Static Program Analysis

Lecture 10: Dataflow Analysis IX (Java Bytecode Verification)

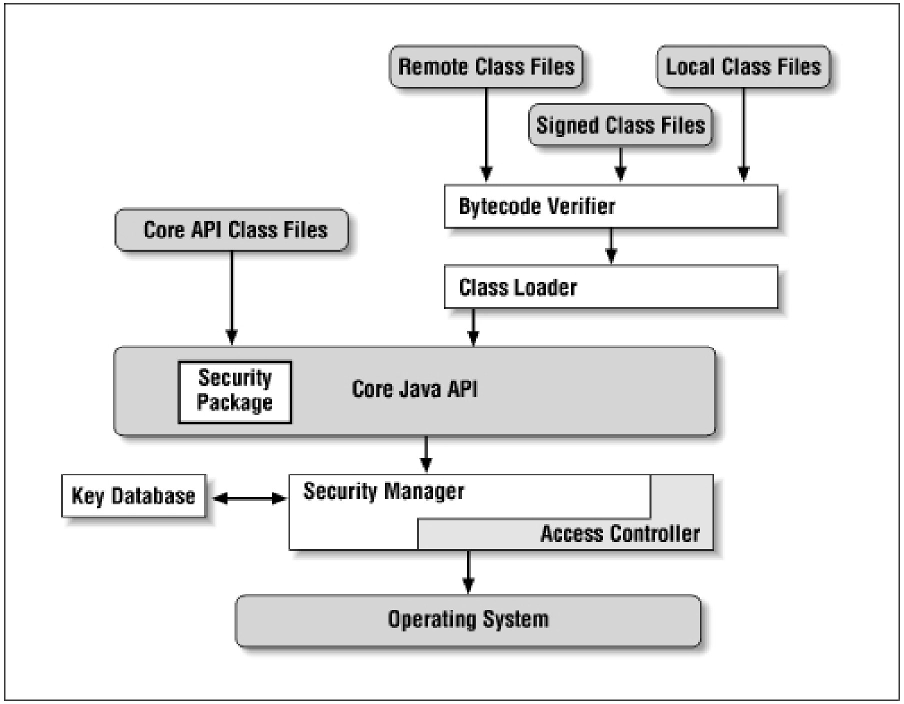
Winter Semester 2016/17

Thomas Noll
Software Modeling and Verification Group
RWTH Aachen University

https://moves.rwth-aachen.de/teaching/ws-1617/spa/

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

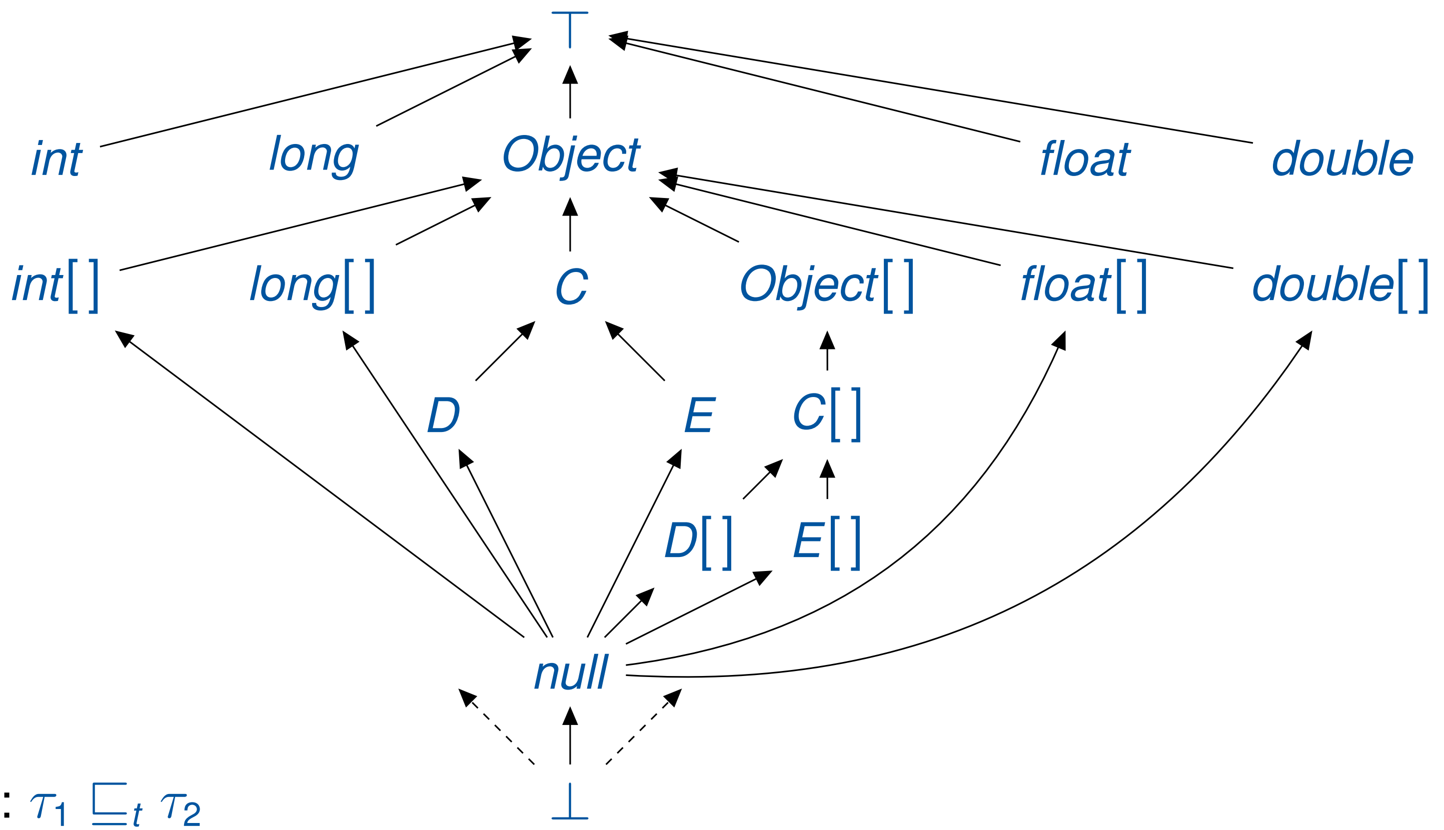
1. Association of type information with register and stack contents
   – set of types forms a complete lattice
2. Simulation of execution of instructions at type level (“symbolic execution”)
3. Use dataflow analysis to cover all concrete executions
4. Modular analysis: proceeds method per method

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

Recap: The Java Virtual Machine

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

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

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

  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

Some Transition Rules

$$
\begin{array}{lrr}
\texttt{iconst\_}z: & (S,R) \to (int.S,R) & \text{if } |S| < m_s \\
\texttt{aconst\_null}: & (S,R) \to (null.S,R) & \text{if } |S| < m_s \\
\texttt{iadd}: & (int.int.S,R) \to (int.S,R) & \\
\texttt{if\_icmpeq}\ l: & (int.int.S,R) \to (S,R) & \\
\texttt{iload}\ n: & (S,R) \to (int.S,R) & \\
 & \multicolumn{2}{r}{\text{if } 0 \le n < m_r, R(n) = int, |S| < m_s} \\
\texttt{aload}\ n: & (S,R) \to (R(n).S,R) & \\
 & \multicolumn{2}{r}{\text{if } 0 \le n < m_r, R(n) \sqsubseteq_t Object, |S| < m_s} \\
\texttt{istore}\ n: & (int.S,R) \to (S,R[n \mapsto int]) & \text{if } 0 \le n < m_r \\
\texttt{astore}\ n: & (\tau.S,R) \to (S,R[n \mapsto \tau]) & \\
 & \multicolumn{2}{r}{\text{if } 0 \le n < m_r, \tau \sqsubseteq_t Object} \\
\texttt{getfield}\ C\ f\ \tau: & (D.S,R) \to (\tau.S,R) & \text{if } D \sqsubseteq_t C \\
\texttt{putfield}\ C\ f\ \tau: & (\tau'.D.S,R) \to (S,R) & \text{if } \tau' \sqsubseteq_t \tau, D \sqsubseteq_t C \\
\texttt{invoke}\ C\ M\ \sigma: & (\tau'_n \ldots \tau'_1.\tau'.S,R) \to (\tau_0.S,R) & \\
 & \multicolumn{2}{r}{\text{if } \sigma = \tau_0(\tau_1,\ldots,\tau_n), \tau'_i \sqsubseteq_t \tau_i \text{ for } 1 \le i \le n, \tau' \sqsubseteq_t C}
\end{array}
$$

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* $i : (S, R) \to (S_1, R_1)$ *and* $i : (S, R) \to (S_2, R_2)$*, then* $S_1 = S_2$ *and* $R_1 = R_2$*.*
3. *(Soundness) If* $i : (S, R) \to (S', R')$*, 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 System I

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

- Labels $Lab :=$ {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
  - $D := \underbrace{Typ^{\leq m_s}}_{\text{stack}} \times \underbrace{\{0, \ldots, m_r - 1\} \to Typ}_{\text{registers}} \cup \{ \underbrace{None}_{\text{least element}}, \underbrace{Error}_{\text{untypeable}} \}$
  - 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
    - $S_1 = \sigma_1 \ldots \sigma_n$, $S_2 = \tau_1 \ldots \tau_n$ (same length $n \in \mathbb{N}$!), $\sigma_i \sqsubseteq_t \tau_i$ for $1 \leq i \leq n$
    - $R_1(i) \sqsubseteq_t R_2(i)$ for $0 \leq i < m_r$

The Dataflow System II

- Extremal value (for parameter types $\tau_1, \ldots, \tau_n$ of M):

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

- Transfer functions $\{\varphi_l \mid l \in Lab\}$:

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

Monotonicity of transfer functions is ensured by the following lemma.

Lemma 10.1

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

Proof.

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

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 worklist algorithm (omitting worklist):

Label	Instruction	Transition rule (w/o conditions)	(S, R_0) (at entry)
1	`astore 0`	$(\tau.S, R) \to (S, R[0 \mapsto \tau])$	$(\mathtt{B}, \top)$
2	`aload 0`	$(S, R) \to (R(0).S, R)$	*None*
3	`iconst_1`	$(S, R) \to (int.S, R)$	*None*
4	`invoke B M C(int)`	$(int.\mathtt{B}.S, R) \to (\mathtt{C}.S, R)$	*None*
5	`astore 0`	$(\tau.S, R) \to (S, R[0 \mapsto \tau])$	*None*
6	`aload 0`	$(S, R) \to (R(0).S, R)$	*None*
7	`getfield C f int`	$(\mathtt{C}.S, R) \to (int.S, R)$	*None*
8	`iconst_0`	$(S, R) \to (int.S, R)$	*None*
9	`if_icmpeq 2`	$(int.int.S, R) \to (S, R)$	*None*
10	`aload 0`	$(S, R) \to (R(0).S, R)$	*None*
11	`areturn`	$(\tau.S, R) \to (\tau.S, R)$	*None*

- Thus: no type errors, expected return type (`C`)

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 (omitting worklist):

Label	Instruction	Transition rule (w/o conditions)	(S, R) (at entry)
l	...	...	...
$l+1$	`iconst_5`	$(S, R) \to (int.S, R)$	$(\varepsilon, \varepsilon)$
$l+2$	`iconst_1`	$(S, R) \to (int.S, R)$	*None*
$l+3$	`putfield A f int`	$(int.\mathtt{A}.S, R) \to (S, R)$	*None*
$l+4$	...	...	*None*

Soundness of Bytecode Verifier

Theorem 10.4

If dataflow analysis yields $\mathsf{AI}_l \neq \mathit{Error}$ for every $l \in \mathit{Lab}$, 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`").*

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)

```
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) (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
  $\Longrightarrow$ 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)

  $\Longrightarrow$ 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 $\Longrightarrow$ can be kept in EEPROM
- Practical limitation: certificates require $\approx$ 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: $\mathcal{O}(|Lab| \cdot (m_s + m_r))$
  - with restriction: $\mathcal{O}(m_s + m_r)$
    - 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)