MÉCANISME DE CONTRÔLE DU FLOT D'INFORMATION DANS UN PROGRAMME: APPROCHE PAR TYPAGE TRIVALUÉ

Mémoire

Erwanne Paméla Kanyabwero

Maîtrise en informatique
Maître ès sciences (M.Sc.)

Québec, Canada

Résumé

Ce mémoire présente un mécanisme d'application de politiques de sécurité grâce à une analyse de types trivaluée sur un langage impératif. Notre analyse a pour but de réduire les faux positifs générés par l'analyse statique, tout en préparant les programmes analysés à être instrumentés. Les faux positifs se produisent dans l'analyse de systèmes informatiques à temps réel quand il manque de l'information au moment de la compilation, par exemple un nom de fichier, et par conséquent, son niveau de sécurité. Notre approche visant à répondre à la question « Y a-t'il violation de la propriété de non-interférence dans le programme ? », l'idée clé est de distinguer les réponses négatives des réponses incertaines. Au lieu de rejeter le programme systématiquement en cas d'incertitude, nous typons les instructions avec un type additionnel, *unknown*, indiquant l'incertitude. Notre travail est fait en préparation à un mécanisme d'application hybride, combinant l'analyse statique par l'analyse de types et l'analyse dynamique par l'instrumentation de code. Durant l'étape d'analyse statique, les réponses positives et négatives sont traitées de manière standard, tandis que les réponses incertaines sont clairement annotées du type incertain, préparant ainsi pour un éventuel passage à la deuxième étape : l'instrumentation. Une preuve que notre système de types est cohérent est donnée, ceci en montrant qu'il satisfait la non-interférence. Les programmes interagissent à travers les canaux de communication. Notre contribution réside dans la manipulation de trois types de sécurité, mais aussi dans le traitement des variables et canaux de communication de manière particulière. Les niveaux de sécurité sont associés aux canaux plutôt qu'aux variables dont les niveaux de sécurité varient selon l'information qu'elles contiennent.

Table des matières

Liste des tableaux

Liste des figures

Au Dieu Tout-Puissant, pour
Sa protection,
à René, mon époux, pour son
amour,
à mes parents, pour leur
soutien,
je dédie ce mémoire.

Remerciements

J'adresse mes vifs remerciements à mes directrices de recherche, les Professeures Nadia Tawbi et Josée Desharnais, pour leur supervision et leurs conseils judicieux pendant la rédaction de ce mémoire. Je remercie aussi les membres du jury, les Professeurs Mohamed Mejri et Danny Dubé, pour avoir accepté d'examiner ce mémoire.

Je tiens à remercier particulièrement mon cher époux, René, qui m'accorde toujours l'attention, le soutien et le réconfort dont j'ai besoin. Je remercie profondément ma famille, surtout mes parents, qui ont toujours cru en moi et dont ce travail fera la fierté. Merci à mes amis de Québec et mes collègues qui ont rendu agréables et instructifs mon environnement de travail et mon séjour à Québec.

Je ne saurais terminer sans rendre grâce au Tout-Puissant qui m'a accordé la force, physique et morale, et le courage d'aller au bout de cet ouvrage.

Introduction

0.1 Motivation et description sommaire de l'approche

Notre dépendance au fonctionnement de systèmes de technologies de l'information et de la communication croît très rapidement. Ces systèmes deviennent de plus en plus complexes et parallèlement, nous assistons à un accroissement des échanges d'informations via l'Internet et toutes sortes de systèmes embarqués tels que les smart cards, les ordinateurs portables, les téléphones mobiles, etc. En plus de la performance de ces systèmes en termes de temps de réponse et de capacité d'opérer, un contrôle efficace de la confidentialité et de l'intégrité de l'information échangée dans ces systèmes s'impose. Il est crucial de s'assurer que cette information ne sera accessible qu'aux entités autorisées.

Le travail présenté dans ce mémoire s'insère dans le cadre des mécanismes d'application de politiques de sécurité. Il porte précisément sur les politiques de contrôle du flot d'information, indiquant comment l'information est transmise entre les variables et les diverses entités dans un système informatique. La politique peut indiquer, par exemple, les droits d'accès d'un objet à une ressource.

Deux types d'approches peuvent être utilisés pour appliquer des politiques de sécurité : les approches basées sur l'analyse statique et celles basées sur l'analyse dynamique. Toutefois, des approches mixtes existent également, afin de tirer avantage des deux méthodes. L'analyse statique consiste à étudier le comportement d'un programme sans en exécuter le code, contrairement à l'analyse dynamique qui surveille le programme en introduisant des tests durant son exécution. L'analyse statique prend en considération toutes les exécutions possibles d'un programme. Ce dernier est rejeté dès qu'une des exécutions est jugée potentiellement dangereuse. L'avantage qui s'en suit est la détection des erreurs de programmation. De plus, il n'y a pas d'impact sur le coût d'exécution,

contrairement à l'analyse dynamique qui entraîne un ralentissement dans l'exécution des programmes. Par contre, l'analyse statique est moins précise car, pour valider statiquement tous les scénarios possibles d'exécution d'un programme, il faut nécessairement le faire par approximation. À titre d'exemple, l'analyse statique ne peut prévoir le comportement d'entités externes au programme comme les usagers. Le manque de ces informations purement dynamiques, ajouté à la complexité inhérente de certains calculs engendrent le problème d'indécidabilité dont nous parlerons plus loin.

Objectifs du mémoire L'approche qui fait l'objet de notre travail est une approche à finalité hybride. En effet, nous présentons une méthode d'analyse statique qui s'aidera, le cas échéant, de l'analyse dynamique pour déterminer si un programme est sécurisé ou pas. Notre objectif est premièrement de réduire les faux positifs occasionnés par l'analyse statique, deuxièmement de préparer les programmes à être instrumentés si nécessaire. Comme mentionné plus haut, nous nous intéressons au contrôle du flot d'information, c'est-à-dire contrôler les fuites d'information dans un programme vers des entités non autorisées. Par exemple, il ne faudrait pas qu'un programme utilisant un numéro de carte de crédit divulgue cette information à une personne non autorisée ; il ne faudrait pas non plus qu'un programme d'authentification qui vérifie un mot de passe secret l'envoie dans un fichier non confidentiel. Ce contrôle est réalisé avec une *analyse par typage* (*type-based analysis*). Le comportement désiré du programme est représenté à l'aide d'un système de types basé sur la syntaxe du langage de programmation. Toutefois, l'analyse statique ne peut être qu'approximative car certaines informations ne peuvent être connues au moment de la compilation. Notre principale contribution a été de concevoir un système de types d'une manière qui fasse ressortir les cas incertains, c'est-à-dire les points du programme dont on ne peut juger statiquement s'ils respectent ou non la politique de sécurité. L'idée clé de notre approche est l'analyse trivaluée qui introduit principalement un troisième niveau de sécurité caractérisant les données incertaines. Au terme de l'analyse, trois résultats peuvent être conclus : le programme *respecte*, *ne respecte pas* ou *respecte peut-être* la non-interférence. Notez que les analyses classiques ne fournissent que les deux premières réponses, et les cas incertains sont analysés de façon pessimiste, ce qui engendre des faux positifs. Nous traitons différemment l'incertitude car les réponses négatives et incertaines sont clairement distinctes. En effet, les points incertains du programme auront été annotés clairement durant l'analyse par typage. Nous préparons ainsi le programme à la suite de nos travaux, qui consistera à insérer des tests dynamiques aux points incertains afin de rendre plus précise l'analyse. Ces tests supplémentaires permettront de valider avec certitude

si les points du programme sont sûrs ou pas. Nous réduisons ainsi les fausses alarmes tout en introduisant un temps d'exécution minimal par l'instrumentation seulement si nécessaire.

Les canaux de communication sont classés en trois niveaux (ou types) de sécurité : *privé* ou confidentiel, *public* ou non-confidentiel et *inconnu*. Ici, par canal de communication, nous entendons tous les canaux par lesquels les programmes interagissent (fichier, canal réseau, etc.). Seuls les types des canaux sont fixés d'avance en privé ou public selon le type d'information qu'ils contiennent. Si le canal n'est pas connu, son type de sécurité correspondra à inconnu. Quant aux variables, les types qui leur sont associés varient selon les informations qu'elles stockent tout le long de l'exécution. Une relation d'ordre est donnée pour ces niveaux de sécurité et le système de types est composé de règles indiquant quels échanges d'information sont permis dépendamment du niveau de sécurité. Un flot d'information est légal d'un objet à l'autre seulement si l'objet source a un niveau de sécurité qui est égal ou plus faible que l'objet cible. Cette politique de sécurité est formalisée par une notion connue sous le nom de *non-interférence* [15].

0.2 Méthodologie et structure du mémoire

Le travail effectué dans le cadre de cette maîtrise s'inscrit dans un projet de plus grande envergure visant à développer un mécanisme d'application de politique de sécurité sur le flot d'information qui se sert à la fois de l'analyse statique et de l'analyse dynamique. Toutefois, notre étude se limite à l'application par approche statique basée sur l'analyse trivaluée, tout en fournissant un cadre de travail pour des travaux futurs d'extension au niveau dynamique par l'instrumentation de code.

Ce mémoire est organisé comme suit : Au chapitre 1, nous présentons les travaux existants sur les mécanismes d'application de politiques de sécurité, l'analyse trivaluée de programmes ainsi que le contrôle du flot d'information. Le chapitre 2 porte sur notre approche, l'analyse sûre du flot d'information par typage trivalué. Enfin, une conclusion est faite au chapitre 2.3.

Chapitre 1

État de l'art

1.1 Introduction

Le présent chapitre fournit un cadre de travail dans lequel s'insère notre approche combinant l'analyse sûre du flot d'information et l'analyse de programmes trivaluée. Nous commençons par un survol des mécanismes d'application des politiques de sécurité. Ensuite, nous présentons quelques applications de la logique trivaluée dans l'analyse de programmes. Enfin, nous décrivons le contrôle du flot d'information sûr ; il s'agit du mécanisme dans lequel nous intégrerons l'analyse trivaluée dans notre approche.

1.2 Mécanismes d'application de politiques de sécurité

Une *politique de sécurité* définit des exécutions de programme jugées acceptables. Nous pouvons citer par exemple les politiques de contrôle d'accès aux ressources, d'allocation des ressources ou de contrôle du flot de données. Il existe une vaste littérature sur les politiques de sécurité, leur caractérisation et leurs mécanismes d'application [16, 21, 29]. Dans le présent mémoire, nous nous intéressons particulièrement aux deux catégories d'approches en matière d'application de politiques de sécurité : les approches basées sur l'analyse statique ainsi que celles basées sur l'analyse dynamique.

1.2.1 Approche statique

Étant donné que notre étude porte principalement sur des techniques d'application de politiques de sécurité basées sur l'analyse statique, nous commençons par une descrip-

tion détaillée de celle-ci. Nous présentons, dans ce chapitre, les différentes approches généralement utilisées en analyse statique à savoir : l'analyse de flot de données et de flot de contrôle, l'interprétation abstraite et le typage.

L'analyse statique englobe les techniques permettant d'extraire des informations sur l'exécution d'un programme sans en exécuter le code. Ces informations extraites décrivent un modèle de comportement du programme. L'analyse statique est souvent utilisée à des fins d'optimisation de code, de compréhension du code pour maintenance, en rétro-ingénierie de logiciels, ainsi que pour la vérification et la certification de code (par exemple, la détection de code malicieux). Elle offre plusieurs avantages : étant donné que l'analyse se fait avant l'exécution du code, il n'y a pas d'impact sur le temps d'exécution de celui-ci. De plus, elle permet de détecter durant le processus de développement du logiciel, des erreurs qui, si elles se produisaient à l'exécution, seraient trop coûteuses à réparer et certaines même fatales.

Toutefois, l'une des principales difficultés à l'analyse statique est le problème d'indécidabilité. En effet, d'après le théorème de Rice [25], toute propriété extensionnelle non-triviale portant sur des programmes est statiquement indécidable. Une propriété extensionnelle est une propriété qui ne dépend que de la sémantique du programme, et non de sa syntaxe. Une propriété est dite non-triviale si elle ne peut être ni toujours vraie, ni toujours fausse pour tous les programmes. L'indécidabilité, quant à elle, fait référence à la théorie de la calculabilité ; une propriété est dite indécidable s'il n'existe pas d'algorithme, terminant dans un nombre fini d'étapes, qui décide si cette propriété est vraie ou fausse. Ainsi, l'analyse statique est indécidable car elle dépend de valeurs reçues dynamiquement par le programme, dont le domaine est infini et que l'utilisateur ne peut prévoir. Par conséquent, il est impossible d'analyser statiquement avec certitude tous les scénarios d'exécution possibles de manière automatique. Toutefois, une alternative raisonnable est d'avoir recours à l'approximation. Autrement dit, quand il n'est pas possible de décider statiquement, il faut opter pour une attitude pessimiste, sacrifiant ainsi la précision pour garantir une décision sûre.

Analyse de flot de données et de flot de contrôle

L'analyse de flot de contrôle permet d'avoir des informations sur l'exécution des instructions du programme. Celle-ci peut être représentée par un graphe, le graphe de flot de contrôle. Il en existe deux types :

Le graphe de flot de contrôle intraprocédural : Il s'agit d'un graphe orienté dont les noeuds indiquent des blocs de base représentant les instructions et les arcs indiquent le transfert du contrôle d'un bloc à l'autre. Par bloc de base, nous entendons une séquence d'instructions consécutives dans laquelle le contrôle (l'exécution) commence au début du bloc et termine à la fin sans possibilité d'arrêt ou de branchement de l'extérieur vers ce bloc d'instructions, sauf au début, ni du bloc vers l'extérieur, sauf à la fin. Un arc pointe vers chaque noeud (bloc de base) pour lequel les instructions peuvent suivre immédiatement celles du noeud courant. Il met en évidence les boucles, instructions conditionnelles, sauts inconditionnels et branchements. Un chemin dans ce graphe représente un chemin d'exécution du programme. Une fois que ce graphe est construit, une variété d'algorithmes peuvent lui être appliquée afin d'obtenir les informations désirées.

Le graphe de flot de contrôle interprocédural ou graphe d'appel : Le flot d'exécution dans un programme peut aussi être représenté par ce graphe qui résume les relations d'appel entre les procédures d'un programme. Les noeuds représentent les procédures du programme et les arcs représentent la relation appelante-appelée.

Notez que les résultats de l'analyse de flot de contrôle peuvent aussi être exprimés sous forme algébrique.

L'analyse de flot de données, quant à elle, informe sur la propagation des données dans un programme. L'analyse peut se faire localement ou globalement. L'analyse locale se fait à l'intérieur du bloc de base, où l'effet de chaque instruction est analysé en fonction de la composition des effets depuis le début du bloc de base au niveau de l'instruction. L'analyse globale s'intéresse plutôt à l'effet de chaque bloc. Elle étudie la propagation des effets tout le long du graphe de contrôle et le résultat est appliqué aux blocs de base.

En général, le problème d'analyse de flot de données revient à résoudre à chaque point du programme des équations de la forme :

$$out(S) = gen(S) \cup (in(S) - kill(S))$$

où

- S est une instruction ou un bloc d'instructions
- $gen(S)$ est l'ensemble des informations générées par S du programme ;

- $kill(S)$ est l'ensemble de toutes les autres informations dans le programme modifiées par l'exécution de S ;
- $in(S)$ est l'ensemble des informations reçues par S,
  $in(S) = \bigcup_{S' \in pred(S)} out(S')$ avec $pred(S)$ qui représente les instructions précédant S ;
- $out(S)$ est l'ensemble des informations sortant de S.

Ces équations sont résolues pour chaque point du programme de façon itérative. L'analyse peut se faire de deux façons dépendamment de l'information que l'on recherche : une analyse en avant permettant de calculer $out(S)$ en fonction de $in(S)$, ou une analyse en arrière pour le calcul de $in(S)$ en fonction de $out(S)$. Notons toutefois que cette analyse n'offre qu'une estimation conservatrice des effets de chaque bloc de base. Par exemple, dans le cas où S est un branchement conditionnel "if", on considère que les informations générées $gen(S)$ sont l'union des informations générées par chaque branche du "if", et les informations détruites $kill(S)$ sont l'intersection des informations qui ne sont pas propagées par chacune des branches.
Voici quelques exemples d'informations calculées par l'analyse de flots de données :

- Définitions visibles : il s'agit de calculer à chaque point du programme les *définitions* pouvant l'atteindre. Une définition de x est une instruction du programme qui affecte ou peut affecter x. Une affectation à x est un exemple d'instruction qui définit x.
- Expressions disponibles : Une expression est dite disponible à un point du programme p si sa valeur courante a été évaluée avant p et si aucune de ses variables n'a été modifiée par la suite.
- Expressions très utilisées : Une expression est dite très utilisée à un point du programme p si elle est utilisée sur tous les chemins allant de p jusqu'à la fin du programme avant qu'aucune de ses variables ne soit modifiée.
- Variables vivantes : Une variable v est dite vivante à un point du programme p, si la valeur de v est utilisée sur au moins un chemin à partir de p jusqu'à la fin du programme avant d'être modifiée.

Mentionnons que les deux premiers calculs sont des analyses avant, alors que les deux derniers sont des analyses arrière.

Analyse par interprétation abstraite

L'interprétation abstraite, est une théorie introduite par Cousot et Cousot [9], qui consiste à approximer le comportement du programme en le représentant à l'aide d'une

sémantique abstraite, non standard, qui est une généralisation de la sémantique concrète du programme. Cette technique permet ainsi de simuler en quelque sorte l'exécution du programme sur des valeurs abstraites, en focalisant sur les informations recherchées.

La sémantique abstraite est un super-ensemble sur toutes les exécutions possibles du programme, ainsi l'analyse abstraite est suffisante, en cas de succès, pour affirmer que le programme ne comporte aucune faille, même si la sur-approximation est susceptible d'occasionner des faux positifs. Il s'agit donc d'une analyse conservatrice.

Toutes les approches d'analyse statique en général sont des cas particuliers d'interprétations abstraites car elles font appel à une approximation des comportements possibles du programme, à cause de l'indécidabilité évoquée plus haut.

Exemple. Voici un exemple illustrant de façon informelle le processus d'interprétation abstraite. Le but ici est de déterminer le signe d'une expression à valeurs entières. Par exemple, l'expression 7×-2, où $\times$ est l'opérateur arithmétique de multiplication, s'évalue en une valeur de signe négatif.
Le langage est défini par la grammaire suivante :

$$e ::= i \mid e_1 \times e_2 \mid -e$$

où i désigne l'ensemble des entiers relatifs ($\mathbb{Z}$) et **Expr** désigne l'ensemble des expressions. La sémantique concrète du langage du programme est donnée par la fonction μ : **Expr** $\rightarrow \mathbb{Z}$ qui permet d'évaluer une expression du langage ci-haut. Les règles de sémantique opérationnelle sont les suivantes :

$$\begin{aligned} \mu(i) &= i \\ \mu(e_1 \times e_2) &= \mu(e_1) * \mu(e_2) \\ \mu(-e) &= -\mu(e) \end{aligned}$$

La sémantique abstraite s'intéresse à évaluer le signe des expressions du langage. Nous définissons la nouvelle sémantique à l'aide d'une fonction d'abstraction, qui associe des valeurs abstraites aux expressions : $\nu \rightarrow \{neg, pos, nul\}$. Elle est définie comme suit :

$-^a$	pos	neg	nul
	neg	pos	nul

$\times^a$	pos	neg	nul
pos	pos	neg	nul
neg	neg	neg	nul
nul	nul	nul	nul

FIGURE 1.1 – Opérateurs abstraits

$$\begin{aligned} \nu(i) &= \begin{cases} neg & si\ i < 0 \\ pos & si\ i > 0 \\ nul & si\ i = 0 \end{cases} \\ \nu(e_1 \times e_2) &= \nu(e_1) *^a \nu(e_2) \\ \nu(-e) &= -^a \nu(e) \end{aligned}$$

Les opérateurs abstraits sont définis par les tables de la Figure 1.1.
L'évaluation abstraite de l'exemple 7×-2 nous donne ainsi :
$\nu(7 \times -2) = \nu(7) \times^a -^a \nu(2) = pos \times^a -^a pos = pos \times^a neg = neg$
La fonction de concrétisation est définie comme suit :

$$\begin{aligned} \rho(neg) &= \mathbb{Z}^- \backslash \{0\} \\ \rho(pos) &= \mathbb{N} \backslash \{0\} \\ \rho(nul) &= \{0\} \end{aligned}$$

À présent, étendons notre langage en y ajoutant l'opérateur d'addition dont la syntaxe est la suivante : $e_1 +^a e_2$. La règle de sémantique opérationnelle pour l'évaluer est la suivante :

$$\nu(e_1 + e_2) = \nu(e_1) +^a \nu(e_2)$$

La fonction d'abstraction y correspondant est donnée par :

$$\rho(e_1 + e_2) = \rho(e_1) +^a \rho(e_2)$$

Remarquez que si nous tentons d'appliquer l'opérateur $+^a$ à deux valeurs, l'une négative, l'autre positive, il est impossible de statuer le signe de la valeur du résultat. Nous introduisons alors une nouvelle valeur abstraite, notée $\top$, qui représente n'importe quelle valeur (négative, positive ou nulle). Les opérateurs abstraits étendus avec la valeur $\top$ sont donnés par la table 1.2.

$-^a$	*pos*	*neg*	*nul*	$\top$
	neg	*pos*	*nul*	$\top$

$+^a$	*pos*	*neg*	*nul*	$\top$
pos	*pos*	$\top$	*pos*	$\top$
neg	$\top$	*neg*	*neg*	$\top$
nul	*pos*	*neg*	*nul*	$\top$
$\top$	$\top$	$\top$	$\top$	$\top$

$\times^a$	*pos*	*neg*	*nul*	$\top$
pos	*pos*	*neg*	*nul*	$\top$
neg	*neg*	*neg*	*nul*	$\top$
nul	*nul*	*nul*	*nul*	*nul*
$\top$	$\top$	$\top$	*nul*	$\top$

FIGURE 1.2 – Opérateurs abstraits étendus

L'introduction de la valeur abstraite $\top$ rend la fonction de concrétisation plus complexe. En effet, quand une valeur est abstraite en $\top$, il y a perte d'information. Ceci illustre le fait que l'interprétation abstraite peut faire perdre de la précision sur l'information d'intérêt par rapport à l'interprétation concrète.

L'interprétation abstraite se résume donc en quatre étapes présentées sommairement dans l'exemple :

1. Définition d'une sémantique formelle, ou concrète, du langage ;
2. Définition d'une sémantique abstraite dépendamment de l'information qui nous intéresse dans l'analyse statique ;
3. Définition d'une fonction d'abstraction qui associe des valeurs abstraites aux valeurs concrètes ;
4. Définition d'une fonction de concrétisation qui calcule les valeurs concrètes représentées par les valeurs abstraites.

Notez que des propriétés relient les deux dernières fonctions et doivent être démontrées.

Analyse par typage

L'analyse par typage permet de garantir, statiquement, à l'aide de systèmes de types, certaines propriétés concernant le comportement du programme. Une propriété fondamentale de ces systèmes est la correction (*correctness*) du système de types, basée sur la citation bien connue de Robert Milner :

> Les programmes bien typés ne provoquent pas d'erreurs de typage ("Well-typed systems don't go wrong") [22]. En d'autres termes, le typage est une approximation statique de ce que devrait être le comportement du programme à l'exécution illustré par un système de types.

Un système de types est une méthode basée sur la syntaxe du langage de programmation, qui classifie les phrases du langage selon le type de valeurs qu'elles calculent. Il permet ainsi de prouver l'absence de mauvais usage des expressions du programme. Le calcul des types se fait de façon compositionnelle : le type d'une expression dépend seulement du type des sous-expressions qui la composent. L'analyse par typage est conservatrice dans le sens qu'il est possible de prouver l'absence d'un comportement non désirable mais pas sa présence, c'est-à-dire qu'on peut rejeter des programmes corrects. Par exemple, si c'est seulement un des branchements d'une "boucle if...else" dans un programme est mal typé, l'analyse par typage considère que ce programme est mal typé même si, à l'exécution, la condition de la boucle mène toujours vers le branchement bien typé.

La première étape de l'analyse par typage consiste en la définition d'un langage de programmation, c'est-à-dire sa syntaxe et sa sémantique opérationnelle. La syntaxe décrit la structure d'un programme bien formé tandis que la sémantique opérationnelle attache un sens à chaque construction syntaxique. L'étape suivante est la définition de la sémantique statique spécifiant les règles de typage des expressions du langage. Alors que la sémantique opérationnelle décrit une relation « *s'évalue à* », la sémantique statique, elle, décrit une relation « *est de type* ».

Les règles de typage sont associées à un environnement de typage Γ qui associe des variables à des types. $dom(\Gamma)$ désigne l'ensemble $\{x \mid \Gamma \; contient [x \mapsto \tau]\}$ et l'accès au type de x dans Γ se fait par $\Gamma(x)$. Les règles de typage sont composées de *jugements de typage* sous forme de séquents $\Gamma \vdash e : \tau$, se lisant ainsi : sous l'environnement de typage Γ, e est de type τ. Elles sont décrites selon le formalisme suivant :

$$\text{(NOM DE LA RÈGLE)} \quad \frac{P_1 \qquad ... \qquad P_n}{Q}$$

où $P_1, ..., P_n$ et Q sont des jugements de typage. $P_1, ..., P_n$ sont les prémisses et Q la conclusion. La conclusion devient assurément vraie dès que toutes les prémisses au-dessus de la barre horizontale sont satisfaites. Ainsi, les règles de typage affirment la validité de certains jugements sur la base d'autres jugements dont nous connaissons la validité. Voici un exemple tiré des règles de typage qui seront présentées plus loin dans notre approche. Cette règle, portant le nom CTE, indique que sous l'environnement de typage Γ, la va-

leur nommée nch est de type τ si le type associé à nch dans l'environnement de typage des constantes $TypeOf$ est bien τ.

$$\text{(CTE)} \quad \frac{TypeOf(nch) = \tau}{\Gamma \vdash nch : \tau}$$

L'ensemble de règles de typage forme un système (formel) de types. Techniquement, les systèmes de types sont des systèmes de preuves formelles : les règles qu'ils contiennent sont utilisées pour déduire étape par étape le type d'un programme, illustré sur des arbres de preuve. Cela est fait grâce à des algorithmes d'inférence des types, dont le plus utilisé est l'algorithme de Hindley-Milner [11, 10, 22].

1.2.2 Approche dynamique

L'analyse dynamique est l'ensemble de toutes les techniques visant à surveiller le comportement d'un programme durant son exécution. L'analyseur dynamique procède par des tests ou autres actions tout le long de l'exécution du programme ayant comme rôle d'empêcher l'exécution d'une action potentiellement dangereuse.

Il existe trois principales approches d'analyse dynamique résumées par Khoury dans [19] :

- **Le monitorage** (ou *monitoring*) : Un moniteur intercepte chaque appel fait par le programme cible à une fonction qui sollicite le noyau. Il vérifie si l'exécution de cette action viole une politique de sécurité qui lui a été fournie. Si tel est le cas, il peut empêcher l'exécution de cette action ou même arrêter le programme cible. Sinon, l'appel est acheminé au noyau et l'exécution se déroule normalement.
- **L'encapsulation**(ou *wrapping*) : Elle consiste à modifier les fonctions systèmes afin d'assurer le respect de la politique de sécurité. Pour cela, une nouvelle version des fonctions systèmes (dll, classes java) est produite dans laquelle la fonction originale est encapsulée dans une série de tests. Ces tests assurent que chaque appel acheminé à la fonction système respecte la politique de sécurité.
- **L'instrumentation de code** : Elle consiste à instrumenter le programme cible, afin de lui ajouter le code nécessaire pour assurer le respect de la politique. Le code ajouté peut consister en des tests, ou

en toutes autres actions que l'on exige du programme cible pour que celui-ci respecte la politique. Cette approche nous intéresse particulièrement pour la suite de nos travaux, qui consistent à ajouter du code visant à tester dynamiquement les points à sécurité incertaine d'un programme. Nous reviendrons sur cela à la section 2.2.3.

1.3 Analyse de programmes trivaluée

Au cours de nos recherches, nous avons étudié des applications de la logique trivaluée en vérification formelle, en particulier dans l'évaluation de modèles. Ces applications nous ont permis de comprendre la logique trivaluée et montrent l'intérêt et l'utilité que suscite celle-ci. Ceci nous a poussé à l'intégrer dans l'analyse sûre du flot d'information ; cependant, il est à noter que la logique présentée dans cette section est appliquée différemment en analyse du flot d'information. La présente section est donc un supplément sur la logique trivaluée permettant de comprendre son rôle mais n'est pas essentiel pour comprendre son application dans notre approche.

Les logiques multivaluées permettent de raisonner sur plusieurs valeurs autres que *vrai* et *faux*. En général, les logiques multivaluées sont utiles en ingénierie du logiciel car elles permettent d'exprimer l'incertitude ou de résoudre l'ordre de priorité en cas de désaccord. La logique trivaluée, comme son nom l'indique, est une logique permettant de raisonner sur un domaine à trois valeurs. Par exemple, cette logique peut être utilisée dans un mécanisme qui, au lieu de se prononcer positivement ou négativement sur un énoncé logique, a la possibilité de dire « peut-être ».

1.3.1 Logique trivaluée pour l'évaluation de modèles

L'*évaluation de modèles* ou "*model-checking*" est une technique de vérification formelle de systèmes. Le comportement de ces derniers est représenté par un modèle discret et la propriété à vérifier, par un formalisme logique. Ensuite, un algorithme appelé *vérificateur de modèle* ("*model-checker*") permet de dire si, oui ou non, la propriété est respectée par le système en question.

Toutefois, comme la vérification de modèles est une méthode formelle appliquée tôt dans la phase de conception, il est fréquent de devoir raisonner

sur des systèmes avec une information incertaine. Ces incertitudes peuvent provenir d'une information incomplète sur le système. Cela peut provenir également du fait que certains comportements ne sont explicitement ni permis, ni interdits, ou de systèmes abstraits par souci d'optimisation, dont l'abstraction entraîne une perte d'information. Les logiques multivaluées, en particulier la logique trivaluée, ont été introduites en évaluation de modèles afin d'offrir un cadre permettant de raisonner sur ces systèmes dont l'espace d'états ne peut être représenté que partiellement [5, 6].

L'*évaluation de modèles trivaluée* se fait par une procédure qui reçoit un système de transition partiel K représentant le comportement du programme et une formule φ venant d'une logique temporelle trivaluée, et retourne une réponse sur le respect de la formule φ par K : *oui*, *non* ou *peut-être*.

Structure de Kripke partielle et logique modale trivaluée

Pour la vérification d'espaces d'états partiels, il est nécessaire d'avoir une façon de modéliser l'absence d'information sur les parties manquantes de l'espace d'états au complet, à la fois au niveau opérationnel (en termes de structures de Kripke, dont la définition est donnée ci-bas) et au niveau logique (en termes de logiques modales).

Définition 1.3.1. *Une* structure de Kripke *(KS) sur un ensemble de propositions atomiques P est un tuple $(S, L, \mathcal{R})$, où S est un ensemble d'états, $\mathcal{R} \subseteq S \times S$ est une relation de transition sur S et $L : S \times P \to \{\text{vrai}, \text{faux}\}$ est une interprétation qui associe une valeur de vérité (*vrai *ou* faux*) à chaque proposition atomique dans P pour chaque état dans S.*

Les *structures de Kripke* ont à l'origine été conçues comme modèles pour la logique modale [17], qui utilise des modalités exprimant la nécessité et la possibilité. Intuitivement, les états dans les structures de Kripke correspondent à différents "mondes" dans lesquels différents faits (les propositions atomiques) sont vrais. L'affirmation qu'un fait est possiblement vrai est interprété pour signifier qu'il y a au moins un état accessible (monde) dans lequel le fait s'accomplit ; l'affirmation qu'un fait est nécessairement vrai signifie que le fait s'accomplit dans tous les états accessibles.

Nous présentons ici l'approche utilisée par Bruns et Godefroid dans [5]. Les auteurs modélisent les espaces d'états partiels par des structures de Kripke

partielles, puis définissent une logique modale trivaluée dont la sémantique est définie à l'égard de ces structures de Kripke partielles. Ils étudient ensuite une relation d'équivalence et un pré-ordre découlant implicitement de la logique modale trivaluée.

Définition 1.3.2. *Soit P un ensemble de propositions atomiques. Une* structure de Kripke partielle *M est un tuple $(S, L, \mathcal{R})$, où S est un ensemble d'états, $L : S \times P \to \{vrai, \bot, faux\}$ est une interprétation qui associe à chaque proposition atomique dans P pour chaque état dans S une valeur de vérité :* vrai*, $\bot$ ou* faux*, et $\mathcal{R} \subseteq S \times S$ est une relation de* transition *sur S.*

Dans la logique modale propositionnelle sur les structures de Kripke partielles, $\bot$ veut dire « peut être vrai ou faux ». La manière simple de définir la conjonction (resp. disjonction) dans cette logique est de la définir comme le minimum (resp. maximum) de ses arguments, sous la relation d'ordre suivante : $faux < \bot < vrai$. La négation, quant à elle, est définie en utilisant la fonction neg qui associe $vrai$ à $faux$, $faux$ à $vrai$ et $\bot$ à $\bot$.
La sémantique de cette logique modale indique la valeur de vérité d'une formule ϕ de cette logique dans un état s d'une structure de Kripke $M = (S, L, \mathcal{R})$. Cette sémantique est notée $[M, s \models \phi]$ et elle est définie de manière inductive comme-suit :

$$\begin{array}{lcl} [(M,s) \models p] & = & L(s,p) \\ [(M,s) \models \neg\phi] & = & neg([(M,s) \models \phi]) \\ [(M,s) \models \phi_1 \wedge \phi_2] & = & min([(M,s) \models \phi_1], [(M,s) \models \phi_2]) \\ [(M,s) \models \Diamond\phi] & = & max(\{[(M,t) \models \phi]\} \mid (s,t) \in \mathcal{R}) \end{array}$$

Un pré-ordre est implicitement défini sur les structures de Kripke partielles en utilisant la logique trivaluée. Soit $\leqslant$ l'ordonnancement sur les valeurs de vérité tel que $\bot \leqslant vrai$, $\bot \leqslant faux$, $x \leqslant x$ et $x \not\leqslant y$ sinon (pour tout $x, y \in \{vrai, \bot, faux\}$). Notez que l'ordre $\leqslant$ diffère du précédent $<$ car le premier reflète le degré de complétude. À titre d'exemple, $vrai$ est plus précis que $\bot$.

Définition 1.3.3. *Soient $M_1 = (S_1, L_1, \mathcal{R}_1)$ et $M_2 = (S_2, L_2, \mathcal{R}_2)$, deux structures de Kripke partielles. Le* pré-ordre de complétude *est la plus grande*

relation $\preceq \subseteq S_1 \times S_2$ *telle que* $s_1 \preceq s_2$ *implique :*

- $\forall p \in P : L_1(s_1, p) \leqslant L_2(s_2, p)$,
- *Si* $(s_1, s_1') \in \mathcal{R}_1$, *alors il existe un* $s_2' \in S_2$ *tel que* $(s_2, s_2') \in \mathcal{R}_2$ *et* $s_1' \preceq s_2'$,
- *Si* $(s_2, s_2') \in \mathcal{R}_2$, *alors il existe un* $s_1' \in S_1$ *tel que* $(s_1, s_1') \in \mathcal{R}_1$ *et* $s_1' \preceq s_2'$.

$s_1 \preceq s_2$ signifie intuitivement que les propositions atomiques dans l'état s_1 peuvent être moins définies que dans l'état s_2.

Théorème 1.3.1. [5] *Soient* $M_1 = (S_1, L_1, \mathcal{R}_1)$ *et* $M_2 = (S_2, L_2, \mathcal{R}_2)$, *deux structures de Kripke partielles* $s_1 \in S_1$ *et* $s_2 \in S_2$, *et soit* Φ *l'ensemble de toutes les formules en logique modale propositionnelle. Alors, on a :*

$$(\forall \phi \in \Phi : [(M_1, s_1) \models \phi] \leqslant [(M_2, s_2) \models \phi]) \text{ ssi } s_1 \preceq s_2$$

En d'autres mots, des structures de Kripke partielles qui sont « plus complètes » selon la relation $\preceq$ ont plus de propriétés définies selon $\leqslant$, i.e., ont plus de propriétés qui s'évaluent à soit *vrai*, soit *faux*. De plus, toute formule qui s'évalue à *vrai* ou *faux* sur une structure de Kripke partielle aura la même valeur de vérité si elle est évaluée sur une structure de Kripke plus complète. Les formules s'évaluant à $\perp$ doivent être évaluées sur une structure de Kripke plus complète pour obtenir une réponse plus précise.

Évaluateur de modèles CTL trivalué

Nous venons de voir les propriétés exprimées en logique modale propositionnelle trivaluée mais Bruns et Godefroid [5] ont aussi étudié la version trivaluée de la logique computationnelle ou *computational tree logic (CTL)* [7]. Les auteurs définissent, en guise d'exemple, la sémantique trivaluée de CTL appliquée à un algorithme de model-checking pour la vérification d'une propriété de la forme $\mathcal{A}(f_1 \ \mathcal{U} \ f_2)$.

En se basant sur la sémantique logique computationnelle, voici la définition de la relation de satisfaction de la formule :

$$s_0 \models \mathcal{A}(f_1 \ \mathcal{U} \ f_2), \text{ si pour tous les chemins } (s_0, s_1, ...),$$
$$\exists i \geqslant 0 : s_i \models f_2 \wedge \forall j : 0 \leqslant j < i \Rightarrow s_j \models f_1.$$

Cela signifie qu'un état respecte la formule $\mathcal{A}(f_1\ \mathcal{U}\ f_2)$ si, pour tous les chemins provenant d'un état, il existe un état pour lequel f_2 est respecté et tous les états qui le précèdent dans ce chemin respectent f_1.

Pour le CTL trivalué, la formule est définie comme suit :

$$\begin{aligned}
[s_0 \models \mathcal{A}(f_1\ \mathcal{U}\ f_2)] &= \min(\{[(s_0, s_1, ...) \models f_1\ \mathcal{U}\ f_2] \mid \\
&\qquad (s_0, s_1, ...) = \text{un chemin}\}) \\
[(s_0, s_1, ...) \models f_1\ \mathcal{U}\ f_2] &= \max(\{[(s_0, s_1, ...) \models f_1\ \mathcal{U}_k\ f_2] \mid k \geqslant 0\}) \\
[(s_0, s_1, ...) \models f_1\ \mathcal{U}_k\ f_2] &= \min_{s_k}(\min_{s_i}(\{[s_i \models f_1] \mid i < k\})\ , \\
&\qquad \{[s_k \models f_2]\})
\end{aligned}$$

Les opérateurs min dans cette définition correspondent à la conjonction et à la quantification universelle dans le CTL classique bivalué, tandis que max correspond à la quantification existentielle.

Pour vérifier cette formule, la procédure standard *au* du model-checker CTL [8] est appelée. Elle fonctionne en gros comme suit : elle est appelée avec une formule f de la forme $\mathcal{A}(f_1\ \mathcal{U}\ f_2)$, un état s_0 et un résultat dans une variable b. Une recherche en profondeur est effectuée sur tous les états à partir de l'état s_0, et les états sont marqués au fur et à mesure qu'ils sont visités. Initialement, si s_0 est étiqueté avec f_2 alors s_0 satisfait d'avance $\mathcal{A}(f_1\ \mathcal{U}\ f_2)$ et la procédure se termine avec la variable b mise à $vrai$. Sinon, si s_0 n'est pas étiqueté avec f_1, alors la procédure se termine en mettant b à $faux$. Sinon, la procédure est appelée récursivement sur tous les successeurs de s_0. Si un appel récursif est effectué sur un état déjà visité, la procédure se termine avec b contenant $faux$.

Cet algorithme a été réadapté pour correspondre au cas de 3-valeurs. $\perp$ est une valeur qui peut être soit *vrai*, soit *faux*. L'idée est donc de vérifier ces deux cas sur la structure de Kripke partielle, ceci en deux volets : de façon pessimiste d'une part, i.e., $\perp$ est considéré comme étant *faux* et d'autre part de façon optimiste, où $\perp$ est pris pour *vrai*. Si le résultat donné dans les deux cas est *vrai* alors la valeur retournée est *vrai* et s'applique aussi à l'espace d'états complet, si l'on obtient *faux* alors la valeur *faux* est retournée. Mais si l'interprétation pessimiste retourne *faux* alors que l'interprétation optimiste a trouvé *vrai*, alors la valeur $\perp$ est retournée, ce qui veut dire que l'espace d'états ne contient pas assez d'informations pour se prononcer définitivement sur la propriété. Notez que le résultat : pessimiste = *vrai*, optimiste = *faux* ne

peut survenir, car une vérification optimiste doit nécessairement retourner *vrai* si la pessimiste a retourné *vrai*.

1.3.2 Logique trivaluée pour l'évaluation de modèles de programmes en Java

Yahav dans [40] présente un cadre de vérification de propriétés de sûreté de programmes Java concurrents dans lequel apparaît la logique trivaluée. Le but est de développer des techniques de vérification à la compilation en détectant des configurations du programme qui pourraient violer des propriétés données. Ce cadre de travail gère le fait que les programmes analysés allouent dynamiquement des objets *threads*, et peuvent donc en créer un nombre illimité.

La sémantique opérationnelle décrivant le comportement de programmes « *multi-thread* » Java est représentée en termes de configurations (ou états). Chaque configuration est caractérisée par un ensemble de prédicats logiques écrits selon un méta-langage basé sur une logique de premier ordre avec fermeture transitive. À titre d'exemple, le prédicat $is\_thread(t)$ désigne le fait que l'objet t est un *thread*. *blocked(t,l)* est un prédicat binaire qui désigne le fait que le *thread* t est bloqué par le verrou l.

Par ce même méta-langage, des propriétés sont extraites des configurations sous forme de formules logiques. Par exemple, la formule $\exists t : is\_thread(t) \wedge held\_by(l, t)$ exprime le fait qu'un *thread* t a fait l'acquisition du verrou l. De la même manière, des propriétés de sûreté sont formalisées à l'aide de formules logiques, telles que la non-interférence entre deux *threads* concurrents, l'absence d'interblocages ou *deadlocks*, etc. Par exemple, voici la formule permettant de détecter un *interblocage*, c'est-à-dire une situation où tous les *threads* sont bloqués mutuellement, chaque *thread* attendant une ressource bloquée par un autre qui lui-même est en attente :

$\forall t : is\_thread(t) \rightarrow \exists l : blocked(t, l)$

Durant l'exécution d'un programme, l'effet d'une déclaration est calculé de façon approximative, produisant ainsi l'ensemble (souvent infini) des *configurations atteignables*. En effet, grâce à un algorithme de recherche en profondeur, l'espace d'états est exploré. Pour toute configuration C telle que C n'est pas déjà un *membre* de l'*espace d'états*, toute configuration C' pouvant

être produite en effectuant une certaine action sur C est explorée et rajoutée à l'espace d'états.

Étant donné que le nombre d'objets pouvant être créé est illimité, l'auteur a introduit la théorie de l'interprétation abstraite. Le modèle d'un programme est représenté de façon conservatrice par un modèle abstrait trivalué de la façon qui suit : plusieurs configurations qui utilisent une même structure logique, c'est-à-dire satisfaisant les mêmes prédicats logiques, sont fusionnées en une même configuration abstraite ayant une valeur de vérité en plus : 1/2. Si un prédicat est évalué à 1/2 pour une configuration donnée, c'est que cette configuration abstrait plusieurs objets et que le prédicat s'évalue à 1 (vrai) pour certains et à 0 (faux) pour d'autres.

La vérification des propriétés est alors adaptée en conséquence pour se faire sur le modèle abstrait du programme. Cette approche a été implémentée dans un prototype nommé 3VMC [39].

1.3.3 Logique trivaluée pour la spécification et la vérification de propriétés de sécurité

Les travaux effectués par Ktari dans [20] abordent le problème de la détection de code malicieux dans des logiciels COTS (« commercial off-the-shelf »). Une approche logique est adoptée qui utilise la logique trivaluée pour la spécification et la vérification de propriétés de sécurité. Il s'agit d'une combinaison de certification de code et d'évaluation de modèles. L'idée est d'extraire un modèle à partir des informations contenues dans le certificat (fourni par le producteur du code). Ce modèle représente ainsi les aspects de sécurité du programme. Ensuite, ce modèle est comparé à une spécification logique de politique de sécurité afin de vérifier si le programme en question est sécuritaire.

Modèle

Le comportement du système est modélisé par un système de transitions étiquetées, dont les arcs sont annotés d'actions. La particularité de ce modèle est l'action δ, qui a pour rôle d'abstraire les actions ne pouvant être connues au moment de la compilation, par exemple si des pointeurs sont utilisés.

Le modèle du système est défini comme suit :

ψ	$::=$	$true \mid false \mid \neg\psi \mid \psi \wedge \psi' \mid \psi \vee \psi'$	(Expressions booléennes)
	$\mid$	X	(Variable)
	$\mid$	$<K>\psi$	(Opérateur de possibilité)
	$\mid$	$[K]\psi$	(Opérateur de nécessité)
	$\mid$	$\mu X.\psi$	(Plus petit point fixe)
	$\mid$	$\nu X.\psi$	(Plus grand point fixe)
K	$::=$	$\{a_i \mid i = 1..n\} \mid K - K'$	(Ensemble d'actions)
a_i	$::=$	*Fonctions critiques*	(Actions)

TABLE 1.1 – Syntaxe de la logique L_μ

Définition 1.3.4. *Le système est un quadruplet* $M = \langle \mathcal{S}, \mathcal{A}ct \cup \{\delta\}, \rightarrow, s_0 \rangle$, *où* $\mathcal{S}$ *est un ensemble d'états,* $\mathcal{A}ct$ *un ensemble d'actions* ($\delta \notin \mathcal{A}ct$), $\rightarrow \subseteq \mathcal{S} \times (\mathcal{A}ct \cup \{\delta\}) \times \mathcal{S}$ *une relation de transition, et* $s_0 \in S$ *l'état initial du modèle.*

Une transition $s \xrightarrow{a} s'$ indique que le programme peut évoluer de l'état s vers s' par l'exécution de l'action a.

Syntaxe de L_μ

Nous présentons ici la syntaxe de la logique utilisée pour spécifier des propriétés de sécurité. Il s'agit d'une extension du μ-calcul modal [14] qui tient compte de l'incertain. Elle est désignée par L_μ et est présentée à la Table 1.1.

Les symboles $\neg$, $\vee$ et $\wedge$ représentent la négation, la disjonction et la conjonction, respectivement. Les formules $<K>\psi$ et $[K]\psi$ utilisent des opérateurs modaux de la logique indexés par un ensemble d'actions K, connus respectivement comme les opérateurs de possibilité et de nécessité. Un état du programme satisfait la formule $<K>\psi$ s'il possède au moins une transition indexée par une action de l'ensemble K menant vers un état satisfaisant la formule ψ. De la même manière, un état satisfait la formule $[K]\psi$ si toutes les transitions partant de lui indexées par les actions de K mènent vers un état satisfaisant la formule ψ. Les formules $\mu X.\psi$ et $\nu X, \psi$ sont des formules récursives et sont connues comme les opérateurs du plus petit et du plus grand point fixe. Les opérateurs de point fixe μ et ν lient toutes les occurrences libres de X dans la formule ψ. Une occurrence de X est libre si elle ne se situe pas dans la portée d'un opérateur μX ou νX. Notons que pour assurer l'existence du plus petit et du plus grand points fixes, toute variable

X doit être sous la portée d'un nombre pair de négations dans la formule ψ. Le terme $K - K'$ indique l'ensemble résultant de la différence de K par rapport à K'.

Une formule de cette logique exprimant une propriété de sécurité serait, par exemple, $[readpassword]$ $<checkpassword>$ $true$, ce qui signifie que pour chaque action *readPassword*, au moins une action *checkpassword* peut être effectuée. D'autres exemples de formules logiques sont donnés dans [20].

Sémantique de L_μ

L'auteur définit une sémantique dénotationnelle trivaluée, qui est celle du μ-calcul modal adaptée afin de permettre de raisonner sur l'incertain. Selon cette sémantique, à cause des transitions δ présentes dans le modèle du programme, une formule logique s'évalue à une paire p d'ensembles d'états : le premier ensemble, correspondant au premier élément de la paire (noté $\pi_1(p)$), contient les états pour lesquels la formule est vraie, l'autre ensemble, correspondant au second élément de la paire (noté $\pi_2(p)$ contient les états pour lesquels la formule est peut-être vraie, y compris les états la satisfaisant. Les formules sont interprétées sous un environnement $e = [\mathcal{V} \rightarrow 2^S]$ qui associe aux variables de $\mathcal{V}$ des ensembles d'états.

La fonction sémantique $[\![.]\!]_e^M : \mathcal{F} \rightarrow 2^S \times 2^S$ est décrite dans [20]. Ainsi, pour toute formule ψ, nous avons la propriété suivante :

$[\![\psi]\!]_e^M = (\pi_1([\![\psi]\!]_e^M), \pi_2([\![\psi]\!]_e^M))$

La vérification d'une propriété de sécurité se base alors sur cette sémantique dénotationnelle et se fait globalement grâce à un algorithme d'évaluation de modèles globale. Cela nécessite donc d'analyser tous les états du modèle pour décider si ce modèle satisfait ou peut satisfaire une formule donnée. Toutefois, des modèles de vérification locale existent et l'auteur présente également un système de preuves basé sur les tableaux, permettant ainsi de décider si un état en particulier, généralement l'état initial, satisfait ou peut satisfaire une formule de la logique L_μ.

1.4 Analyse sûre du flot d'information

Parmi les mécanismes existants d'application de la sécurité, nous nous intéressons au contrôle du flot d'information car celui-ci concerne la protection de l'information au niveau de sa propagation. En effet, les mécanismes comme le contrôle d'accès, les pare-feux, le chiffrement des données et les logiciels anti-virus n'offrent des restrictions qu'au niveau de l'accès à l'information et non au niveau de sa propagation pendant l'exécution. Par exemple, le contrôle d'accès à un fichier ne garantit pas qu'une fois l'information lue, il n'y aura pas fuite d'information lors de sa propagation. De même, le mécanisme de chiffrement ne peut garantir que le receveur des données les utilisera à bon escient une fois déchiffrées. Un utilisateur peut accéder à une information confidentielle parce qu'il y est autorisé mais, en l'utilisant de façon non sécuritaire, la révéler au public. D'où la nécessité de se pencher sur l'analyse sûre (ou sécurisée) du flot d'information. Il s'agit d'une technique utilisée pour prévenir la mauvaise utilisation des données dans un système informatique durant son exécution. Cela est réalisé en répartissant les variables et autres entités du programme dans diverses classes de sécurité puis en restreignant la façon dont les informations transitent entre ces variables d'après le niveau de sécurité de la classe. Le but de cette section est de présenter les principes de l'analyse sûre du flot d'information, particulièrement l'analyse sûre par typage que nous adoptons comme mécanisme de contrôle.

Denning a introduit cette approche [13] en montrant comment l'analyse statique peut être utilisée pour contrôler le flot d'information selon une politique de sécurité. D'autres études qui suivent cette approche sont citées dans [26]. La propriété de sécurité que ces analyses cherchent à assurer est la *non-interférence*, une propriété fondamentale du flot d'information qui garantit qu'aucune information, durant l'exécution, ne passe d'un objet à un autre objet de niveau de sécurité inférieur.

Parmi les travaux effectués sur le contrôle du flot d'information, nous pouvons citer les extensions de langages de programmation qui intègrent une forme d'analyse statique : Smith [30] et Volpano et Smith [37] qui ont conçu une analyse par typage pour un langage impératif ; Pottier et Simonet qui traitent dans [24] le langage fonctionnel ML, prenant en charge les références, les exceptions et le polymorphisme ; Myers dans [23] applique statiquement la politique de flot d'information dans JFlow, une extension de Java anno-

tant les variables avec des niveaux de sécurité, rendant la vérification du flot d'information plus précise et flexible ; Banerjee et Naumann étendent le langage orienté objet Java dans [1]. Barthe et al, dans [2] et Terauchi et Aiken dans [36] ont étudié la formulation logique de la non-interférence afin d'utiliser des techniques de vérification automatiques ou interactives tels que les prouveurs de théorèmes et l'évaluation de modèles (ou *model-checking*), qui sera expliquée plus loin. Dans [28], Sabelfeld et Sands étendent les approches de systèmes de types afin de supporter les fonctions d'ordre supérieur et le non-déterminisme. Finalement, nous mentionnons les travaux portant sur d'autres notions de la non-interférence, la non-interférence possibiliste et probabiliste, qui traitent la sûreté du flot d'information pour des programmes concurrents, voir [3, 27, 32, 33]

1.4.1 Principes de base

Notions de base sur les treillis

Nous rappelons ici les notions de relation d'ordre et de treillis.

Définition 1.4.1. *(Ensemble partiellement ordonné)*[1] $(\mathcal{L}, \sqsubseteq)$ *est un ensemble partiellement ordonné si et seulement si :*

1. $\mathcal{L}$ *est un ensemble*
2. $\sqsubseteq$ *est une relation d'ordre partiel sur* $\mathcal{L}$*, c'est-à-dire qu'elle est :*
   - *réflexive :* $\forall x \in \mathcal{L}, x \sqsubseteq x$,
   - *antisymétrique :* $\forall x, y \in \mathcal{L}, x \sqsubseteq y \wedge y \sqsubseteq x \Rightarrow x = y$,
   - *transitive :* $\forall x, y, z \in \mathcal{L}, x \sqsubseteq y \wedge y \sqsubseteq z \Rightarrow x \sqsubseteq z$.

Définition 1.4.2. *(Majorant, Minorant) Étant donné un ensemble partiellement ordonné* $(\mathcal{L}, \sqsubseteq)$ *:*

- *L'ensemble des majorants de* P *dans cet ensemble, noté* $P\uparrow$*, est :* $P\uparrow = \{x \in \mathcal{L} : \forall y \in P : y \sqsubseteq x\}$
- *L'ensemble des minorants de* P *dans cet ensemble, noté* $P\downarrow$*, est :* $P\downarrow = \{x \in \mathcal{L} : \forall y \in P : x \sqsubseteq y\}$.

1. Un ensemble est dit totalement ordonné si, en plus d'être partiellement ordonné, on a : $\forall x, y \in \mathcal{L} : (x \sqsubseteq y) \vee (y \sqsubseteq x)$

Définition 1.4.3. *(Borne supérieure, Borne inférieure) Soient un ensemble partiellement ordonné* $(\mathcal{L}, \sqsubseteq)$ *et* $P \subseteq \mathcal{L}$ *:*

- *La borne supérieure d'un ensemble* P *dans* $(\mathcal{L}, \sqsubseteq)$*, notée* $\sqcup P$*, désigne le plus petit des majorants de l'ensemble, s'il existe,*
- *La borne inférieure d'un ensemble* P *dans* $(\mathcal{L}, \sqsubseteq)$*, notée* $\sqcap P$*, désigne le plus grand des minorants de l'ensemble, s'il existe.*

Définition 1.4.4. *(Treillis) Un treillis* $(\mathcal{L}, \sqsubseteq)$ *est un ensemble partiellement ordonné où chaque ensemble fini* $P \subseteq \mathcal{L}$ *a une borne supérieure (aussi appelée « join » ou union) et une borne inférieure (aussi appelée « meet » ou intersection).* $\top$ *et* $\bot$ *sont les éléments maximal et minimal du treillis, respectivement, s'ils existent.*

Niveaux de sécurité

Le principe fondamental de l'analyse sûre du flot d'information est de classifier les variables du programme en différents *niveaux de sécurité*. Suivant l'approche de Denning [12] ainsi que Bell et LaPadula [4], ces niveaux de sécurité forment un treillis $(SC, \leqslant)$ sur lequel la politique portant sur le flot d'information est définie. SC désigne l'ensemble des classes de sécurité. Cet ensemble est partiellement ordonné par $\leqslant$. Par exemple, les données peuvent être classées selon leur degré de confidentialité et ainsi nous distinguons le niveau de sécurité élevé noté H pour « *high* » (données confidentielles) et le niveau faible noté L pour « *low* » (données publiques), selon l'ordre $L \leqslant H$. L est l'élément $\bot$, tandis que H est l'élément $\top$. De plus, $\sqcap\{H, L\} = L$ et $\sqcup\{H, L\} = H$. Les données peuvent aussi être classées selon le degré d'intégrité, avec un niveau de sécurité *intact* (l'information ne peut avoir été altérée) ou *corrompu* (l'information peut avoir été altérée).

La politique de sécurité consiste à proscrire les fuites d'information de données H vers L (ou intact à corrompu). De façon plus générale, on cherche à éviter tout transfert d'information d'un niveau de sécurité donné vers des données de niveau de sécurité inférieur.

Il existe deux types de flots d'information à contrôler :

- Les flots d'information explicites, qui se produisent lorsqu'une instruction transfère directement le contenu d'un objet (variable, canal, etc.) dans un autre objet.

– Les flots d'information implicites, qui sont plus subtils car ils se produisent indirectement lors de l'exécution, ou la non-exécution, d'une instruction qui cause un flot explicite tout en étant conditionnée par la valeur d'une expression.

Pour illustrer cette différence, considérons l'exemple de [12] :
if $a = 0$ **then** $b := c$.
Il existe un flot d'information explicite de `c` vers `b` car le contenu de `c` est assigné à `b`. Il y a aussi un flot implicite de `a` vers `b` puisque le contenu assigné à `b` dépend de la valeur de `a`. Dans ce sens, le contenu de `b` révèle une information concernant la valeur de `a`.

Non-interférence

La non-interférence est une propriété qui garantit qu'en effectuant des changements sur les entrées confidentielles d'un programme, cela n'apporte aucun changement sur les sorties publiques du programme. Intuitivement, ceci assure qu'il n'y a pas de dépendance des données publiques sur les données confidentielles du programme et donc qu'il est impossible de déduire de l'information confidentielle en observant les données publiques.
Prenons l'exemple suivant : $x := secret$; **if** 1 **then** $public := x$ **end**. Il est clair que si la valeur de sortie de `public` venait à être lue, il y aurait violation de la non-interférence puisque `public` dépend de l'entrée privée `secret`.
Considérons un autre exemple :

if $secret = 0$ **then**
  $public := 0$
else
  $public := 1$
end

Nous remarquons, dans cet exemple, que la valeur de la donnée visible `public` à la sortie du programme dépend de la valeur contenue dans la variable confidentielle `secret`.
La non-interférence peut être formalisée avec la sémantique des langages de programmation. Ici deux niveaux de sécurité H et L sont considérés. Nous partons du principe que l'exécution d'un programme fait passer celui-ci d'un état à un autre. L'état d'entrée est une paire $s = (s_h, s_l)$ représentant les

valeurs d'entrée du programme de niveau H et celles de niveau L, respectivement. Lors de l'exécution, soit le programme s'exécute avec succès et termine dans un état $s' = (s'_h, s'_l)$ représentant les valeurs de sortie du programme, soit le programme diverge. Par conséquent, la sémantique d'un programme P est une relation de transition $[\![P]\!] : S \to S_\perp$ (où S est l'ensemble des états du programme, $S_\perp = S \cup \{\perp\}$ et $\perp \notin S$). Cette relation associe à un état d'entrée $s \in S$, soit un état de sortie $[\![P]\!]s \in S$, soit l'état $\perp$ si le programme ne s'exécute pas jusqu'à la fin. Un changement sur les données d'entrée confidentielles peut être décrit par une relation d'équivalence $=_L$ entre deux états définie de la façon suivante :

Définition 1.4.5. *Étant donné deux états* $s, s' \in S$, $s =_L s'$ ssi $s_l = s'_l$.

La capacité qu'a un attaquant de distinguer deux états est alors caractérisée par une relation $\sim_L$ sur deux comportements du système. La notion d'indistinguabilité de comportements dépend de la propriété de sécurité désirée ; elle pourrait correspondre tout simplement à la relation d'équivalence, c'est-à-dire : $s \sim_L s'$ ssi $s, s' \in S$ implique $s =_L s'$.

Formellement, la non-interférence est définie comme suit :

Définition 1.4.6. *Soit une relation d'équivalence* $=_L$ *sur deux états* $s_1, s_2 \in S$. *Un programme* P *est sécurisé ssi* $\forall s_1, s_2 \in S \mid s_1 =_L s_2 \Longrightarrow [\![P]\!]s_1 \sim_L [\![P]\!]s_2$.

Ce qui signifie « si deux états d'entrée possèdent les mêmes valeurs publiques, alors l'exécution du programme sur ces états produit des comportements indistinguables pour un observateur malveillant ».

Mentionnons que la notion de non-interférence décrite ci-haut s'applique surtout à des programmes déterministes car nous considérons que l'exécution du programme conduit à un seul état. Toutefois, elle a été adaptée pour des programmes non déterministes. Nous pouvons citer ici les travaux sur la non-interférence *possibiliste* et la non-interférence *probabiliste* [3, 27, 31, 32, 38]. La non-interférence *possibiliste* considère les sorties possibles de programmes concurrents. La non-interférence *probabiliste* considère une distribution de probabilité sur les sorties possibles d'une exécution du programme.

1.4.2 Détection de flot d'information non sûr par typage

Parmi les méthodes d'analyse statique du flot d'information, certaines emploient des systèmes de type. Comme décrit à la section 1.2.1, un système de types est un système formel composé de règles de typage permettant de déduire des jugements sur les programmes. Dans notre cas, nous nous intéressons à la politique concernant le flot d'information sûr (ou sécurisé). Chaque expression du programme reçoit un type de sécurité. Celui-ci est une étiquette qui décrit comment l'expression peut être utilisée conformément à la politique de sécurité. La vérification du respect de cette politique revient alors à vérifier si le programme est bien typé.

L'analyse à l'aide de systèmes de types a plusieurs avantages cités dans [37]. Il s'agit d'une spécification formelle qui distingue clairement les politiques de sécurité des algorithmes qui les mettent en application dans les programmes. Dans l'analyse de flot d'information par typage, cette distinction introduit une notion de *cohérence* ("*soundness*") qui s'apparente à la non-interférence. De façon intuitive, la cohérence exprime le fait que les variables dans un programme bien typé n'« interfèrent » pas avec les variables de niveau de sécurité inférieur. Ceci est formalisé dans un théorème de sûreté de types puis prouvé. Les règles de typage du flot d'information sûr combinent la correction classique de type ("*type correctness*") et l'application du flot d'information sûr. De plus, le contrôle du flot d'information allant du niveau de sécurité inférieur à supérieur (de L vers H par exemple) est facilement intégré dans les règles de typage. Finalement, un système de types peut être automatisé en utilisant des techniques d'*inférence de type*, pour analyser le flot d'information sûr dans les programmes.

Dans ce qui suit, nous considérons le système de types de Volpano et al. [37] afin d'illustrer l'analyse de flot d'information sûr par typage.

Types des flots d'information sûrs

En général, les données du programmes sont typées selon deux catégories. Dans la première catégorie figurent les *types des données*, notés τ, qui correspondent aux différentes classes de sécurité de SC. Ces types sont assignés aux expressions du programme. L'autre catégorie est composée des *types des phrases*, notés ρ. Il inclut les types des données ainsi que les types affectés

aux variables, de la forme $\tau\ var$, et aux commandes (instructions pouvant assigner des valeurs aux variables), de la forme $\tau\ cmd$.

Une variable de type $\tau\ var$ contient de l'information de sécurité τ ou inférieure, tandis qu'une commande de type $\tau\ cmd$ ne peut contenir que des assignations vers des variables de sécurité τ ou supérieure. La restriction sur les variables est nommée *Simple sécurité*, et la dernière sur les commandes est connue sous le nom de *propriété de confinement* ; cette dernière permet de contrôler les flots d'information implicites.

Règles de typage des flots d'information sûrs

Les règles de typage décrivent comment les instructions doivent être utilisées afin de garantir des flots d'information implicites et explicites sûrs. Considérons les règles de typage données dans [37]. Les jugements de typage sont de la forme $\lambda, \gamma \vdash p : \rho$ où γ représente un typage d'emplacement et γ représente un typage d'identificateur de variable. Ce jugement signifie : une phrase p a le type ρ, en supposant que λ donne les types des emplacements et γ donne les types de tous les identificateurs libres de p. Un typage d'identificateur est une fonction finie associant des types ρ aux identificateurs, tandis qu'un typage d'emplacement est une fonction finie associant des types τ aux emplacements. $\delta(x)$ désigne le type assigné à x par δ.

En plus des règles de typage, il existe des règles de sous-typage de la forme $\rho_1 \subseteq \rho_2$. Il s'agit de l'ordre partiel $\leqslant$ qui est étendu à une relation de sous-typage $\subseteq$. Cette relation est contravariante dans les types des commandes, c'est-à-dire, si $\tau \leqslant \tau'$ alors $\tau'\ cmd \subseteq \tau\ cmd$. Ces règles permettent la coercion de type. Ainsi, une phrase de type τ peut être considérée de type τ' si $\tau \subseteq \tau'$. Inversement, une commande de type $\tau\ cmd$ peut avoir le type $\tau'\ cmd$ si $\tau'\ cmd \subseteq \tau\ cmd$, c'est-à-dire si $\tau' \subseteq \tau$.

Afin de montrer comment le typage permet de détecter des flots d'information illicites, examinons quelques règles de typage. Voyons la règle de typage suivante pour l'assignation :

$$\text{(ASSIGN)} \quad \frac{\lambda; \gamma \vdash e : \tau\ var, \qquad \lambda; \gamma \vdash e' : \tau}{\lambda; \gamma \vdash e := e' : \tau\ cmd}$$

Cette règle indique comment l'assignation devrait être typée de façon sûre sous certaines conditions exprimées sous la forme de prémisses. Il s'agit de contrôler le flot explicite de e' vers e. En gros, cette règle indique que pour

que la commande soit bien typée, il faut que e et e' soient de même type. Toutefois, la règle de sous-typage SUBTYPE permet aussi le flot d'information dans le cas où e est de type supérieur à celui de e'.

$$\text{(SUBTYPE)} \quad \frac{\lambda; \gamma \vdash p : \rho, \quad \rho \subseteq \rho'}{\lambda; \gamma \vdash p : \rho'}$$

Considérons maintenant la règle de typage du branchement conditionnel, afin d'illustrer le contrôle du flot implicite.

$$\text{(IF)} \quad \frac{\lambda; \gamma \vdash e : \tau, \quad \lambda; \gamma \vdash c : \tau\ cmd, \quad \lambda; \gamma \vdash c' : \tau\ cmd}{\lambda; \gamma \vdash \ \textbf{if}\ e\ \textbf{then}\ c\ \textbf{else}\ c' : \tau\ cmd}$$

Cette règle requiert que la sentinelle e soit de même type que c et c'. Par le sous-typage, on permet à la sentinelle d'avoir un type différent des commandes, tout en s'assurant qu'il ne s'agit pas d'un flot implicite vers le haut. En effet, par la règle de sous-typage ci-haut, le type de e peut être contraint à un type de sécurité élevé, ou le type des commandes contraint à un type bas grâce à la règle CMD suivante.

$$\text{(CMD)} \quad \frac{\tau \subseteq \tau'}{\tau'\ cmd \subseteq \tau\ cmd}$$

Ainsi, indirectement, le seul flot implicite permis est celui d'un type à un autre qui lui est inférieur. Considérons l'exemple suivant tiré de [37] :

if $x = 1$ **then** $y := 0$ **else** $y := 1$

Supposons que x ne prend que les valeurs 0 ou 1 et qu'elle est de type H. Ici, la valeur finale de y peut laisser deviner la valeur de x, suivant la branche qui a été prise. Pour détecter ce flot implicite de x vers y, la règle IF est appliquée. Celle-ci permet à l'instruction de s'exécuter seulement si y est de type H, ainsi c et c' seront de type $H\ cmd$. Si y était de type L, on aurait pu essayer d'utiliser la règle CMD, mais comme $L\ cmd \nsubseteq H\ cmd$, ça ne marche pas. De même, appliquer la règle SUBTYPE sur x ne marche pas car $H \nsubseteq L$.

Une preuve de cohérence de type est faite ; cette cohérence est exprimée comme une propriété de non-interférence, définie précédemment. En effet, cette propriété dit que si un programme est bien typé, alors les variables d'un type donné n'interfèrent pas avec les variables de type inférieur.

1.4.3 Contrôle de flot d'information par typage sensible au flot

Nous terminons cette section par un aperçu de l'approche de Hunt et Sands [18] ; il s'agit d'un contrôle de flot d'information par typage sensible au flot, ce qui veut dire que le type de sécurité assigné aux variables peut changer tout le long de l'analyse du programme. Cette approche est, en ce sens, semblable à la nôtre.

Les auteurs ont travaillé sur un langage de programmation impératif simple, semblable à celui de Volpano *et al.* dans [37], présenté précédemment. La table 1.2 illustre le système de types. Comme précédemment, nous considérons un ensemble de deux niveaux de sécurité H (*high*) et L (*low*). Ils forment un treillis ordonné par la relation $\sqsubseteq$, selon l'ordre $L \sqsubseteq H$. Le supremum $\sqcup$ sur les types de sécurité est défini en respect à cet ordre. La relation $\sqsubseteq$ est appliquée aux environnements de typage comme suit :

$$\Gamma_1 \sqsubseteq \Gamma_2 \; ssi \; \forall x \in dom(\Gamma_1) \cap dom(\Gamma_2) : \Gamma_1(x) \sqsubseteq \Gamma_2(x)$$

Les jugements de typage des commandes sont de la forme $pc \vdash \Gamma \; \{C\} \; \Gamma'$ où Γ et Γ' sont des environnements de typage, et pc est le niveau de sécurité du contexte dans lequel s'exécute la commande C (dans le but de contrôler les flots implicites). La présence de deux environnements de typage indique la sensibilité au flot, car Γ associe des types aux variables avant l'exécution de C, tandis que Γ' leur associe de nouveaux types, le cas échéant, après l'exécution de C. Le type d'une expression E est exprimée comme suit : $\Gamma \vdash E : t$, ce qui signifie que l'expression E est de type t sous Γ.

La règle de typage de l'assignation illustre bien la sensibilité au flot. Cette règle indique que le type de x change. Il reçoit le supremum du type de l'expression et du contexte. Par contre, selon l'approche de Volpano *et al.* qui est insensible au flot, toutes les variables ont un type assigné d'avance. Ainsi, pour permettre l'exécution de cette commande, il suffit de vérifier si les variables dans l'expression E ont un type inférieur ou égal au type de x ; son type ne change pas. Remarquez que dans la règle If, le typage des commandes est effectué sous le contexte $p \sqcup t$, ce qui indique si l'expression est de type H, les commandes doivent être évaluées sous un contexte élevé, pour éviter les flux implicites illégaux.

$$\text{Skip}\ \frac{}{p \vdash \Gamma\ \{\mathbf{skip}\}\ \Gamma}$$

$$\text{Assign}\ \frac{\Gamma\ \vdash E : t}{p \vdash \Gamma\ \{x := E\}\ \Gamma[x \mapsto p \sqcup t]}$$

$$\text{Seq}\ \frac{p \vdash \Gamma\ \{C_1\}\ \Gamma' \qquad p \vdash \Gamma'\ \{C_2\}\ \Gamma''}{p \vdash \Gamma\ \{C_1; C_2\}\ \Gamma''}$$

$$\text{If}\ \frac{\Gamma\ \vdash E : t \qquad p \sqcup t \vdash \Gamma\ \{C_i\}\ \Gamma' \qquad i = 1, 2}{p \vdash \Gamma\ \{\mathbf{if}\ E\ C_1\ C_2\}\ \Gamma'}$$

$$\text{While}\ \frac{\Gamma \vdash E : t \qquad p \sqcup t \vdash \Gamma\ \{C\}\ \Gamma}{p \vdash \Gamma\ \{\mathbf{while}\ E\ C\}\ \Gamma}$$

$$\text{Sub}\ \frac{p_1 \vdash \Gamma_1\ \{C\}\ \Gamma_1'}{p_2 \vdash \Gamma_2\ \{C\}\ \Gamma_2'} \qquad p_2 \sqsubseteq p_1, \Gamma_2 \sqsubseteq \Gamma_1, \Gamma_1' \sqsubseteq \Gamma_2'$$

TABLE 1.2 – Règles de typage sensible au flot

1.5 Conclusion

Le but de ce chapitre était de présenter les travaux existants sur les notions intervenant dans notre approche : les mécanismes d'application de politique de sécurité, l'analyse trivaluée de programmes et l'analyse sûre du flot d'information. Le prochain chapitre présente notre approche : contrôler le flot d'information par l'analyse de types trivaluée.

Chapitre 2

Analyse sûre du flot d'information par typage trivalué

2.1 Introduction

L'analyse décrite à la section 1.4 se base sur la classification statique des données en deux niveaux de sécurité : élevé (H) ou bas (L). Toutefois, il peut arriver que l'on soit confronté à des données dépendant d'une information obtenue dynamiquement. En effet, les programmes interagissent avec un environnement externe qui ne peut être prédit entièrement à la compilation. Or, dans ces cas d'incertitude, les analyses classiques bivaluées font des choix pessimistes et par conséquent génèrent un bon nombre de faux positifs. Pour réduire ces derniers, nous proposons, dans le présent chapitre, le contrôle du flot d'information par une analyse par typage trivaluée sur un langage impératif. Le rôle de l'analyse de types trivaluée est de permettre de tenir compte de la possibilité de ne pas être capable d'attribuer un niveau de sécurité à une donnée.

Trois réponses peuvent résulter de l'analyse : le programme respecte la non-interférence, le programme ne respecte pas la non-interférence ou il est impossible de conclure statiquement si le programme respecte ou pas la non-interférence. Ce troisième cas survient lorsque certaines données ne sont fournies qu'à l'exécution du programme. L'idée clé de notre approche est donc de distinguer les réponses positives des réponses incertaines. Pour cela,

nous introduisons un troisième niveau de sécurité « incertain » (U pour *unknown*), indiquant l'incertitude quant à la sécurité de l'information. Les points d'incertitude du programme sont alors bien mis en évidence. Les travaux futurs consisteront à compléter l'analyse par une instrumentation de code, visant à insérer des tests dynamiques à ces points incertains du programme afin de résoudre l'incertitude.

Notons que pour les résultats certains de la vérification, « oui, le programme est non-interférent » et « non, le programme n'est pas non-interférent », l'analyse par typage suffit, sans avoir recours à l'instrumentation. Le programme est accepté dans le cas de non-interférence, il est rejeté sinon. Ceci montre que notre approche ne rajoutera pas de temps d'exécution supplémentaire dans ces cas.

Nous visons le contrôle de deux types de flots, comme expliqués à la section 1.4.1 : les *flots explicites* qui ont lieu quand le contenu d'une variable est transféré directement dans une autre variable, et les *flots implicites* qui se produisent quand le contenu assigné à une variable dépend d'une autre variable, c'est-à-dire, la sentinelle d'une structure conditionnelle.

Précisons également que notre approche se classe parmi les mécanismes d'application de sécurité sensibles au flot de données (*flow-sensitive*). En effet, nous permettons aux variables du programme de changer de type de sécurité durant l'analyse de types.

Enfin, nous distinguons les canaux de communication des autres variables d'un programme. Par canal de communication, nous entendons tous les objets par lesquels un programme peut acquérir ou fournir des informations à l'extérieur, par exemple les fichiers, le clavier de saisie, les canaux réseau, etc. Leurs types de sécurité sont fixés d'avance, tandis que les variables changent de type selon ce qu'elles contiennent.

2.2 Notre approche

2.2.1 Définition du langage

Nous illustrons notre approche sur un langage impératif simple, qui est une variante de celui présenté par Volpano *et al.* dans [37], adapté pour fonctionner avec les niveaux de sécurité trivalués.

Syntaxe

Soit $\mathcal{V}$ l'ensemble des identificateurs de variables, et $\mathcal{C}$ l'ensemble des canaux de communication. Dans le reste de ce document, nous utilisons de façon générique la notation suivante : les variables sont $x \in \mathcal{V}$, et il existe deux types de constantes : $n \in \mathbb{N}$ et $nch \in \mathcal{C}$. La syntaxe est la suivante :

$$
\begin{array}{llll}
\textit{(instructions)} & p ::= & e \mid c & \\
\textit{(expressions)} & e ::= & x \mid n \mid nch \mid e_1 \ \textbf{op} \ e_2 & \\
\textit{(commands)} & c ::= & \textbf{skip} \mid x := e \mid c_1 ; c_2 & \\
 & & \textbf{if} \ e \ \textbf{then} \ c_1 \ \textbf{else} \ c_2 \ \textbf{end} \mid \textbf{while} \ e \ \textbf{do} \ c \ \textbf{end} \mid & \\
 & & \textbf{receive}_{\textbf{c}} \ x_1 \ \textbf{from} \ x_2 \mid & \\
 & & \textbf{receive}_{\textbf{n}} \ x_1 \ \textbf{from} \ x_2 \mid & \\
 & & \textbf{send} \ x_1 \ \textbf{to} \ x_2 &
\end{array}
$$

Les instructions peuvent être des expressions ou des commandes. Les valeurs peuvent être uniquement des entiers ou des noms de canaux. Les valeurs booléennes sont représentées par 0 pour *faux* et autre que 0 pour *vrai*. Nous désignons par **op** tout opérateur arithmétique ou logique sur les entiers ainsi que les opérateurs de comparaison sur les canaux de communication. Les commandes correspondent, pour la plupart, aux instructions impératives classiques : l'assignation, **skip**, la boucle, les instructions conditionnelles et la composition séquentielle.

Nous supposons que deux programmes ne peuvent communiquer qu'à travers les canaux de communication. Nous supposons qu'un programme a accès à un pointeur indiquant le prochain élément à être lu dans un canal et que l'envoi sur un canal ajoutera l'information de façon à ce qu'elle soit lue dans l'ordre *First-In-First-Out*. Quand une information est lue dans un canal, elle ne disparaît pas, seul le pointeur de lecture est mis à jour. Le contenu observable du canal reste inchangé. Notre langage de programmation est séquentiel. Les processus externes ne peuvent lire ou écrire que dans des canaux publics. Expliquons plus amplement les instructions servant d'accès aux canaux :

- **receive**$_{\textbf{c}}$ x_1 **from** x_2 : se lit « *receive content* ». Cette instruction lit la valeur du canal nommé x_2 et assigne son contenu à la variable x_1.
- **receive**$_{\textbf{n}}$ x_1 **from** x_2 : se lit « *receive name* ». Au lieu de recevoir des données de x_2, x_1 reçoit un autre nom de canal, qui sera éventuellement

utilisé plus loin dans le programme. x_1 doit alors être traitée comme un canal et non une variable entière.

– **send** x_1 **to** x_2 : cette instruction permet d'écrire dans le canal dont le nom est x_2 le contenu de la variable x_1.

La raison pour laquelle nous avons deux commandes *receive* différentes est la conséquence directe de notre choix de distinguer les variables des canaux. Ce sera clarifié lors de l'explication du typage des commandes.

Sémantique opérationnelle

Le comportement du programme est défini par une sémantique opérationnelle structurelle illustrée à la table 2.1. Une instruction p est exécutée sous une mémoire $\mu : \mathcal{V} \to \mathbb{N} \cup \mathcal{C}$. La sémantique spécifie donc comment les *configurations* $\langle p, \mu \rangle$ évoluent, soit en une valeur, soit en une autre configuration, soit en une mémoire. L'évaluation des expressions sous la mémoire n'occasionnent pas d'« effets de bord » qui changent l'état de la mémoire. Au contraire, le rôle des commandes est d'être exécutées et changer l'état. Ainsi, nous avons deux règles d'évaluation :

– $\langle e, \mu \rangle$ conduit à une valeur résultant de l'évaluation de l'expression e sur la mémoire μ ; cette transition est désignée par $\to_e$,

– $\langle c, \mu \rangle$ conduit à une mémoire produite par l'exécution de la commande c sur la mémoire μ ; cette transition est désignée par $\to$.

L'exécution de **skip** garde la mémoire inchangée. L'assignation $x := e$ donne comme résultat la mémoire μ où la valeur de la variable x est mise à jour ; celle-ci contient maintenant l'évaluation de e.

receive$_\mathbf{c}$ x_1 **from** x_2 et **receive**$_\mathbf{n}$ x_1 **from** x_2 s'évaluent presque de la même manière. L'information contenue dans le canal x_2 est lue et assignée à la variable x_1. La particularité est que dans la règle RECEIVE-VAL, la lecture de x_2 résulte en une valeur entière, tandis que pour la règle RECEIVE-NAME la lecture de x_2 donne un nom de canal. Ici, nous introduisons une fonction générique $read(channel)$ représentant l'action d'acquérir de l'information d'un canal (ex. lire une ligne dans un fichier, recevoir la saisie du clavier, etc.). Le contenu du canal reste le même après les deux sortes de **receive**.

send x_1 **to** x_2 met à jour le canal x_2 avec la valeur de la variable x_1. Ceci est fait par la fonction générique nommée $update(canal, information)$, qui représente l'action de mise à jour du canal avec une information donnée.

$$\text{(VAL)}\ \langle v, \mu\rangle \rightarrow_e v,$$

$$\text{(VAR)}\ \langle x, \mu\rangle \rightarrow_e \mu(x),$$

$$\text{(OP)}\ \frac{\langle e_1, \mu\rangle \rightarrow_e v_1 \qquad \langle e_2, \mu\rangle \rightarrow_e v_2 \qquad v_1\ \mathbf{op}\ v_2 = n}{\langle e_1\ \mathbf{op}\ e_2, \mu\rangle \rightarrow_e n}$$

$$\text{(SKIP)}\ \langle \mathbf{skip}, \mu\rangle \rightarrow \mu$$

$$\text{(ASSIGN)}\ \frac{\langle e, \mu\rangle \rightarrow_e v}{\langle x := e, \mu\rangle \rightarrow \mu[x \mapsto v]}$$

$$\text{(RECEIVE-VAL)}\ \frac{x_2 \in dom(\mu) \qquad read(\mu(x_2)) = n}{\langle \mathbf{receive_c}\ x_1\ \mathbf{from}\ x_2, \mu\rangle \rightarrow \mu[x_1 \mapsto n]}$$

$$\text{(RECEIVE-NAME)}\ \frac{x_2 \in dom(\mu) \qquad read(\mu(x_2)) = nch}{\langle \mathbf{receive_n}\ x_1\ \mathbf{from}\ x_2, \mu\rangle \rightarrow \mu[x_1 \mapsto nch]}$$

$$\text{(SEND)}\ \frac{x_1 \in dom(\mu)}{\langle \mathbf{send}\ x_1\ \mathbf{to}\ x_2, \mu\rangle \rightarrow \mu, update(\mu(x_2), \mu(x_1))}$$

$$\text{(COND)}\ \frac{\langle e, \mu\rangle \rightarrow_e n \qquad n \neq 0 \qquad \langle c_1, \mu\rangle \rightarrow \mu'}{\langle \mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2\ \mathbf{end}, \mu\rangle \rightarrow \mu'}$$

$$\text{(COND)}\ \frac{\langle e, \mu\rangle \rightarrow_e n \qquad n = 0 \qquad \langle c_2, \mu\rangle \rightarrow \mu'}{\langle \mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2\ \mathbf{end}, \mu\rangle \rightarrow \mu'}$$

$$\text{(LOOP)}\ \frac{\langle e, \mu\rangle \rightarrow_e n \qquad n = 0}{\langle \mathbf{while}\ e\ \mathbf{do}\ c\ \mathbf{end}, \mu\rangle \rightarrow \mu}$$

$$\text{(LOOP)}\ \frac{\langle e, \mu\rangle \rightarrow_e n \qquad n \neq 0 \qquad \langle c; \mathbf{while}\ e\ \mathbf{then}\ c\ \mathbf{end}, \mu\rangle \rightarrow \mu'}{\langle \mathbf{while}\ e\ \mathbf{do}\ c\ \mathbf{end}, \mu\rangle \rightarrow \mu'}$$

$$\text{(SEQUENCE)}\ \frac{\langle c_1, \mu\rangle \rightarrow \mu' \qquad \langle c_2, \mu'\rangle \rightarrow \mu''}{\langle c_1; c_2, \mu\rangle \rightarrow \mu''}$$

TABLE 2.1 – Sémantique opérationnelle structurelle

Notez que le contenu de la variable x_2, c'est-à-dire le nom du canal, ne change pas ; par conséquent, la mémoire μ reste inchangée. Le contenu du canal est mis à jour après un **send**

Une instruction conditionnelle **if** e **then** c_1 **else** c_2 **end** s'évalue à c_1 ou c_2, dépendant du fait que e s'évalue à $vrai$ (autre que zéro) ou $faux$ (zéro), respectivement. Notez que l'instruction **if** e **then** c **end** est en fait **if** e **then** c **else skip end**.

Pour l'évaluation des boucles, si la condition booléenne s'évalue à *faux*, on n'entre pas dans la boucle et la mémoire reste inchangée ; si la condition s'évalue à *vrai*, le corps de la boucle c est exécuté suivi séquentiellement de la re-exécution de l'instruction **while**.

Si l'évaluation de c_1 transforme la mémoire μ en μ' alors le résultat de $c_1; c_2$ est donné par l'évaluation de c_2 sur la nouvelle mémoire μ'.

2.2.2 Système de types de sécurité

Nous présentons maintenant le système de types de sécurité qui sert à vérifier si un programme, écrit dans le langage décrit ci-haut, satisfait la non-interférence, la satisfait peut-être ou ne la satisfait pas du tout. Les types de sécurité sont définis comme suit :

$$\begin{array}{ll} \textit{(Types des données)} & \tau ::= \ L \mid H \mid U \\ \textit{(Types des instructions)} & \rho ::= \tau \ \textit{val} \mid \tau \ \textit{chan} \mid \tau \ \textit{cmd} \end{array}$$

Nous considérons un ensemble de trois niveaux de sécurité $SL = \{L, U, H\}$. Cet ensemble est étendu à un treillis $(SL, \sqsubseteq)$ selon l'ordre suivant : $L \sqsubseteq U \sqsubseteq H$ (nous utilisons les symboles $\sqsupseteq$ and $\sqsupset$ comme d'habitude). C'est en respect à cet ordre que le supremum $\sqcup$ et l'infimum $\sqcap$ sur les types de sécurité sont définis. Nous élevons cet ordre aux types des instructions et l'associons de manière classique, tout en supposant que les opérations $\sqcup$ et $\sqcap$ renvoient $\bot$ quand l'ordre est appliqué sur des instructions de différents types. Par exemple, $H \ \textit{chan} \sqcup H \ \textit{val} = \bot$. Nous avons aussi besoin d'une relation plus faible, $\preceq$, définie comme suit :

$$x \not\preceq y \ \text{ ssi } \ x = H \text{ et } y = L.$$

Cette relation peut être interprétée comme « peut-être $\sqsubseteq$ » : elle est utilisée pour assurer que le rejet d'un programme ne se fera que s'il existe un flot de H vers L ; ceci sera expliqué plus amplement dans la suite.

Une étiquette est ajoutée au type d'une instruction dans le but d'indiquer si une information est une valeur entière, un nom de canal ou une commande. En typant un programme, les types de sécurité sont assignés aux variables, canaux et commandes – et au contexte d'exécution. La signification des types est la suivante. Une variable de type $\tau \ \textit{val}$ a un contenu de type de sécurité τ ; un canal de type $\tau \ \textit{chan}$ peut contenir de l'information de type

τ ou inférieur (en effet, un canal privé doit pouvoir contenir ou recevoir à la fois de l'information privée et publique). Le typage des commandes est standard mais a une signification légèrement différente : une commande de type τ *cmd* est garantie de contenir seulement des flots vers des canaux dont les types de sécurité sont τ ou supérieur. Ainsi, s'il s'agit d'une commande de type L *cmd* alors il peut contenir un flot vers un canal de type L *chan*.

Notre système de types possède deux propriétés intéressantes : la *simple sécurité* qui s'applique aux expressions et le *confinement* qui s'applique aux commandes[34]. La *simple sécurité* dit qu'une expression e de type τ *val* ou τ *chan* contient uniquement des variables de niveau de sécurité τ ou inférieur. La simple sécurité assure que le type de la variable est cohérent avec le principe énoncé dans le précédent paragraphe. Quant au *confinement*, il dit qu'une commande c ayant le type τ *cmd* exécutée sous un contexte de type pc permet seulement des flots vers des canaux de type $\tau \sqcup pc$ ou supérieur, ceci dans le but d'éviter un flot d'information d'un canal donné vers un autre de type de sécurité inférieur (H vers L par exemple). Ces deux propriétés sont utilisées pour prouver la non-interférence.

Les règles de typage sont illustrées à la table 2.2. Un *jugement de typage* a la forme $\Gamma, pc \vdash p : \rho, \Gamma'$, où Γ et Γ' sont des environnements de typage, associant à une variable son type de sécurité τ *val* ou τ *chan*, indiquant son niveau de sécurité ; pc est le type de sécurité du contexte. Un programme est typé dans un contexte de type L ; selon les types de sécurité des conditions, certains blocs d'instruction sont typés d'un contexte de niveau supérieur, comme cela sera expliqué plus tard. Le jugement de typage peut être lu de cette manière : Étant donné l'environnement de typage initial Γ et un contexte de typage pc, la commande p a le type ρ produisant, le cas échéant, un nouvel environnement Γ'. Notez que la présence d'un environnement de typage final résultant de l'exécution d'une commande est motivé par le fait que notre analyse de types est sensible au flot. Ainsi, Γ' indique un nouvel environnement prenant en compte les modifications de types apportées par le typage de la commande. Puisque les expressions n'affectent pas l'état de la mémoire, l'environnement de typage reste le même. Dans ce cas, Γ' est omis. Puisque le type des canaux est constant, il existe un environnement de typage particulier pour les constantes de canal, nommé $TypeOf\_Channel$ qui est donné avant l'analyse. Dans les règles de typage, α représente l'étiquette *val* ou *chan*, selon le contexte.

Il existe trois opérateurs sur les environnements de typage que nous devons définir : $\Gamma\dagger[x \mapsto \rho]$, $\Gamma \sqcup \Gamma'$ et $\overline{\Gamma}$. Le premier est une mise à jour standard de l'environnement, où l'image de x devient ρ, peu importe si x était dans le domaine original ou pas. Pour la règle de la structure conditionnelle, nous devons faire une union d'environnements où leurs variables communes reçoivent comme type de sécurité le supremum de leurs types, et les variables de canaux reçoivent le type U si les types sont différents dans les deux environnements. Plus précisément, nous étendons $\sqcup$ aux environnements comme suit : $\mathrm{dom}(\Gamma \sqcup \Gamma') = \mathrm{dom}(\Gamma) \cup \mathrm{dom}(\Gamma')$, et

$$\Gamma \sqcup \Gamma'(x) = \begin{cases} \Gamma(x) & \text{si } x \in \mathrm{dom}(\Gamma) \backslash \mathrm{dom}(\Gamma') \\ \Gamma'(x) & \text{si } x \in \mathrm{dom}(\Gamma') \backslash \mathrm{dom}(\Gamma) \\ U & \text{si } \Gamma(x) = \tau \ \mathit{chan} \neq \tau' \ \mathit{chan} = \Gamma'(x) \\ \Gamma(x) \sqcup \Gamma'(x) & \text{sinon.} \end{cases}$$

Notez que, comme pour l'application de $\sqcup$ aux types des valeurs et ceux des instructions, $(\Gamma \sqcup \Gamma')(x)$ peut retourner la variable d'incompatibilité de type $\bot$ si Γ et Γ' sont incompatibles sur la variable x, par exemple si $\Gamma(x)$ est une valeur et $\Gamma'(x)$ est un canal (cela ne peut arriver que si Γ et Γ' proviennent de différents branchements d'une commande **if**).

Le dernier opérateur sur les environnements de typage à définir est lié à l'instrumentation. Nous introduisons une variable spéciale *_instr* dont le type (maintenu dans l'environnement de typage) indique si le programme a besoin d'être instrumenté ou pas. Si l'image de *_instr* est U ou H c'est qu'il faut instrumenter, L sinon. Initialement, $\Gamma(\mathit{\_instr}) = L$. Sa valeur est mise à U ou H par l'opérateur supremum dans la règle RECEIVE-NAME_S et par l'opérateur désigné par « $\overline{\ }$ », dans la règle SEND_S. La définition de $\overline{\Gamma}$ dans SEND_S est la suivante :

$\overline{\Gamma} = \Gamma \dagger [\mathit{\_instr} \mapsto U]$ si $(\tau_1 \sqcup pc) = \tau = U$ ou $(\tau_1 \sqcup pc) \not\sqcup \tau$
$\overline{\Gamma} = \Gamma$ sinon.

Avant d'expliquer les règles en détails, voyons plus en profondeur la relation $\preceq$, que l'on retrouve dans la règle SEND_S par exemple. Comme vu précédemment, cette relation est utilisée pour assurer que le rejet d'un programme n'arrivera que dans le cas où il existe un flot de H vers L. Au cours de l'analyse d'un programme, nous distinguons les flots sûrs des flots incertains. Par exemple, les flots de U vers H ou de L vers U sont sûrs car peu importe les types qu'auront les variables incertaines à l'exécution (L ou H),

les flots seront toujours sûrs. Toutefois, selon le véritable type de la variable typée U à l'exécution, un flot comme U vers L ou U vers U peut être sûr ou pas. Une analyse conservatrice aurait rejeté un programme avec de tels flots mais la nôtre marquera le programme comme nécessitant d'être instrumenté et cela sera préservé tout le long de l'analyse de types.

Voyons la règle de typage de l'instruction **send** : l'envoi de x_1 sur le canal x_2 est accepté par le système de types si $(\tau_1 \sqcup pc) \preceq \tau$, où τ_1 (resp. τ) est le type de x_1 (resp. x). Ceci implique, par définition de $\preceq$, que si, par exemple, x_1 a un contenu H – ou si le contexte de typage est H – tandis que x_2 est un canal L, alors la règle ne peut être appliquée, et par conséquent, le programme est rejeté. Voyons ensuite ce qui arrive dans les autres cas. Supposons que $(\tau_1 \sqcup pc) \preceq \tau$ et $(\tau_1 \sqcup pc) \sqsubseteq \tau$ mais $\neg((\tau_1 \sqcup pc) = \tau = U)$; dans ce cas, l'image de *_instr* dans Γ reste inchangée. Ceci signifie que le flot de x_1 vers x_2 est sûr sans aucun doute. Cependant, dans tous les autres cas, c'est-à-dire H vers U, U vers L et U vers U, il y a un risque de fuite d'information, ainsi $\overline{\Gamma}(\_instr) = U$, indiquant le besoin d'instrumentation. La raison pour laquelle nous utilisons une relation particulière $\preceq$ au lieu de $\sqsubseteq$ est de permettre au typage de continuer jusqu'à la fin en cas d'incertitude. Si nous avions utilisé la relation d'ordre habituelle $\sqsubseteq$, les flots tels que H vers U et U vers L auraient été rejetés, alors qu'il y a possibilité que la variable de type U soit une variable sûre. Gérer cette incertitude est la principale contribution de notre approche et permet de réduire le nombre de faux positifs.

Dans des travaux connexes, le typage fait appel à des *jugements de sous-typage* de la forme $\rho_1 \subseteq \rho_2$ ou $\rho_1 \leqslant \rho_2$ [34, 37]. Prenons le cas de deux types de sécurité τ et τ' tels que $\tau \leqslant \tau'$; alors toute information de type τ peut être traitée comme donnée de type τ'. De même, si une commande assigne des valeurs uniquement à des variables de niveau H ou supérieur alors, *a fortiori*, il en assigne à des variables de type L et supérieur. Par conséquent, nous aurons une règle contravariante de sous-typage pour exprimer $H\ cmd \subseteq L\ cmd$. Dans notre travail, nous n'utilisons pas de sous-typage. Les contraintes sont ajoutées directement dans les règles de typage. Au lieu d'utiliser des coercions de type grâce au sous-typage, une instruction reçoit un type fixe, qui est le type le plus « sûr ». Par exemple, d'après nos règles de typage, pour deux expressions e_1 et e_2 de type τ et τ' respectivement, e_1 **op** e_2 est typé par $\tau_1 \sqcup \tau_2$. Pour deux commandes c et c' typées τ et τ' res-

pectivement, la composition à travers le séquencement ou les conditionnelles est de type $\tau \sqcap \tau'$.

$$\text{(CHAN\_S)}\ \frac{\mathit{TypeOf\_Channel}(nch) = \tau}{\Gamma, pc \vdash nch : \tau\ \mathit{chan}} \qquad \text{(INT\_S)}\ \Gamma, pc \vdash n : L\ \mathit{val}$$

$$\text{(OP\_S)}\ \frac{\Gamma, pc \vdash e_1 : \tau_1\,\alpha, \qquad \Gamma, pc \vdash e_2 : \tau_2\,\alpha}{\Gamma, pc \vdash e_1\ \mathbf{op}\ e_2 : (\tau_1 \sqcup \tau_2)\ \mathit{val}} \qquad \text{(VAR\_S)}\ \frac{\Gamma(x) = \tau\,\alpha}{\Gamma, pc \vdash x : \tau\,\alpha}$$

$$\text{(SKIP\_S)}\ \Gamma, pc \vdash\ \mathbf{skip}\ :\ H\ \mathit{cmd}$$

$$\text{(ASSIGN-VAL\_S)}\ \frac{\Gamma, pc \vdash e : \tau\ \mathit{val}}{\Gamma, pc \vdash x := e : (\tau \sqcup pc)\ \mathit{cmd}, \Gamma\dagger[x \mapsto (\tau \sqcup pc)\ \mathit{val}]}$$

$$\text{(ASSIGN-CHAN\_S)}\ \frac{\Gamma, pc \vdash e : \tau\ \mathit{chan} \qquad pc \preceq \tau}{\Gamma, pc \vdash x := e : \tau\ \mathit{cmd}, \Gamma_{assign}}$$

$$\text{(RECEIVE-VAL\_S)}\ \frac{\Gamma(x_2) = \tau\ \mathit{chan}}{\Gamma, pc \vdash \mathbf{receive_c}\ x_1\ \mathbf{from}\ x_2 : (\tau \sqcup pc)\ \mathit{cmd}, \Gamma_{receive-v}}$$

$$\text{(RECEIVE-NAME\_S)}\ \frac{\Gamma(x_2) = \tau\ \mathit{chan} \qquad pc \preceq \tau}{\Gamma, pc \vdash \mathbf{receive_n}\, x_1\, \mathbf{from}\, x_2 : \tau\ \mathit{cmd}, \Gamma_{receive-n}}$$

$$\text{(SEND\_S)}\ \frac{\Gamma(x_1) = \tau_1\,\alpha, \quad \Gamma(x_2) = \tau\ \mathit{chan}, \qquad (\tau_1 \sqcup pc) \preceq \tau}{\Gamma, pc \vdash \mathbf{send}\ x_1\ \mathbf{to}\ x_2 : \tau\ \mathit{cmd}, \overline{\Gamma}}$$

$$\text{(COND\_S)}\ \frac{\Gamma, pc \vdash e : \tau_0\ \mathit{val} \qquad \Gamma, (pc \sqcup \tau_0) \vdash c_1 : \tau_1\ \mathit{cmd}, \Gamma' \qquad \Gamma, (pc \sqcup \tau_0) \vdash c_2 : \tau_2\ \mathit{cmd}, \Gamma'' \qquad \Gamma' \sqcup \Gamma'' \sqsupset \bot}{\Gamma, pc \vdash\ \mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2\ \mathbf{end} : (\tau_1 \sqcap \tau_2)\ \mathit{cmd}, \Gamma' \sqcup \Gamma''}$$

$$\text{(LOOP1\_S)}\ \frac{\Gamma, pc \vdash e : \tau_0\ \mathit{val} \qquad \Gamma, (pc \sqcup \tau_0) \vdash c : \tau\ \mathit{cmd}, \Gamma' \qquad \Gamma = \Gamma \sqcup \Gamma' \sqsupset \bot}{\Gamma, pc \vdash \mathbf{while}\ e\ \mathbf{do}\ c\ \mathbf{end} : \tau\ \mathit{cmd}\ \Gamma \sqcup \Gamma'}$$

$$\text{(LOOP2\_S)}\ \frac{\Gamma, pc \vdash e : \tau_0\ \mathit{val} \quad \Gamma, (pc \sqcup \tau_0) \vdash c : \tau\ \mathit{cmd}, \Gamma' \quad \Gamma \neq \Gamma \sqcup \Gamma' \sqsupset \bot \quad \Gamma \sqcup \Gamma', (pc \sqcup \tau_0) \vdash \mathbf{while}\ e\ \mathbf{do}\ c\ \mathbf{end} : \tau'\ \mathit{cmd}, \Gamma''}{\Gamma, pc \vdash \mathbf{while}\ e\ \mathbf{do}\ c\ \mathbf{end} : \tau'\ \mathit{cmd},\ \Gamma''}$$

$$\text{(SEQUENCE\_S)}\ \frac{\Gamma, pc \vdash c_1 : \tau_1\ \mathit{cmd}, \Gamma' \qquad \Gamma', pc \vdash c_2 : \tau_2\ \mathit{cmd}, \Gamma''}{\Gamma, pc \vdash c_1; c_2 : (\tau_1 \sqcap \tau_2)\ \mathit{cmd}, \Gamma''}$$

Avec $\Gamma_{assign} = (\Gamma \sqcup [\_instr \mapsto \tau \sqcup pc]) \dagger [x \mapsto \tau\ \mathit{chan}]$
$\Gamma_{receive-v} = \Gamma \dagger [x_1 \mapsto (\tau \sqcup pc)\ \mathit{val}]$
$\Gamma_{receive-n} = (\Gamma \sqcup [\_instr \mapsto \tau \sqcup pc]) \dagger [x_1 \mapsto U\ \mathit{chan}]$

TABLE 2.2 – Règles de typage

Passons alors à l'explication des règles de typage illustrées à la table 2.2. CHAN_S assigne des types aux constantes de canal, tandis que VAR_S assigne des types aux variables. Nous supposons que tous les entiers ont le type de sécurité L et **skip** a le type H *cmd.*, le plus petit type de sécurité des

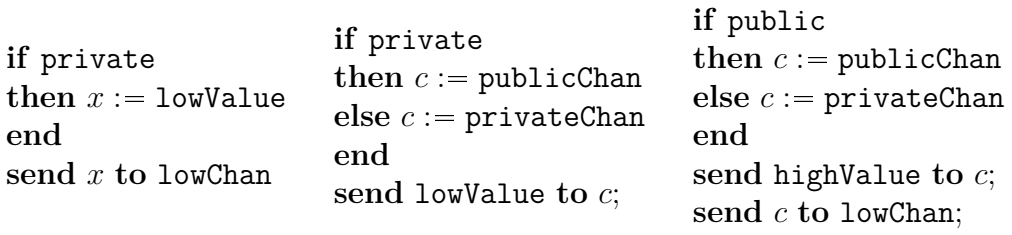

FIGURE 2.1 – Traitement des variables de valeur et de canal dans différentes branches d'un **if**

expressions et des commandes, respectivement. OP_S assigne au résultat de l'opération la plus petite borne supérieure des types des opérandes.

ASSIGN-VAL_S et RECEIVE-VAL_S mettent à jour l'environnement en associant à la variable modifiée le type $(\tau \sqcup pc)$ *val*. De cette façon, si nous sommes dans un bloc d'instructions devant être privé (c'est-à-dire $pc = H$), alors la variable modifiée aura le type H. Ceci permet de prévenir les flots implicites dans les instructions **while** et **if** quand la condition est de type H.

ASSIGN-CHAN_S et RECEIVE-NAME_S modifient tous une variable de canal ; dans le premier cas, un type a déjà été assigné au canal, et dans le dernier, le canal n'est pas connu et ainsi la variable de canal reçoit le type U. Dans les deux cas, le typage est fait sous la condition $pc \preceq \tau$ et cause l'instrumentation si $pc \sqcup \tau \sqsupseteq U$. En effet, si la source (e dans un cas, x_2 dans l'autre) est de type U ou H, l'instrumentation doit être faite afin d'insérer un test qui vérifiera si le véritable type de x_1 est L : si c'est le cas, aucun envoi ne sera permis sur ce canal, pour éviter un flot vers un type de sécurité inférieur (et ainsi le canal devrait être marqué comme non sûr durant l'analyse dynamique). En résumé, il y a trois cas : si $pc \not\preceq \tau$, c'est-à-dire, $pc = H$ et $\tau = L$, le programme est rejeté ; sinon, le programme est accepté et instrumenté sauf si $pc = L = \tau$. Le cas de l'assignation est illustré par le second programme de la figure 2.1 (qui utilise une règle **if** qui sera expliquée dans la suite). Dans la dernière ligne, une information de contenu public est envoyée dans c, ce qui ne peut être permis car de l'information sur la condition `private` serait révélée : l'envoi ne peut être bloqué à ce point ; le signalement doit être donné plus tôt, et cela quand c est rendu public alors que le contexte est privé (à cause de la condition). Nous choisissons de rejeter une telle faille, qui survient exactement quand $pc \not\preceq \tau$ (voir la prochaine section pour un aperçu de comment l'instrumentation gère le cas

$pc = U$).

Pour la règle RECEIVE-VAL_S, x_1 reçoit le supremum du type du canal x_2 et pc, comme il reçoit le contenu de x_2 sous le type du contexte pc. Notez que RECEIVE-VAL_S donne à la variable x_1 un type $(\tau \sqcup pc)$ *val* tandis que RECEIVE-NAME_S assigne un type U *chan*. Ceci montre clairement la distinction des deux commandes $\mathbf{receive_c}$ et $\mathbf{receive_n}$: la première lit une valeur sur le canal tandis que la dernière lit un nom de canal.

Pour les raisons expliquées plus haut, la règle SEND_S exige que le canal x_2 sur lequel x_1 est envoyé soit de type « peut-être supérieur » (ou égal) à celui de x_1 et du contexte pc. Ainsi, x_2 doit satisfaire $(\tau_1 \sqcup pc) \preceq \tau$; cette règle assigne à l'instruction **send** le type du canal x_2.

La règle COND_S exige que les branchements c_1 et c_2 soient typés sous le contexte $pc \sqcup \tau_0$, ceci dans le but d'éviter les flots implicites de la condition vers les branches. Un exemple typique de cette situation est le premier programme de la figure 2.1. Comme la variable x est typée sous un contexte de sécurité élevée (à cause de la sentinelle `private`), elle obtiendra le type H, et ainsi le programme sera rejeté, car x ne peut être envoyé sur un canal public. L'environnement de typage produit, comme l'indique la règle COND_S est calculée avec l'opérateur $\sqcup$. Expliquons maintenant pourquoi $\sqcup$ est défini différemment pour les variables de canal et variables de valeur. Si Γ et Γ', les environnements associés aux deux branches du **if**, diffèrent sur une variable de valeur, nous préférons être pessimistes, et assignons le supremum des deux types de sécurité. Un utilisateur voulant réduire les faux positifs pourrait assigner le type U à cette variable, et laisser la décision finale à l'analyse dynamique. Dans le cas des variables de canal, nous n'avons pas le choix car le nom de canal peut être privé s'il provient d'une source privée mais son contenu peut être public (et vice versa), comme illustré par le dernier programme de la figure 2.1. La dernière ligne doit faire rejeter le programme parce que le branchement **else** fait de c une information privée. Par conséquent, on pourrait conclure que le bon typage de c lors du typage de la commande **if** est H mais l'avant-dernière ligne **send** `highValue` **to** c; aurait requis que c soit typé L *chan* pour que le programme soit rejeté. Nous devons donc donner à c le type U, d'où la définition de $\sqcup$.

SEQUENCE_S donne à la commande la plus grande borne inférieure des types des instructions séquentielles. Remarquez que c_2 est typée à partir de l'environnement mis à jour par la commande c_1.

Notre analyse de types est sensible au flot, nous devons donc analyser le **while** de manière itérative. Pour illustrer ce fait, considérons le programme de gauche dans la figure suivante :

		Eval. & mises à jour à la 1ère iteration	...à la 2e	...à la 3e
1.	**receive**$_{\mathbf{c}}$ h **from** `private`;	$h \mapsto H$		
2.	$e, x_1, x_2, x_3 := 0$;	$e, x_1, x_2, x_3 \mapsto L\ val$		
3.	**while** $e < 5$ **do**	$e < 5 : L\ val$		
4.	**send** x_3 **to** `public`;	$[L \sqcup L \preceq L,$ $pc \sqcup L = L]$		
		$\_instr \mapsto L$	ok	ok
5.	$x_3 := x_2$;	$x_3 \mapsto L\ val$	-	$x_3 \mapsto H\ val$
6.	$x_2 := x_1$;	$x_2 \mapsto L\ val$	$x_2 \mapsto H\ val$	-
7.	$x_1 := h$;	$x_1 \mapsto H\ val$	-	-
8.	$e := e + 1$	-	-	-
9.	**end**			

FIGURE 2.2 – La commande **while** a besoin d'une analyse itérative

Dans cet exemple, c'est seulement à la quatrième itération que nous détectons une faille de sécurité, quand x_3 entre finalement dans le corps du **while** avec un contenu privé et est envoyé sur un canal public. Le typage correspondant est illustré de façon informelle dans le côté droit de la figure. Le besoin d'une analyse de types itérative est traité par deux règles. La première, LOOP_S, est la condition d'arrêt : le typage de la commande c sous Γ n'augmente le type d'aucune variable, et ainsi, peu importe le nombre de répétitions de c, les types des variables ne sont pas augmentés. Par contre, dans LOOP2_S, si l'environnement est modifié, nous itérons en utilisant le supremum de l'environnement produit et l'original. Nous affirmons que ce processus termine parce qu'il y a un nombre fini de variables dans c, un nombre fini de types (donc un nombre fini d'augmentations non-triviales du type de toute entité du programme) et l'opération $\sqcup$ entre l'environnement de typage original et celui produit est monotone. En effet, les types des variables sont modifiées de façon monotone par rapport à $\sqsubseteq$ pour les variables de valeur, et par rapport à l'ordre suivant pour les variables de canal : $L \rightarrow H \rightarrow U$.

2.2.3 Instrumentation

Nous décrivons sommairement la suite de nos travaux portant sur l'instrumentation. Considérons l'exemple de la figure suivante illustrant le typage d'un programme nécessitant une instrumentation.

	Évaluations & Mises à jour
1. **receive**$_{\mathbf{c}}$ h **from** `private` ;	$h \mapsto H\ \ val$
2. **if** *public* **then**	$[pc \sqcup \tau_0 = L \Rightarrow$ le contexte des branches est $L]$
3. **receive**$_{\mathbf{n}}$ x **from** `private` ;	$[pc = L],\ x \mapsto U\ \ chan,\ \_instr \mapsto U$
4. **send** h **to** x	$[pc = L,\ H \sqcup pc \preceq U],\ \_instr \mapsto U$
5. **end**	

FIGURE 2.3 – Un programme générant des valeurs de sécurité incertaines et appelant à l'instrumentation

La règle RECEIVE-NAME_S donne à x le type de sécurité incertain $U\ \ chan$ puisqu'un nom de canal est reçu ; de plus, puisque la source est un canal privé, l'instrumentation est appelée (pour bloquer le canal x s'il se révèle être de type public). Un autre appel à l'instrumentation est illustré dans cet exemple, car il y a un flot explicite à l'instruction 4 de la variable h, dont le type est H, vers x, dont le type est inconnu.

Quand l'analyse de types conclura que le programme nécessite une instrumentation, le programme sera modifié de deux façons. Les tests seront insérés avant chaque instruction problématique, pour vérifier si elle peut être exécutée sans danger ; pour faire cela, nous devons mettre à jour, à l'exécution, le type des variables modifiées ainsi que le type du contexte, et ainsi le programme doit être modifié en conséquence. Nous avons évidemment accès à l'environnement *TypeOf_Channel*, défini a priori. Nous conserverons les types des variables dans une fonction finie, *TypeOf_Var*, et le type de sécurité du contexte dans une pile, *Ctxt*. Nous empilerons le type d'une condition dans *Ctxt* au début d'une structure conditionnelle ou répétitive, et nous dépilerons après chaque fin de structure.

Dans notre système de types, il existe trois règles qui appellent à l'instrumentation ; ces « appels » se produisent quand la variable *_instr* reçoit la valeur U ou H. Il s'agit des règles ASSIGN-CHAN_S, RECEIVE-NAME_S et SEND_S. Nous implémenterons un algorithme d'inférence de types de

manière à ce que les instructions soient identifiées de façon unique soit par des étiquettes, soit par leurs numéros de ligne. L'algorithme d'inférence sauvegardera l'identifiant de l'instruction à instrumenter, ainsi que les types, variables, expressions et instructions qu'elle implique. L'étape d'instrumentation insérera un test avant chaque instruction à instrumenter. À titre d'exemple, l'instrumentation du programme de la figure 2.3 résultera au programme de la figure suivante :

```
1.      receive_c h from private;
add_1.  Update(TypeOf_Var, h, (TypeOf_Channel(private)
                                  ⊔ top(Ctxt)) val);
2.      if public then
add_2.  push(L ⊔ top(Ctxt), Ctxt)                ⟨car public : L  val⟩
3.        receive_n x from private;
add_3.    Update(TypeOf_Var, x, TypeOf_Channel(x));
          if TypeOf_Var(x) = L ∧
              (TypeOf_Var(private) ⊔ top(Ctxt) = H)
          then unsecure(x) = true    else unsecure(x) = false;
add_4.  if (TypeOf_Var(h) ⋤ TypeOf_Channel(x)) ∨ unsecure(x)
              then alert
4.      else send h to x end
5.      end;
add_5.  pop(Ctxt)
```

FIGURE 2.4 – L'instrumentation projetée du programme de la figure 2.3

$Update(M, x, v)$ est une fonction qui met à jour l'image de l'entrée $x \mapsto v$ dans M, tandis que *push*, *pop* et *top* sont les fonctions standard sur les piles. Notez que l'instruction **receive$_n$** marque la variable x comme non sûre et cette information est testée quand on arrive à l'instruction **send** de la ligne `add_4`. Par conséquent, si une variable publique avait été reçue sur un canal privé à la ligne `3`, alors l'alerte aurait été envoyée – et une action doit être prise pour prévenir la fuite d'information, arrêter le programme ou ignorer l'envoi par exemple.

Cet aperçu consiste en une première approche. Nous pensons qu'il est possible d'optimiser l'instrumentation en utilisant l'analyse de flot de données. Par exemple, au lieu d'ajouter un test avant une instruction **send**, c'est-à-dire avant que l'information soit envoyée, nous pourrions l'insérer après

l'instruction $\mathbf{receive_n}$, si l'on sait déjà que le canal sera utilisé dans un **send** et qu'il pourrait occasionner un flot illégal. Ceci permettrait de faire le test plus tôt dans l'exécution du programme et donc de gagner en temps d'analyse.

2.2.4 Preuve de cohérence

Cette section présente la preuve de cohérence de notre système de types, c'est-à-dire nous prouvons qu'un programme bien typé selon notre méthode respecte la non-interférence. Notez que nous ne prouvons la non-interférence que pour les programmes typés, sans instrumentation nécessaire. Dans le cas où le programme est typé sous réserve d'être instrumenté, la non-interférence sera garantie lors de l'instrumentation.
La preuve s'inspire de celle donnée dans [37]. Elle se fait en plusieurs étapes :

1. Nous prouvons que le système possède les propriétés énoncées précédemment : la *simple sécurité* s'appliquant aux expressions et le *confinement* s'appliquant aux commandes. Ces propriétés sont nécessaires pour prouver la non-interférence ;
2. Nous définissons formellement la non-interférence en passant préalablement par la définition d'une notion d'équivalence de mémoires, la τ-équivalence ;
3. Nous énonçons et prouvons le théorème de cohérence du système de types.

Rappelons que le terme α remplace l'étiquette *val* ou *chan* selon le contexte.

Simple sécurité et confinement

Lemme 2.2.1. *(Simple sécurité) Soient* Γ *un environnement de typage, pc un type de contexte et e une expression. Si* $\Gamma, pc \vdash e : \tau\,\alpha$*, alors, pour toute variable x dans e, telle que* $\Gamma(x) = \tau'\,\alpha$*, nous avons* $\tau' \sqsubseteq \tau$*.*

PREUVE. Par induction sur la structure de l'expression e.

1. Étape de base : Soit l'expression n ou nch. Puisqu'il n'y a aucune variable dans e, la simple sécurité est respectée de façon triviale. Maintenant supposons que l'expression e soit x. Selon la règle VAR_S, nous avons $\Gamma, pc \vdash x : \tau\,\alpha$; alors $\Gamma(x) = \tau\,\alpha$, donc $\tau \sqsubseteq \tau$. La simple sécurité est encore respectée.

2. Étape d'induction : Supposons que e soit l'opérateur **op**. $\Gamma, pc \vdash e_1$ **op** $e_2 : (\tau_1 \sqcup \tau_2)$ *val*. Nous avons $\forall i = 1, 2$, e_i a le type $\tau_i\, \alpha$; donc par induction, pour tous les x_i dans l'expression e_i tels que $\Gamma(x_i) = \tau_i'$ *val*, nous avons $\tau_i' \sqsubseteq \tau_i \sqsubseteq (\tau_1 \sqcup \tau_2)$. Par conséquent, dépendamment du plus grand type, les variables dans les deux expressions sont de type soit inférieur, soit égal au type du résultat. Ainsi, la simple sécurité est aussi respectée.

Lemme 2.2.2. *(Confinement) Soit* Γ, Γ' *deux environnements de typage, pc un type de contexte et c une commande. Si, après l'exécution de c telle que* $\langle c, \mu\rangle \rightarrow \mu'$, *nous avons* $\Gamma, pc \vdash c : \tau\ cmd, \Gamma'$, *avec* $\Gamma'(\_instr) = L$, *alors, pour toute variable x telle que* $\mu(x) \neq \mu'(x)$, *et* $\Gamma'(x) = \tau'\, \alpha$, *nous avons* $\tau' \sqsupseteq \tau \sqcup pc$.

PREUVE. Par induction sur la structure de la commande c. L'étape de base comprend cinq étapes ; les cas $c =$ **skip** et $c =$ **send** sont triviaux, car μ reste inchangé.

1. Supposons $\Gamma, pc \vdash x := e : \tau\ cmd, \Gamma'$ et $\langle x := e, \mu\rangle \rightarrow \mu'$. Selon la règle d'évaluation de l'assignation, la variable modifiée est x. Si la règle ASSIGN-VAL_S est appliquée, $\tau = \tau_e \sqcup pc$, où τ_e est le type de l'expression e. Celle-ci reçoit le même type, τ *val*, ainsi $\tau \sqsupseteq \tau_e \sqcup pc$. Si la règle ASSIGN-CHAN_S est appliquée, la variable modifiée x reçoit le type τ *chan* et la commande reçoit le type $\tau\ cmd$, avec $pc \preceq \tau$. Nous avons deux cas : soit $pc \sqsubseteq \tau$, donc $\tau \sqsupseteq \tau \sqcup pc$; soit $pc \not\sqsubseteq \tau$ et $\Gamma(\_instr) = H$ ou U, donc le confinement dépendra de l'instrumentation.

2. Supposons $\Gamma, pc \vdash$ **receive$_c$** x_1 **from** $x_2 : \tau\ cmd, \Gamma'$. Selon la règle d'évaluation RECEIVE-VAL, la variable modifiée est x_1. Selon la règle RECEIVE-VAL_S, $\Gamma'(x_1) = \tau$ *chan*, donc le confinement est respecté.

3. Supposons la commande **receive$_n$** telle que $\Gamma \vdash$ **receive$_n$** x_1 **from** $x_2 : \tau\ cmd, \Gamma'$. Selon la règle d'évaluation RECEIVE-NAME, la variable modifiée est x_1. Comme x_1 est une nouvelle variable de nom de canal, nous ne pouvons nous prononcer sur son type de sécurité. Ainsi, nous lui donnons le type U. Nous avons alors deux cas : soit $\tau \sqcup pc = L$, donc $U \sqsupseteq \tau \sqcup pc$; soit $\tau \sqcup pc \neq L$ et $\Gamma(\_instr) = H$ ou U, donc le confinement dépendra de l'instrumentation.

Étape d'induction :

1. Supposons la commande $c_1; c_2$ telle que $\Gamma, pc \vdash c_1; c_2 : \tau\ cmd, \Gamma''$. Selon la règle d'évaluation SEQUENCE, nous avons $\langle c_1, \mu\rangle \rightarrow \mu'$ et $\langle c_2, \mu'\rangle \rightarrow \mu''$. Aussi, nous avons $\forall i = 1, 2\ , \Gamma, pc \vdash c_i : \tau_i\ cmd, \Gamma'$; ainsi, par induction, pour toute variable x modifiée dans μ' telle que $\Gamma'(x) = \tau'\,\alpha$, nous avons $\tau' \sqsupseteq \tau_1 \sqcup pc$. Or, $\tau_1 \sqsupseteq (\tau_1 \sqcap \tau_2) = \tau$. Ensuite, toujours, par induction, pour toute variable x modifiée dans μ'' telle que $\Gamma''(x) = \tau''\,\alpha$, nous avons $\tau'' \sqsupseteq \tau_2 \sqcup pc$. Or, $\tau_2 \sqsupseteq (\tau_1 \sqcap \tau_2) = \tau$. Ainsi, le type des variables modifiées dans μ'' indiquent que le confinement est respecté.
   Supposons la commande **if − else** telle que $\Gamma, pc \vdash$ **if** e **then** c_1 **else** c_2 **end** : $\tau\ cmd, \Gamma_+$, où $\Gamma_+ = \Gamma' \sqcup \Gamma''$. Nous avons deux évaluations possibles : soit on a $\langle c_1, \mu\rangle \rightarrow \mu'$, soit on a $\langle c_2, \mu\rangle \rightarrow \mu'$. Dans le premier cas, par induction, pour tout x_1 avec une nouvelle valeur dans mu', tel que $\Gamma'(x_1) = \tau_1'\,\alpha$, $\tau_1' \sqsupseteq \tau_1 \sqcup (pc \sqcup \tau_0)$. Dans le deuxième cas, toujours par induction, pour tout x_2 avec une nouvelle valeur dans mu', tel que $\Gamma''(x_2) = \tau_2'\,\alpha$, $\tau_2' \sqsupseteq \tau_2 \sqcup (pc \sqcup \tau_0)$. Par définition de $\sqcup$ sur les environnements, $\Gamma_+(x_1) \sqsupseteq \Gamma'(x_1)$ et $\Gamma_+(x_2) \sqsupseteq \Gamma''(x_2)$, sauf dans le cas où une variable de canal est mis à U dans Γ_+. Dans ce dernier cas, l'instrumentation sera appelée. Pour les autres cas, on sait que, selon la règle COND_S, $\tau = \tau_1 \sqcup \tau_2$. Ainsi, $\Gamma_+(x_1) \sqsupseteq \tau$ et $\Gamma_+(x_2) \sqsupseteq \tau$ et $\Gamma_+(x_1) \sqsupseteq pc$ et $\Gamma_+(x_2) \sqsupseteq pc$. Par conséquent, nous avons $\Gamma_+(x_1) \sqsupseteq \tau \sqcup pc$ et $\Gamma_+(x_2) \sqsupseteq \tau \sqcup pc$. Le confinement est donc satisfait.
   Supposons que $c =$ **while**. Le confinement est prouvé par induction pour chaque itération.

Pour compléter les propriétés que nous utilisons dans la preuve de cohérence, nous rajoutons le lemme suivant que nous tirons de [37]. Ce lemme pouvant être prouvé trivialement par induction structurelle sur l'exécution de la commande, nous omettons la preuve.

Lemme 2.2.3. *Soient μ et μ' des mémoires d'exécution et c une commande. Si $(c, \mu) \rightarrow \mu'$, $x \in dom(\mu)$ et que x n'est pas modifiée dans c, alors $\mu(x) = \mu'(x)$.*

Définition de la non-interférence La *non-interférence* signifie en gros qu'un changement sur une donnée d'entrée du programme ne cause pas de changement sur les données de sortie de niveau de sécurité inférieur à la donnée altérée. Cette politique de sécurité permet de manipuler et modifier des

données privées, pourvu que les sorties visibles du programme ne révèlent pas abusivement d'information sur ces données. Les données entrent dans le programme grâce aux canaux par les instructions $\mathbf{receive_n}$ ou $\mathbf{receive_c}$, tandis que leurs sorties sont accessibles sur les canaux de sortie via l'instruction **send**. Par conséquent, nous définissons une relation $\sim_\tau$ sur les mémoires de canaux d'un programme qui caractérise le pouvoir d'observation d'un utilisateur malveillant sur les entrées et sorties du programme à travers les canaux de communication.

Comme la relation $\sim_\tau$ dépend de l'intégrité des informations contenues dans les canaux de communication, nous définissons préalablement la notion d'égalité de canal :

Définition 2.2.1. *(Égalité de canaux) Deux canaux de communication nch_1 et nch_2 sont dits égaux, noté $nch_1 =_{ch} nch_2$, si $content(nch_1) = content(nch_2)$, où $content(ch)$ est la séquence des données contenues dans le canal ch.*

Intuitivement, $nch_1 =_{ch} nch_2$ signifie que les canaux nch_1 et nch_2 ont exactement le même contenu. Par conséquent, il va de soi que deux canaux égaux le restent après une même mise à jour.

Définition 2.2.2. *(τ-équivalence) Deux mémoires μ et ν sont dites τ-équivalentes, noté $\mu \sim_\tau \nu$, si $\forall x \in dom(\mu) \cap dom(\nu)$: $(\Gamma(x) = \tau'\ chan \wedge \tau' \sqsubseteq \tau) \Rightarrow \mu(x) =_{ch} \nu(x)$.*

Puisque nous considérons une seule classe observable L, nous utilisons dans la suite $\sim_L$, cela pouvant être généralisé à plus de classes. La relation $\sim_L$ relie deux mémoires indifférenciables aux yeux d'un utilisateur malveillant qui n'a accès qu'au contenu public.

Bien que nous nous intéressions uniquement au contenu des canaux, car ces derniers sont observables de l'extérieur du programme, il faut aussi nous assurer que les variables de valeur locales, communes à deux mémoires L-équivalentes, initialisées avec les mêmes valeurs dans les deux mémoires, gardent la même information tout le long de l'exécution du programme. En effet, le contenu d'une variable locale est susceptible d'être envoyé sur un canal observable. Ceci est formalisé par le lemme suivant :

Lemme 2.2.4. *Étant donné deux mémoires $\mu \sim_L \nu$, si pour toute variable entière x de type L val, nous avons $\mu(x) = \nu(x)$, (noté $\mu \sim_L^{val} \nu$), alors après l'exécution de programme sous μ et ν, produisant les mémoires μ' et ν' respectivement, nous aurons $\mu' \sim_L^{val} \nu'$.*
La preuve est omise ; il est clair que les variables ne peuvent être modifiées qu'avec un contenu provenant soit de canaux égaux (par $\sim_L$), soit de variables initialement à valeurs égales.

Nous pouvons alors donner la définition de la non-interférence, une variante de [35] :

Définition 2.2.3. *(Non-interférence) Un programme P satisfait la non-interférence si, pour deux mémoires données $\mu \sim_L \nu$ telles que $\mu \sim_L^{val} \nu$, les mémoires μ' et ν' produites par l'exécution de P sous μ et ν, respectivement, sont aussi L-équivalentes (si les deux exécutions se terminent avec succès).*
Dans cette définition, nous supposons que deux exécutions de P auraient exactement les mêmes processus externes d'écriture sur les canaux publics.

Preuve de cohérence

Théorème 2.2.1. *(Théorème de cohérence) Soient P un programme bien typé sous l'environnement Γ, sans besoin d'être instrumenté, et deux mémoires μ et ν L-équivalentes, telles que $\mu \sim_L^{val} \nu$. Si l'exécution de P sur μ et ν se termine avec succès, produisant les mémoires μ'et ν', respectivement, alors $\mu' \sim_L \nu'$.*

PREUVE. La preuve se fait par induction sur la structure de l'exécution $(P, \mu) \to \mu'$. P est bien typé sous Γ, sans besoin d'être instrumenté, signifie qu'on a $\Gamma, pc \vdash P : \tau\ cmd, \Gamma'$, avec $\Gamma'(\_instr) = L$.
L'étape de base a 6 cas.

1. Supposons que P soit une évaluation d'expression. Il n'y a pas de changement en mémoire, donc $\mu' \sim_L \nu'$ est, bien évidemment, respecté.
2. Supposons que $P =$ **skip**. Évident.
3. Supposons que $P =$ **receive**$_\mathbf{c}$ x_1 **from** x_2. L'exécution de P produit $\mu' = \mu[x_1 \mapsto read(\mu(x_2))]$ et $\nu' = \nu[x_1 \mapsto read(\nu(x_2))]$. Il est clair que $\mu' \sim_L \nu'$ puisque le contenu du canal x_2 est lu mais n'est pas modifié.
4. Supposons que $P =$ **receive**$_\mathbf{n}$ x_1 **from** x_2. L'exécution de P conduit à $\mu' = \mu[x_1 \mapsto nch_1]$ et $\nu' = \nu[x_1 \mapsto nch_2]$, où $nch_1 = read(\mu(x_2))$ et $nch_2 = read(\nu(x_2))$. Nous avons deux cas :

- $\tau = L$. Par hypothèse, $\mu(x_2) =_{ch} \nu(x_2)$, car $\mu \sim_L \nu$ et par définition 2.2.1, nous avons $nch_1 =_{nch} nch_2)$; ainsi, $\mu'(x_1) =_{ch} \nu'(x_1)$. De plus, x_2 n'est pas modifié. Par conséquent, $\mu' \sim_L \nu'$.
- $\tau \not\sqsubseteq L$. Le résultat est prouvé par le confinement 2.2.2.

5. Supposons que $P = x := e$. Ce cas est semblable à $\mathbf{receive_c}$.
6. Supposons que P soit la commande **send** x_1 **to** x_2 avec $\Gamma(x_1) = \tau_1\,\alpha$ et $\Gamma(x_2) = \tau_2\,\alpha$. L'exécution de P se termine avec les mêmes mémoires et applique les fonctions $update(\mu(x_2), \mu(x_1))$ et $update(\nu(x_2), \nu(x_1))$. Nous avons deux cas à considérer respectant la règle SEND_S :
   - $\tau = L$. Comme $\Gamma(\_instr)$ n'est pas mis à U, nous avons $\tau_1 \sqcup pc \leqslant \tau$ et $\neg(\tau_1 \sqcup pc = \tau = U)$. Donc, $\tau_1 = pc = L$. Or, par hypothèse, nous avons $\mu \sim_L^{val} \nu$ ($\alpha = val$), et $\mu \sim_L \nu$ ($\alpha = chan$) ; ce qui signifie que pour toute variable x_0 (entier ou canal) telle que $\Gamma(x_0) = \tau_0\,\alpha$ et $\tau_0 \sqsubseteq L$, $\mu(x_0) = (ou =_{ch}) \nu(x_0)$. Donc $\mu(x_1) = (ou =_{ch}) \nu(x_1)$. Par conséquent, étant donné que les canaux dans μ et ν sont égaux et mis à jour avec la même information, nous concluons que $\mu'(x_2) =_{ch} \nu'(x_2)$ et ainsi $\mu' \sim_L \nu'$.
   - $\tau \not\sqsubseteq L$. Comme précédemment, nous concluons que $\tau_1 = L$, puisque $\Gamma(\_instr)$ n'est pas mis à U. Il est donc clair que $\mu' \sim_L \nu'$ est respecté puisqu'aucun canal de type L n'est modifié.

L'étape d'induction a trois cas.

1. Supposons que $P =$ **if** e **then** c_1 **else** c_2 **end** avec $\Gamma, pc \vdash c_i : \tau_i\ cmd$, $i = 1$ ou 2 et $\Gamma, pc \vdash e : \tau_0\ val$. Par la règle COND_S, $\tau = \tau_1 \sqcap \tau_2$. Nous avons deux cas :
   - $\tau = L$. Par hypothèse d'induction, l'évaluation de e satisfait la L-équivalence ; par conséquent, si $\langle e, \mu\rangle \rightarrow_e n_1$ et $\langle e, \nu\rangle \rightarrow_e n_2$, alors $n_1 = n_2$. Ce qui signifie que l'exécution de P sous μ et ν conduit soit à $\langle c_1, \mu\rangle$ et $\langle c_1, \nu\rangle$, soit à $\langle c_2, \mu\rangle$ et $\langle c_2, \nu\rangle$. Dans les deux cas, par induction, les mémoires produites sont L-équivalentes.
   - $\tau \not\sqsubseteq L$. Par le lemme de confinement 2.2.2, nous savons que P modifie seulement les variables de type supérieur ou égal à $\tau \sqcup pc$. Ainsi, pour toute variable x modifiée dans P telle que $\Gamma'(x) = \tau'\,\alpha$, nous avons $\tau' \sqsupseteq (\tau \sqcup pc) \sqsupset L$; ainsi, $\tau' \sqsupset L$. Cela signifie que, s'il existe un x_0 tel que $x_0 \in dom(\Gamma')$, $\Gamma'(x_0) = \tau_0\,\alpha$ et $\tau_0 \sqsubseteq L$,

alors x_0 n'est pas modifié dans P. Donc par le lemme 2.2.3, nous avons $\mu(x_0) = \mu'(x_0)$ et $\nu(x_0) = \nu'(x_0)$; ainsi, $\mu'(x_0) = \nu'(x_0)$ puisque $\mu(x_0) = \nu(x_0)$ par hypothèse. Par conséquent, $\mu' \sim_L \nu'$ est satisfait.

2. Supposons que $P = c_1; c_2$. Si $\tau = L$, par hypothèse, les mémoires μ et ν "débutent dans un même état", puisque $\mu \sim_L \nu$. Donc par induction, l'exécution de c_1 conduit à des mémoires μ_1 et ν_1 qui sont L-équivalentes. Par conséquent, toujours par induction, exécuter c_2 sous μ_1 et ν_1 produit deux mémoires L-équivalentes. Dans le cas où $\tau \not\sqsubseteq L$, le résultat est prouvé par le lemme de confinement, de manière semblable au branchement.

3. Supposons que $P =$ **while** e **then** c **end**. Si $\tau = L$. Par hypothèse d'induction, l'évaluation de e satisfait la L-équivalence ; par conséquent, si $\langle e, \mu\rangle \rightarrow_e n_1$ et $\langle e, \nu\rangle \rightarrow_e n_2$, alors $n_1 = n_2$. Ce qui signifie que l'exécution de P sous μ et ν conduit soit à μ et ν, soit à $\langle c;$ **while** e **then** c **end**$, \mu\rangle$ et $\langle c;$ **while** e **then** c **end**$, \nu\rangle$. Dans le premier cas, le résultat est prouvé par hypothèse. Dans le deuxième cas, le résultat est prouvé par induction. Si $\tau \not\sqsubseteq L$, le résultat est prouvé par le lemme de confinement, de manière semblable au branchement.

2.2.5 Discussions

Parlons de la génération des faux positifs, c'est-à-dire, les cas de rejet d'un programme n'étant pas potentiellement dangereux. Un rejet de programme peut se produire lors de l'application d'une de ces trois règles : ASSIGN-CHAN_S , RECEIVE-NAME_S et SEND_S, quand $pc \not\leq \tau$, ou $\tau_1 \sqcup pc \not\leq \tau$. Cela n'est possible que pour la paire de valeurs (H, L), pour laquelle $H \not\leq L$. Si ni pc, ni τ_1, ni τ est incertain (de type U), il n'y aura pas rejet, uniquement une instrumentation. Selon nos règles, le type L ne peut être assigné à une variable que s'il s'agit de son vrai type, mais H peut être le résultat du supremum fait dans la règle COND_S ou LOOP_S. Un faux positif peut donc se produire lors du typage de **if** ou **while** dont la sentinelle prévient un mauvais branchement d'être pris ; un exemple serait le premier programme de la figure 2.5.

Si `public` est toujours faux, le programme est sécurisé, pourtant SEND_S le rejettera ; ceci est dû au supremum de la règle COND_S. Un utilisateur

$x := 0$;	$c :=$ `highChannel`
if `public`	**if** `private`
then $x :=$ `highValue` **end**;	**then** $c :=$ `lowChan`
send x **to** `lowChan`.	**end**.

FIGURE 2.5 – Faux positifs : incertitude générée par le **if** et l'assignation

qui ne veut aucun faux positif pourrait changer les paramètres afin que $H\ val \sqcup L\ val = U\ val$ au lieu de $H\ val$. Une autre situation où un faux positif peut se produire est lié à l'assignation et la réception d'un nom de canal. Considérons le second programme de la figure 2.5. Si `private` est toujours faux ou si c n'est jamais utilisé plus loins dans le programme, ce dernier est inoffensif mais pourtant rejeté. Ici aussi, il y a moyen d'éviter le faux positif : c'est d'introduire un quatrième type de sécurité, B, qui marquerait c comme non sûr et serait testé par SEND_S – ceci fait partie des travaux en suite à notre étude.

En résumé, les cas de faux positifs que nous détectons se produisent dans tous les autres analyses statiques, mais ils sont tolérés. Toutefois, nous suggérons des manières de les contourner.

2.3 Conclusion

Ce chapitre a porté sur le développement d'un système de types trivalué qui vérifie statiquement la non-interférence pour un langage de programmation impératif. La finalité de ce système est de rejeter (resp. accepter) les programmes ne respectant pas (resp. respectant) la non-interférence et de préparer à être instrumentés les programmes respectant peut-être la non-interférence.

Des travaux allant dans le même sens que notre approche ont été présentés par Khoury [19] pour la détection de code malicieux. Son approche est différente de la nôtre sur le point de l'analyse statique car il utilise des systèmes de type à effets tandis que notre approche se base sur des systèmes de types classiques annotés par des types de sécurité. Toutefois, nos deux approches suivent un raisonnement similaire de combinaison de l'analyse statique et de l'analyse dynamique, ceci afin de tirer avantage autant du faible coût d'exécution de l'analyse statique que de la précision de l'analyse dynamique.

Conclusion

Dans ce mémoire, nous nous sommes intéressés à l'application d'une politique de sécurité en rapport avec le contrôle du flot d'information dans un programme par une analyse de types trivaluée. Il s'agissait de la politique de non-interférence. Afin de définir notre cadre de travail, nous avons fourni, en premier lieu, un état de l'art au chapitre 1. Celui-ci présentait l'application de politiques de sécurité en général, le contrôle du flot d'information sûr, en particulier le contrôle par typage, et enfin, l'analyse trivaluée de programmes à travers quelques-unes de ses applications.

En deuxième lieu, nous avons décrit notre approche au chapitre 2. Comme son nom l'indique, cette approche visait à contrôler la sûreté du flot d'information dans un programme grâce à une analyse par typage trivaluée, ceci grâce à un système de types trivalué conçu dans le but de gérer le manque d'information au moment de la compilation de programme. Notre principale contribution a été la gestion du typage trivalué. Un programme est considéré bien typé, mal typé ou incertain. Dans le premier cas, le programme peut être exécuté sans danger, dans le second cas, le programme est rejeté et doit être modifié, et dans le troisième cas, l'instrumentation de code doit être faite pour pouvoir garantir le respect de la non-interférence. Cette approche permet d'éliminer des faux positifs dus aux approximations faites par l'analyse statique conservatrice, et introduit un temps d'exécution supplémentaire seulement si nécessaire. Nous obtenons moins de faux positifs que les approches purement statiques car nous envoyons à l'instrumentation certains programmes qu'elles auraient normalement rejetés.

Les travaux futurs viseront l'extension de notre analyse de types, dans le but d'en faire complètement une analyse de sécurité hybride, utilisant l'analyse de flot de données et l'instrumentation de code. Il s'agira d'insérer des tests dynamiques avant les instructions qui ont été détectées comme potentiellement non sûres durant l'analyse de types.

Bibliographie

[1] Anindya Banerjee and David A. Naumann. Secure information flow and pointer confinement in a java-like language. In *Proceedings of the IEEE Computer Security Foundations Workshop*, 2002.

[2] Gilles Barthe, Pedro R. D'Argenio, and Tamara Rezk. Secure information flow by self-composition. In *Proceedings of the IEEE workshop on Computer Security Foundations*, 2004.

[3] Gilles Barthe and Leonor Prensa Nieto. Secure information flow for a concurrent language with scheduling. *Journal of Computer Security*, 15 :647–689, 2007.

[4] David Elliott Bell. Looking back at the bell-la padula model. In *Computer Security Applications Conference, 21st Annual*, pages 15 pp. –351, dec. 2005.

[5] Glenn Bruns and Patrice Godefroid. Model checking partial state spaces with 3-valued temporal logics. In *Proceedings of the 11th International Conference on Computer Aided Verification*, CAV '99, pages 274–287, London, UK, 1999. Springer-Verlag.

[6] Marsha Chechik, Benet Devereux, Steve Easterbrook, and Arie Gurfinkel. Multi-valued symbolic model-checking. *ACM Transactions on Software Engineering and Methodology*, 12 :2003, 2003.

[7] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, pages 52–71, London, UK, 1982. Springer-Verlag.

[8] Edmund M. Clarke, E. Allen Emerson, and A. Prasad Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8 :244–263, 1986.

[9] Patrick Cousot and Radhia Cousot. Abstract interpretation : a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '77, pages 238–252, New York, NY, USA, 1977. ACM.

[10] Luís Damas and Robin Milner. Principal type-schemes for functional programs. In *POPL* [11], pages 207–212.

[11] Richard A. DeMillo. Conference record of the ninth annual acm symposium on principles of programming languages, albuquerque, new mexico, usa, january 1982. In *POPL*. ACM Press, 1982.

[12] Dorothy E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19 :236–243, May 1976.

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

[14] Aarhus Universitet. Regionale EDB-center and D.C. Kozen. *Results on the propositional mu-calculus*. DAIMI PB. DAIMI, 1982.

[15] Joseph A. Goguen and José Meseguer. Security Policies and Security Models. *IEEE Symposium on Security and Privacy*, pages 11–20, Avril 1982.

[16] Kevin W. Hamlen, Greg Morrisett, and Fred B. Schneider. Computability classes for enforcement mechanisms. *ACM Transactions on Programming Languages and Systems*, 28(1) :175–205, January 2006.

[17] G.E. Hughes and M.J. Cresswell. *A New Introduction to Modal Logic*. Routledge, September 1996.

[18] Sebastian Hunt and David Sands. On flow-sensitive security types. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, January 2006.

[19] Raphaël Khoury. Détection du code malicieux : système de type à effets et instrumentation de code. Master's thesis, Université Laval, 2005.

[20] Béchir Ktari. A 3-valued logic for the specification and the verification of security properties. In *Proceedings of the LICS'03 workshop on Foundations of Computer Security (FCS)*, pages 15–25, june 2003.

[21] Jay Ligatti, Jarred Ligatti, Lujo Bauer, and David Walker. Edit automata : Enforcement mechanisms for run-time security policies. *International Journal of Information Security*, 4 :2–16, 2003.

[22] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17 :348–375, 1978.

[23] Andrew C. Myers. Jflow : Practical mostly-static information flow control. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, 1999.

[24] François Pottier and Vincent Simonet. Information flow inference for ML. *ACM Transactions on Programming Languages and Systems*, 25 :117–158, January 2003.

[25] Henry Gordon Rice. Classes of Recursively Enumerable Sets and Their Decision Problems. *Transactions of the American Mathematical Society*, 74(2) :358–366, 1953.

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

[27] Andrei Sabelfeld and David Sands. Probabilistic noninterference for multi-threaded programs. In *Proceedings of the IEEE workshop on Computer Security Foundations*, 2000.

[28] Andrei Sabelfeld and David Sands. A per model of secure information flow in sequential programs. *Higher-Order and Symbolic Computation*, 14(1) :59–91, 2001.

[29] Fred B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3(1) :30–50, February 2000.

[30] Geoffrey Smith. A new type system for secure information flow. In *Proceedings of the IEEE workshop on Computer Security Foundations*, pages 115–125, 2001.

[31] Geoffrey Smith. A new type system for secure information flow. In *In 14th Computer Security Foundations Workshop*, pages 115–125. IEEE Computer Society Press, 2001.

[32] Geoffrey Smith. Probabilistic noninterference through weak probabilistic bisimulation. In *Proceedings of the IEEE Computer Security Foundations Workshop*, june-2 july 2003.

[33] Geoffrey Smith. Improved typings for probabilistic noninterference in a multi-threaded language. *Journal of Computer Security*, 14(6) :591–623, 2006.

[34] Geoffrey Smith. Principles of secure information flow analysis. In *Malware Detection*, pages 297–307. Springer-Verlag, 2007.

[35] Geoffrey Smith. Principles of secure information flow analysis. In *Malware Detection*, volume 27, pages 291–307. Springer, 2007.

[36] Tachio Terauchi and Alexander Aiken. Secure information flow as a safety problem. In *Proceedings of the International Symposium on Static Analysis*, 2005.

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

[38] Dennis Volpano and Geoffrey Smith. Probabilistic noninterference in a concurrent language. *Journal of Computer Security*, 7(2-3) :231–253, March 1999.

[39] Eran Yahav. *3VMC user's manual*. Disponible sur `http://www.math.tau.ac.il/~yahave`.

[40] Eran Yahav. Verifying safety properties of concurrent java programs using 3-valued logic. *SIGPLAN Not.*, 36 :27–40, January 2001.