

Bytecode-Verifikation

Im Rahmen des Hauptseminars

Nachweis von Sicherheitseigenschaften für Java Card durch approximative Programmauswertung

Veranstalter: Prof. T. Nipkow, Dr. M. Strecker

Christian Pacher (pacherc@in.tum.de)

24. Januar 2002

Inhaltsverzeichnis

1. Einleitung.....	2
2. Die Java Virtual Machine.....	3
2.1 JVM Komponenten.....	3
2.1.1 Java Stack.....	4
2.1.1.1 Operandenstack.....	4
2.1.1.2 Variablensegment.....	5
2.1.2 Arbeitsspeicher.....	5
2.1.3 CPU.....	5
2.2 Java Bytecode.....	7
3. Bytecode Verifikation & die Sicherheitsarchitektur von Java.....	11
3.1 Bytecode Verifier.....	12
3.2 Class Loader.....	12
3.3 Security Manager.....	13
4. Bytecode Verifikation.....	15
4.1 4-Phasen Verifikation.....	15
4.2 Datenflußanalyse.....	16
4.2.1 Sun Spezifikation.....	16
4.2.2 on-card Lightweight Bytecode Verifikation.....	22
4.2.3 on-card Verifikation nach Leroy.....	24
5. Subroutinen.....	27
6. Literatur.....	28

1. Einleitung

Während zu Beginn des Internetzeitalters Webseiten hauptsächlich zur Bereitstellung von Informationen dienten, brachte die Einführung von Java-Applets wesentliche Veränderungen mit sich. Applets sind kleine Programme, die in Internetseiten eingebunden, Interaktivität und Funktionalitäten ermöglichen. Diese Programme werden über das Netz transportiert und anschließend lokal ausgeführt. Das Gefahrenpotential, welches hierbei in einer modernen Kommunikationsstruktur entsteht, ist ausgesprochen groß. Es bedarf einer geeigneten Sicherheitsarchitektur, welche den Quellcode der übertragenen Programme genauestes untersucht und deren „Gutartigkeit“ garantiert. Ansonsten wäre es einfach mittels dieser Applets, beispielsweise private Informationen auszulesen, die Stabilität des betroffenen Systems zu beeinträchtigen oder trojanische Pferde zu installieren.

Die Sicherheitsarchitektur der Java Virtual Machine versucht diesen Anforderungen gerecht zu werden. In dieser Ausarbeitung wird eine Komponente der JVM genauer beschrieben, der Bytecode Verifier.

Für das Verständnis des Java Bytecode-Verifikationsprozess ist es notwendig die Java Virtual Machine (JVM) in ihren wesentlichen Zügen zu kennen. Deswegen wird in Kapitel 2 eine kurze Erläuterung ihrer Funktionsweise gegeben.

Des weiteren ist es sinnvoll Bytecode Verifikation als elementaren Bestandteil der Java-Sicherheitsarchitektur zu verstehen. Diese ist mehrschichtig, und da erst das Zusammenwirken der verschiedenen Komponenten die erforderliche Sicherheit gewährleistet, werden auch diese Komponenten in Kapitel 3.2 und 3.3 erläutert.

In bezug auf Java Card ist die gewöhnliche Bytecode Verifikation in dem Sinne, wie sie für gewöhnliche Web-Applets durchgeführt wird, nicht realisierbar. Dies liegt in erster Linie an den hohen Speicheranforderungen der Verifikation. Java Cards bieten derzeit Speicherkapazitäten von wenigen Kilobytes. Dies ist nicht ausreichend für die „herkömmliche“ Bytecode Verifikation. Zwar verfügt die Java Card noch über eine weitere Speicherkapazität, das EEPROM, doch eignet sich dieses aufgrund der langsamen Schreibgeschwindigkeit nicht als Arbeitsspeicher für den Verifikationsprozess. Es werden spezielle abgewandelte Verifikationsalgorithmen vorgestellt, die mit ausgesprochen niedrigen Systemressourcen arbeiten, und sich damit auch für den Einsatz auf Java Cards eignen.

2. Die JVM

Die Java Virtual Machine ist eine abstrakte Stackmaschine. Stackmaschine bedeutet, dass die meisten Operanden von Bytecode Instruktionen aus einem Stack entnommen, und Ergebnisse dort wieder abgelegt werden. Zwar besitzt die JVM auch Register, aber werden diese nicht wie üblich direkt für beispielsweise arithmetische oder logische Operationen verwendet. Die JVM mit ihren Eigenschaften (Befehlssatz, Registersatz, Speicherverwaltung, usw.) wird durch Spezifikationen von SUN definiert. Die Realisierung einer JVM geschieht meist in Software, beispielsweise besitzen die Browser Internet-Explorer und Netscape jeweils eigene Implementierungen der JVM.

Die JVM besitzt einen Befehlssatz von ca. 200 Befehlen. Im wesentlichen handelt es sich dabei um:

- Befehle zur Stackmanipulation
- Sprungbefehle (bedingt und unbedingt)
- Speicher- / Ladebefehle für Variablen und Konstanten
- Lese- / Speicherbefehle von Feldern eines Objektes oder einer Klasse
- Befehle zum Aufrufen von Methoden
- logische & arithmetische Befehle

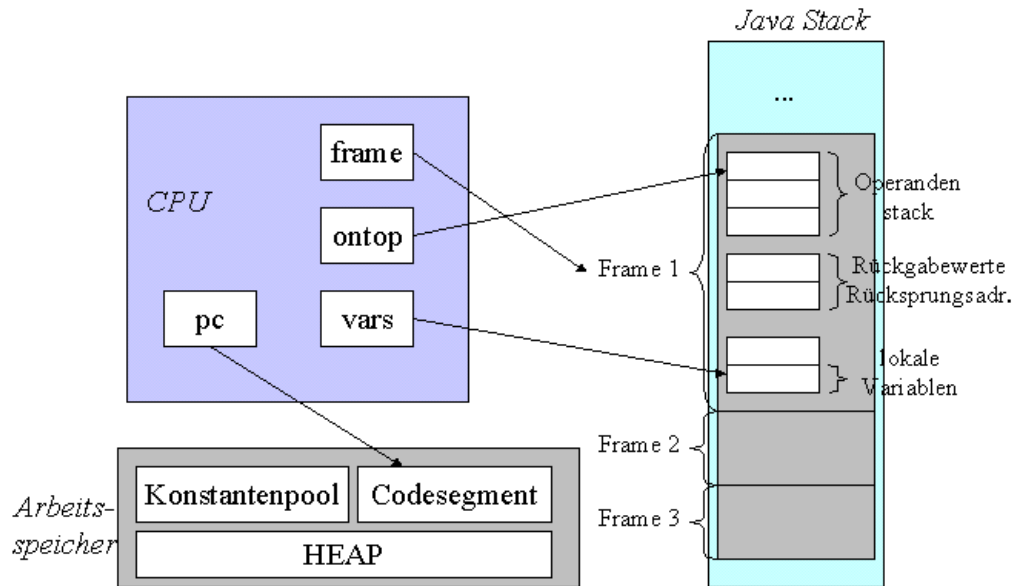
Der Maschinencode der JVM ist der sogenannte Java Bytecode. Dieser ist plattformunabhängig. D.h. Java-Compiler auf unterschiedlichen Plattformen müssen bei gleichem Quellcode semantisch äquivalenten Bytecode erzeugen. Führt nun eine plattformspezifische JVM auf einem System die Bytecodeinstruktionen aus, so erzeugt die JVM entsprechende Maschinenbefehle.

2.1 JVM-Komponenten

Die JVM läßt sich in drei Komponenten unterteilen,

- eine CPU,
- den Java-Stack und
- den Arbeitsspeicher.

schematischer Aufbau der JVM:



2.1.1 Java Stack:

Der Java Stack dient zur Verwaltung der einzelnen Frames. Ein Frame wird angelegt, wenn eine Methode gestartet wird. Er enthält die für die Methode relevanten Informationen (lokale Variablen, Methodenparameter, Execution Environment) und repräsentiert die Methode zu ihrer Laufzeit. Der „oberste“ Frame entspricht immer der aktuell ausgeführten Methode. Beim Verlassen einer Methode wird der entsprechende Frame wieder vom Java-Stack entfernt. Der Java Stack besteht aus 32-Bit breiten Einträgen. Datentypen von größerer Länge werden über mehrere Positionen gespeichert.

2.1.1.1 Operandenstack:

Betrachtet man einen Frame näher, so kann man zwei wesentliche Bereiche ausmachen: den Operandenstack und die lokalen Variablen.

Der Operandenstack ist einer der wichtigsten Bestandteile der JVM. Er ist nicht zu verwechseln mit dem Java-Stack selbst, vielmehr ist er ein dynamisch erzeugter Teil in ihm. Hier werden notwendige Operanden für Instruktionen geladen und Ergebnisse dieser abgelegt. Beim Start einer Methode und der entsprechenden Erzeugung des Frames ist der Operandenstack leer. Wenn im Verlauf der Ausführung der Methode beispielsweise die Instruktion `iadd` (Addition der beiden obersten Elemente des Operanden Stacks) ausgeführt werden soll, so ist offensichtlich, dass zuvor der Operandenstack mit dem für diese Instruktionen notwendigen Integerwerten geladen werden musste. Beim Verlassen der Methode bleibt auf dem Stack lediglich ein eventuell notwendiger Rückgabeparameter zurück.

2.1.1.2. Variablensegment:

Auch das Variablensegment ist Bestandteil eines Frames. Es dient zur Verwaltung lokaler Variablen. Werden Parameter an eine Methode übergeben, so werden diese in den lokalen Variablen gespeichert. Zur weiteren Verarbeitung können diese dann mittels Load-Anweisungen in den Operandenstack geschrieben werden. Store-Anweisungen speichern in die lokalen Variablen.

2.1.2 Arbeitsspeicher:

Codesegment:	Hier befinden sich u.a. die abzuarbeitenden Bytecodeinstruktionen der Methoden
Konstantenpool:	enthält sämtliche in der Methode vorkommenden Konstanten u.a. Typdefinitionen, Methodensignaturen, Namen von Klassen und Variablen...
Heap:	hier werden neu erstellte Objekte abgelegt. Die Speicherverwaltung wird von der JVM durchgeführt. Nicht mehr benötigte Objekte werden von dem Garbage Collector „entfernt“.

2.1.3 CPU

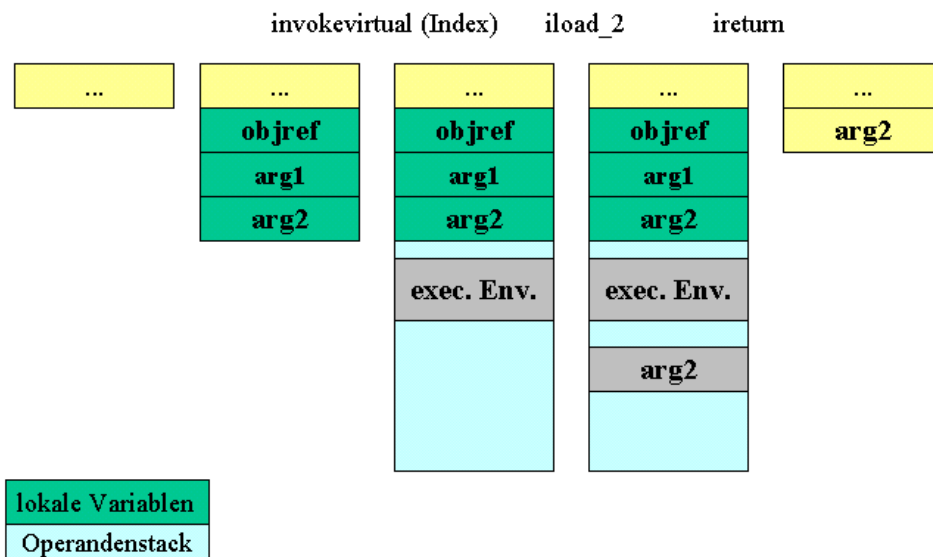
Register (32 Bit):

Ontop: Zeiger auf den obersten Wert des Operandenstacks

Vars: Zeiger auf die lokalen Variablen der aktuellen Methode

Frame: Zeiger auf den aktuell ausgeführten Frame

Pc: Zeiger auf den im Codesegment folgenden Befehl. Die JVM liest die Instruktion (das erste Byte) ein und erkennt anhand dieser, ob und wie viele weitere Bytes (ersichtlich durch die notwendigen Operanden) eingelesen werden müssen.



Es folgt ein kurzes Beispiel, welches die Funktionsweise der JVM beschreibt. Es handelt sich um den Aufruf einer Methode.

- Zu Beginn müssen verschiedene Werte auf den Java Stack gelegt werden.
 - Objektreferenz (gibt das Objekt von der Methode an, auf dem die Methode ausgeführt werden soll)
 - Argumente mit der die Methode aufgerufen wird
- Nun wird der Befehl `invokevirtual` aufgerufen, gefolgt von einem zwei Byte großen Verweis auf einen Record im Konstantenpool. Anhand der Methodensignatur die im Konstantenpool festgehalten ist, kann die gesuchte Methode in der Methodenliste gefunden werden. Ein neuer Frame wird für die eben aufgerufene Methode erzeugt, an der Spitze des Java Stacks. Das `vars`-Register wird zur 0ten lokalen Variable des Frames und ist eine Referenz auf das aktuelle Objekt. Die Argumente `arg1` und `arg2`

werden zur 1ten und 2ten lokalen Variablen. Anhand des Code Attributs der Methode wird ermittelt, wie viele Speicherslots für die weiteren lokalen Variablen benötigt werden. Diese werden freigegeben. Anschließend wird das frame-Register aktualisiert, und sämtliche Registerwerte in der sogenannten Execution Environment (siehe Beispiel) gespeichert, um später eine Rückkehr zu ermöglichen.

- Nun wird die erste Instruktion der Methode ausgeführt. Der Befehl `iload_2` lädt die 2te lokale Variable auf den Stack.
- In diesem Beispiel endet die Methode mit einem `ireturn`. Der Stack wird bis zur Objektreferenz abgebaut und die Registerwerte wiederhergestellt. Lediglich der gewünschte Integerwert der zurückgegeben werden sollte, verbleibt auf dem Stack.

Jetzt wird der Code der vorangegangenen Methode fortgeführt.

Werden zur Laufzeit eines Java Programms verschiedene Threads gestartet, so besitzt jeder Thread einen eigenen Java-Stack (also eigene Register und Operandenstacks). Alle Threads benutzen aber denselben Heap.

2.2 DER JAVA BYTECODE:

Java Bytecode wird i.d.R. durch einen Java-Compiler (beispielsweise `javac`) erzeugt. Grundlage ist hierfür der Java-Quellcode. Es sei hier allerdings bemerkt, dass aber noch weitere Möglichkeiten zu Bytecodeerzeugung existieren, beispielsweise mittels dem Bytecode-Assembler `Jasmin`. Diese Tatsache hat vor allen Dingen sicherheitsrelevante Konsequenzen. Bei Bytecode, der mittels einem Java Compiler erzeugt worden ist, kann man davon ausgehen, dass dieser „gutartig“ ist. `Jasmin` hingegen ist mächtiger in Bezug auf die Erzeugung von Bytecode und ist beispielsweise auch in der Lage fehlerhaften Code zu erzeugen. Diese Erkenntnis ist in Zusammenhang mit der Bytecode-Verifikation von besonderer Relevanz.

Der Byte Code einer .class Datei besteht aus 6 Bereichen

- Dateikopf
- Konstantenpool
- Klassenbeschreibung (Interfaces, Superklasse, Zugriffsrechte)
- Array von Datenfeldern
- Array mit Methoden (Attribut: u.a. Bytecode jeder einzelnen Methode)
- Zusätzliche Informationen (Attribut: u.a. Source File Attribut)

Dateikopf:

Hier befinden sich die Dateikennung 0xCAFEBAE und Versionsnummern

Konstantenpool:

In dem Konstantenpool sind sämtliche, in einer Klasse vorkommenden Konstanten abgelegt. Es existieren 12 verschiedene Typen, u.a. Integer, Float, Class, Methodref.

Datenfeld:

Hier werden datenfeldspezifische Informationen gespeichert

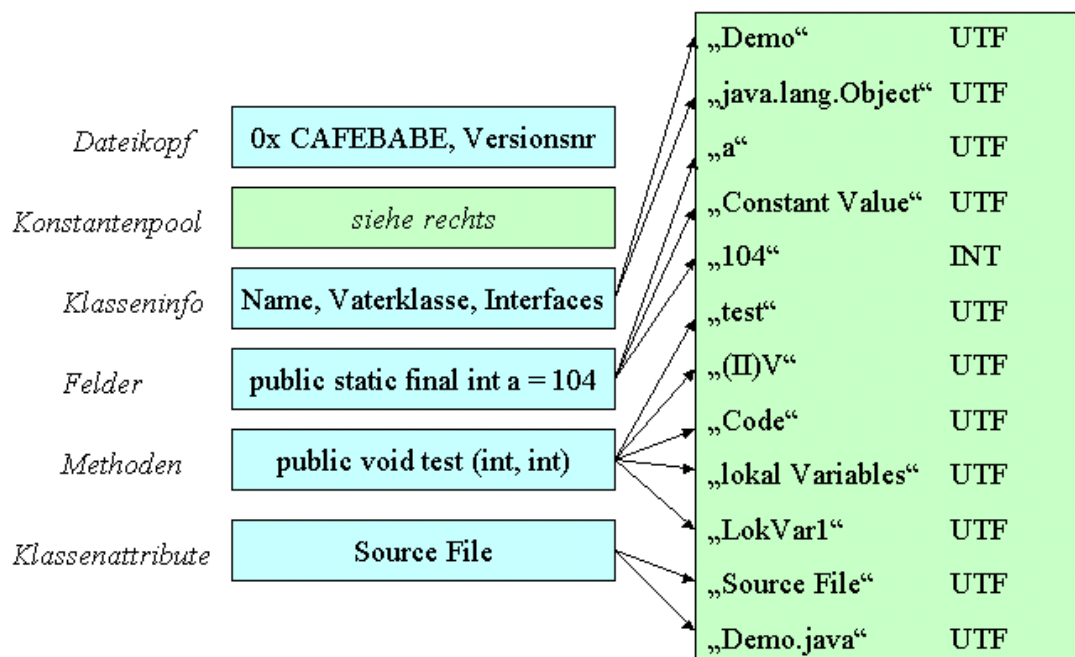
- Modifizierer (Zugriffsrechte)
- Name des Felds (Verweis auf Konstantenpool)
- Typ des Felds (Verweis auf Konstantenpool)
- Weitere Infos (Attribute)

Methoden

- Modifizierer
- Name (Verweis auf Konstantenpool)
- Signatur (Verweis auf Konstantenpool)
- Bytecode Sequenz (Code-Attribut)
 - Codename
 - Codelänge
 - Max Größe des Operandenstacks
 - Max Anzahl an lokalen Variablen

- Code (Array von Bytes) erstes Byte ist Befehlsnummer, für Operanden anschließend Verweise auf Konstantenpool
- Ausnahmetabelle
- Weitere Infos
- Weitere Infos

Das folgende Beispiel demonstriert in vereinfachter Form das Bytecode-Format einer Klasse Demo.java mit einer Auflistung von einigen Konstantenpooleinträgen. Wie bereits erwähnt, beinhaltet der Konstantenpool sämtliche, in der Klasse vorkommenden Konstanten, weshalb in den unterschiedlichen Bereichen Verweise auf den Pool zu finden sind.



Die Darstellung des Bytecodes erfolgt meist durch Repräsentation der Instruktionen von Methoden. Dies entspricht selbstverständlich nicht dem tatsächlichen Format des Bytecodes.

Das folgende Beispiel wurde aus einem Text „die virtuelle Maschine“ von R.Schaback, Georg-August-Universität Göttingen entnommen.

Wir sehen nach, wie Java das folgende Programm zur Berechnung eines Skalarprodukts übersetzt:

```
public static double skalp (double[] a, double[] b)
{
double sum=0.0;
for (int i=0; i<a.length; i++)
sum+=a[i]*b[i];
return sum;
}
```

Es handelt sich um eine statische Methode mit zwei Parametern. Deshalb befinden sich in der aufgerufenen Methode nach dem Aufruf die Referenzen auf a und b in den lokalen Variablen 0 und 1. Man sieht, daß die Variable sum in den lokalen Variablen 2 (und 3, wegen der Belegung von zwei Variablenindizes bei double- und long- Werten) gespeichert wird, während i in Variable 4 kommt.

Method double skalp(double[], double[])

Bytenummer und Befehl	Kommentar
0 dconst_0	holt double-Konstante 0.0 auf Stack
1 dstore_2	Speichert diese (= sum) nach Variable 2 (und 3), Stack leer
2 iconst_0	holt int-Konstante 0 auf Stack
3 istore 4	Speichert diese (= i) nach Variable 4, Stack leer
5 goto 23	Springt zum Schleifentest nach pc = 23
8 dload_2	legt sum aus Variable 2 (und 3) auf Stack
9 aload_0	holt Referenz auf a[] aus Variable 0 auf Stack
10 iload 4	legt i auf Stack über a[] und sum
12 daload	holt double-Wert von a[i] auf Stack, darunter liegt noch sum
13 aload_1	holt Referenz auf b[] aus Variable 1 auf Stack, darunter a[i], sum
14 iload 4	legt i erneut auf Stack, darunter b[], a[i], sum
16 daload	holt b[i] auf Stack, darunter a[i], sum
17 dmul	Multiplikation, danach ist a[]*b[], sum auf Stack
18 dadd	Addition, danach ist a[]*b[]+sum auf Stack
19 dstore_2	speichere das Ergebnis nach sum, Stack ist jetzt leer.
20 iinc 4 1	inkrementiere i um 1 in lokaler Variablen 4, Stack leer.
23 iload 4	Schleifentest: hole i auf Stack
25 aload_0	hole Referenz auf a[] auf Stack, darunter i
26 arraylength	ergibt a.length auf Stack, darunter i
27 if_icmplt 8	wenn i < a.length, dann nach pc=8 wegspringen, d.h. neuer Durchlauf
30 dload_2	es ist i >= a.length, deshalb weiter: sum auf Stack legen
31 dreturn	Rücksprung mit double auf Stack

3. Bytecodeverifikation & die Sicherheitsarchitektur von Java 1.0

Gefahren die von Applets ausgehen können wurden von Edward Felten und Gary McGray in folgenden Kategorien klassifiziert.

Art der Bedrohung:	Javas Abwehr:	Bsp.:
Systemveränderung	stark	Löschen von Daten
Verletzung der Privatsphäre	stark	Fälschen von emails
Lahmlegen von Systemressourcen	schwach	Prozessorblockierung
Belästigung	schwach	Anzeigen unerwünschter Bilder

Um Angriffe aus den ersten beiden Kategorien zu verhindern, wurde von SUN ein Sicherheitsmodell, die sogenannte Sandbox, eingeführt. Der Wirkungsbereich eines Applets wird durch die Sandbox eingeschränkt. Hier ist es nicht mehr in der Lage beispielsweise Dateien des ausführenden Systems auszulesen, oder Netzwerkverbindungen aufzubauen.

Das Sandboxmodell baut auf drei Ebenen auf.

- Byte Code Verifier:
Der gesamte auszuführende Code wird vor und während der Ausführung auf die Einhaltung der Sicherheitsregeln überprüft. Es findet eine genaue Formatüberprüfung sowie eine Daten- und Kontrollflussanalyse statt.
- Class Loader:
Der Klassenlader lädt Klassen über ein Netzwerk und stellt dabei sicher, dass beim Laden der Klassen keine Namens- oder Zugriffseinschränkungen verletzt werden.
- Security Manager

Zusätzlich ist Java selbst eine sichere Sprache. Es existieren keine Pointer, sondern nur Objektreferenzen. Pointer-Fehler sind eine häufige Ursache für die Verletzung von Sicherheitsregeln. Auch besitzt sie den sogenannten Garbage Collector (eine automatische Speicherverwaltung), der selbständig den nicht mehr dynamisch allokierten Speicher freigibt.

3.1 Byte Code Verifier

Der Byte Code Verifier stellt u.a. sicher, dass

- zur Laufzeit keine Zugriffsbeschränkungen übergangen werden
- Methoden mit zulässigen Argumenten initialisiert werden
- kein Stack Over- Underflow entsteht
- keine unzulässigen Datentypumwandlungen stattfinden

Der Bytecode Verifikationsprozess soll die „*Gutartigkeit*“ eines Bytecodes überprüfen. Schlägt die Verifikation einer Klasse fehl, so wird diese nicht mehr weiterverarbeitet. Diese Daten- und Kontrollflussanalyse ist ein aufwendiges Verfahren. Lokaler Programmcode der im CLASSPATH liegt und JDK Bytecode durchlaufen den Verifikationsprozess nicht.

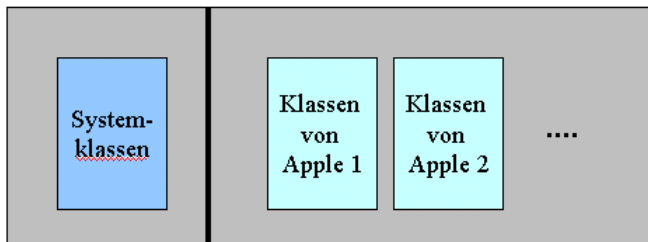
Die essentielle Notwendigkeit des Byte Code Verifiers demonstriert folgendes Beispiel:

Eine fehlerhafte Implementierung des BV in einer frühen Version des Internet-Explorers von Microsoft erlaubte eine unzulässige Typumwandlung von Double- oder Long-Werten in eine Referenz. Damit war es möglich, Zeiger umzusetzen und nicht erlaubte Operationen durchzuführen, oder Zugriff auf geheime Informationen zu erlangen.

3.2 Class Loader

Die Aufgabe des Class Loaders besteht darin, Klassen zu laden und dabei zu verhindern, dass es zu Namenskonflikten kommt. Beispielsweise darf ein Applet keine Systemklassen wie `java.lang.SecurityManager` durch eigene Definitionen überschreiben.

Auch muß verhindert werden, daß es zu Namenskonflikten unter Applets kommt. Kein Applet darf ohne die Mitarbeit des betroffenen Applets auf dessen Methoden oder Variablen zugreifen. Jedes Applet hat seinen eigenen Class Loader, der die dazugehörigen Klassen in getrennten Namensräumen installiert.



Der Class Loader ist nicht direkter Bestandteil der JVM, sondern eine Subklasse von `java.lang.ClassLoader`.

3.3 Security Manager

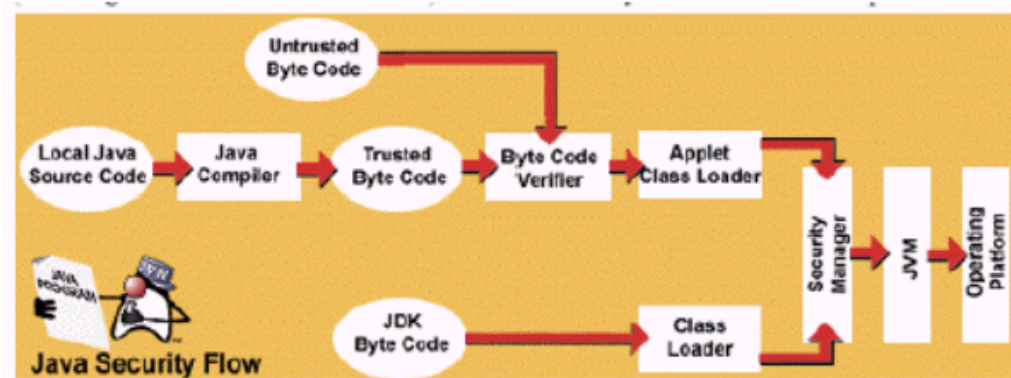
Der Security Manager ist eine abstrakte Klasse. Webbrowser erzeugen anhand dieser Klasse einen eigenen Security Manager und implementieren so ihre eigenen Sicherheitsstrategien. Da ein Browser immer nur einen Sicherheitsmanager besitzen kann, wird verhindert, dass ein Applet seinen eigenen Sicherheitsmanager installiert.

Der Sicherheitsmanager besitzt Methoden, die zur Laufzeit aufgerufen werden, um zu überprüfen, ob bestimmte Operationen in der aktuellen Umgebung zulässig sind.

Hierbei werden in der Regel folgende Operationen überprüft

- es dürfen keine lokalen Dateien erstellt, gelesen oder gelöscht werden
- es dürfen keine Verzeichnisse erstellt werden
- es dürfen keine Inhalte der Verzeichnisse dargestellt werden
- es dürfen keine Dateiattribute ermittelt werden
- es dürfen keine Netzwerkverbindungen aufgebaut werden
- es dürfen keine Systemeigenschaften gelesen werden
- es dürfen keine Betriebssystemaufrufe mit Hilfe von `Runtime.exec()` ausgeführt werden
- es darf der Java Interpreter nicht mit Hilfe von `Runtime.exit` beendet werden
- es dürfen keine DLLs mit Hilfe `Runtime.loadLibrary` geladen werden

- es dürfen keine Threads erzeugt oder manipuliert werden, die nicht Teil des Applets sind
- es dürfen keine Sicherheitsmanager und Klassenlader erzeugt werden
- es dürfen keine neuen Klassen erzeugt werden, die bereits Teil der lokalen Systemklassen sind



Die folgende Grafik demonstriert den Java Security Flow

Das Sicherheitsmodell von Java 2 unterscheidet sich in vielen Punkten von seiner Vorgängerversion. Während Applets in der alten Version entweder alle Rechte (sofern es sich um vertrauenswürdigen Code handelt) oder nur die Rechte der Sandbox besitzen konnte, lassen sich nun die Zugriffsrechte feiner einstellen. Im übrigen wird bezüglich der Sicherheitsstrategie nicht mehr zwischen Applets und Applikationen unterschieden.

Java Klassen werden nun bestimmte Schutzbereiche zugewiesen. Diese werden im Prinzip durch ihre Ressourcen definiert, auf die die Java Klasse augenblicklich zugreifen darf.

An dieser Stelle soll nicht weiter auf die Sicherheitsarchitektur der neueren Java Versionen eingegangen werden. Wichtig ist in diesem Zusammenhang viel mehr die Funktionalitäten des Security Managers, des Class Loaders und des Bytecode Verifiers verstanden zu haben.

4. Bytecode Verifikation

Bytecode Verifikation wird für jede Klasse eines Applets, mit Ausnahme der abstrakten Klassen, ausgeführt. Sie entspricht in etwa der Ausführung des Bytecodes mit Typen anstatt mit den tatsächlichen Werten.

4.1. 4 Phasen Verifikation.

Die Spezifikation von SUN beschreibt 4 Phasen in denen verschiedene Tests durchgeführt werden müssen.

1. Phase: beim Laden

Zu Beginn wird überprüft, ob der zu untersuchende Bytecode den grundlegenden Spezifikationen nach Lindholm & Yellin (The Java Virtual Machine Specification) entspricht. Hierunter fällt unter anderem die Überprüfung, ob zu Beginn der Datei die Kennung OxCAFEBAFE steht, ob die Datei vollständig ist und alle Attribute die richtige Länge besitzen.

2. Phase: beim Binden

In dieser Phase findet eine erweiterte Formatüberprüfung statt. Hier wird u.a. sichergestellt, dass als final deklarierte Klassen keine Subklassen besitzen, dass final Methoden nicht überschrieben werden, dass jede Klasse eine Vaterklasse hat, dass alle Konstantenpooleinträge korrekt formatiert sind, und dass alle Datenfelder und Methodenaufrufe im Konstantenpool auf gültige Namen und Typdeskriptoren verweisen.

3. Phase: während der Linkphase

Für jede einzelne Methode wird hier eine globale Daten- und Kontrollflussanalyse durchgeführt.

Dabei wird insbesondere überprüft ob

- Methoden mit den entsprechenden Parameter aufgerufen werden.
- Variablen vor Benutzung initialisiert worden sind.
- Felder nur Werte korrekten Typs zugewiesen bekommen.
- Stacküberläufe bzw. Stackunterläufe stattfinden.
- Der Operandenstack immer die gleiche Höhe aufweist, egal auf welchem Ausführungspfad man dorthin gelangt.
- Instruktionen immer mit passenden Operandentypen aufgerufen werden.

- Bei Sprunganweisungen nicht in die Mitte einer anderen Instruktion gesprungen wird, sondern immer an den Beginn.

Die Datenflussanalyse wird in Kapitel 4.2 genauer beschrieben.

4. Phase: während der Laufzeit

Aus Geschwindigkeitsgründen werden bestimmte Tests, die bereits in Phase 3 hätten durchgeführt werden können, erst in Phase 4 zur Laufzeit ausgeführt.

Bestimmte Typenkonflikte können zum Augenblick des Linkes nur sehr schwer oder gar nicht erkannt werden. Dies demonstriert folgendes Beispiel:

```
Void test(B[] x, B y) {  
    x[0] = y  
}
```

Man nehme an, A sei eine Unterklasse von B. Demnach ist A[] ebenfalls eine Unterklasse von B[]
Würde x nun vom Typ A[] sein und y vom Typ B, käme es zu einem Typenkonflikt, der während der Linkzeit nicht erkannt werden könnte, da die Typen zu diesem Augenblick noch nicht bekannt sind.

Es müssen also Tests während der Laufzeit stattfinden.

Auch wird hier überprüft ob:

- auf ein Feld oder eine Methode zugegriffen werden darf.
- das entsprechende Feld oder die Methode überhaupt in der dazugehörigen Klasse vorhanden ist.

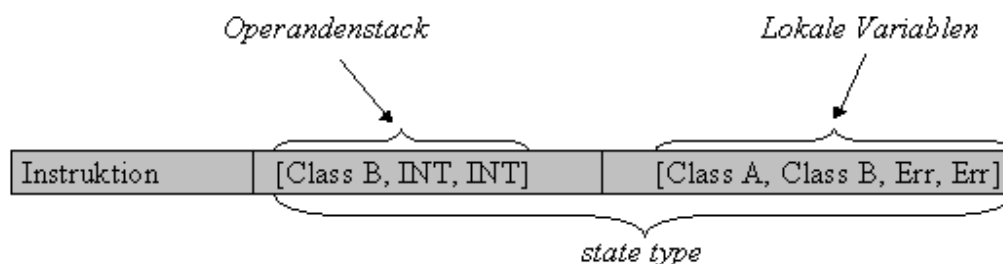
4.2 Datenflußanalyse

4.2.1 Suns Spezifikation

Ein iterativer Algorithmus, der all die für Phase 3 beschriebenen Aufgaben erfüllt, ist sehr komplex. Er wird nun in seiner grundlegenden Struktur beschrieben, ohne dass dabei beispielsweise die Behandlung von Long und Double Werten (sie belegen jeweils 2 Speicherzellen im Operandenstack und bei den lokalen Variablen und dürfen zur Laufzeit nicht auf unerlaubte Weise in zwei 32 Bit Werte gespalten werden), oder die Verifikation von Subroutinenaufrufen berücksichtigt wird. Jede Methode wird einzeln verifiziert.

Im folgenden wird der Begriff *state type* benutzt:

Ein *state type* beschreibt den einer Instruktion zugewiesenen Operandenstack und lokalen Variablen. Wenn eine Bytecodeinstruktion beispielsweise einen Integerwert vom Stack benötigt, so könnte der *state type* dieser Instruktion wie folgt aussehen.



1. Initialisiere den Operandenstack als leer und die lokalen Variablen mit den Typen der Parameter, die an die Methode übergeben werden sollen.
2. Nun markiere die erste Instruktion des Methoden Bytecodes. Der in Schritt 1 erzeugte *state type* von dem Operandenstack und den lokalen Variablen werden dieser Instruktion zugeordnet.
3. Suche eine markierte Instruktion aus. Überprüfe, ob der Operandenstack und die lokalen Variablen, die dieser Instruktion zugewiesen worden sind, von korrektem Typ sind. Liegt beispielsweise nur ein Wert auf dem Operandenstack und lautet die Instruktion *iadd*, so schlägt der Verifikationsvorgang fehl und die Methode wird zurückgewiesen.

Existiert keine markierte Instruktion, so wird der Verifikationsvorgang erfolgreich beendet.

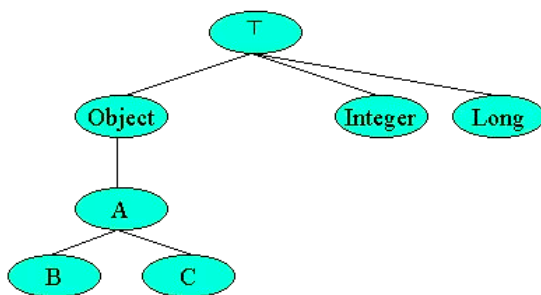
4. Wende die Instruktion nun auf die Operandentypen an. Nun verändert sich wahrscheinlich der Operandenstack (Bsp. durch iadd) oder die lokalen Variablen und es entsteht ein neuer state type. Dieser muss gespeichert werden.
5. Der neue state type muss nun mit der Nachfolgeinstruktion verknüpft werden. Hierbei muss wie folgt differenziert werden:
 - die Nachfolgeinstruktion ist tatsächlich die im Codesegment folgende
 - (unbedingter Sprung) die Nachfolgeinstruktion ist das Sprungziel einer goto-Anweisung
 - (bedingter Sprung) es gibt zwei Nachfolgeinstruktionen. Sprungziel und die nächste Instruktion
 - (exception handler) ist die Instruktion durch einen e.h. geschützt, dann ist die erste Anweisung des Ausnahmebehandlers eine Nachfolgeinstruktion
 - athrow und return Anweisungen besitzen keine Nachfolgeinstruktionen (es sei denn, sie sind durch einen Ausnahmebehandler geschützt)
6. Nachdem nun die Nachfolgeinstruktionen gefunden worden sind, muss der neu entstandene state type von Operandenstack und lokalen Variablen auf die Instruktion angewendet werden. Dabei gibt es zwei zu unterscheidende Fälle.
 - Die Nachfolgeinstruktion war noch nicht markiert:
Ordne den neuen state type dieser Instruktion zu und markiere diese Instruktion.
 - Die Nachfolgeinstruktion war bereits markiert:
das bedeutet, dass diese Instruktion bereits auf einem anderen Ausführungspfad besucht worden ist. Nun muss der bereits zugewiesene state type für diese Instruktion mit dem neu zuzuweisendem state type verglichen werden. Kombiniere die beiden state type zu einem. Ist nicht möglich, da sie nicht kompatible Typen besitzen, so muss die Methode zurückgewiesen werden.
Unterscheidet sich der durch die Kombination neu entstandene state type von dem alten, dann markiere Nachfolgeinstruktion.
Bleibt der state type auch nach Kombination der gleiche, so lösche die Markierung.
7. Gehe zu 3

In Schritt 6 wird zur der Einfachheit halber der Begriff „Kombiniere“ verwendet. Dies soll genauer erklärt werden:

Kombiniere bedeutet hier, dass für die Elemente des Operandenstacks und für die Elemente der lokalen Variablen der kleinste allgemeine Typ zu finden ist. Geht man davon aus, dass in einer Typenhierarchie eine partielle Ordnung herrscht, so lässt sich dieser kleinste allgemeine Typ leicht finden.

Eine partielle Ordnung ist eine Relation mit folgenden Eigenschaften:

- Transitivität
- Reflexivität
- Antisymmetrie



In der hier vorliegenden Baumstruktur der Typen ist beispielsweise der kleinste allgemeine Typ der Typen B und C der Typ A. D.h. diese Typen sind kombinierbar. Anders jedoch im Beispiel Integer und C. Diese beiden Typen sind nicht kompatibel. Der kleinste allgemeine Typ wäre der unerlaubte Typ τ . Stellt der Verifikationsprozeß fest, dass auf einen solchen Typen zugegriffen wird, schlägt die Verifikation fehl. Alle lokalen Variablen der Methode, die beim Anlegen des Frames noch keine Werte zugewiesen bekommen haben, werden mit dem Typ τ initialisiert. Dies verhindert den Zugriff auf „nicht überschriebene“ Register und damit das nicht erlaubte Auslesen von Informationen.

Im Endeffekt läuft der Algorithmus auf eine Fixpunktsuche hinaus. Der Fixpunkt ist dann gefunden, wenn eine Konstellation von state types gefunden worden ist, die sich nicht mehr ändern. Am Beispiel eines straight-line Codes (Code ohne Verzweigungen) ist der Fixpunkt nach einem Durchlauf gefunden, da die Instruktionen immer nur einmal besucht werden und jeder Instruktion ohnehin nur ein state type zugewiesen werden kann.

Das folgende Beispiel demonstriert den Verifikationsprozess am Beispiel eines straight-line Codes:

<i>Instruktion</i>	<i>stack</i>	<i>lokale Variablen 0, 1</i>
Load 0	[]	[Class B, integer]
Store 1	[Class B]	[Class B, integer]
Load 0	[]	[Class B, Class B]
Getfield F A	[Class B]	[Class B, Class B]
...	[Class A]	[Class B, Class B]

A ist eine Superklasse von B, F ist ein Feld von A

Die Methode hat bei ihrer ersten Instruktion einen leeren Stack. Die erste lokale Variable enthält eine Referenz auf das aktuelle Objekt, die zweite lokale Variable einen Übergabeparameter. Die Operation Load 0 lädt den Typ der lokalen Variablen 0 in den Stack. Store1 speichert den Typ des obersten Stackelement in der lokalen Variablen 1, der Integer Typ wird überschrieben. Load 0 lädt erneut den Inhalt der lokalen Variablen 0 auf den Stack. Getfield F A lädt das Feld F von einem Objekt A.

Wäre die Methode nun zu Ende, so wäre ein Fixpunkt erreicht, da die state types der einzelnen Instruktionen sich nicht mehr ändern können. Der Code wäre abgearbeitet und der Verifikationsprozess verlief erfolgreich in linearer Zeit.

Komplizierter wird es, wenn Verzweigungen auftreten. Das vorangegangene Beispiel wird durch einen unbedingten Sprung erweitert.

Load 0	[]	[Class B, integer]	} state type
Store 1	[Class B]	[Class B, integer]	
Load 0	[]	[Class B, Class B]	
Getfield F A	[Class B]	[Class B, Class B]	
Goto -3	[Class A]	[Class B, Class B]	
Load 0	[]	[Class B, integer]	} method type
Store 1	[Class A]	[Class B, Err]	
Load 0	[]	[Class B, Class A]	
Getfield F A	[Class B]	[Class B, Class A]	

Bei der 2ten Instruktion (Store 1) muss nun überprüft werden, ob sich die beiden state types von den beiden Vorgängerinstruktionen kombinieren lassen. Der gemeinsame Supertyp von A und B ist A, daher wird der Stackinhalt durch den Typ A ersetzt. Bei der lokalen Variablen 1 wird es ein wenig komplizierter, da Integer ein Primitivtyp ist und sich demnach nicht mit anderen Typen kombinieren lassen kann. Der einzige Typ, der in der partiellen Ordnung über A und Integer steht, ist der allgemeinste Typ \perp (repräsentiert durch Err). Dieser wird nun in der lokalen Variablen 1 eingetragen. Err ist als Instruktionsoperand unbrauchbar und seine Verwendung würde einen Fehler im Verifikationsprozess auslösen. Da \perp in diesem Beispiel aber noch von der gleichen Instruktion mittels einer Store-Anweisung überschrieben wird, treten hier keine Probleme auf.

Nachdem sich die Typen der zweiten Instruktion durch die Kombination geändert haben, muss die Überprüfung der Nachfolgeroperationen ebenfalls erneut durchgeführt werden. Erst wenn sich nach einem Durchlauf keine der State Types mehr verändern, ist der gesuchte Fixpunkt erreicht. Die resultierende Liste von state Types beschreibt eine Typenkonstellation, die sich zum Ausführen der Bytecodeinstruktionen eignet. Sie wird method type genannt.

Die Speicheranforderungen für einen straight-line Code sind ausgesprochen gering. Geht man davon aus, dass jeder Typ durch 3 Bytes repräsentiert werden kann, so benötigt man eine Speicherkapazität von:

$$3*S + 3*V \text{ Bytes}$$

wobei S für den maximalen Stackinhalt und V für die Anzahl der lokalen Variablen steht.

Es muss immer nur der aktuelle state type gespeichert werden, der von der Nachfolgeinstruktion weiterverarbeitet werden soll.

Wenn Verzweigungen auftreten, ist wesentlich mehr Speicher erforderlich. Instruktionen können während der Verifikation mehrfach analysiert werden bis der Fixpunkt erreicht wird. Hierbei müssen die berechneten state types zwischengespeichert werden. Die Menge der state types benötigt

$$(3S + 3N + 3) * B \text{ Bytes.}$$

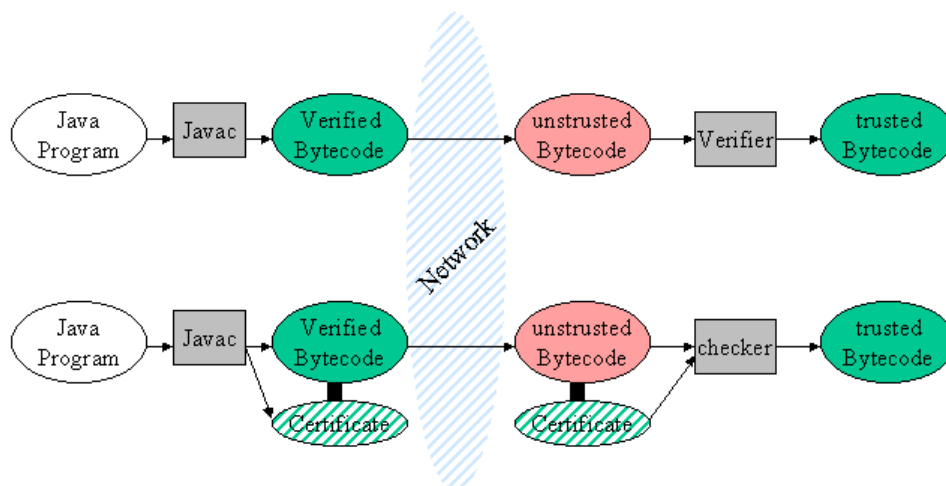
B steht für die Anzahl von Instruktionen mit mehreren Vorgängern

3 Bytes sind notwendig um den PC der Instruktion und die Stackhöhe zu speichern.

4.2.2 Lightweight Bytecode Verifikation:

Die größten Schwierigkeiten der Bytecode Verifikation in bezug auf JavaCard liegt in den hohen Speicheranforderungen, die während des Prozesses notwendig sind. JavaCards besitzen wie bereits erwähnt eine sehr geringe Speicherkapazität von wenigen Kilobytes. Der notwendige Speicher für die Verifikation nach oben beschriebenen Algorithmus ist in der Regel wesentlich höher.

Eine Möglichkeit dieses Problem zu umgehen liegt darin, die Verifikation off-card durchzuführen. Mittels kryptographischer Methoden (z.B. digitale Signaturen) könnte sichergestellt werden, dass ein auszuführendes Applet bereits von einem vertrauenswürdigen System überprüft worden ist. Ein wesentlicher Nachteil besteht hier allerdings darin, dass die Sicherheit dieses Lösungsansatz von der Vertrauenswürdigkeit der Verifikationsstelle und der Geheimhaltung des private Keys der Verifikationsstelle abhängt.



Praktischer wäre es also, wenn eine Verifikation on-card durchgeführt werden könnte. Hierzu eignet sich die von Eva und Kristoffer Rose beschriebene lightweight Bytecode Verifikation. Auch hier findet der eigentliche Verifikationsprozeß ebenfalls off-card statt, allerdings wird dabei ein Zertifikat erstellt, welches die notwendigen Informationen enthält, um zu einem späteren Augenblick die lightweight Bytecode-Verifikation durchzuführen. Anstatt nun on-card den ganzen Verifikationsprozeß zu wiederholen reicht es aus zu überprüfen, ob das Zertifikat dem Bytecode entspricht. Das Problem reduziert sich von einem Typ-Rekonstruktionsproblem auf ein Typ-Überprüfungsproblem, wobei der Typ dem Zertifikat gleich kommt.

Das Zertifikat ist nichts anderes als der Method Type, der Liste von State Types, die nach Abschluss der Verifikation den Instruktionen des Bytecode zugeordnet sind. Nun muss on-card lediglich überprüft werden, ob sich das Zertifikat auf den Bytecode anwenden lässt. Hierzu werden die Bytecodeinstruktionen einfach auf typenebene ausgeführt. Jede Instruktion muss vor der Ausführung einen passenden state type besitzen. Durch ihre Ausführung wird ein neuer state type erzeugt, der sich mit dem bereits durch das Zertifikat gegebenen state type der Nachfolgeinstruktionen kombinieren lässt. D.h. die gegebenen Typen der State types sind entweder gleich den neu berechneten Typen oder „allgemeiner“.

Diese Typenüberprüfung findet in linearer Zeit statt. Der notwendige Speicherbedarf entspricht der Größe des Zertifikats.

Load 0	[]	[Class B, integer]	} Zertifikat
Store 1	[Class A]	[Class B, Err]	
Load 0	[]	[Class B, Class A]	
Getfield F A	[Class B]	[Class B, Class A]	
Goto -3	[Class A]	[Class B, Class A]	

Das Zertifikat benötigt im übrigen nicht die vollständige Liste der state types.

Load 0		
Store 1	[Class A]	[Class B, Err]
Load 0		
Getfield F A		
Goto -3		

Es reicht aus die state types von den Instruktionen zu speichern, die verschiedene Vorgängerinstruktionen besitzen. Die verbleibenden state types können wie bei dem straight-line Code in linearer Zeit berechnet werden.

Verglichen mit Suns Implementation des Bytecode Verifiers für gewöhnlich Web-Applets, ist die lightweight Bytecode Verifikation wesentlich schneller und benötigt kaum Speicherressourcen. Die Größe des Zertifikats entspricht in unkomprimiertem Zustand etwa der des Bytecodes. Es kann in das EEPROM geladen werden, da während der Verifikation keine Schreibzugriffe notwendig sind. Auch ist dieses Verfahren wesentlich sicherer als die Alternative „Gutartigkeit“ von Bytecodes mittels Digitaler Signaturen zu garantieren.

Ein Nachteil, der durch dieses Verfahren entsteht liegt darin, dass zusätzlich zum Bytecode das Zertifikat mit übertragen werden muss.

4.2.3 on-card Verifikation nach Leroy:

Der Verifikationsalgorithmus von Leroy baut auf folgender Überlegung auf:

Die großen Speicheranforderungen bei der herkömmlichen Bytecode Verifikation entstehen durch das Speichern der state types von allen Schlüsselinstruktionen (Instruktionen die zu Verzweigungen gehören).

Ginge man davon aus, dass alle diese Instruktionen einen leeren Stack besäßen, so würde sich die Größe des State types stark reduzieren. Gilt außerdem dass, die Typen der lokalen Variablen sich nach Initialisierung nicht mehr verändern dürfen, so müssten auch sie nur einmal gespeichert werden anstatt für jede Schlüsselinstruktion.

Der nun beschriebene Algorithmus, der Bytecode mit diesen beiden Bedingungen verifiziert, ist eine Erweiterung des Verifikationsalgorithmus für einen straight-line Code und sieht wie folgt aus:

- Wird eine Instruktion mit mehreren Nachfolgern erreicht, so überprüfe, ob der Stack nach Ausführung dieser Sprungoperation und dem Entfernen eventuell für diese Sprungoperation notwendig gewesener Argumente, leer ist.
Wird eine Instruktion mit mehreren Vorgängern erreicht, so überprüfe, ob der Stack leer ist.
- Wird eine Store-Instruktion (Speichern in lokale Variablen erreicht), so überprüfe, ob sich der neue Typ mit dem bereits vorhandenen Typ in der lokalen Variablen kombinieren läßt. Sei A der zu speichernde Typ und B der vorhandene. Haben A und B einen gemeinsamen Vorgänger, so setze diesen Typ ein.
- Da sich die Typen der lokalen Variablen entsprechend den Store-Operationen ändern können, muss nach einem solchen Typwechsel der Algorithmus von neuem starten und so oft durchgeführt werden, bis sich die Werte in den lokalen Variablen nicht mehr verändern. Dieser Prozess gleicht der bereits beschriebenen Fixpunktsuche.

3S + 3N Bytes Arbeitsspeicher reichen für diesen Algorithmus aus, da wie bei der straight-line Bytecode Verifikation lediglich der zu übergebende Operandenstack und die lokalen Variablen gespeichert werden müssen.

Anders als bei den bisher beschriebenen Verifikationsalgorithmen werden hier nicht initialisierte Register mit dem Typ \perp (subtyp aller Typen) anstatt mit dem Typ \top (Supertyp aller Typen) belegt. Ein Problem bei

diesem Algorithmus besteht nämlich darin, dass nicht überprüft wird, ob Register vor ihrer Benutzung initialisiert worden sind. Würde man die Register zu Beginn mit \top initialisiert werden, so würden sich diese Typen nicht mehr ändern können, da bei Store Anweisungen nun ja immer der kleinste allgemeine Typ eingesetzt wird, also \top .

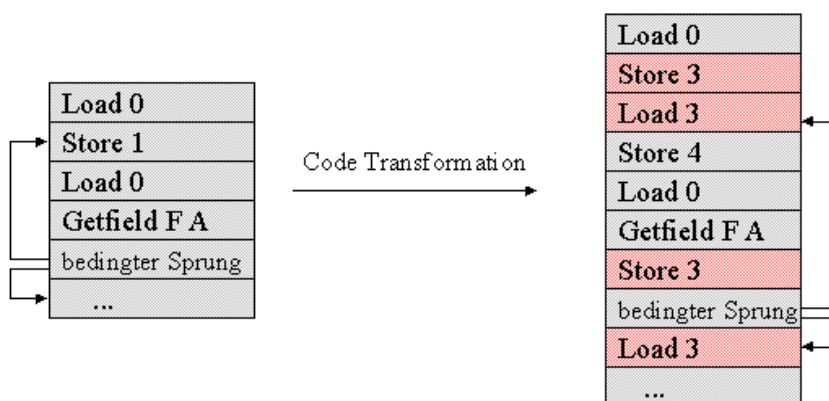
Wenn nun aber nicht initialisierte Register mit dem Typ \perp belegt werden, und eine Instruktion mit diesen Typ als Argument ausgeführt werden soll, so wird ebenfalls eine Fehlerbehandlung gestartet.

Dieses bereits demonstrierte Beispiel würde von diesem Algorithmus nicht verifiziert werden können, da die lokale Variable 1 zur Laufzeit mit zwei verschiedenen Typen (Class B und integer) belegt sein kann.

Load 0	[]	[Class B, integer]
Store 1	[Class B]	[Class B, integer]
Load 0	[]	[Class B, Class B]
Getfield F A	[Class B]	[Class B, Class B]
...	[Class A]	[Class B, Class B]

Um einen geeigneten Code zu erzeugen, der den beiden Anforderungen gerecht wird, muß eine off-card-Transformation des Bytecodes vorgenommen werden. Zum einem muß hierbei sichergestellt werden, dass die Stackhöhe bei Ausführung einer Verzweigungsanweisung und einer Instruktion mit mehreren Vorgängern immer gleich 0 ist. Zum anderen dürfen die Typen der lokalen Variablen nicht durch inkompatible Typen überschrieben werden.

Stack-Normalisierung:



Nach jeder verzweigenden Instruktion werden mittels store-Anweisungen die verbleibenden Stack-Elemente in die lokalen Variablen gespeichert. Bevor Zielinstruktion ausgeführt werden, muss der Stack wieder durch Load-Anweisungen in den ursprünglichen Zustand gebracht werden.

In diesem Beispiel erfolgt die Stack-Normalisierung sehr ineffizient. Bestimmte Instruktionen könnten durch eine optimierte Normalisierung entfernt werden. Auch werden hier 4 Register benutzt. Eine Reduzierung durch mehrfache Benutzung von Registern wäre aber möglich.

Registerneuverteilung:

Eine einfache Möglichkeit, die passenden Registerbelegungen zu erzeugen, bestünde darin, jedes Register, welches mit mehreren Typen verwendet wird, in verschiedene Register zu zerlegen, so dass schließlich jedem Register nur noch ein Typ zugewiesen ist (siehe Beispiel). Es existieren aber wesentlich effizientere Methoden Register zu verwenden. Werte, die typkompatibel sind, und zur Laufzeit unterschiedliche Lebenszeiten haben, können dieselben Register benutzen.

Bemerkenswert ist, dass eine optimierte Transformation zu einer nur sehr geringen Vergrößerung des Bytecodes führt. Noch überraschender ist, dass sich die durchschnittlich notwendige Anzahl an lokalen Variablen, trotz der zusätzlich geforderten Bedingung, sogar leicht reduziert. Dies demonstriert das Transformationsergebnis von folgenden Paketen.

Package	Code size (bytes)			Resident size (bytes)			Registers
	Orig.	Transf.	Incr.	Orig.	Transf.	Incr.	
java.lang	92	91	-1%	320	319	-0.3%	0.0%
javacard.framework	4047	4142	+2.3%	5393	5488	+1.8%	+0.3%
com.sun.javacard.HelloWorld	100	99	-1%	220	219	-0.5%	0.0%
com.sun.javacard.JavaPurse	2558	2531	-1%	3045	3018	-0.8%	-8.3%
com.sun.javacard.JavaLoyalty	207	203	-1.9%	365	361	-1%	0.0%
com.sun.javacard.installer	7043	7156	+1.6%	8625	8738	+1.3%	-7.5%
Total	14047	14222	+1.2%	17968	18143	+0.9%	-4.2%

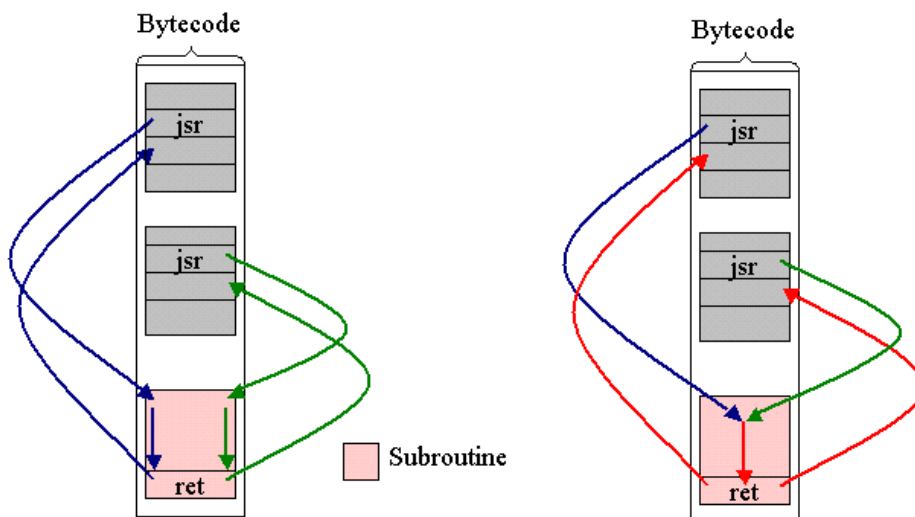
Diese Angaben entstammen dem Text „on-Card Bytecode Verification for Java Card“ von Xavier Leroy. Das Ergebnis lässt vermuten, dass Bytecodeerzeugung durch SUNs Java-Compiler relativ ineffizient stattfindet.

5. Subroutinen:

In dieser Ausarbeitung wurden die Grundzüge der Bytecode-Verifikation gezeigt. Nicht beschrieben wurde ein wichtiger Bestandteil der Verifikation: die Behandlung von Subroutinen. Zum Ende soll deswegen noch ein kurzer Einblick in diese Thematik gegeben werden.

Subroutinen sind „geteilter Bytecode“ in einer Methode. Sie können aus verschiedenen Positionen im Bytecode durch Instruktion `jsr` angesprungen werden. Nach Abarbeitung der Subroutine wird mittels der Instruktion `ret` wieder an die Nachfolgeinstruktion von dem `jsr` Befehl zurückgesprungen. Die `ret` entnimmt die Rücksprungadresse aus einer lokalen Variablen, in der sie zwischengespeichert wurde.

Das Problem ist nun, dass auf unterschiedlichen Positionen in diese Subroutineblöcke gesprungen werden kann und dabei die lokalen Variablen mit unterschiedlichen Typen belegt sein können. Der in dieser Ausarbeitung vereinfachte Verifikationsalgorithmus würde nun versuchen diese zu kombinieren und damit „falsche“ state types erzeugen.



Damit es also nicht zu diesem (für die Verifikation eventuell entscheidenden) Verlust an Präzision kommt, müssen Subroutinen gesondert behandelt werden. Bei dem von Leroy beschriebenen Verfahren allerdings können `jsr` Instruktionen wie gewöhnliche Sprunganweisungen behandelt werden. Das Kombinieren von lokalen Variablen entfällt in seinem Prozess, da ja nur noch mit einer einzigen Liste von Registern gearbeitet wird, die für alle Instruktionen gelten.

6. Literatur:

- Tim Lindhol, Frank Yellin, The Java™ Virtual Machine Specification
- Xavier Leroy, Java bytecode verification: an overview
- Xavier Leroy, On-card bytecode verification for Java card
- Gerwin Klein, Tobias Nipkow, Verified Lightweight Bytecode Verification
- Karsten Sohr, Die Sicherheitsaspekte von mobilem Code
- Eva Rose, Kristoffer Rose, Lightweight Bytecode Verification
- J.M.Joller, Java Security