Test fonctionnel de conformité vis-à-vis d'une politique de contrôle d'accès*

Frédéric Dadeau, Amal Haddad, and Thierry Moutet

Laboratoire d'Informatique de Grenoble (LIG)
UMR 5217
681, rue de la Passerelle
BP. 72
38402 Saint-Martin d'Hères cedex
FRANCE

{Frederic.Dadeau,Amal.Haddad,Thierry.Moutet}@imag.fr

Résumé Les travaux présentés dans cet article s'articulent autour de la validation du contrôle d'accès défini par des politiques de sécurité. Nous nous intéressons à la validation par génération de tests à partir d'un modèle fonctionnel écrit en B. Nous utilisons l'outil Meca, qui prend en entrée un modèle fonctionnel et une description d'une politique de sécurité sous la forme de machines abstraites B, et qui génère un noyau de sécurité pour le modèle fonctionnel. Ce noyau de sécurité est en charge d'intercepter tous les accès des sujets aux objets, et restreint les comportements à ceux satisfaisant les exigences de sécurité.

Nous proposons une approche définissant des objectifs de tests de conformité vis-à-vis d'une politique de contrôle d'accès. L'objectif est de s'assurer que les appels à des opérations dans un contexte licite (resp. illicite) sont bien autorisés (resp. sont bien bloqués) sur l'implantation sous test. Nous présentons une étude de cas complète, sur une application de type porte-monnaie électronique, modélisée en B et implantée en Java, et agrémentée d'une politique de sécurité discrétionnaire.

Mots-clés : Contrôle d'accès, test de conformité, Meca, politique de sécurité, noyau de sécurité

1 Introduction

Les systèmes actuels sont de plus en plus demandeurs en terme de sécurité. Par sécurité, on entend généralement cinq propriétés essentielles : l'*authentification*, qui permet de s'assurer de l'identité d'une entité donnée ou de l'origine d'une communication ou d'un fichier, la *confidentialité* qui empêche la divulgation d'une information tenue secrète, l'*intégrité*, qui vise à garantir qu'une information ne sera pas modifiée sans en avoir eu l'autorisation, la *disponibilité*, qui garantit l'accès aux informations pour les utilisateurs autorisés, et la *non-répudiation* qui empêche le reniement d'action ou de messages anciens.

* Ce travail a été en partie financé par le projet RNTL POSE (ANR-05-RNTL-01001)

Dans l'objectif d'évaluer la sécurité, les *Critères Communs* (CC) ont récemment été introduits. Il s'agit d'une norme (ISO 15408) qui valide la sécurité d'un système du point de vue logiciel et matériel. Les critères communs définissent une cible de sécurité (TOE – Target Of Evaluation) qui doit présenter un certain nombre d'exigences se répartissant en deux catégories : les exigences fonctionnelles de sécurité et les exigences d'assurance de sécurité. Les premières évaluent les mécanismes déployés pour garantir la sécurité, tandis que les secondes visent à garantir la bonne mise en œuvre de ces mécanismes et leur adéquation à la cible de sécurité. Nous nous intéressons plus particulièrement au cas du contrôle d'accès à des informations sensibles, qui est une sous-classe de la classe "Protection des données de l'utilisateur" (FDP – Functional Data Protection), appartenant à la catégorie des exigences fonctionnelles de sécurité. Le contrôle d'accès vise à s'assurer que des *sujets* ont ou non le droit d'accéder à des *objets*. Les droits ou les interdictions sont définis à l'aide de règles dépendantes d'attributs de sécurité.

Un système de contrôle d'accès doit intercepter toutes les tentatives d'accès non autorisés. La définition d'un tel système se base sur les trois éléments suivants. (*i*) Les politiques de sécurité, qui décrivent des règles de haut niveau de contrôle d'accès et décident lesquels sont autorisés ou interdits. (*ii*) Les modèles de sécurité, qui sont des formalisations des politiques de sécurité et de leur fonctionnement. Ils sont utilisés pour prouver des propriétés de sécurité dans les systèmes. Enfin, (*iii*) les mécanismes de sécurité définissent des fonctions de bas niveau (locigielles et matérielles) permettant d'implanter les contrôles imposés par la politique de sécurité.

On distingue différents types de politiques de contrôles d'accès. Les politiques discrétionnaires (Discretionary Access Control – DAC) [9] accordent au possesseur d'une information (généralement son créateur) tous les droits d'accès, ainsi que la possibilité de passer ces droits à d'autres utilisateurs à leur discrétion. Les politiques obligatoires (Mandatory Access Control – MAC) [3,4] décrètent des règles incontournables qui régissent les droits des sujets et des objets. Enfin, les politiques basées sur les rôles (Role-Based Access Control – RBAC) [6] permettent d'accorder des droits d'accès à des sujets en fonction des rôles qu'ils jouent dans une organisation.

Un travail préliminaire [7] a présenté l'outil Meca, qui considère un modèle fonctionnel d'un système, exprimé en B, et une politique de sécurité décrite par une machine abstraite B, et qui génère un noyau de sécurité interceptant les accès illicites des sujets aux objets. Les politiques de contrôle d'accès acceptées par Meca peuvent être de n'importe lequel des trois types précédents (DAC, MAC ou RBAC). Nous souhaitons coupler ce mécanisme avec un moyen de générer automatiquement des tests de conformité, de manière à s'assurer que la sécurité liée au système est correctement implantée. Notre objectif est donc de produire des tests guidés par ces principes de sécurité. Pour ce faire, nous considérons un modèle de sécurité dissocié du modèle fonctionnel du système. Nous nous appuyons sur les résultats produits par l'analyse de modèle dans Meca pour produire ces tests. Nous illustrons notre approche sur une étude de

cas complète, partant d'une spécification d'un porte-monnaie électronique, pour lequel on souhaite sécuriser l'accès, jusqu'à son implantation en Java.

Cet article s'organise de la manière suivante. Nous présentons en partie 2 l'approche que nous avons mise en place. Après un rapide survol des principales fonctionnalités de la méthode B en partie 3, nous introduisons l'utilisation d'un modèle dédié à la sécurité en partie 4. Nous expliquons la contribution principale de ces travaux, la génération de tests basée sur ce modèle de sécurité, en partie 5. Une expérimentation sur une étude de cas est donnée dans la partie 6. Puis, nous discutons de la technique employée et des extensions ultérieures en partie 7. Enfin, nous présentons les conclusions et les perspectives de ces travaux en partie 8.

2 Présentation de l'approche

L'objectif des travaux présentés dans cet article est de définir un mécanisme de génération de jeux de tests dédiés à exercer le contrôle d'accès d'une application. Basiquement, le contrôle d'accès se caractérise par un ensemble de sujets qui sont autorisés, ou non, à manipuler des objets. Ainsi, les interactions entre sujets et objets sont décrites par l'intermédiaire de permissions ou d'interdictions.

Les tests que nous nous proposons de générer s'appuient sur ce mécanisme, en souhaitant explorer le plus grand nombre de possibilités concernant les conditions d'utilisation des objets. Nous souhaitons effectuer des tests de conformité du contrôle d'accès, ce qui sous-entend vérifier que l'implantation permet bien l'accès dans des conditions licites d'utilisation et refuse bien l'accès dans des conditions illicites. Nous adoptons une approche de test fonctionnel boîte-noire, basée sur un modèle [2]. Dans cette configuration, un modèle est utilisé pour produire les cas de tests et donner l'oracle. L'implantation est, quant à elle, développée à part, sans lien avec le modèle, si ce n'est l'existence de points de connexion à travers des API définies dans le cahier des charges.

Pour ce faire, nous disposons de l'outil Meca, qui permet, à partir d'un ensemble de règles données dans un modèle de sécurité et d'un modèle fonctionnel, de produire un noyau de sécurité qui contrôle les appels des différentes opérations pour n'autoriser que les exécutions des opérations qui sont licites, et détecter (et signaler) les appels aux opérations qui sont illicites. Le modèle fonctionnel sécurisé produit par Meca fournit l'oracle, qui consiste à vérifier la conformité entre la prédiction du modèle et la réalité de l'implantation : dans des conditions d'activation supposées similaires, si une implantation permet un accès, ce dernier est-il autorisé sur le modèle ? On notera qu'en terme de sécurité, l'implantation peut dans certains cas refuser des accès pourtant autorisés par le modèle de sécurité. En effet, il est possible que certains aspects purement fonctionnels aient été omis dans le modèle de sécurité et qu'une implantation soit plus restrictive dans ses cas d'utilisation (par exemple, des limitations introduites sur les domaines des variables).

Notre approche se résume par la Fig. 1. A partir d'une spécification informelle des besoins, nous considérons deux modèles, un modèle fonctionnel et un modèle

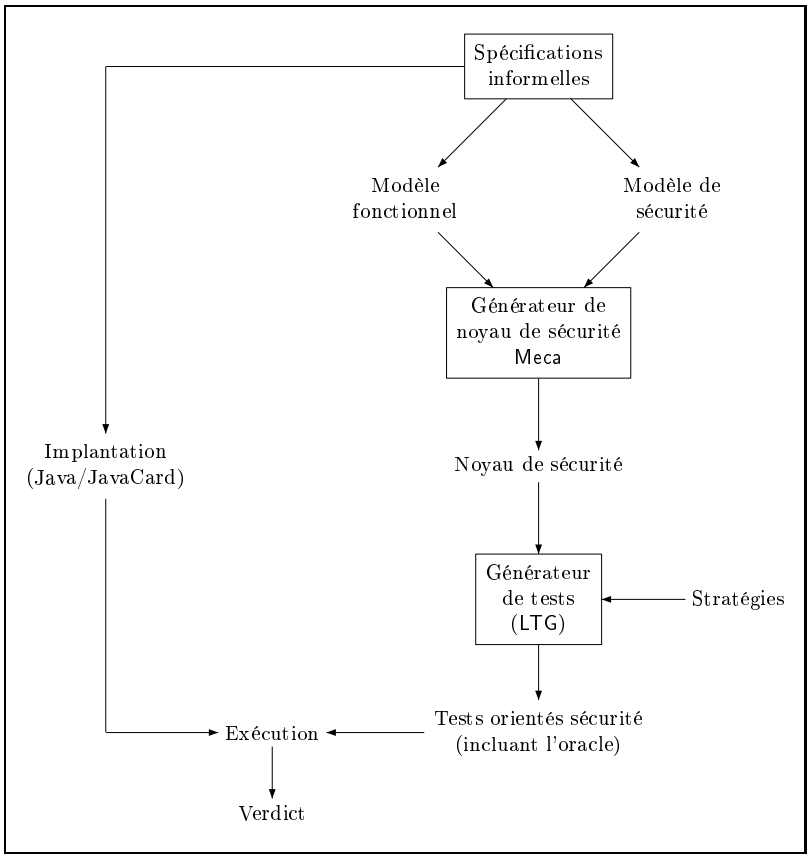

Fig. 1. Principe de l'approche proposée

de sécurité, tous deux écrits en B. Le modèle fonctionnel est plus ou moins détaillé, mais il doit introduire les éléments sur lesquels portent la politique de sécurité (attributs de sécurité, opération à sécuriser, etc.). Dans notre cas, les objets contrôlés sont les opérations du modèle B fonctionnel. En parallèle, nous concevons un autre modèle B, décrivant les règles d'accès des sujets aux objets contenus dans le modèle fonctionnel.

Ces deux modèles sont fournis en entrée de l'outil Meca qui produit un noyau de sécurité en charge de rajouter des vérifications, protégeant les accès des sujets aux objets en fonction de la politique d'accès choisie. Ceci nous produit une sur-couche du modèle fonctionnel, que nous désignons par "noyau de sécurité". Nous utilisons ensuite *LEIRIOS Test Generator* (LTG) [13,8], un générateur automatique de tests fonctionnels à partir d'un modèle B basé sur des critères de couverture du modèle. Nous verrons en partie 5 en quoi le modèle fonctionnel

sécurisé généré par Meca est adapté à la stratégie de génération de tests proposée par LTG. Ces tests sont ensuite concrétisés et exécutés sur une implantation Java. Les prédictions issues de l'exécution du modèle nous permettent d'établir le verdict du test.

Notons que l'implantation Java sur laquelle sont joués les tests a été developpée indépendamment du modèle fonctionnel et du modèle de sécurité. Elle représente ainsi la vision que le développeur a du cahier des charges. Le processus de test consiste à s'assurer qu'aucun élément inhérent à la sécurité de l'application n'a été oublié lors du développement. Pour évaluer notre approche, nous exerçons les tests générés sur des mutants, variantes de notre implantation. Ces variations sont guidées, dans le sens où elles affectent la vérification des conditions de sécurité dans l'implantation, simulant des erreurs classiques.

3 La méthode B

La *méthode B* [1] est une méthode formelle permettant le développement de logiciels corrects par construction. Cette caractéristique repose essentiellement sur des mécanismes permettant de vérifier par la preuve, à l'aide d'outils dédiés, qu'un système vérifie certaines propriétés exprimées avec des notations mathématiques. Elle permet en particulier d'écrire une spécification formelle sous la forme de *machines abstraites*. Celles-ci comportent des *données* (qui peuvent être exprimées par des types simples comme des entiers ou des booléens mais également par des ensembles ou des relations), des *propriétés invariantes* sur ces données et des *services* permettant d'initialiser et de faire évoluer ces données par l'intermédiaire de substitutions. Ces derniers sont représentés par des opérations.

A partir d'une machine abstraite, il est possible de continuer son développement par un travail de reformulation de la spécification et d'enrichissement du modèle initial. Ce processus de raffinemement est un mécanisme intégré de la méthode B. Son objectif est de garantir qu'à chaque étape du raffinement, on produit un modèle conforme au modèle réalisé à l'étape précédente. Cette validité est assurée aux moyens de vérifications statiques et d'obligation de preuves. Le dernier niveau de raffinement correspond à l'implantation. A cette étape, toutes les données ou substitutions manipulées doivent avoir un équivalent informatique. Il est ainsi possible de faire une traduction du modèle formel dans un langage exécutable comme Ada ou C. Cette traduction peut même être automatisée au moyen d'outils spécialisés comme l'*AtelierB*.

Dans le cadre de notre expérimentation, nous avons utilisé le langage B pour modéliser une application bancaire de type porte-monnaie électronique. La variable *balance* correspond au montant disponible sur la carte. Un *code pin utilisateur* permet de sécuriser la lecture, le crédit et le débit sur la carte. Un *code pin banque* permet de réinitialiser le *code pin utilisateur* en cas de blocage de celui-ci. Enfin, une variable *mode* représente le cycle de vie de la carte. Cette variable peut avoir trois états. En mode personnalisation (`PERSO`), les seuls traitements

```
MACHINE
    epurse
SETS
    MODE={PERSO,USE,INVALID}
DEFINITIONS
    BPC == 0..9999
VARIABLES
    bpc,bptry,isBankAuth,      /* code pin banque, nombre d'essai */
    mode,isOpenSess,...        /* mode et ouverture de session */
INVARIANT
    bpc ∈ BPC ∧ bptry ∈ 0..3 ∧ mode ∈ MODE ∧ ...
OPERATIONS
    setBpc(pin) =
       PRE
           pin ∈ BPC ∧          /* typage du pin en paramètre */
           isOpenSess = btrue ∧ /* la session doit être ouverte */
           mode = PERSO         /* bpc est modifiable en mode PERSO */
       THEN
           bpc := pin ||        /* mise à jour code pin banque */
           bptry := 3           /* compteur d'essai initialisé à 3 */
       END
    ...
END
```

Fig. 2. Schéma du modèle fonctionnel de l'application pour l'opération setBpc

autorisés correspondent à l'installation des codes pin. Une fois la configuration des codes pin effectuée, la carte passe en mode utilisation (`USE`). Celui-ci permet d'effectuer n'importe quel traitement relatif au solde à condition de s'authentifier au préalable au moyen du bon code pin. Enfin, le mode devient invalide (`INVALID`) à l'issu d'un blocage du code pin utilisateur. Dans ce mode, il est seulement possible de réinitialiser le code pin utilisateur après s'être authentifié avec le code pin banque.

La Figure 2 présente le modèle fonctionnel de l'application qui a été utilisée pour notre expérimentation. Seule l'opération `setBpc` est présentée. Cette opération permet de fixer le code pin de la banque, ce qui ne peut avoir lieu que dans le cas où une session a été ouverte et si la carte est en phase de personnalisation. Les propriétés de sécurité ne sont pas toutes exprimées de manière explicite dans cette spécification fonctionnelle même si celles-ci doivent être prises en compte au niveau de l'implantation. Un modèle dédié à la sécurité complète donc les aspects fonctionnels. Ceci est présenté dans la partie suivante.

4 Modèle dédié à la sécurité

Nous commençons par présenter le format du modèle de sécurité. Nous nous intéressons ensuite au noyau de sécurité produit par Meca.

4.1 Format du modèle de sécurité

Comme nous l'avons représenté en Fig. 1, les entrées de Meca sont d'une part un modèle fonctionnel et d'autre part un modèle de politique de sécurité. Le modèle fonctionnel contient une description des entités sensibles du système ainsi que leur comportement fonctionnel.

Le modèle de politique de sécurité contient d'abord la description des deux entités qui interviennent dans le contrôle d'accès : les *sujets* et les *objets*. Les objets représentent les entités passives du modèle fonctionnel, dont les accès doivent être sécurisés. Les sujets sont les entités actives pour lesquelles on souhaite contrôler l'accès aux objets. Le modèle contient ensuite la liste des permissions qui sont données dans le contrôle d'accès. Pour Meca, il n'est possible de décrire que des permissions exprimant des autorisations. Implicitement tout ce qui n'apparaît pas dans les permissions est considéré comme interdit.

Nous avons également proposé un format de contrôle d'accès des utilisateurs (UAC - User Access Control) en fonction de conditions portant sur des attributs de sécurité. Il permet de contrôler l'exécution des opérations. La Figure 3 décrit le format d'entrée de Meca du type UAC. On remarquera que cette machine ne présente pas d'opérations et que les permissions sont exprimées à travers une constante.

Dans le cadre du modèle UAC, les permissions accordées aux sujets dépendent entre autres de l'état interne du système. Ainsi, certaines entités du système restreignent les permissions accordées dans le cadre d'une politique de sécurité. De telles entités sont appelées des *attributs de sécurité*. Par exemple, le terminal administratif est autorisé à enregistrer le code de la banque seulement lorsque la carte se trouve dans le mode personnalisation. Ceci se traduit dans le modèle UAC par la règle suivante :

$$\texttt{(mode = PERSO)} \Rightarrow \texttt{(terminalAdministratif} \mapsto \texttt{setBpc)} \in \texttt{permission}$$

```
MACHINE
    uac_Policy
SETS
    SUBJECT={s1,s2,...};          /* sujets du contrôle d'accès */
    OBJECT={op1,op2,...}          /* opérations du modèle fonctionnel */
CONSTANTS
    permission, ...
PROPERTIES
    permission ∈ SUBJECT ↔ OBJECT ∧
    condition ⇒ (s1 ↦ op1) ∈ permission ∧
    ...
```

Fig. 3. Schéma général du format du modèle de sécurité

4.2 Format du noyau de sécurité

Meca génère un noyau de sécurité qui correspond au modèle fonctionnel de l'application, auquel ont été ajoutées les conditions de sécurité qui servent à faire appliquer le contrôle d'accès. Ces conditions sont synthétisées à partir des permissions définies dans le modèle de politique de sécurité. En configurant certaines options de Meca, il est possible de paramétrer la forme du noyau de sécurité généré. Ainsi, Meca propose de générer un noyau qui est soit de forme offensive, soit de forme défensive.

Dans la forme défensive, chaque opération contient une précondition, plus deux branchements conditionnels protégeant l'accès à l'opération. Dans la précondition, on retrouve uniquement les informations de typage des paramètres d'appel, extraits du modèle fonctionnel. Les deux décisions doivent être satisfaites pour accéder à l'opération protégée. La première décision est la condition de sécurité. Elle correspond à une reformulation des règles du modèle de sécurité et pour l'opération considérée. La seconde décision est la condition fonctionnelle, qui correspond à la précondition de l'opération dans le modèle fonctionnel. La Figure 4 représente le format général du noyau de sécurité défensif pour le format UAC. On notera qu'une opération du noyau de sécurité est produite pour chaque opération du modèle fonctionnel qui est contrôlée.

```
MACHINE
    uac_def_SecurityKernel
OPERATIONS
    rs, rf, o1,...,oN ⟵ execute_op(su,i1,...,iN) ≙
       PRE
          su ∈ SUBJECT ∧ ...          /* typage des paramètres */
       THEN
          IF Condition_Secu           /* condition du mod. de sécurité */
          THEN     rs := OK ||
             IF Condition_Func        /* précondition du mod. fonct. */
             THEN  rf := OK ||
                o1,...,oN := op(i1,...,iN) /* opération du mod. fonct. */
             ELSE  rf := KO || skip
             END
          ELSE     rs := KO ||
             IF    Condition_Func     /* précondition du mod. fonct. */
             THEN  rf := OK || skip
             ELSE  rf := KO || skip
             END
          END
       END
...
```

Fig. 4. Schéma général du format du noyau de sécurité défensif pour le format UAC

Dans le cadre de notre expérimentation, nous utilisons la forme défensive du noyau de sécurité pour faire de la génération de cas de test. Un autre intérêt de cette forme est de pouvoir faire de la surveillance à l'exécution en utilisant un moniteur basé sur le noyau de sécurité. Son rôle est de gérer tout appel aux opérations de l'application en interceptant tout accès illicite. Comme nous l'avons présenté au début de cette partie, Meca peut aussi générer une forme offensive du noyau de sécurité. Dans ce cas, les deux conditions telles qu'elles apparaissent dans la forme défensive sont rassemblées au sein de la précondition. Cette forme est plutôt utilisée pour faire de la preuve et n'est pas l'objet de cet article.

Sachant que notre objectif se place dans le cadre du contrôle d'accès, nous avons choisi de séparer les deux conditions afin d'obtenir un contrôle plus fin. A cet effet, nous avons utilisé deux paramètres de retour, `rs` et `rf`, qui nous permettent d'observer, pour chaque appel, la satisfaction respective de la condition de sécurité et de la condition fonctionnelle.

Pour que le comportement fonctionnel de l'opération soit assuré (ce qui se traduit par l'appel de l'opération issue du modèle fonctionnel), il est nécessaire que la condition de sécurité et la condition fonctionnelle soient satisfaites (`rs=OK` et `rf=OK`). Lorsque la condition de sécurité n'est pas satisfaite (`rs=KO`), l'opération ne doit jamais être exécutée.

Un exemple d'opération du noyau de sécurité est donné en Fig. 5. On retrouve la *condition fonctionnelle* qui régit son exécution, issue du modèle fonctionnel : $\texttt{isOpenSess} = \texttt{TRUE} \wedge \texttt{pin} \in \texttt{BPC} \wedge \texttt{mode} = \texttt{PERSO}$, ainsi que la *condition de sécurité* générée par Meca, en fonction du modèle de sécurité : $\texttt{mode} = \texttt{PERSO} \wedge \texttt{su} = \texttt{terminalAdministratif}$.

5 Test de conformité à une politique de sécurité

Nous nous intéressons dans cette partie à la description du test vis-à-vis d'une politique de contrôle d'accès. Nous commençons par définir les tests que nous souhaitons obtenir. Puis, nous présentons comment s'établit le verdict de conformité par rapport au contrôle d'accès. Pour finir, nous présentons l'outil qui réalisera le calcul des cas de tests.

5.1 Stratégies de test du contrôle d'accès

C'est le noyau de sécurité qui est exploité pour produire les tests de conformité. En effet, celui-ci traduit l'aspect opérationnel de la politique de sécurité. Une opération du noyau de sécurité matérialise une tâche dans le système ; ici, il s'agit d'exécuter une opération du modèle fonctionnel par le noyau de sécurité.

Notre but est de nous assurer que lorsque la condition de sécurité est satisfaite, l'opération agit comme prévu, en fonction de sa précondition. Plus précisément, si cette dernière est satisfaite, le système évolue, dans le cas contraire, il reste dans le même état et aucune valeur de retour de l'opération du modèle fonctionnel ne peut être élaborée. Ainsi, seule l'activation licite d'une opération

```
rs,rf ← execute_setBpc(pin,su) ≙
    PRE
        su ∈ SUBJECT ∧ pin ∈ BPC
    THEN
        IF mode = PERSO ∧ su = terminalAdministratif
        THEN
            rs := OK ||
            IF isOpenSess = btrue ∧ pin ∈ BPC ∧ mode = PERSO
            THEN rf := OK || setBpc(pin)
            ELSE rf := KO
            END
        ELSE
            rs := KO ||
            IF isOpenSess = btrue ∧ pin ∈ BPC ∧ mode = PERSO
            THEN rf := OK
            ELSE rf := KO
            END
        END
    END
```

Fig. 5. L'opération `setBpc` du noyau de sécurité

est susceptible de modifier l'état interne du système et/ou d'accéder aux données du système. Pour ce faire, il est nécessaire de procéder à une phase d'observation qui nous permettra de décider du verdict du test.

Du point de vue de la stratégie de génération de test, nous cherchons à exploiter le comportement de chaque opération avec la satisfaction, ou la non-satisfaction de sa condition de sécurité. Ainsi, nous nous intéressons à la couverture des conditions de sécurité et des conditions fonctionnelles. Plus précisément, nous souhaitons que nos tests exhibent chacune des combinaisons suivantes :

- Satisfaction des conditions de sécurité et fonctionnelle (`rs` = `OK` et `rf` = `OK`)
- Satisfaction de la condition de sécurité et non-satisfaction de la condition fonctionnelle (`rs` = `OK` et `rf` = `KO`)
- Non-satisfaction de la condition de sécurité et satisfaction de la condition fonctionnelle (`rs` = `KO` et `rf` = `OK`)
- Non-satisfaction de la condition de sécurité et non-satisfaction de la condition fonctionnelle (`rs` = `KO` et `rf` = `KO`)

L'implantation doit tenir compte de la sécurité. Pour s'assurer qu'elle respecte bien la politique de sécurité, nous proposons de jouer les cas de test générés à partir du noyau de sécurité sur l'implantation. Lors de l'exécution des cas de test sur l'implémentation, la spécification nous permet d'établir les verdicts des tests pour décider si le test réussit ou échoue. L'établissement du verdict est discuté à présent.

5.2 Établissement du verdict

L'établissement du verdict se base sur les sorties produites par l'implantation par rapport aux sorties calculées par le modèle. Le modèle B à partir duquel sont calculés les tests encapsule les appels et retourne systématiquement une valeur représentant la satisfaction de la condition de sécurité (`rs`) et une valeur représentant la satisfaction de la condition fonctionnelle (`rf`) (cf. partie 4.2).

Nous supposons que l'implantation sous test est capable de faire la distinction entre une terminaison normale et une terminaison erronée, dûe sans distinction à une condition de sécurité incorrecte ou une condition fonctionnelle incorrecte. On notera que cette hypothèse est réaliste vis-à-vis d'une implantation de type JavaCard. En effet, un tel programme retourne un code nommé Status Word dont la valeur, définie dans le cahier des charges, indique les éventuelles erreurs (typiquement, 9000 signifie une terminaison normale, toute autre valeur signifie une terminaison anormale parce qu'une erreur d'un certain type est détectée). Nous faisons l'hypothèse qu'une terminaison anormale de l'application n'entraîne aucune évolution du système, et que celui-ci n'est susceptible d'évoluer que lorsque l'opération a terminé normalement.

Nous devons établir le verdict à partir des résultats produits par l'implantation et des résultats attendus donnés par le modèle de sécurité, et nous renseignant sur la conformité de l'implémentation. Ces verdicts sont établis suivant le Tableau 1.

Les tests qui nous intéressent sont ceux qui détectent une non-conformité liée spécifiquement à la condition de sécurité, c'est-à-dire lorsque l'implantation permet l'exécution normale du programme alors qu'une erreur était attendue (numéros (2), (3) et (4)), ou lorsqu'une erreur est détectée dans le programme alors que le modèle n'en attendait pas (numéro (5)). Typiquement, le cas (3) est intéressant car l'implantation a autorisé l'exécution de l'opération alors que le modèle prévoyait une erreur due au contrôle d'accès, ce qui signifie que l'implantation a, par exemple, oublié de prendre en compte un cas de refus de l'accès à l'application. Tout ceci correspond à une notion de conformité au sens strict (col. 3 du tableau).

Résultat donné par le programme	Modèle fonctionnel (`rs`,`rf`)	Verdict du test (conf. stricte)	Verdict du test (conf. du refus)	#
Terminaison normale	(`OK`,`OK`)	*Succès*	*Succès*	(1)
	(`OK`,`KO`)	*Échec*	*Échec*	(2)
	(`KO`,`OK`)	*Échec*	*Échec*	(3)
	(`KO`,`KO`)	*Échec*	*Échec*	(4)
Terminaison anormale (erreur de sécurité ou erreur fonctionnelle)	(`OK`,`OK`)	*Échec*	*Succès*	(5)
	(`OK`,`KO`)	*Succès*	*Succès*	(6)
	(`KO`,`OK`)	*Succès*	*Succès*	(7)
	(`KO`,`KO`)	*Succès*	*Succès*	(8)

Tab. 1. Etablissement du verdict en fonction du retour du programme

Néanmoins, la spécification peut possiblement accepter plus de comportements que l'implantation. Par exemple, le fait que le solde sur le porte-monnaie soit limité peut entraîner un comportement restreint une fois le programme embarqué sur une carte. De ce fait, nous ne cherchons pas à établir une conformité stricte entre l'implantation et le modèle de sécurité, mais nous cherchons plutôt à nous assurer que les comportements autorisés par l'application sont effectivement des comportements licites. De ce fait, nous distinguons dans le tableau la conformité "stricte" (tous les comportements autorisés par l'application sont autorisés par le modèle, et tous les comportements refusés par l'implantation sont refusés par le modèle) et la conformité relative au "refus" (col. 4 du tableau) : tous les comportements refusés par le modèle sont refusés par l'application, et l'application peut refuser des comportements autorisés dans le modèle.

Nous nous intéressons à présent à l'outil LTG que nous utilisons pour générer les suites de tests.

5.3 L'outil LEIRIOS Test Generator

LEIRIOS Test Generator (LTG) [8,13] est un outil de génération automatique de tests fonctionnels. Il peut prendre en paramètres différents formats dont des spécifications écrites en B. Les tests produits par LTG se décomposent comme illustré en Fig. 6.

Les cibles de tests sont d'abord calculées par rapport au modèle du système à tester. Une cible de test est l'activation d'un comportement extrait d'une opération du modèle B. Un comportement se définit basiquement comme un chemin d'exécution possible dans l'opération. Le moteur d'animation de LTG, basé sur l'animation symbolique à contraintes, est ensuite utilisé pour produire le *préambule*. Il s'agit d'une trace d'exécution menant, depuis l'état initial, à un état cible. Le *corps* du test en lui-même consiste en l'activation du comportement sélectionné, à partir du contexte définissant la cible de test. Une phase d'*observation* intervient ensuite. Celle-ci permet l'appel à des opérations spécifiques du modèle afin d'obtenir la valeur de variables du modèle. Cette phase permettra ensuite, lors de l'exécution du test, de comparer les résultats rendus par l'implantation par rapport aux résultats attendus, calculés sur le modèle. Enfin, le *postambule* est une séquence d'appels d'opérations permettant de remettre le système dans

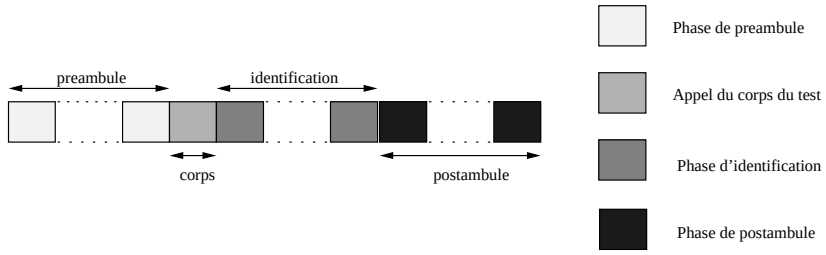

Fig. 6. Composition d'un cas de test produit par LTG

l'état initial. Il permet d'enchaîner les tests, sans avoir à passer par une remise à zéro manuelle du système, qui n'est pas toujours possible.

Calcul des cas de test Le processus de génération de tests de LTG s'appuie sur la couverture du modèle, au niveau structurel, au niveau des décisions et au niveau des données.

Couverture structurelle des opérations. Les tests produits par LTG ciblent la couverture des opérations du modèle fonctionnel. Pour ce faire, LTG considère les différents chemins du graphe de flot de contrôle de l'opération, qu'il nomme comportements.

Exemple 1 (Découpage en comportements) Pour cet exemple, nous considérons l'opération `execute_setBpc(su,pin)` extraite du noyau de sécurité produit par Meca, et présentée dans la Fig. 5. Cette opération se décompose en quatre comportements, correspondant aux imbrications de IF-THEN-ELSE, nommés cpt_1, cpt_2, cpt_3 et cpt_4, comme suit :

cpt_1	su ∈ SUBJECT ∧ pin ∈ BPC ∧ mode = PERSO ∧ su = terminalAdministratif ∧ isOpenSess = btrue ⟹ rs := OK ∧ rf := OK ‖setBpc(pin)
cpt_2	su ∈ SUBJECT ∧ pin ∈ BPC ∧ mode = PERSO ∧ su = terminalAdministratif ∧ isOpenSess ≠ btrue ⟹ rs := OK ‖ rf := KO
cpt_3	su ∈ SUBJECT ∧ pin ∈ BPC ∧ (mode ≠ PERSO ∨ su ≠ terminalAdministratif) ∧ isOpenSess = btrue ⟹ rs := KO ‖ rf := OK
cpt_4	su ∈ SUBJECT ∧ pin ∈ BPC ∧ (mode ≠ PERSO ∨ su ≠ terminalAdministratif) ∧ isOpenSess ≠ btrue ⟹ rs := KO ‖ rf := KO

Une cible de test est représentée par la condition d'activation du comportement considéré. Cette condition d'activation est calculée comme étant la conjonction des prédicats composant un comportement. Par exemple, la condition d'activation du comportement cpt_3 de l'exemple précédent est : su ∈ SUBJECT ∧ pin ∈ BPC ∧ (mode ≠ PERSO ∨ su ≠ terminalAdministratif) ∧ isOpenSess = btrue.

Couverture des décisions des opérations. En complément, LTG propose d'affiner la couverture du modèle en sélectionnant un critère de couverture des décisions relatives aux comportements [10]. Pour ce faire, il applique différentes réécritures portant sur les disjonctions du prédicat définissant la cible de test. Chacune de ces réécritures permet de satisfaire un critère spécifique de couverture des décisions. Le Tableau 2 décrit ces réécritures, ainsi que les critères associés [11].

La réécriture 1 laisse la disjonction telle qu'elle, ne produisant ainsi qu'une seule cible de test, saitsfaisant indifféremment P_1 ou P_2. La réécriture 2 produit deux cibles de test, visant la satisfaction de chacun des littéraux composant la disjonction. La réécriture 3 produit également deux cibles de test visant l'activation de chacun des littéraux composant la disjonction de manière exclusive

Nř	Réécriture de $P_1 \vee P_2$	Critère de couverture des décisions
1	$P_1 \vee P_2$	Decision Coverage (DC)
2	P_1, P_2	Condition/Decision Coverage (C/DC)
3	$P_1 \wedge \neg P_2$, $\neg P_1 \wedge P_2$	Modified Condition/Decision Coverage (MC/DC)
4	$P_1 \wedge P_2$, $P_1 \wedge \neg P_2$, $\neg P_1 \wedge P_2$	Multiple Condition Coverage (MCC)

Tab. 2. Critères de couverture des décisons en fonction des réécritures

par rapport aux autres littéraux. Enfin, la réécriture 4 permet de produire trois cibles de test, de manière à exhiber toutes les possibilités de satisfaire une disjonction.

Couverture des données. LTG propose également une gestion assez fine des valeurs des variables et des paramètres d'entrée d'une opération sous test. Ainsi, toutes les données qui ne sont pas précisées dans la cible du test peuvent se voir appliquer une sélection consistant à choisir par exemple : une seule valeur, des valeurs aux bornes des domaines (pour les domaines ordonnés), ou encore une énumération de toutes les valeurs possibles pour les domaines finis.

Lien avec le modèle fonctionnel sécurisé LTG est un outil qui se base sur l'analyse d'un modèle pour produire des tests boîte-noire pertinents. Nous nous sommes fixés des besoins de test liés au modèle du noyau de sécurité, décrits dans la partie précédente. La stratégie de génération de tests de LTG satisfait les critères que nous nous sommes fixés.

- La couverture structurelle des opérations nous permet de considérer tous les cas d'exécution d'une opération, par rapport à ses conditions d'accès et fonctionnelles.
- La couverture des disjonctions permet de jouer finement sur les différentes configurations du système provoquant l'autorisation ou le refus de l'accès. De plus, un mécanisme de résolution de contraintes élimine les cibles inconsistantes.
- La couverture des données permet l'énumération des paramètres d'entrée, augmentant ainsi le nombre de cas de tests. Dans notre cas, en énumérant toutes les valeurs possibles des sujets (paramètres `su`) nous sommes en mesure de couvrir de manière exhaustive tous les cas d'accès (ou de refus) pour chacun des sujets sur les différents objets.

6 Expérimentation

Nous nous intéressons à présent à la mise en œuvre de ces techniques dans le cadre d'une expérimentation[1]

[1] Les spécifications, les modèles et l'application utilisés pour cette expérimentation sont disponibles à l'adresse :
`http://www-lsr.imag.fr/Les.Personnes/Amal.Haddad/AFADL07/`

6.1 Génération de tests abstraits

LTG impose que le modèle fonctionnel fourni en entrée ne contienne pas d'appels d'opérations dans les substitutions, et que les données aient des domaines finis. Nous modifions donc le noyau de sécurité pour aplatir les appels d'opérations et nous bornons la variable représentant le solde du porte monnaie (initialement déclarée comme INT).

Nous donnons le noyau de sécurité modifié en entrée de LTG, qui génrère les cibles relatives aux branches d'exécutions de ce modèle. Nous choisissons le critère de couverture des décisions MCC de manière à avoir le plus de combinaisons possibles. De plus, pour chaque paramètre d'opération, nous choisissons d'énumérer toutes les valeurs possibles. Il en va de même pour les variables d'état qui sont impliquées dans les conditions de sécurité et les conditions fonctionnelles des opérations ciblées. Les variables numériques (comme les codes pin) sont instanciées aux limites, en laissant LTG sélectionner une valeur extremum du domaine des variables. Nous énumérons toutes les valeurs des variables ayant un domaine fini et de faible cardinalité (par exemple, le mode).

Intuitivement, ces choix nous permettent de tester les conditions d'accès pour tous les sujets possibles (les sujets étant en paramètre des opérations testées). De plus, l'énumération des valeurs des variables d'état nous permet de couvrir un grand nombre de configurations à partir desquelles réaliser les tests.

Nous ne choisissons pas d'opération d'observation ; nous laissons au pilote de test le soin d'observer la terminaison d'une opération. Pour simplifier, nous choisissons de créer de nouveaux objets pour chaque cas de test, ce qui nous dispense de générer des postambules.

Le nombre de tests est donné par le tableau 3, pour chacune des opérations du modèle fonctionnel. Au total, notre suite de test se compose de 203 cas de test. On remarque que les nombres de cibles générées pour `setHpc` et `getBalance` sont supérieures aux nombres de cas de tests. Ceci est dû aux combinaisons de littéraux, issus de la réécriture choisie, qui produisent des cibles de tests inatteignables.

Opération	Nombre de cibles	Nombre de cas de test
`setBcp`	18	18
`beginSession`	6	6
`endSession`	6	6
`setHpc`	36	24
`authBank`	18	18
`checkPin`	24	32
`getBalance`	24	15
`debit`	42	46
`credit`	29	45
Total	**231**	**203**

Tab. 3. Résultat de la génération de tests pour l'exemple

6.2 Concrétisation des tests

Les tests abstraits produits par LTG sont ensuite concrétisés pour être exécutés sur l'implantation sous test. Celle-ci se présente sous la forme d'une classe Java qui décrit le porte-monnaie et dispose des mêmes points d'entrée que le modèle fonctionnel. Les méthodes sont susceptibles de déclencher des exceptions relatives au non-respect des conditions du cahier des charges.

Un pilote de test spécifique, écrit en Java, est en charge de l'exécution des cas de test et de l'observation de la terminaison des méthodes. Soit la méthode termine normalement et aucune exception n'est déclenchée, soit la méthode déclenche une exception et celle-ci est rattrapée par le pilote de test. Le verdict est ensuite établi par une fonction qui considère cette terminaison, ainsi que les valeurs attendues pour `rs` et `rf` issues du modèle, et qui calcule le verdict, *Succès* ou *Échec*, selon le tableau 1 (colonne "refus").

Nous avons joué notre suite de tests sur notre application, et nous n'avons détecté aucune erreur. Nous avons donc cherché à évaluer la pertinence de notre suite de tests. Pour ce faire, nous nous sommes tournés vers une analyse mutationnelle.

6.3 Évaluation des tests produits

L'analyse mutationnelle consiste à introduire volontairement des erreurs dans un programme correct. Une variante erronnée du programme est appelée *mutant*. Ces mutants servent à mesurer l'efficacité d'une suite de test, au sens où cette dernière doit être capable de détecter que les mutants sont incorrects. Quand la suite de test identifie une erreur dans un mutant, on dit que le mutant est tué.

Nous avons généré 30 mutants de notre application (considérée comme référence). Les mutations que nous avons introduites concernent spécifiquement l'implantation des vérifications des conditions d'accès. Les mutations que nous avons introduites concernent l'évaluation des différentes conditions d'accès régissant l'exécution des méthodes de l'application. Elles simulent des erreurs classiques de programmation. Ainsi, nous avons effectué les modifications suivantes :

- remplacement de `&&` par `||`, et inversement ;
- remplacement d'un prédicat d'une conjonction ou d'une disjonction par `true` ou `false` ;
- négation des prédicats dans les `if` ;
- suppression d'une vérification.

Ces mutations guidées représentent des erreurs de programmation les plus classiques. Notre suite de test a réussi à tuer les 30 mutants, nous confortant dans l'efficacité de la méthode de génération de tests que nous avons proposée.

7 Discussion

Pour s'assurer de la sécurité d'une application, il est nécessaire de vérifier que si une opération n'a pas été appelée dans des conditions d'accès licites, alors le

système doit rester dans le même état, ou alors ne pas effectuer de modification qui compromettrait ultérieurement la sécurité du système. C'est l'hypothèse que laquelle nous sommes partis.

Dans notre expérimentation, si l'opération de l'application termine anormalement (en déclenchant une exception), nous avons supposé qu'aucune variable n'était modifiée. Néanmoins, ce n'est pas nécessairement le cas. D'une part, le modèle peut admettre des évolutions des attributs de sécurité lorsque les accès sont refusés, tels que des compteurs de ratification. D'autre part, notre notion d'observation des résulats obtenus étant assez grossière, nous ne sommes pas en mesure de nous assurer que si aucun attribut n'est supposé évoluer dans le modèle, c'est effectivement le cas dans l'implantation.

Pour résoudre le premier point, nous sommes actuellement en train de travailler à l'évolution du modèle de sécurité, de manière à ce qu'il prenne en compte la dynamique des attributs de sécurité. Ceci aura nécéssairement un impact sur les formats, à la fois du modèle de sécurité et du noyau de sécurité.

Pour nous assurer qu'en dehors des conditions d'accès licites le système n'évolue pas, comme identifié dans le second point, il est nécessaire d'observer les valeurs des variables d'état. Ceci peut être réalisé de manière directe, si le système sous test fournit des primitives permettant de récupérer des valeurs de variables (potentiellement sensibles) du système. Cette hypothèse est réaliste ; en effet, les fabricants de carte sont parfois amenés à ajouter de telles primitives dans le cadre de la phase de validation, avant de les retirer pour la version finale du logiciel. Néanmoins, nous aimerions ne pas avoir à demander ce genre de contraintes à notre système ; nous envisageons donc une autre solution.

Notre seul point de jonction entre le modèle et l'application étant les opérations, nous proposons de nous assurer que le système n'a pas évolué en considérant une phase d'observation particulière. Celle-ci consiste à tenter d'effectuer, à la suite du corps du test, des opérations qui devraient être refusées du point de vue du contrôle d'accès. Si ces opérations peuvent effectivement être activées sur l'implantation alors, même si le corps du test (l'opération que l'on teste à l'origine) est conforme aux prédictions, il rend possible l'exécution d'actions illicites, ce qui est une erreur. Dans ce cas, le test serait un échec. Concrètement, nous chercherions à activer, suite au corps du test, toutes les opérations qui sont susceptibles de déclencher une erreur d'accès (sur le modèle `rs=KO` et `rf=OK`). De ce fait, nous sommes capables d'étendre notre pouvoir de décision du verdict du test, sans pour autant demander une fonctionnalité supplémentaire à l'application.

Nous travaillons actuellement à la mise en place de telles stratégies de détection, employant une approche combinatoire et une technique de filtrage des cas de tests vis-à-vis du résultat obtenu.

8 Conclusion et Perspectives

Nous avons présenté dans cet article une approche visant à valider de manière automatique une implantation vis-à-vis d'une politique de sécurité décrivant le

contrôle d'accès. Nous nous sommes focalisés sur une politique décrivant des permissions. Nous avons défini une notion de conformité d'une application vis-à-vis du contrôle d'accès. Notre approche considère un noyau de sécurité, sous la forme d'un modèle B, dont la couverture permet de réaliser, avec l'outil LTG, des tests pertinents par rapport aux propriétés d'accès. Pour valider notre démarche, nous avons proposé une expérimentation sur un exemple réaliste d'un porte-monnaie électronique, auquel est associée une politique de sécurité.

Cette approche est proposée dans le cadre du projet RNTL POSE[2] qui vise à valider la sécurité des systèmes embarqués de type carte à puce.

Peu de travaux similaires se sont intéressés à l'utilisation des modèles pour la génération du test du contrôle d'accès. L'approche classique est l'audit. Néanmoins, des approches ont été menées dans le cadre du test de contrôle d'accès dans des réseaux, une politique de sécurité décrivant ainsi un ensemble de permissions pour accéder à des services par des machines. L'approche présentée dans [12] utilise une formalisation par automates de Mealy pour décrire la politique de sécurité du réseau. Les cas de tests sont ensuite générés classiquement à partir de ces machines à états finis. Dans le cadre du projet POTESTAT [5], les auteurs formalisent le contrôle d'accès par un ensemble de formules temporelles, destinées à tester le contrôle d'accès d'un réseau. Le test de contrôle d'accès se prête bien au domaine des réseaux car il s'agit d'un système moins complexe à modéliser que le fonctionnement d'une application de type carte à puce. Ainsi, un modèle de sécurité suffit pour décrire une cible de test, le test en lui-même se résumant à envoyer des paquets sur le réseau, à partir d'une configuration précise (adresse IP, sous-réseau, etc.) et d'observer si les paquets sont acceptés ou rejetés. De ce fait, les attributs du sécurité du réseau sont complètement indépendants et ne sont pas sujet à des évolutions. Il n'est donc pas nécessaire de modéliser le réseau pour générer des tests. Notre approche se distingue des précédentes du fait que nous utilisons, en plus du modèle de sécurité, un modèle fonctionnel décrivant le système. Celui-ci nous permet d'être plus proche de l'implantation et de nous confronter à n'importe quel système, du moment que son comportement fonctionnel peut être modélisé.

Dans cet article, nous nous sommes intéressés à la notion de contrôle d'accès qui est une sous-classe de la classe FDP des CC. Nous pensons que ces travaux peuvent s'étendre à d'autres spécifications fonctionnelles de sécurité (SFR), telles que la gestion des droits d'accès (classe Administration de la sécurité – FMT) des CC, ou la protection des données (sous-classe FDP_RIP de la classe FDP).

D'autre part, nous souhaiterions formaliser la notion de couverture du contrôle d'accès. Nous avons, dans ce papier, considéré cette notion de manière intuitive et elle nous a guidé dans le choix de critères de couverture du modèle par l'outil LTG. L'objectif de cette étape est de pouvoir ensuite mesurer cette couverture et ainsi pouvoir évaluer différentes suites de test en fonction de celle-ci. En conséquence, nous pensons utiliser cette mesure de couverture pour filtrer des cas de tests, ou travailler sur la réduction de suites de tests, qui pourraient par exemple être produites par une approche combinatoire. Cette volonté est inscrite dans

[2] http ://www.rntl-pose.info

la sous-classe ATE_COV, de la classe Assurance des Tests (ATE) des Critères Communs (CC), qui décrit les exigences relatives à la complétude de la couverture des tests vis-à-vis des descriptions des fonctions de sécurité.

Références

1. J.-R. Abrial. *The B-book : assigning programs to meanings.* Cambridge University Press, 1996.
2. B. Beizer. *Black-Box Testing : Techniques for Functional Testing of Software and Systems.* John Wiley & Sons, New York, USA, 1995.
3. D. Elliot Bell and Leonard J. LaPadula. Secure computer systems : A mathematical model, volume ii. *Journal of Computer Security*, 4(2/3) :229–263, 1996.
4. K. J. Biba. Integrity Considerations for Secure Computer Systems. Technical Report ESD-TR-76-372, USAF Electronic Systems Division, Hanscom Air Force Base, Bedford, Massachusetts, April 1977.
5. Vianney Darmaillacq, Jean-Claude Fernandez, Roland Groz, Laurent Mounier, and Jean-Luc Richier. Test generation for network security rules. In M. Ümit Uyar, Ali Y. Duale, and Mariusz A. Fecko, editors, *TestCom*, volume 3964 of *Lecture Notes in Computer Science*, pages 341–356. Springer, 2006.
6. Ferraiolo D.F. and Kuhn Richard. Role-Based Access Control. In *Proceedings of the 15th NIST-NSA National Computer Security Conference*, pages 554–563, Baltimore, MD, USA, October 1992. Nat'l Inst. Standards and Technology.
7. Amal Haddad. Meca : A tool for access control models. In *B'2007, the 7th Int. B Conference*, volume 4355 of *LNCS*, pages 281–284, Besançon, France, January 2007. Springer.
8. E. Jaffuel and B. Legeard. LEIRIOS Test Generator : Automated Test Generation from B Models. In *B'2007, the 7th Int. B Conference*, volume 4355 of *LNCS*, pages 277–281, Besançon, France, January 2007. Springer.
9. Butler W. Lampson. Protection. *SIGOPS Oper. Syst. Rev.*, 8(1) :18–24, 1974.
10. B. Legeard, F. Peureux, and M. Utting. Controlling test case explosion in test generation from B formal models. *Software Testing, Verification and Reliability, STVR*, 14(2) :81–103, 2004.
11. A.J. Offutt, Y. Xiong, and S. Liu. Criteria for generating specification-based tests. In *Proceedings of the 5th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'99)*, pages 119–131, Las Vegas, USA, October 1999. IEEE Computer Society Press.
12. Diana Senn, David A. Basin, and Germano Caronni. Firewall conformance testing. In Ferhat Khendek and Rachida Dssouli, editors, *TestCom*, volume 3502 of *Lecture Notes in Computer Science*, pages 226–241. Springer, 2005.
13. M. Utting and B. Legeard. *Practical Model-Based Testing : A Tools Approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.