Static Program Analysis

Lecture 10: Dataflow Analysis IX (The Java Bytecode Verifier)

Thomas Noll

Lehrstuhl für Informatik 2
(Software Modeling and Verification)

RWTH Aachen University

noll@cs.rwth-aachen.de

http://www-i2.informatik.rwth-aachen.de/i2/spa11/

Summer Semester 2011

Dauerwerbesendung

- **Informatik-Kolloquium:**
  Uwe Schöning (Univ. Ulm):

  Das SAT-Problem und lokale Suchmethoden

  15:00 Uhr, AH 4

Dauerwerbesendung

- **Informatik-Kolloquium:**
  Uwe Schöning (Univ. Ulm):

  Das SAT-Problem und lokale Suchmethoden

  15:00 Uhr, AH 4

- **HiWis wanted:**
  - Organisatorische Unterstützung der

    Aachen Concurrency and Dependability Week
  - 05.-10.09.2011, SuperC
  - Technischer Support in Hörsälen, Bestuhlung, Kaffeepausen, ...
  - Vertrag über 1 Monat (ca. 900 €)
  - Ggf. Besuch der Vortragsveranstaltungen möglich
  - Kontakt: Sabrina von Styp
    (`sabrina.von-styp@cs.rwth-aachen.de`)

Outline

Java Bytecode

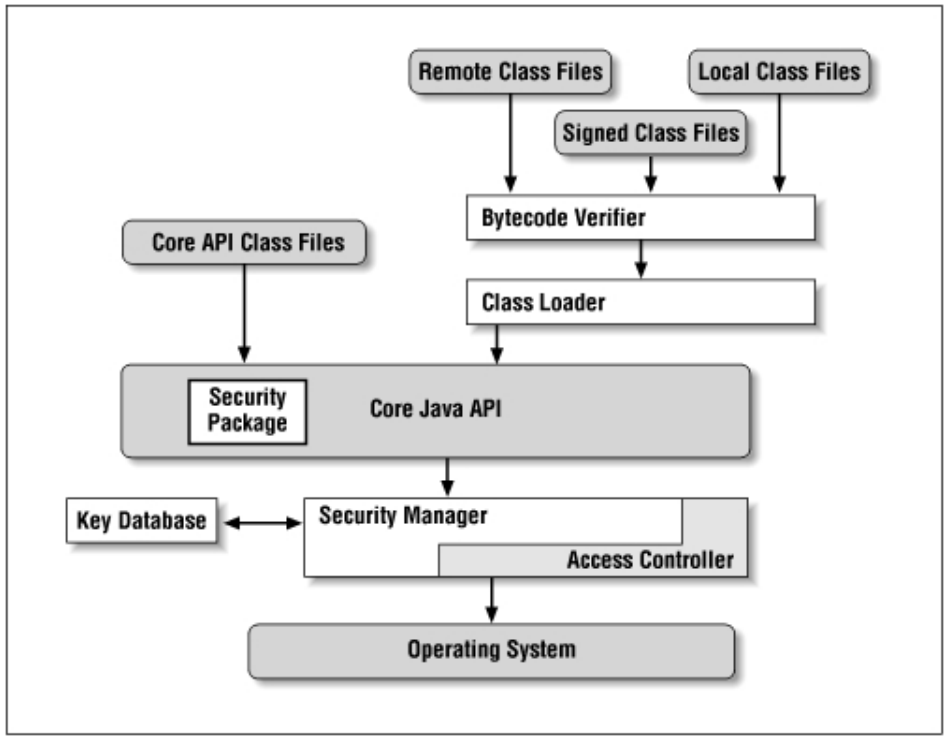
- Intermediate language between high-level language and machine code
- Execution on Java Virtual Machine (JVM)

Java Bytecode

- Intermediate language between high-level language and machine code
- Execution on Java Virtual Machine (JVM)
- **Advantages:**
  - architecture independency (especially for web applications)
  - faster than pure (i.e., source code) interpretation
- **Problem:** security issues
  - destruction of data
  - modification of data
  - disclosure of personal information
  - modification of other programs

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



The Java Virtual Machine

- 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

Example: Factorial Function

Example 10.1 (Factorial function)

Source Java code:

```
static int factorial(int n)
{ int res;
  for (res = 1; n > 0; n--) res = res * n;
  return res; }
```

Example: Factorial Function

Example 10.1 (Factorial function)

Source Java code:

```
static int factorial(int n)
{ int res;
  for (res = 1; n > 0; n--) res = res * n;
  return res; }
```

Corresponding JVM bytecode:

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

JVM Instruction Set (excerpt)

`iload` n: push integer from register n

`istore` n: pop integer into register n

`iconst_`z: push integer z

`aconst_null`: push null reference

`iadd`: add two topmost integers on stack and push sum

`getfield` C f τ: pops reference to object (of class C) and pushes value of field f (of type τ)

`putfield` C f τ: pops value v (of type τ) and reference to object o (of class C) and assigns v to field f of o

`new` C: creates new object (of class C) and pushes reference

`invoke` C M $\tau_0(\tau_1, \ldots, \tau_n)$: pops values $v_1, \ldots, v_n$ (of type $\tau_1, \ldots, \tau_n$) and reference to object (of class C), calls method M with parameters $v_1, \ldots, v_n$, and pushes return value (of type τ_0)

`if_icmpeq` l: pop two topmost integers from stack and jump to line l if equal

`ireturn`: return to caller with integer result on top of stack

($\approx$ 200 instructions in total)

Malicious Bytecode

Example 10.2 (Malicious bytecode)

```
1: iconst_5
2: iconst_1
3: putfield A f int
```

interprets second stack entry (5) as reference to object of class `A` and assigns first stack entry (1) to field `f` of this object

Outline

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`)

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

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
3. Use dataflow analysis to cover all concrete executions
4. Analysis proceeds method per method

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

Outline

Types

The set of types, *Typ*, is composed of

- Primitive types:
  - *int* (covering *boolean*, *byte*, *char*, *short*)
  - *long*
  - *float*
  - *double*

Types

The set of types, Typ, is composed of

- Primitive types:
  - *int* (covering *boolean*, *byte*, *char*, *short*)
  - *long*
  - *float*
  - *double*
- Object reference types: C for every class name C

Types

The set of types, *Typ*, is composed of

- Primitive types:
  - *int* (covering *boolean*, *byte*, *char*, *short*)
  - *long*
  - *float*
  - *double*
- Object reference types: C for every class name C
- Array types: $\tau[]$ for every primitive or object reference type τ

Types

The set of types, Typ, is composed of

- Primitive types:
  - *int* (covering *boolean*, *byte*, *char*, *short*)
  - *long*
  - *float*
  - *double*
- Object reference types: C for every class name C
- Array types: $\tau[]$ for every primitive or object reference type τ
- Method types: $\tau_0(\tau_1, \ldots, \tau_n)$ for $n \in \mathbb{N}$, $\tau_i \in Typ$

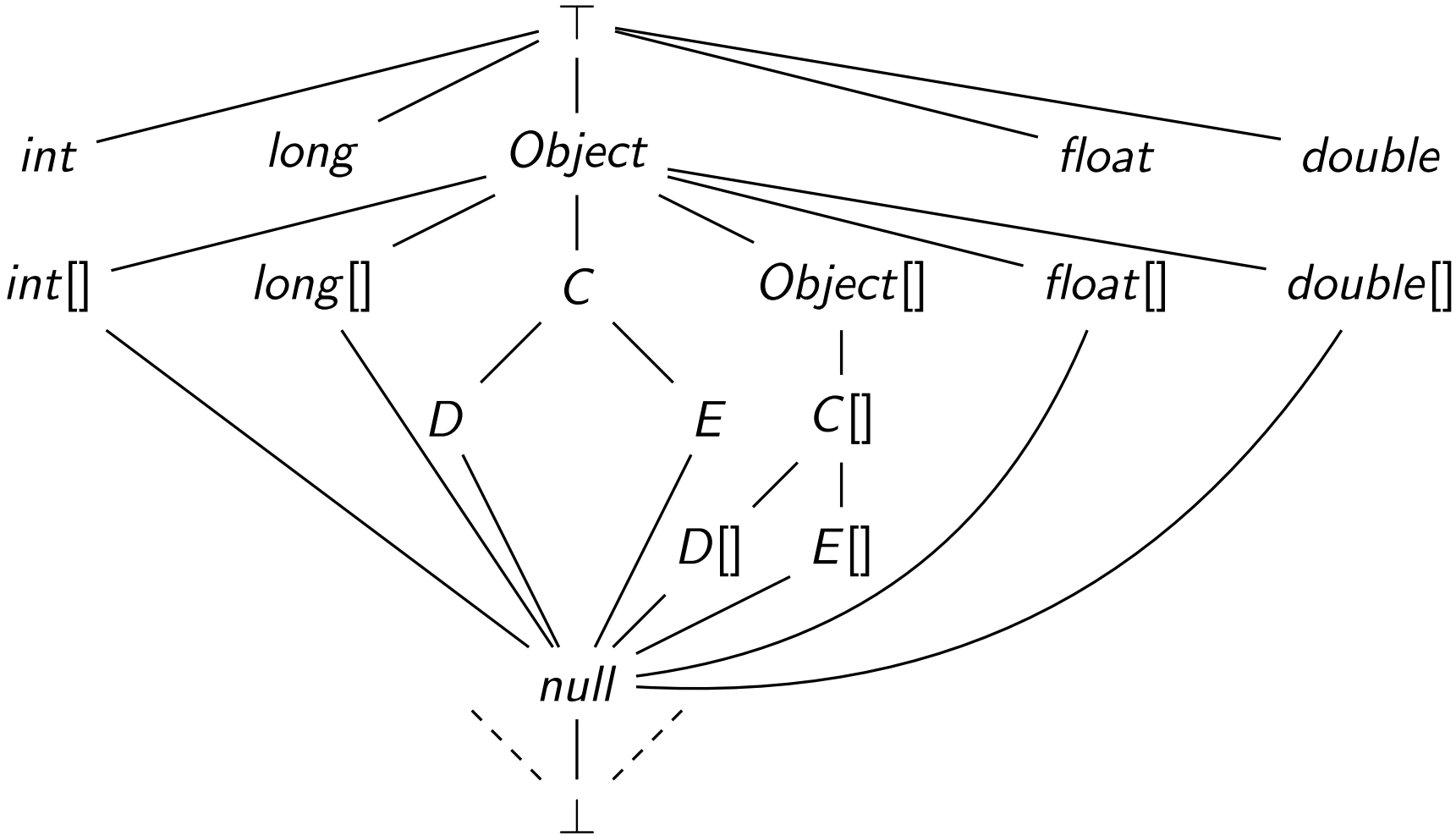
Types

The set of types, *Typ*, is composed of

- Primitive types:
  - *int* (covering *boolean*, *byte*, *char*, *short*)
  - *long*
  - *float*
  - *double*
- Object reference types: C for every class name C
- Array types: $\tau[]$ for every primitive or object reference type τ
- Method types: $\tau_0(\tau_1, \ldots, \tau_n)$ for $n \in \mathbb{N}$, $\tau_i \in \mathit{Typ}$
- Special types:
  - *null* (null reference)
  - *Object* (any object)
  - $\top$ (contents of uninitialized registers, i.e., any value)
  - $\bot$ (absence of any value)

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 $S \in Typ^{\leq m_s}$
  - register type $R : \{0, \ldots, m_r - 1\} \to Typ$

The Type-Level Abstract Interpreter I

- **Idea:** execute JVM instructions on types (rather than concrete values)
  - stack type $S \in Typ^{\leq m_s}$
  - 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

The Type-Level Abstract Interpreter I

- **Idea:** execute JVM instructions on types (rather than concrete values)
  - stack type $S \in Typ^{\leq m_s}$
  - 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

The Type-Level Abstract Interpreter II

Some transition rules:

$\texttt{iconst\_}z$: $(S, R) \to (int.S, R)$ if $|S| < m_s$

$\texttt{aconst\_null}$: $(S, R) \to (null.S, R)$ if $|S| < m_s$

$\texttt{iadd}$: $(int.int.S, R) \to (int.S, R)$

$\texttt{if\_icmpeq}\ l$: $(int.int.S, R) \to (S, R)$

$\texttt{iload}\ n$: $(S, R) \to (int.S, R)$ if $0 \leq n < m_r, R(n) = int, |S| < m_s$

$\texttt{aload}\ n$: $(S, R) \to (R(n).S, R)$ if $0 \leq n < m_r, R(n) \sqsubseteq_t Object, |S| < m_s$

$\texttt{istore}\ n$: $(int.S, R) \to (S, R[n \mapsto int])$ if $0 \leq n < m_r$

$\texttt{astore}\ n$: $(\tau.S, R) \to (S, R[n \mapsto \tau])$ if $0 \leq n < m_r, \tau \sqsubseteq_t Object$

$\texttt{getfield}\ C\ f\ \tau$: $(D.S, R) \to (\tau.S, R)$ if $D \sqsubseteq_t C$

$\texttt{putfield}\ C\ f\ \tau$: $(\tau'.D.S, R) \to (S, R)$ if $\tau' \sqsubseteq_t \tau, D \sqsubseteq_t C$

$\texttt{invoke}\ C\ M\ \sigma$: $(\tau'_n \ldots \tau'_1.\tau'.S, R) \to (\tau_0.S, R)$ if $\sigma = \tau_0(\tau_1, \ldots, \tau_n), \tau'_i \sqsubseteq_t \tau_i$ for $1 \leq i \leq n, \tau' \sqsubseteq_t C$

Some Theoretical Properties

Lemma 10.3

1. $(Typ, \sqsubseteq_t)$ *is a complete lattice satisfying ACC.*

Some Theoretical Properties

Lemma 10.3

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* $R_1 = R_2$*.*

Some Theoretical Properties

Lemma 10.3

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')*).*

Some Theoretical Properties

Lemma 10.3

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* □

Outline

The Dataflow System I

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

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

The Dataflow System I

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

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

The Dataflow System I

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

- Labels $L := \{\text{line numbers of Java bytecode}\}$
- Extremal label $E := \{1\}$ (forward problem)
- Flow relation F: for every $l \in L$,

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

The Dataflow System I

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

- Labels $L := \{\text{line numbers of Java bytecode}\}$
- Extremal label $E := \{1\}$ (forward problem)
- Flow relation F: for every $l \in L$,

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

- 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!), $\sigma_i \sqsubseteq_t \tau_i$ 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

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

with parameter types $\tau_1, \ldots, \tau_n$ of M

The Dataflow System II

- Extremal value

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

  with parameter types $\tau_1, \ldots, \tau_n$ of M
- Transfer functions $\{\varphi_l \mid l \in L\}$ are defined by

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

The Dataflow System II

- Extremal value

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

  with parameter types $\tau_1, \ldots, \tau_n$ of M
- Transfer functions $\{\varphi_l \mid l \in L\}$ are defined by

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

Monotonicity of transfer functions is ensured by the following lemma.

Lemma 10.4

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* □