Program analysis for software security

Thomas Jensen
with a lot of assistance from David Pichardie

IRISA

Outline

Aspects of software security

The security of an information system depends on a variety of properties

- secure communication with the outer world
- proper protection of the data stored and manipulated by the system
- the user
- the **software** implementing its functions.

The software can have a number of problems :

- buffer overflows,
- bad typing of data,
- lack of control of access to data and services,
- illicit flow of information.

Improving software security

The problem with bad software can be addressed in a number of ways

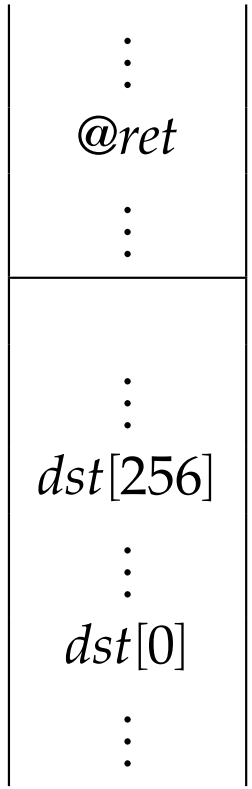
- better software engineering practice
- dynamic monitoring of executions
- static validation (test, analyse,. . .)

Example : Buffer overflow

The following code allows remote attackers to execute arbitrary code. . .

```
void foo(char * src){ char dst[256] = strcpy(dst,src);  ... }
```

Principle of the attack



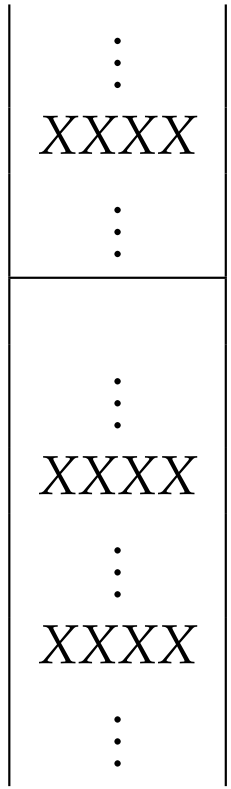
Example : Buffer overflow

The following code allows remote attackers to execute arbitrary code. . .

```
void foo(char * src){ char dst[256] = strcpy(dst,src);  ... }
```

Principle of the attack

Provide an argument that is bigger than the buffer



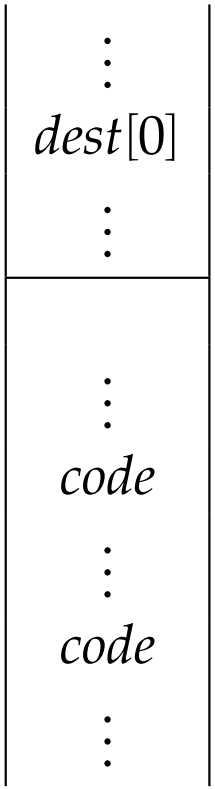
Example : Buffer overflow

The following code allows remote attackers to execute arbitrary code. . .

```
void foo(char * src){ char dst[256] = strcpy(dst,src);  ... }
```

Principle of the attack

Send code and modify the return address of method



Analysis of array indexing

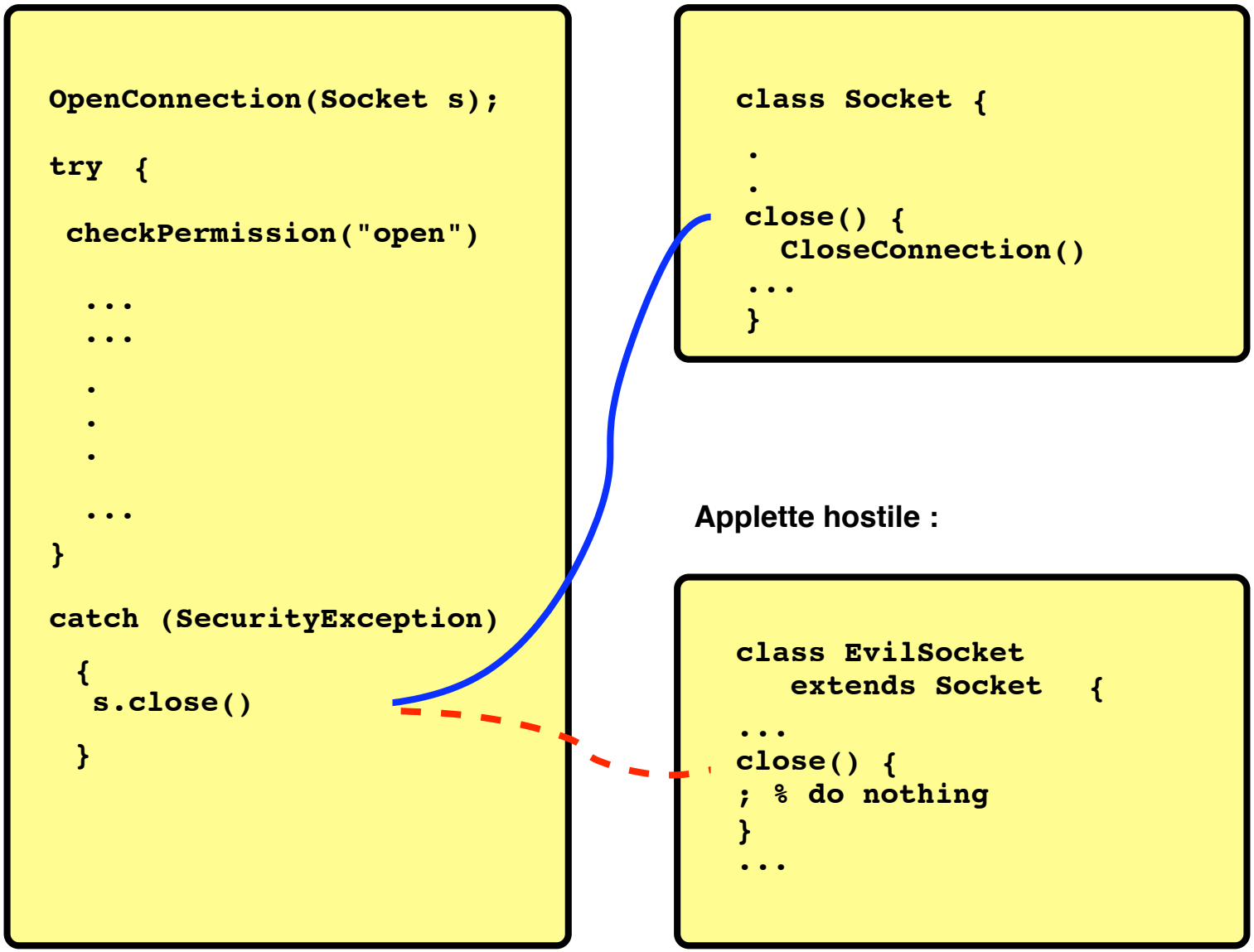
```
  //      PRE: True
static int bsearch(int key, int[] vec) {
  // (I'_1) |vec_0| = |vec| ∧ 0 ⩽ |vec_0|
   int low = 0, high = vec.length - 1;
  // (I'_2) |vec_0| = |vec| ∧ 0 ⩽ low ⩽ high + 1 ⩽ |vec_0|
   while (0 < high-low) {
  // (I'_3) |vec_0| = |vec| ∧ 0 ⩽ low < high < |vec_0|
       int mid = (low + high) / 2;
  // (I'_4) |vec| − |vec_0| = 0 ∧ low ⩾ 0 ∧ mid − low ⩾ 0 ∧
  //        2 · high − 2 · mid − 1 ⩾ 0 ∧ |vec_0| − high − 1 ⩾ 0
       if (key == vec[mid]) return mid;
       else if (key < vec[mid]) high = mid - 1;
       else low = mid + 1;
  // (I'_5) |vec_0| = |vec| ∧ −1 + low ⩽ high ∧ 0 ⩽ low ∧ 5 + 2 · high ⩽ 2 · |vec|
   }
  // (I'_6) 0 ⩽ |vec_0|
   return -1;
} //    POST: −1 ⩽ res < |vec_0|
```

Security-related software features

Static analysis can help with security-related problems like :

- buffer overflows and safe accesses to arrays,
- well-typing, proper data encapsulation and access control,
- information leakage and covert channels
- resource control and denial-of-service
- confidentiality and proper authentication
- exceptional executions (null references, division by zero,. . .)

Exceptional control flow



Analyse statique

Analyse statique

Analyse statique de programmes : obtenir de l'information sur le comportement d'un programme sans l'exécuter.
Une analyse statqiue

- doit terminer,
- et donc peut rendre une information approchée,
- mais ne doit jamais fournir une information fausse

$$17 \sim \textbf{odd} \qquad 17 \sim \mathbb{Z} \qquad 17 \not\sim \textbf{even}$$

Exemples

- Valeurs possibles d'une variable
  - entiers : parité, intervalles, rélations linéaires, . . .
  - programmes à objets : classes possibles d'objets stockés dans une variable.
- Flot de données :
  - quelles données atteignent quels points d'un programme ?
- Flot de contrôle :
  - quelles méthodes peuvent réellement être appelées par un appel virtuel ?
- Exceptions possibles
  - lancées et non-attrapées par un programme.

Exemple : définitions possibles

Déterminer l'ensemble des définitions (i.e affectations) qui peuvent atteindre un point de programme.

Exemple : la fonction factorielle.

```
1 : y := x;
2 : z := 1;
3 : while y > 1 do
4 :     z := z * y;
5 :     y := y - 1;
6 : y := 0
```

Les définitions $(\mathtt{y}, 1)$ et $(\mathtt{y}, 5)$ peuvent atteindre le point de programme 3. Pour chaque label l, calculer

$$\begin{array}{lcl} DP_e(l) & = & \text{les définitions qui arrivent à } l. \\ DP_s(l) & = & \text{les définitions qui sortent de } l. \end{array}$$

Définitions possibles : les équations (1)

Chaque instruction donne lieu à une équation.
Une affectation efface les définitions précédentes.

$$
\begin{aligned}
DP_s(1) &= DP_e(1) \setminus \{(\mathtt{y}, l) : l \in \mathbf{Lab}\} \cup \{(\mathtt{y}, 1)\}. \\
DP_s(2) &= DP_e(2) \setminus \{(\mathtt{z}, l) : l \in \mathbf{Lab}\} \cup \{(\mathtt{z}, 2)\}. \\
DP_s(3) &= DP_e(2) \\
DP_s(4) &= DP_e(4) \setminus \{(\mathtt{z}, l) : l \in \mathbf{Lab}\} \cup \{(\mathtt{z}, 4)\}. \\
DP_s(5) &= DP_e(5) \setminus \{(\mathtt{y}, l) : l \in \mathbf{Lab}\} \cup \{(\mathtt{y}, 5)\}. \\
DP_s(6) &= DP_e(6) \setminus \{(\mathtt{y}, l) : l \in \mathbf{Lab}\} \cup \{(\mathtt{y}, 6)\}.
\end{aligned}
$$

Définitions possibles : les équations (2)

Les définitions possible après une instruction sont également valable avant l'instruction suivante.
Le flux vers une instruction donne lieu à une équation :

$$
\begin{aligned}
DP_e(1) &= \{(\mathtt{x}, ?) : \mathtt{x} \text{ est une variable dans le programme}\} \\
DP_e(2) &= DP_s(1) \\
DP_e(3) &= DP_s(2) \cup DP_s(5) \\
DP_e(4) &= DP_s(3) \\
DP_e(5) &= DP_s(4) \\
DP_e(6) &= DP_s(3)
\end{aligned}
$$

Définitions possibles : la solution

$$
\begin{array}{lcl}
DP_e(1) & = & \{(\mathtt{x},?),(\mathtt{y},?),(\mathtt{z},?)\} \\
DP_e(2) & = & \{(\mathtt{x},?),(\mathtt{y},1),(\mathtt{z},?)\} \\
DP_e(3) & = & \{(\mathtt{x},?),(\mathtt{y},1),(\mathtt{y},5),(\mathtt{z},2),(\mathtt{z},4)\} \\
DP_e(4) & = & \{(\mathtt{x},?),(\mathtt{y},1),(\mathtt{y},5),(\mathtt{z},2),(\mathtt{z},4)\} \\
DP_e(5) & = & \{(\mathtt{x},?),(\mathtt{y},1),(\mathtt{y},5),(\mathtt{z},4)\} \\
DP_e(6) & = & \{(\mathtt{x},?),(\mathtt{y},1),(\mathtt{y},5),(\mathtt{z},2),(\mathtt{z},4)\} \\
 & & \\
DP_s(1) & = & \{(\mathtt{x},?),(\mathtt{y},1),(\mathtt{z},?)\} \\
DP_s(2) & = & \{(\mathtt{x},?),(\mathtt{y},1),(\mathtt{z},2)\} \\
DP_s(3) & = & \{(\mathtt{x},?),(\mathtt{y},1),(\mathtt{y},5),(\mathtt{z},2),(\mathtt{z},4)\} \\
DP_s(4) & = & \{(\mathtt{x},?),(\mathtt{y},1),(\mathtt{y},5),(\mathtt{z},4)\} \\
DP_s(5) & = & \{(\mathtt{x},?),(\mathtt{y},5),(\mathtt{z},4)\} \\
DP_s(6) & = & \{(\mathtt{x},?),(\mathtt{y},6),(\mathtt{z},2),(\mathtt{z},4)\}
\end{array}
$$

et on observe que $(\mathtt{y},1),(\mathtt{y},5) \in DP_e(3)$.

Définitions possibles : le cadre

Domaine abstrait : $\mathcal{P}(\textbf{Var} \times \textbf{Lab})$.
Pour chaque étiquette dans un programme Pg :

$$\begin{array}{lcl} kill(l : \texttt{x} := \texttt{a}) & = & \{(\texttt{x}, ?)\} \cup \{(\texttt{x}, l') : l' \in Lab\} \\ kill(l : \texttt{skip}) & = & \emptyset \\ kill(l : \texttt{b}) & = & \emptyset \end{array}$$

$$\begin{array}{lcl} gen(l : \texttt{x} := \texttt{a}) & = & \{(\texttt{x}, l)\} \\ gen(l : \texttt{skip}) & = & \emptyset \\ gen(l : \texttt{b}) & = & \emptyset \end{array}$$

+ des équations pour le flux entre instructions.

$$DP_e(l) = \begin{cases} \{(\texttt{x}, ?) : \texttt{x} \text{ est une variable}\} & \text{si } \textbf{init}(l) \\ \bigcup \{DP_s(l') : (l', l) \in \textbf{flow}(Pg)\} \end{cases}$$
$$DP_s(l) = (DP_e(l) \setminus kill(l : \texttt{instr})) \cup gen(l : \texttt{instr})$$

Résolution de systèmes d'équations

Une analyse peut être divisée en deux phases :

1. A partir d'un programme, produire un système d'équations.
2. Résoudre le système

Méthode de résolution générique : itération vers un point fixe.
Cadre général :

$$\begin{cases} x_1 &= f_1(x_1, \ldots, x_n) \\ &\vdots \\ x_n &= f_n(x_1, \ldots, x_n) \end{cases}$$

Exemple : $DP_e(3) = DP_s(2) \cup DP_s(5)$ avec $f = \cup$
Deux conditions :

1. le domaine abstrait est un *treillis*
2. $f_1, \ldots, f_n$ sont des fonctions *monotones*.

Ordres partiels et treillis

Définition Une relation $\sqsubseteq$ est **un ordre partiel** si elle est **réflexive**, **anti-symétrique** et **transitive**.

Définition La **borne supérieure** de deux éléments a et b d'un ordre partiel, notée $a \sqcup b$, est le plus petit élément tel que

$$a \sqsubseteq a \sqcup b \qquad b \sqsubseteq a \sqcup b.$$

De façon duale, la borne inférieure $a \sqcap b$ est le plus grad élément tel que

$$a \sqcap b \sqsubseteq a \qquad a \sqcap b \sqsubseteq b.$$

Définition Un **treillis** A est un ordre partiel tel que

1. il existe un plus petit élément, $\bot$ et un plus grand élément $\top$.
2. $a \sqcup b$ et $a \sqcap b$ existent pour toute paire d'éléments $a, b \in A$

Quelques treillis

- L'ensemble de parties d'un ensemble M, ordonné par inclusion

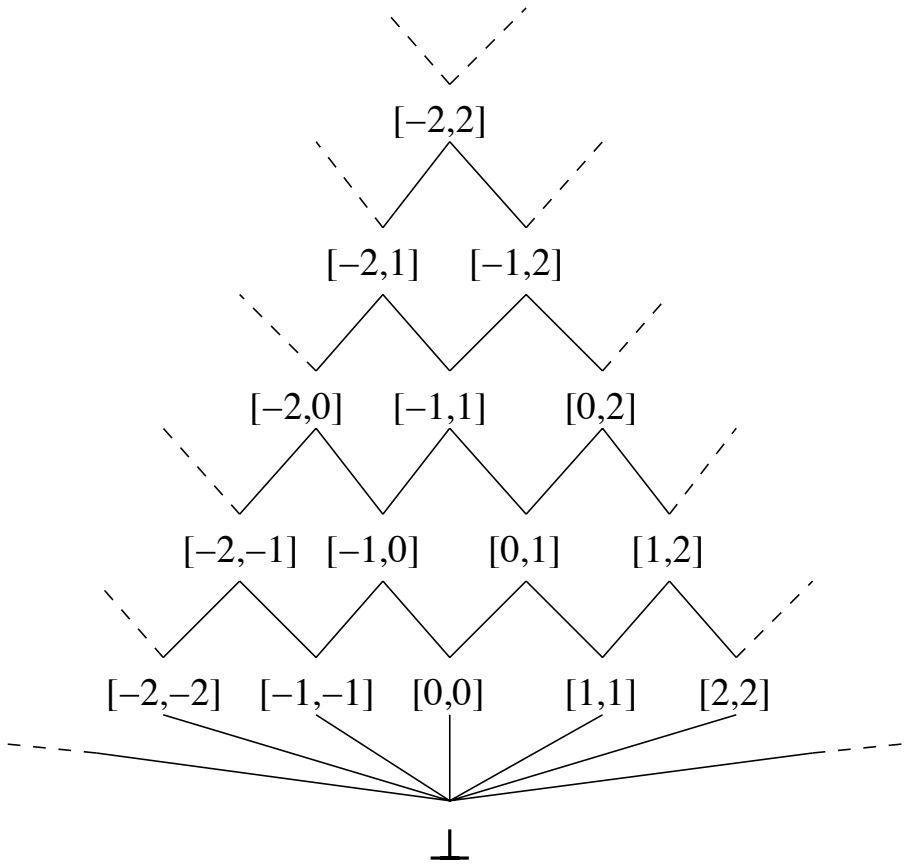
$$(P(M), \subseteq, \cup),$$

- $(P(M), \supseteq, \cap)$,
- produit de deux treillis :

$$(a_1, a_2) \sqsubseteq_\times (b_1, b_2) \quad \Leftrightarrow \quad a_1 \sqsubseteq_1 b_1 \ \wedge \ a_2 \sqsubseteq_2 b_2.$$

- intervalles $[n; m]$, ordonnés par $\subseteq$.

Le treillis d'intervalles



Calcul de point fixe itératif

Quand f est une fonction de n variables :

Initialisation		$i$ème itération	
$x_1^0 = \bot$	$\dots$	$f_1(x_1^i, \dots, x_n^i)$	$\dots$
$x_2^0 = \bot$	$\dots$	$f_2(x_1^i, \dots, x_n^i)$	$\dots$
$\vdots$			
$x_n^0 = \bot$	$\dots$	$f_n(x_1^i, \dots, x_n^i)$	$\dots$

The Java byte code verifier

Java

Java : a class-based, object-oriented programming language.

```
public class Bicycle{

    private int gear;

    private int id;

    private static int numberOfBicycles = 0;

    public Bicycle(int startCadence, int startSpeed, int startGear){
        gear = startGear;
        cadence = startCadence;
        id = ++numberOfBicycles;}

     public void setGear(int newValue){
        gear = newValue;}

public class MountainBike extends Bicycle {

    // the MountainBike subclass has one field
    public int seatHeight;
    ... }
```

Java byte code : factorial

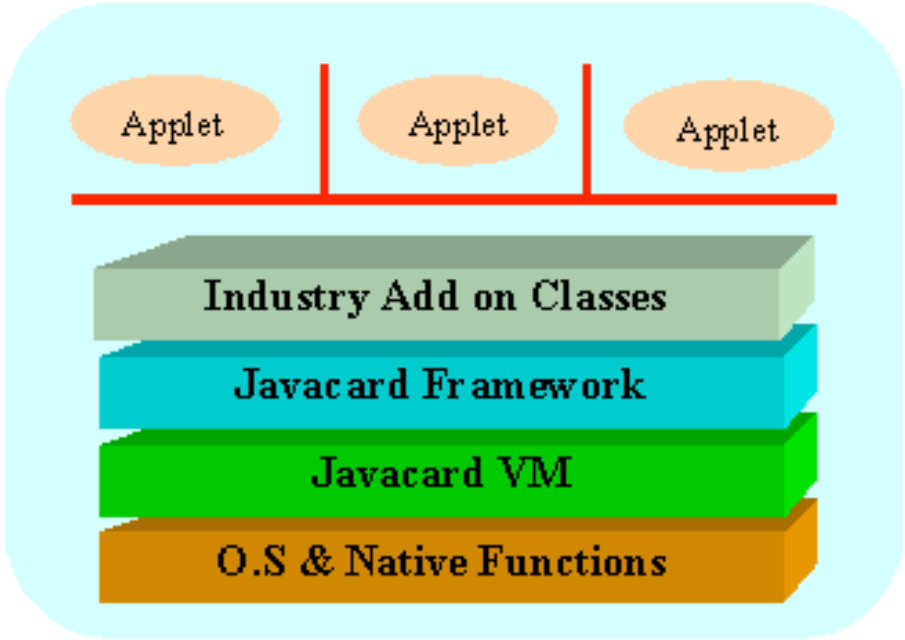
Source code

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

Byte code

```
method static int factorial(int), 2 registers, 2 stack slots
0: iconst 1 // push the integer constant 1
1: istore 1 // store it in register 1 (the res variable)
2: iload 0 // push register 0 (the n parameter)
3: ifle 14 // if negative or null, go to PC 14
6: iload 1 // push register 1 (res)
7: iload 0 // push register 0 (n)
8: imul // multiply the two integers at top of stack
9: istore 1 // pop result and store it in register 1
10: iinc 0, -1 // decrement register 0 (n) by 1
11: goto 2 // go to PC 2
14: iload 1 // load register 1 (res)
15: ireturn // return its value to caller
```

Java virtual machine



Java byte code verification

Java byte code verification is done at class loading time.

Check :

- structural correctness of class file
- no jumps out of methods
- operands get arguments of correct type
- objects and local variables are initialized before being used

Typing of byte code instructions

Typing as a data flow problem

Verification checks that arguments to operands are of correct type.

eg., adding a reference and an integer is not correct

Compute for each program point

- a type for each local variable and
- a type for the operand stack

This can be seen as a **data flow analysis** where data is replaced by types.

Operational semantics

We recall some rules of the operational semantics of Java byte code :

$$\frac{\text{instructionAt}_P(m, pc) = \texttt{push}\ c}{\langle\!\langle h, \langle m, pc, l, s\rangle :: sf\rangle\!\rangle \longrightarrow \langle\!\langle h, \langle m, pc+1, l, c :: s\rangle :: sf\rangle\!\rangle}$$

$$\frac{\text{instructionAt}_P(m, pc) = \texttt{pop}}{\langle\!\langle h, \langle m, pc, l, v :: s\rangle :: sf\rangle\!\rangle \longrightarrow \langle\!\langle h, \langle m, pc+1, l, s\rangle :: sf\rangle\!\rangle}$$

$$\frac{\text{instructionAt}_P(m, pc) = \texttt{numop}\ op}{\langle\!\langle h, \langle m, pc, l, n_1 :: n_2 :: s\rangle :: sf\rangle\!\rangle \longrightarrow \langle\!\langle h, \langle m, pc+1, l, [\![op]\!](n_1, n_2) :: s\rangle :: sf\rangle\!\rangle}$$

$$\frac{\text{instructionAt}_P(m, pc) = \texttt{load}\ x}{\langle\!\langle h, \langle m, pc, l, s\rangle :: sf\rangle\!\rangle \longrightarrow \langle\!\langle h, \langle m, pc+1, l, l[x] :: s\rangle :: sf\rangle\!\rangle}$$

$$\frac{\text{instructionAt}_P(m, pc) = \texttt{if}\ pc' \qquad n = 0}{\langle\!\langle h, \langle m, pc, l, n :: s\rangle :: sf\rangle\!\rangle \longrightarrow \langle\!\langle h, \langle m, pc', l, s\rangle :: sf\rangle\!\rangle}$$

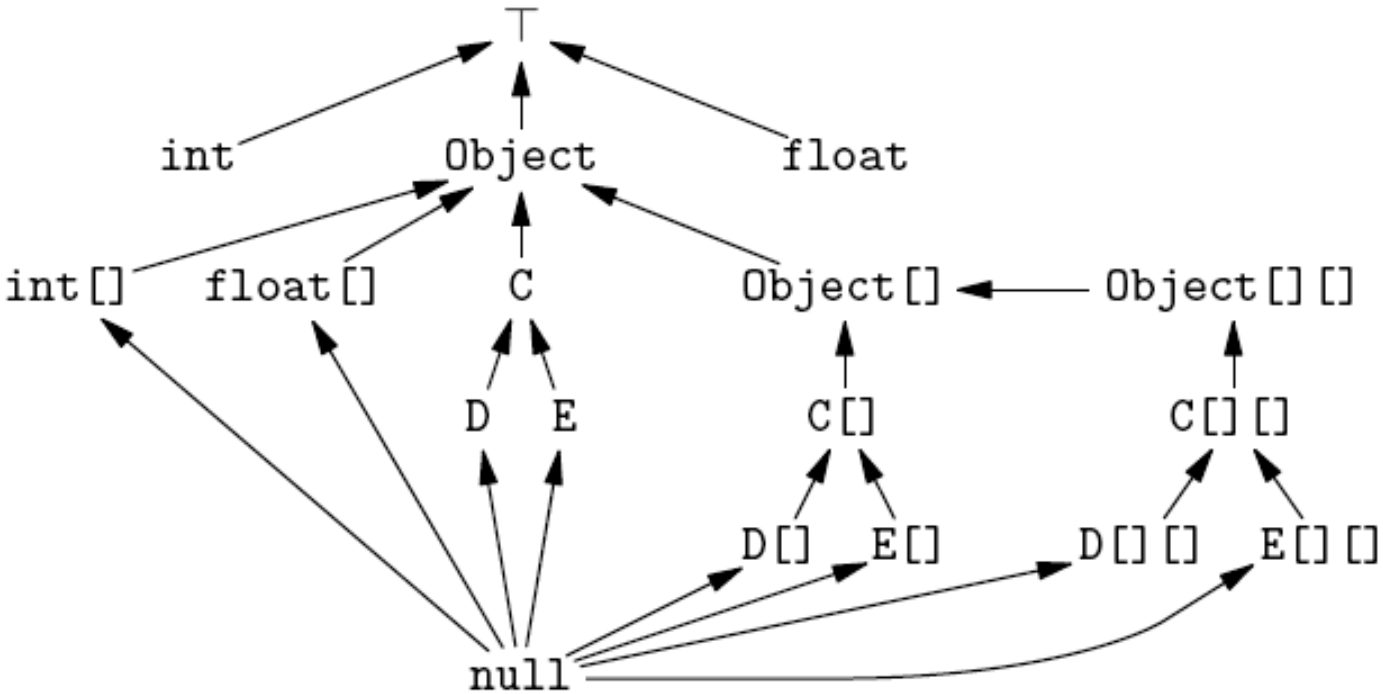
Typing as data flow analysis

```
iconst n : (S; R)  →  (int:S; R) if |S| < MaxStack
iadd : (int:int:S; R)  →  (int:S; R)
iload n : (S; R)  →  (int:S; R)
   if 0 < n < MaxReg and R(n) = int and |S|  < MaxStack
istore n : (int:S; R)  →  (S; R[n  →  int]) if 0 < n < MaxReg
aconst null : (S; R)  →  (null:S; R) if |S|  < MaxStack
aload n : (S; R)  →  (R(n):S; R)
   if 0 < n < MaxReg and R(n) <: Object and |S|  < MaxStack
astore n : (t:S; R)  →  (S; R[n  →  t])
   if 0 < n < MaxReg and t <: Object
getfield C:f:t : (t':S;R) →  (t:S; R) if t' <: C
putfield C:f:t : (t1:t2:S; R)  →  (S; R) if t1 <: t and t2 <: C
```

Here, $<:$ is the sub-class relation.

When control flow from more than one program point, we take the least upper bound.

The lattice of types



Example : typing factorial

```
method static int factorial(int), 2 registers, 2 stack slots
0: iconst 1 //
1: istore 1 //
2: iload 0 //
3: ifle 14 //
6: iload 1 //
7: iload 0 //
8: imul //
9: istore 1 //
10: iinc 0, -1 //
11: goto 2 //
14: iload 1 //
15: ireturn //
```

Example : typing factorial

```
method static int factorial(int), 2 registers, 2 stack slots
0: iconst 1 //  L : int,undef   S : []
1: istore 1 //
2: iload 0 //
3: ifle 14 //
6: iload 1 //
7: iload 0 //
8: imul //
9: istore 1 //
10: iinc 0, -1 //
11: goto 2 //
14: iload 1 //
15: ireturn //
```

Example : typing factorial

```
method static int factorial(int), 2 registers, 2 stack slots
0: iconst 1 //  L : int,undef   S : []
1: istore 1 //  L : int,undef   S : [int]
2: iload 0 //
3: ifle 14 //
6: iload 1 //
7: iload 0 //
8: imul //
9: istore 1 //
10: iinc 0, -1 //
11: goto 2 //
14: iload 1 //
15: ireturn //
```

Example : typing factorial

```
method static int factorial(int), 2 registers, 2 stack slots
0: iconst 1 //  L : int,undef   S : []
1: istore 1 //  L : int,undef   S : [int]
2: iload 0 //  L : int,int   S : []
3: ifle 14 //
6: iload 1 //
7: iload 0 //
8: imul //
9: istore 1 //
10: iinc 0, -1 //
11: goto 2 //
14: iload 1 //
15: ireturn //
```

Example : typing factorial

```
method static int factorial(int), 2 registers, 2 stack slots
0: iconst 1 //  L : int,undef   S : []
1: istore 1 //  L : int,undef   S : [int]
2: iload 0 //  L : int,int   S : []
3: ifle 14 //  L : int,int   S : [int]
6: iload 1 //
7: iload 0 //
8: imul //
9: istore 1 //
10: iinc 0, -1 //
11: goto 2 //
14: iload 1 //
15: ireturn //
```

Example : typing factorial

```
method static int factorial(int), 2 registers, 2 stack slots
0: iconst 1 //  L : int,undef   S : []
1: istore 1 //  L : int,undef   S : [int]
2: iload 0 //  L : int,int   S : []
3: ifle 14 //  L : int,int   S : [int]
6: iload 1 //  L : int,int   S : []
7: iload 0 //
8: imul //
9: istore 1 //
10: iinc 0, -1 //
11: goto 2 //
14: iload 1 //  L : int,int   S : []
15: ireturn //
```

Example : typing factorial

```
method static int factorial(int), 2 registers, 2 stack slots
0: iconst 1 //  L : int,undef   S : []
1: istore 1 //  L : int,undef   S : [int]
2: iload 0 //  L : int,int   S : []
3: ifle 14 //  L : int,int   S : [int]
6: iload 1 //  L : int,int   S : []
7: iload 0 //  L : int,int   S : [int]
8: imul //
9: istore 1 //
10: iinc 0, -1 //
11: goto 2 //
14: iload 1 //  L : int,int   S : []
15: ireturn //
```

Example : typing factorial

```
method static int factorial(int), 2 registers, 2 stack slots
0: iconst 1 //  L : int,undef   S : []
1: istore 1 //  L : int,undef   S : [int]
2: iload 0 //  L : int,int   S : []
3: ifle 14 //  L : int,int   S : [int]
6: iload 1 //  L : int,int   S : []
7: iload 0 //  L : int,int   S : [int]
8: imul //  L : int,int   S : [int : : int]
9: istore 1 //
10: iinc 0, -1 //
11: goto 2 //
14: iload 1 //  L : int,int   S : []
15: ireturn //
```

Example : typing factorial

```
method static int factorial(int), 2 registers, 2 stack slots
0: iconst 1 //  L : int,undef   S : []
1: istore 1 //  L : int,undef   S : [int]
2: iload 0 //  L : int,int   S : []
3: ifle 14 //  L : int,int   S : [int]
6: iload 1 //  L : int,int   S : []
7: iload 0 //  L : int,int   S : [int]
8: imul //  L : int,int   S : [int : : int]
9: istore 1 //  L : int,int   S : [int]
10: iinc 0, -1 //
11: goto 2 //
14: iload 1 //  L : int,int   S : []
15: ireturn //
```

Example : typing factorial

```
method static int factorial(int), 2 registers, 2 stack slots
0: iconst 1 //  L : int,undef   S : []
1: istore 1 //  L : int,undef   S : [int]
2: iload 0 //  L : int,int   S : []
3: ifle 14 //  L : int,int   S : [int]
6: iload 1 //  L : int,int   S : []
7: iload 0 //  L : int,int   S : [int]
8: imul //  L : int,int   S : [int : : int]
9: istore 1 //  L : int,int   S : [int]
10: iinc 0, -1 //  L : int,int   S : []
11: goto 2 //
14: iload 1 //  L : int,int   S : []
15: ireturn //
```

Example : typing factorial

```
method static int factorial(int), 2 registers, 2 stack slots
0: iconst 1 //  L : int,undef   S : []
1: istore 1 //  L : int,undef   S : [int]
2: iload 0 //  L : int,int   S : []
3: ifle 14 //  L : int,int   S : [int]
6: iload 1 //  L : int,int   S : []
7: iload 0 //  L : int,int   S : [int]
8: imul //  L : int,int   S : [int : : int]
9: istore 1 //  L : int,int   S : [int]
10: iinc 0, -1 //  L : int,int   S : []
11: goto 2 //  L : int,int   S : []
14: iload 1 //  L : int,int   S : []
15: ireturn //
```

Example : typing factorial

```
method static int factorial(int), 2 registers, 2 stack slots
0: iconst 1 //  L : int,undef   S : []
1: istore 1 //  L : int,undef   S : [int]
2: iload 0 //    L : int,int    S : []
3: ifle 14 //  L : int,int   S : [int]
6: iload 1 //  L : int,int   S : []
7: iload 0 //  L : int,int   S : [int]
8: imul //  L : int,int   S : [int : : int]
9: istore 1 //  L : int,int   S : [int]
10: iinc 0, -1 //  L : int,int   S : []
11: goto 2 //  L : int,int   S : []
14: iload 1 //  L : int,int   S : []
15: ireturn //
```

Example : typing factorial

```
method static int factorial(int), 2 registers, 2 stack slots
0: iconst 1 //  L : int,undef   S : []
1: istore 1 //  L : int,undef   S : [int]
2: iload 0 //  L : int,int   S : []
3: ifle 14 //  L : int,int   S : [int]
6: iload 1 //  L : int,int   S : []
7: iload 0 //  L : int,int   S : [int]
8: imul //  L : int,int   S : [int : : int]
9: istore 1 //  L : int,int   S : [int]
10: iinc 0, -1 //  L : int,int   S : []
11: goto 2 //  L : int,int   S : []
14: iload 1 //  L : int,int   S : []
15: ireturn //  L : int,int   S : [int]
```

Object creation and initialization

In Java byte code, objects of class `C` are created in two steps :

- they are *allocated* by `new`
- and *initialized* by a call to the constructor method `<init>` of the class `C`.

```
new C    // create uninitialized instance of class C
dup      // duplicate reference
....     // compute args to constructor
invokespecial C.<init>
```

The Java byte code language definition says :

> *It is an* **`error`** *to use an object (except assigning values to local fields) before all its constructors have been called.*

Similarly, it is an error to read from a local variable (a register) before it has been assigned a value.

Object initialization analysis

Mark objects types with "not yet completely initialized"

Change status of all objects of a given type when exiting the constructor of that method.

```
 0: new C      // stack type after: C̄0
 3: dup                           //    C̄0, C̄0
 4: new C                         //    C̄0, C̄0, C̄4
 7: dup                           //    C̄0, C̄0, C̄4, C̄4
 8: aconst_null                   //    C̄0, C̄0, C̄4, C̄4, null
 9: invokespecial C.<init>        //    C̄0, C̄0, C
12: invokespecial C.<init>        //    C
15: ...
```

Static and dynamic semantics

Static and dynamic semantics

We now present (part of) a formalization of the Java byte code verifier, due to Freund and Mitchell.

For a selection of instructions we will show

- operational (dynamic) semantics
- typing rules (static semantics).

The code environment

Java byte code verification verifies each class **separately**

The class hierarchy is described by an environment Γ :

$$\Gamma^C : \quad \textit{Class-Name} \rightharpoonup \left\langle \begin{array}{l} \texttt{super} : \textit{Class-Name} \cup \{\textit{None}\}, \\ \texttt{interfaces} : \text{set of } \textit{Interface-Name}, \\ \texttt{fields} : \text{set of } \textit{Field-Ref} \end{array} \right\rangle$$

$$\Gamma^M : \quad \textit{Method-Ref} \rightharpoonup \left\langle \begin{array}{l} \texttt{code} : \textit{Instruction}^+, \\ \texttt{handlers} : \textit{Handler}^* \end{array} \right\rangle$$

where a *Method-Ref* has form

$$\langle \textit{Class} - \textit{Name}, \textit{Label}, \textit{Method} - \textit{Type} \rangle.$$

Well-typing of methods

Verifying an environment Γ involves checking the well-formedness of the class hierarchy and type checking each method.

Judgment for methods :

$$\Gamma, F, S \vdash P, H : sig$$

Intuitively,

In environemnt Γ with typings F and S, the method with instruction table P (and exception handlers H) has signature sig

Well-typing of methods

A method is well-typed if

[METH CODE]

$$\frac{\begin{array}{rl} m \neq \texttt{<init>} & \\ \Gamma \vdash F_{\mathrm{TOP}}[0 \mapsto \sigma, 1..|\alpha| \mapsto \alpha] <: F_1 & \\ S_1 = \epsilon & \\ G_1 = \{\epsilon\} & \\ \forall i \in Dom(P).\ \Gamma, F, S, i \vdash P : \{\!|\sigma, m, \alpha \to \gamma|\!\}_{\mathrm{M}} & \textit{instructions well-typed} \\ \forall i \in Dom(H).\ \Gamma, F, S \vdash H[i] \text{ handles } P & \textit{handlers well-typed} \\ \forall i \in Dom(P).\ G, i \vdash P \text{ labeled} & \textit{labeling exists for instructions} \\ \forall i \in Dom(H).\ G, H[i] \vdash P \text{ labeled} & \textit{and handlers} \end{array}}{\Gamma, F, S \vdash P, H : \{\!|\sigma, m, \alpha \to \gamma|\!\}_{\mathrm{M}}}$$

Serves to define the predicate

$$\Gamma : wt$$

which, informally, states that every element of Γ is well-typed.

Stack operations

[PUSH]

$\Gamma, F, S, i \vdash P : \{\!|\sigma_m, m, \alpha_m \to \gamma_m|\!\}_{\mathtt{M}}$

$P[i] = \mathtt{push}\ v$
$v \in$ values of type τ $\tau \in Prim \cup \{\mathtt{Null}\}$
$\Gamma \vdash \tau \cdot S_i <: S_{i+1}$ $\Gamma \vdash F_i <: F_{i+1}$ $i+1 \in Dom(P)$

[POP]

$\Gamma, F, S, i \vdash P : \{\!|\sigma_m, m, \alpha_m \to \gamma_m|\!\}_{\mathtt{M}}$

$P[i] = \mathtt{pop}$
$S_i = \tau \cdot \beta$
$\Gamma \vdash \beta <: S_{i+1}$ $\Gamma \vdash F_i <: F_{i+1}$ $i+1 \in Dom(P)$

Assignment to local variables

[LOAD]

$$\Gamma, F, S, i \vdash P : \{\!|\sigma_m, m, \alpha_m \to \gamma_m|\!\}_{\mathtt{M}}$$

$P[i] = \texttt{load } x$
$x \in Dom(F_i)$
$\Gamma \vdash F_i[x] \cdot S_i <: S_{i+1}$ $\Gamma \vdash F_i <: F_{i+1}$ $i+1 \in Dom(P)$

[STORE]

$$\Gamma, F, S, i \vdash P : \{\!|\sigma_m, m, \alpha_m \to \gamma_m|\!\}_{\mathtt{M}}$$

$P[i] = \texttt{store } x$
$x \in Dom(F_i)$ $S_i = \tau \cdot \beta$
$\Gamma \vdash \beta <: S_{i+1}$ $\Gamma \vdash F_i[x \mapsto \tau] <: F_{i+1}$ $i+1 \in Dom(P)$

Object manipulations

[GET FIELD]

$\Gamma, F, S, i \vdash P : \{\!|\sigma_m, m, \alpha_m \to \gamma_m|\!\}_{\mathtt{M}}$

$P[i] = \mathtt{getfield}\ \{\!|\varphi, l, \kappa|\!\}_{\mathtt{F}}$
$\Gamma \vdash S_i <: \varphi \cdot \beta$ $\{\!|\varphi, l, \kappa|\!\}_{\mathtt{F}} \in \Gamma[\varphi].\mathtt{fields}$
$\Gamma \vdash \kappa \cdot \beta <: S_{i+1}$ $\Gamma \vdash F_i <: F_{i+1}$ $i + 1 \in Dom(P)$

[PUT FIELD]

$\Gamma, F, S, i \vdash P : \{\!|\sigma_m, m, \alpha_m \to \gamma_m|\!\}_{\mathtt{M}}$

$P[i] = \mathtt{putfield}\ \{\!|\varphi, l, \kappa|\!\}_{\mathtt{F}}$
$\Gamma \vdash S_i <: \kappa \cdot \varphi \cdot \beta$ $\{\!|\varphi, l, \kappa|\!\}_{\mathtt{F}} \in \Gamma[\varphi].\mathtt{fields}$
$\Gamma \vdash \beta <: S_{i+1}$ $\Gamma \vdash F_i <: F_{i+1}$ $i + 1 \in Dom(P)$

Control operations

[IF]

$\Gamma, F, S, i \vdash P : \{\!|\sigma_m, m, \alpha_m \to \gamma_m|\!\}_{\mathtt{M}}$

$P[i] = \texttt{ifeq}\ L$
$\tau \in \mathit{Simple\text{-}Ref} \cup \mathit{Prim} \cup \mathit{Array}$ $\Gamma \vdash S_i <: \tau \cdot \tau \cdot \beta$
$\Gamma \vdash \beta <: S_{i+1}$ $\Gamma \vdash F_i <: F_{i+1}$ $i + 1 \in \mathit{Dom}(P)$
$\Gamma \vdash \beta <: S_L$ $\Gamma \vdash F_i <: F_L$ $L \in \mathit{Dom}(P)$

Method call : operational semantics

Method calls

$$\frac{\begin{array}{c}\text{instructionAt}_P(m,pc) = \texttt{invokevirtual}\ M \\ h(loc) = o \qquad m' = \text{Lookup}(M, class(o)) \\ f' = \langle m', 1, V, \varepsilon\rangle \qquad f'' = \langle m, pc, l, s\rangle\end{array}}{\langle\!\langle h, \langle m, pc, l, loc :: V :: s\rangle :: sf\rangle\!\rangle \rightarrow \langle\!\langle h, f' :: f'' :: sf\rangle\!\rangle}$$

$$\frac{\text{instructionAt}_P(m,pc) = \texttt{return} \qquad f' = \langle m', pc', l', s'\rangle}{\langle\!\langle h, \langle m, pc, l, v :: s\rangle :: f' :: sf\rangle\!\rangle \rightarrow \langle\!\langle h, \langle m', pc' + 1, l', v :: s'\rangle :: sf\rangle\!\rangle}$$

Method call : typing rules

[INV VIRT]

$\Gamma, F, S, i \vdash P : \{\!|\sigma_m, m, \alpha_m \to \gamma_m|\!\}_{\mathtt{M}}$

$P[i] = \texttt{invokevirtual}\ N$
$N = \{\!|\varphi, m', \alpha \to \gamma|\!\}_{\mathtt{M}}$ $m' \neq \texttt{<init>}$ $\Gamma \vdash S_i <: \alpha \bullet (\varphi \cdot \beta)$ $N \in Dom(\Gamma)$
$\gamma \neq \texttt{void} \Rightarrow \Gamma \vdash \gamma \cdot \beta <: S_{i+1}$ $\gamma = \texttt{void} \Rightarrow \Gamma \vdash \beta <: S_{i+1}$ $\Gamma \vdash F_i <: F_{i+1}$ $i + 1 \in Dom(P)$

[RETURN]

$\Gamma, F, S, i \vdash P : \{\!|\sigma_m, m, \alpha_m \to \gamma_m|\!\}_{\mathtt{M}}$

$P[i] = \texttt{return}$
$\gamma_m = \texttt{void}$

What about `return` x ?

Correctness of typing rules

The correctness is expressed in the following theorem :

Intuitively

> *if the current state is well-typed and execution can proceed, then the resulting state is well-typed*

Formally, for a Γ satisfying $\Gamma : wt$:

$$
\begin{aligned}
&\forall A, A', h, h'. \\
&\quad \Gamma \vdash h \ \texttt{wt} \\
&\quad\ \wedge\ \mathit{GoodStack}(\Gamma, A, h) \\
&\quad\ \wedge\ \Gamma \vdash A; h \rightarrow A'; h' \\
&\quad \Rightarrow \Gamma \vdash h' \ \texttt{wt} \\
&\quad\ \wedge\ \mathit{GoodStack}(\Gamma, A', h')
\end{aligned}
$$

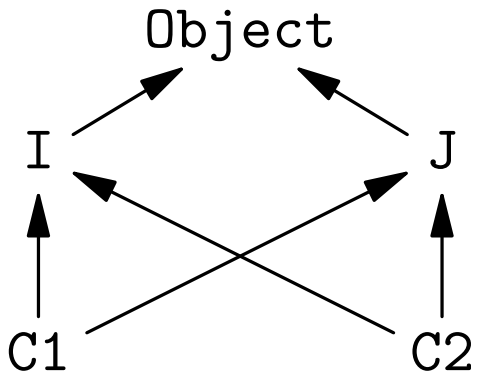
Interfaces and sub-routines

The problem with interfaces

Interfaces can be used as types of variables, stack locations *etc*

Classes can implement **several** interfaces

$\Rightarrow$ certain least upper bounds do not exist.



Sun's solution : treat interfaces as `Objects` and check at run-time whether an object implements an interfaces.

Verification of sub-routines

`jsr l` : jump to program point `l`, pushing the address of the follwing instruction

`ret n` : recover a return address from register `n` and jump to it.

Problems :
- sub-routine entries are *merge points*
- may limit precision

This complicates the byte code verification —
for a relatively **little gain**.

```
0: jsr 100 // register 0 undef
3: ...
50: iconst_0
51: istore_0
52: jsr 100  // register 0 int
55: iload_0
56: ireturn
...
100: astore_1
101: ...  // don't touch register 0
110: ret 1
```

Summary

Java byte code verification

- checks `.class` files when loaded into a Java virtual machine
- part of the secuuity architecture of Java
- verifies
  - well typing
  - object initialization
- formalized as a set of constraint rules

We have treated a subset of Java BC here

- basic sequential, procedural Java byte code
- interfaces and subroutines
- we did not deal with exceptions and arrays

Literature

There exists a vast literature on Java byte code verification.

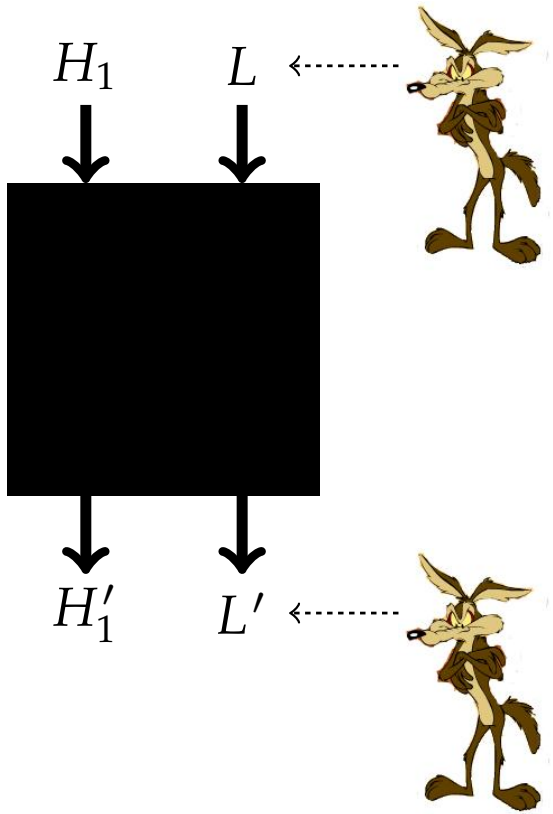
The present course is based on two classics

- Freund and Mitchell
- X. Leroy

Information flow type system

Non-interference

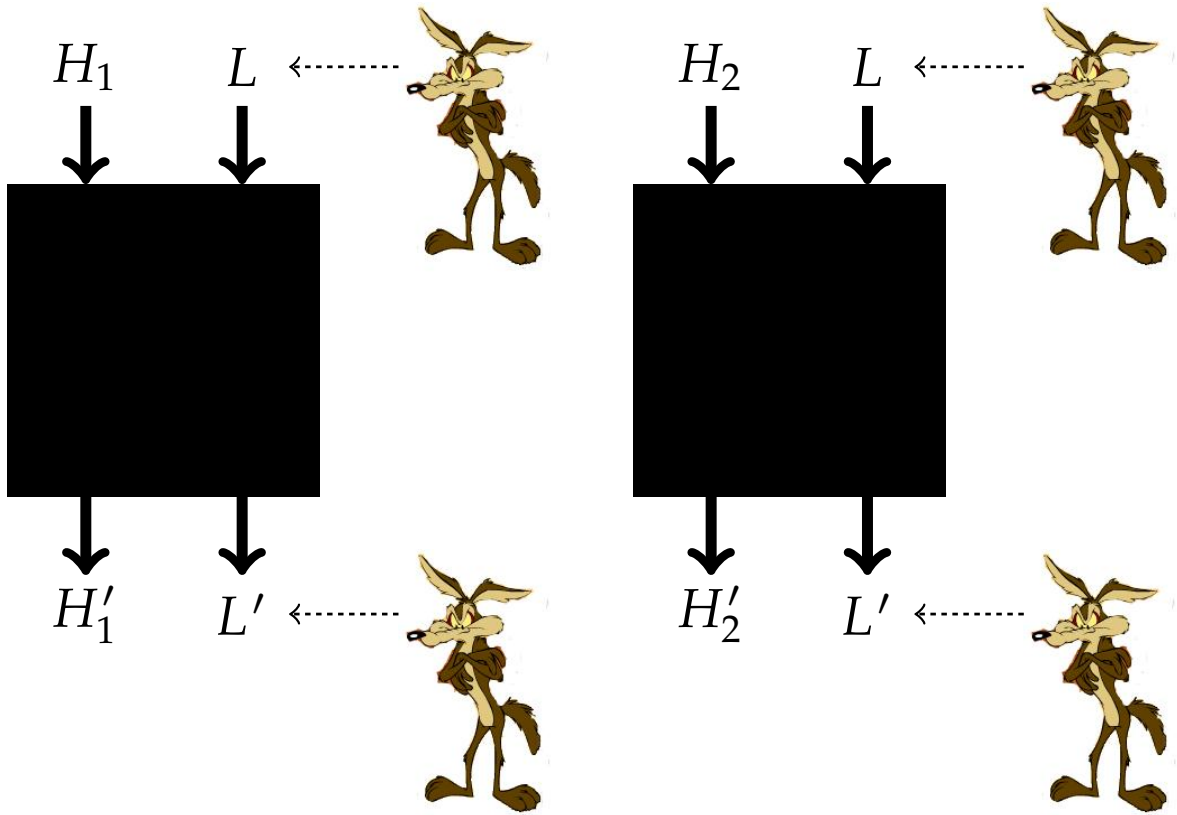
"Low-security behavior of the program is not affected by any high-security data." Goguen & Meseguer 1982



High = confidential Low = public

Non-interference

"Low-security behavior of the program is not affected by any high-security data." Goguen & Meseguer 1982



High = confidential Low = public

Motivation

We will present a simple information flow type system[1] and prove it enforces the semantic non-interference property on well typed programs.

[1] D. Volpano and G. Smith, *A Type-Based Approach to Program Security*, Theory and Practice of Software Development, 1997.

Program syntax

We consider a standard WHILE language but we mix arithmetic and boolean expressions.

$$
\begin{array}{rll}
\textbf{Expr} ::= & n & n \in \mathbb{Z} \\
\mid & x & x \in \mathbb{V}_H \uplus \mathbb{V}_L \\
\mid & \textbf{Expr}\ o\ \textbf{Expr} & o \in \{+, -, \times, \ldots\} \\
\mid & \textbf{Expr}\ c\ \textbf{Expr} & c \in \{=, \neq, <, \leqslant, \ldots\} \\
\mid & \textbf{Expr}\ b\ \textbf{Expr} & b \in \{\text{and}, \text{or}\} \\
\textbf{Stm} ::= & x\ :=\ \textbf{Expr} & \\
\mid & \texttt{if}\ \textbf{Expr}\ \texttt{then}\ \textbf{Stm}\ \texttt{else}\ \textbf{Stm} & \\
\mid & \texttt{while}\ \textbf{Expr}\ \texttt{do}\ \textbf{Stm} & \\
\mid & \textbf{Stm}\ ;\ \textbf{Stm} &
\end{array}
$$

The set of variable is partitioned in two disjoint sets :

- $\mathbb{V}_H$: high (or secret) variables
- $\mathbb{V}_L$: low (or public) variables

Secure programs

Intuitively[2], a program is *secure* (or *non interferent*) if the final values of low variables do not depend on the initial values of the high variables.

Examples : are this programs secure or not ?

1. `h := l`
2. `l := h`
3. `if (h1>h2) then {l := 1} else {l := 2}`
4. `while (h) do { l := l+1 }; l := 0`

[2] This notion will be defined formally when presenting the semantic of the language.

A lattice of security levels

We consider here only two kind of informations (low and high), but the information flow policy can be defined as a lattice of *security levels*.

We note $\sqsubseteq$ the corresponding partial order.

We say there is a flow of information from x to y if the value of the variable x depends on the value of the variable y.

If x (resp. y) is of level k_x (resp. k_y), the flow is

- licit if $k_x \sqsubseteq k_y$
- illicit if $k_x \not\sqsubseteq k_y$

A simple information flow type system (1/2)

Non-interference can be enforced by an *information flow type system*

Typing judgment for expressions : $e \in \mathbf{Expr}, \tau \in \{L, H\}$

$$\vdash e : \tau$$

Meaning : the expression e depends only on variable of level τ or lower.

Typing rules :

$$\text{CONST}\frac{}{\vdash n : L} \qquad \text{VAR}\frac{x \in \mathbb{V}_\tau}{\vdash x : \tau} \qquad \text{BINOP}\frac{\vdash e_1 : \tau \qquad \vdash e_2 : \tau}{\vdash e_1 \; o \; e_2 : \tau}$$

$$\text{EXP-SUBTYP}\frac{\vdash e : \tau_1 \qquad \tau_1 \sqsubseteq \tau_2}{\vdash e : \tau_2}$$

Exemple

A type derivation for $\vdash h + 1 : H$

$$\text{BINOP}\ \frac{\text{VAR}\ \dfrac{h \in \mathbb{V}_H}{\vdash h : H} \qquad \text{EXP-SUBTYP}\ \dfrac{\text{CONST}\ \dfrac{}{\vdash 1 : L} \qquad L \sqsubseteq H}{\vdash 1 : H}}{\vdash h + 1 : H}$$

Exercice : give another type derivation for $\vdash h + 1 : H$

A simple information flow type system (2/2)

Typing judgment for statements : $S \in \mathbf{Stm}, \tau \in \{L, H\}$

$$\vdash S : \tau$$

Meaning : the only variables modified by statement S are of level τ or higher.

Typing rules :

$$\text{ASSIGN}\ \frac{x \in \mathbb{V}_\tau \qquad \vdash e : \tau}{\vdash x := e : \tau} \qquad \text{SEQ}\ \frac{\vdash S_1 : \tau \qquad \vdash S_2 : \tau}{\vdash S_1 \ ; \ S_2 : \tau}$$

$$\text{IF}\ \frac{\vdash e : \tau \qquad \vdash S_1 : \tau \qquad \vdash S_2 : \tau}{\vdash \texttt{if}\ e\ \texttt{then}\ S_1\ \texttt{else}\ S_2 : \tau} \qquad \text{WHILE}\ \frac{\vdash e : \tau \qquad \vdash S : \tau}{\vdash \texttt{while}\ e\ \texttt{do}\ S : \tau}$$

$$\text{STM-SUBTYP}\ \frac{\vdash S : \tau_2 \qquad \tau_1 \sqsubseteq \tau_2}{\vdash S : \tau_1}$$

The subtype relation on statements is *contravariant* !

Exercise

Try to type the following statements (give a derivation type, if possible) :

$$\texttt{if}\ (l)\ \texttt{then}\ h := l\ \texttt{else}\ l := 0$$

$$\texttt{if}\ (h)\ \texttt{then}\ h := l\ \texttt{else}\ l := 0$$

A natural semantics

$$\begin{aligned} \textbf{State} &= \textbf{Var} \to \mathbb{Z} \\ [\![\cdot]\!] &\in \textbf{Expr} \to \textbf{State} \to \mathbb{Z} \text{ (semantics of expression)} \\ (\cdot,\cdot) \Downarrow \cdot &\subseteq (\textbf{Stm} \times \textbf{State}) \times \textbf{State} \text{ (semantics of statement)} \end{aligned}$$

$$\begin{aligned} [\![n]\!]s &= \mathcal{N}[\![n]\!] \\ [\![x]\!]s &= s(x) \\ [\![e_1 + e_2]\!]s &= [\![e_1]\!]s + [\![e_2]\!]s \\ &\dots \end{aligned}$$

$$(x := e, s) \Downarrow s[x \mapsto [\![e]\!]s] \qquad \frac{(S_1, s) \Downarrow s' \quad (S_2, s') \Downarrow s''}{(S_1; S_2, s) \Downarrow s''}$$

$$\frac{(S_1, s) \Downarrow s' \quad [\![e]\!]s = 1}{(\texttt{if}\ e\ \texttt{then}\ S_1\ \texttt{else}\ S_2, s) \Downarrow s'} \qquad \frac{(S_2, s) \Downarrow s' \quad [\![e]\!]s = 0}{(\texttt{if}\ e\ \texttt{then}\ S_1\ \texttt{else}\ S_2, s) \Downarrow s'}$$

$$\frac{(S, s) \Downarrow s' \quad (\texttt{while}\ e\ \texttt{do}\ S, s') \Downarrow s'' \quad [\![e]\!]s = 1}{(\texttt{while}\ e\ \texttt{do}\ S, s) \Downarrow s''} \qquad \frac{[\![e]\!]s = 0}{(\texttt{while}\ e\ \texttt{do}\ S, s) \Downarrow s}$$

The observational power of an attacker

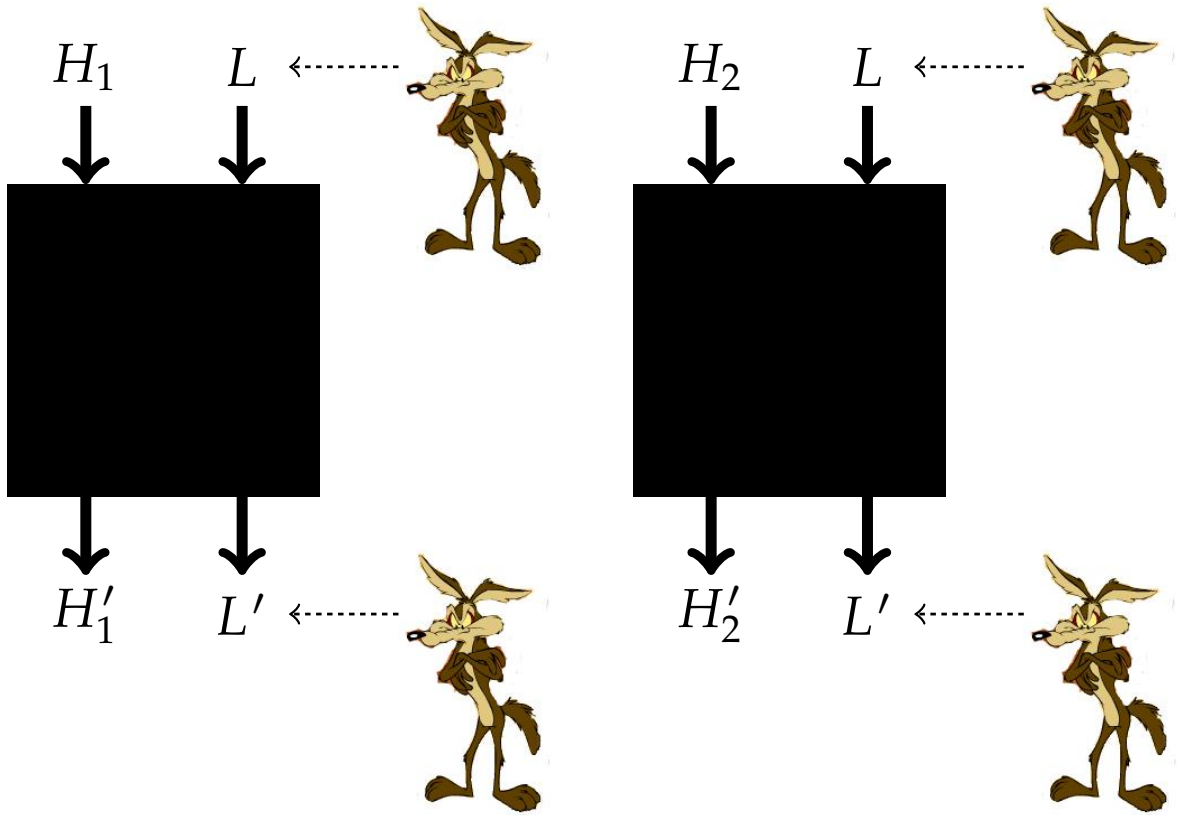
The attacker only see low variable before and after executions.

We model his observational power with an *indistinguishability* relation $\sim \subseteq \textbf{State} \times \textbf{State}$ between states.

$$s_1 \sim s_2 \text{ iff } \forall x \in \mathbb{V}_L,\ s_1(x) = s_2(x)$$

Non-interference

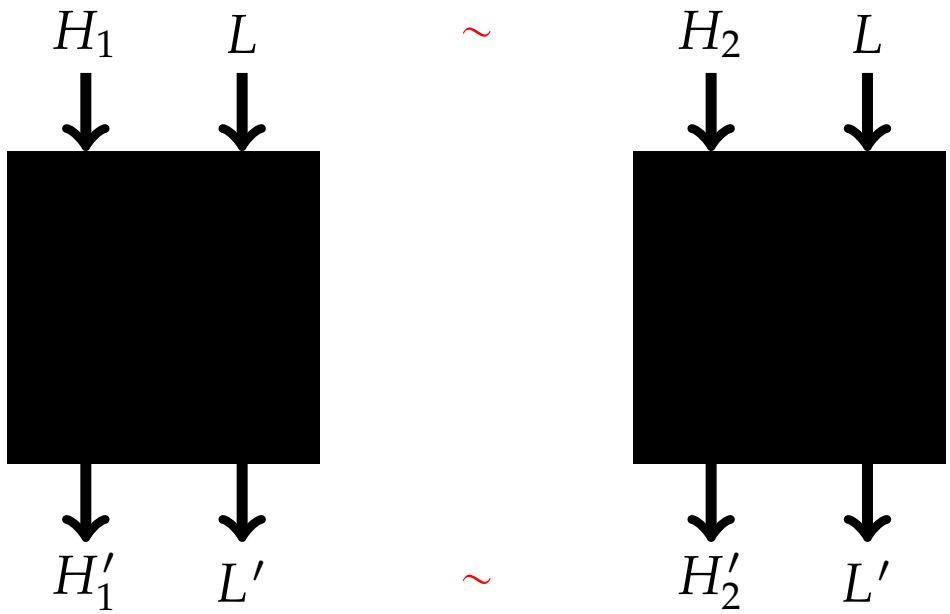
"Low-security behavior of the program is not affected by any high-security data." Goguen & Meseguer 1982



High = confidential Low = public

Non-interference

"Low-security behavior of the program is not affected by any high-security data." Goguen & Meseguer 1982



High = confidential Low = public

Type soundness

A statement S is said *non interferent* iff $\left.\begin{array}{r} s_1 \sim s_2 \\ (S, s_1) \Downarrow s_1' \\ (S, s_2) \Downarrow s_2' \end{array}\right\}$ implies $s_1' \sim s_2'$

Theorem

Every typable statement (i.e. such that $\exists \tau, \vdash S : \tau$) is non interferent.

Type soundness proof : step 1

We need a new set of typing rules

$$\text{CONST}' \frac{}{\vdash_s n : \tau} \qquad \text{VAR}' \frac{x \in \mathbb{V}_\tau \qquad \tau \sqsubseteq \tau'}{\vdash_s x : \tau'} \qquad \text{BINOP} \frac{\vdash_s e_1 : \tau \qquad \vdash_s e_2 : \tau}{\vdash_s e_1 \; o \; e_2 : \tau}$$

$$\text{ASSIGN}' \frac{x \in \mathbb{V}_\tau \qquad \vdash_s e : \tau \qquad \tau' \sqsubseteq \tau}{\vdash_s x := e : \tau'} \qquad \text{SEQ} \frac{\vdash_s S_1 : \tau \qquad \vdash_s S_2 : \tau}{\vdash_s S_1 \; ; \; S_2 : \tau}$$

$$\text{IF}' \frac{\vdash_s e : \tau \qquad \vdash_s S_1 : \tau \qquad \vdash_s S_2 : \tau \qquad \tau' \sqsubseteq \tau}{\vdash_s \texttt{if}\; e \;\texttt{then}\; S_1 \;\texttt{else}\; S_2 : \tau'}$$

$$\text{WHILE}' \frac{\vdash_s e : \tau \qquad \vdash_s S : \tau \qquad \tau' \sqsubseteq \tau}{\vdash_s \texttt{while}\; e \;\texttt{do}\; S : \tau'}$$

This type system is *syntax-directed* : at most one rule can be used for each statement (or expression).

Type soundness proof : step 1

Lemma (Sub-typing property)

For all $e \in \mathbf{Expr}, \tau, \tau' \in \{L, H\}$, $\vdash_s e : \tau$ *and* $\tau \sqsubseteq \tau'$ *implies* $\vdash_s e : \tau'$.
For all $S \in \mathbf{Stm}, \tau \in \{L, H\}$, $\vdash_s S : \tau'$ *and* $\tau \sqsubseteq \tau'$ *implies* $\vdash_s S : \tau$.

Proof. By induction on the typing judgment.

The new system is equivalent to the previous one.

Lemma

For all $e \in \mathbf{Expr}, \tau \in \{L, H\}$, $\vdash e : \tau$ *implies* $\vdash_s e : \tau$.
For all $S \in \mathbf{Stm}, \tau \in \{L, H\}$, $\vdash S : \tau$ *implies* $\vdash_s S : \tau$.

Proof. By induction on the typing judgment.

Lemma

For all $e \in \mathbf{Expr}, \tau \in \{L, H\}$, $\vdash_s e : \tau$ *implies* $\vdash e : \tau$.
For all $S \in \mathbf{Stm}, \tau \in \{L, H\}$, $\vdash_s S : \tau$ *implies* $\vdash S : \tau$.

Type soundness proof : step 2

Lemma (Low expressions)

For all $e \in$ **Expr**, $s_1, s_2 \in$ **State**, *if* $s_1 \sim s_2$ *and* $\vdash_s e : L$ *then,* $[\![e]\!]s_1 = [\![e]\!]s_2$.

Proof. By induction on e.

Lemma (Side-effect of high statements)

For all $S \in$ **Stm**, $s, s' \in$ **State**, *if* $(S, s) \Downarrow s'$ *and* $\vdash_s S : H$ *then* $s \sim s'$.

Proof. By induction on the judgment $(S, s) \Downarrow s'$.

Type soundness proof : final step

Theorem (Type soundness)

For all $S \in \mathbf{Stm}$, $s_1, s_2, s_1', s_2' \in \mathbf{State}$, $\tau \in \{L, H\}$, *if* $s_1 \sim s_2$, $(S, s_1) \Downarrow s_1'$, $(S, s_2) \Downarrow s_2'$ *and* $\vdash S : \tau$ *then* $s_1' \sim s_2'$.

Proof. By induction on the judgment $(S, s_1) \Downarrow s_1'$ and case analysis on $(S, s_2) \Downarrow s_2'$.

Conclusions

- Type checking is computable but non-interference is not
  Exercice : give an example of non-interferent program that is not typable.
- We have ignored some information channels :
  - timing channels

    ```
    if h>0 then l:=0 else { h:=h+1; l:=0 }
    ```

    measuring the run-time of this program may reveal secret informations.
  - termination channels

    ```
    while h>0 do skip
    ```

  - electro-magnetic activity (DPA attacks)
- Non-inteference is sometimes too strong a property.
  Example : a password checker always reveal some secret.
  Need for a theory of safe declassification