Verified Java Bytecode Verification

Gerwin Klein

gerwin.klein@nicta.com.au

Abstract: Der Bytecode Verifier ist ein essenzieller Bestandteil der Sicherheitsarchitektur der Programmierplattform Java. Meine Dissertation stellt eine formale, ausführbare Spezifikation des Bytecode Verifiers vor sowie den Beweis, dass diese korrekt ist. Die Formalisierung, vollständig im Theorembeweiser Isabelle durchgeführt, besteht aus einem abstrakten Framework für Bytecode-Verifikation, das mit zunehmend ausdrucksstarken Typsystemen instanziiert wird. Diese decken sämtliche wichtigen Eigenschaften der Java-Plattform ab. Die Formalisierung liefert zwei ausführbare, verifizierte Bytecode Verifier: den Standard-Algorithmus, wie er auf normalen Desktop-Rechnern benutzt wird, sowie einen Lightweight Bytecode Verifier für eingebettete Systeme mit Ressourcenbeschränkungen wie z.B. Java SmartCards.

1 Motivation

Mydoom richtet Milliardenschäden an	(FAZ.NET, 03.02.2004)
Zombie-Rechner: Heim-PC als Spam-Versender	(Spiegel Online, 24.03.2004)
Mass-Mailing-Würmer nehmen kein Ende	(Heise Online, 29.04.2004)

Meldungen wie diese zeigen, dass Computersicherheit längst nicht mehr nur ein Thema für große Institutionen wie Banken oder Regierungen ist, sondern jeden betreffen kann, der einen Rechner benutzt. Die häufigste Angriffsmethode bestand in den letzten Jahren darin, den Benutzer dazu zu bringen, den Schädling selbst zur Ausführung zu bringen, der beispielsweise durch Email-Attachments Verbreitung findet. Die meisten Ausführungsumgebungen und Betriebssysteme treffen keine Vorkehrungen, die es erlauben würden, unbekannten Code ohne Sicherheitsrisiko auszuführen. Die einzige Gegenwehr scheint zu sein, Benutzer anzuhalten, keine unbekannten Programme auszuführen. Wie erfolgreich diese Strategie ist, lässt sich der Presse entnehmen.

Es ist jedoch keineswegs unmöglich, solche Vorkehrungen zu treffen und Code aus unbekannter Quelle ohne Sicherheitsrisiko auszuführen. Es wurden in der Forschung einige Lösungen für das Problem vorgeschlagen. Die Java-Plattform enthält eine solche Lösung: die sog. *Sandbox*. Diese Sandbox ist eine Sicherheitsschicht, die es erlaubt, Programme so einzuschränken, dass der Zugriff auf Hardware- und Betriebssystemkomponenten flexibel geregelt und überprüft werden kann. Die Sandbox stützt sich auf folgende drei Sicherheitsmechanismen:

- Java wird nicht direkt in Maschinensprache übersetzt, sondern in eine Zwischen-

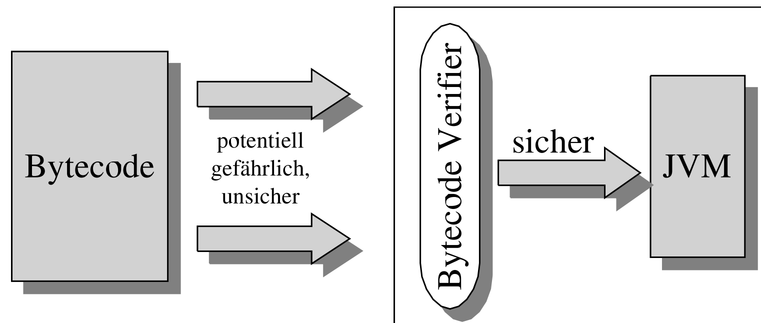

Abbildung 1: Die Java Laufzeitumgebung

sprache, den sog. Bytecode. Dieser wird von einer virtuellen Maschine (JVM) ausgeführt.

- Der Zugriff auf geschützte Ressourcen wird innerhalb der Sprache Java durch ein Set geeigneter Klassen [Go99] geregelt. Dies erlaubt eine sehr flexible Implementierung verschiedenster Zugriffs-Richtlinien.
- Vor der Ausführung eines Programms wird der Bytecode statisch verifiziert, um sicherzustellen, dass die o.g. Sicherheitsmaßnahmen greifen können. Nur verifizierte Programme werden zur Ausführung weitergereicht (Abb. 1).

Die letzgenannte dieser drei Komponenten ist der Bytecode Verifier (BV). Er ist ein zentraler Teil der Sicherheits-Architektur: ein Fehler im BV, der zur Ausführung eines unsicheren Applets führt, macht die gesamte Sandbox hinfällig. In der Literatur findet sich eine ganze Reihe solcher Fehler [MF96, MF99]; erst vor wenigen Monaten gab Microsoft ein als kritisch eingestuftes Sicherheitsupdate heraus, das den BV zum Inhalt hatte. Auf der anderen Seite ist der BV selbst ein umfangreiches und komplexes Programm, das eine ausgeklügelte Datenfluss-Analyse vornimmt. Es ist also gleichzeitig sowohl notwendig als auch höchst nichttrivial, seine Korrektheit sicherzustellen.

Meine Dissertation präsentiert eine vollkommen formale, ausführbare und maschinengeprüfte Spezifikation einer repräsentativen Teilmenge der JVM und des Bytecode Verifiers zusammen mit dem Beweis, dass der vorgestellte Bytecode Verifier korrekt ist.

Mit Ausnahme der Arbeit von Freund [Fr00] untersuchte die Literatur über den BV nur isolierte Eigenschaften oder zeigte – wenn überhaupt – nur Beweisskizzen. Dies liegt keineswegs an schlechter Arbeit der einzelnen Autoren: Der schiere Umfang der involvierten Formalisierungen macht es fast unmöglich, im Rahmen einer normalen Veröffentlichung einen detaillierten, vollständigen Beweis zu präsentieren, den Leser verstehen geschweige denn nachvollziehen oder überprüfen können.

Es ist wenig überraschend, dass sich in Freunds Arbeit, die einen solchen detaillierten Beweis für einen relativ großen Ausschnitt des BV liefert, gerade die Art von kleinen Fehlern

findet, die in einer formalen Entwicklung dieser Größe und Komplexität ohne Maschinenunterstützung zu erwarten sind. Freund hat hervorragende Pionierarbeit geleistet, und trotz kleiner Fehler sind seine Hauptaussagen wahrscheinlich wahr. Doch genau das ist der Punkt: sie sind nur *wahrscheinlich* wahr. Um sicher zu sein, müsste man einige subtile Änderungen vornehmen, um die vorhandenen Fehler zu beseitigen, und danach mehr als 100 Seiten komplexer formaler Beweise erneut manuell überprüfen.

Dieses Grundproblem wird in meiner Arbeit durch die Benutzung eines maschinellen Theorembeweisers gelöst. Die Formalisierung des BV in Isabelle/HOL [NPW02] verringert weder die Komplexität noch den Umfang des Problems, sie verschlimmert zunächst sogar beides, weil man an wesentlich strengere Regeln gebunden ist, als sie in allgemeinen mathematischen Beweisen üblich sind. Weder kleine notationelle Ungenauigkeiten, noch Auslassungen in der Beweisführung sind möglich. Sie hat jedoch folgende entscheidende Vorteile:

Korrektheit Offensichtlich ist das Vertrauen in die Korrektheit der Beweise erheblich gestärkt, da sie maschinell überprüft sind. Solange ein Beweis nicht gerade einen Fehler im Theorembeweiser selbst benutzt, ist er ohne jeden Zweifel korrekt. Isabelle ist ein Theorembeweiser im sog. LCF-Stil [GMW79]. Der korrektheitskritische Teil ist in einem sehr kleinen Beweiskern isoliert. Aus diesem Grund sind korrektheitskritische Fehler im System äußerst selten.

Validierung Die Spezifikation ist ausführbar und kann gegenüber existierenden Implementierungen der JVM und des BV validiert werden. Sie kann auch als Referenzimplementierung verwendet werden. Der traditionelle Ansatz zur Validierung bleibt natürlich ebenfalls erhalten: Es ist möglich, Dokumente aus Isabelle-Spezifikationen zu generieren und sie mit der offiziellen Sprachspezifikation Punkt für Punkt zu vergleichen.

Lesbarkeit Die Korrektheit einer Aussage ist nicht unbedingt der einzige Zweck eines Beweises. Es ist oft von gleichem oder größerem Interesse, zu welchen tieferen Einsichten und neuem Verständnis ein Beweis führt. Nicht zuletzt aus diesem Grund werden immer neue Beweise zu mathematischen Theoremen veröffentlicht, deren Richtigkeit längst allgemein bekannt ist. Diese wichtige Eigenschaft formaler Beweise geht bei den meisten mechanischen Theorembeweisern verloren. Die Beweissprache Isabelle/Isar, die ich in großen Teilen der Formalisierung verwendet habe, stellt die volle Formalität eines maschinellen Theorembeweisers sicher, und macht gleichzeitig durch ihre lesbare Form die Beweisargumentation auch dem Menschen direkt zugänglich.

Meine Arbeit konzentriert sich auf den Bytecode Verifier – auf den Algorithmus genauso wie die Eigenschaften der Bytecode-Sprache, die ihn betreffen. Von diesen behandle ich Klassen, Objekte, Vererbung, virtuelle Methoden, Ausnahmebehandlung, Bytecode-Subroutinen und Arrays. Wie schon erwähnt, wurden in der Literatur die meisten dieser Sprachbestandteile in Isolation untersucht. Hier dagegen wird schrittweise eine Sprache aufgebaut, die alle diese Bestandteile enthält, und die Ansätze der Literatur konsequent auf

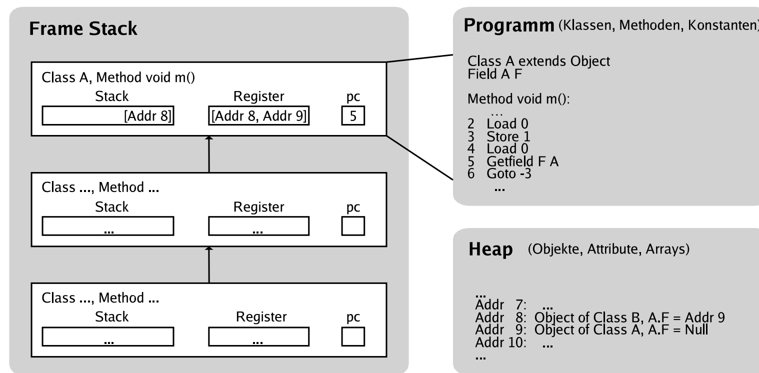

Abbildung 2: Die Virtuelle Maschine

die gesamte Sprache verallgemeinert. Trotz der subtilen Interaktion der einzelnen Komponenten, die ein solches Gesamt-Modell interessant machen, war es möglich, das resultierende System als korrekt und sicher nachzuweisen. Die Spezifikation liefert am Ende zwei ausführbare, verifizierte Bytecode Verifier: den iterativen Standard-Algorithmus sowie einen sog. Lightweight Bytecode Verifier für eingebettete Systeme mit Ressourcenbeschränkungen.

Die Arbeit wurde im Rahmen des EU-Projekts VerifiCard erstellt, das sich mit der formalen Modellierung und Verifikation der JavaCard-Plattform und darauf laufender Applikationen beschäftigte. An dem Projekt waren Partner aus mehreren europäischen Ländern, Industrie und akademischer Forschung beteiligt. Die Plattform-Modellierung untersuchte die Quellsprache JavaCard, die zugehörige virtuelle Maschine sowie einen Compiler zwischen diesen beiden Ebenen. In der SmartCard-Applet-Verifikation wurden sowohl algorithmische Analysen als auch deduktive Methoden untersucht. Das Projekt wurde mittlerweile erfolgreich abgeschlossen. Insbesondere die Plattform-Verifikation, die meine Arbeit als wichtigen Teil enthält, wurde als besonders erfolgreich bewertet. Ein Folge-Projekt ist in Planung.

Die nächsten beiden Abschnitte stellen zunächst das Thema Bytecode-Verifikation etwas genauer vor und geben danach einen kurzen Überblick über die wesentlichen Aspekte der Formalisierung.

2 Bytecode-Verifikation

Der BV verifiziert, dass Bytecode-Programme sich an gewisse Spielregeln der virtuellen Maschine halten. Abbildung 2 zeigt das Innenleben der JVM.

Sie umfasst drei wichtige Komponenten: statische Informationen über das Programm (im Bild rechts oben dargestellt), den Zustand dynamisch erzeugter Objekte (rechts unten) und

schließlich den aktuellen Ausführungszustand des Programms (links). Die statischen Informationen beinhalten sowohl die Klassenstruktur des Programms inklusive aller Feld- und Methoden-Deklarationen als auch den Bytecode aller Methoden. Dynamisch erzeugte Objekte werden nach dem Schema der zugehörigen Klasse im Heap abgelegt. Der restliche Ausführungszustand der Maschine ist im sog. Frame-Stack organisiert. Dieser enthält einen Eintrag für jede aktive Methode, der beim Aufruf der Methode erzeugt und bei Beendigung wieder gelöscht wird. Jeder der Frame-Stack-Einträge enthält einen Operanden-Stack für die Berechnung von Ausdrücken, einen Satz von Registern, in dem Parameter und lokale Variablen abgelegt werden, den aktuellen Programmzähler innerhalb der Methode und die Information, welcher Methode der Eintrag zuzuordnen ist.

Im Gegensatz zu üblichen Maschinensprachen sind die Instruktionen der JVM typisiert. Dies bedeutet, dass z.B. die `iadd`-Instruktion zwei Integer-Werte als Operanden auf dem Stack erwartet und nicht auf Adressen andgewendet werden kann. Analog dazu darf die `getfield`-Instruktion nur auf Objekt-Referenzen der richtigen Klasse angewendet werden, und nicht etwa auf Fließkommazahlen.

Die JVM verlässt sich bei der Ausführung auf folgende Regeln:

Typisierung Alle Instruktionen werden mit Argumenten richtiger Anzahl, Ordnung und Typs versorgt. Dies bezieht sich auf Argumente jeglicher Art, insbesondere auf dem Operanden-Stack, in den Registern und auf dem Heap. Diese Eigenschaft sorgt dafür, dass das Typsystem der JVM nicht verletzt wird und die Ausführung weitgehend ohne Laufzeittests fehlerfrei ablaufen kann.

Überlauf Keine Instruktion versucht auf den leeren Operanden-Stack zuzugreifen oder mehr Elemente auf den Stack zu laden als statisch in der Methode angegeben. Jede Instruktion greift nur auf Register zu, die statisch in der Methode angegeben wurden. Diese Eigenschaft stellt sicher, dass Operanden-Stack und Register effizient auf herkömmlicher Hardware implementiert werden können, da die maximale Größe bereits vor Ausführung des Programms feststeht.

Programmzähler Der Programmzähler zeigt zu jedem Zeitpunkt auf eine Instruktion im Bytecode. Insbesondere darf der Programmzähler am Ende der Methode nicht ins Leere zeigen oder bei einem Sprung in die Mitte einer Instruktionscodierung. Diese Eigenschaft ermöglicht eine effiziente Implementierung des Programmzählers ohne Laufzeittests.

Initialisierte Register Analog zur *Definite-Assignment*-Analyse für lokale Variablen auf der Quellsprache müssen in der JVM alle Register außer den Parametern zuerst beschrieben werden, bevor aus ihnen gelesen werden kann. Dies erspart eine explizite Default-Initialisierung der Register beim Methoden-Aufruf.

Initialisierte Objekte Bevor auf Felder oder Methoden eines Objekts zugegriffen wird, muss der Konstruktor des Objekts aufgerufen worden sein. Jeder Konstruktor muss seinerseits den Konstruktor der Superklasse aufrufen, bevor er selbst auf Methoden und Felder des Objekts zugreifen darf. Diese Eigenschaft garantiert, dass Objekte ohne äußeren Einfluss einen konsistenten internen Zustand herstellen können. Sie

Instruktion	Stack	Register
`load 0`	($[]$,	$[Class\ B, Integer]$)
`store 1`	($[Class\ A]$,	$[Class\ B, Err]$)
`load 0`	($[]$,	$[Class\ B, Class\ A]$)
`getfield F A`	($[Class\ B]$,	$[Class\ B, Class\ A]$)
`goto -3`	($[Class\ A]$,	$[Class\ B, Class\ A]$)

Abbildung 3: Eine Typisierungs-Tabelle

ist besonders für die Implementierung der systemeigenen Sicherheits-Klassen von zentraler Bedeutung.

Der Zweck des Bytecode-Verifiers ist es, vor Ausführung des Programms sicherzustellen, dass diese Eigenschaften erfüllt sind. Der BV löst diese Aufgabe durch abstrakte Interpretation des Bytecode-Programms. Anstelle konkreter Werte benutzt er Typen, um das Programm abzuarbeiten.

Abbildung 3 zeigt ein vereinfachtes Beispiel. Die linke Spalte der Tabelle enthält Instruktionen, die rechte Spalte enthält die Typen der möglichen Werte in Stack und Registern direkt vor Ausführung der jeweiligen Instruktion. Für jede Instruktion werden Anzahl, Reihenfolge und Typ der Argumente überprüft. Im Falle der `load 0`-Instruktion in der ersten Zeile von Abbildung 3 ist das Argument das Register 0. Dieses Register existiert und der Stack ist groß genug, das Ergebnis aufzunehmen. Die Instruktion `load 0` bewegt den Wert des Registers 0 auf den Stack. Die zweite Zeile der Tabelle ergibt sich zum Teil aus der symbolischen Ausführung der ersten. Es wäre zunächst zu erwarten, dass der Stack den Typ $[Class\ B]$ enthält (den vorherigen Wert von Register 0) und dass sich die Typen der Register nicht ändern. Unter der Annahme, dass A eine Superklasse von B ist, kann die zweite Zeile der Tabelle aber trotzdem noch als korrekte Approximation angesehen werden: jedes B-Objekt ist auch ein A-Objekt und der Integer-Wert in Register 1 wird nun mit Err als nicht benutzbar gekennzeichnet. Der Grund für diese Verallgemeinerung liegt in der letzten Instruktion der Tabelle: Die Ausführung von `goto -3` führt ebenfalls zu Zeile zwei, und auch für die Register- und Stack-Werte der `goto`-Instruktion müssen die Einträge in Zeile zwei gelten.

Der BV berechnet die Typisierungs-Tabelle durch schrittweise abstrakte Ausführung der Instruktionen. Trifft diese Ausführung auf eine schon berechnete Stelle wie bei der `goto`-Instruktion im Beispiel, so werden die berechneten Werte zum kleinsten Obertypen verallgemeinert, der die Situation noch korrekt beschreibt. Die Berechnung wird mit dem neuen Wert wiederholt, bis sich keine Änderungen mehr an der Tabelle ergeben.

In meiner Arbeit zeige ich, dass diese Fixpunkt-Iteration für alle Programme terminiert und eine korrekte Approximation der Ausführung liefert.

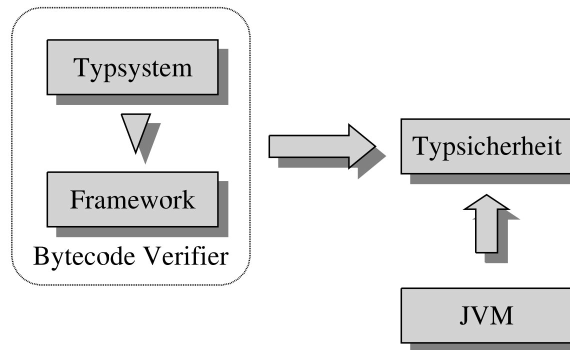

Abbildung 4: Die Formalisierung

3 Verifikation

Abbildung 4 zeigt die Hauptkomponenten der Formalisierung im Überblick.

Die zentrale Eigenschaft, die bewiesen wird, ist die Typsicherheit (rechts im Bild). Dieses Modul umfasst die abstrakte Definition der Eigenschaft, die konkreten Instanziierungen für verschiedene Typsysteme und natürlich ihren Beweis. Sie ruht auf der Formalisierung der virtuellen Maschine (rechts unten) und ihres Ausführungsverhaltens. Das Haupt-Objekt des Typsicherheits-Beweises ist der BV (links im Bild). Die Formalisierung der BV besteht aus einem abstrakten Framework, das mit verschiedenen konkreten Typsystem instanziiert werden kann. Das Framework beinhaltet die algorithmischen Aspekte wie z.B. den oben erwähnten Terminationsbeweis der Datenfluss-Analyse. Im konkreten Typsystem wird beispielsweise festgelegt, wie der Initialisierungs-Status von Objekten auf der Typisierungsebene kodiert werden kann und wie sich diese Kodierung in Relation zu der bestehenden Subtyp-Ordnung verhält.

Bei einer Formalisierung dieser Größe ist es wichtig, die Hauptaussage verständlich und übersichtlich zu halten, so dass klar bleibt, was das bewiesene Theorem genau aussagt. Die Hauptaussage meiner Arbeit ist die Typsicherheit der Java-Bytecode-Sprache. Sie ist als Äquivalenz zweier Maschinen formuliert: einer offensiven und einer defensiven JVM. Die defensive JVM unterscheidet zwischen normalen und fehlerhaften Zuständen. Sie überprüft zur Laufzeit direkt vor Ausführung jeder einzelnen Instruktion explizit alle oben erwähnten Sicherheitseigenschaften. Die offensive JVM entspricht einer normalen Implementierung und lässt diese zusätzlichen Überprüfungen entfallen.

Das Haupttheorem lautet: Wenn der BV ein Programm als sicher akzeptiert, dann ist das Verhalten der defensiven JVM auf diesem Programm gleich dem der offensiven JVM.

Dies bedeutet, dass alle Laufzeittests der defensiven Maschine erfolgreich sein werden und die defensive Maschine demzufolge durch eine offensive Implementierung ersetzt werden kann. Die formale Korrektheit des Theorems ist durch den Beweiser Isabelle sichergestellt. Die formale Bedeutung des Theorems ist leicht zu erfassen, da es direkt über Konzepte der

JVM zur Laufzeit redet. Diese kann man isoliert vom Rest der Formalisierung überprüfen und verstehen, ohne komplizierte Kodierungen im Typsystem nachvollziehen zu müssen.

4 Zusammenfassung

Die Hauptbeiträge meiner Arbeit sind die folgenden:

- Meine Arbeit stellt eine der umfangreichsten Anwendungen des Theorembeweisers Isabelle dar. Sie zeigt, dass formale Verifikations-Technologie die erforderliche Reife erlangt hat, um reale Probleme realer Systeme zu fassen und zu lösen.
- Java-Bytecode-Verifikation ist sowohl in der Industrie als auch in der akademischen Forschung ein sehr aktives Feld. Trotz dieser Tatsache ist meine Formalisierung nach wie vor eine der vollständigsten veröffentlichten Formalisierungen des Bytecode Verifiers. Zum Zeitpunkt der Veröffentlichung war sie die umfangreichste und genaueste solche Formalisierung, die in einem Theorembeweiser durchgeführt wurde. Mittlerweile existiert eine weitere, die von Umfang und Detail mit der meinen vergleichbar ist [Du03]. Sie wurde von Partnern im EU-Projekt VerifiCard erstellt.
- Abstrakt gesehen besteht der BV aus einem Typsystem für die JVM sowie aus einem Typinferenz-Algorithmus, der durch Datenfluss-Analyse implementiert wird. Die Einteilung der Formalisierung in ein abstraktes Framework und separate Instanziierungen macht es möglich, den algorithmischen Aspekt der Datenfluss-Analyse formal klar und sauber vom Typsystem zu trennen. Die resultierende einheitliche Behandlung verschiedener Bytecode-Verfikations-Algorithmen führte zu einer verbesserten Version des Lightweight Bytecode Verifiers, die sowohl die ursprüngliche Version von Rose [Ro02, RR98] als auch die industrielle Implementierung von Sun Microsystems [Su00] an Allgemeinheit übertrifft. Diese Algorithmen sind nicht nur theoretisch ausführbar: es wurden lauffähige Prototypen direkt aus der Spezifikation generiert.
- In meiner Arbeit untersuche ich eine Reihe verschiedener Typsysteme für die JVM. Darunter befinden sich verschiedene Versionen des sog. *Merging*-Typsystems, wie es in der JVM-Spezifikation beschrieben wird sowie ein neueres mengen-basiertes System, das sich besonders gut zur Analyse von Bytecode-Subroutinen eignet. Bytecode-Subroutinen wurden in der Forschung lange Zeit als gravierendes Problem der Java-Bytecode-Sprache angesehen, da sie im BV schwierig zu analysieren sind. Dies ging so weit, dass die entsprechenden Instruktionen aus dem Befehlssatz für eingebettete Systeme gestrichen wurden. Meine Formalisierung greift erste theoretische Überlegungen [Co01] zu diesem Problem auf, erweitert diese auf eine repräsentative Teilmenge der JVM und zeigt schließlich in einem mathematischen, mechanisch überprüften Beweis, dass die Lösung sowohl korrekt als auch sicher ist. Diese repräsentative Teilmenge der JVM umfasst alle Bestandteile, die für die Ausführungs-Sicherheit relevant sind. Sie umfasst insbesondere die formal aufwändige Objekti-

nitialisierung, die für die Funktion der Kontroll-Mechanismen in Java von zentraler Bedeutung ist.

Die folgende Tabelle gibt eine Übersicht über die Größe des Frameworks und der vier verschiedenen Instanziierungen. Jede Instanziierung ist eine Erweiterung der darüber liegenden Zeile und modelliert alle BV-Bestandteile, die in der darüber liegenden Zeile enthalten waren. Die Quellen der Formalisierung und die daraus generierten Beweisdokumente sind im Internet verfügbar [Kl03].

Modul	Isabelle-Code (Zeilen)	Beweisdokument (Seiten)
Framework	3.500	87
Exceptions	11.100	245
Objekt-Initialisierung	14.300	303
Bytecode-Subroutinen	13.600	293
Arrays	15.200	333

Nicht behandelt wurden in der Arbeit einige Konzepte, die in der Literatur als bereits gelöst galten oder die nur zu einem größeren Umfang der Formalisierung geführt hätten, ohne entscheidend zum Stand der Forschung beizutragen. In weiterführenden Arbeiten habe ich diese Konzepte hinzugefügt, darunter Interfaces, statische Methoden, statische Felder etc., und den Instruktionssatz von 22 abstrakten Instruktionen in der Formalisierung auf den vollständigen Satz von über 200 Instruktionen der JVM erweitert. Der relativ geringe Aufwand von ca. 6 Personenmonaten zeigt die Flexibilität und Tragfestigkeit der Dissertation.

Werdegang

Gerwin Klein schloss sein Studium der Informatik mit Nebenfach Mathematik an der Technischen Universität München im Jahr 1999 mit Auszeichnung ab. Für seine Arbeit über die Generierung von graphischen Editoren erhielt er den Diplom-Preis der Siemens AG. Während des Studiums war Herr Klein an mehreren Projekten beteiligt, u.a. am preisgekrönten Entwicklungswerkzeug AutoFocus, das an der TU München und von der Validas AG weiterentwickelt wird. Er ist Autor des Scannergenerators JFlex, der weltweit als Standardwerkzeug für lexikalische Analyse in Java eingesetzt wird. Herr Klein promovierte, ebenfalls mit Auszeichnung, im Jahr 2003 unter Betreuung von Tobias Nipkow an der TU München. Während der Promotion über die Verifikation des Java Bytecode Verifiers war er in der Lehre tätig, nahm am EU-Projekt VerifiCard teil und trug zur Entwicklung des Theorembeweisers Isabelle bei. Derzeit forscht er als Research Scientist bei National ICT Australia und lehrt als Conjoint Lecturer an der Universität von New South Wales in Sydney, Australien. Er ist Autor von 10 Publikationen.

Literatur

[Co01] Coglio, A.: Simple verification technique for complex Java bytecode subroutines. Technischer Bericht. Kestrel Institute. Dez. 2001.

[Du03] Dufay, G.: *Vérification formelle de la plate-forme Java Card*. Dissertation. Université de Nice Sophia Antipolis. 2003.

[Fr00] Freund, S. N.: *Type Systems for Object-Oriented Intermediate Languages*. Dissertation. Stanford University. 2000.

[GMW79] Gordon, M. J., Milner, A. J. und Wadsworth, C. P.: *Edinburgh LCF: A Mechanised Logic of Computation*. Volume 78 of *Lecture Notes in Computer Science*. Springer. 1979.

[Go99] Gong, L.: *Inside Java 2 Platform Security*. The Java Series. Addison Wesley. 1999.

[Kl03] Klein, G.: *Verified Java Bytecode Verification*. Dissertation. Institut für Informatik, Technische Universität München. 2003.
`http://www4.in.tum.de/~kleing/diss/`.

[MF96] McGraw, G. und Felten, E. W.: *Java Security: Hostile Applets, Holes and Antidotes*. John Wiley and Sons. 1996.

[MF99] McGraw, G. und Felten, E. W.: *Securing Java: Getting Down to Business with Mobile Code*. John Wiley and Sons. 2nd. 1999.

[NPW02] Nipkow, T., Paulson, L. C. und Wenzel, M.: *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. Volume 2283 of *Lecture Notes in Computer Science*. Springer. 2002.

[Ro02] Rose, E.: *Vérification de Code d'Octet de la Machine Virtuelle Java. Formalisation et Implantation*. Dissertation. Université Paris VII. 2002.

[RR98] Rose, E. und Rose, K.: Lightweight bytecode verification. In: *OOPSLA'98 Workshop Formal Underpinnings of Java*. 1998.

[Su00] Sun Microsystems. CLDC and the K Virtual Machine (KVM). 2000.
`http://java.sun.com/products/cldc/`.