Automatic generation of vulnerability tests for the Java Card byte code verifier

Aymerick Savary
GRIL, Département Informatique,
Université de Sherbrooke,
Québec, Canada
Email: aymerick.savary@usherbrooke.ca

Marc Frappier
GRIL, Département Informatique,
Université de Sherbrooke,
Québec, Canada
Email: marc.frappier@usherbrooke.ca

Jean-Louis Lanet
XLIM/DMI/SSD,
87 rue d'Isles, 87000 Limoges,
France
Email: jean-louis.lanet@unilim.fr

***Abstract*—The byte code verifier is an essential component of the Java Card security platform. Test generation to assess its implementation is mandatory, however, comprehensive test plans are too intricate to be handmade. Therefore, automating their generation is an interesting avenue. Our approach is based on formal methods, an important asset to find a preamble, a postamble, or the entire set of test cases for each instruction. The proposed vulnerability tests confirm that a behavior rejected by the model is also rejected by its implementation. Our results show that the techniques put forward achieve such a goal, through a simplified language.**

I. Introduction

Java est un langage qui, dès le début de sa conception, a été étudié pour assurer la sécurité. Différentes contraintes ont permis d'atteindre un tel niveau de sécurité, par exemple le typage fort ou bien encore l'exécution dans un contexte pour chaque programme. L'un des composants de java assurant cette sécurité est le BCV (*Byte Code Verifier*) qui s'assure par une analyse statique du respect de toutes les contraintes imposées par la JVM (*Java Virtual Machine*). Dans le cadre de cet article, nous allons nous intéresser à Java Card, qui est un sous-ensemble de Java dédié pour les cartes à puce. L'un des principaux problèmes que l'on peut rencontrer sur cette plate-forme d'exécution est le manque de place pour stocker le BCV. En effet, on ne dispose en général que de quelques Ko de ROM et de peu de RAM. Par conséquent, cela peut conduire à un BCV n'implémentant pas complètement la spécification, et donc des risques de vulnérabilité. Il devient alors intéressant de procéder à des campagnes de tests pour s'assurer de leur conformité. Tous ces tests reposent sur le principe de boîte noire, car nous ne connaissons pas le code utilisé par un BCV en particulier. L'un des avantages de cette stratégie est que l'on pourra réaliser une suite de tests générique pour toutes les implémentations de BCV. En effet, un byte code valide ou invalide pour la spécification doit aussi l'être pour toute implémentation. Une des principales difficultés de la génération de cas de tests réside dans la complexité de la spécification des contraintes de sécurité, qui nécessite une analyse des chemins d'exécution du byte code. L'utilisation de méthodes formelles offre la possibilité de raisonner sur un modèle sans ambiguïté. Malheureusement, les recherches sur la génération de tests à partir de modèles formels n'ont été menées pour le moment que pour vérifier qu'un comportement autorisé par un modèle l'est aussi par son implémentation. Dans cet article nous cherchons à générer un jeu de tests correspondant à des comportements interdits par la spécification devant être rejetée par le BCV.

Dans la section 2, nous commencerons par expliquer les techniques liées à la sécurité mise en place pour la plateforme Java Card. Puis nous décrirons rapidement comment générer une suite de tests à l'aide d'un modèle formel. Dans la section 3, nous expliquerons la théorie de notre approche pour générer la suite de tests de vulnérabilité pour un sous-langage. Enfin dans la section 4, nous expliquerons comment elle peut être automatisée et comment nous obtenons une suite de tests en pratique.

II. Fondements

Java Card est une plateforme semblable aux autres éditions de Java, ne se différenciant (au moins pour la version Classic) que par trois aspects: la restriction sur le langage, l'environnement d'exécution et le cycle de vie des applets. En raison des contraintes de ressources, celle-ci doit être divisée en deux parties, l'une s'exécutant en dehors de la carte, l'autre à l'intérieur. Sa sécurité est assurée par plusieurs éléments agissant de façon complémentaire et distribuée dans ces différentes parties. À l'extérieur de la carte, le BCV effectue une analyse statique qui garantit la validité du code à charger dans la carte. Lors du chargement, la couche logicielle Global Platform assure l'authentification et l'intégrité des applets chargées. Une fois les applets sur la carte, le pare-feu se charge de maintenir une ségrégation entre chaque programme.

A. La sécurité de la plate forme Java Card

La plate-forme Java Card est un environnement multiapplications dans laquelle les données critiques (clés de chiffrement, etc.) doivent être protégées contre les accès malveillants provenant de commandes ou d'autres applets. Pour obtenir cet isolement, la technologie Java utilise la

vérification de type, le chargeur de classes et le gestionnaire de sécurité afin de créer des espaces de noms pour les applications. Dans une carte à puce, ce modèle est amendé: la vérification de type est exécutée en dehors de la carte en raison de contraintes de mémoire, le chargeur de classes est unique et ne peut être surchargé et le gestionnaire de sécurité est remplacé par le pare-feu.

1) Enjeux de la vérification de byte code: Permettre de charger du code dans la carte après déploiement soulève les mêmes questions qu'avec les applets Java classiques. Le système doit vérifier que les règles de typage et de visibilité validées par le compilateur Java sont conservées lors du chargement du byte code.

Le langage Java est fortement typé, ce qui signifie que chaque variable et chaque expression disposent d'un type qui est déterminé au moment de la compilation. Pourtant, les types des variables locales ainsi que les éléments de la pile d'évaluation ne sont pas immuables, même durant l'exécution de la méthode. Il faut donc s'assurer par exécution symbolique et un calcul de point fixe que chaque opération ne peut être exécutée qu'avec des opérandes ayant le bon type.

Le BCV est responsable de cette vérification, et c'est un composant de sécurité indispensable dans le modèle d'isolement de Java. Cependant, un défaut dans le vérificateur peut provoquer l'acceptation d'une applet mal typée pouvant entrainer une attaque par confusion de type.

2) Le pare-feu: La séparation entre les différentes applets est réalisée par le pare-feu qui est basé sur la notion de contextes associés à chaque package de Java Card. Quand une applet est créée, le JCRE (Java Card Runtime Environment) utilise un identifiant unique, celui du package auquel elle appartient, pour marquer le propriétaire de chaque objet créé par cette applet. Si deux applets sont des instances de classes provenant du même package Java Card, elles sont considérées comme appartenant au même contexte. Un contexte a un rôle particulier, celui de super-utilisateur, qui est dédié au JCRE. Ceci lui permet d'accéder aux objets d'un autre contexte sur la carte.

3) La vérification de byte code: La vérification de fichier CAP est le processus offensif de la JCVM. Il a pour rôle de s'assurer que le fichier envoyé à la carte respecte la spécification. Il prend en entrée un fichier à vérifier, l'analyse puis l'autorise à être chargé ou non. Il est possible qu'un code valide soit rejeté dû a approxiamtion faite par l'interprétation abstraite, ce qui n'est pas gênant d'un point de vue de la sécurité. En revanche, si un code non valide est autorisé à être exécuté pour une implémentation donnée d'un BCV, cela correspond à une vulnérabilité qu'un attaquant peut exploiter.

Cette vérification se découpe en deux modules, à savoir le vérifieur de structure et le vérifieur de type, qui s'appliquent chacun à un type de vérification particulière.

La vérification de structure concerne l'analyse du flot de données qui est reçu par la carte à puce. Lors du chargement, les composants constituant le fichier CAP sont envoyés l'un après l'autre afin d'être installés sur la carte. Cependant, il est tout à fait possible que la structure même des composants soit altérée et que certains éléments du composant soient alors inaccessibles.

Le vérificateur de structure vérifie que les champs composant le fichier CAP sont bien formés. Pour cela, il analyse les composants contenus dans ce fichier et vérifie la cohérence du tout, pour assurer le respect des contraintes de la JCVM. Deux types de contraintes sont présents, à savoir les contraintes internes et les contraintes externes. Pour les contraintes internes, cela consiste à analyser pour chaque composant si son format est valide. Pour les contraintes externes, il faut prendre en compte plusieurs composants et vérifier que toutes les contraintes qu'un composant impose à un autre sont respectées. D'autres tests, plus complexes, par exemple l'absence de cycle dans la hiérarchie d'héritage ou encore le respect des contraintes de visibilité (public/private/restrict) sont effectués.

La vérification de type est la partie complexe du processus de vérification. Il s'agit d'établir, pour chaque point du programme, le type de chaque variable locale et de chaque élément de la pile. Ainsi, il est possible, une fois cette information obtenue, de vérifier si l'utilisation des variables locales et des éléments de la pile s'effectue correctement du point de vue du typage. Notons en préambule que la vérification de programmes Java peut se faire méthode par méthode, car les sauts d'instructions ne peuvent se faire qu'à l'intérieur d'une méthode.

Pour effectuer cette tâche complexe, il nous faut définir plusieurs éléments: un treillis pour comparer les types, des fonctions de transfert permettant de faire évoluer du point de vue du typage les variables locales et la pile, et un algorithme de parcours des instructions contenues dans l'application. Cet algorithme réalise une exécution symbolique afin de passer à travers tous les chemins d'exécution du code pour s'assurer que les préconditions de chaque instruction sont bien respectées.

4) Attaque sur carte à puce: Les attaques actuelles contre les cartes à puce sont essentiellement des attaques matérielles voir [1], [2]. Ces attaques nécessitent généralement des moyens physiques et des outils adaptés (lasers, oscilloscopes, etc.) réduisant par leur sophistication le nombre d'attaquants. Par contre, de nouvelles attaques, dites logiques, commencent à voir le jour et reposent souvent sur la découverte d'une faille et de son exploitation. Ce type d'attaques, plus difficile à réaliser, mais ne nécessitant aucun matériel, est donc à la portée d'un plus grand nombre d'attaquants. [3] propose plusieurs attaques sur Java Card en utilisant un code mal typé. L'idée est d'exploiter une confusion de type entre les tableaux de types primitifs différents. Il est possible de lire ou d'écrire dans des tableaux de types différents en utilisant une faille dans le mécanisme de transaction de Java Card. Nous avons montré dans [4] qu'un fichier structurellement mal formé pourrait permettre d'introduire un virus dans une Java Card.

Il est très fortement probable que d'autres faiblesses soient encore présentes dans les implémentations de vérification du byte code. Dans la section 2.1, nous avons vu que le vérifieur est composé de deux entités, l'une vérifiant la structure et l'autre le type. Ces entités sont complexes et totalement différentes du point de vue de leur stratégie de tests à mettre en oeuvre. Deux projets différents ont donc été constitués afin de traiter chacun d'entre eux. Le travail à réaliser sur le vérifieur de structure est en cours et fera l'objet d'une publication ultérieure . Dans le cadre de ce papier, l'accent sera uniquement mis sur le vérifieur de type.

B. La génération de tests à l'aide de modèle

La construction de logiciels de qualité ne peut désormais plus se passer de certaines vérifications. Pour arriver à avoir une bonne confiance en ces programmes, des techniques ont été mises au point et parmi celles-ci on trouve le test. Dans la plupart des cas, la confiance apportée par une bonne campagne de tests permet de valider le fonctionnement correct d'un programme.

Les modèles formelles offrent quelques avantages non négligeables qui peuvent être réutilisés pour effectuer et automatiser la génération de tests. De plus, la manipulation d'un modèle comparativement à un code source, améliore la simplicité de raisonnement et de manipulation. Cependant, le modèle d'un système pour son implémentation est parfois différent de celui utilisé pour son test. En effet, les tests peuvent s'appliquer seulement sur une sous-partie du modèle correspondant à une couche d'abstraction de haut niveau.

Suivant cette idée, Bull, cités par [5], a mené des recherches dans lesquelles est exposé comment il est possible à l'aide de modèles formelles de générer des suites de tests. Par la suite de nombreux travaux tels que ceux de [6] ont permis de passer certaines barrières et d'arriver à des solutions acceptables. Cependant, les études concernent pour le moment uniquement des tests vérifiant qu'un comportement accepté par un modèle l'est aussi par son implémentation.

III. Approche

Notre objectif est de tester à l'aide d'un modèle formel qu'une implémentation d'un BCV ne possède pas de vulnérabilité, ou, autrement dit, qu'il n'autorisera pas l'exécution d'un programme qui ne devrait pas l'être. Dans notre approche, ne possédant pas le code de l'implémentation du BCV mais simplement sa spécification, nous nous plaçons dans une stratégie de boîte noire. De plus, nous cherchons à identifier précisément les zones de vulnérabilité pour pouvoir par la suite corriger les problèmes mis en évidence. Pour ce faire, nous allons partir de la spécification informelle du BCV, puis la formaliser dans un modèle. Ensuite, à l'aide de ce modèle, nous générons une suite de fichiers invalides au regard de la spécification. Enfin, on envoie successivement les fichiers invalides à une carte possédant un BCV et on stocke la liste de ceux autorisés à être exécutés. Pour confirmer la présence de vulnérabilités, il suffit de consulter la liste des fichiers autorisés.

Le nombre d'instructions de Java Card étant relativement important, il serait long de toutes les prendre en compte pour étudier la faisabilité de notre approche. Le byte code Java Card est composé d'une centaine d'instructions et nombre de ces instructions possèdent des préconditions quasiment similaires, ce qui conduit à effectuer le même type de tests de multiples fois. Dans notre approche, nous avons donc préféré commencer à travailler sur un sous-langage. Un modèle simplifié de la machine virtuelle Java est proposé dans [7]. Les instructions choisies dans ce modèle représentent l'ensemble des flots de contrôle et de manipulation des données existant dans la machine Java originale et comporte seulement 12 instructions. Il a déjà été utilisé dans d'autres recherches sur Java comme [8]. C'est donc une bonne abstraction pour valider notre approche.

Pour illustrer nos propos, nous étudierons l'instruction `Aload x` qui est l'une des instructions les plus complexes de notre sous-langage. Nous donnons ici sa spécification informelle.

Aload x : empile l'objet de type `Obj` de la variable locale x dans la pile d'exécution. La taille maximum de la pile est symbolisée par $pilePleine$. La taille de la pile est obtenue grâce à la fonction $size(pile)$. Le type de l'objet référencé par x est dénoté par $VarLoc(x)$, où x est un indice d'un vecteur. L'indice x doit appartenir au domaine des variables locales pour ne pas accéder à des zones mémoire interdites. Enfin, le programme ne doit pas être terminé. Le contenu de la variable à la position x est stocké au sommet de pile.
PRE $size(pile) < pilePleine \wedge x \in dom(VarLoc) \wedge VarLoc(x) = \texttt{Obj} \wedge halt = 0$
POST $pile(size(pile)) := VarLoc(x) \wedge size(pile) + 1$

A. Définition des objectifs

Un byte code est rejeté lorsqu'une de ses instructions ne satisfait pas la spécification du langage. Dans cette spécification, chaque instruction est définie à l'aide d'une précondition à satisfaire et d'une postcondition décrivant les effets de l'instruction. La tâche du BCV est de vérifier que la précondition est satisfaite pour chaque instruction du byte code. On distingue deux cas: un byte code contenant i) une seule instruction invalide, ou ii) plusieurs instructions invalides. S'il contient une seule instruction invalide, on sait alors que le BCV peut détecter cette erreur. Si le byte code contient plusieurs instructions invalides, il est impossible de savoir si le BCV est capable de détecter chacune des erreurs. Il est donc préférable d'avoir une seule erreur par cas de test. Nous posons comme hypothèse qu'il est très peu probable qu'un BCV détecte tous les cas avec une seule erreur, mais qu'il ne détecte pas certains cas contenant plusieurs erreurs. Nous

allons donc nous restreindre aux cas de test contenant une seule erreur.

Si une précondition contient une conjonction, il est préférable que l'erreur soit localisée dans un seul élément de cette conjonction, afin de faciliter l'identification de l'erreur dans le BCV. Notons que notre objectif est de détecter les vulnérabilités, afin de les corriger. Nous ne nous intéressons pas à exploiter ces vulnérabilités, ce qui justifie de s'intéresser strictement aux cas de test contenant une seule erreur bien ciblée. Nous appelons *instruction sous test* (IST) l'instruction fautive dans un byte code. Nous appelons *préambule* les instructions apparaissant avant l'IST et *postambule* les instructions apparaissant après l'IST.

Pour générer un fichier contenant le byte code, il faut d'abord choisir une IST, générer un préambule menant à cette IST et générer un postambule menant à un état final valide de la machine virtuelle. Pour générer le préambule, il faut déterminer l'état de la machine virtuelle entraînant la violation de la spécification par l'IST. Pour générer le postambule, il faut déterminer l'effet de l'IST et atteindre ensuite un état final valide.

B. Génération des IST

Commençons par étudier la structure des préconditions. La spécification impose pour une même instruction plusieurs contraintes, qui sont représentées sous forme normale conjonctive. Si on procède à une négation de toute la précondition, nous sommes sûr de couvrir tous les cas interdits pour une instruction. Cela correspond à: $\neg(c_1 \wedge c_2) \rightsquigarrow \{\neg c_1 \wedge c_2,\ c_1 \wedge \neg c_2,\ \neg c_1 \wedge \neg c_2\}$. La notation $\neg c \rightsquigarrow \{c_1, \ldots, c_n\}$ signifie que la négation de c est ré-écrite en n cas de tests c_i. Puisque nous cherchons à identifier le plus précisément possible la source d'une erreur, le cas $\neg c_1 \wedge \neg c_2$ où il y a plus d'une erreur sera généralement ignoré, sauf dans le cas où c_1 et c_2 portent sur la même structure de données. Lorsque deux contraintes portent sur la même structure, il faut les re-grouper pour les traiter en bloc, afin de considérer les cas où deux contraintes sont niées en même temps. On appelle *sous-ensemble de clauses* ces contraintes concernant la même structure de données. Par exemple, si on considère trois contraintes c_{1_1}, c_{1_2} et c_2, avec les deux premières s'appliquant sur la même structure de données et la dernière sur une autre structure, nous avons : $\neg((c_{1_1} \wedge c_{1_2}) \wedge c_2) \rightsquigarrow \{\neg(c_{1_1} \wedge c_{1_2}) \wedge c_2,\ (c_{1_1} \wedge c_{1_2}) \wedge \neg c_2\}$. De cette façon, nous obtenons toujours des instructions invalides, mais ne possédant qu'un nombre limité de contraintes fausses. Ainsi, pour des sous-ensembles de clauses c_1 et c_2, nous obtenons: $\neg(c_1 \wedge c_2) \rightsquigarrow \{\neg c_1 \wedge c_2,\ c_1 \wedge \neg c_2\}$. À l'intérieur d'un sous-ensemble de clauses, si on rencontre une conjonction, nous appliquons cette fois-ci la négation classique de la logique sans supprimer de cas: $\neg(c_{1_1} \wedge c_{1_2}) \rightsquigarrow \{\neg c_{1_1} \wedge c_{1_2},\ c_{1_1} \wedge \neg c_{1_2},\ \neg c_{1_1} \wedge \neg c_{1_2}\}$. Bien que les contraintes de notre sous-langage ne comportent pas de disjonction, leur traitement serait: $\neg(c_1 \vee c_2) \rightsquigarrow \{\neg c_1 \wedge \neg c_2\}$.

Nous allons maintenant identifier les différents types de préconditions que l'on peut rencontrer. Nous en avons trois formes: i) celle sur les opérations arithmétiques, ii) celles sur l'appartenance à un ensemble et iii) celles sur le type. Pour les opérateurs de l'arithmétique, nous avons:

- $\neg(a = b) \rightsquigarrow \{a < b,\ a > b\}$
- $\neg(a > b) \rightsquigarrow \{a = b,\ a < b\}$
- $\neg(a < b) \rightsquigarrow \{a = b,\ a > b\}$
- $\neg(a \neq b) \rightsquigarrow \{a = b\}$

Une opération d'appartenance est simplement remplacée par sa négation:

- $\neg(a \in B) \rightsquigarrow \{a \notin B\}$
- $\neg(a \notin B) \rightsquigarrow \{a \in B\}$

Ensuite viennent les préconditions concernant le typage. Nous disposons d'un treillis de types constitué de trois éléments (sans compter `Top` et `Bottom`): le type entier noté `Int`, le type objet noté `Obj` et le type objet non initialisé noté `Uobj`. Le type `Obj` hérite de `Uobj`, puis `Uobj` et `Int` héritent de `Top`. La figure 1 représente ce treillis.
La négation d'un élément appartenant à ce treillis respecte

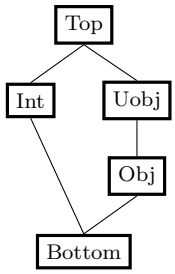

Fig. 1. Treillis de type

le principe d'héritage. Nous obtenons donc:

- $\neg(\texttt{Int}) \rightsquigarrow \{\texttt{Uobj},\ \texttt{Obj}\}$
- $\neg(\texttt{Obj}) \rightsquigarrow \{\texttt{Uobj},\ \texttt{Int}\}$
- $\neg(\texttt{Uobj}) \rightsquigarrow \{\texttt{Int}\}$

Certaines négations entraînent un nombre de cas de tests très important. Par exemple, la formule $a = b$ retourne un ensemble contenant beaucoup d'éléments. Pour traiter ce problème, nous utilisons une technique de tests aux bornes, en ré-écrivant les prédicats comme suit. Supposons que $a \in T_a$, $b \in T_b$, $inf(B) = \{x | x \in T_a \wedge x < B\}$

et $sup(B) = \{x | x \in T_a \wedge x > B\}$ alors on obtient:

$$\begin{array}{lll} a = b & \rightsquigarrow & \{a = b \wedge a = min(T_a \cap T_b), \\ & & \ a = b \wedge a = max(T_a \cap T_b)\} \\ a > b & \rightsquigarrow & \{a = b + 1 \wedge b = min(T_b), \\ & & \ a = b + 1 \wedge a = max(T_a), \\ & & \ a = max(T_a) \wedge b = min(T_b)\} \\ a < b & \rightsquigarrow & \{a = b - 1 \wedge a = min(T_a), \\ & & \ a = b - 1 \wedge b = max(T_b), \\ & & \ a = min(T_a) \wedge b = max(T_b)\} \\ a \in B & \rightsquigarrow & \{a = min(B \cap T_a), \\ & & \ a = max(B \cap T_a)\} \\ a \notin B & \rightsquigarrow & \{a = min(inf(B)), \\ & & \ a = max(inf(B)), \\ & & \ a = min(sup(B)), \\ & & \ a = max(sup(B))\} \end{array}$$

Pour illustrer notre approche, considérons le résultat de la négation des préconditions de `Aload x` qui possède quatre contraintes que nous allons devoir tester: i) la pile ne doit pas être pleine, ii) le paramètre x appartient au domaine des variables locales, iii) le contenu de la variable locale à la position x doit être du type `Obj` et iv) le programme ne doit pas être terminé.

La première étape consiste à grouper les contraintes en sous-ensembles de clauses. Les seules contraintes que nous avons à grouper ici sont la $2^{ème}$ et la $3^{ème}$. Nous obtenons donc trois sous-ensembles de clauses: i, (ii et iii), iv.

Commençons par traiter le cas i de la pile non pleine. Nous obtenons deux branches à tester. La première correspondant à une taille de pile supérieure strictement à la taille maximum autorisée pour une pile normale. Ce cas est non-satisfaisable et éliminé, car la taille de la pile doit être inférieure ou égale à sa taille maximale. La deuxième branche de test correspond au cas où la taille de la pile est égale à sa taille maximale. Ce cas là est satisfaisable, il est donc conservé et seras noté a).

Pour le cas (ii et iii), si x appartient bien au domaine des variables locales, il y a deux solutions possibles pour la négation sur le type de la variable locale à la position x: `Uobj`, `Int`. Il faut donc tester ces deux cas, car ils sont satisfaisables et que nous noterons aa) et aaa). Ensuite il y a les cas où le type est respecté mais où x n'appartient pas au domaine des variables locales. Ce cas-là est non-satisfaisable, de même que le dernier cas où le typage et l'appartenance ne sont pas respectés.

Finalement, pour le cas iv où le programme ne doit pas être terminé, sa négation est que le programme doit être terminé. Il est reprénté par av).

C. Génération du préambule

Maintenant que nous connaissons l'état dans lequel nous voulons positionner la machine, il nous faut trouver un chemin permettant d'y accéder. Pour cela, nous allons partir de l'état initial du BCV et simuler son exécution afin de trouver une suite d'instructions valides conduisant à l'IST. Il y a trois possibilités dans cette recherche: i) on trouve un chemin; le problème est résolu; ii) on ne trouve pas de chemin après avoir parcouru tout le graphe de transition; il n'y a donc pas de solution; iii) la recherche ne termine pas dans un temps raisonnable: le graphe de transition est trop grand pour être parcouru en entier; il existe peut-être une solution, mais elle ne peut être atteinte dans un temps raisonnable, ou bien il n'existe pas de solution.

La génération du préambule de l'instruction `Aload x` s'applique sur les quatre branches de test que nous avons trouvé précédemment.

Pour la branche a), il nous faut un `Obj` dans les variables locales que l'on pourra retrouver à la position x. Cette contrainte peut être valide dès l'initialisation de la machine. Ensuite, il faut trouver une suite d'instructions telle que la taille de la pile soit égale à sa taille maximum. On peut simplement exécuter l'instruction «`push 0`» jusqu'à atteindre ce cas. Pour les branches aa) et aaa), il suffit de mettre les bons éléments dans les variables locales dès l'initialisation. Pour la branche av), il suffit d'initialiser la machine telle que l'une des variables locales soit un élément de type `Obj`, puis d'exécuter l'instruction «`Halt`» qui est disponible dès l'initialisation de la machine.

D. Génération du postambule

Dans le but de ramener le programme dans un état final, il faut, comme pour le préambule, trouver une suite d'instructions d'un état donné jusqu'à l'état final. La seule chose que l'on connaît du système est son état à un moment donné.

Pour l'instruction `Aload x`, nous avons quatre cas différents et donc quatre postambules différents. Pour ce faire, il faut donc calculer l'état de la machine après exécution de l'IST pour déterminer les instructions qui nous permettent d'arriver à une fin de programme correct. De même que précédemment, nous avons quatre cas.

Pour la branche i) où l'on a une pile pleine avant exécution de l'instruction `Aload x`. Après son exécution, la taille de la pile dépasse sa valeur maximale définie dans la spécification. Il faut donc trouver si on peut exécuter d'autres instructions par la suite. Dans ce cas, nous avons par exemple «`Pop`» qui nous permet de retirer un élément de la pile et dont les préconditions autorisent son exécution. Pour atteindre l'instruction terminale, il suffit de dépiler tous les éléments, ce qui peut se faire à l'aide de l'instruction «`Pop`». Ainsi, notre postambule consiste à exécuter l'instruction «`Pop`» tant qu'il reste des éléments et ensuite d'exécuter l'instruction «`Halt`».

Pour les deux branches ii) et iii) où l'on empile un élément qui n'est pas de type `Obj`, on peut les traiter de la même façon que précédemment en exécutant l'instruction «`Pop`» jusqu'à obtenir une pile vide et enfin exécuter l'instruction «`Halt`».

Finalement, pour la branche iv) où le programme est terminé à la fin du préambule, on exécute ensuite l'instruction

`Aload x`. Il devient alors impossible de trouver un postambule valide non-vide, puisqu'il est interdit d'exécuter une instruction suite à un «`Halt`». Le postambule est donc vide dans ce cas.

IV. Implémentation

Dans cette section, nous appliquons la stratégie présentée à la section III en utilisons la méthode Event-B [9] pour spécifier le système et l'outil ProB [10] pour générer les cas de test.

A. Modélisation

Chaque instruction a été modélisée comme un événement. Les modifications qu'elle apporte sur la pile ou sur les variables locales sont effectuées grâce à ses actions. Pour les contraintes, elles sont modélisées comme des gardes. Pour permettre un traitement simple des différents sous-ensembles de clauses, chacun d'eux est modélisé dans une seule garde.

Nous devons stocker certaines informations sur le type des éléments que manipulent les instructions. Nous avons expliqué précédemment que nous disposions d'un treillis de type. Ce treillis n'est en fait pas modélisé, car il ne nous apporte aucune information intéressante pour les tests que nous cherchons à trouver. Le typage a été simplement modélisé comme un ensemble $TYPE$ contenant tous les types possibles, à savoir `Int`, `Obj` et `Uobj`. En revanche, le treillis est présent dans le mécanisme de négation des gardes, affin de déterminer si un test doit être gardé.

Pour stocker l'état du système, nous avons deux structures de données, à savoir les variables locales et la pile. À cela nous ajoutons la variable booléenne *halt* qui indique que le programme est terminé ou non.

La structure des variables locales est un vecteur de taille fixe et dès l'initialisation elle contient des éléments. De plus, son contenu ne sera jamais vide durant toute l'exécution du programme. Nous l'avons donc modélisé comme une fonction totale allant d'un ensemble d'entiers naturels de taille fixe, dans $TYPE$, qui dénote les éléments du treillis de types.

Pour la pile, nous savons qu'il s'agit d'une simple pile de taille fixe qui contient des $TYPE$ et qu'à l'initialisation elle est vide. Nous l'avons modélisée comme une fonction partielle allant d'un ensemble d'entiers naturels de taille fixe, dans $TYPE$.

Regardons maintenant en détail comment modéliser les gardes. Pour savoir si notre modélisation est correcte, il faut considérer que les gardes doivent être construites pour être niées à un moment donné tout en respectant la spécification dans le cas contraire. La spécification impose certaines contraintes, soit sur la pile, soit sur les variables locales, que nous allons devoir reporter dans notre modèle. Il faut aussi rajouter une garde permettant de savoir quand le programme est terminé et ainsi interdire l'exécution d'autres instructions.

Nous avons tout d'abord les contraintes correspondant au typage de la pile. La négation d'une telle contrainte, imposant que l'élément en sommet de pile soit d'un type donné, conduit à deux branches de tests. La première donne des cas où on possède bien un élément, mais qui n'est pas du type voulu. Dans ce cas, il suffit de remplacer ce type par ses compléments dans le treillis de type. Dans la deuxième branche, nous avons à tester le cas où la pile est vide. Nous voyons par rapport à notre technique de négation des gardes que nous nous retrouvons dans le cas d'une conjonction dans un seul sous-ensemble de clauses, auquel on a supprimé le cas $\neg c_1 \wedge \neg c_2$. Ce dernier cas est en fait éliminé par le modèle, car il est impossible d'avoir une pile vide contenant un élément d'un type donné. La contrainte sur le typage en sommet de pile est donc modélisée comme une seule garde contenant la contrainte sur le type de l'élément en sommet de pile et que la pile n'est pas vide. Par exemple : $card(dom(stack)) > 0 \wedge (card(dom(stack)) \mapsto Int) \in stack$.

Le deuxième type de structures de données sur lequel sont imposées des contraintes est les variables locales. Pour le typage de celle-ci, nous n'avons pas le problème des variables locales vides, car nous savons que cela n'arrivera jamais. En effet, nous partons du principe que le vérificateur de structure fait correctement son travail. Comme pour la pile, il suffit de remplacer ce type par ses compléments dans le treillis de type. Comme la position à laquelle doit être lue le type est passée en paramètre, il faut aussi vérifier que ce paramètre appartient bien au domaine des variables locales. Nous devons tester un typage faux ou référer à un indice en dehors du vecteur des variables locales. Nous voyons par rapport à notre technique de négation des gardes que nous nous retrouvons encore dans le cas d'une conjonction, mais cette fois-ci le cas $\neg c_{1_1} \wedge \neg c_{1_2}$ est intéressant. La contrainte sur les variables locales est donc modélisée comme une seule garde. Par exemple : $x \in dom(varLoc) \wedge varLoc(x) = Obj$.

Pour s'assurer qu'aucune autre instruction n'est exécutée après la fin du programme on utilise une variable booléenne *halt* représentant cette information. Ainsi toutes les instructions doivent posséder cette garde. L'instruction terminale va simplement mettre à `vrai` cette variable pour dire que le programme est terminé. De cette façon, nous obtenons un interblocage permettant d'arrêter les recherches dans une telle branche si la solution n'a pas été trouvée. Nous avons donc pour toutes les instructions : $halt = 0$.

Aux contraintes imposées par la spécification, il faut rajouter d'autres gardes permettant d'avoir un modèle prouvé et que l'on peut animer avec ProB. Bien entendu ces gardes ne devront pas faire l'objet de tests. Pour cela, Event-B possède un système d'étiquetage des gardes que nous allons pouvoir réutiliser pour identifier celles imposées par la spécification et celles rajoutées pour le bon fonctionnement de modèle.

Prenons l'exemple de l'instruction `Aload x`:

event Aload_x
  any x
  where
    @grd1 $halt = 0$
    @grd2 $card(dom(stack)) < stack\_size\_max$
    @grd3 $x \in dom(varLoc) \land varLoc(x) = Obj$
  then
    @act1 $stack := stack \Leftarrow \{card(dom(stack)) + 1 \mapsto varLoc(x)\}$
  end

B. Génération concrète des cas de tests

Cette génération vas se dérouler en six étapes : i) négation des préconditions, ii) détermination d'un préambule, iii) création de l'IST par raffinement, iv) détermination du contexte après exécution de l'IST, v) détermination du postambule, vi) compilation et envoi de la trace pour le système sous test.

La négation des préconditions sera implémentée par un pluggin Rodin (Éclipse) qui doit parcourir le modèle et générer une liste de buts à atteindre pour exécuter les IST. Il met en application ce qui a été expliqué dans la section III-B.

Ensuite, chacun de ces buts est transmis à ProB par l'intermédiaire de son interface dans Rodin, dans le but de trouver un préambule. Cela nous permet de placer la machine dans le contexte souhaité pour effectuer le test.

Si on arrive à trouver un préambule, on raffine les événements (instructions) qui ont besoin de ce contexte. On supprime la garde dans le raffinement, afin d'exécuter l'événement de l'IST. Le contexte assure que nous sommes à l'IST. Pour obtenir le nouveau contexte après exécution de l'IST, il nous suffit d'exécuter l'instruction qu'on vient de créer par raffinement.

À partir du contexte obtenu, nous cherchons avec ProB un postambule valide. Si ProB nous en trouve un alors nous avons une trace nous permettant d'effectuer le test.

Pour que la solution soit testable sur un vrai système, il faut ensuite exporter la trace, la compiler et la charger pour la cible à tester. Pour ce faire, nous avons développé deux librairies pour carte à puce, l'une permettant de recompiler les fichiers CAP et l'autre pour charger les programmes invalides tout en gérant le protocole d'authentification mutuelle. Pour le moment, ces deux outils ne sont pas utilisés, mais lorsque nous passerons à Java Card, ils nous permettront de rapidement arriver à une exécution entièrement fonctionnelle.

V. Conclusion

Cet article présente nos travaux en cours sur la génération de tests de vulnérabilité pour les BCV de Java Card. Nous avons proposé une méthodologie de génération de suites de tests, basée sur l'utilisation de méthodes formelles. Cette approche permet de déceler des vulnérabilités dans des programmes reliés à la sécurité et pas seulement à s'assurer de la conformité des programmes. De plus, notre approche permet d'identifier précisément la source de l'erreur d'implémentation en introduisent seulement une erreur par test.

À l'aide d'un modèle en Event-B des instructions du sous-langage choisi, de l'outil ProB et des différents programmes mis en place, nous pouvons obtenir la négation des préconditions, la construction des préambules et postambules pour chacune des instructions. Dans le but de valider notre approche, nous avons développé un vérifieur de notre sous-langage à tester dans lequel nous pouvons introduire des erreurs d'implémentation.

Pour le moment l'automatisation de certaines parties comme la négation des préconditions n'est pas complétée. Pour cela il nous reste à développer un programme qui prendra en charge le passage entre les six étapes de la génération des tests. Une fois tout cela fait, nous pourrons passer à la modélisation du langage Java Card. Quand nous passerons au langage Java Card, il sera peut-être nécessaire d'identifier d'autres stratégies de simplification.

References

[1] C. Aumüller, P. Bier, W. Fischer, P. Hofreiter, and J. Seifert, "Fault attacks on RSA with CRT: Concrete results and practical countermeasures," *Cryptographic Hardware and Embedded Systems-CHES 2002*, pp. 81–95, 2003.

[2] G. Piret and J. Quisquater, "A differential fault attack technique against SPN structures, with application to the AES and KHAZAD," *Cryptographic Hardware and Embedded Systems-CHES 2003*, pp. 77–88, 2003.

[3] W. Mostowski and E. Poll, "Malicious code on Java Card smartcards: Attacks and countermeasures," *Smart Card Research and Advanced Applications*, pp. 1–16, 2008.

[4] J. Iguchi-Cartigny and J. Lanet, "Developing a Trojan applet in a Smart Card," *Journal in Computer Virology*, vol. 6, pp. 343–351, November 2010.

[5] J. Dick and A. Faivre, "Automating the generation and sequencing of test cases from model-based specifications," in *FME'93: Industrial-Strength Formal Methods*. Springer, 1993, pp. 268–284.

[6] M. Satpathy, M. Butler, M. Leuschel, and S. Ramesh, "Automatic testing from formal specifications," 2007.

[7] S. N. Freund and J. C. Mitchell, "A type system for object initialization in the java bytecode language," vol. 21. New York, NY, USA: ACM, November 1999, pp. 1196–1250.

[8] L. Casset and J. L. Lanet, "How to formally specify the java bytecode semantivs using the b method," in *Proceedings of the Workshop on Object-Oriented Technology*. London, UK: Springer-Verlag, 1999, pp. 104–105. [Online]. Available: http://portal.acm.org/citation.cfm?id=646779.704976

[9] Http://www.event-b.org.

[10] Http://www.stups.uni-duesseldorf.de/ProB.