# Obfuscation de code et de preuve — Rapport de bibliographie

François Dupressoir

Stage dans le projet Lande — Irisa

## 1 Introduction

Le développement de l'informatique portable a permis de formuler de nouveaux problèmes. Nous nous intéresserons ici à deux problèmes de sécurité apparemment orthogonaux : la distribution de programme sûrs et la distribution sûre de programme. Dans le premier problème, l'utilisateur ne fait pas confiance au distributeur du programme. Celui-ci doit donc fournir, avec le programme, une preuve que celui-ci satisfait la politique de sécurité de l'utilisateur. Cependant, on veut limiter la quantité d'informations transmises, tout en limitant aussi la quantité de calculs nécessaires à la vérification de la preuve, les ressources d'un appareil portable étant fortement limitées, et les coûts de communication étant souvent importants sur ce genre de machine. Dans le deuxième problème, le distributeur ne fait pas confiance à l'utilisateur, et souhaite protéger ses algorithmes en empêchant l'ingénierie inverse sur un programme souvent distribué sous une forme à fort contenu sémantique (bytecode) qui la facilite. Ici encore, plusieurs contraintes s'imposent de manière naturelle. On souhaite que le programme ainsi protégé soit encore exécutable sans matériel spécifique, qu'il ait encore la même fonction, et qu'il ne soit pas trop inefficace par rapport au programme original.

### 1.1 Preuve de programmes portables

Ainsi, nous souhaitons disposer d'un système de preuves effectuées *avant* la distribution du programme, mais vérifiable facilement et rapidement, sans pour autant augmenter la taille du téléchargement de manière trop importante. La technique du code porteur de preuve *(Proof-Carrying Code*, PCC par la suite*)* fournit une solution à ce problème, en intégrant aux programmes un certificat de preuve (qui consiste souvent en des propriétés ponctuelles, à la *triplets de Hoare*) très facilement vérifiable.
Dans ce domaine, l'interprétation abstraite apporte un formalisme qui permet de rendre systématique l'implémentation des analyses qui interviennent dans le prouveur et dans le vérifieur de preuve, tout en fournissant un cadre théorique suffisant pour prouver la correction de cette implémentation.

## 1.2 Obfuscation de programmes

Le deuxième inconvénient du logiciel portable est que la distribution doit se faire dans un format indépendant de la machine où le programme sera exécuté, soit un format qui contient encore une grande quantité d'informations sémantiques (un bytecode quelconque, Java, par exemple). Il nous faut donc disposer d'un moyen de protéger, sinon le code source, les algorithmes propriétaires qui y sont utilisés. Parmi les différentes solutions envisagées (*dongles*, exécution protégée...), celle qui semble la plus efficace pour une distribution à grande échelle est l'obfuscation (pour une comparaison plus avant des différentes méthodes de protection des logiciels Java, voir [CTL97]).
L'obfuscation consiste à modifier le flôt de contrôle et le flôt de données du programme pour rendre sa compréhension difficile (voire impossible) et ainsi éviter l'ingénierie inverse. Différentes méthodes ont été proposées pour ce faire, et la plupart sont décrites dans [CTL97]. Nous en décrirons plusieurs.

## 1.3 Obfuscation et PCC

Bien que PCC et obfuscation aient comme objectif un même souci de sécurité, les techniques employées à l'heure actuelle pour les implémenter sont, nous allons le voir, souvent incompatibles. Par exemple, l'implémentation d'un système de PCC demande au fournisseur d'expliciter des invariants sémantiques de son programme, alors qu'une obfuscation efficace de ce programme nécessiterait que cet invariant soit rendu opaque, obfusqué. De plus les transformations profondes souvent introduites par l'obfuscation au graphe de flôt de contrôle d'un programme rend beaucoup moins précise l'analyse du programme obfusqué. Il est donc nécessaire, pour garantir la sécurité pour le distributeur d'un programme comme pour son utilisateur, de rendre compatible les deux techniques.
L'interprétation astraite fournit une base théorique intéressante pour étudier les techniques de vérification de programme et les techniques d'obfuscation, et éventuellement pour les faire coopérer. Elle fera donc l'objet d'une première partie. La seconde partie concernera les techniques de PCC, présentées tout d'abord dans leur généralité, puis plus précisément dans le cadre du travail de stage. Enfin, la dernière partie traitera de l'obfuscation, présentant en premier lieu les techniques syntaxiques traditionnelles, avant de présenter une récente application de l'interprétation abstraite à l'étude de l'obfuscation.

# 2 Interprétation abstraite

## 2.1 La théorie

Cousot et Cousot [CC92, CC04b] sont à conseiller pour une présentation plus approfondie, mais encore abordable, de l'interprétation abstraite.

### 2.1.1 Abstraction

On souhaite étudier un programme $P$ vis-à-vis d'une certaine sémantique. Soit $Env$ l'ensemble des environnements (une représentation de l'état de la mémoire) possibles de ce programme dans cette sémantique. On appelle $\mathcal{A} = \mathcal{P}(Env)$ l'ensemble des propriétés de $P$, et on le munit d'une structure de treillis

complet : $(\mathcal{A}, \subseteq, \bigcup, \bigcap)$. Un abstraction de $\mathcal{A}$ est alors un ensemble $\overline{\mathcal{A}} \subseteq \mathcal{A}$. On dit que $p \in \mathcal{A}$ est correctement approchée par $\overline{p} \in \overline{\mathcal{A}}$ si $p \subseteq \overline{p}$ (on ne considère que des sur-approximations).
On veut de plus que, pour toute propriété $p$ dans $\mathcal{A}$, si $Q_p \subseteq \overline{\mathcal{A}}$ est l'ensemble des approximations correctes de $p$ ($Q_p = \left\{\overline{p} \in \overline{\mathcal{A}} | p \subseteq \overline{p}\right\}$), alors $Q_p$ admet un plus petit élément. On dit alors que $\overline{\mathcal{A}}$ est une *bonne abstraction* de $\mathcal{A}$.

**Bonne abstraction : fermeture supérieure**

**Définition 1** *Soit $(A, \sqsubseteq, \sqcup, \sqcap)$ un treillis complet. Une fonction $\rho \in A \to A$ est une* fermeture supérieure *si elle est :*
- *monotone ($\forall x, y \in A, x \sqsubseteq y \Rightarrow \rho(x) \sqsubseteq \rho(y)$),*
- *extensive ($\forall x \in A, x \sqsubseteq \rho(x)$),*
- *idempotente ($\forall x \in A, \rho(\rho(x)) = \rho(x)$).*

Il est facile de démontrer qu'une interprétation $\overline{\mathcal{A}} \subseteq \mathcal{A}$ est une bonne abstraction si elle est de la forme $\overline{\mathcal{A}} = \rho(\mathcal{A}) = \{\rho(p) | p \in \mathcal{A}\}$ où $\rho$ est une fermeture supérieure. On désignera par la suite par $uco(\mathcal{P}(\mathcal{A}))$ l'ensemble des fermetures supérieures *(upper closure operator)* de $\mathcal{P}$ lorsque $\mathcal{P}$ est un ensemble partiellement ordonné. Une fermeture supérieure étant aussi une bonne abstraction, $uco(\mathcal{P})$ désigne aussi l'ensemble des bonnes abstractions de $\mathcal{P}$.

**Bonne abstraction : connexion de Galois** Même avec une telle caractérisation, nous travaillons encore dans le même monde concret, restreint à quelques propriétés. Une abstraction supplémentaire peut-être utile pour systématiser la construction des abstractions, en oubliant les objets sous-jacents et en s'intéressant uniquement à la structure. Cette abstraction supplémentaire est fournie par les connexions de Galois.

**Définition 2** *Soit $(L_1, \sqsubseteq_1, \sqcup_1, \sqcap_1)$ et $(L_2, \sqsubseteq_2, \sqcup_2, \sqcap_2)$ deux treillis complets. Une paire de fonctions $\alpha \in L_1 \to L_2$ et $\gamma \in L_2 \to L_1$ est une* connexion de Galois *si elle vérifie*

$$\forall x_1 \in L_1, \forall x_2 \in L_2, \alpha(x_1) \sqsubseteq_1 x_2 \Leftrightarrow x_1 \sqsubseteq_2 \gamma(x_2)$$

*Si $\gamma$ est injective, on parle alors d'*injection de Galois*.*

On peut voir que $\gamma \circ \alpha$ est une fermeture supérieure de $L_1$, et il en découle qu'un treillis $(A^\#, \sqsubseteq^\#, \sqcup^\#, \sqcap^\#)$ est une bonne abstraction de $(\mathcal{A}, \subseteq, \cup, \cap)$ s'il existe une connexion de Galois entre $(\mathcal{A}, \subseteq, \cup, \cap)$ et $(A^\#, \sqsubseteq^\#, \sqcup^\#, \sqcap^\#)$.

### 2.1.2 Quelques remarques

**Un résultat important** Un des résultats les plus importants en interprétation abstraite concerne le transfert des points fixes. Il permet, en particulier, d'utiliser l'interprétation abstraite pour approcher les solutions de problèmes de points fixes du monde concret dans le monde abstrait.

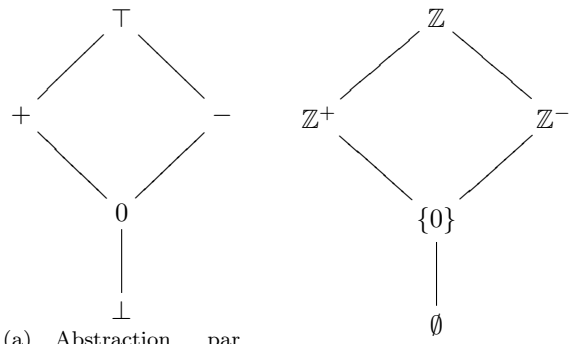
**Théorème 1** *Étant donné une connexion de Galois $(\mathcal{A}, \sqsubseteq, \sqcup, \sqcap) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (\mathcal{A}^\#, \sqsubseteq^\#, \sqcup^\#, \sqcap^\#)$, une fonction $f^\# \in \mathcal{A}^\# \to \mathcal{A}^\#$ monotone, et une fonction $f \in \mathcal{A} \to \mathcal{A}$ qui vérifient $\alpha \circ f \sqsubseteq f^\# \circ \alpha$. On a alors $\alpha(lfp(f)) \sqsubseteq lfp(f^\#)$. Le cas d'égalité est vrai aussi.*

**Composition de connexions de Galois** Un résultat qui pourra nous être utile par la suite, nous le verrons, concerne la composition des interprétations abstraites. En effet, la composée de deux connexions de Galois est une connexion de Galois. Ceci nous permet d'effectuer des transformations en série sans perdre la correction (mais en perdant de la précision si on se contente de composer les opérateurs abstraits).

**Remarques sur la portée de l'interprétation abstraite** Trouver une connexion de Galois ne suffit pas à implémenter les fonctions abstraites. Dans la pratique, en effet, la fonction $\alpha$ est rarement calculable, et on doit souvent se contenter d'approcher encore les résultats théoriques qui ne donnent que des spécifications.

### 2.1.3 Un exemple

On considère un programme à une seule variable entière. On a donc $\mathcal{A} = \mathcal{P}(\mathbb{Z})$. Prenons, dans un premier temps, $\overline{\mathcal{A}} = \{\emptyset, \mathbb{Z}^+, \mathbb{Z}^-, \mathbb{Z}\}$. Le propriété $p_1 = \{z \in \mathbb{Z} | z > 1\}$ (être plus grand que 1) est correctement approchée par $\mathbb{Z}^+$ et $\mathbb{Z}$. En revanche, $\overline{\mathcal{A}}$ n'est pas une bonne abstraction car la propriété $p_2 = \{0\}$ (être nul) est correctement approchée par $\mathbb{Z}^+$, $\mathbb{Z}^-$ et $\mathbb{Z}$, or $\{\mathbb{Z}^+, \mathbb{Z}^-, \mathbb{Z}\}$ n'a pas de plus petit élément.
Considérons maintenant l'abstraction $\overline{\mathcal{A}} = \{\emptyset, \{0\}\, \mathbb{Z}^+, \mathbb{Z}^-, \mathbb{Z}\}$. Celle-ci est une bonne abstraction car elle est clairement l'image de $\mathcal{A}$ par une fermeture supérieure. Le but de chercher une connexion de Galois serait de s'affranchir des notations propres à $\mathbb{Z}$, et de travailler dans le treillis abstrait de la figure 1(a) au lieu de celui de la figure 1(b). La différence peut paraître minime, mais elle devient




(a) Abstraction par connexion de Galois

(b) Abstraction par fermeture supérieure

FIG. 1 – Deux abstractions de $\mathcal{P}$

importante lors de la génération systématique des opérateurs, permettant de ne réimplémenter que les transformations d'abstraction, et non les opérateurs abstraits (les $f^{\#}$).

## 2.2 La pratique : analyse statique

En analyse statique, on veut prouver des propriétés sémantiques sur l'état d'un programme à chaque point de son exécution (par exemple, aucun poin-

teur nul n'est déréférencé, aucune division par 0 n'est effectuée...). En toute généralité, l'ensemble des états accessibles au cours de l'exécution d'un programme n'est pas calculable. Il convient donc d'approximer les propriétés syntaxiques tout en conservant la correction de l'analyse. On effectue donc une sur-approximation de ces propriétés, entrant de cette manière dans le domaine de l'interprétation abstraite. De plus, effectuer une analyse statique revient à résoudre un problème de point fixe sur la propriété concrète étudiée. L'interprétation abstraite permettant de transférer les points fixes concrets dans le domaine abstrait, elle est toute indiquée pour calculer des sur-approximations de ces points fixes.
De plus, nous venons de voir que l'interprétation abstraite fournissait, en plus d'une mesure de la qualité d'une abstraction, une spécification des opérateurs abstraits. Cela rend plus aisée la génération systématique d'analyses statiques une fois connue la propriété à étudier et son abstraction.
Cousot et Cousot [CC92] présentent quelques exemples simples d'analyses statiques formulées par interprétation abstraite.

# 3 PCC

## 3.1 Généralités

La technique du PCC repose sur l'observation que les vérifications statiques permettent de meilleurs performances que les vérifications à l'exécution, comme par exemple celles utilisées par la machine virtuelle Java pour les indexations de tableaux. Ainsi, il est préférable de vérifier au chargement d'une application, d'une bibliothèque, d'un module, etc. qu'elle est conforme à la politique de sécurité du système ou de l'application qui l'utilise. Ceci doit pouvoir être fait sous diverses formes (bytecode ou code natif), et ne doit nécessiter aucune hypothèse de confiance par rapport au distributeur du fichier binaire considéré.
A ces nécessités, on peut ajouter la remarque que tout effort pour développer un système de vérification de programme se heurte à un choix entre :

- la taille du certificat (de la preuve) pour un programme donné,
- la simplicité de la vérification (en temps et en ressources).

Necula [Nec02] présente une méthode générale pour implémenter du PCC et compare deux possibilités pour représenter les preuves. L'une très générale, permet une vérification très rapide de propriétés complexes tandis que l'autre, à base d'oracles, ne permet que de vérifier des propriétés simples, mais réduit la représentation de la preuve par un facteur 20.
Le schéma général qu'il propose, tiré de [Nec02] est donné en figure 2. Le binaire envoyé par le distributeur contient l'exécutable et le certificat de preuve. L'exécutable est tout d'abord fourni à un générateur de conditions, semblable à celui utilisé lors de la génération du certificat. Celui-ci génère des invariants à vérifier à chaque point critique du programme (envoi de message, accès à un tableau, accès mémoire hors-limites...) et les fournit à un vérifieur. Le vérifieur reçoit les conditions et s'aide des invariants fournis par les certificat pour vérifier :

1. que les propriétés exprimées font partie de l'espace des propriétés qu'on souhaite vérifier,
2. que le certificat correspond bien au programme fourni,
3. que le certificat prouve l'adhésion du programme à la politique de sécurité.

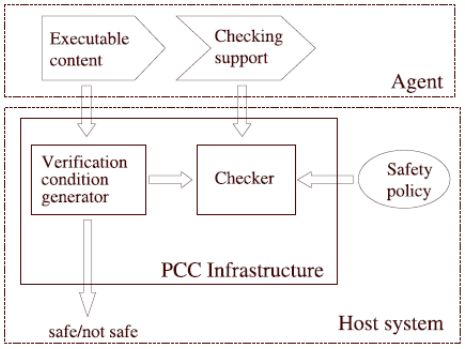



Fig. 2 – Un schéma général d'implémentation de PCC

## 3.2 PCC à base d'interprétation abstraite

Albert, Puebla et Hermenegildo [APH04] remarquent que cette recherche d'invariants pour générer un certificat, et la vérification rapide de ces invariants correspondaient à une recherche de point fixe en analyse statique, et en interprétation abstraite. Ils définissent donc un système de PCC pour des programmes logiques fondé sur l'interprétation abstraite, leur permettant d'utiliser le caractère systématique de cette dernière pour dériver de nombreuses propriétés de manière aisée, au lieu de définir une nouvelle analyse pour chaque propriété à étudier. Il suffit en effet de calculer une « bonne abstraction »du programme vis à vis de la propriété étudiée (la qualité de l'abstraction étant donnée par la théorie de l'interprétation abstraite) et de trouver ses points fixes pour générer le certificat. Le vérifieur de son côté, calcule à nouveau cette abstraction du programme, mais se contente de vérifier que les invariants fournis constituent bien un point fixe de cette abstraction. Ceci permet de vérifier relativement rapidement des propriétés complexes, tout en n'utilisant qu'une représentation minimale des invariants, spécifique à la propriété étudiée.
Besson, Jensen, Pichardie et Turpin [BJP06, BJPT07] exploitent à nouveau cette idée d'utiliser l'interprétation abstraite pour générer et vérifier les certificats, appliquée cette fois à (un sous-ensemble) du bytecode Java. De plus, le système de vérification ainsi défini est certifié en Coq, permettant une confiance encore plus élevée en la sûreté du contenu exécutable.
Cependant, si l'analyse ainsi définie se comporte généralement bien sur du code compilé proprement, elle perd énormément de précision lorsque certains motifs sont présents dans l'environnement concret. Il est clair que des modifications comme celles dont nous allons parler maintenant, intervenant directement sur le graphe de flôt de contrôle, peuvent introduire ces motifs et donc faire perdre de son intérêt au système de preuve.
C'est sur le système de preuve défini par l'équipe Lande ([BJPT07]) que le travail de stage portera.

# 4 Obfuscation

## 4.1 Obfuscation « syntaxique »

Collberg, Thomborson et Low [CTL97] définissent une transformation $\tau : P \mapsto P'$ comme une transformation d'obfuscation si elle conserve le comportement observable de $P$ (c.-à-d. si $P$ ne termine pas ou termine sur un état d'erreur, alors $P'$ peut terminer ou pas, et si $P$ termine, alors $P'$ termine et avoir les même sorties). Ils définissent deux propriétés des transformations d'obfuscation, la puissance (*potency*) et la résistance (*resilience*) qui mesurent, respectivement, la complexité[1] ajoutée au programme et la difficulté d'inverser la transformation sur un programme donné. Le principal problème dans ce domaine vient du fait que la complexité est subjective, et qu'il existe autant (voire plus) de métriques pour la mesurer que de transformations d'obfuscation, empêchant tout effort de réelle comparaison de ces méthodes.
Cependant, on peut distinguer deux grandes classes de transformations, orthogonales et complémentaires pour avoir un outil d'obfuscation puissant.

### 4.1.1 Obfuscation du flôt de contrôle

Les transformations de cette classe modifient le flôt de contrôle du programme de manière à rendre plus complexe la compréhension de l'algorithme sous-jacent. La plupart d'entre elles se basent sur l'utilisation de *prédicats opaques*, des prédicats dont la valeur est connue à la compilation, mais dont la connaissance n'est pas directe à l'analyse. Leur utilisation permet d'insérer du code mort (qui peut donc être arbitraire et troubler un attaquant, voir figure 3), de faire interférer des méthodes (en mélangeant le code de deux méthodes $m$ et $m'$ et en les gardant par un prédicat qui vaut *true* à chaque appel de $m$ et *false* à chaque appel de $m'$, par exemple). L'inconvénient de ces méthodes est que les prédicats opaques peuvent être détectés par des analyses dynamiques, rendues encore plus faciles par les outils d'instrumentation (voir [Dal07] pour une discussion sur ce point). D'autres transformations, comme la parallélisation

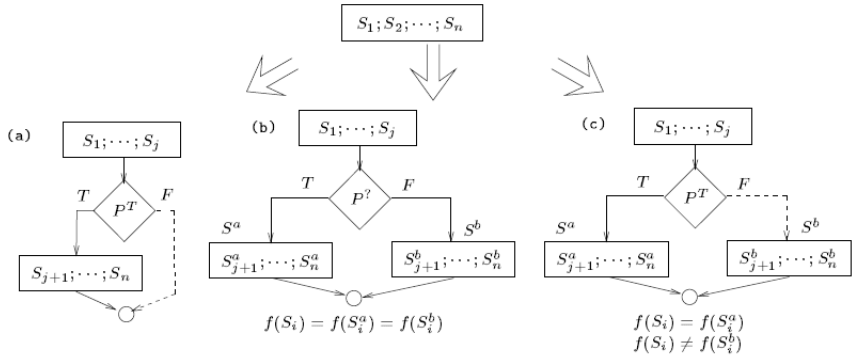



FIG. 3 – Un exemple d'obfuscation de flôt de contrôle tiré de [CTL97]

de code, ou la transformation du graphe de flôt de contrôle en un graphe non

[1] qui n'a rien à voir avec la complexité algorithmique

réductible (non représentatif d'une construction du langage source) sont plus puissantes, mais plus coûteuses.

### 4.1.2 Obfuscation des données

Les transformations de cette classe modifient le flôt de données du programme. Souvent, elles s'intéressent à la représentation des données en mémoire (représentation d'entiers sous forme d'objets, par exemple), ou à la structure même de la hiérarchie de classes. Elles sont souvent plus puissantes que les transformations du flôt de contrôle, mais elles sont aussi plus chères à mettre en œuvre en temps d'exécution.

### 4.1.3 Discussion sur l'utilité de l'obfuscation

Une des grandes discussions porte sur l'utilité de l'obfuscation en tant qu'outil de protection du logiciel. En effet, mis à part quelques transformations triviales qui ne sont pas réversibles (suppression du formatage, renommage des variables...), toutes les transformations actuellement utilisées sont réversibles. En fait, Appel argue même que les transformations inverses sont dans NP ([App01]). Cependant, [CTL97] montre que c'est la solution la plus adaptée à une distribution de bytecode à grande échelle, à l'heure actuelle.

## 4.2 Obfuscation « sémantique »

L'idée d'utiliser l'interprétation abstraite à des fins de protection du logiciel a été introduite pour la première fois dans [CC04a], où Cousot et Cousot définisent de manière générale une technique de filigranage *(watermarking)* de code source par interprétation abstraite. Mais le filigranage ne protège que contre la distribution non autorisée, et non contre l'ingénierie inverse qui nous occupe ici. Dans sa thèse ([Dal07]), Mila Dalla Preda s'intéresse à l'interprétation abstraite dans le cadre de la détection de *malware*, et en particulier dans le cadre de la désobfuscation des *malware* pour permettre leur reconnaissance par signature. Elle est ainsi amenée à donner une nouvelle définition de ce qu'est une transformation obfusquante, ainsi qu'à exprimer différentes techniques d'obfuscation comme des interprétations abstraites. Son étude se fonde sur l'observation que l'ensemble des exécutions possibles d'un programme (sa sémantique de traces) peut-être approchée de différentes manières (par exemple, sa sémantique naturelle dénotationnelle, sa sémantique grands-pas...) et qu'il existe un ordre partiel entre ces différentes approximations.

### 4.2.1 Treillis des sémantiques abstraites

Plus formellement, désignons par $\Sigma^+$ la sémantique de trace finie maximale (la même idée s'applique en réalité à la sémantique de trace infinie maximale $\Sigma^\omega$, mais la définition de *comportement observable* d'un programme permet de ne s'intéresser qu'aux exécutions finies), que nous appellerons par la suite sémantique concrète. Toutes les autres sémantiques (finies) peuvent être obtenues par abstraction de $\Sigma^+$. Ainsi, $uco(\mathcal{P}(\Sigma^+))$ désigne le treillis des fermetures supérieures des sémantiques, et un élément $\phi \in uco(\mathcal{P}(\Sigma^+))$ est une propriété sémantique. Dans ce treillis, $\bot$ est la propriété concrète qui décrit la sémantique

elle-même (chaque sémantiques est sa propre abstraction), et $\top$ est la propriété abstraite où toutes les sémantiques sont confondues ($\Sigma^+$ abstrait toutes les sémantiques).

### 4.2.2 Définitions

**Puissance** Travailler au niveau sémantique permet de s'affranchir de la notion de comportement observable d'un programme pour définir les transformations d'obfuscation. En effet, la relation entrées/sorties d'un programme qui définit ce comportement observable n'est ici rien d'autre qu'une abstraction de la sémantique de trace. Dalla Preda donne donc une définition formelle à la notion de *puissance* d'une transformation.

**Définition 3** *Une transformation $\tau$ est* puissante *s'il existe une sémantique $\phi \in uco(\mathcal{P}(\Sigma^+))$ et un programme $P$ tel que $\phi(P) \neq \phi(\tau(P))$.*

La puissance d'une transformation $\tau$ peut donc être définie comme la propriété $\delta_\tau$ la plus concrète préservée par $\tau$, soit $\delta_\tau = \sqcap \{\phi \in uco(\mathcal{P}(\Sigma^+)) | \forall P : \phi(P) = \phi(\tau(P))\}$. On dit alors que $\tau$ est un *$\delta$-obfuscateur* si $\delta = \delta_\tau$ et $\delta_\tau \neq \bot$. Intuitivement, une transformation est d'autant plus puissante qu'elle cache des propriétés sémantiques plus concrètes.

**Obfuscateur classique** Une conséquence directe de cette définition est qu'elle généralise de manière directe la définition de transformation d'obfuscation de Collberg et coll. [CTL97]. En effet, la sémantique dénotationnelle d'un programme est une abstraction de son comportement observable, puisqu'elle met en relation ses entrées et ses sorties. Ainsi, la classe $\mathbb{O}$ d'obfuscateurs définie par Collberg, Thomborson et Low peut être caractérisée comme l'ensemble des $\delta$-obfuscateurs avec $\delta \sqsubseteq DenSem$, où $DenSem$ désigne la sémantique dénotationnelle.

**Comparer les transformations** De même, en utilisant cette définition de la puissance d'une transformation, il devient facile de comparer les transformations d'obfuscation en comparant simplement les positions relatives de leur puissance dans le treillis des sémantiques abstraites. De plus, même si leurs puissances sont incomparables, il est toujours possible de définir une puissance *par rapport à une propriété sémantique $\phi$*, qui représente, de manière informelle, la « profondeur d'abstraction »à laquelle $\phi$ est cachée par chaque transformation.

### 4.2.3 Interprétation

Il peut être difficile de comprendre pourquoi cette définition correspond bien à l'idée qu'on se fait d'un obfuscateur. Nous donnons donc ici une interprétation très personnelle de celle-ci. Les sémantiques abstraites peuvent représenter un phénomène observable. La sémantique de trace maximale représente l'observation continue de l'ensemble de l'état de la mémoire à chaque étape de calcul ; la sémantique dénotationnelle, représente l'observation des états mémoire significatifs pour le calcul. Ainsi, obfusquer une propriété sémantique revient à la rendre inutilisable par un attaquant incapable d'observer à un tel niveau, l'obligeant à inverser le travail d'abstraction après avoir observé un comportement à un niveau moins concret que la puissance de la transformation, un peu comme un médecin doit effectuer un diagnostic à partir de ses manifestations

(symptômes) abstraites.
La transformation qui remplace chaque variable par son double, par exemple, obfusque les propriétés sémantiques « est pair »et « est impair », mais préserve les propriétés de signe. Un attaquant qui cherche à vérifier une propriété de parité ne peut le faire sans d'abord inverser l'abstraction qui les a obfusquées.
Enfin, dans cette définition, toute transformation de programme est un obfuscateur potentiel. Dans sa thèse [Dal07], Dalla Preda étudie la puissance d'obfuscation de la propagation de constantes, une analyse statique très utilisée en optimisation, et comme base pour d'autres analyses statiques plus complexes.

# 5 Conclusion

Nous avons vu ici les bases de l'interprétation abstraite, et nous avons expliqué comment celle-ci pouvait être utilisée pour implémenter un système de PCC, et comment elle permettait d'étudier, de modéliser, et de définir de manière formelle l'obfuscation. Nous avons aussi montré l'intérêt de pouvoir à la fois prouver et obfusquer un programme, tout en mettant en évidence la difficulté de faire les deux à la fois de manière efficace. Le but de ce stage sera donc, dans un premier temps, d'étudier comment rendre le système de preuve défini par Besson, Jensen, Pichardie et Turpin [BJPT07] moins sensible à l'obfuscation, ou du moins de définir des classes d'obfuscateurs qui ne font pas perdre à leur analyseur sa précision, sans pour autant diffuser à travers les invariants des informations rendant la désobfuscation trop facile. En parallèle, on pourra étudier la possibilité d'obfusquer du code porteur de preuve en obfusquant la preuve en même temps que le code. Dans un second temps, il pourra être intéressant d'étudier la piste de la composition d'interprétations abstraites et de connexions de Galois pour pouvoir composer analyse statique et obfuscation, et en particulier, il pourra être intéressant d'étudier l'importance de l'ordre de cette composition (obfuscation puis analyse, ou analyse puis obfuscation), et les conséquences de cette composition (qui peut bien sûr entrelacer diverses transformations d'obfuscation et analyses statiques) sur la précision des preuves. Enfin, il pourrait aussi être intéressant d'étudier la possibilité d'intégrer l'obfuscation et la preuve au processus de compilation en modélisant le processus de compilation en une suite d'interprétations abstraites appropriées.

# Références


[APH04] Elvira Albert, Germán Puebla et Manuel V. Hermenegildo : Abstraction-carrying code. *In Logic for Programming, Artificial Intelligence and Reasoning*, pages 380–397, 2004.

[App01] Andrew W. Appel : Deobfuscation is in NP (draft), August 2001.

[BJP06] Frédéric Besson, Thomas Jensen et David Pichardie : Proof-carrying code from certified abstract interpretation and fixpoint compression. *Theoretical Computer Science*, 364(3):273–291, 2006.

[BJPT07] Frédéric Besson, Thomas Jensen, David Pichardie et Tiphaine Turpin : Result certification for relational program analysis. Rapport de recherche 6333, INRIA Rennes, IRISA, Campus universitaire de Beaulieu, 35042 Rennes Cedex, France, October 2007.

[CC92] Patrick Cousot et Radhia Cousot : Abstract interpretation and application to logic programs. *Journal of Logic Programming*, 13(2–3):103–179, 1992.

[CC04a] Patrick Cousot et Radhia Cousot : An abstract interpretation-based framework for software watermarking. *In Conference Record of the Thirtyfirst Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 173–185, Venice, Italy, janvier 14-16 2004. ACM Press, New York, NY.

[CC04b] Patrick Cousot et Radhia Cousot : *Basic Concepts of Abstract Interpretation*, pages 359–366. Kluwer Academic Publishers, 2004.

[CTL97] Christian Collberg, Clark Thomborson et Douglas Low : A taxonomy of obfuscating transformations. Rapport technique 148, Department of Computer Science – The University of Auckland, July 1997.

[Dal07] Mila Dalla Preda : *Code Obfuscation and Malware Detection by Abstract Interpretation.* Thèse de doctorat, Università degli Studi di Verona – Dipartimento di Informatica, 2007.

[Nec02] George Necula : *Proof-Carrying Code. Design and Implementation.*, pages 264 – 288. Kluwer Academic Publishers, 2002.