en simulant leur effet sur la pile. Ceci peut s'avérer très coûteux, d'autant plus qu'il peut y avoir un très grand nombre d'instructions entre l'empilement d'un opérande sur la pile et son dépilement, c'est-à-dire son utilisation. Or en Jimple, il suffit de localiser les instructions qui manipulent la variable correspondante. De plus, le nombre très restreint d'instructions Jimple, comparé à la multitude d'instructions Bytecode, facilite grandement l'implémentation d'une analyse statique.

La génération de code Jimple à partir de Bytecode dans le framework Soot passe par 5 étapes :

1. génération de code Jimple simple non typé ;
2. une première phase d'optimisation ;
3. séparation des différentes variables pour préparer le typage ;
4. typage du code Jimple ;
5. une dernière phase d'optimisation.

Dans la suite de ce rapport, nous ne nous intéresserons pas aux étapes d'optimisation 2 et 5 que nous avons désactivées lors de l'analyse, car nous souhaitons garder une forme intermédiaire la plus fidèle possible au Bytecode initial, pour permettre la transposition des résultats de notre analyse statique.

La première étape de génération du code Jimple consiste à produire un code 3 adresses simple, non typé. Ceci est réalisé en associant à chaque opérande de pile une variable Jimple locale non typée, puis en transformant les références d'opérandes de piles implicites en références explicites de variables locales. La figure 3.1 illustre ces transformations.

	Instructions Bytecode	Simulation de pile	Instructions Jimple
1	iload 1	[l1]	$stack0 = l1
2	iconst 5	[l1, 5]	$stack1 = 5
3	iload 2	[l1, 5, l2]	$stack2 = l2
4	iload 3	[l1, 5, l2, l3]	$stack3 = l3
5	iadd	[l1, 5, l2+l3]	$stack2 = $stack2 + $stack3
6	imul	[l1, 5*(l2+l3)]	$stack1 = $stack1 * $stack2
7	idiv	[l1/ (5*(l2+l3))]	$stack0 = $stack0/$stack1
8	...	[l1/ (5*(l2+l3)), ...]	...
9	...	[l1/ (5*(l2+l3))]	...
10	istore 0	[]	l0 = $stack0

FIGURE 3.1 – Exemple de programme

	Instructions Bytecode		Instructions Jimple
1	ldc #10	1	$stack0 = "analyse_statique"
2	astore 1	2	l1 = $stack0
3	bipush 10	3	$stack0 = 10
4	istore 2	4	l2 = $stack0

FIGURE 3.2 – Exemple de code Jimple simple non typé

Le code Jimple obtenu suite à la première étape ne peut pas être typé. En effet, l'exemple de la figure 3.2 illustre le fait que la même variable Jimple puisse être utilisée avec différents types. En l'occurence, la variable $\$stack0$ est tantôt une chaîne de caractère, tantôt un entier. Ceci est possible car les spécifications de la JVM permettent l'affectation de types différents aux variables locales ainsi qu'aux opérandes de pile, du moment qu'aucun conflit n'existe.

L'idée pour s'affranchir de ce problème est alors de renommer ces variables pour éviter qu'elles soient réutilisées. Pour cela, le framework Soot utilise le concept de *webs* [Muc97], qui capture les différentes définitions possibles pour une variable ainsi que ses différentes utilisations possibles. Nous parlerons dans la suite de ce rapport d'une **variable définie** ou une **variable utilisée** par une instruction selon les définitions 5 et 6. Le concept de *webs* est formellement défini en 7.

Définition 5 (Une variable définie par une instruction)**.** Une instruction définit une variable si elle assigne une valeur à cette variable.

Définition 6 (Une variable utilisée par une instruction)**.** Une instruction utilise une variable si elle lit la valeur de cette variable.

Définition 7 (Un web)**.** Un web est un ensemble S de références, minimal et clos. Cet ensemble est clos dans le sens où pour chaque définition $d \in S$, toutes les utilisations possibles de d sont dans S. De même, pour chaque utilisation $u \in S$, toutes les définitions possibles de u sont dans S.

Un algorithme de construction des *webs* d'un programme est fourni dans [Muc97]. Considérons l'exemple illustré par la figure 3.3.

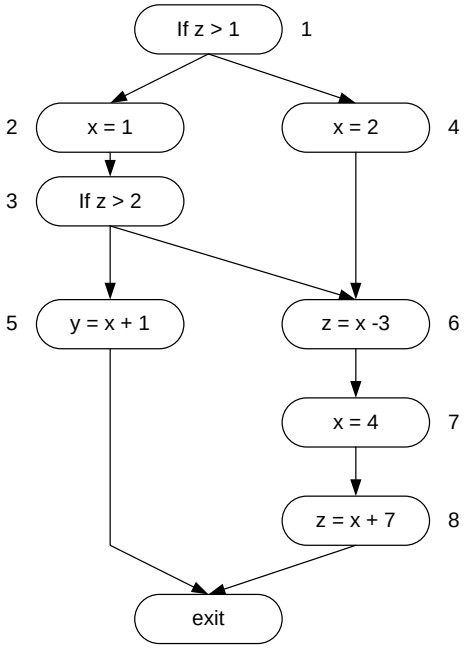

FIGURE 3.3 – Exemple de programme

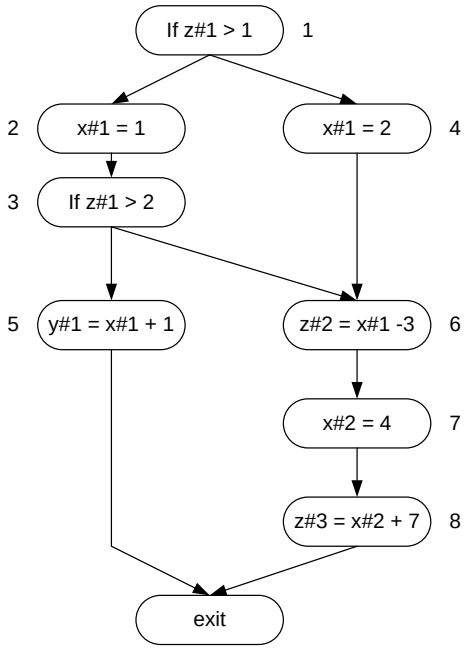

FIGURE 3.4 – Séparation des variables pour l'exemple 3.3

Définitions d	**Utilisations u**
x :2	x :5
x :4	x :6
x :7	x :8

FIGURE 3.5 – Les différentes définitions et utilisations de la variable x

La variable x est définie à l'instruction 2, 4 et 7, alors qu'elle est utilisée à l'instruction 5, 6 et 8 comme l'illustre la figure 3.5. Nous notons $x : l$ la définition ou l'utilisation de la variable x à l'instruction dont le label est l. Prenons la définition $x : 2$ par exemple. Cette définition a deux utilisations possibles, $x : 5$ et $x : 6$. Or l'utilisation $x : 5$ n'a qu'une seule définition possible qui est $x : 2$, alors que l'utilisation $x : 6$ a deux définitions possible ; $x : 2$ et $x : 4$. Aussi, la définition $x : 4$ n'a qu'une seule utilisation possible qui est $x : 6$. Nous avons ainsi construit notre premier ensemble minimal et clos, et donc le premier web. Le deuxième web est composé des définitions et utilisations restantes, c'est-à-dire $x : 7$ et $x : 8$. En effet, la définition $x : 7$ n'est utilisé qu'une seule fois par $x : 8$ et l'utilisation $x : 8$ n'est défini qu'une seule fois par $x : 7$. Donc cet ensemble est bien clos et minimal.

Une fois l'ensemble des *webs* construit, il est alors possible de séparer les différentes variables locales en associant chaque web à une seule et unique variable, ce qu'illustre la figure 3.4 pour le programme de la figure 3.3. Les variables locales ainsi obtenues peuvent donc être typées, à condition que le programme soit typable, ce qui est le cas de tous les programmes générés correctement par un compilateur Java.

L'étape numéro 4 de la production du code Jimple consiste à affecter des types aux différentes variables. L'approche adoptée dans *Soot* consiste à construire un ensemble de restrictions de types pour chaque variable locale en parcourant le code. Ces restrictions sont ensuite utilisées pour en déduire le type de la variable en question.

3.4 Analyse statique de flot de données

L'analyse statique de flot de données a pour objectif de définir, pour chacune des instructions d'un programme, une approximation d'un ensemble de propriétés concernant les données manipulées. Elle est réalisée en spécifiant un ensemble d'équations locales décrivant les propriétés dûes à chaque instruction, puis en résolvant des équations globales décrivant la propagation de ces propriétés au sein d'instructions successives. Elle nécessite la représentation du programme sous forme de graphe de flot de contrôle (CFG).

Nous présentons dans la suite l'analyse de vivacité des variables, un exemple classique d'analyse statique de flot de données, ainsi que l'analyse des post-dominateurs d'un noeud du CFG. Ces deux analyses, implémentées dans le framework Soot, sont utilisées lors de notre analyse statique de flux d'information.

3.4.1 Analyse de vivacité des variables

Cette analyse détermine pour un point donné du programme analysé, quelles variables sont définies et potentiellement utilisées plus tard. Les compilateurs s'en servent souvent lors de l'allocation de registres ou l'élimination de code inutile. Une définition de la propriété de vivacité d'une variable est donnée en 8.

Définition 8 (La vivacité d'une variable)**.** Une variable x est vive dans une instruction i si l'une des conditions suivantes est réalisée :

1. x est utilisée dans l'instruction i ;
2. x est vive après l'exécution de i, sans être redéfinie par i.

Dans l'exemple de la figure 3.6, la variable y est utilisée à la première instruction, mais n'est plus utilisée plus tard. Donc la variable y est vive à la première instruction seulement. Notons que cette variable doit être définie bien avant ce bloc d'instruction. La variable x, par contre, est définie à la première instruction, puis est utilisée à la dernière instruction. Or il existe un chemin d'exécution où x n'est pas redéfinie entre la première et la dernière instruction, elle est donc vive pour tout ce bloc d'instructions. Quant à la variable z, elle n'est jamais utilisée, bien que définie à la deuxième et quatrième instruction. Elle n'est donc vive à aucune instruction.

L'analyse de vivacité peut être formalisée en analyse de flot de données en arrière (*backward analysis*) : le parcours du CFG débute par le point de sortie du programme et remonte dans le sens contraire au chemin d'exécution. Cette analyse associe deux ensembles à chaque instruction i du programme, $genSet(i)$ et $killSet(i)$, définis par les équations locales 3.1. $GenSet(i)$ est l'ensemble des variables utilisées par i, alors que $KillSet(i)$ est l'ensemble des variables définies par i.

$$\begin{aligned} GenSet(i) &= \text{ensemble de variables utilisées par } i \\ KillSet(i) &= \text{ ensemble variables définies par } i \end{aligned} \tag{3.1}$$

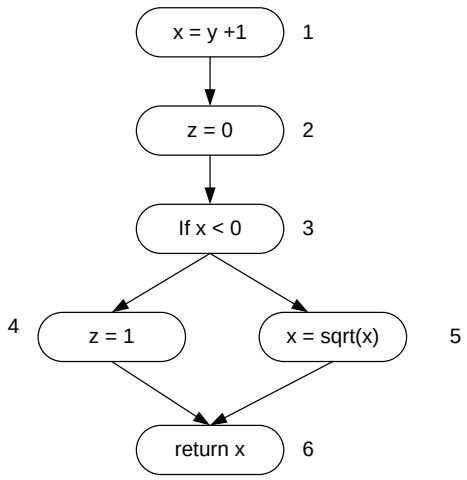

FIGURE 3.6 – Exemple de programme et le résulat de l'analyse de vivacité de ses variables

L'analyse de vivacité résout ensuite pour chaque instruction i, les équations globales 3.2 des ensemble $inSet(i)$ et $outSet(i)$.

$$\begin{aligned} OutSet(i) &= \cup_{s:succ(i)} InSet(s) \\ InSet(i) &= GenSet(i) \cup (OutSet(i) \setminus KillSet(i)) \end{aligned} \tag{3.2}$$

L'ensemble $OutSet(i)$ représente les variables vives après l'exécution de l'intruction i. On considère par définition, qu'une variable est vive après l'instruction i si elle est vive dans au moins un des successeurs de i car on cherche à déterminer toutes les variables potentiellement vives, d'où l'utilisation de l'union. Si l'on cherchait à déterminer les variables sûrement vives après une instruction donnée, on aurait plutôt utilisé l'intersection des ensembles $InSet(s)$, où s est un successeur de l'instruction i. Ce choix est très important et dépend fortement de l'utilisation que nous voulons faire des résultats de cette analyse. Si cette analyse est utilisée pour l'élimination de code inutile, nous choisirons de faire une sur-approximation des variables vives car nous voudrons absolument éviter de supprimer des définitions de variables qui pourraient être utilisées plus tard. Nous opterons alors pour l'union -comme nous l'avons fait dans cet exemple- plutôt que l'intersection.

L'ensemble $InSet(i)$ par contre représente les variables vives avant l'exécution de l'instruction i. L'analyse de flux de données résout ces équations grâce au parcous du CFG et la propagation des ensembles $inSet$ et $outSet$, jusqu'à trouver un point fixe et donc une solution à ces équations globales.

La figure 3.7 illustre le résultat de cet analyse sur l'exemple de la figure 3.6 :

3.4.2 Post-dominateur dans un CFG

Dans un graphe de flot de contrôle, un noeud m post-domine un noeud n si tous les chemins d'exécution de n vers le point de sortie passe par le noeud m. Cette relation définit un ordre partiel pour les noeud du CFG. Dans l'exemple de la figure 3.3, les noeuds 6, 7 et 8 post-dominent le noeud 4. Par contre seul le point de sortie $exit$ post-domine les noeuds 1, 2 et 3.

On appelle aussi un post-dominateur immédiat du noeud n l'unique noeud qui est différent de n, qui post-domine n et qui est post-dominé par tous les autres post-dominateurs de n. Pour une instruction conditionnelle, le post-dominateur immédiat est le point de jonction de toutes les branches de cette instruction. Le noeud *exit* est par exemple le post-dominateur immédiat du noeud 1 dans la figure 3.3.

Les post-dominateurs des noeuds d'un CFG peuvent être déterminés grâce à une analyse de flot de données en arrière. Les équations de cette analyse sont fournies dans la figure 3.3. Un exemple de cette analyse est aussi déroulé figure 3.8.

$$\begin{aligned} OutSet(i) &= \cap_{s:succ(i)} InSet(s) \\ InSet(i) &= outSet(i) \cup \{i\} \end{aligned} \tag{3.3}$$

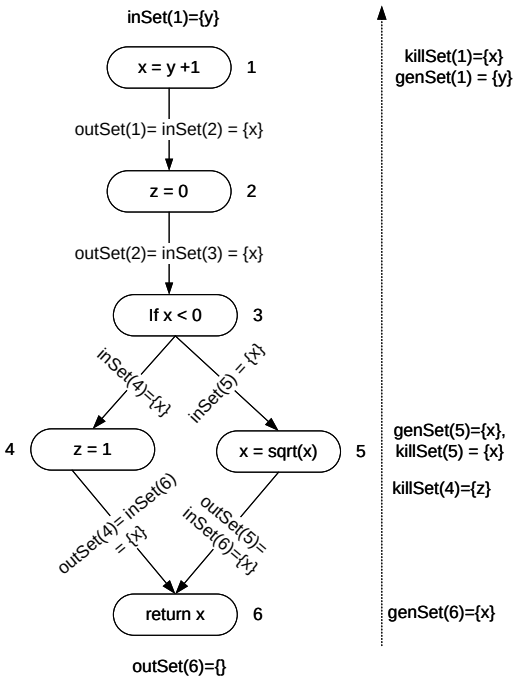

FIGURE 3.7 – Analyse de vivacité sur l'exemple figure 3.6

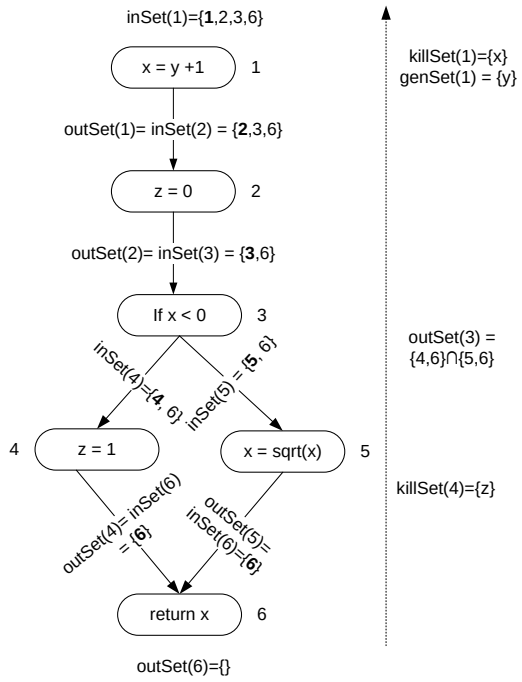

FIGURE 3.8 – Analyse des post-dominateur sur l'exemple de la figure 3.6

Chapitre 4

Travaux réalisés

4.1 Introduction

L'analyse statique que nous souhaitons réaliser est une analyse intra-méthode. En effet, pour un programme Java, il est souvent impossible de déterminer, avec précision, quelle méthode est appelée lors de l'exécution grâce à une analyse statique. Une analyse inter-procédurale nécessiterait alors de faire de grossières approximations. Notre analyse construit construit donc un profil pour chacune des méthodes appartenant aux classes Java analysées. Ce profil est ensuite utilisé pour annoter la méthode correspondante.

Cette analyse statique est réalisée sur la représentation intermédiaire Jimple des méthodes. Elle fait l'hypothèse que la JVM modifiée, réalisant l'analyse dynamique, soit capable d'affecter des labels de sécurité aux variables manipulées, que ces variables soient des paramètres de méthodes, opérandes de pile, objets ou attributs d'objets. De plus, cette JVM est capable de charger les annotations d'une méthode avant son exécution et de les interpréter en termes de flux d'information et de points de contrôle, qui pointent tous deux vers des instructions Bytecode de la méthode analysée. Ces points de contrôle correspondent à un ensemble de compteurs ordinaux d'instructions Bytecode pour lesquelles il est nécessaire que la JVM effectue certaines actions afin de suivre les flux d'information, avant de pouvoir poursuivre l'exécution à l'instruction suivante. Ces actions sont explicitées dans la suite de ce rapport. Notons que l'analyse que nous proposons n'est pas une analyse à fil sécurisé car nous raisonnons sur une exécution séquentielle des méthodes analysées.

4.2 Suivi des flux explicites

Un flux d'information explicite de y vers x est généré à chaque fois que le contenu d'une variable y ou une information dérivée de ce contenu est transféré explicitement dans la variable x. Dans le cas d'une affectation $x = y + z$ par exemple, il y a un flux explicite des variables y et z vers la variable x. Nous disons dans la suite de ce rapport que x **dépend** de y et z. Aussi, l'ensemble $\{y, z\}$ des variables dont x dépend est appelé l'**ensemble de dépendance** de x.

4.2.1 Description

L'analyse proposée pour permettre le suivi des flux explicites est une analyse de flux de données en avant (*forward analysis*). Le parcours du CFG débute par les points d'entrée de la méthode,

puis continue dans le sens du chemin d'exécution. Nous souhaitons tenir compte avec précision des flux explicites grâce à cette analyse. Cependant, comme nous l'avons discuté dans la section 3.4.1, la perte d'information est inhérente à l'analyse statique puisque nous devons choisir entre une sur-approximation ou une sous-approximation des flux générés par deux chemins d'exécution différents qui se rejoignent. Ceci est illustré par la figure 4.1.

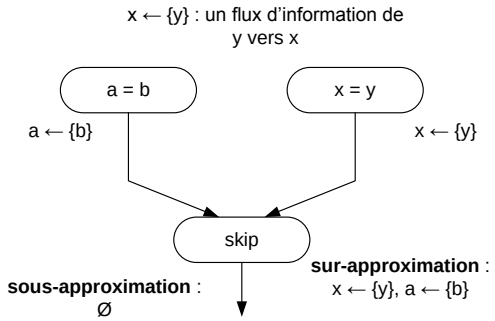

FIGURE 4.1 – Approximation dûe à l'analyse statique

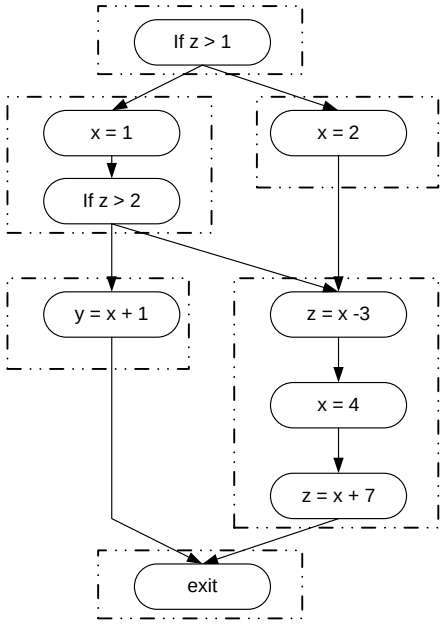

FIGURE 4.2 – Exemple de découpage d'une méthode en blocs de base

Nous nous affranchissons de ce problème en découpant la méthode analysée en blocs de base et en restreignant l'analyse statique à chacun de ces blocs. **Un bloc de base** est une séquence d'instruction, ayant un seul point d'entrée (la première instruction) et un seul point de sortie (la dernière instruction) [All70]. Ceci veut dire qu'à chaque fois que la première instruction d'un bloc est exécutée, les instructions suivantes de ce bloc sont exécutées dans l'ordre et une seule fois. Un bloc de base ne contient donc ni des boucles ni des branchements. Il peut biensûr avoir plusieurs blocs successeurs ou prédécesseurs. Il peut aussi être lui même un bloc successeur dans le cas des boucles. La figure 4.2 illustre un exemple de découpage d'une méthode en blocs de base.

4.2.2 Présentation de l'analyse statique des flux explicites

L'analyse statique de flot de données proposée est réalisée pour chacun des blocs de base de la méthode analysée. Ces analyses étant indépendantes les unes des autres, l'analyse dynamique se charge de mettre bout à bout les résultats obtenus pour chaque bloc de base. En effet, à la dernière instruction de chaque bloc, l'analyse statique force la JVM modifiée à mettre à jour les labels de sécurité associés aux variables manipulées dans ce bloc, avant de poursuivre l'exécution du bloc successeur. Nous appellerons l'action de ce point de contrôle **save_labels()**. Dans la suite, nous formaliserons l'analyse statique des flux explicites pour un bloc de base donné. Nous ne considérons

dans cette sous-section que les flux générés par des types primitifs Java puisque nous discuterons les références dans la sous-section 4.2.3.

L'analyse proposée associe deux ensembles d'associations à chaque instruction i du bloc de base Jimple analysé, $inSet(i)$ et $outSet(i)$. Ces ensembles associe une variable à son ensemble de dépendance. L'ensemble associée à une variable v (que nous noterons $inSet(i)[v]$ ou $outSet(i)[v]$) est l'ensemble des variables dont v dépend explicitement. Cela signifie qu'il y a un flux d'information explicite à partir de toutes les variables appartenant à l'ensemble $inSet(i)[v]$ (respectivement $outSet(i)[v]$) vers la variable v. L'ensemble $inSet(i)[v]$ est l'ensemble des flux d'information explicites générés avant l'exécution de l'instruction i, alors que l'ensemble $outSet(i)[v]$ représente les flux explicites générés après l'exécution de l'instruction i.

Les équations de cette analyse de flux de données en avant sont explicitées en 4.1. L'ensemble d'associations $inSet$ du point d'entrée du bloc de base **est initialisé** à l'ensembles d'associations vide $\{\}$.

$$\begin{aligned} inSet(i) &= outSet(pred(i)) \\ outSet(i) &= genSet(i) \bigcup inSet(i) \\ genSet(i) &= \begin{cases} \{v \leftarrow D_v\} \text{ si } i \text{ définit une variable } v.\ D_v \text{ est l'ensemble} \\ \qquad \text{des variables utilisées pour définir } v \\ \{\} \text{ sinon} \end{cases} \end{aligned} \tag{4.1}$$

La première équation de 4.1 copie dans l'ensemble d'associations $inSet$ de l'instruction i l'ensemble d'associations $outSet$ de son prédécesseur. La seconde équation introduit un ensemble d'associations $genSet(i)$ ainsi qu'un opérateur $\bigcup$.

L'ensemble d'associations $genSet(i)$ correspond aux flux d'information explicites générés par l'instruction i. Il associe la variable éventuellement définie par l'instruction i à son ensemble de dépendances explicites, c'est-à-dire les variables utilisées par l'instruction i.

L'opérateur $\bigcup$ est défini pour deux ensembles d'associations et rend un ensemble d'associations. Cet opérateur n'est pas commutatif car **il résout les dépendances des variables utilisées** dans une instruction en fonction des flux générés par les instructions qui la précèdent. Il est défini formellement en 9.

Définition 9 (L'opérateur $\bigcup$)**.** Soient V et W deux ensembles de variables.

Soient R et S deux ensembles d'associations tels que :

$R = \cup_{v \in V}\{v \leftarrow R[v]\}$ et $S = \cup_{v \in W}\{v \leftarrow S[v]\}$. $R[v]$ ou $S[v]$ est l'ensemble de dépendances de la variable v.

L'opérateur $\bigcup$ est alors défini comme suit :

$$\begin{aligned} R \bigcup S &= \left(\cup_{v \in V} \{v \leftarrow R[v]\} \right) \bigcup \left(\cup_{v \in W} \{v \leftarrow S[v]\} \right) \\ &= \left(\cup_{v \in V} \{v \leftarrow f_S(R[v])\} \right) \cup \left(\cup_{v \in W} \{v \leftarrow S[v]\} \right) \end{aligned}$$

où f_S est défini comme suit :

$f_S(R[v]) = (\cup_{x \in R[v] \cap W} S[v]) \cup (\cup_{x \in R[v] \setminus W} \{x\})$

L'exemple de la figure 4.3 illustre l'analyse de flux de données selon la formalisation que nous avons proposées précédemment. Considérons l'instruction numéro 3 de cet exemple. Cette instruction définit la variable $\$stack0\#2$ en utilisant les variables $\$stack0$ et $\$stack1$. L'ensemble d'associations $outSet(3)$ va donc prendre en compte ces flux d'information explicites, tout en resolvant les

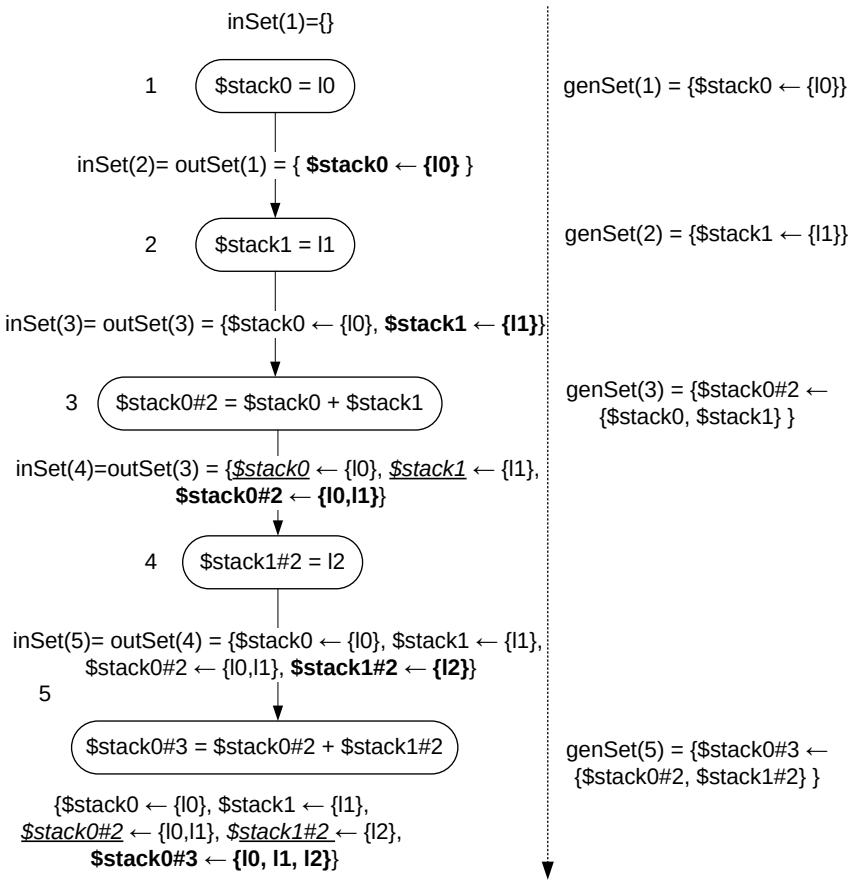

FIGURE 4.3 – Exemple de l'analyse de flux de données

dépendances présentes dans $inSet(3)$, des deux variables $\$stack0$ et $\$stack1$:

$$\begin{aligned} genSet(3) &= \{\$stack0\#2 \leftarrow \{\$stack0, \$stack1\}\} \\ inSet(3) &= \{\$stack0 \leftarrow \{l0\}, \$stack1 \leftarrow \{l1\}\} \\ outSet(3) &= \{\$stack0 \leftarrow \{l0\}, \$stack1 \leftarrow \{l1\}, \$stack0\#2 \leftarrow \{l1, l2\}\} \end{aligned}$$

Nous pouvons aussi remarquer grâce à l'exemple de la figure 4.3, que les ensemble d'associations $inSet$ et $outSet$ vont très vite exploser en taille. En effet, du fait de l'utilisation des $webs$ pour typer le code Jimple, un très grand nombre de variables est généré, alors que ces variables peuvent correspondre en réalité à une seule et même position sur la pile. L'opérande de pile $\$stack0$ par exemple, empilé à la position 0 suite à l'exécution de l'instruction numéro 1, est dépilé après l'exécution de l'instruction numéro 3. Il est alors remplacé par $\$stack0\#2$, sa somme avec $\$stack1$. Donc cette variable $\$stack0$ ne sera vraisemblablement plus utilisée après l'exécution de l'instruction numéro 3. Pourtant, nous propageons pour l'instant ses dépendances, jusqu'à la fin du bloc analysé.

Nous adressons ce problème grâce à l'analyse de vivacité des variables. En effet, si une variable n'est plus vive dans une instruction i, c'est-à-dire qu'elle n'est plus utilisée dans la suite de la méthode et si cette variable est présente dans l'ensemble $inSet(i)$, alors nous la supprimons de l'en-

semble d'associations *outSet* car nous n'avons plus besoin de garder l'ensemble de ses dépendances. Ce raisonnement n'est valable que pour les variables locales et les opérandes de piles de type primitif car ils ont une portée limitée à la méthode. S'ils ne sont pas utilisées à partir d'une certaine instruction, on peut oublier leurs dépendances puisque nous n'en aurons plus besoin par la suite.

Nous introduisons pour cela un ensemble de variables $killSet(i)$ pour chaque instruction i, ainsi qu'un opérateur $\diagdown$ défini en 10. L'ensemble de variables $killSet(i)$ permet de supprimer les dépendances des variables qui sont présentes dans l'ensemble d'associations $inSet(i)$ et qui ne sont pas vives dans l'instruction i. L'opérateur $\diagdown$ est donc défini pour un ensemble d'associations et un ensembles de variables.

Définition 10 (L'opérateur $\diagdown$)**.** Soient V et W des ensemble de variables.
Soit R un ensemble d'associations, tel que $R = \cup_{v \in V}\{v \leftarrow R[v]\}$.
L'opérateur $\diagdown$ est alors défini comme suit :
$R \diagdown W = \cup_{v \in V \setminus W}\{v \leftarrow R[v]\}$ où $V \setminus W$ est la différence ensembliste de V et W.

Les équations de la nouvelle analyse proposée sont explicitées en 4.2. Ces équations introduisent l'opérateur $\bigcup_i$ paramétré par l'instruction i et défini en 11. Le fonctionnement de cet opérateur est le même que celui présenté en 9, sauf qu'il résout les dépendances de associations de $genSet(i)$ avant la suppression des variables qui ne sont plus vives. Il résout donc ces dépendances par rapport à l'ensemble $inSet(i)$.

$$
\begin{aligned}
inSet(i) &= outSet(pred(i)) \\
outSet(i) &= genSet(i) \bigcup_i (inSet(i) \diagdown killSet(i)) \\
genSet(i) &= \begin{cases} \{v \leftarrow D_v\} \text{ si } i \text{ définit une variable } v.\ D_v \text{ est l'ensemble} \\ \qquad \text{des variables utilisées pour définir } v \\ \{\} \text{ sinon} \end{cases} \\
killSet(i) &= \text{ l'ensemble des variables présentes dans inSet(i) et qui ne sont plus vives dans } i
\end{aligned}
\tag{4.2}
$$

Définition 11 (L'opérateur $\bigcup_i$)**.** Soient V, W et Z trois ensembles de variables.
Soient R et S deux ensembles d'associations tels que :
$R = \cup_{v \in V}\{v \leftarrow R[v]\}$ et $S = \cup_{v \in W}\{v \leftarrow S[v]\}$.
Soit l'ensemble d'association $inSet(i) = \cup_{v \in Z}\{v \leftarrow inSet(i)[v]\}$ associé à l'instruction i.
L'opérateur $\bigcup_i$ est alors défini comme suit :

$$
\begin{aligned}
R \bigcup_i S &= \left(\cup_{v \in V} \{v \leftarrow R[v]\} \right) \bigcup_i \left(\cup_{v \in W} \{v \leftarrow S[v]\} \right) \\
&= \left(\cup_{v \in V} \{v \leftarrow f_{inSet(i)}(R[v])\} \right) \cup \left(\cup_{v \in W} \{v \leftarrow S[v]\} \right)
\end{aligned}
$$

où f_S est défini comme suit : $f_{inSet(i)}(R[v]) = (\cup_{x \in R[v] \cap Z} inSet(i)[v]) \cup (\cup_{x \in R[v] \setminus Z}\{x\})$

Le résultat de cette nouvelle analyse sur l'exemple 4.3 est illustré par la figure 4.4. Les variables supprimées grâce à l'ensemble *killSet* sont barré d'un trait dans cet exemple. Ce raisonnement concernant la vivacité des variables n'est par contre pas valable pour les objets qui peuvent avoir

une portée globale. En effet, comme nous allons le voir dans la section 4.2.3, l'analyse proposée précédemment est modifiée pour prendre en compte la particularité des objets ou des références en général. Notons que l'analyse de vivacité est réalisée sur l'ensemble de la méthode et pas seulement sur le bloc de base, contrairement à l'analyse statique des flux explicites.

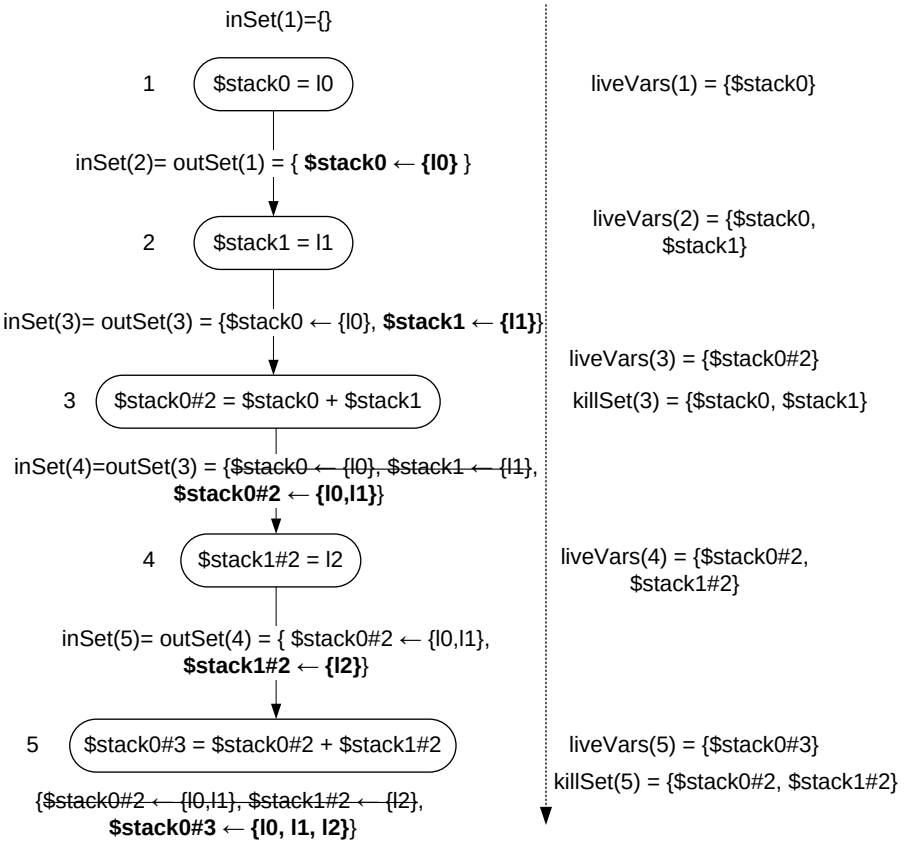

FIGURE 4.4 – Exemple de l'analyse de flux de données

Enfin, à la fin de chaque bloc de base, l'analyse statique marque la dernière instruction avec un point de contrôle **save_labels()** forçant la JVM modifiée à mettre à jour les labels de sécurité des variables présentes dans l'ensemble d'associations *outSet*. Ces variables vives donc, peuvent être des opérandes de piles ou des variables locales.

Pour permettre la mise à jour des labels de sécurité, l'analyse dynamique doit définir un ensemble de labels SC ainsi qu'une relation $\oplus : SC \times SC \mapsto SC$. Cette relation $\oplus$ dépend du modèle de politique de flux choisi. Dans le cas d'une politique de flux en treillis (définition 4) par exemple, $l_1 \oplus l_2$ sera la borne supérieur minimal des deux labels de sécurité l_1 et l_2. Les flux d'information $a \leftarrow \{x, y, z\}$ définira alors le label de sécurité $\underline{a}$ pour la variable a, tel que $\underline{a} = \underline{x} \oplus (\underline{y} \oplus \underline{z})$ où $\underline{x}$, $\underline{y}$ et $\underline{z}$ sont respectivement les labels de sécurité associés aux variables x, y et z.

4.2.3 Les références

La manipulation des références peut poser problème lors de notre analyse statique. En effet, considérons l'exemple présenté figure 4.5. Cet exemple stocke l'attribut statique $Test.n$ dans la variable locale à la position numéro 1. Il réinitialise ensuite cet attribut en lui affectant la valeur 0. Le résultat de l'analyse statique proposée en 4.1 pour cet exemple est illustré par la figure 4.6. Rappelons que la variable $stack0$ par exemple a été supprimée de $outSet(3)$ car cette variable n'est plus utilisée par la suite. Nous considérons par contre dans cette exemple que les variables $l1$, et $TEST.n$ sont toujours vives à l'instruction 4.

	Instructions Bytecode	Instructions Jimple
1	`getstatic TEST.n : int`	`$stack0 = <TEST: int n>`
2	`istore 1`	`l1 = $stack0`
3	`iconst 0`	`$stack0#2 = 0`
4	`putstatic TEST.n : int`	`<TEST: int n> = $stack0#2`

FIGURE 4.5 – Cas particulier de la manipulation des références

$$
\begin{aligned}
outSet(1) &= \{\$stack0 \leftarrow \{TEST.n\}\} \\
outSet(2) &= \{\$stack0 \leftarrow \{TEST.n\}, \{l1 \leftarrow \{TEST.n\}\}\} \\
outSet(3) &= \{\{l1 \leftarrow \{TEST.n\}\}, \$stack0\#2 \leftarrow \{\}\} \\
outSet(4) &= \{\{l1 \leftarrow \{TEST.n\}\}, TEST.n \leftarrow \{\}\}
\end{aligned}
$$

FIGURE 4.6 – Résultat de l'analyse statique de l'exemple figure 4.5

Le problème soulevé par cet exemple, est que la variable $l1$ dépend selon notre analyse statique de l'attribut statique $TEST.n$, alors que $TEST.n$ n'a aucune dépendance car une valeur constante lui a été assignée. Nous pourrions penser alors que la variable locale $l1$ n'a aucune dépendance elle non plus, alors qu'il aurait fallu la mettre à jour avec le label de sécurité de $TEST.n$ avant que ce dernier ne soit redéfini. Une solution naïve consisterait à ajouter un point de contrôle avant la modification de l'attribut $TEST.n$, pour mettre à jour les labels de sécurité de toutes les variables qui en dépendent. Toutefois, nous allons voir grâce à l'exemple de la figure 4.7 que ce problème est plus compliqué que cela, notamment à cause des alias.

Cet exemple stocke dans la variable locale $l2$ une deuxième référence du tableau pointé par la variable locale $l1$. Il modifie ensuite deux fois l'élément se trouvant à l'index 1 de ce tableau. L'analyse statique de flux de donnée, telle que nous l'avons proposée est incapable de détecter que le même élément du tableau est modifié à l'instruction 10 et à l'instruction 14. La solution naïve qui consisterait à mettre à jour les labels de sécurité des variables qui dépendent d'un attribut d'objet ou d'un élément de tableau, avant que ce dernier ne soit modifié, ne pourrait marcher que dans le cas où une analyse d'alias serait effectuée au préalable. De plus, cette analyse d'alias devrait être inter-procédural puisque les paramètres d'une méthode peuvent très bien être des alias pointant vers le même objet, chose qu'on ne pourrait détecter en se restreignant à une analyse intra-méthode.

```
1  byte $stack1, $stack1#2
2  int  l3, l4, $stack2, $stack2#2
3  int[] l1, l2, $stack0, $stack0#2, $stack0#3
4
5  $stack0 = l1;
6  l2 = $stack0;
7  $stack0#2 = l1;
8  $stack1 = 1;
9  $stack2 = l3;
10 $stack0#2[$stack1] = $stack2;
11 $stack0#3 = l2;
12 $stack1#2 = 1;
13 $stack2#2 = l4;
14 $stack0#3[$stack1#2] = $stack2#2;
```

FIGURE 4.7 – Cas particulier de la manipulation des références

Nous avons décidé de faire une sur-approximation grossière, en considérant que toutes les références d'objets sont des alias du même objet. Nous considérons aussi que toutes les références de tableaux sont des alias du même tableau. Ainsi, à chaque fois qu'un attribut d'objet ou qu'un élément d'un tableau est modifié, l'analyse statique ajoute un point de contrôle. Dans le cas de la modification d'un attribut d'objet, le point de contrôle force la JVM à mettre à jour les labels de sécurité de toutes les variables qui dépendent d'un attribut d'objet, avant de poursuivre l'exécution. Dans le cas de la modification d'un tableau par contre, le point de contrôle force la JVM à mettre à jour les labels de sécurité de toutes les variables qui dépendent d'un élément de tableau. L'analyse statique joint alors à ce point de contrôle une action **save_labels(R)** comme l'illustre la figure 4.8, où R est un tableau associatif exprimant les dépendances des variables à mettre à jour. Ces variables sont ensuite ajoutées à l'ensemble *killSet* pour réinitialiser les dépendances pris en compte par l'analyse dynamique.

Cette approximation grossière pourrait forcer la JVM modifiée à mettre à jour assez souvent les labels de sécurité des variables pour certains programmes. En effet, nous pouvons imaginer une méthode effectuant un calcul complexe qui ne fait intervenir que des attributs d'objets. Dans ce cas là, les performances auxquelles nous pouvons nous attendre en seront nettement dégradées. Cependant, nous pensons que les pratiques de programmation en Java privilégient au maximum l'usage de variables locales et surtout d'opérandes de pile pour des raisons d'optimisation et de performance, plutôt que l'usage intensif des attributs d'objets.

4.2.4 Les appels de méthodes

D'après les spécifications de la JVM[1], lors de l'appel d'une méthode non statique sur une instance d'objet O, la méthode appelante empile la référence de l'instance O, puis les arguments de la méthode appelée. Suite à l'invocation de cette dernière, la JVM dépile la référence de l'instance O ainsi que les arguments de la méthode appelée, puis crée une nouvelle *frame* pour cette dernière. La

1. http ://java.sun.com/docs/books/jls/second_edition/html/jTOC.doc.html

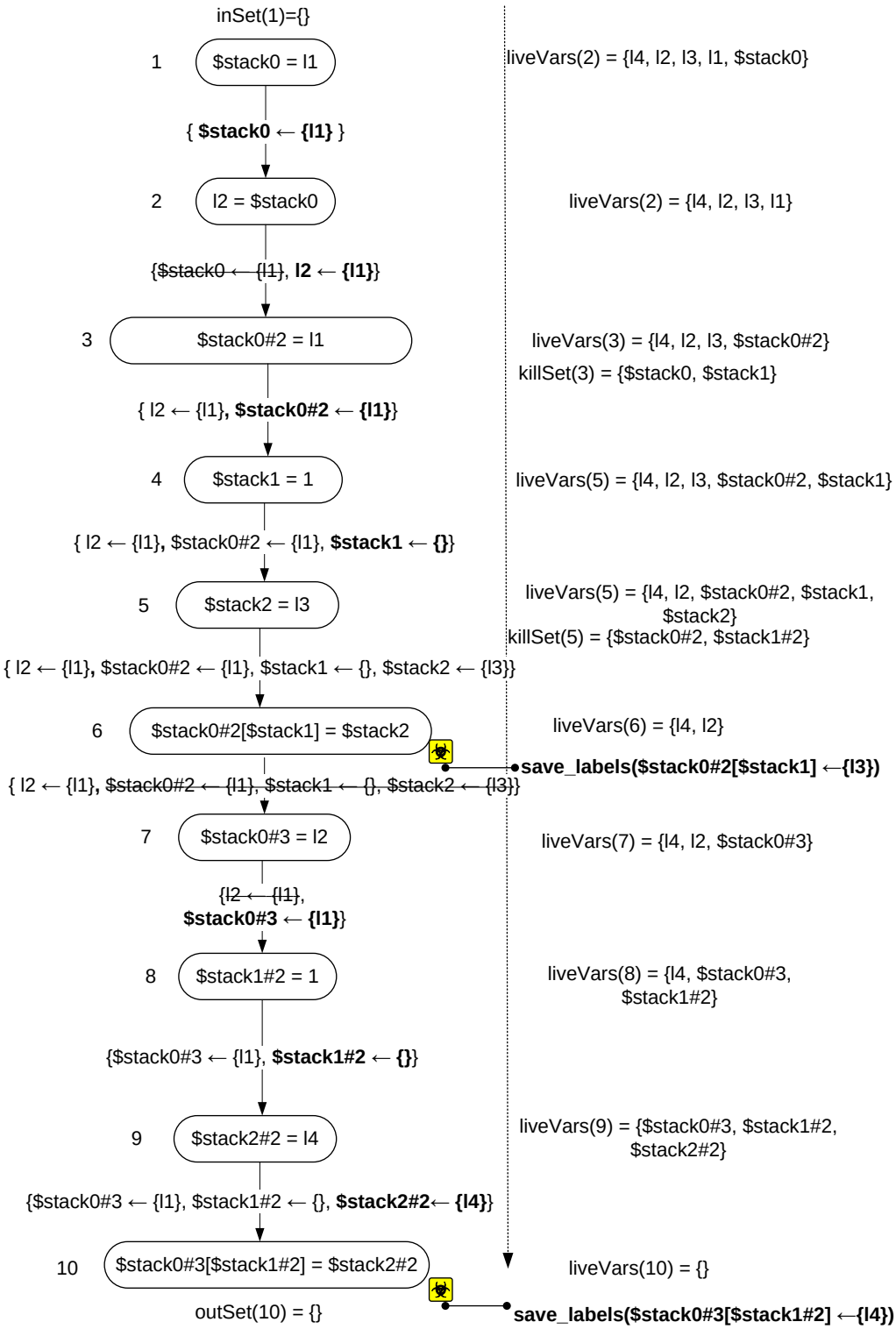

FIGURE 4.8 – Résultat de l'analyse statique de l'exemple figure 4.5, prenant en compte l'analyse de vivacité

référence de l'instance est alors stocké dans la variable locale à la position 0. Le premier argument est stocké à la position 1, le deuxième argument à la position 2 et ainsi de suite. L'appel d'une méthode statique se fait de la même manière, sauf que la méthode appelante n'empile pas de référence d'instance d'objet. Le premier argument de la méthode appelée se trouve alors dans la variable locale à la position 0 de la nouvelle *frame*, le deuxième à la position 1 et ainsi de suite.

L'analyse statique que nous proposons insère un point de contrôle **save_labels()** à chaque appel de méthode, pour forcer la JVM à mettre à jour tous les labels de sécurité des variables manipulées par la méthode appelante. En effet, si nous décidons de retarder cette mise à jour de label jusqu'au premier point de contrôle rencontré dans l'une des méthodes appelées, il faudrait . La JVM calcule alors les labels de sécurité des paramètres de la méthode appelée et les associe aux positions correspondantes sur la nouvelle pile. Ces labels sont utilisés par la suite pour déduire les labels de sécurité des variables manipulées par la méthode appelée.

Lors du retour de la méthode appelée, la valeur de retour éventuelle est empilée sur sa pile. Suite à l'exécution de l'instruction de retour, la JVM empile cette valeur de retour sur la pile de la méthode appelante avant de libérer complètement la pile de la méthode appelée. Or, comme l'instruction de retour est un point de sortie d'un bloc de base, c'est donc un point de contrôle **save_labels()**. La JVM calcule alors préalablement les labels de sécurité des variables manipulées par la méthode appelée, et associe donc un label de sécurité à la valeur de retour aussi. Ce label de sécurité est donc associé à la valeur de retour empilé sur la pile de la méthode appelante.

4.3 Suivi des flux implicites

4.3.1 Description

L'analyse statique proposée pour le suivi des flux implicites permet à la JVM modifiée de maintenir une pile de labels de sécurité, associés aux variables générant des flux implicites. Cette pile est ensuite utilisée pour propager ces labels de sécurité aux variables modifiées au sein des branches. Dans la suite, nous parlerons de flux implicites directs pour les flux implicites concernant les variables modifiées au sein d'une branche exécutée. Nous parlerons aussi de flux implicites indirects concernant les variables modifiées au sein d'une branche non exécutée. La figure 4.9 illustre ces deux types de flux d'information que nous différencions à présent.

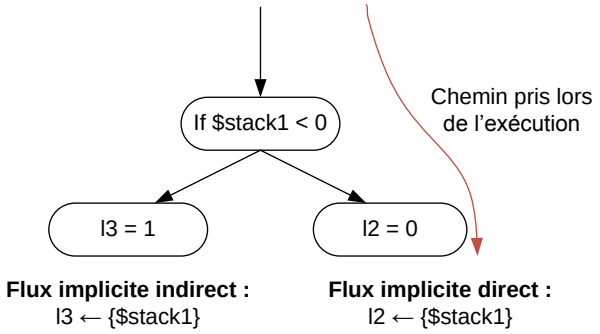

FIGURE 4.9 – Flux implicites directs et indirects

L'analyse statique que nous proposons prend en compte les flux implicites directs, ainsi que certains flux implicites indirects.

4.3.2 Présentation de l'analyse statique des flux implicites directs

Notre analyse statique permet la prise en compte des flux implicites directs grâce au parcours du CFG de la méthode analysée. Cette analyse insère un point de contrôle à chaque branchement conditionnel, ainsi qu'un second point de contrôle à l'instruction de jonction des deux branches, c'est-à-dire au post-dominateur immédiat du branchement conditionnel. Comme l'illustre la figure 4.10, le premier point de contrôle force la JVM modifiée à évaluer puis empiler le label de sécurité associé aux variables dont la condition de branchement dépend, grâce à l'action **push(V)**, où V est un ensemble de variables. Le deuxième point de contrôle force la JVM à dépiler ce label de sécurité avant l'exécution de l'instruction au point de jonction des branches, grâce à l'action **pop(i)** où i est un entier précisant le nombre d'élément à dépiler. Le point de jonction ou post-dominateur immédiat du branchement conditionnel est calculé grâce à une analyse de flux de données en arrière que nous avons présenté dans la sous-section 3.4.2. L'analyse statique détermine alors les prédécesseurs du post-dominateur immédiat, pour les marquer d'un point de contrôle.

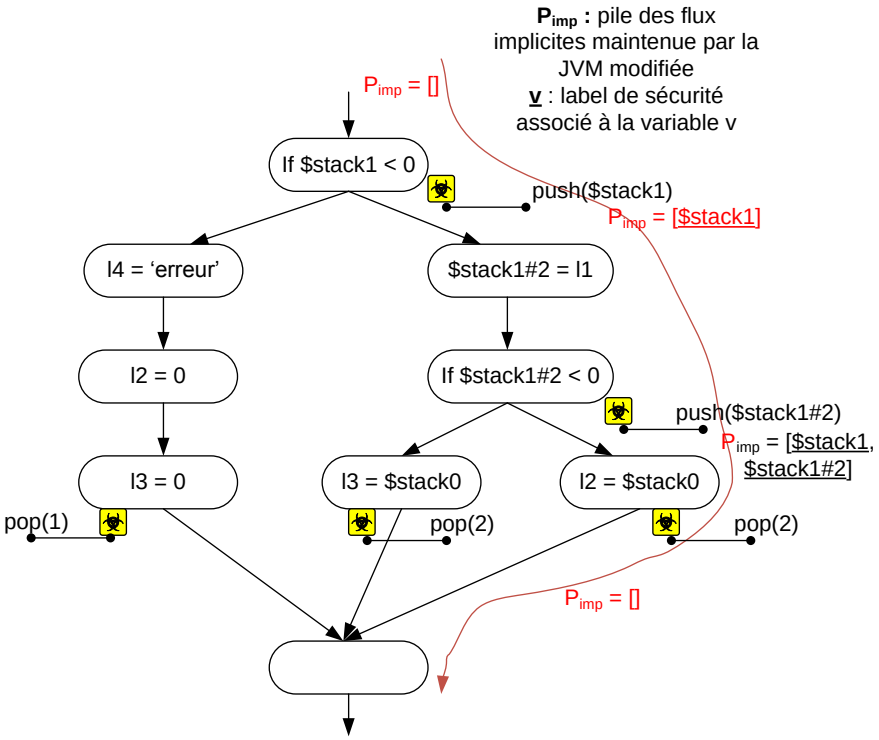

FIGURE 4.10 – Analyse statique des variables générant des flux implicites

Il est plus judicieux d'instrumenter ainsi la JVM modifiée pour construire, dynamiquement, la pile contenant les labels de sécurité des variables générant des flux implicites. En effet, comme le label de sécurité associé à une condition de branchement peut changer au cours de l'exécution d'une des branches (si les variables intervenant dans cette condition sont redéfinies), il faut évaluer ce label de sécurité au moment de l'exécution de l'instruction conditionnelle, ce qui revient à construire la pile dynamiquement.

Grâce à la pile des flux implicites, la JVM modifiée peut tenir compte des flux implicites directs. En effet, à chaque fois qu'elle prend en compte les flux explicites concernant une variable v et ses

dépendances explicites D_v, elle propage les labels de sécurité des variables de D_v vers v. Il suffit alors de propager aussi les labels de la pile des flux implicites vers v.

4.3.3 Les appels de méthode

Lorsque l'appel d'une méthode se trouve au sein d'une des branches dépendant d'un branchement conditionnel c, toutes les variables modifiées au sein de cette méthode dépendent de ce branchement conditionnel. Il y a donc un flux implicite à partir de la condition présente dans la méthode appelante, vers toutes les variables modifiées dans la méthode appelée. La JVM modifiée doit alors transmettre l'état de la pile des flux implicites à la méthode appelée pour tenir compte de ces flux implicites.

Une fois l'un des points de sortie de la méthode appelée atteint, la JVM reprend l'exécution de la méthode appelante. Elle reprend alors la pile des flux implicites à l'état précédant l'appel de méthode. En effet, nous considérons que l'exécution des instructions suivant l'appel de méthode ne dépend pas des instructions de la méthode appelée, sauf dans certains cas discutés ci-après. La portée des flux implicites générés au sein de la méthode appelée est donc limitée à cette méthode et c'est pour cela que nous reprenons la pile des flux implicites à l'état précédent l'exécution de cette méthode.

Cette hypothèse est fausse dans trois cas :

1. la méthode appelée contient une boucle infinie. Les instructions suivant l'appel de méthode ne sont alors jamais exécutées ;
2. la méthode appelée quitte le programme exécuté.
3. la méthode appelée génère une exception. Cette exception peut causer l'arrêt du programme ou peut être gérée par l'une des méthode appelantes. Les instructions suivant l'appel de méthode ne sont donc sûrement pas exécutées. Notons que si l'exception est gérée par la méthode appelée, alors les instructions suivant l'appel de méthode sont bien exécutées.

Les cas numéro 1 et 2 correspondent à la (non) terminaison du programme contrôlé. Certes, la terminaison d'un programme peut causer une fuite d'information, jugée très souvent minime. Ce problème reste en tout cas l'un des challenges ouverts dans le domaine du contrôle de flux d'information. Notre analyse ne tient pas compte de la terminaison du programme contrôlée. Elle est donc insensible à la terminaison.

Le dernier cas est plus problématique, et concerne la gestion des flux implicites dûs aux exceptions que nous discuterons dans sous-section 4.3.5.

4.3.4 Présentation de l'analyse statique des flux implicites indirects

Afin de tenir compte des flux d'information implicites indirects, nous complétons l'analyse statique réalisée pour les flux implicites directs par une analyse de flux de données en arrière. Cette analyse permet de déterminer quelles variables sont modifiées au sein des différentes branches, pour permettre à la JVM modifiée de mettre à jour leur label de sécurité lors de l'exécution, quelque soit la branche exécutée.

Avant de présenter cette analyse, nous exposons d'abord la proposition 1 concernant le code Jimple. Cette proposition affirme que toute instruction du code Jimple a au plus deux successeurs si nous ignorons les exceptions lors de la construction du CFG de la méthode analysée. En effet, la seule instruction réalisant un branchement en Jimple est l'instruction if. Or comme le code Jimple est un code 3 adresse, une instruction if ne peut définir que deux successeurs possibles

car le premier opérande du code Jimple est utilisé pour l'instruction if, le deuxième opérande est utilisé pour la condition du branchement et le troisième opérande est utilisé pour l'instruction cible du branchement si la condition est vraie. Donc suite à l'exécution d'une instruction if, le flot de contrôle peut soit passer à l'instruction suivante si la condition du branchement est fausse ou passer à l'instruction cible du branchement si la condition du branchement est vraie. Donc si nous ignorons le changement de flot de contrôle dû aux exceptions, toute instruction ayant deux successeurs est une instruction if.

Proposition 1. *Si l'on ignore les exceptions lors de la construction du CFG, alors toute instruction a au plus deux successeurs. De plus, une instruction ayant deux successeurs est une instruction if.*

L'analyse que nous proposons pour le suivi des flux implicites indirects prend en compte tous les attributs de classes qui pourraient être modifiés lors de l'exécution, puisqu'ils ont une portée globale. Par contre, elle prend en compte seulement les variables locales et les opérandes de pile qui sont vifs au point de jonction de chaque branche, c'est-à-dire au post-dominateur immédiat. En effet, les variables locales et opérandes de piles ayant une portée locale à la méthode, il n'y a pas besoin de mettre à jour leur label de sécurité dès qu'elles ne sont plus utilisées au sein de la méthode.

Cette analyse associe à chaque instruction i deux piles d'ensembles $inSet(i)$ et $outSet(i)$ contenant les variables modifiées au sein des deux branches d'une instruction conditionnelle. Les équations de cette analyse sont explicitées figure 4.3.

$$
\begin{aligned}
outSet(i) &= \begin{cases} inSet(succ(i)) \text{ si } i \text{ a un seul successeur} \\ inSet(s_2) \bigsqcup_i inSet(s_1) \text{ si } i \text{ est une instruction } if \text{ dont les deux successeurs} \\ \qquad \text{sont } s_1 \text{ et } s_2 \end{cases} \\
inSet(i) &= \begin{cases} popMergeLast(outSet(i)) \text{ si i est une instruction if} \\ pushNewSet(outSet(i), n) \text{si i est un prédécesseur d'un post-dominateur} \\ \qquad \text{immédiat, marqué d'une action pop(n)} \\ addToLastSet(outSet(i), v) \text{ si v est une variable définie par l'instruction i} \\ addToLastSet(pushNewSet(outSet(i), n), v) \text{ si v est définie par l'instruction i} \\ \qquad \text{et i est un prédécesseur d'un post-dominateur immédiat marqué} \\ \qquad \text{d'une action pop(n)} \\ outSet(i) \text{ sinon} \end{cases}
\end{aligned}
\tag{4.3}
$$

Ces équations font intervenir un opérateur $\bigsqcup$ paramétré par l'instruction i et défini pour deux piles d'ensembles ayant le même nombre d'éléments, ainsi que différents autres opérateurs définis pour une pile d'ensemble. Une définition formelle de chacun de ces opérateurs est présenté en 12, 13 et 14. L'ensemble *outSet* du point de sortie de la méthode **est initialisé** avec une pile contenant un seul ensemble vide.

Cette analyse a été réalisée de sorte que le nombre d'éléments des piles $outSet(i)$ et $inSet(i)$ reflète la profondeur de branchements conditionnels à l'instruction i. L'opérateur $popMergeLast$ dépile le dernier ensemble et rajoute toutes les variables qu'il contient à l'avant dernier ensemble, sauf les variables locales et opérandes de piles non vifs au post-dominateur immédiat. L'opérateur

pushNewSet empile un ensemble vide à chaque fois que la profondeur de branchements conditionnels augmente. Le nombre d'ensembles à empiler est déterminé grâce aux résultats de l'analyse précédentes des flux implicites directs, puisque comme l'illustre la figure 4.10, une instruction peut être le post-dominateur immédiat de plusieurs branchements. L'opérateur *addToLastSet*, par contre, ajoute la variable définie par une instruction au dernier ensemble de la liste *inSet*.

Définition 12 (Les opérateur *popMergeLast* et *pushNewSet*)**.** Soient L une pile d'ensembles à n éléments.

Alors la pile M à $n-1$ éléments telle que $M = popMergeLast(L)$ est définie comme suit :

$$M[k] = \begin{cases} L[k] \text{ si } 1 \leq k < n-1 \\ L[n-1] \cup L[n] \text{ si } k = n-1 \end{cases}$$

où $\cup$ est l'opérateur d'union défini pour les ensembles.

La pile N à $n+m$ éléments telle que $N = pushNewSet(L, m)$ et $m \geq 1$ est définie comme suit :

$$N[k] = \begin{cases} L[k] \text{ si } 1 \leq k \leq n \\ \{\emptyset\} \text{ si } n+1 \leq k \leq n+m \end{cases}$$

Définition 13 (L'opérateur *addToLastSet*)**.** Soient L une pile d'ensembles à n éléments. Soit v une variable.

Alors la liste $O = addToLastSet(L, v)$ est définie comme suit :

$$O[k] = \begin{cases} L[k] \text{ si } 1 \leq k < n \\ L[n] \cup \{v\} \text{ si } k = n \end{cases}$$

Définition 14 (L'opérateur $\bigsqcup$ paramétré par l'instruction conditionnelle i)**.** Soient L_1 et L_2 deux piles d'ensembles à n éléments.

Soit V_i l'ensemble des variables locales et piles d'opérandes vives au post-dominateur immédiat de l'instruction i.

Nous notons $Loc(L_1[k])$ la projection des variables de l'ensemble $L_1[k]$ sur les variables locales et opérandes de piles. Notons aussi $Class(L_1[k])$ la projection des variables de l'ensemble $L_1[k]$ sur les variables de classes (attributs statiques).

La pile d'ensembles L à n éléments telle que $L = L_1 \bigsqcup_i L_2$ est définie comme suit :

$$L[k] = \begin{cases} L_1[k] \text{ si } 1 \leq k \leq n-1 \\ Class(L_1[k]) \cup Class(L_2[k]) \cup \Big(\big(Loc(L_1[k]) \cup Loc(L_2[k])\big) \cap V_i\Big) \text{ si } k = n \end{cases}$$

L'opérateur $\bigsqcup_i$ paramétré par l'instruction conditionnelle i réalise l'union des variables appartenant aux deux derniers éléments des piles $inSet(s_1)$ et $inSet(s_2)$, où s_1 et s_2 sont les deux successeurs de i, tout en supprimant les variables locales et opérandes de piles qui ne sont plus vifs au post-dominateur immédiat de l'instruction i. Cet opérateur est définie pour deux piles ayant le même nombre d'éléments car les ensembles *inSet* des deux successeurs de l'instruction i ont le même nombre d'éléments. De plus, comme l'affirme la proposition 2, tous les éléments de ces deux ensembles *inSet* sont égaux sauf leur dernier élément. Bien que nous ne démontrons pas cette proposition, il est intuitif de se rendre compte grâce notamment à l'exemple de la figure 4.11, que le dernier élément de l'ensemble *inSet* d'un successeur s de i contient des variables modifiées dans les chemins d'exécution reliant s au post-dominateur immédiat de i, alors que les autres éléments de *inSet* contiennent les variables modifiées dans les chemin d'exécution reliant le post-dominateur immédiat de i au point de sortie de la méthode analysée.

Proposition 2 (Les ensembles $inSet(s)$ des successeurs s d'une instruction conditionnelle i)**.** *Soit i une instruction conditionnelle et s_1, s_2 ses deux successeurs. Alors*

$\text{card}(inSet(s_1)) = \text{card}(inSet(s_2))$

et $\forall k$*, tel que* $1 \leq k \leq \text{card}(inSet(s_1)) - 1$*, alors* $inSet(s_1)[k] = inSet(s_2)[k]$

Un exemple de cette analyse est illustré par la figure 4.11. Le dernier ensemble de la pile $outSet(i)$ de chaque instruction conditionnelle i correspond aux variables potentiellement modifiées au sein des deux branches définis par i et qui sont vives au post-dominateur immédiat de i. En effet, si nous considérons l'instruction 1 de cet exemple, le dernier élément de $outSet(1)$ est $\{l2, l3\}$. Or ces deux variables sont justement les seules variables qui sont vives au post-dominateur immédiat de l'instruction 1, c'est-à-dire l'instruction 9.

Afin de permettre la prise en compte des flux implicites indirects lors de l'analyse dynamique, nous annotons le post-dominateur immédiat de chaque instruction conditionnelle i avec le dernier élément de la pile $outSet(i)$. L'analyse dynamique évalue alors les nouveaux labels de sécurité des variables modifiées au sein des branches exécutées et non exécutées en utilisant la pile des flux implicites.

L'analyse que nous avons proposé ne prend en compte que les flux implicites indirects concernant les variables locales et les opérandes de piles, ainsi que variables de classe. Elle ne peut pas prendre en compte les flux implicites indirects concernant les attributs d'objets car nous ne connaissons pas les références des objets concernés par ce flux implicites indirects lors de l'exécution. Ceci nécessiterait de réaliser une analyse statique d'alias et suppose donc une analyse inter-procédural comme le propose Chandra et Franz [CF07]. Nair et al. [NSCT07] par contre, adopte une approche qui consiste à ne pas dépiler la pile des flux implicites au post-dominateur immédiat d'une instruction conditionnelle si un attribut d'objet, dont la référence ne peut pas être déterminé à l'exécution, est modifié au sein d'une branche non exécutée. Ceci pourrait permettre de tenir compte des flux implicites indirects si cet attribut d'objet est utilisé par la suite. Cette approche, que nous pensons trop conservatrice, nécessite alors le dépilement de la pile des flux implicites manuellement pour éviter le problème de la contamination des labels de sécurité.

ne permet de prendre en compte que les flux implicites indirects concernant les objets dont la référence est connue, avant l'exécution de l'instruction conditionnelle. Considérons l'exemple où une référence d'un objet ne peut être connue qu'au moment de l'exécution grâce à un appel de méthode. Si cet appel de méthode se trouve dans une branche non exécutée, l'analyse dynamique ne peut déterminer la référence de cet objet afin de tenir compte des flux implicites indirects le concernant.

Les analyses proposées pour le suivi des flux d'information implicites, directs ou indirects, ne prennent pas en compte les flux générés par les exceptions. Nous discutons ce dernier point dans la sous-section suivante.

4.3.5 Les exceptions

Les exceptions génèrent des flux d'informations implicites dûs au changement de flot de contrôle. Nous pouvons distinguer deux types d'exceptions :

- les exceptions non gérées par le programme qui causent son arrêt ;
- les exceptions gérées par le programme, que ce soit par la méthode courante ou l'une des méthode appelantes.

Notre analyse est une analyse insensible à la terminaison du programme et ne prend donc pas en compte les exceptions non gérées. Pour ce qui est des exceptions gérées, elles peuvent l'être par

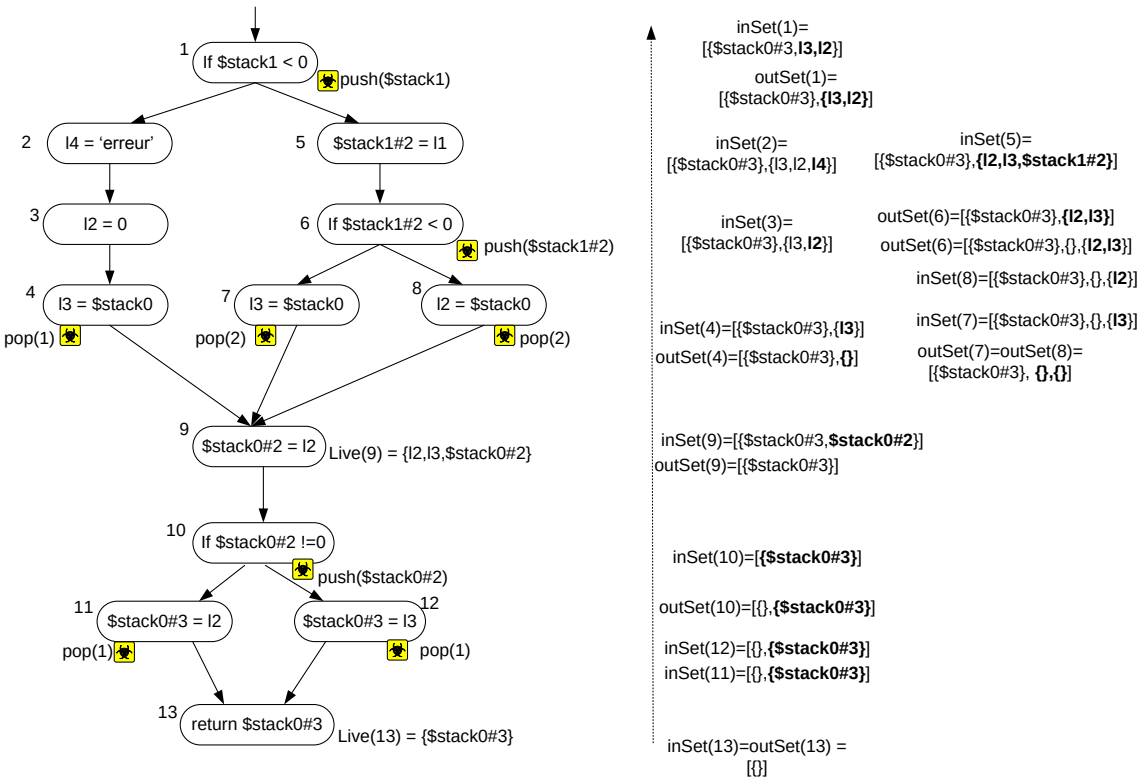

FIGURE 4.11 – Exemple d'analyse statique des flux implicites indirects

l'une des méthodes appelantes ou par la méthode courante.

Si l'exception est gérée par l'une des méthodes appelantes, ceci cause le changement du flot de contrôle vers une autre méthode. Or comme nous proposons une analyse intra-méthode, nous ne pouvons tenir compte dans l'analyse statique, des flux d'information engendrés par l'éventuelle levée de cette exception. Nous devrions considérer une analyse inter-procédural pour pouvoir tenir compte des flux générés par ces exceptions.

En ce qui concerne les exceptions gérées par la méthode courante, il est possible de tenir compte des flux d'informations qu'elles génèrent puisque le flot de contrôle change au sein de la même méthode. Pour l'instant, notre analyse des flux implicites ignore ce type d'exceptions car nous pensons que cela entrainerait une contamination excessive des labels de sécurité (*label creep*). En effet, toutes les variables utilisées dans des instructions pouvant lever une exception génèrent des flux implicites. Or la prise en compte de ces flux implicites reviendraient à considérer toutes ces instructions comme une suite d'instructions conditionnelles qui nécessiteraient la prise en compte des flux implicites directs et indirects. Cette approche nous semble trop conservatrice car la levée d'exceptions relève de cas particuliers anormals qui ne font pas partie de l'exécution courante des programmes contrôlés. Nous pensons tout de même développer une analyse prenant en compte ces exceptions pour pouvoir comparer les résultats à une analyse qui les ignore afin d'affirmer ou infirmer cette conviction.

4.4 Résumé

Nous avons présenté dans cette section l'analyse statique développée pour le suivi des flux d'information. Cette analyse intra-méthode est réalisée sur une représentation intermédiaire du Bytecode : Jimple. Elle est constituée de plusieurs étapes dont la première est la construction du CFG de la méthode, réalisée grâce à Soot. Ensuite, une analyse de vivacité des variables, implémentée aussi par Soot, est effectuée. Les résultats de cette analyse sont utilisés tout au long des étapes suivantes. S'ensuit alors une analyse de flot de données pour chaque bloc de base de la méthode analysée. Cette analyse que nous avons implémenté, permet la prise en compte des flux explicites par l'analyse dynamique, grâce notamment à des points de contrôles forçant la JVM modifiée à mettre à jour les labels de sécurité associés à tout ou une partie des variables. Un parcours du CFG, que nous avons implémenté, est ensuite réalisé pour insérer des points de contrôles pour chaque instruction conditionnelle ainsi que les prédécesseurs de leur post-dominateur immédiat. Ces points de contrôles permettent lors de l'analyse dynamique de construire une pile de labels de sécurité, associés aux variables générant des flux implicites. Finalement, une dernière analyse de flot de donnée est réalisée, pour permettre à l'analyse dynamique la prise en compte des flux implicites indirects. Cette dernière analyse ainsi que les annotations de méthodes avec les résultats de notre analyse statique n'ont pas encore été implémenté. Cependant, nous avons déjà pensé à la manière de transposer les résultats de notre analyse statique effectuée sur du code Jimple au Bytecode Java. En effet, nous disposons grâce à Soot de l'offset de l'instruction Bytecode correspondant à chaque instruction Jimple. Donc nous pouvons associer chaque ensemble $outSet$ à son instruction Bytecode correspondante. De plus, nous avons utilisé grâce à Soot un nommage particulier des variables en Jimple, qui nous permet d'associer avec précision à chaque variable Jimple soit une position précise sur la pile, soit une variable locale précise.

Chapitre 5

Travaux futurs et conclusion

Durant la première partie de ce stage, nous avons travaillé sur un modèle d'analyse statique permettant le suivi hybride des flux d'information au sein des programmes Java, pour la détection d'intrusion. Nous avons présenté notre approche dans ce mémoire, ainsi que les hypothèses que nous avons fait concernant l'analyse dynamique.

L'analyse statique que nous avons proposée permet la construction d'un profil pour chaque méthode analysée. Ce profil contient des informations sur les flux générés par la méthode, qu'ils soient explicites ou implicites. Il inclue aussi des points de contrôles instrumentant les actions réalisées lors de l'analyse dynamique pour la mise à jour des labels de sécurité. Nous avons implémenté l'analyse de flot de données prenant en compte les flux explicites générés par la méthode analysée, ainsi que le parcours du CFG permettant la prise en compte des flux implicites directs. Nous allons implémenter dans la suite de ce stage l'analyse dynamique des flux implicites indirects prenant compte les variables locales, les opérandes de pile ainsi que les variables de classe. Nous allons aussi faire en sorte d'annoter les méthodes analysées avec les profils réalisés grâce à notre analyse statique avant la fin de la première partie de ce stage.

Durant la deuxième partie de ce stage, nous nous attacherons à la conception et à l'implémentation de l'analyse dynamique des flux d'information. Nous apporterons les modifications nécessaires à la JVM réalisant cette analyse, pour lui permettre de charger les profils des méthodes avant leur exécution et d'interpréter ces profils en termes de flux d'information et de points de contrôle. Nous réaliserons aussi le mécanisme de gestion des labels de sécurité capable d'associer un label à chacune des variables manipulées lors de l'exécution d'un programme. Ce mécanisme s'intégrera avec Blare, l'outil de détection d'intrusion basé sur le suivi des flux d'information au niveau OS. De plus, Nous réaliserons des tests sur des applications réalistes pour évaluer les résulats de notre analyse de suivi des flux d'information et les performances engendrés par l'approche hybride que nous proposons.

La suite de ces travaux pourrait concerner l'amélioration de l'analyse statique que nous proposons. En effet, une analyse d'alias couplé à une analyse intra-procédurale pourrait permettre d'améliorer la précision du suivi des flux d'information implicites indirects. Elle pourrait aussi permettre l'optimisation de l'analyse dynamique en diminuant le nombre de points de contrôles insérés lors de l'analyse statique.

Table des figures

Bibliographie

[All70] Frances E. Allen. Control flow analysis. *SIGPLAN Not.*, 5(7) :1–19, 1970.

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

[Bib77] K. Biba. Integrity considerations for secure computer systems. Technical Report N°ESD-TR 76-372, MITRE Co., April 1977.

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

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

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

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

[HMMT07] G. Hiet, L. Mé, B. Morin, and V. Viet Triem Tong. Monitoring both os and program level information flows to detect intrusions against network servers. In *IEEE Workshop on "Monitoring, Attack Detection and Mitigation"*, 2007.

[HMZ+07] Guillaume Hiet, Ludovic Mé, Jacob Zimmermann, Christophe Bidan, Benjamin Morin, and Valérie Viet Triem Tong. Détection fiable et pertinente de flux d'informations illégaux. In *Proc of the 2nd Conference on Security in Network Architectures and Information Systems (SARSSI)*, 2007.

[HRU76] Michael A. Harrison, Walter L. Ruzzo, and Jeffrey D. Ullman. Protection in operating systems. *Communication of the ACM*, 19(8) :461–471, 1976.

[HS06] S. Hunt and D. Sands. On flow-sensitive security types. In *Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 79–90. ACM, 2006.

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

[LB73] Leonard J. LaPadula and D. Elliott Bell. Secure computer systems : A mathematical model. MTR-2547 (ESD-TR-73-278-II) Vol. 2, MITRE Corp., Bedford, may 1973.

[Muc97] S.S. Muchnick. *Advanced compiler design and implementation.* Morgan Kaufmann, 1997.

[Mye99] A.C. Myers. JFlow : Practical mostly-static information flow control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 228–241. ACM, 1999.

[NSCT07] Srijith K. Nair, Patrick N. D. Simpson, Bruno Crispo, and Andrew S. Tanenbaum. A virtual machine based information flow control system for policy enforcement. In *First International Workshop on Run Time Enforcement for Mobile and Distributed Systems (REM 2007)*, pages 1–11, Dresden, Germany, 2007.

[RS10] Alejandro Russo and Andrei Sabelfeld. Dynamic vs. static flow-sensitive security analysis. *Computer Security Foundations Symposium, IEEE*, 0 :186–199, 2010.

[SCS98] R. Sekar, Yong Cai, and Mark Segal. A specification-based approach for building survivable systems. In *Proceedings of the 21st National Information Systems Security Conference (NISSC'98)*, pages 338–347, Crystal City, VI, October 1998.

[VRH98] R. Vallee-Rai and L.J. Hendren. Jimple : Simplifying java bytecode for analyses and transformations. 1998.

[VS97] D. Volpano and G. Smith. A type-based approach to program security. In *Theory and Practice of Software Development*, 1997.

[VSI96] Dennis Volpano, Geoffrey Smith, and Cynthia Irvine. A sound type system for secure flow analysis. *Journal in Computer Security*, 4(2-3) :167–187, 1996.

[ZMB03] Jacob Zimmermann, Ludovic Mé, and Christophe Bidan. An improved reference flow control model for policy-based intrusion detection. In *Proceedings of the 8th European Symposium on Research in Computer Security (ESORICS)*, October 2003.