Politique de contrôle d'accès multi-niveaux : test de conformité vis à vis des flots avec l'outil FoCaL

Matthieu Carlier, Catherine Dubois*

Lionel Habib, Mathieu Jaume†

Résumé

Les politiques de contrôle d'accès multi-niveaux permettent de confiner l'information en garantissant des propriétés de confidentialité et d'intégrité. Ces politiques reposent sur un ensemble de niveaux de sécurité muni d'un ordre partiel. Elles permettent d'une part de spécifier quels sont les accès autorisés dans un système d'information lorsque ce système se trouve dans un certain état, et, d'autre part, d'exprimer, en terme de niveau de sécurité, les flots d'information autorisés durant la vie de ce système. Nous présentons ici l'utilisation d'un outil de test intégré à l'environnement de développement FoCaL pour vérifier l'adéquation entre ces deux lectures sémantiques d'une politique de contrôle d'accès multi-niveaux. Cet outil permet de détecter des erreurs dans une spécification formelle mais aussi dans une implantation. Nous montrons également l'incidence de la façon d'écrire les propriétés à tester sur le mécanisme de génération des jeux de test.

Mots clés : Contrôle d'accès, Test, Méthodes formelles, FoCaL

1 Introduction

Les méthodes formelles peuvent être définies comme des méthodes soutenues par une rigueur mathématique. Leur utilité première est leur imperméabilité à toute ambiguïté. Cela est très apprécié lorsqu'il s'agit de spécifier (décrire ce qui est requis d'un système) ou encore lorsqu'il s'agit de développer (architecture, code). De plus, elles permettent très naturellement de démontrer le respect de certaines propriétés et donc, d'augmenter la confiance dans le système, que ce soit sur la cohérence de sa spécification ou sur la bonne adéquation de son développement à cette spécification. Lorsqu'il s'agit de systèmes logiciels critiques, ces propriétés peuvent être vitales. Si la mise en œuvre des méthodes formelles permet de garantir les propriétés

*CPR - CEDRIC - ENSIIE, Evry, France

†SPI - LIP6 - UPMC, Paris, France

essentielles d'un algorithme, celles des mécanismes d'exécution d'un programme ou encore des propriétés de sécurité d'un système d'information, cette mise en œuvre a un coût certain : temps de développement, connaissances requises, technicité du développement ... Aussi, il convient de chercher à réduire ce coût non seulement en étudiant les techniques de réutilisation de développements formels, mais aussi en outillant ces méthodes.

Un développement formel consiste généralement à déclarer les objets considérés, à spécifier les propriétés attendues de ces objets, à en donner une définition puis à prouver les propriétés énoncées. C'est souvent l'étape de preuve qui est la plus chronophage. Il est donc préférable de n'aborder cette étape que lorsque les définitions et propriétés considérées sont cohérentes. Toutefois, l'écriture des définitions, spécifications et propriétés requiert un niveau de détail élevé et est donc sujette à des omissions et/ou des erreurs. Avant de chercher à prouver formellement les propriétés, il peut donc être avantageux de les tester afin de détecter d'éventuelles erreurs.

Dans cet article, nous utilisons l'atelier FoCaL [4] qui permet de spécifier, d'implanter et de prouver la correction d'une implantation par rapport à une spécification, et ce petit à petit. Cet atelier intègre un outil de test, FoCalTest, développé par M. Carlier et C. Dubois [1]. Cet outil permet de trouver des erreurs dans une implantation ou une spécification. Un des objectifs premiers de cet outil est de permettre d'apprécier la validité d'une propriété requise par une implantation avant de chercher à la démontrer. L'outil met en œuvre une approche *boîte noire* où les jeux de test sont générés aléatoirement. Le verdict du test est donné par l'évaluation de la propriété sous test (qui doit donc avoir une forme exécutable) en utilisant l'implantation sous test. L'article décrit l'utilisation de FoCalTest pour corriger le développement formel d'un modèle de contrôle d'accès multi-niveaux. Les politiques de contrôle d'accès multi-niveaux permettent de confiner l'information en garantissant des propriétés de confidentialité et d'intégrité. Ces politiques reposent sur un ensemble de niveaux de sécurité muni d'un ordre partiel. Elles permettent d'une part de spécifier quels sont les accès autorisés dans un système d'information lorsque ce système se trouve dans un certain état, et, d'autre part, d'exprimer, en terme de niveau de sécurité, les flots d'information autorisés durant la vie de ce système. Dans cet article nous illustrons comment la phase de test peut aider à vérifier l'adéquation entre ces deux lectures sémantiques d'une politique de contrôle d'accès multi-niveaux. Nous montrons également l'incidence de la façon d'écrire les propriétés à tester sur le mécanisme de génération des jeux de test.

Dans la section 2, nous décrivons brièvement l'atelier Focal et son outil de test. La section 3 introduit et formalise les notions de politique de confidentialité et de politique de flots. Dans la section suivante, nous décrivons les erreurs que nous introduisons dans les spécifications et/ou dans le code, erreurs que nous tentons de retrouver avec l'outil FoCalTest. Notons que l'une de ces erreurs correspond explicitement à une erreur de spécification

faite dans [8]. La section 5 décrit la mise en œuvre dans l'atelier FoCaL et explicite les résultats obtenus lors de l'utilisation de FoCalTest. Enfin nous concluons et ouvrons quelques perspectives.

Travaux connexes

La vérification de politiques de contrôle d'accès est un sujet qui a suscité de nombreux travaux récemment, en utilisant des approches diverses. Dans [10, 9, 2], les auteurs décrivent un algorithme permettant de détecter la « fuite d'information » correspondant à l'exemple présenté dans la section précédente à partir d'une spécification de politique par un système de réécriture (la description de politiques de sécurité *via* un système de réécriture est présentée dans [2]). Dans [5], les auteurs présentent une approche basée sur des techniques de model-checking pour la vérification de propriétés de sécurité que doivent garantir des politiques de contrôle d'acces. D'autres travaux abordent cette vérification par le biais du test comme par exemple [11] et [7]. Un point commun entre ces travaux et notre approche concerne en particulier l'injection de fautes dans une politique de contrôle d'accès et l'usage de techniques de test mutationnel. Cependant [11] et [7] exploitent un modèle de fautes et des opérateurs mutationnels spécifiques aux politiques de contrôle d'accès alors que notre expérimentation se place dans le cadre d'un atelier général. Dans cet article nous avons injecté des erreurs dans la fonction de transition uniquement alors que les opérateurs mutationnels de [11] et [7] permettent d'agir sur différents constituants de la politique de contrôle d'accès.

2 FoCaL et FoCaLTest

L'atelier FoCaL

L'environnement FoCaL fournit un langage construit sur OCaml et Coq pour spécifier, programmer et prouver des unités de bibliothèques. Il permet donc à la fois d'écrire des programmes et de prouver certaines propriétés qu'ils vérifient. Les unités de bibliothèques sont traduites par le compilateur FoCaL selon différents formats : vers du code source OCaml pour aboutir à des programmes exécutables, vers un format appelé FocDoc générant des fichiers XML de documentation, et vers du code source Coq afin de vérifier les preuves. L'environnement FoCaL propose un langage d'expression de propriétés qui peuvent être vérifiées automatiquement et/ou prouvées interactivement soit à l'aide de Coq, soit à l'aide de Zenon, un outil permettant de construire une preuve en s'appuyant sur la structure particulière des objets spécifiés dans l'environnement FoCaL. Cette preuve est ensuite automatiquement vérifiée par Coq. En outre, le langage de FoCaL permet un développement modulaire par passage progressif de la spécification à

l'implantation grâce aux traits objets dont il dispose (héritage, redéfinition, instanciation, etc).

En FoCaL, les structures sont représentées par des espèces. L'implantation des bibliothèques peut être effectuée étape par étape grâce à la notion d'héritage qui relie les espèces entres elles. Une espèce définit une structure par un ensemble de méthodes. A l'intérieur d'une espèce, certaines méthodes peuvent être uniquement déclarées (donc non définies), ce qui permet de les manipuler sans connaître leur implantation réelle. La notion d'héritage permet de définir une espèce à partir d'autres espèces préalablement définies en héritant de toutes les méthodes de ses parents. Les collections permettent de coder des domaines particuliers et sont associées à des espèces complètement définies, toutes leurs méthodes doivent être définies et toutes leurs obligations de preuve déchargées. Les entités sont les éléments manipulés à l'intérieur des espèces. Le type support (*carrier* en anglais) d'une espèce est le type de ses entités. Chaque espèce contient un unique type support, éventuellement hérité. Les méthodes représentent les opérations de base ou les propriétés des structures. Il y a cinq catégories de méthodes : le *type support* (déclaré sous forme d'un type abstrait, ou bien lié à un type de données (`int`, `list(string)`, etc)), les *signatures* représentant les opérations *déclarées* (introduction du nom et du type de l'opération), les *fonctions* représentant les opérations *définies* (introduction du nom, du type et de la définition de l'opération), les *propriétés* (déclarées) et les *théorèmes* (prouvés, donc définis) vérifiés par l'espèce et impliquant des obligations de preuve. De plus, à toute espèce est implicitement associée une interface, obtenue en effaçant le corps des méthodes définies (type support, fonctions ou théorèmes).

L'atelier FoCaL a été utilisé avec succès, entre autres, pour implanter une bibliothèque de calcul formel, pour spécifier et valider la politique de sécurité au sol des aéroports [3] et est actuellement utilisé pour implanter une bibliothèque de modèles de contrôle d'accès [6]. De plus, comme nous le verrons par la suite, l'outil FoCalTest offre la possibilité de tester automatiquement un développement effectué dans l'atelier FoCaL.

FoCalTest

FoCalTest est l'outil de test intégré à FoCaL. Il permet de confronter automatiquement une propriété à une implantation. Il génère des jeux de test, les soumet et rend un rapport de test au format XML. L'outil FoCalTest prend donc en entrée d'une part des propriétés et d'autre part une implantation des méthodes référencées dans les propriétés. Les propriétés sous test, de la forme présentée dans la figure 1, sont en forme prénexe sans quantificateur existentiel. Les α_i sont des formules construites avec les connecteurs $\vee$, $\wedge$ et $\neg$, A, A_i et B_i dénotent des prédicats qui au niveau implantation correspondent à des appels de méthodes FoCaL. Un exemple de propriété

$$\forall X_1 \in \tau_1 \ldots X_n \in \tau_n.\ \alpha_1 \Rightarrow \cdots \Rightarrow \alpha_n \Rightarrow (A_1^1 \vee \cdots \vee A_{n_1}^1) \wedge \cdots \wedge (A_1^m \vee \cdots \vee A_{n_m}^m)$$

$$\alpha ::= \alpha \vee \alpha \mid \alpha \wedge \alpha \mid A$$

FIG. 1 – Forme des propriétés testées

testable est donnée ci-dessous :

$$\forall x \in \texttt{int}\ y \in \texttt{int}.\ \texttt{equal}(x, y) \wedge \texttt{int_inf}(x, y) \Rightarrow \texttt{equal}(y, x)$$

Dans cette propriété, `equal` et `int_inf` sont des prédicats logiques pourvus d'une implantation, respectivement, l'égalité et la relation d'ordre des entiers machines.

La forme retenue des propriétés testables permet de prendre en compte un large éventail de propriétés, elle recouvre la presque totalité des propriétés énoncées dans la bibliothèque standard distribuée avec l'atelier FoCaL. Elle permet d'exprimer des propriétés algébriques, des propriétes à la *pré-post* ou encore des relations métamorphiques. Notons que l'on peut relâcher quelque peu cette forme en admettant des propriétés quantifiées sur des variables d'un type fini (relaxation de la forme prénexe). Nous y reviendrons ultérieurement dans la section 5. Pour les besoins du test, les propriétés sous test sont réécrites en un ensemble de propriétés plus simples appelées *propriétés élémentaires*. La conjonction des propriétés élémentaires donne une propriété équivalente à la propriété initiale. Cette transformation est présentée dans [1]. Les propriétés élémentaires sont de la forme (les A_i et B_i ont la même signification que dans les propriétés testables) :

$$\forall X_1 \in \tau_1 \ldots X_n \in \tau_n.\ A_1 \Rightarrow \cdots \Rightarrow A_n \Rightarrow B_1 \vee \cdots \vee B_m$$

On distingue deux parties dans de telles propriétés, la précondition et la conclusion : la précondition est la conjonction des A_i et la conclusion est la prémisse droite de la suite d'implications, soit la formule $B_1 \vee \cdots \vee B_m$.

L'approche adoptée par FoCalTest consiste à prendre une à une les formes élémentaires, et pour chacune, générer des jeux de test et les appliquer sur la propriété sous test en utilisant l'implantation précisée par l'utilisateur. Un jeu de test pour une propriété élémentaire est une suite de valeurs $X_1 = v_1, \ldots X_n = v_n$. Il est considéré comme valide s'il satisfait la précondition de la propriété. Si la précondition n'est pas satisfaite, le jeu de test n'est pas pertinent car il ne permet pas d'apprécier la conclusion. Un jeu de test valide est ensuite utilisé pour évaluer la conclusion. On utilise l'implantation pour vérifier la satisfaction de la précondition et de la conclusion par un jeu de test donné. Si la conclusion est vérifiée alors le jeu de test *passe*, dans le cas contraire FoCalTest a trouvé un contre-exemple qui met

en évidence que l'implantation des méthodes mises en jeu dans la propriété sous test ne respecte pas la propriété sous test. Dans cet article, les jeux de test sont générés automatiquement et de manière pseudo-aléatoire. Les générateurs sont construits automatiquement par FoCalTest qui s'appuie pour cela sur la structure des types des variables quantifiées de la propriété. Pour plus de détails sur la construction du harnais de test voir [1]. De plus, FoCalTest peut être utilisé pour une phase de test de non régression. En effet il est possible de rejouer des jeux de test obtenus lors d'une phase de test antérieure en donnant en argument un rapport de test, i.e. un fichier XML qui décrit les différents jeux de test.

3 Contrôle d'accès multi-niveaux

Un des aspects de la sécurité en informatique concerne le contrôle des accès aux données d'un système pour lequel différentes politiques de sécurité peuvent être mises en application. Ce contrôle nécessite de développer des mécanismes permettant de filtrer les accès afin de ne laisser passer que ceux autorisés, et donc de définir une politique de sécurité, c'est-à-dire la caractérisation des accès permis. Le programme chargé de mettre en application cette politique, le moniteur de référence, est souvent considéré comme l'une des clés de voûte de la sécurité d'un système. Sa conception et son développement doivent être menés de manière à garantir sa fiabilité et sa sûreté. En effet, toute faille au sein de ce programme pourrait entraîner des violations de la politique de sécurité. L'emploi des méthodes formelles permet de garantir qu'un moniteur de référence chargé du contrôle des accès dans un système maintient une politique de contrôle d'accès donnée. Cette propriété est bien sûr cruciale pour la plupart des systèmes d'information.

Politique de confidentialité

Dans cet article, nous considérons le développement formel en FoCaL d'un modèle de contrôle d'accès multi-niveaux permettant de garantir la propriété de confidentialité des informations au sein d'un système. Ce développement est effectué à plusieurs niveaux de spécification. Tout d'abord il décrit comment est représenté le système d'information considéré, c-a-d quels sont les paramètres de sécurité mis en jeu et quelles sont les informations qui décrivent l'état du système. Les paramètres de ce développement sont un ensemble $\mathcal{S}$ de sujets, un ensemble $\mathcal{O}$ d'objets, un ensemble $\mathcal{A}$ de modes d'accès et un treillis fini $(\mathcal{L}, \preceq, \curlyvee, \curlywedge)$ de niveaux de sécurité. Un état du système est alors décrit par un triplet (m, f_s, f_o) où m est l'ensemble des accès courants effectués simultanément dans le système et $f_s : \mathcal{S} \rightarrow \mathcal{L}$ (resp. $f_o : \mathcal{O} \rightarrow \mathcal{L}$) est une fonction associant un niveau de sécurité aux sujets (resp. aux objets). Les éléments de m sont des accès représentés par des triplets de la forme (s, o, a) exprimant qu'un sujet s accède à un objet o selon le

mode d'accès a. Nous considérons dans la suite l'ensemble $\mathcal{A} = \{\mathsf{read}, \mathsf{write}\}$ et nous notons Σ l'ensemble des états.

La politique de contrôle d'accès est alors spécifiée : elle permet de caractériser le sous-ensemble des états du système qui sont sûrs, c-a-d qui respectent la politique de sécurité. Dans le cadre du contrôle d'accès multi-niveaux, la confidentialité peut s'exprimer par les deux propriétés suivantes.

(i) La propriété MAC connue sous le nom de « *no read-up property* » exprime qu'un sujet ne peut accéder en lecture à un objet que si son niveau de sécurité est supérieur à celui de l'objet accédé :

$$\mathsf{MAC}((m, f_s, f_o)) \Leftrightarrow \forall s \in \mathcal{S}\ \forall o \in \mathcal{O} \quad (s, o, \mathsf{read}) \in m \Rightarrow f_s(s) \succeq f_o(o) \quad (1)$$

(ii) La propriété MAC⋆ connue sous le nom de « *no write-down property* » permet d'éviter qu'un sujet « malicieux » recopie de l'information sensible à un niveau de sécurité inférieur :

$$\begin{array}{ll} & \mathsf{MAC}^\star((m, f_s, f_o)) \\ \Leftrightarrow & \forall s \in \mathcal{S}\ \forall o_1, o_2 \in \mathcal{O} \\ & ((s, o_1, \mathsf{read}) \in m \wedge (s, o_2, \mathsf{write}) \in m) \Rightarrow f_o(o_1) \preceq f_o(o_2) \end{array} \quad (2)$$

On dira qu'un état est sûr lorsqu'il satisfait ces deux propriétés.

Ensuite, l'ensemble $\mathcal{R}$ des requêtes permettant de modifier l'état du système est défini. Ces requêtes permettent à un utilisateur de mettre à jour les informations du système ou d'accéder aux objets. Nous considérons ici les deux requêtes :

- $\langle +, s, o, a \rangle$: le sujet s demande à accéder à l'objet o selon le mode a,
- $\langle -, s, o, a \rangle$: le sujet s demande à relâcher (supprimer) son accès sur l'objet o selon le mode a.

Ces deux notions (politique et langage de requêtes) permettent de définir la notion de modèle de contrôle d'accès. Il reste alors à définir un moniteur de référence implantant le modèle : il s'agit de la donnée d'un ensemble d'états initiaux et d'une fonction de transition $\tau : \mathcal{R} \times \Sigma \rightarrow \{\mathsf{yes}, \mathsf{no}\} \times \Sigma$ entre états dont la définition est donnée dans la table 1.

Bien sûr, la fonction de transition considérée doit satisfaire une propriété de correction vis-à-vis de la politique considérée, qui exprime que cette fonction ne produit que des états sûrs lorsqu'elle est appliquée à un état sûr :

$$\begin{array}{l} \forall \sigma \in \Sigma\ \forall R \in \mathcal{R} \\ (\mathsf{MAC}(\sigma) \wedge \mathsf{MAC}^\star(\sigma)) \Rightarrow (\mathsf{MAC}(\pi_2(\tau(R, \sigma))) \wedge \mathsf{MAC}^\star(\pi_2(\tau(R, \sigma)))) \end{array} \quad (3)$$

où π_2 est un opérateur de projection permettant d'obtenir la deuxième composante d'une paire.

Politique de flots

Une politique de contrôle d'accès multi-niveaux permet de spécifier quels sont les accès autorisés lorsque le système se trouve dans un certain état mais

$$\tau(R,(m,f_s,f_o)) = \begin{cases} (\mathsf{yes},(m \cup \{(s,o,\mathsf{read})\},f_s,f_o)) \\ \quad \text{if} \quad R = \langle +,s,o,\mathsf{read}\rangle \\ \quad \wedge \ f_o(o) \preceq f_s(s) \\ \quad \wedge \ \left\{ o' \in \mathcal{O} \mid \ (s,o',\mathsf{write}) \in m \quad \wedge \neg(f_o(o) \preceq f_o(o')) \right\} = \emptyset \\ (\mathsf{yes},(m \cup \{(s,o,\mathsf{write})\},f_s,f_o)) \\ \quad \text{if} \quad R = \langle +,s,o,\mathsf{write}\rangle \\ \quad \wedge \ \left\{ o' \in \mathcal{O} \mid \ (s,o',\mathsf{read}) \in m \quad \wedge \neg(f_o(o') \preceq f_o(o)) \right\} = \emptyset \\ (\mathsf{yes},(m \setminus \{(s,o,a)\},f_s,f_o)) \quad \text{if} \quad R = \langle -,s,o,a\rangle \\ (\mathsf{no},(m,f_s,f_o)) \quad \text{otherwise} \end{cases}$$

TAB. 1 – Fonction de transition τ

permet aussi de spécifier les flots d'information qui sont autorisés durant la vie du système. Dans le cas d'une politique de confidentialité, la propriété requise sur les flots exprime que l'information contenue dans un objet o_1 ne peut migrer dans un objet o_2 que si le niveau de o_1 est inférieur au niveau de o_2. Il n'y a ainsi pas de recopie d'informations sensibles dans des objets de bas niveaux. Pour exprimer formellement cette propriété, nous définissons tout d'abord les flots engendrés par les accès courants d'un état. Le contenu d'un objet o_1 est diffusé dans un objet o_2, ce que l'on note $o_1 \hookrightarrow^{OO} o_2$ s'il existe un ensemble de sujets dont les accès sur les objets du système permettent de recopier l'information de o_1 dans o_2, cette information pouvant transiter par des objets intermédiaires. Ainsi, la relation de flots engendrée par l'ensemble m des accès courants d'un état σ est définie par :

$$\hookrightarrow_{\sigma}^{OO} = \left\{ o_1 \hookrightarrow^{OO} o_2 \mid \left(\begin{array}{l} \exists s_1, \cdots, s_k, s_{k+1} \in \mathcal{S} \ \exists o^1, \cdots, o^k \in \mathcal{O} \\ \left\{ \begin{array}{l} (s_1, o_1, \mathsf{read}), (s_1, o^1, \mathsf{write}), \\ (s_2, o^1, \mathsf{read}), (s_2, o^2, \mathsf{write}), \\ \cdots, \\ (s_i, o^{i-1}, \mathsf{read}), (s_i, o^i, \mathsf{write}), \\ \cdots, \\ (s_{k+1}, o^k, \mathsf{read}), (s_{k+1}, o_2, \mathsf{write}) \end{array} \right\} \subseteq m \\ \vee o_1 = o_2 \end{array} \right) \right\}$$

Dans le cadre de la politique de confidentialité considérée ici, la propriété requise sur les flots peut maintenant s'écrire formellement comme suit :

$$\begin{array}{l} \forall \sigma = (m, f_s, f_o) \in \Sigma \ \forall o_1, o_2 \in \mathcal{O} \\ (\mathsf{MAC}(\sigma) \wedge \mathsf{MAC}^\star(\sigma) \wedge o_1 \hookrightarrow_{\sigma}^{OO} o_2) \Rightarrow f_o(o_1) \preceq f_o(o_2) \end{array} \tag{4}$$

4 L'erreur est humaine ...

Conduire un développement formel est une activité qui nécessite une grande rigueur et qui, comme toute activité humaine, est sujette à l'erreur. Dans cette section, nous introduisons des erreurs (sur la spécification et sur le code) dans la formalisation de la politique présentée ci-dessus. C'est sur ces exemples de développements formels erronés que nous mettrons en œuvre l'outil de test FoCalTest afin d'en illustrer l'utilisation.

Erreurs dans la spécification

En 1988, J. McLean [8] introduit une « algèbre des modèles de sécurité » permettant d'exprimer des politiques de contrôle d'accès multi-niveaux et illustre le cadre qu'il propose en considérant une politique de confidentialité. J. McLean introduit également la notion d'accès conjoints : certaines opérations ne peuvent être autorisées que si elles sont demandées par un certain groupe de sujets. Dans le cas le plus général, les ensembles de sujets qui pourront conjointement soumettre une requête sont des parties non vides quelconques de $\mathcal{S}$ (toute opération est initiée par au moins un sujet). Sans accès conjoints, les seuls ensembles autorisés à soumettre une requête sont les singletons construits à partir de $\mathcal{S}$. La propriété de sécurité $\mathsf{MAC}^\star$ définie en (2) est alors (re)définie comme suit :

> [8] *a state is $\star$-secure if for any subjects S_1, S_2 and objects o_1, o_2, if $(S_1, o_1, \mathsf{read}) \in m$ and $(S_2, o_2, \mathsf{write}) \in m$ and the classification of o_1 dominates that of o_2, then $S_1 \cap S_2 = \emptyset$*

Ici, m est l'ensemble des accès courants, S_1 et S_2 sont des ensembles de sujets et un accès est un triplet (S, o, a) exprimant que les sujets présents dans l'ensemble S accèdent conjointement à l'objet o selon le mode a. La formalisation de cet énoncé permet d'obtenir, par contraposition, la spécification suivante :

$$((S_1, o_1, \mathsf{read}) \in m \wedge (S_2, o_2, \mathsf{write}) \in m \wedge S_1 \cap S_2 \neq \emptyset) \Rightarrow \neg(f_o(o_2) \prec f_o(o_1))$$

Or, si on se limite au cas où S_1 et S_2 sont réduits à des singletons, c'est-à-dire si on ne considère pas les accès conjoints, on obtient finalement :

$$\begin{array}{l} \forall s \in \mathcal{S}\ \forall o_1, o_2 \in \mathcal{O} \\ ((s, o_1, \mathsf{read}) \in m \wedge (s, o_2, \mathsf{write}) \in m) \Rightarrow \neg(f_o(o_2) \preceq f_o(o_1)) \end{array} \tag{5}$$

les deux propriétés (2) et (5) ne sont pas équivalentes. En effet, pour que ces deux propriétés soient équivalentes il faudrait que l'ordre sur les niveaux de sécurité soit total. Or, il ne s'agit que d'un ordre partiel puisque l'ensemble des niveaux de sécurité n'est muni que d'une structure de treillis. La propriété (5) n'exprime malheureusement pas la propriété souhaitée sur les flots et nous verrons dans la prochaine section que l'outil de test FoCalTest

$$
\begin{aligned}
&\tau_{\mathsf{f}}(R, (m, f_s, f_o)) \\
= &\begin{cases}
(\mathsf{yes}, (m \cup \{(s, o, \mathsf{read})\}, f_s, f_o)) \\
\quad \text{si} \quad R = \langle +, s, o, \mathsf{read} \rangle \\
\quad \wedge \ \boxed{f_s(s) \preceq f_o(o)}^{\,1} \\
\quad \wedge \ \left\{ o' \in \mathcal{O} \mid \ (s, o', \mathsf{write}) \in m \ \ \wedge \boxed{\neg(f_o(o') \preceq f_o(o))}^{\,2} \right\} = \emptyset \\
(\mathsf{yes}, (m \cup \{(s, o, \mathsf{write})\}, f_s, f_o)) \\
\quad \text{if} \quad R = \langle +, s, o, \mathsf{write} \rangle \\
\quad \wedge \ \left\{ o' \in \mathcal{O} \mid \ (s, o', \mathsf{read}) \in m \ \ \wedge \boxed{\neg(f_o(o) \preceq f_o(o'))}^{\,3} \right\} = \emptyset \\
(\mathsf{yes}, (m \setminus \{(s, o, a)\}, f_s, f_o)) \quad \text{if} \quad R = \langle -, s, o, a \rangle \\
(\mathsf{no}, (m, f_s, f_o)) \quad \text{otherwise}
\end{cases}
\end{aligned}
$$

TAB. 2 – Fonction de transition erronée

engendrera des jeux de test mettant en évidence des flots d'information non autorisés par la politique de confidentialité.

Erreurs dans l'implantation

Nous introduisons également trois versions erronées de la fonction de transition afin de pouvoir illustrer l'utilisation de l'outil FoCalTest sur du code incorrect vis-à-vis d'une spécification correcte. Le but est d'évaluer la capacité de FoCalTest à générer des jeux de test qui détectent de telles erreurs. Ces trois erreurs sont données dans la table 2 dans la même fonction bien qu'elles soient implantées dans trois fonctions différentes afin de ne tester qu'une erreur à la fois (les erreurs introduites sont encadrées et numérotées, et peuvent correspondre à des erreurs d'inattention lors du développement).

5 Mise en œuvre avec FoCalTest

Nous montrons à présent la mise en œuvre de la détection automatique des erreurs décrites dans la section 4. Pour cela, nous allons d'abord présenter la propriété (4) qui utilise la spécification de $\mathsf{MAC}^\star$ proposé par J. McLean. Cette propriété sera ensuite testée à l'aide de FoCalTest. Une version modifiée de la propriété (4), pour laquelle FoCalTest trouve des jeux de test valides de manière plus efficace, sera ensuite présentée. Enfin, nous nous intéresserons aux tests des fonctions de transition erronées présentées dans la table 2.

Fixons tout d'abord l'implantation de référence utilisée dans le cadre de nos tests, et tout particulièrement le treillis de sécurité. Il s'agit de celui décrit dans la figure 2. Nous avons donc 3 objets et 3 sujets associés à chacun

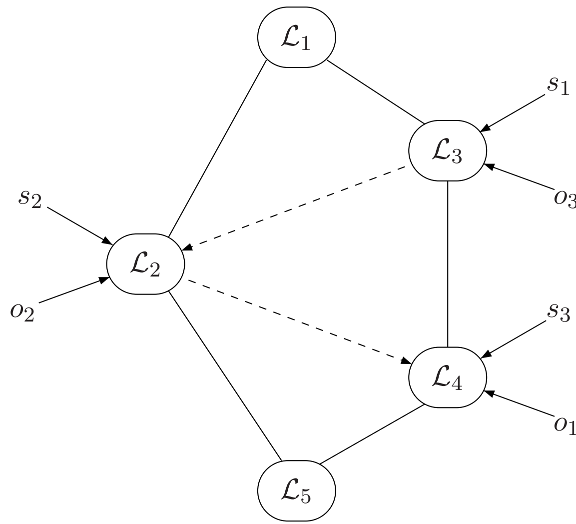

Fig. 2 – Treillis pris en exemple

des trois niveaux de sécurité $\mathcal{L}_1$, $\mathcal{L}_2$ et $\mathcal{L}_3$. Dans ce treillis, par exemple le sujet s_2 et l'objet o_2 sont associés au niveau de sécurité $\mathcal{L}_2$. Ce treillis est celui qui est utilisé dans tous les exemples de cette section. Les flèches en pointillés montre l'unique flot d'information descendant permis dans la spécification de J. McLean.

FoCalTest permet de tester des propriétés en forme prénexe. Dans les exemples qui suivent, les propriétés testées utilisent les prédicats MAC et $\mathsf{MAC}^\star$ ainsi que la relation de flot $\hookrightarrow_\sigma^{OO}$. Chacune des ces trois composantes contient des quantificateurs universels et existentiels. Nous ne pouvons pas utiliser directement la définition des ces propriétés, en effet leur utilisation pour former la propriété (4) donne lieu à une formule qui n'est pas en forme prenexe. Pour pouvoir utiliser l'outil FoCalTest, il a fallu transformer cette propriété de manière à faire disparaître les quantificateurs. Pour cela, nous remarquons que les variables quantifiées sont soit du type des objets, soit du type des sujets. Ces deux types étant finis (il n'y a que 3 objets et 3 sujet dans notre système), nous pouvons aisément remplacer MAC et $\mathsf{MAC}^\star$ par des fonctions qui vérifient ces propriétés sur l'ensemble des valeurs que peuvent prendre les variables quantifiées. Dans la suite, nous avons appliqué cette transformation sur les propriétés MAC et $\mathsf{MAC}^\star$ qui sont ainsi implantées par les fonctions `mac_fun` et `mac_star_fun`. De même, la relation $\hookrightarrow_\sigma^{OO}$ est implantée par la fonction `flow`. Cette transformation a été effectuée à la main dans le cadre de cet article. Elle peut néanmoins être réalisée automatiquement par FoCalTest à l'aide, par exemple, de combinateurs correspondant aux deux sortes de quantificateurs.

Test de la spécification

Nous testons ici deux propriétés : la première correspond à l'énoncé de (4) de la page 8 modifié comme expliqué précédemment ; la deuxième propriété est une version modifiée, quoique équivalente, de la première propriété. L'écriture de cette deuxième propriété permet à FoCalTest de générer des

jeux de test plus efficacement. La première propriété (6) énonce le fait que si un état σ respecte la politique et qu'il existe un flot d'information entre deux objets o_1 et o_2, alors le niveau de sécurité associé à l'objet o_1 doit être inférieur au niveau de sécurité associé à l'objet o_2.

$$\begin{array}{l}\forall \sigma \in \Sigma \quad \forall o_1, o_2 \in \mathcal{O} \\ (\texttt{mac_fun}(\sigma) \wedge \texttt{mac_star_fun}(\sigma) \wedge \texttt{flow}(\sigma, o_1, o_2)) \Rightarrow f_o(o_1) \preceq f_o(o_2)\end{array} \tag{6}$$

Nous avons demandé à FoCalTest de générer 1000 jeux de test valides (i.e. qui vérifie la précondition). Ceci a été répété 5 fois de suite de manière indépendante. Un résumé du rapport de test résultant est présenté dans la table 3. La deuxième colonne du tableau donne le nombre de jeux de test générés pour obtenir les 1 000 jeux de test voulus, la quatrième colonne donne le nombre de jeux de test valides qui ont invalidé la conclusion ; il s'agit donc du nombre de contre-exemples trouvés. Enfin, la dernière colonne donne le nombre de jeux de test qui exhibent un flot descendant. Nous avons

N° de soumission	*Nb de JT générés*	*Nb de JT valides*	*Nb de contre-exemples*	*Nb de flots descendants*
1	588 461	1 000	720	10
2	586 898	1 000	704	7
3	586 901	1 000	732	9
4	563 220	1 000	720	5
5	592 745	1 000	717	10

Tab. 3 – Rapport de test de la propriété (6)

lancé FoCalTest sur un ordinateur doté d'un processeur Intel Core 2 Duo cadencé à 2,33GHz. Dans chacune des 5 exécutions de FoCalTest, les jeux de test ont été trouvés en une vingtaine de secondes environ. On remarque que 70% des jeux de test valides sont des contre-exemples à la propriété de J. McLean et que FoCalTest a été capable de trouver le flot descendant (celui de la figure 2) dans chacune des 5 expériences. Néanmoins, on remarque que le taux de jeux de test valides générés est faible, il est de 1 jeu de test valide pour 587 jeux de test rejetés.

Dans un deuxième temps, nous avons envisagé une variante de la propriété précédente :

$$\forall \sigma \in \Sigma \; (\texttt{mac_fun}(\sigma) \wedge \texttt{mac_star_fun}(\sigma)) \Rightarrow \texttt{all_flows_correct}(\sigma) \tag{7}$$

Cette propriété énonce le fait que si un état σ respecte la politique (toujours celle correspondant à la propriété (5)), alors tous les flots d'information de l'état σ doivent être corrects (fonction `all_flow_correct`), c'est-à-dire ils ne doivent pas être descendants. La fonction `all_flow_correct` génère de manière combinatoire tous les flots et teste leur correction. Ainsi, la seule variable quantifiée est l'état du système et la précondition est réduite à la simple vérification de MAC et MAC*. Le tableau 4 résume le rapport de

test de cette propriété. De même que pour la propriété précédente, nous avons demandé à FoCalTest de générer 5 fois 1 000 jeux de test. Les colonnes du tableau ont la même signification que les colonnes du tableau 3. Nous remarquons que pour cette propriété, le nombre de jeux de test re-

N° de soumission	*Nb de JT générés*	*Nb de JT valides*	*Nb de contre-exemples*	*Nb de flots descendants*
1	1 281	1 000	8	0
2	1 273	1 000	12	0
3	1 282	1 000	11	1
4	1 264	1 000	12	1
5	1 309	1 000	8	0

TAB. 4 – Rapport de test de la propriété (7)

jetés est bien inférieur à celui de l'exemple précédent, le ratio est ici de 1,3. En revanche, bien que FoCalTest trouve toujours des contre-exemples à la propriété, leur nombre est bien inférieur à celui obtenu lors du test de la propriété précédente. De plus, FoCalTest n'a pas été en mesure de trouver le flot d'information descendant dans chacun des cas. En effet, cela s'explique par le fait que pour la propriété (6), nous imposons que l'état testé contienne au moins un flot d'information entre deux objets alors que pour la propriété (7), cela n'est pas imposé. Ainsi, dans la pratique, un grand nombre de jeux de test de la propriété (7) porte sur l'état vide qui ne contient évidemment pas de flot d'information descendant.

Ces deux exemples montrent que réécrire une propriété afin de rendre sa précondition plus faible et donc mieux adaptée au test aléatoire permet effectivement d'améliorer le ratio du nombre de jeux de test validés sur le nombre de jeux de test rejetés mais réduit aussi la probabilité de trouver des erreurs.

Test de l'implantation

La propriété sous test est présentée en (8) ci-dessous. Cette propriété stipule que la fonction de transition appliquée à un état sûr et à n'importe quel type d'accès renvoie un état sûr. Cette propriété est la transcription pour FoCalTest de la propriété (3) présentée en page 7.

$$\forall \sigma \in \Sigma\, \forall R \in \mathcal{R} \left(\begin{array}{l} (\texttt{mac_fun}(\sigma) \wedge \texttt{mac_star_fun_correct}(\sigma)) \Rightarrow \\ \left(\begin{array}{l} \quad \texttt{mac_fun}(\pi_2(\tau_{\mathsf{blp}}(R, \sigma))) \\ \wedge\, \texttt{mac_star_fun_correct}(\pi_2(\tau_{\mathsf{blp}}(R, \sigma))) \end{array} \right) \end{array} \right) \quad (8)$$

Dans cette section nous nous intéressons à une application de FoCalTest au test mutationnel. Nous rappelons que le test mutationnel permet d'évaluer la capacité d'un jeu de test à trouver des erreurs dans une implantation en présence d'un certain type d'erreur. Un programme mutant est un programme dans lequel on a volontairement ajouté une erreur. On créé ainsi

N^o de mutant	*Nb de contre-exemples*
1	12
2	20
3	392

Tab. 5 – Rapport de test pour les mutants pour 10 000 cas de test

un certain nombre de mutants, on soumet ensuite un même jeu de test au programme d'origine et aux mutants. Si un mutant donne le même résultat que le programme initial, on dit que le mutant passe le jeu de test, si au contraire un mutant réagit différemment, le mutant est tué. Le ratio nombre de mutants tués/nombres de mutants donne le score de mutation qui doit être le plus élevé possible.

Dans la suite, nous avons généré un ensemble de jeux de test témoin aléatoirement en se basant sur l'implantation sans erreur et composé de 10 000 jeux de test valides. Nous avons ensuite créé 3 mutants en injectant des fautes dans la fonction de transition τ, les erreurs injectées sont décrites dans le tableau 2. Il s'agit d'erreur classique, l'inversion des éléments dans une comparaison. Nous avons ensuite utilisé la capacité de FoCalTest de réinjecter un ensemble de jeux de test à partir d'un rapport de test pour soumettre les jeux de test témoins. Les résultats sont présentés dans la table 5. Celle-ci montre que chaque mutant a été tué au moins une fois. Par exemple, le mutant numéro 3 a été tué pour 392 jeux de test pour 10 000 présentés.

6 Conclusion

Dans cet article nous avons illustré l'utilisation de l'outil FoCalTest dans le cadre du développement formel d'un modèle de contrôle d'accès. Nous avons dans un premier temps modélisé les notions d'état sûr et de flot descendant. La premier modèle testé à l'aide de FoCalTest reproduisait une erreur de spécification introduite par J. McLean dans un article ancien. L'outil de test a en effet permis d'identifier l'erreur et d'exhiber le flot interdit. Dans un deuxième temps, nous avons introduit des erreurs dans le code, plus précisément dans la fonction de transition du modèle, produisant ainsi des mutants. Là encore, l'outil nous a permis de détecter les erreurs.

L'outil de test utilisé se fonde sur une technique de génération aléatoire des jeux de test. Ceux-ci doivent de plus vérifier une contrainte, la précondition de la propriété sous test. Si la contrainte est très forte, il est très probable qu'il faille générer de nombreux jeux de test avant d'en trouver un qui respecte la précondition. Dans l'article nous avons montré que l'écriture même de la propriété influait sur les résultats de la campagne de test. Néanmoins cette génération aléatoire est, même dans notre cas très contraint, fructueuse. Un travail en cours par M. Carlier et C. Dubois

concerne la génération de jeux de test en utilisant une approche orientée contraintes, permettant de faire entrer la précondition dans la construction du jeu de test elle-même. Le texte des méthodes référencées dans la précondition de la propriété sous test est exploité pour construire l'ensemble des contraintes qui définit le jeu de test. On passe alors à une approche de type *boîte blanche*.

Remerciements. Nous remercions T. Hardin, C. Morisset et F. Pessaux pour leur aide précieuse ainsi que les rapporteurs pour leurs remarques constructives. Ce travail est financé par l'action SSURF, ANR-06-SETI-016.

Références

[1] M. Carlier and C. Dubois. Functional testing in the focal environment. In *Test And Proof*, volume 4966 of *LNCS*, pages 84–98. Springer, 2008.

[2] A. Santana de Oliveira. *Réécriture et Modularité pour les Politiques de Sécurité*. Phd thesis, Université Henri Poincaré, 2008.

[3] D. Delahaye, J.F. Etienne, and V. Donzeau-Gouge. Certifying airport security regulations using the Focal environment. In *FM 2006 : Formal Methods, 14th International Symposium on Formal Methods, Proceedings*, volume 4085 of *LNCS*, pages 48–63. Springer, 2006.

[4] C. Dubois, T. Hardin, and V. Viguié Donzeau-Gouge. Building certified components within focal. In *Revised Selected Papers from the Fifth Symposium on Trends in Functional Programming, TFP 2004*, volume 5 of *Trends in Functional Programming*, pages 33–48. Intellect, 2006.

[5] D.P. Guelev, M.Ryan, and P.Y. Schobbens. Model-checking access control policies. In *Information Security, 7th International Conference, ISC 2004*, volume 3225 of *LNCS*, pages 219–230. Springer, 2004.

[6] M. Jaume and C. Morisset. A formal approach to implement access control. *J. of Information Assurance and Security*, 2 :137–148, 2006.

[7] E. Martin and T. Xie. A fault model and mutation testing of access control policies. In *WWW '07, 16th international conference on World Wide Web*, pages 667–676. ACM, 2007.

[8] J. McLean. The algebra of security. In *Proc. IEEE Symposium on Security and Privacy*, pages 2–7. IEEE Computer Society Press, 1988.

[9] C. Morisset. *Sémantique des systèmes de contrôle d'accès*. PhD thesis, Université Pierre et Marie Curie, Paris, France, 2007.

[10] C. Morisset and A. Santana de Oliveira. Automated detection of information leakage in access control. In *2nd International Workshop on Security and Rewriting Techniques (SecReT'07)*, 2007.

[11] T. Mouelhi, Y. Le Traon, and B. Baudry. Testing security policies : going beyond functional testing. In *ISSRE'07 (Int. Symposium on Software Reliability Engineering)*, 2007.