Hauptseminar

Nachweis von Sicherheitseigenschaften für JavaCard
durch approximative Programmauswertung
WS 2001 / 2002
Verantstalter: Prof. Tobias Nipkow, Martin Strecker

Thema

Sichere Interaktion von SmartCard-Applets

Bearbeitet von

Clovis Yemou
yemou@in.tum.de

28 Februar 2002

Zusammenfassung

Dieser Seminarbeitrag beschreibt einen Ansatz, mit dem ein auf einer Javacard neu zu ladendes Applet zertifiziert werden kann. Die von uns angebotenen Sicherheitsprüfungen ergänzen die bereits auf der Karte bestehenden Sicherheitsfunktionen. Der $firewall$[7]kontrolliert eine sichere Interaktion zwischen zwei Applets während unsere Analyse eine globale Sicht erlaubt und die Aufspürung unerlaubter Informationsflüsse zwischen mehreren Applets ermöglicht. Für unsere Fallstudie haben wir den elektronischen Geldbeutel von Gemplus ausgewählt. Dieser Geldbeutel besteht aus einer Java-basierten Chip-Karte, auf der eine Protokoll definiert wurde, das Applet-Attributen und Applet-Methoden Sicherheitsstufen zuordnet und erlaubten Informationsflüsse zwischen den Sicherheitsstufen festlegt. Wir stellen eine auf $Model\ checking$ basierte Technik vor, mit der unerlaubte Informationsflüsse zwischen Applets aufgespürt werden können.

Im ersten Abschnitt stellen wir den multi-applikativen Kontext und die Fallstudie vor. Im zweiten erklären wir die auf der Karten existierenden Sicherheitsmechanismen und zeigen ihre Grenzen, mit dem Ziel, diese begrenzten Sicherheitsmechanismen zu ergänzen, indem wir unerlaubte Informationsflüsse zwischen Applets aufspüren. Ausgehend von Bytecode bauen wir zu diesem Zweck ein SMV-Modell auf und prüfen schließlich, daß das Modell die Sicherheitseigenschaften erfüllt. Die ausgewählte Sicherheitspolitik und die zugeordneten Sicherheitseigenschaften werden im dritten Abschnitt behandelt. Der vierte Abschnitt erklärt, wie man das SMV-Modell konstruiert und die Eigenschaften verifiziert.

Inhaltsverzeichnis

1 Kontext und Fallstudie **3**

1.1 Multiapplikative SmartCards 3

1.2 Fallstudie: Electronic Purse 3

2 Sicherheit und Java Card Sicherheitsmechanismus **3**

3 Multiapplikative Sicherheitspolitik **5**

3.1 Sicherheitspolitik 6

3.2 Sicherheitseigenschaften 6

4 Applet Zertifizierung **8**

4.1 Globale Analyse-Technik 8

4.2 Lokale Analyse-Technik 9

4.3 Beispiel-Analyse 12

5 Referenzen **13**

1 Kontext und Fallstudie

1.1 Multiapplikative SmartCards

Eine neue Generation von Chip-Karten kommt auf den Markt: die multi-applikativen SmartCards. Die Besonderheit solcher Karten besteht darin, daß es möglich ist, nach der Kartenerstellung Anwendungen auf die Karte nachzuladen und, daß viele beliebige Anwendungen auf derselben Karte durchgeführt werden können. Für solche multi-applikativen Karten wurden Betriebsysteme wie Java Card, Multos und vor kurzem $WindowsforSmartCards$ angeboten. Für uns ist An dieser Stelle nur $Java - Card$ vom Interesse. Nach dieser Norm werden die Anwendungen für multi-applikativen Karten in Form von Java-Applets programmiert.

1.2 Fallstudie: Electronic Purse

Ein klassisches Beispiel von multi-applikativen Karten ist der von Gemplus entwickelte elektronische Geldbeutel, der ein $purse$ Applet und zwei $loyalty$ Applets: ein Treuepunktesammel-Programm von Air France und ein ein Treuepunktesammel-Programm von einer Wagen-Verleihagentur (RentaCar). Das Basis-Applet $purse$ verwaltet sowohl Soll und Haben von dem Geldbeutel als auch eine Log-Datei von allen Transaktionen. Da in verschieden Währungen eingekauft wird (z.B. Francs und Euro) verwaltet das $purse$ Applet auch Konvertierungen. Ein Loyalty(Applet) wird je nach Benutzerwunsch auf die Karte geladen, die Treuepunkte (z.B. Miles fürs Air France Treuepunktesammel-Programm) je nach Einkauf gewährt. Das $loyalty$-Applet muß also in der Lage sein, mit dem $purse$-Applet Informationen auszutauschen. Ein $loyalty$-Applet will beispielsweise wissen, welche Transaktionen gemacht wurden, um die Anzahl seiner Treuepunkte zu aktualisieren. Auch vorgesehen ist die Möglichkeit von Abmachungen zwischen Treuepunktesammel-Programmen. Ein Air France Flugticket kann zum Beispiel mit RentaCar Punkten und Air France Meilen gekauft werden!
Wir haben für unsere Fallstudie den elektronischen Geldbeutel ausgewählt.

2 Sicherheit und Java Card Sicherheitsmechanismus

Die Frage der Sicherheit ist bei Chip-Karten ein sehr wichtiges Thema. Es ist aber bei multi-applikativen Karten um so mehr wichtiger. Dieser Typ von Karte bringt viele Teilnehmer ins Spiel: Den Hersteller der Karte (card issuer), den Anwendungsanbieter (application Provider) und den Benutzer (card holder oder user). Der card issuer ist in der Regel derjenige, der für die Sicherheit der erstellten Karte verantwortlich ist. Er vertraut den Anwendungsanbietern nicht: Die Anwendungen (Applets)können böswillig oder einfach falsch programmiert oder nicht authentisch sein.
Um das Interesse für das Thema zu wecken, möchten wir hier auf die Einschränkungen von klassischen Java Sicherheitsfunktionen eingehen. Die java Sicherheitsfunktionen wie der $Bytecode$ Prüfer oder der $Security\ manager$[1] wurden entworfen, um zu vermeiden, daß böswillige Applets die lokalen Ressourcen beschädigen. Diese Funktionen erzielen einen Schütz von Applets und Ressourcen vor böswilligen Applets.

Für eine mögliche Entwicklung von multi-applikativen Karten hat die JavaCard-Norm ein neues Mittel eingeführt, um eine sichere(indirekte) Interaktion zwischen Applets zu ermöglichen. Ein Applet kann durch eine gemeinsame genuzte Schnittstelle die Methode eines anderen Applets aufrufen. In unserer Fallstudie hat beispielsweise das $purse$-Applet eine gemeinsame Schnittstelle mit den $loyalty$-Applets, um ihnen die Ausführung ihrer Transaktionen zu ermöglichen und auch haben die Loyalty-Applets eine gemeinsame Schnittstelle um den Partner-Applets die Zuteilung ihrer Punkte zu gestatten. Da diese neue Art von Interaktion zwischen Applets jenseits des Anwendungsfeldes klassischer Java Sicherheitsfunktionen steht, hat JavaCard eine neue Sicherheitsfunktion namens $firewall$[7] definiert. Diese Sicherheitsfunktion sorgt dafür, daß nur die zu den gemeinsamen Schnittstellen gehörigen Methoden von den anderen Applets aufgerufen werden können.
Wir nehmen an, daß alle Java-Sicherheitsfunktionen und Java-Card im elektronischen Geldbeutel enthalten sind. Diese Funktionen decken jedoch nicht alle Bedrohungen, besonders die Bedrohung von unerlaubter Interaktion zwischen Applets bei multi-applikativen Karten. Die folgende Abbildung verdeutlicht eine solche unerlaubte Interaktion beim elektronischen Geldbeutel:

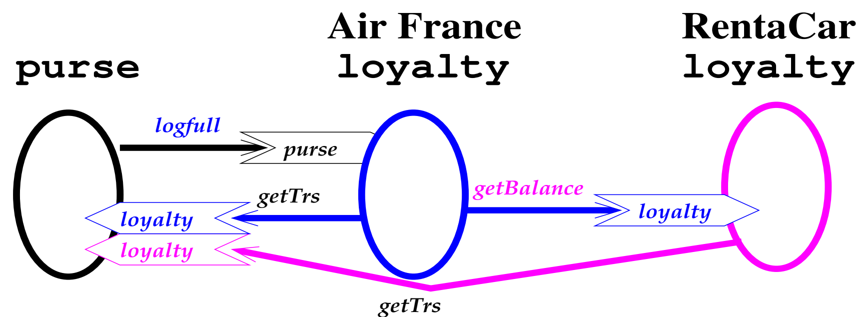

Abbildung 1: applet interaction

Das $purse$-Applet bietet den $loyalty$-applets einen “logfull”-Dienst an: wenn der Speicherplatz für die Speicherung aller Transaktionen voll ist, teilt das $purse$-Applet den den Dienst anfordernden $loyalty$-Applets mit, daß der Speicherplatz voll ist. Dadurch merken sie, daß sie ihre Transaktionen einfordern müssen, bevor diese durch neue überschrieben werden können. Wir nehmen an, das Air France-Applet hat einen “logfull”-Dienst angefordert, das RentaCar-Applet aber nicht. Wenn der Speicherplatz für die Speicherung aller Transaktionen voll ist, ruft das $purse$-Applet die Methode $logfull$ vom Air France-Applet auf. Dieses treibt mit **getTrs** vom Purse-Applet seine Transaktionen ein. Das Air France-Applet will aber vorher seine Punkte mit $getBalance$ vom RentaCar-Applet bei dem Partner (RentaCar)einholen. In diesem Fall checkt das RentaCar-Applet, daß der Speicherplatz voll ist und holt mit **getTrs** vom Purse-Applet seine Transaktionen ein.

Offensichtlich fließt (unerlaubt!) eine Menge bestimmter vertrauliche Informationen vom Air France-Applet zum RentaCar-Applet!
Der Java-Card Sicherheitsmechanismus verbietet zwar einem geladenen Applet Informationen und Ressourcen unerlaubt zu beobachten, zu verändern und zu benutzen aber die oben erwähnten unerlaubten Informationsflüsse werden nicht von der $firewall$[7] verboten, denn alle aufgerufenen Methoden gehören zu den gemeinsamen Schnittstellen. Ausgehend von bestimmten definierten Sicherheitseigenschaften wollen wir in der Lage sein, solche unerlaubten Informationsflüsse aufzuspüren , um sicherzustellen, daß es tatsächlich nur erlaubte Informationsflüsse zwischen neuen Applets und bereits geladenen Applets existieren! Die folgende Abbildung zeigt unsere Betrachtungsweise:

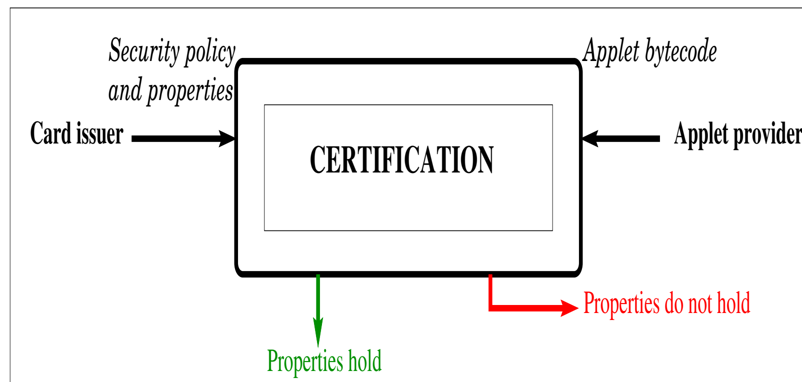

Abbildung 2: Applet Beglaubigung

Nehmen wir an, der Anwendungsanbieter möchte auf eine Karte ein neues Applet laden. Er liefert den Bytecode dieses Applets. Der Hersteller der Karte hat auf der Karte eine Sicherheitspolitik (security policy) und Sicherheitseigenschaften definiert, die erfüllt werden sollen. Wir bieten Tools und Techniken an, die dem Hersteller der Karte ermöglichen, zu entscheiden, ob Sicherheitseigenschaften von dem neuen Applet erfüllt werden (Diese Techniken basieren auf dem Bytecode und setzen das "Model checking" ein). Das Applet kann nur geladen werden, wenn die Eigenschaften erfüllt sind, ansonsten wird es abgewiesen.

3 Multiapplikative Sicherheitspolitik

Um diese Vorgehensweise zu verwirklichen, müssen wir zunächst eine zu den multiapplikativen Karten passende Sicherheitspolitik und die dazugehörigen Sicherheitseigenschaften definieren.

3.1 Sicherheitspolitik

Wir setzen die mehrstufige Sicherheitspolitik ($multilevel\ security\ policy$[2])ein, die für die multi-applikativen Karten entworfen wurde. Jedem Anbieter von Applets wird eine Sicherheitsstufe zugeordnet und den gemeinsamen Daten eine besondere Sicherheitsstufe. Im Geldbeutelbeispiel gibt es eine Sicherheitsstufe für jedes Applet: AF fur Air France, P für $purse$ und RC für RentaCar, und Sicherheitsstufen für die gemeinsamen genutzten Daten: AF+RC für Daten, die Air France und RentaCar gemeinsam haben, AF+P für die von Air France und $purse$, usw. Es wird eine Ordnung $\preceq$ eingesetzt, um Informationsflüsse zwischen Applets zu erlauben oder zu verbieten. Bei der eingesetzten Sicherheitspolitik gilt AF+P $\preceq$ AF und AF+P$\preceq$ P, d.h. Informationen können von AF+P nach P oder AF fließen. Das Air France-Applet und das $purse$-Applet können auf die Information zugreifen, die sie gemeinsam austauschen. Um die Tatsache zu modellieren, daß eine mögliche Appletkommunikation nur durch die teilbaren Schnittstellen gestatten ist, sind Informationsflüsse AF, P und RC nicht erlaubt.
Die Menge aller Sicherheitsstufen mit der Ordnung $\preceq$ (L,$\preceq$) bilden eine " lattice" (Verband)-Struktur: Obere Stuffen *private* und untere Stufen *public*!

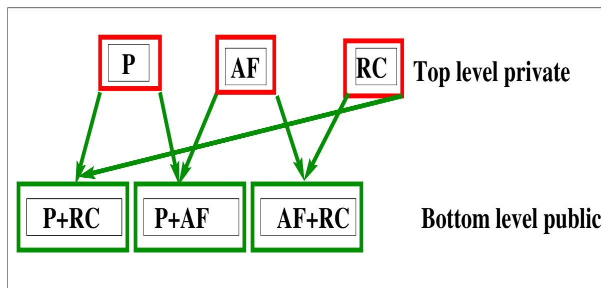

Abbildung 3: Verband-Struktur

3.2 Sicherheitseigenschaften

Nun müssen wir Sicherheitseigenschaften definieren, die erfüllt werden müssen. Wir haben das $Secure\ dependencies$(sichere Abhängigkeiten)-Modell[3]ausgewählt, das bei Systemen eingesetzt wird, wo böswillige Anwendungen anderen Anwendungen vertrauliche Informationen weitergeben könnten. Dieses Modell, wie andere Fluß-Modelle wie $non-interference$[4], stellt sicher, daß Abhängigkeiten zwischen Variablen nicht ausgenutzt werden können, um indirekte Kommunikationskanäle herzustellen. Wir setzen beim elektronischen Geldbeutel das Modell ein: unerlaubte Interaktionen werden aufgespürt, indem wir die Abhängigkeiten zwischen Systemvariablen überprüfen. Das Einsetzen dieses Modells erfordert drei Schritte. Man muß zunächst den System-

Variablen Stufeninformationen hinzufügen. Dann kann man also die Sicherheitseigenschaft ausdrücken, die erfüllt werden muß: der Wert einer Variable mit der Stufe l ist nur vom Wert der Variablen mit Stufe kleiner gleich l abhängig. Im zweiten Schritt definieren wir ausreichende Bedingungen, mit denen die Eigenschaft erfüllt ist (solche Bedingungen erleichtern das Prüfen mit Werkzeugen wie das $model\ checking$). Wir stellen schließlich fest, daß diese ausreichenden Bedingungen nicht vom Wert der System-Variablen abhängig sind, sondern nur von ihrer Sicherheitsstufe. Es genügt dann, durch eine "Systemsabstraktion", wobei die Variablenwerte durch ihre Stufen ersetzt werden, die definierten Bedingungen zu überprüfen. Wir wollen nun auf die drei Schritte ausführlich eingehen.
Wir unterscheiden drei Type von Variablen: Die Eingabe-Variablen, deren Werte nicht vom System abgerechnet werden, die Ausgabe-Variablen, die vom System abgerechnet werden und direkt beobachtbar sind und interne Variablen, die nicht beobachtbar sind. Nachdem wir allen Eingabe- und Ausgabe-Variablen eine Sicherheitsstufe zugeordnet haben, müssen wir schließlich folgende Bedingung überprüfen: Der Wert einer Ausgabe-Variable mit der Stufe l ist nur von den Werten der Variablen, deren Stufe kleiner gleich l, ist abhängig (gem. der Ordnung $\preceq$). D.h. jedes Paar von Programmausführungen, die mit den gleichen Werten für alle Eingabevariablen mit Stufe kleiner gleich l anfangen, muß den gleichen Wert für die Ausgabevariablen mit Stufe l berechnen. Mit den bekannten Prüfwerkzeugen ist diese Eigenschaft nicht einfach zu prüfen, deswegen wollen wir hinreichende Bedingungen finden, die mit Werkzeugen wie der $modell\ checker$ leicht einzusetzen sind.
Für jede Programmvariable ist es einfach, die Menge der Variablen zu bestimmen, von denen sie synktaktisch abhängig ist. Da die Werte dieser Variablen vom Programm abgerechnet werden, genügt es, zu zeigen, daß eine Variable mit der Stufe l nur von den Variablen mit der Stufe kleiner gleich l syntaktisch abhängig ist.
Diese Eigenschaft kann jedoch die internen Variablen verwenden. Da es nicht immer möglich ist, solchen Variablen eine Stufe zuzuordnen, definieren wir für eine Variable den Begriff "$abgerechnete\ Stufe$" einer Variable. Die abgerechnete Stufe einer Eingabevariable wird als ihre Sicherheitsstufe definiert. Ist die Variable keine Eingabevariable, ist die abgerechnete Stufe die obere Schranke der abgerechneten Stufen aller Variablen, von denen sie synktaktisch abhängig ist. Um die Sicherheit zu prüfen müssen wir lediglich zeigen, daß die abgerechnete Stufe einer Ausgabevariable in jedem Programmzustand kleiner gleich ihrer Sicherheitsstufe ist.
Die Stufen von den Systemvariablen kommen nicht in der neuen Bedingung vor, und da wir für die Prüfung der Sicherheitseigenschaften das Einsetzen von $model\ checker$ anstreben, ist es wichtig, die Größe des Wertebereichs zu reduzieren, um Probleme mit kombinatorischem Wachstum zu vermeiden. Um die Sicherheit zu prüfen, genügt es, zu zeigen daß die vorige Eigenschaft im jedem abstrahierten Programmzustand erfüllt ist, wo die Werte der Variablen durch ihre abgerechneten Stufen ersetzt werden.

4 Applet Zertifizierung

4.1 Globale Analyse-Technik

Die folgende Abbildung zeigt die möglichen Methodeninteraktionen, die sich im Allgemeinen auf einer multi-applikative Karte befinden:

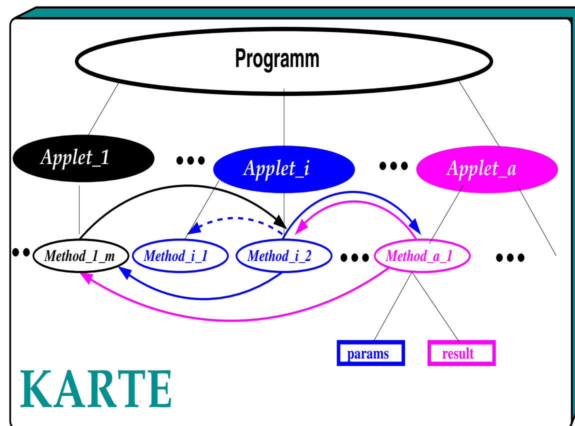

Abbildung 4: Methodeninteraktion

Die Umsetzung unserer Vorgehensweise wirft in der Praxis zwei Fragen auf und zwar zu welchen Variablen können wir eine Sicherheitsstufe zuordnen? Und welches Programm betrachten wir? (Eine Methode eines Applets, alle Methoden eines Applets, alle Methoden aller sich auf der Karte befindenden Applets?)
Wir sind in der Lage, zu den Attributen eines Applets und zu den Methodenaufrufen zwischen Applets Sicherheitsstufen zuzuordnen. In Abwesendheit ordnen wir allen Attributen aus dem Air France (bzw. Purse und RentaCar)Applet die Stufe AF (bzw. P und RC). Da Air France die Methoden $getbalance$ oder $debit$ von RentaCar aufrufen darf, ordnen wir diesen Interaktionen die Sicherheitsstufe RC+AF. Ebenfalls, da das Purse Applet die Methode $logfull$ von Air France aufrufen darf, ordnen wir dieser Interaktion die Sicherheitsstufe AF+P zu. Schließlich, ordnen wir zu dem Aufruf der vom Air France Applet (bzw. vom RentaCar Applet)aufrufbaren Methode $getTransaction$ des $Purse$-Applets die Sicherheitsstufe P+AF (bzw. P+RC) zu.
Wir haben uns für die Betrachtung eines Methodenaufrufs entschieden. Mit folgender “assume-guarantee”-Überprüfung prüfen wir lokal alle Methoden eines Applets ab ; auch, wenn Methoden anderer Applets von diese aufgerufen werden.
Die Methode $getBalance$ des RentaCar Applets wird beispielsweise von der Methode $logfull$ des Air france Applets aufgerufen. Wir analysieren getrennt die beide Methoden. Wir prüfen in der Methode $logfull$ des Air France Applets , daß die Stufen

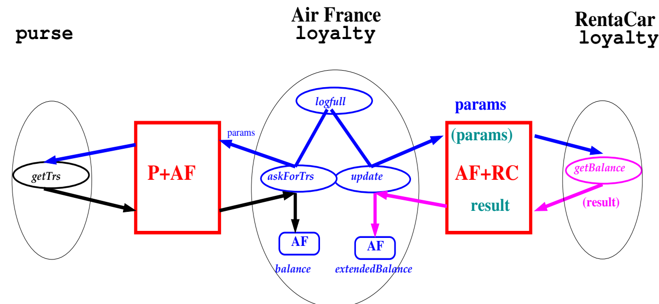

Abbildung 5: assume-guarantee

ihrer Parameter kleiner der Stufe dieser Interaktion (RC+AF) sind. Und wir nehmen an, RC+AF ist die Stufe von dem Ergebnis dieses Methodenaufrufs. Bei der Analyse der Methode $getBalance$ des RentaCar-Applets stellen wir sicher, die Stufe des Ergebnisses ist kleiner der Stufe der Interaktion und wir nehmen an, die Parameter haben die Stufe RC+AF.
Die gleiche Überprüfung gilt für Attribute innerhalb eines Applets. Wenn ein Attribut gelesen wird, nehmen wir an, daß seine Stufe die zugeordnete Sicherheitsstufe ist. Und bei Änderung des Attributes stellen wir sicher, daß die neue Stufe kleiner als die Sicherheitsstufe des Attributes ist. Mit der obigen "assume-guarantee"-Überprüfung kontrollieren wir , innerhalb eines Applets, zu einem bestimmten Zeitpunkt nur bestimmter Mengen von Methoden (nicht aber alle gleichzeitig).

4.2 Lokale Analyse-Technik

Ausgehend von Bytecode konstruieren wir ein SMV-Modell für jede Methode. An dieser Stelle wollen nicht auf eine ausführliche Beschreibung der SMV-Spezifikationen eingehen(siehe[5]), aber es wichtig zu wissen, daß eine SMV-Variable in zwei unterschiedlichen Arten definiert werden kann: entweder durch eine Gleichung ($var := expr$), oder durch eine Initialisierung und einen Ausdruck, die bei jedem Übergang ihren Wert für den nächsten Schritt angibt ($init() := val; next(var) := expr$).
Unsere Vorgehensweise für die Prüfung der Sicherheitseigenschaften, ausgehend von Bytecode, besteht aus drei Schritten:

- Abstraktion: die Variablenwerte werden durch ihre abgerechneten Stufen abstrahiert;
- Hinreichende Bedingung: wir prüfen ein "Invariant", das eine hinreichende Bedingung für die Sicherheitseigenschaft darstellt;
- model checking: das Invariant wird durch das "model checking" geprüft.

Wir wollen unsere Technik durch eine vereinfachte version der Methode $logfull()$ des Air France-Applets verdeutlichen. Diese Methode ruft direkt die Method $getbalance$ des RentaCar Applets auf und Aktualisiert das Attribut $ExtendedBalance$.

$Method\ void\ logfull()$
0 $aload_0$
1 $invokespecial_108$<$Method\ int\ getbalance()$>
4 $aload_1$
5 $aload_0$
6 dup
7 $getfield_220$ <$Field\ int\ ExtendedBalance$>
10 $iload_1$
11 $iadd$
12 $putfield_220$ <$Field\ int\ ExtendedBalance$>
15 $return$

Abstraktion wir modellieren den Bytecode der Methode $logfull$ durch ein $Modul$[5], das folgende Variablen enthält:

- pc: der Programm Zähler;
- $mem[i]$: Feld für Speicherplätze;
- $stack[i]$: Feld für Operandenstapel;
- sP: Stapelzeiger;
- $ByteCode$: Name der aktuellen Anweisung.

Die Variablenwerte werden durch Stufen abstrahiert. Die Stufen definiert man in einem Modul namens $Levels$, so daß eine Stufe durch einen booleschen Wert dargestellt wird. Die abstrahierten Variablen sind also vom Typ boolean oder ein boolesches Feld. Der Programmzählerwert wird nicht abstrahiert, er gibt die Reihenfolge der Anweisungen an. Ebenfalls, wird der Stapelzeigerwert beibehalten, er gibt den Index des nächsten freien Speicherplatzes an.

$L : levels;$
$pc : -1..9;$
$mem : array\ 0..1\ of\ boolean;$
$stck : array\ 0..1\ of\ boolean;$
$sP : -1..1;$
$ByteCode : \{invoke_108, load_0, return, nop, store\ _1, dup, load_1,$
$getfield_220, op, putfield_220\};$

Die Ausführung des Bytecodes fängt in der Zeile 0 des Programms an. Bei der Initialisierung ist der Stapel leer, liegt die Stufe der Parameter am Speicherplatz 0, der

die Stufe der Interaktion darstellt, die wir analysieren (d.h. AF+P). Die Stufe der Interaktion wird durch eine Konjunktion der Stufen $L.AF$ und $L.P$ codiert.
$init(pc) := 0; init(sP) := 0; init(mem[0]) := L.AF \, \& L.P;$
$for(i = 0; i < 2; i = i + 1)\{init(stck[i]) := L.AF \& L.P;\}$
Die Schleife definiert den Programmzählerwert und den Wert der aktuellen Anweisung. Sie ist eine direkte Übersetzung des Java Bytecodes. Die Ausführung ist am Ende, Wenn pc den Wert -1 hat und die aktuelle Anweisung nop ist, der gar nichts tut. Jede Anweisung, wie in[6], stellt einen "Java-Bytecode" Anweisung dar. Da wir uns für den Speichertyp und für den Speicherplatz nicht interessieren, stellt beispielsweise die Anweisung $load_0$ die Java Anweisungen (aload_i, iload_i, lload_i,...) dar. Ebenso werden die binären Operationen (iadd, ladd, iannd, ior, ...) durch die Anweisung op modelliert.

$(next(pc), ByteCode) :=$
$switch(pc)$ {
-1: $(-1, nop)$;
$0 : (pc + 1, load_0)$;
$1 : (pc + 1, invoke_108)$;
$2 : (pc + 1, store_1)$;
$3 : (pc + 1, load_0)$;
$4 : (pc + 1, dup)$;
$5 : (pc + 1, getfield_220)$;
$6 : (pc + 1, load_1)$;
$7 : (pc + 1, op)$;
$8 : (pc + 1, putfield_220)$;
$9 : (-1, return)$; };

Der folgende Teil des SMV-Modells beschreibt die Wirkung der Anweisungen auf die Variablen. Die Anweisungen rechnen für jede Variable die Stufen ab. Die Anweisung $load$ lädt die Stufe eines Speicherplatzes in den Stapel, Die Anweisung $store$ löscht die Spitze des Stapel und speichert diese Stufe in einem Speicherplatz, die Anweisung dup dupliziert die Stapelspitze. Die Anweisung op berechnet das Maximum der Stufen von zwei auf der Stapelspitze liegenden Speicherplätzen. Das Maximum zweier Stufen $l1 \vee l2$ wird durch die Disjunktion von diesen zwei Stufen $l1vl2$ modelliert. Die Anweisung $invoke$ stapelt die Parameter ab und stapelt das Ergebnis des Aufrufs auf. Nach der assum-guarantee Überprüfung, wird angenommen, daß das Ergebnis des Aufrufs der Methode $getBalance$ die Stufe L.AF $\wedge$ L.RC besitzt. Die Anweisung $getfield$ stapelt die Stufe des Attributs $ExtendedBalance$ auf, die hier L.AF beträgt. Schließlich stapelt die Anweisung $putfield$ die Stufe des Attributs $ExtendedBalance$ ab.

$switch(ByteCode)$ {
nop :;
$load_0$: $\{next(stck[sP]) := mem[\mathbf{0}]; next(sP) := sP - 1;\}$
$load_1$: $\{next(stck[sP]) := mem[\mathbf{1}]; next(sP) := sP - 1;\}$
$store_1$: $\{next(mem[1]) := stck[sP + 1]; next(sP) := sP\mathbf{+1};\}$
dup_0: $\{next(stck[sP]) := stck[sP + 1]; next(sP) := sP\mathbf{-1};\}$

op_0: $\{next(stck[sP+\mathbf{2}]) := stck[sP+1] \mid stck[sP+2]; next(sP) := sP+1;\}$
$invoke_108 : \{next(stck[sP]) := L.AF \,\&L.P; next(sP) := sP+\mathbf{1};\}$
$getfield_220 : \{next(stck[sP]) := L.AF;\}$
$putfield_220 : \{next(sP) := sP+\mathbf{2};\}$
$return$:;}

Invariant Wir haben schon erklärt, wie man die Stufe für jede Variable bestimmt und welche Sicherheitsstufen zu den Attributen und Interaktionen zugeordnet werden. Das Invariant, das wir prüfen wollen heißt dann, daß die abgerechnete Stufe der Variablen, die wir kontrollieren möchten, immer kleiner gleich der zugelassenen Stufe ist.
Bei der $logfull$-Methode werden wir uns für zwei Eigenschaften interessieren: Die eine prüft, ob die Interaktion zwischen $logfull$ und $getBalance$ korrekt ist und die andere sorgt dafür, daß $logfull$ das Attribut $ExtendedBalance$ nicht mißbraucht. Die Eigenschaft $Smethod$+_108 heißt, wenn die aktuelle Anweisung ein Aufruf der Methode $getBalance$ ist, muß die Stufe der übergegebenen Parameter (die Stapelspitze) kleiner als der Stufe der Interaktion AF+RC sein. Die Eigenschaft $Sfield$+_220 heißt, wenn die aktuelle Anweisung ein Änderung des Attributs $ExtendedBalance$ ist, muß die Stufe seines neuen Wertes (die Stapelspitze) kleiner gleich der Stufe des Attributs AF sein.
Da $logfull$ kein Ergebnis ausgibt, ist es nicht notwendig, die Eigenschaft $Sresult$, die besagt, die Stufe der Ausgabe (die Stapelsspitze) muß kleiner gleich der Stufe der Interaktion AF+P.

$Smethod+_108$
assertG(ByteCode=invoke_108->(stck[sP+1]->L.AF& $L.RC$));
$Sfield$+_220
$assert\ G\ (ByteCode = putfield$_220->(stck[sP+1]->$L.AF$));
$Sresult$
$assert\ G\ (ByteCode = return\text{->}stck[sP+1]\text{->}L.AF\ \&\ L.P)$);

4.3 Beispiel-Analyse

Nachdem wir das abstrahierte Modell und das Invariant gewonnen haben, setzen wir SMV[5] ein, um zu prüfen, daß das Modell von dem Invariant erfüllt wird. Falls die Eigenschaft nicht erfüllt ist, erzeugt der model checker ein Gegenbeispiel, das eine Ausführung des Bytecodes darstellt, die zu einem Zustand führt, bei dem die Eigenschaft nicht erfüllt ist.
Bei der Prüfung der $Smethod$_108 wird ein Sicherheitsproblem liegen: Die Interaktion $logfull$ zwischen $purse$ und $Air\ France$ hat die Stufe $P + AF$, der Kanal $getBalance$ die Stufe $AF+RC$. Offensichtlich hängt der Methodenaufruf $getBalance$ von dem Methodenaufruf $logfull$ ab. Es entsteht dann eine unerlaubte Abhängigkeit einer Variable mit Stufe $P+AF$ nach einer Variable mit Stufe $AF+RC$. Eine mögliche Lösung wäre das Verboten des Aufrufs anderer $Loyalty$-Methoden (z.B. $getBalance$) bei der Ausführung von der Methode $logfull$.

5 Referenzen

Literatur

[1] T. Lindholm and F Yellin. *The virtual Java Machine Specifications.* Adison Wesley, 1997.

[2] Pierre Girard. Which security policy for multiapplication smart cards? In USENIX *workshop on smartcard technology*, 1786.

[3] P. Bibier and F. Cuppens. A Logical View of Secure Dependencies. *Journal of Computer Security*, 1(1):99-129, 1992.

[4] J. Goguen and J. Meseguer. Unwinding and Inference Control. In IEEE *Symposium on Security and Privacy*, Aokland, 1984.

[5] K.L. McMilan. *The SMV Language.* Cadence Berkeley Labs, 1999.

[6] Pieter H. Hartel, Michel J. Butler, and Moshe Levy. The operational semantics of a Java secure processor. Technical report DSSE-TR-98-1, Declarative systems and sofware Engeneering group, University of Southampton, Highfield, Southampton SO17 1BJ, UK, 1998.

[7] Sun Microsystems. *Java Card 2.1 Realtime Environment (JCRE)Specifications.* Februar, 1999.