

Static Program Analysis

Lecture 10: Dataflow Analysis IX (Java Bytecode Verification)

Thomas Noll

Lehrstuhl für Informatik 2
(Software Modeling and Verification)



noll@cs.rwth-aachen.de

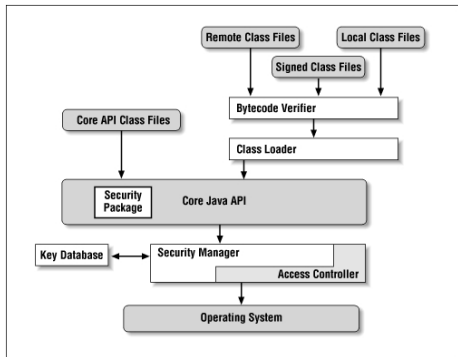
<http://moves.rwth-aachen.de/teaching/ws-1415/spa/>

Winter Semester 2014/15

- 1 Recap: The Java Virtual Machine
- 2 The Dataflow Analysis
- 3 Examples of Bytecode Verification
- 4 Further Issues in Bytecode Verification

Java Security: the Sandbox

- **Insulation layer** providing indirect access to system resources
- Hardware access via **API classes and methods**
- **Bytecode verification** upon uploading
 - well-typedness
 - proper object referencing
 - proper control flow



- Conventional **stack-based abstract machine**
- Supports **object-oriented features**: classes, methods, etc.
- **Stack** for intermediate results of expression evaluations
- **Registers** for source-level local variables and method parameters
- Both part of **method activation record**
(and thus preserved across method calls)
- Method entry point specifies **required number** of registers (m_r) and stack slots (m_s ; for memory allocation)
- (Most) instructions are **typed**

Correctness of Bytecode

Conditions to ensure **proper operation**:

Type correctness: arguments of instructions always of expected type

No stack over-/underflow: never push to full stack or pop from empty stack

Code containment: PC must always point into the method code

Register initialization: load from non-parameter register only after store

Object initialization: constructor must be invoked before using class instance

Access control: operations must respect visibility modifiers
(**private/protected/public**)

Options:

- **dynamic checking** at execution time (“defensive JVM approach”)
 - expensive, slows down execution
- **static checking** at loading time (here)
 - verified code executable at full speed without extra dynamic checks

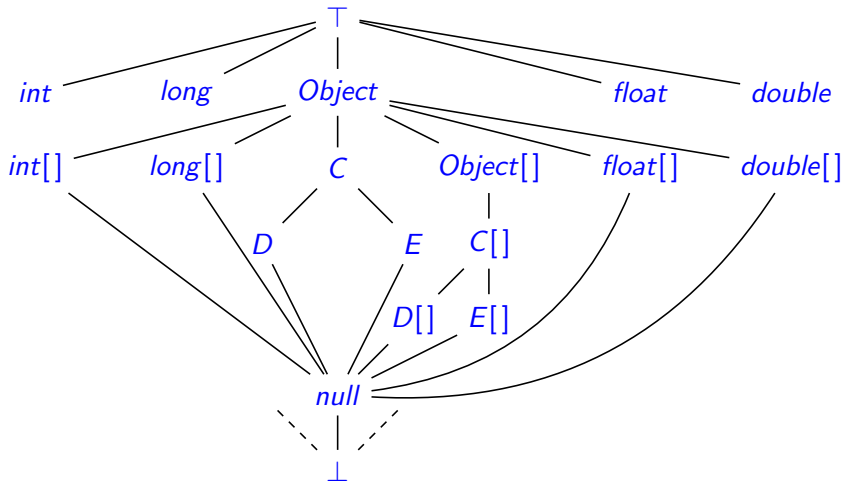
Summary: **dataflow analysis** applied to **type-level abstract interpretation** of JVM

- ① Association of **type information** with register and stack contents
 - set of types forms a complete lattice
- ② Simulation of **execution of instructions** at type level
- ③ Use **dataflow analysis** to cover all concrete executions
- ④ Modularity: analysis proceeds method **per method**

(see X. Leroy: *Java Bytecode Verification: Algorithms and Formalizations*, Journal of Automated Reasoning 30(3-4), 2003, 235–269)

The Subtyping Relation (excerpt)

(C , D , E user-defined classes; D , E extending C)



Notation: $\tau_1 \sqsubseteq_t \tau_2$

The Type-Level Abstract Interpreter I

- **Idea:** execute JVM instructions on **types** (rather than concrete values)
 - **stack type** $S \in Typ^{\leq m_s}$ (top to the left)
 - **register type** $R : \{0, \dots, m_r - 1\} \rightarrow Typ$
- Represented as **transition relation**

$$i : (S, R) \rightarrow (S', R')$$

where

- i : current instruction
- (S, R) : stack/register type before execution
- (S', R') : stack/register type after execution
- **Errors** (type mismatch, stack over-/underflow, ...) denoted by absence of transition

The Type-Level Abstract Interpreter II

Some transition rules:

<code>iconst_z</code> :	$(S, R) \rightarrow (int.S, R)$	if $ S < m_s$
<code>aconst_null</code> :	$(S, R) \rightarrow (null.S, R)$	if $ S < m_s$
<code>iadd</code> :	$(int.int.S, R) \rightarrow (int.S, R)$	
<code>if_icmpeq l</code> :	$(int.int.S, R) \rightarrow (S, R)$	
<code>iload n</code> :	$(S, R) \rightarrow (int.S, R)$	if $0 \leq n < m_r, R(n) = int, S < m_s$
<code>aload n</code> :	$(S, R) \rightarrow (R(n).S, R)$	if $0 \leq n < m_r, R(n) \sqsubseteq_t Object, S < m_s$
<code>istore n</code> :	$(int.S, R) \rightarrow (S, R[n \mapsto int])$	if $0 \leq n < m_r$
<code>astore n</code> :	$(\tau.S, R) \rightarrow (S, R[n \mapsto \tau])$	if $0 \leq n < m_r, \tau \sqsubseteq_t Object$
<code>getfield C f τ</code> :	$(D.S, R) \rightarrow (\tau.S, R)$	if $D \sqsubseteq_t C$
<code>putfield C f τ</code> :	$(\tau'.D.S, R) \rightarrow (S, R)$	if $\tau' \sqsubseteq_t \tau, D \sqsubseteq_t C$
<code>invoke C M σ</code> :	$(\tau'_n \dots \tau'_1.\tau'.S, R) \rightarrow (\tau_0.S, R)$	if $\sigma = \tau_0(\tau_1, \dots, \tau_n), \tau'_i \sqsubseteq_t \tau_i$ for $1 \leq i \leq n, \tau' \sqsubseteq_t C$

Some Theoretical Properties

Lemma

- 1 (Typ, \sqsubseteq_t) is a *complete lattice satisfying ACC*.
- 2 (*Determinacy*) The transitions of the abstract interpreter define a partial function: If $i : (S, R) \rightarrow (S_1, R_1)$ and $i : (S, R) \rightarrow (S_2, R_2)$, then $S_1 = S_2$ and $R_1 = R_2$.
- 3 (*Soundness*) If $i : (S, R) \rightarrow (S', R')$, then for all concrete states (s, r) matching (S, R) , the defensive JVM will not stop with a run-time type exception when applying i to (s, r) (but rather change to some (s', r') matching (S', R')).

Proof.

see X. Leroy: *Java Bytecode Verification: Algorithms and Formalizations* □

- 1 Recap: The Java Virtual Machine
- 2 The Dataflow Analysis
- 3 Examples of Bytecode Verification
- 4 Further Issues in Bytecode Verification

The Dataflow System I

The **dataflow system** $S = (Lab, E, F, (D, \sqsubseteq), \iota, \varphi)$ for a method M :

- **Labels** $Lab := \{\text{line numbers of Java bytecode}\}$

The Dataflow System I

The **dataflow system** $S = (Lab, E, F, (D, \sqsubseteq), \iota, \varphi)$ for a method M :

- **Labels** $Lab := \{\text{line numbers of Java bytecode}\}$
- **Extremal label** $E := \{1\}$ (forward problem)

The Dataflow System I

The **dataflow system** $S = (Lab, E, F, (D, \sqsubseteq), \iota, \varphi)$ for a method M :

- **Labels** $Lab := \{\text{line numbers of Java bytecode}\}$
- **Extremal label** $E := \{1\}$ (forward problem)
- **Flow relation** F : for every $l \in Lab$,
$$\left\{ \begin{array}{ll} (l, m), (l, l+1) \in F & \text{if } l: \text{ conditional jump to } m \\ (l, m) \in F & \text{if } l: \text{ unconditional jump to } m \\ - & \text{if } l: \text{ return instruction} \\ (l, l+1) & \text{otherwise} \end{array} \right.$$

The Dataflow System I

The **dataflow system** $S = (Lab, E, F, (D, \sqsubseteq), \iota, \varphi)$ for a method M :

- **Labels** $Lab := \{\text{line numbers of Java bytecode}\}$

- **Extremal label** $E := \{1\}$ (forward problem)

- **Flow relation** F : for every $l \in Lab$,

$$\begin{cases} (l, m), (l, l+1) \in F & \text{if } l: \text{ conditional jump to } m \\ (l, m) \in F & \text{if } l: \text{ unconditional jump to } m \\ - & \text{if } l: \text{ return instruction} \\ (l, l+1) & \text{otherwise} \end{cases}$$

- **Complete lattice** (D, \sqsubseteq) where

- $D := \underbrace{Typ^{\leq m_s}}_{\text{stack}} \times \underbrace{\{0, \dots, m_r - 1\}}_{\text{registers}} \rightarrow Typ \cup \{ \underbrace{None}_{\text{least element}}, \underbrace{Error}_{\text{untypeable}} \}$

- for every $(S, R) \in D$, $None \sqsubseteq (S, R)$ and $(S, R) \sqsubseteq Error$

- $(S_1, R_1) \sqsubseteq (S_2, R_2)$ iff

- $S_1 = \sigma_1 \dots \sigma_n$, $S_2 = \tau_1 \dots \tau_n$ (same length!), $\sigma_i \sqsubseteq_t \tau_i$ for $1 \leq i \leq n$
- $R_1(i) \sqsubseteq_t R_2(i)$ for $0 \leq i < m_r$

- Extremal value

$$v := (\tau_n \dots \tau_1, \underbrace{(\top, \dots, \top)}_{m_r \text{ times}})$$

with parameter types τ_1, \dots, τ_n of M

- Extremal value

$$v := (\tau_n \dots \tau_1, \underbrace{(\top, \dots, \top)}_{m_r \text{ times}})$$

with parameter types τ_1, \dots, τ_n of M

- Transfer functions $\{\varphi_l \mid l \in Lab\}$ are defined by

$$\varphi_l(S, R) := \begin{cases} (S', R') & \text{if } l : i \text{ and } i : (S, R) \rightarrow (S', R') \\ Error & \text{otherwise} \end{cases}$$

The Dataflow System II

- Extremal value

$$v := (\tau_n \dots \tau_1, \underbrace{(\top, \dots, \top)}_{m_r \text{ times}})$$

with parameter types τ_1, \dots, τ_n of M

- Transfer functions $\{\varphi_l \mid l \in Lab\}$ are defined by

$$\varphi_l(S, R) := \begin{cases} (S', R') & \text{if } l : i \text{ and } i : (S, R) \rightarrow (S', R') \\ Error & \text{otherwise} \end{cases}$$

Monotonicity of transfer functions is ensured by the following lemma.

Lemma 10.1

If $i : (S, R) \rightarrow (S', R')$ and $(S_1, R_1) \sqsubseteq (S, R)$, then there exists $(S'_1, R'_1) \in D$ such that $i : (S_1, R_1) \rightarrow (S'_1, R'_1)$ and $(S'_1, R'_1) \sqsubseteq (S', R')$.

Proof.

see X. Leroy: *Java Bytecode Verification: Algorithms and Formalizations* □

- 1 Recap: The Java Virtual Machine
- 2 The Dataflow Analysis
- 3 Examples of Bytecode Verification**
- 4 Further Issues in Bytecode Verification

Example of Correct Bytecode

Example 10.2

- Method declared by `method static C ... (B)` with $m_s = 2$, $m_r = 1$
- Classes `B` and `C` with $C \sqsubseteq_t B$
- `B` (and thus `C`) provides method `M` of type `C(int)`, field `f` of type `int`
- Application of fixpoint iteration: on the board

Label	Instruction	Transition rule (w/o conditions)
1	<code>astore 0</code>	$(\tau.S, R) \rightarrow (S, R[0 \mapsto \tau])$
2	<code>aload 0</code>	$(S, R) \rightarrow (R(0).S, R)$
3	<code>iconst_1</code>	$(S, R) \rightarrow (int.S, R)$
4	<code>invoke B M C(int)</code>	$(int.B.S, R) \rightarrow (C.S, R)$
5	<code>astore 0</code>	$(\tau.S, R) \rightarrow (S, R[0 \mapsto \tau])$
6	<code>aload 0</code>	$(S, R) \rightarrow (R(0).S, R)$
7	<code>getfield C f int</code>	$(C.S, R) \rightarrow (int.S, R)$
8	<code>iconst_0</code>	$(S, R) \rightarrow (int.S, R)$
9	<code>if_icmpeq 2</code>	$(int.int.S, R) \rightarrow (S, R)$
10	<code>aload 0</code>	$(S, R) \rightarrow (R(0).S, R)$
11	<code>areturn</code>	$(\tau.S, R) \rightarrow (\tau.S, R)$

Example of Malicious Bytecode

Example 10.3 (cf. Example 9.4)

- Assumption: class **A** provides field **f** of type **int**
- Program interprets second stack entry (5) as reference to **A**-object and assigns first stack entry (1) to field **f**
- $m_s = 2, m_r = 0$
- Application of worklist algorithm: on the board

Label	Instruction	Transition rule (w/o conditions)
1	<code>iconst_5</code>	$(S, R) \rightarrow (int.S, R)$
2	<code>iconst_1</code>	$(S, R) \rightarrow (int.S, R)$
3	<code>putfield A f int</code>	$(int.A.S, R) \rightarrow (S, R)$
4	<code>...</code>	

Theorem 10.4

If dataflow analysis yields $AI_l \neq \text{Error}$ for every $l \in \text{Lab}$, then the analyzed method *will not stop with a run-time type exception* when run on the JVM. Here run-time type exceptions refer to

- using instruction operands of wrong type (“Expecting to find ... on stack”),
- method return values of wrong type (“Wrong return value”),
- type-incompatible assignments to fields (“Incompatible type for setting field”),
- different stack sizes at the same location (“Inconsistent stack height”),
- stack overflows (i.e., more than m_s entries) (“Stack size too large”), and
- stack underflows (i.e., pop from empty stack) (“Unable to pop operand off an empty stack”).

- 1 Recap: The Java Virtual Machine
- 2 The Dataflow Analysis
- 3 Examples of Bytecode Verification
- 4 Further Issues in Bytecode Verification

Extended Basic Blocks

- **Idea:** set up dataflow equations for sequences of instructions (rather than single instructions)
- **Extended basic blocks:** maximal sequence of instructions with
 - jump targets only at beginning
 - (conditional or unconditional) jump and return instructions only at end

Extended Basic Blocks

- **Idea:** set up dataflow equations for sequences of instructions (rather than single instructions)
- **Extended basic blocks:** maximal sequence of instructions with
 - jump targets only at beginning
 - (conditional or unconditional) jump and return instructions only at end

Example 10.5 (cf. Example 9.3)

```
method static int factorial(int), 2 registers, 2 stack slots
  1: istore 0      // store n in register 0
  2: iconst_1     // push constant 1
  3: istore 1     // store res in register 1
  4: iload 0      // push register n
  5: ifle 12      // if <= 0, go to end
  6: iload 1      // push res
  7: iload 0      // push n
  8: imul         // res * n on top of stack
  9: istore 1     // store in res
 10: iinc 0, -1   // decrement n
 11: goto 4       // go to loop header
 12: iload 1      // push res
 13: ireturn     // return res to caller
```

(12 instructions)

Extended Basic Blocks

- **Idea:** set up dataflow equations for sequences of instructions (rather than single instructions)
- **Extended basic blocks:** maximal sequence of instructions with
 - jump targets only at beginning
 - (conditional or unconditional) jump and return instructions only at end

Example 10.5 (cf. Example 9.3)

```
method static int factorial(int), 2 registers, 2 stack slots
  1: istore 0      // store n in register 0
  2: iconst_1     // push constant 1
  3: istore 1     // store res in register 1
  4: iload 0      // push register n
  5: ifle 12      // if <= 0, go to end
  6: iload 1      // push res
  7: iload 0      // push n
  8: imul        // res * n on top of stack
  9: istore 1     // store in res
 10: iinc 0, -1   // decrement n
 11: goto 4       // go to loop header
 12: iload 1      // push res
 13: ireturn     // return res to caller
```

(4 extended basic blocks)

Bytecode Verification on Small Devices

(for details see X. Leroy: *Java Bytecode Verification: Algorithms and Formalizations*)

- **Problem:** bytecode verification is **expensive**
 - ⇒ can exceed resources of small embedded systems (mobile phones, smart cards, PDAs, ...)

(for details see X. Leroy: *Java Bytecode Verification: Algorithms and Formalizations*)

- **Problem:** bytecode verification is **expensive**
 - ⇒ can exceed resources of small embedded systems (mobile phones, smart cards, PDAs, ...)
- **Example: Java SmartCard**
 - 8-bit microprocessor
 - \approx 2 kB RAM (volatile, fast)
 - \approx 80 kB EEPROM (persistent, slow)
 - \approx 100 kB ROM (operating system)
 - ⇒ RAM too small to store dataflow infos

(for details see X. Leroy: *Java Bytecode Verification: Algorithms and Formalizations*)

- **Problem:** bytecode verification is **expensive**
 - ⇒ can exceed resources of small embedded systems (mobile phones, smart cards, PDAs, ...)
- **Example: Java SmartCard**
 - 8-bit microprocessor
 - \approx 2 kB RAM (volatile, fast)
 - \approx 80 kB EEPROM (persistent, slow)
 - \approx 100 kB ROM (operating system)
 - ⇒ RAM too small to store dataflow infos
- **Solutions:**
 - Use **EEPROM** to hold verifier data structures (slow)
 - **Off-card verification** using **certificates** (see following slides)
 - **On-card verification** with **off-card code transformation** (see following slides)

Off-Card Verification Using Certificates

(also: “lightweight bytecode verification using certificates”)

- Inspired by “proof-carrying code approach”
- Bytecode producer **attaches type information** to bytecode (“certificates”)
- Embedded system **checks well-typedness** of code (rather than inferring types)
- Advantages:
 - type checking **faster** than inference (no fixpoint iteration)
 - only **reading access** to certificates \implies can be kept in EEPROM
- Practical limitation: certificates require \approx **50% of size of annotated code**
- Implementation: **Sun’s K Virtual Machine** (KVM)

On-Card Verification with Off-Card Transformation

- Standard bytecode verification (solving dataflow equations using fixpoint iteration) on **normalized bytecode**
- Bytecode restrictions:
 - only **one register type** shared by all control points
(= entry points of extended basic blocks)
 - **stack empty** before each jump target and after each jump instruction
(= entry/exit points of extended basic blocks)
- **Space complexity** of bytecode verification
($|Lab|/m_s/m_r$ = number of blocks/stack entries/registers):
 - without restriction: $\mathcal{O}(|Lab| \cdot (m_s + m_r))$
 - with restriction: $\mathcal{O}(m_s + m_r)$
- Restrictions ensured by off-card (i.e., compile-time) **code transformation**
 - stack normalizations around jumps
 - register re-allocation by graph coloring
 - can increase code size and number of used registers
(but negligible on “typical” Java Card code)