
Bytecode Verifikation

im Rahmen des Hauptseminars:

**Nachweis von Sicherheitseigenschaften für JavaCard durch
approximative Programmauswertung**

Christian Pacher (pacher@in.tum.de)
24. Januar 2002

Bytecode-Verifikation:

ein elementarer Bestandteil der Java-Sicherheitsarchitektur

Struktur:

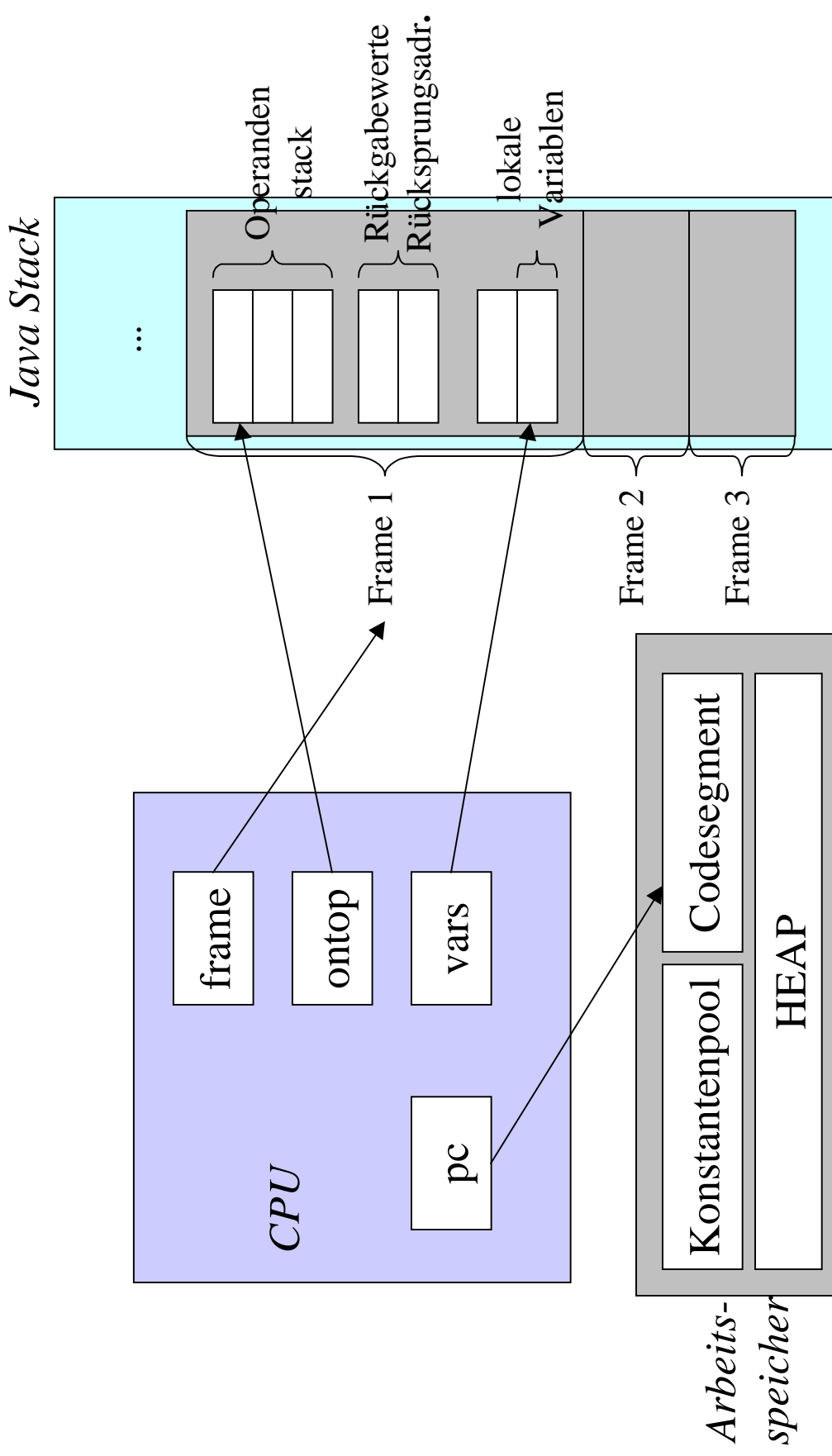
1. Die Java Virtual Machine
2. Bytecode-Format / Darstellung
3. Bytecode-Verifikation
4. BV und JavaCard

JVM (Java Virtual Machine)

- ist eine abstrakte Stackmaschine
- meist in „Software“ realisiert
- definiert durch Spezifikationen von SUN
- verarbeitet plattformunabhängigen Java-Bytecode

- besteht aus drei Komponenten:
 - Java Stack
 - CPU
 - Arbeitsspeicher

schematischer Aufbau der JVM:



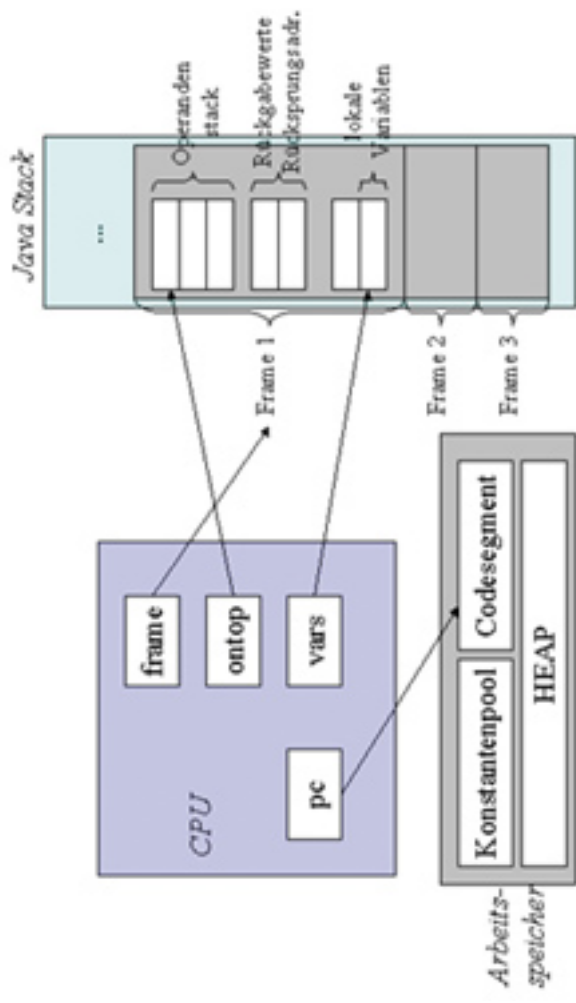
Java-Stack:

- lokale Variablen:
Objektreferenz, Methodenpar.

- Operandenstack:
Operandenverwaltung

Arbeitsspeicher:

- Heap: Speicher für Objekte
- Codesegment: Bytecode Instruktionen
- Konstantenpool: enthält Konstanten, Signaturen von Methoden, ...



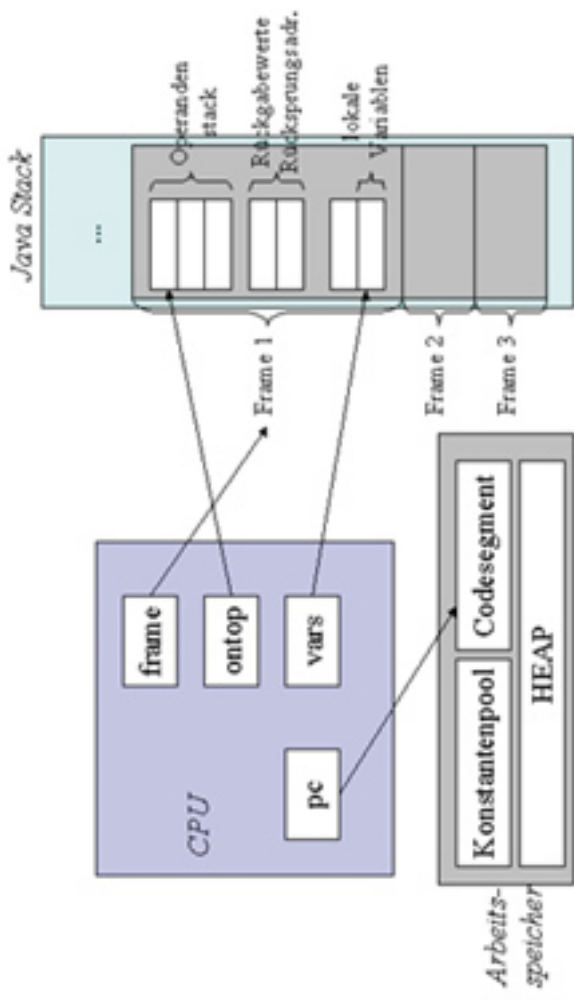
CPU-Register:

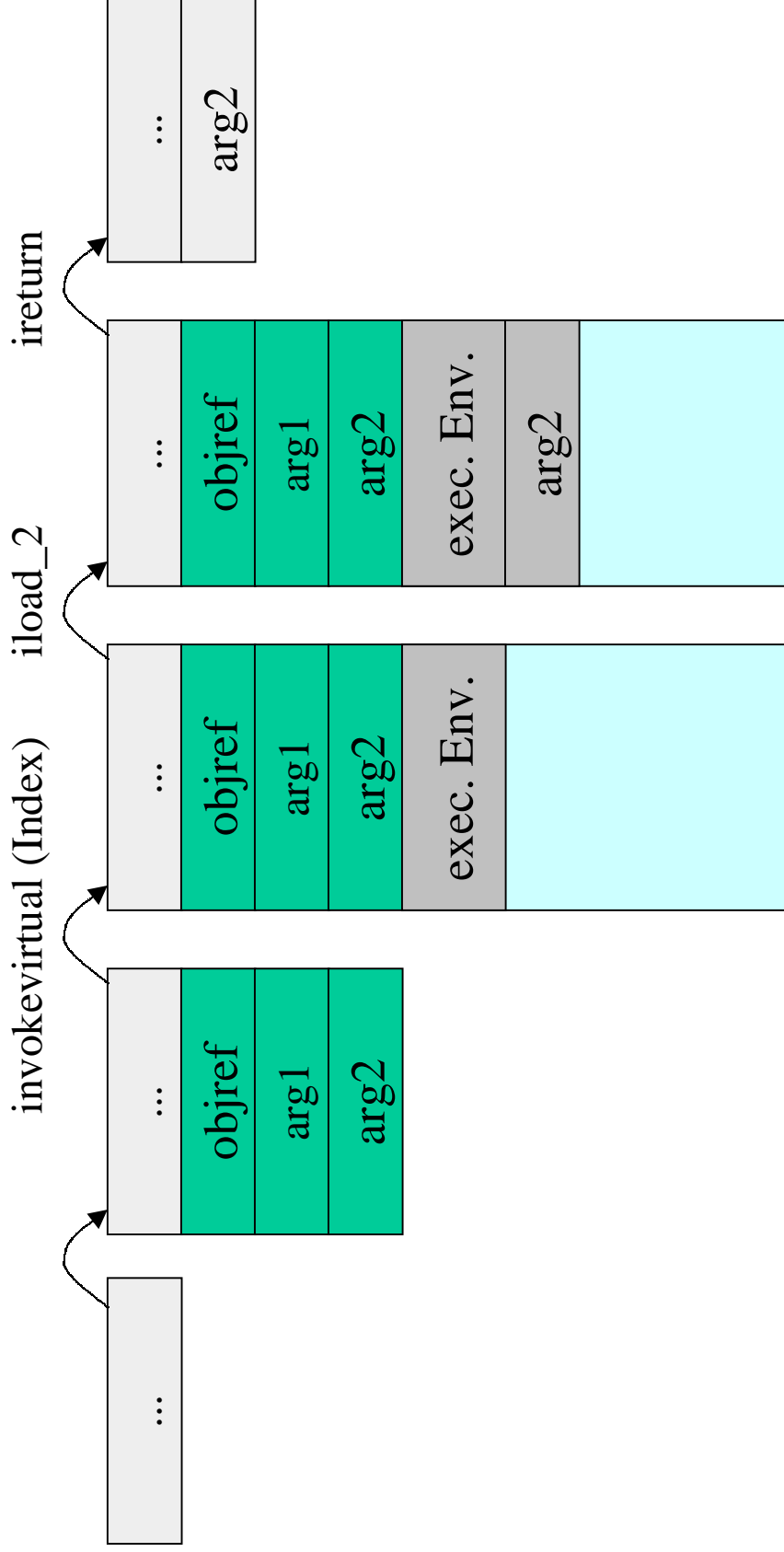
- frame:
zeigt auf „obersten“ Frame

- vars:
zeigt auf die 0te lokale Variable

- ontop:
zeigt auf „oberstes“ Element im Operandenstack

- pc:
Program Counter zeigt auf aktuell ausgeführte Instruktion im Codesegment




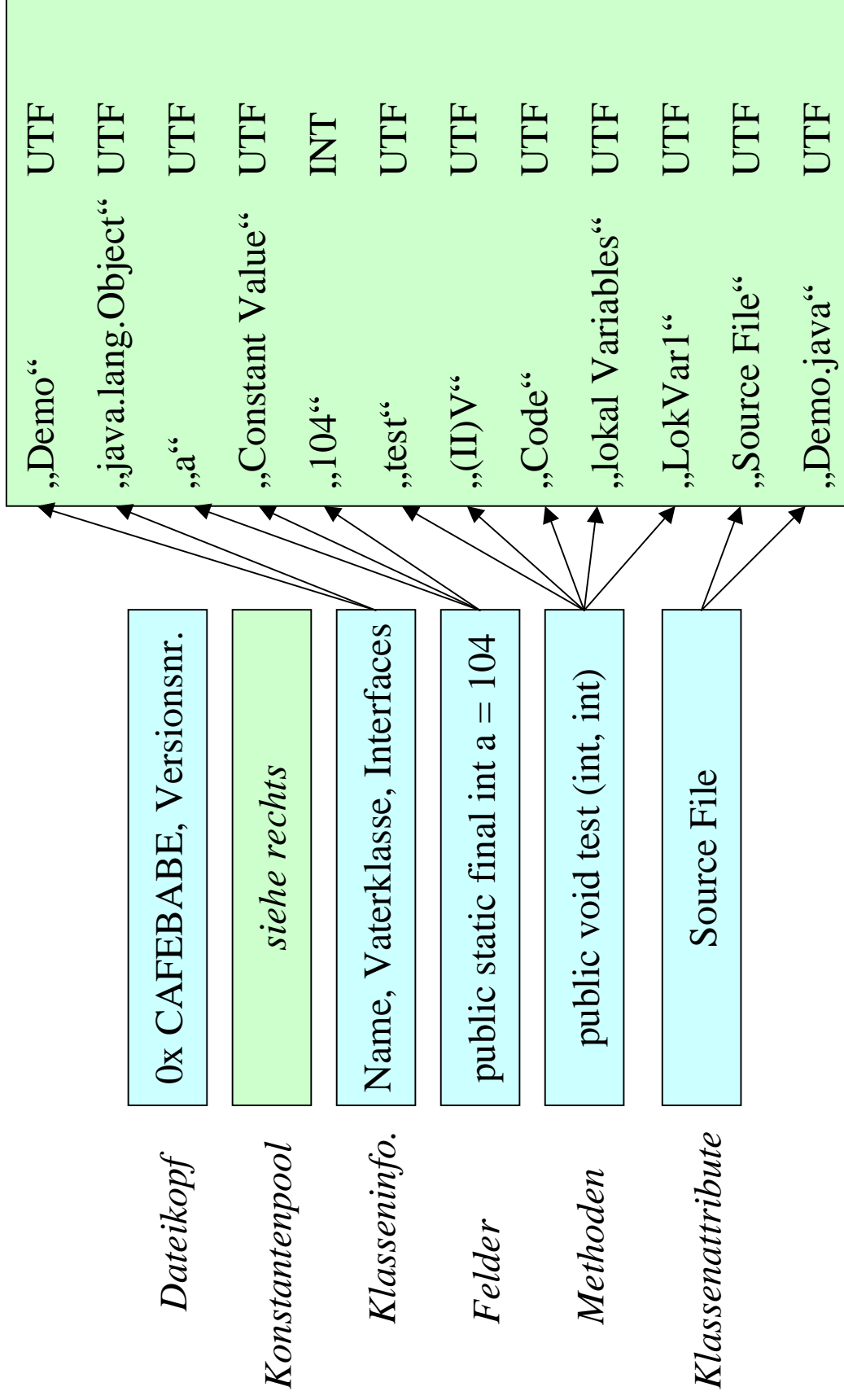


Byte-Code:

Bytecode-Repräsentation:

```
public class test {  
    public static void bsp()  
    {  
        int a = -1  
        double b = 1.0  
        double c = b + 3.0  
    }  
}  
  
method void bsp()  
0 iconst_m1 //push -1  
1 istore_0 //store -> a  
2 dconst_1 //push 1.0  
3 dstore_1 //store -> b  
4 dload_1 //push -> b  
5 ldc2_w #13 <Double 3.000..>  
8 dadd  
9 dstore_3 //store c
```





Sicherheitsarchitektur von Java

baut auf drei Ebenen auf:

- Byte Code Verifier
- Class Loader
- Security Manager

außerdem:

- Java ist eine sichere Sprache
z.B. keine Pointer, GC

Byte Code Verifier:

- stellt sicher, daß der Bytecode „gutartig“ ist
- besteht aus
 - syntaktischer Formatüberprüfung
 - 0xCAFEBAE, Dateilänge, Attributlänge
 - Bytecode-Verifikation
- Daten- und Kontrollflußanalyse
- Laufzeittests

Gefahren bei fehlerhafter Implementierung:

Bsp: unzulässige Typumwandlung von Double -> Referenz

1 Phase: *grundlegende Formatüberprüfung:*

- 0xCAFEBABE, Dateilänge, Attributlänge

2. Phase: *erweiterte Formatüberprüfung:*

- final Klassen / Methoden dürfen nicht überschrieben werden,
- Formatüberprüfung von Konstantenpoolinträgen,
- Überprüfung der Verweise auf Konstantenpool

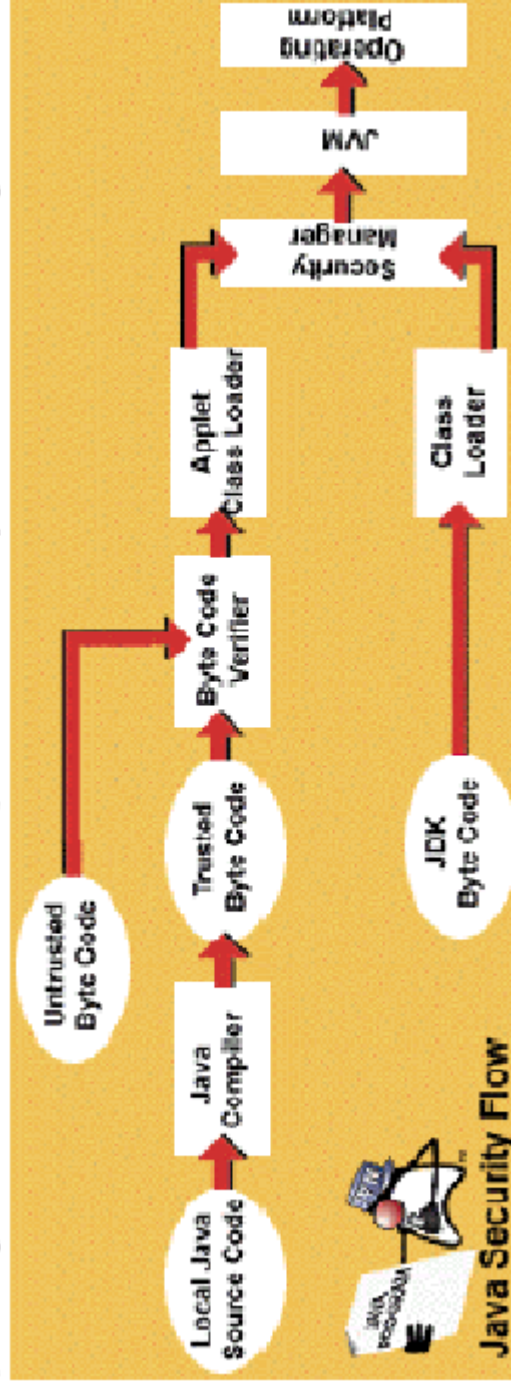
3. Phase: *Daten- & Kontrollflußanalyse*

- Instruktionen besitzen korrekte Operanden
- keine Stacküberläufe / Unterläufe
- Variablen werden vor Benutzung initialisiert
- Operandenstack besitzt gleiche Höhe
- Methodenaufruf mit geeigneten Parametern
- unabhängig von Ausführungspfad

4 Phase: Laufzeittests:

- nicht erkennbare Typkonflikte:
- Void test(B[] x, B y){
 x[0] = y }

Java Security Flow

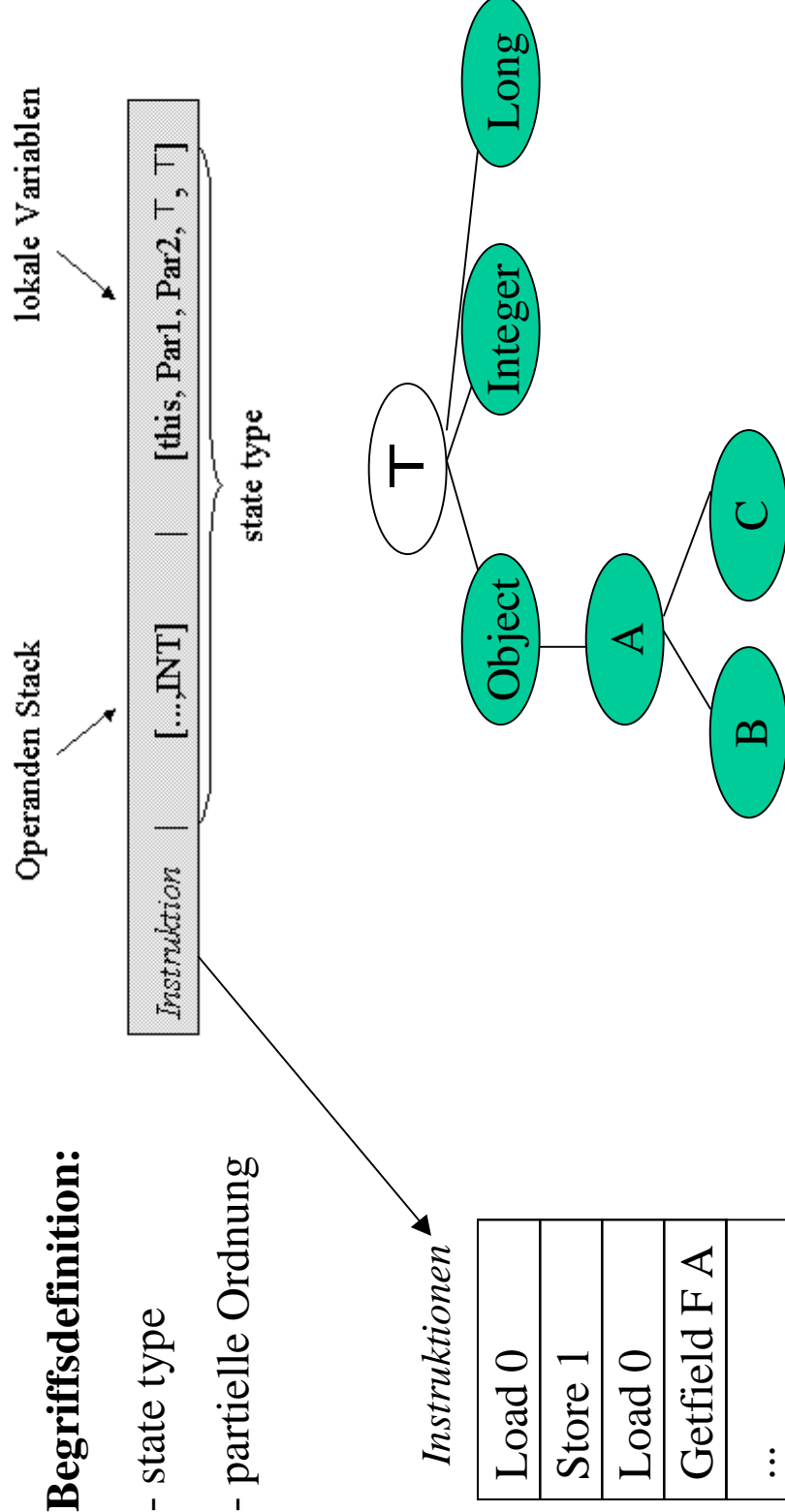


Daten- & Kontrollflußanalyse:

Begriffsdefinition:

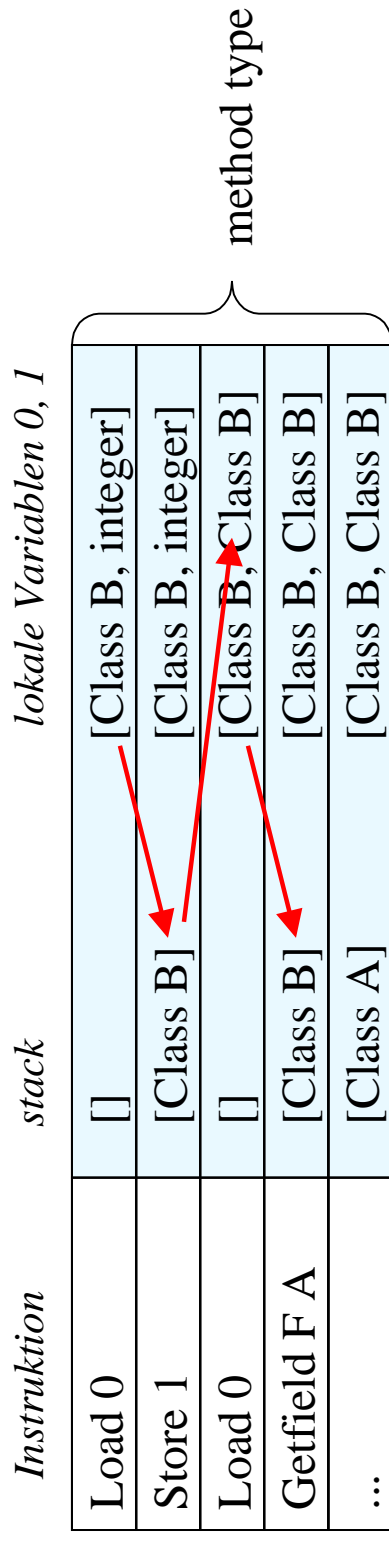
- state type

- partielle Ordnung



Daten- & Kontrollflußanalyse:

am Beispiel eine straight-line Codes

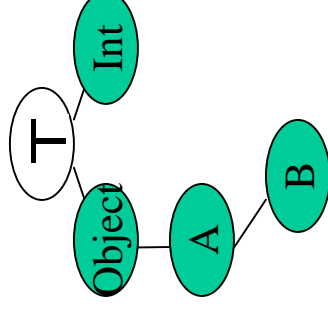


max. Speicherbedarf: $3*S + 3*V$ Bytes

Daten- & Kontrollflußanalyse:

am Beispiel eine Bytecodes mit Verzweigungen:

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



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

max. Speicherbedarf: $(3S + 3N + 3) * B$ Bytes.

Daten- & Kontrollflußanalyse:

Zusammenfassung:

- iteratives Verfahren
- Fixpunkt ist gefunden, wenn method type unverändert bleibt
- Verifikation scheitert bei:
 - Verwendung von Typ \top als Operand
 - Instruktionaufruf mit falschen Operanden
 - verschiedenen Stackhöhen von unterschiedlichen Ausführungspfaden
- jede Methode wird einzeln verifiziert

alternative Verfahren:

- Verifikationsnachweis mittels digitaler Signaturen
- Hashwert, private Key-Verschlüsselung

Vorteil:

on-card ist keine Verifikation mehr notwendig

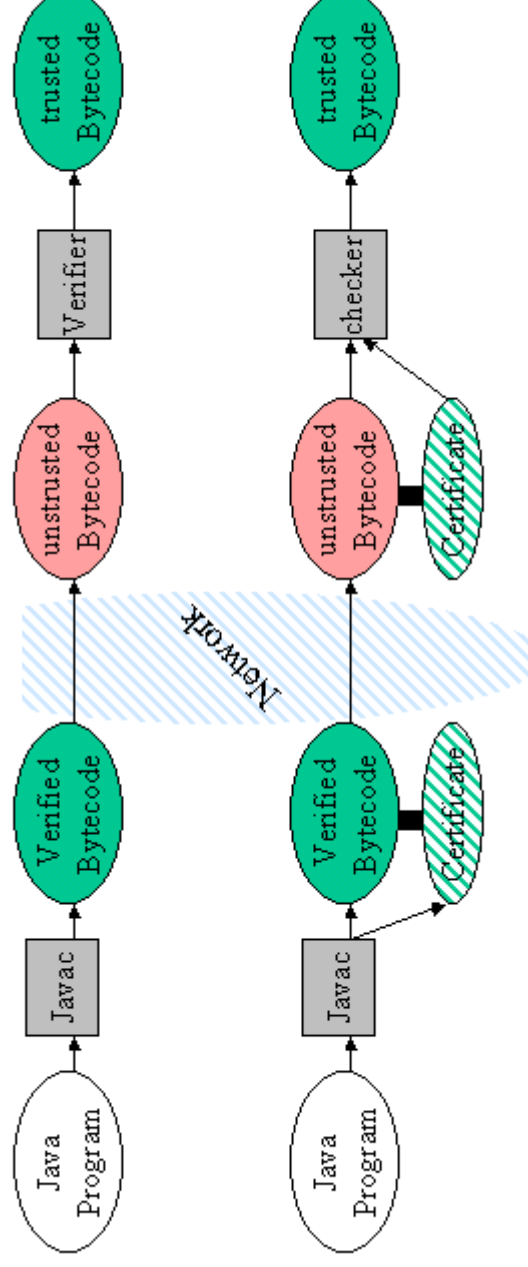
Problem:

- Sicherheit hängt von Geheimhaltung eines private Keys ab
- zusätzlich zum Bytecode muß die digitale Signatur übertragen werden

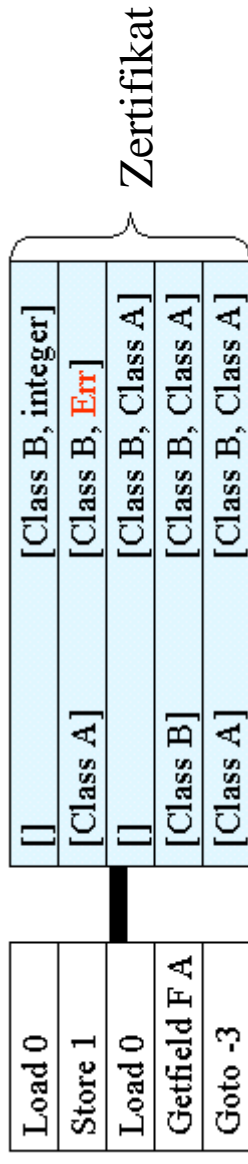
lightweight Bytecode Verifikation:

- von Kristoffer und Eva Rose veröffentlichtes BV-Verfahren
- bei off-card Verifikation wird Zertifikat erstellt
- mittels Zertifikat wird on-card Verifikation durchgeführt

=> Typ-Rekonstruktionsproblem => Typ-Überprüfungsproblem



lightweight Bytecode Verifikation:



Verifikationsprozeß benötigt:

- lineare Zeit
- Speicheranforderungen: Zertifikatgröße + $(3*S+3*N)$ Bytes

Vorteil: - sicherer als BV-Nachweis durch d.S.

- schneller als gewöhnliche BV

Nachteil: - Zertifikat muß zusätzlich übertragen werden

Verifikation nach Leroy:

- Idee:
- Stackinhalte bei *branches* und *joins* ist leer
 - lokale Variablen dürfen nur mit kompatiblen Typen benutzt werden

erweiterter straight-line Bytecode Verifikationsalgorithmus:

- überprüfe bei *branch* ob Stack leer ist
- überprüfe bei *joins* ob Stack leer ist
- überprüfe ob bei Store-Anweisungen Typenkompatibilität gilt
- ändert sich Typ einer lokalen Variablen => starte von neuem

Verifikation nach Leroy:

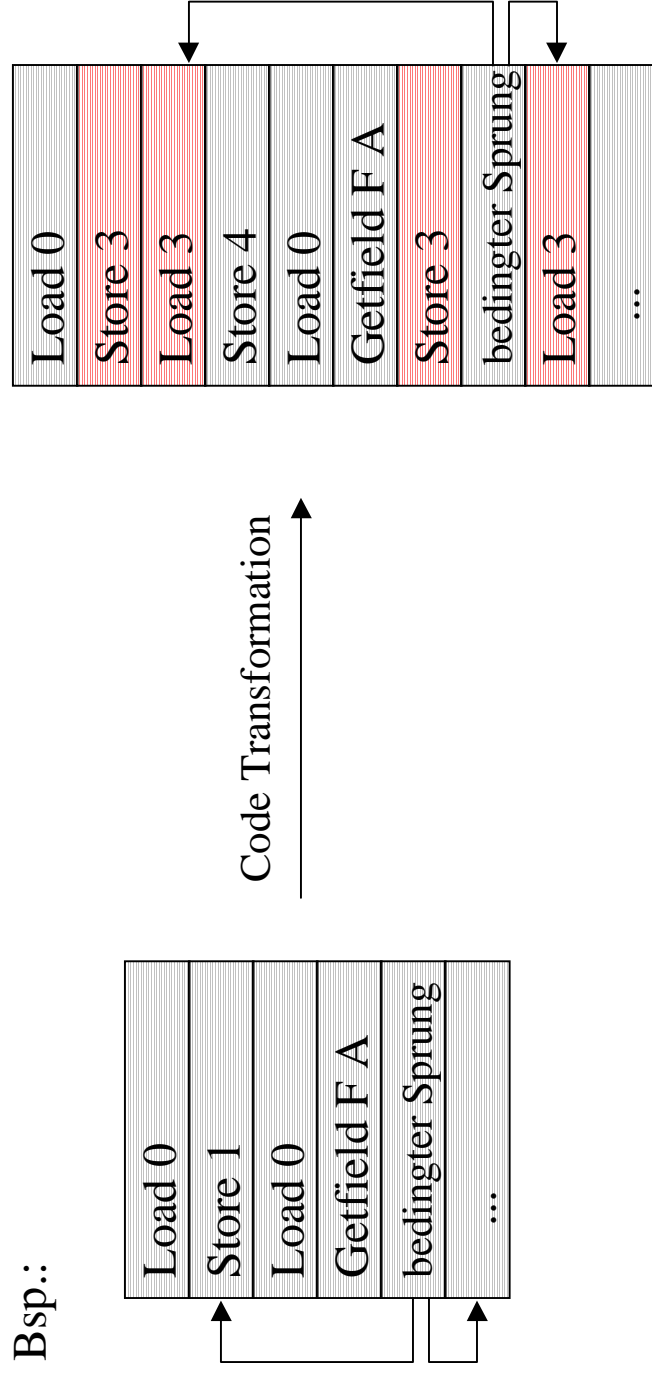
lokale Variable besitzt zwei nicht compatible Typen:

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

off-card Bytecode Transformation erforderlich:

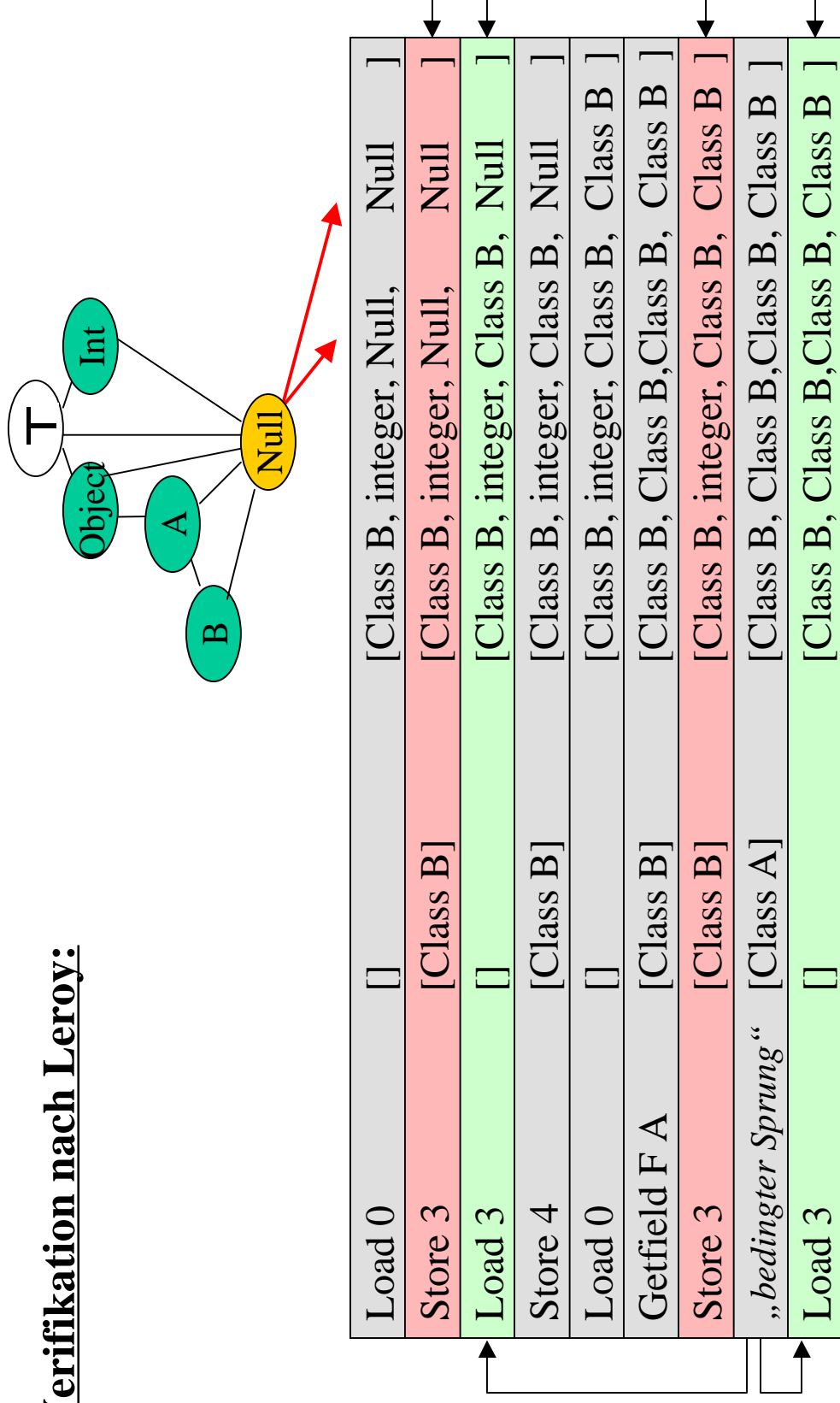
- Registerverwaltung
- Stacknormalisierung

Verifikation nach Leroy:



Variablenverwaltung und Instruktionsablauf sind optimierbar

Verifikation nach Leroy:



Speicheranforderungen: $(3 * S + 3 * N)$ Bytes

Verifikation nach Leroy:

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

Vorteile: - keine Übertragung von Zertifikat oder d.S. erforderlich
 - geringe Speicheranforderungen

Nachteil: - off-card-Transformation notwendig
 - Geschwindigkeitsverlust

Literatur:

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