A Type System for the JVM
Motivation: Security
Scenarios

- Web browser downloads and executes JVM applet. In your own address space.
- SmartCard applet (one of many!) is executed in shared address space.
The problem

Should this bytecode be allowed:

\[\ldots, \text{Push (Addr 217), Push v, Putfield F D, \ldots}\]

No! Must not allow access to arbitrary addresses. Why: no hardware protection, software checks expensive.

\[\implies \text{no Addresses in bytecode!}\]

How about

\[\ldots, \text{Push (Intg 217), Push v, Putfield F D, \ldots}\]

No! Semantics underdefined, anything may happen, avoid!
The solution

Aim: Security without runtime checks (eg no type tags).

Method: Check statically against all kinds of attacks: address spoofing, stack over/underflow, . . .

JVM needs proper type system
to detect malicious or erroneous bytecode
Overview

1. Define type system (declaratively)

2. Prove type safety:
   When executing well-typed programs, check-instr is always true when exec-instr is called.

3. Implement type system as a static analyzer ("bytecode verifier", later)
Jinja versus JVM

- Jinja: local variables + result
  JVM: local variables + stack

- Jinja: type of local variables is declared and fixed
  JVM: type of local variables is not declared and may change from instruction to instruction; same for stack.

The JVM type system describes for each program point the types in the local variables and on the stack.
### Example

<table>
<thead>
<tr>
<th>Code</th>
<th>Stack</th>
<th>Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Push</strong> <em>(Intg 1)</em></td>
<td>[]</td>
<td><em>(Class C, Integer)</em></td>
</tr>
<tr>
<td>Load 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IAdd</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Store 0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Motto

Types are abstract descriptions of values/states

Instructions (operating on values) can be abstracted to operate on types

“Abstract interpretation”
We treat only the exception-free fragment of the JVM
No system exceptions, no user exception
Type System
At each program point, the Jinja type of each stack element must be uniquely defined.
Register types

datatype 'a err = Err | OK 'a

• \( ty_I = ty \) err list \((LT :: ty_I)\)

For a given program point and a given register:

**\( Err \)** : the Jinja type of the register is not defined uniquely. Not an error, but register cannot be read (is “unusable”).

**\( OK T \)** : the Jinja type of the register is definitely \( T \).
# Example: OK vs Err

<table>
<thead>
<tr>
<th>Code</th>
<th>Stack</th>
<th>Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>IfFalse 4</td>
<td>[Boolean]</td>
<td>[OK (Class C)]</td>
</tr>
<tr>
<td>Push (Intg 1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Store 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Goto 3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Push (Bool True)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Store 0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Reasons for Err

- Uninitialized local variables
- Storage optimization:
  two distinct local variables share the same register

Example: \( \text{if } (b) \{ V_1: T_1; \_ \} \text{ else } \{ V_2: T_2; \_ \} \)
Instruction/state type, method type

- $ty_i = ty_s \times ty_l \quad (\tau :: ty_i)$
- Instructions can be unreachable:
  $ty_i' = ty_i \text{ option} \quad (\tau :: ty_i')$

  *None*: instruction is unreachable
  *[\tau]*: state before/after instruction has type $\tau$.

- Method type: $ty_i' \text{ list} \quad (\tau s :: ty_i' \text{ list})$
### An example of the full truth

<table>
<thead>
<tr>
<th>Code</th>
<th>Stack</th>
<th>Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Goto 2</td>
<td>$[[\text{Boolean}], \ [\text{Err}]]$</td>
<td></td>
</tr>
<tr>
<td>Pop</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Push $\text{(Bool True)}$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Store 0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Well-typed Method
We consider a fixed method in a fixed JVM program $P$:

$$C_M :: \text{cname}$$ the class the method is defined in

$$Ts :: \text{ty list}$$ parameter types

$$T_r :: \text{ty}$$ return type

$$mxs :: \text{nat}$$ maximum stack size

$$mxl_0 :: \text{nat}$$ number of local variables (w/o parameters)

$$mxl :: \text{nat}$$ number of registers ($$mxl = 1 + |Ts| + mxl_0$$)

$$is :: \text{instr list}$$ instructions
The method is well-typed iff there is a method type $\tau s$ such that

- $|\tau s| = |is|$
- Each instruction is *applicable* to the corresponding instruction type in $\tau s$.  
  *Eg:* no $\text{Pop}$ when $ST = []$.
- The state after the execution of the instruction is “compatible” with all successor positions.  
  *Eg:* if $ST = [T]$ before $\text{Pop}$ then $ST = []$ afterwards.
**Well-typed method (almost formal)**

<table>
<thead>
<tr>
<th>_</th>
<th>_</th>
</tr>
</thead>
<tbody>
<tr>
<td>$pc$: $I$</td>
<td>$\tau$</td>
</tr>
<tr>
<td>_</td>
<td></td>
</tr>
<tr>
<td>$pc'$: $I'$</td>
<td>$\tau'$</td>
</tr>
<tr>
<td>_</td>
<td></td>
</tr>
</tbody>
</table>

For every $pc$ we must have

- *$\text{app } I_{\tau}$ is true*
- for every successor $pc'$ of $pc$: *$\text{eff } I_{\tau} \subseteq \tau'$*
Applicability (wrt types!)

\[ app_i :: instr \Rightarrow ty_i \Rightarrow bool \]

\[ app_i (Load \ n) (ST,LT) = (n < |LT| \land LT_{[n]} \neq Err \land |ST| < mxs) \]
\[ app_i (Store \ n) (T \cdot ST, LT) = (n < |LT|) \]
\[ app_i (Push \ v) (ST,LT) = (|ST| < mxs \land typeof v \neq None) \]
\[ app_i (Pop \ T \cdot ST,LT) = \text{True} \]
\[ app_i (Getfield \ F D)(T \cdot ST, LT) = \]
\[ (\exists T_f. \ P \vdash D \text{ sees } F:T_f \text{ in } D \land P \vdash T \leq \text{Class } D) \]
\[ app_i (Putfield \ F D)(T_1 \cdot T_2 \cdot ST, LT) = \]
\[ (\exists T_f. \ P \vdash D \text{ sees } F:T_f \text{ in } D \land P \vdash T_2 \leq \text{Class } D \land P \vdash T_1 \leq T_f) \]
\[ app_i (New \ C) (ST,LT) = (\text{is-class } P \ C \land |ST| < mxs) \]
\[ app_i (Checkcast \ C) (T \cdot ST,LT) = (\text{is-class } P \ C \land \text{is-ref } T \ T) \]
Applicability (wrt types!)

\[app_i \text{ IAdd (} \text{Integer} \cdot \text{Integer} \cdot ST, LT) = True\]
\[app_i \text{ CmpEq (} T_0 \cdot T_2 \cdot ST, LT) = (T_0 = T_2 \lor \text{is-ref}T T_0 \land \text{is-ref}T T_2)\]
\[app_i \text{ (Goto } b)(ST, LT) = True\]
\[app_i \text{ (IfFalse } b)(\text{Boolean} \cdot ST, LT) = True\]
\[app_i \text{ Return (} T \cdot ST, LT) = (P \vdash T \leq T_r)\]
Applicability (wrt types!)

\[ app_i (\text{Invoke } M \ n) \ (ST,LT) = \]
\[ n < |ST| \land \]
\[ (ST[n] \neq NT \rightarrow \]
\[ (\exists \ C \ D \ Ts \ T \ m. \]
\[ \quad ST[n] = \text{Class } C \land P \vdash C \text{ sees } M: Ts \rightarrow T = m \text{ in } D \land \]
\[ P \vdash \text{rev (take } n \ ST) \ [\leq] \ Ts) ) ]

In all other cases:
\[ app_i i (ST,LT) = False \]
Effect (wrt types!)

\[ \text{eff}_i :: \text{instr} \Rightarrow \text{ty}_i \Rightarrow \text{ty}_i \]

\[ \text{eff}_i (\text{Load } n) (\text{T}, \text{LT}) = (\text{let } \text{OK } T = \text{LT}[n] \text{ in } (\text{T} \cdot \text{ST}, \text{LT})) \]

\[ \text{eff}_i (\text{Store } n) (\text{T} \cdot \text{ST}, \text{LT}) = (\text{ST}, \text{LT}[n := \text{OK } T]) \]

\[ \text{eff}_i (\text{Push } v) (\text{ST}, \text{LT}) = (\text{let } [T] = \text{typeof } v \text{ in } (\text{T} \cdot \text{ST}, \text{LT})) \]

\[ \text{eff}_i \text{ Pop } (\text{T} \cdot \text{ST,LT}) = (\text{ST, LT}) \]

\[ \text{eff}_i (\text{Getfield } F D) (\text{T} \cdot \text{ST,LT}) = (\text{snd} (\text{field } P D F) \cdot \text{ST, LT}) \]

\[ \text{eff}_i (\text{Putfield } F D) (\text{T}_1 \cdot \text{T}_2 \cdot \text{ST,LT}) = (\text{ST, LT}) \]

\[ \text{eff}_i (\text{New } C) (\text{ST,LT}) = (\text{Class } C \cdot \text{ST, LT}) \]

\[ \text{eff}_i (\text{Checkcast } C) (\text{T} \cdot \text{ST,LT}) = (\text{Class } C \cdot \text{ST, LT}) \]
**Effect (wrt types!)**

\[
eff_i \text{ IAdd } (\text{Integer} \cdot \text{Integer} \cdot \text{ST}, \text{LT}) = (\text{Integer} \cdot \text{ST}, \text{LT})
\]

\[
eff_i \text{ CmpEq } (T_1 \cdot T_2 \cdot \text{ST}, \text{LT}) = (\text{Boolean} \cdot \text{ST}, \text{LT})
\]

\[
eff_i (\text{Goto } b) (\text{ST}, \text{LT}) = (\text{ST}, \text{LT})
\]

\[
eff_i (\text{IfFalse } b) (\text{Boolean} \cdot \text{ST}, \text{LT}) = (\text{ST}, \text{LT})
\]

\[
eff_i (\text{Invoke } M n) (\text{ST}, \text{LT}) =
\]

\[
\text{let Class } C = \text{ST}[n]; (D, Ts, T_r, b) = \text{method } P C M \text{ in } (T_r \cdot \text{drop } (n + 1) \text{ ST}, \text{LT})
\]

\[
eff_i \text{ Return } (T \cdot \text{ST}, \text{LT}) = \text{ don’t care}
\]
Lifting applicability and effect

Remember: \( ty_i' = ty_i \text{ option} \)

None :: \( ty_i' \) means “unreachable”

\[
\text{app :: instr } \Rightarrow \text{ ty}_i' \Rightarrow \text{ bool }
\]

\[
\text{app instr None } = \text{ True }
\]

\[
\text{app instr } [\tau] = \text{ app}_i \text{ instr } \tau
\]

\[
\text{eff :: instr } \Rightarrow \text{ ty}_i' \Rightarrow \text{ ty}_i'
\]

\[
\text{eff instr None } = \text{ None }
\]

\[
\text{eff instr } [\tau] = [\text{eff}_i \text{ instr } \tau]
\]
Successor instructions

\[\text{succs} :: \text{instr} \Rightarrow \text{pc} \Rightarrow \text{pc}\]

\[\text{succs (Goto } b) \text{ pc} = \{\text{nat}\(\text{int pc} + b)\}\]

\[\text{succs (IfFalse } b) \text{ pc} = \{\text{pc} + 1, \text{nat}\(\text{int pc} + b)\}\]

\[\text{succs Return pc} = \{\}\]

In all other cases:

\[\text{succs } i \text{ pc} = \{\text{pc}+1\}\]
Orderings
\[ \tau \subseteq \tau' \]

\( \tau \) describes a smaller set of JVM states than \( \tau' \)

Examples:

- \(((\text{Class } C), []) \subseteq ((\text{Class } D), [])\) if \( P \vdash C \leq^* D \)
- \((([], [\text{OK Integer}])) \subseteq ([], [\text{Err}])\)

Construction of \( \subseteq \) on \( ty_i' = (ty \, list \times ty \, err \, list) \) option: by recursion over the type structure

Base case: \( \subseteq \) on \( ty \) is subtype relation \( \leq \)
on pairs and lists

Given: \( \sqsubseteq :: 'a \Rightarrow 'a \Rightarrow bool \)
\( \sqsubseteq :: 'b \Rightarrow 'b \Rightarrow bool \)

On pairs: \( \sqsubseteq :: 'a \times 'b \Rightarrow 'a \times 'b \Rightarrow bool \)
\[
(a,b) \sqsubseteq (a',b') \equiv (a \sqsubseteq b \land a' \sqsubseteq b')
\]

On lists: \( \sqsubseteq :: 'a \text{ list} \Rightarrow 'a \text{ list} \Rightarrow bool \)
\[
xs \sqsubseteq ys \equiv (|xs| = |ys| \land (\forall i < |xs|. xs[i] \sqsubseteq ys[i]))
\]
Given: $\sqsubseteq :: 'a \Rightarrow 'a \Rightarrow \text{bool}$

Extension: $\sqsubseteq :: 'a \text{ option} \Rightarrow 'a \text{ option} \Rightarrow \text{bool}$

\[
\begin{align*}
\text{None} \sqsubseteq x &= \text{True} \\
[a] \sqsubseteq [b] &= a \sqsubseteq b
\end{align*}
\]

$\implies$ None is bottom element
Given: $\sqsubseteq :: 'a \Rightarrow 'a \Rightarrow \text{bool}$

Extension: $\sqsubseteq :: 'a \text{ err} \Rightarrow 'a \text{ err} \Rightarrow \text{bool}$

$x \sqsubseteq \text{Err} = \text{True}$

$\text{OK a} \sqsubseteq \text{OK b} = a \sqsubseteq b$

$\Rightarrow \text{Err is top element}$
Example

\[
\begin{align*}
\text{None} & \sqsubseteq \left( [\text{Class C}, [\text{OK Boolean}, \text{OK Integer}]] \right) \\
& \sqsupseteq \left( [\text{Class C}, [\text{OK Boolean}, \text{Err}]] \right) \\
& \sqsupseteq \left( [\text{Class C}, [\text{Err}, \text{Err}]] \right) \\
& \sqsupseteq \left( [\text{Class Object}, [\text{Err}, \text{Err}]] \right)
\end{align*}
\]
Remember the fixed method context

\[ C_M :: \text{cname} \quad \text{the class the method is defined in} \]

\[ T_s :: \text{ty list} \quad \text{parameter types} \]

\[ T_r :: \text{ty} \quad \text{return type} \]

\[ mxs :: \text{nat} \quad \text{maximum stack size} \]

\[ mxl_0 :: \text{nat} \quad \text{number of local variables (w/o parameters)} \]

\[ mxl :: \text{nat} \quad \text{number of registers} \quad (mxl = 1 + |T_s| + mxl_0) \]

\[ is :: \text{instr list} \quad \text{instructions} \]
The method is well-typed w.r.t. $\tau s$ iff

- $is \neq [] \land |\tau s| = |is|$
- $\forall ST \ LT. [(ST, LT)] \in set \tau s \rightarrow |ST| < mxs \land |LT| = mxl$
- $[([], OK (Class C_M) \cdot map \ OK \ Ts \ @ \ replicate \ mxl_0 \ Err)] \subseteq \tau_s[0]$
- $\forall p < |is|. \forall q \in succs is[p] \ p. \ q < |is|$
- $\forall p < |is|. \ app is[p] \tau_s[p]$
- $\forall p < |is|. \forall q \in succs is[p] \ p. \ eff is[p] \tau_s[p] \subseteq \tau_s[q]$
The method declaration $M : Ts \to T_r = (mxs, mxl_0, is, [])$ in class $C_M$ is well-formed w.r.t. $\tau s$ iff

- All parameter types are valid: $\forall T \in \text{set } Ts. \text{ is-type } P T$
- The result type is valid: $\text{is-type } P T_r$
- The declaration is well-typed w.r.t. $\tau s$ (as defined above).
A JVM program $P$ is well-formed w.r.t.

$$\Phi :: \text{name} \Rightarrow \text{mname} \Rightarrow \text{ty} \Rightarrow \text{list}$$

iff it is a well-formed Jinja program where each declaration of a method $M$ in class $C_M$ is well-formed w.r.t. $\Phi C_M M$ (as defined above, not as in Jinja!).

Well-formed program
Type safety of the JVM
Conformance

JVM cannot get stuck but check-instr can be violated.

Let $P$ be well-formed w.r.t. $\Phi$.

1. Define conformance between JVM state and $P,\Phi$.
2. Show that execution of $P$ preserves conformance.
3. Show that if the state conforms to $P,\Phi$, check-instr is true.
Conformance of stack and registers

- Stack:

\[ P, h \vdash [v_1, \ldots, v_m] [:\leq] [T_1, \ldots, T_n] \equiv \]
\[ m = n \land P, h \vdash v_1 :\leq T_1 \land \ldots \land P, h \vdash v_n :\leq T_n \]

- Registers:

\[ P, h \vdash [v_1, \ldots, v_m] [:\leq_T] [U_1, \ldots, U_n] \equiv \]
\[ m = n \land P, h \vdash v_1 :\leq_T U_1 \land \ldots \land P, h \vdash v_n :\leq_T U_n \]

where

\[ P, h \vdash v :\leq_T Err \]

\[ (P, h \vdash v :\leq_T OK T) = (P, h \vdash v :\leq T) \]
Conformance

- Frame \((stk, loc, C, M, pc)\) conforms to \(P, \Phi\):
  - \(P \vdash C \text{ sees } M : \ldots \rightarrow \ldots = (\ldots, is, \ldots) \text{ in } C\)
  - \((\Phi \ C \ M)_{[pc]} = [(ST, LT)]\)
  - \(P, h \vdash stk \[\leq\] ST \land P, h \vdash loc \[\leq_{\top}\] LT\)
  - if it is not the top frame: \(is_{[pc]} = \text{Invoke } M_0 \ n_0\) and \(M_0\) is the method in the frame above.

- JVM state:
  \[P, \Phi \vdash (_, h, frs) \sqrt{\text{iff}}\]
  \(P \vdash h \sqrt{\text{iff}} \text{ and every frame in } frs \text{ conforms to } P, \Phi.\)
Type safety

**Thm** (Preservation of conformance) If $P$ is well-formed w.r.t. $\Phi$ and $P,\Phi \vdash \sigma \checkmark$ and $\text{exec } P \sigma = [\sigma']$ then $P,\Phi \vdash \sigma' \checkmark$.

**Proof** by case distinction over the instructions.

**Thm** If $P$ is well-formed w.r.t. $\Phi$ and $P,\Phi \vdash \sigma \checkmark$ and the computation of $\text{exec } P \sigma$ calls $\text{exec-instr}$ ... then $\text{check-instr}$ ... (with the same parameters) is true.

**Proof** by case distinction over the instructions.