Static Program Analysis

Lecture 10: Dataflow Analysis IX (Java Bytecode Verification)

Winter Semester 2016/17

Thomas Noll
Software Modeling and Verification Group
RWTH Aachen University

https://moves.rwth-aachen.de/teaching/ws-1617/spa/
Recap: The Java Virtual Machine

Java Security: the Sandbox

- **Insulation layer** providing indirect access to system resources
- **Hardware access** via API classes and methods
- **Bytecode verification** upon loading
  - well-typedness
  - proper object referencing
  - proper control flow
Recap: The Java Virtual Machine

The Java Virtual Machine

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

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
Recap: The Java Virtual Machine

The Java Bytecode Verifier

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 (“symbolic execution”)
3. Use dataflow analysis to cover all concrete executions
4. Modular analysis: proceeds method per method

Recap: The Java Virtual Machine

The Subtyping Relation (excerpt)

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

Notation: \( \tau_1 \sqsubseteq \tau_2 \)
Recap: The Java Virtual Machine

The Type-Level Abstract Interpreter

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

- 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
Recap: The Java Virtual Machine

Some Transition Rules

\begin{align*}
\text{iconst}_z &: (S, R) \rightarrow (\text{int}.S, R) & \text{if } |S| < m_s \\
\text{aconst_null} &: (S, R) \rightarrow (\text{null}.S, R) & \text{if } |S| < m_s \\
\text{iadd} &: (\text{int}.\text{int}.S, R) \rightarrow (\text{int}.S, R) \\
\text{if/icmpeq l} &: (\text{int}.\text{int}.S, R) \rightarrow (S, R) \\
\text{iload n} &: (S, R) \rightarrow (\text{int}.S, R) & \text{if } 0 \leq n < m_r, R(n) = \text{int}, |S| < m_s \\
\text{aload n} &: (S, R) \rightarrow (R(n).S, R) & \text{if } 0 \leq n < m_r, R(n) \sqsubseteq_t \text{Object}, |S| < m_s \\
\text{istore n} &: (\text{int}.S, R) \rightarrow (S, R[n \mapsto \text{int}]) & \text{if } 0 \leq n < m_r \\
\text{astore n} &: (\tau.S, R) \rightarrow (S, R[n \mapsto \tau]) & \text{if } 0 \leq n < m_r, \tau \sqsubseteq_t \text{Object} \\
\text{getfield C f} \tau &: (D.S, R) \rightarrow (\tau.S, R) & \text{if } D \sqsubseteq_t C \\
\text{putfield C f} \tau &: (\tau'.D.S, R) \rightarrow (S, R) & \text{if } \tau' \sqsubseteq_t \tau, D \sqsubseteq_t C \\
\text{invoke C M} \sigma &: (\tau'_n \ldots \tau'_1.\tau'.S, R) \rightarrow (\tau_0.S, R) & \text{if } \sigma = \tau_0(\tau_1, \ldots, \tau_n), \tau'_i \sqsubseteq_t \tau_i \text{ for } 1 \leq i \leq n, \tau' \sqsubseteq_t C
\end{align*}
Recap: The Java Virtual Machine

Some Theoretical Properties

Lemma

1. \((\text{Typ}, \subseteq_t)\) is a complete lattice satisfying ACC.

2. (Determinacy) The transitions of the abstract interpreter define a partial function:
   \[ i : (S, R) \rightarrow (S_1, R_1) \text{ and } i : (S, R) \rightarrow (S_2, R_2), \text{ then } S_1 = S_2 \text{ and } R_1 = R_2. \]

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
The Dataflow Analysis

The Dataflow System I

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

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

- **Complete lattice** $(D, \sqsubseteq)$ where
  
  - $D := \text{Typ}^{\leq m_s}_{\text{stack}} \times \{0, \ldots, m_r - 1\} \rightarrow \text{Typ} \cup \{\text{None}, \text{Error}\}$
  
  - for every $(S, R) \in D$, $\text{None} \sqsubseteq (S, R)$ and $(S, R) \sqsubseteq \text{Error}$
  
  - $(S_1, R_1) \sqsubseteq (S_2, R_2)$ iff
    
    - $S_1 = \sigma_1 \ldots \sigma_n$, $S_2 = \tau_1 \ldots \tau_n$ (same length $n \in \mathbb{N}$!), $\sigma_i \sqsubseteq \tau_i$ for $1 \leq i \leq n$
    
    - $R_1(i) \sqsubseteq R_2(i)$ for $0 \leq i < m_r$
The Dataflow Analysis

The Dataflow System II

- **Extremal value** (for parameter types $\tau_1, \ldots, \tau_n$ of $M$):
  \[
  \iota := (\tau_n \ldots \tau_1, (\underbrace{T, \ldots, T})_m)
  \]

- **Transfer functions** $\{\varphi_l \mid l \in \text{Lab}\}$:
  \[
  \varphi_l(S, R) := \begin{cases} (S', R') & \text{if } l : i \text{ and } i : (S, R) \to (S', R') \\ \text{Error} & \text{otherwise} \end{cases}
  \]

**Monotonicity** of transfer functions is ensured by the following lemma.

**Lemma 10.1**

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

**Proof.**

see X. Leroy: *Java Bytecode Verification: Algorithms and Formalizations*
Examples of 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 \( C(\text{int}) \), field \( f \) of type \( \text{int} \)
- Application of worklist algorithm (omitting worklist):

<table>
<thead>
<tr>
<th>Label</th>
<th>Instruction</th>
<th>Transition rule (w/o conditions)</th>
<th>((S, R_0)) (at entry)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>astore 0</td>
<td>((\tau.S, R) \rightarrow (S, R[0 \mapsto \tau]))</td>
<td>((B, \top))</td>
</tr>
<tr>
<td>2</td>
<td>aload 0</td>
<td>((S, R) \rightarrow (R(0).S, R))</td>
<td>None</td>
</tr>
<tr>
<td>3</td>
<td>iconst_1</td>
<td>((S, R) \rightarrow (\text{int}.S, R))</td>
<td>None</td>
</tr>
<tr>
<td>4</td>
<td>invoke B M C(\text{int})</td>
<td>((\text{int}.B.S, R) \rightarrow (C.S, R))</td>
<td>None</td>
</tr>
<tr>
<td>5</td>
<td>astore 0</td>
<td>((\tau.S, R) \rightarrow (S, R[0 \mapsto \tau]))</td>
<td>None</td>
</tr>
<tr>
<td>6</td>
<td>aload 0</td>
<td>((S, R) \rightarrow (R(0).S, R))</td>
<td>None</td>
</tr>
<tr>
<td>7</td>
<td>getfield C f \text{int}</td>
<td>((C.S, R) \rightarrow (\text{int}.S, R))</td>
<td>None</td>
</tr>
<tr>
<td>8</td>
<td>iconst_0</td>
<td>((S, R) \rightarrow (\text{int}.S, R))</td>
<td>None</td>
</tr>
<tr>
<td>9</td>
<td>if_icmpeq 2</td>
<td>((\text{int}.\text{int}.S, R) \rightarrow (S, R))</td>
<td>None</td>
</tr>
<tr>
<td>10</td>
<td>aload 0</td>
<td>((S, R) \rightarrow (R(0).S, R))</td>
<td>None</td>
</tr>
<tr>
<td>11</td>
<td>areturn</td>
<td>((\tau.S, R) \rightarrow (\tau.S, R))</td>
<td>None</td>
</tr>
</tbody>
</table>

- Thus: no type errors, expected return type \((C)\)
Examples of Bytecode Verification

Example of Malicious Bytecode

Example 10.3 (cf. Example 9.4)

- Assumption: class \( A \) provides field \( f \) of type \( \text{int} \)
- 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 (omitting worklist):

<table>
<thead>
<tr>
<th>Label</th>
<th>Instruction</th>
<th>Transition rule (w/o conditions) (( S, R )) (at entry)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( l )</td>
<td>( \ldots )</td>
<td>( \ldots )</td>
</tr>
<tr>
<td>( l + 1 )</td>
<td>( \text{iconst}_5 )</td>
<td>( (S, R) \rightarrow (\text{int}.\ S, R) ) ( (\varepsilon, \varepsilon) )</td>
</tr>
<tr>
<td>( l + 2 )</td>
<td>( \text{iconst}_1 )</td>
<td>( (S, R) \rightarrow (\text{int}.\ S, R) ) ( \text{None} )</td>
</tr>
<tr>
<td>( l + 3 )</td>
<td>( \text{putfield} \ A \ f \ \text{int} ) (( \text{int}.\ A.S, R )) ( \rightarrow (S, R) ) ( \text{None} )</td>
<td></td>
</tr>
<tr>
<td>( l + 4 )</td>
<td>( \ldots )</td>
<td>( \ldots ) ( \text{None} )</td>
</tr>
</tbody>
</table>
Examples of Bytecode Verification

Soundness of Bytecode Verifier

Theorem 10.4

If dataflow analysis yields $A_l \neq \text{Error}$ for every $l \in \text{Lab}$, then the analysed 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").
Further Issues in Bytecode Verification

Extended Basic Blocks

- **Idea:** set up transfer functions 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)

```java
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 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) (4 extended basic blocks)
```
Further Issues in Bytecode Verification

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, ...)

- **Example:** Java SmartCard
  - 8-bit microprocessor
  - \(\leq 5\) kB RAM (volatile, fast)
  - \(\leq 256\) kB EEPROM (persistent, slow)
  - \(\leq 256\) 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)
Further Issues in Bytecode Verification

Off-Card Verification Using Certificates

(aka “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 can be kept in EEPROM
- Practical limitation: certificates require ≈ 50% of size of annotated code
- Implementation: Sun’s K Virtual Machine (KVM)
Further Issues in Bytecode Verification

On-Card Verification with Off-Card Transformation

- Standard bytecode verification (solving dataflow equations using fixpoint iteration) on normalized bytecode
- Bytecode restrictions:
  - organised as extended basic blocks (cf. Slide 10.19)
  - 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: $O(|Lab| \cdot (m_s + m_r))$
  - with restriction: $O(m_s + m_r)$
    - $m_s$: for stack type analysis of single extended basic block
    - $m_r$: for global register type
- 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)