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:

$$\ldots, \mathit{Push}\ (\mathit{Addr}\ 217), \mathit{Push}\ v, \mathit{Putfield}\ F\ D, \ldots$$

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

$\Longrightarrow$ no *Addr*esses in bytecode!

How about

$$\ldots, \mathit{Push}\ (\mathit{Intg}\ 217), \mathit{Push}\ v, \mathit{Putfield}\ F\ D, \ldots$$

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

Code	Stack	Registers
`Push` (*Intg 1*)	[]	[*Class C*, *Integer*]
`Load` *1*		
`IAdd`		
`Store` *0*		

Motto

Types are abstract descriptions of values/states

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

“Abstract interpretation”

Restriction

We treat only the exception-free fragment of the JVM
No system exceptions, no user exception

Type System

Stack type

- $ty_s = ty\ list$ ($ST :: ty_s$)

At each program point,
the Jinja type of each stack element
must be uniquely defined.

Register types

datatype $'a\ err = Err \mid OK\ 'a$

- $ty_l = ty\ err\ list$ $\quad (LT :: ty_l)$

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

Code	Stack	Registers
`IfFalse` *4*	[*Boolean*]	[*OK* (*Class C*)]
`Push` (*Intg 1*)		
`Store` *0*		
`Goto` *3*		
`Push` (*Bool True*)		
`Store` *0*		

Reasons for Err

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

  Example: `if` (b) $\{V_1{:}T_1; \_\}$ `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:
  $ty_i' = ty_i$ *option* $\quad (\tau :: ty_i')$

  None : instruction is unreachable

  $\lfloor \tau \rfloor$: state before/after instruction has type τ.

- Method type: ty_i' *list* $\quad (\tau s :: ty_i'$ *list*$)$

An example of the full truth

Code	Stack Registers
`Goto` *2*	$\lfloor([\mathit{Boolean}], \quad [\mathit{Err}])\rfloor$
`Pop`	
`Push` $(\mathit{Bool\ True})$	
`Store` *0*	

Well-typed Method

A fixed context

We consider a fixed method in a fixed JVM program P:

C_M	:: *cname*	the class the method is defined in
Ts	:: *ty list*	parameter types
T_r	:: *ty*	return type
mxs	:: *nat*	maximum stack size
mxl_0	:: *nat*	number of local variables (w/o parameters)
mxl	:: *nat*	number of registers ($mxl = 1 + \|Ts\| + mxl_0$)
is	:: *instr list*	instructions

Well-typed method (informal)

The method is well-typed iff there is a method type τs such that

- $|\tau s| = |is|$
- Each instruction is *applicable* to the corresponding instruction type in τs.
  Eg: no Pop when $ST = []$.
- The state after the execution of the instruction is "compatible" with all successor positions.
  Eg: if $ST = [T]$ before Pop then $ST = []$ afterwards.

Well-typed method (almost formal)

⋮	
$pc\colon l$	τ
⋮	
$pc'\colon l'$	τ'
⋮	

For every *pc* we must have

- $\mathit{app}\ l\ \tau$ is true
- for every successor pc' of pc: $\mathit{eff}\ l\ \tau \sqsubseteq \tau'$

Applicability (wrt types!)

$app_i :: instr \Rightarrow ty_i \Rightarrow bool$

$app_i\ (\texttt{Load}\ n)\ (ST,LT) = (n < |LT| \wedge LT_{[n]} \neq Err \wedge |ST| < mxs)$

$app_i\ (\texttt{Store}\ n)\ (T \cdot ST, LT) = (n < |LT|)$

$app_i\ (\texttt{Push}\ v)\ (ST,LT) = (|ST| < mxs \wedge typeof\ v \neq None)$

$app_i\ \texttt{Pop}\ (T \cdot ST,LT) = True$

$app_i\ (\texttt{Getfield}\ F\ D)\ (T \cdot ST, LT) =$

$(\exists T_f.\ P \vdash D\ sees\ F{:}T_f\ in\ D \wedge P \vdash T \leq Class\ D)$

$app_i\ (\texttt{Putfield}\ F\ D)\ (T_1 \cdot T_2 \cdot ST, LT) =$

$(\exists T_f.\ P \vdash D\ sees\ F{:}T_f\ in\ D \wedge P \vdash T_2 \leq Class\ D \wedge P \vdash T_1 \leq T_f)$

$app_i\ (\texttt{New}\ C)\ (ST,LT) = (is\text{-}class\ P\ C \wedge |ST| < mxs)$

$app_i\ (\texttt{Checkcast}\ C)\ (T \cdot ST,LT) = (is\text{-}class\ P\ C \wedge is\text{-}refT\ T)$

Applicability (wrt types!)

$$app_i\ \texttt{IAdd}\ (\mathit{Integer} \cdot \mathit{Integer} \cdot ST, LT) = \mathit{True}$$

$$app_i\ \texttt{CmpEq}\ (T_1 \cdot T_2 \cdot ST, LT) = (T_1 = T_2 \vee \mathit{is\text{-}refT}\ T_1 \wedge \mathit{is\text{-}refT}\ T_2)$$

$$app_i\ (\texttt{Goto}\ b)\ (ST, LT) = \mathit{True}$$

$$app_i\ (\texttt{IfFalse}\ b)\ (\mathit{Boolean} \cdot ST, LT) = \mathit{True}$$

$$app_i\ \texttt{Return}\ (T \cdot ST, LT) = (P \vdash T \leq T_r)$$

Applicability (wrt types!)

$$
\begin{aligned}
&app_i\ (\texttt{Invoke}\ M\ n)\ (ST, LT) = \\
&n < |ST| \wedge \\
&(ST_{[n]} \neq NT \longrightarrow \\
&\quad (\exists C\ D\ Ts\ T\ m. \\
&\qquad ST_{[n]} = Class\ C \wedge P \vdash C\ sees\ M: Ts \rightarrow T = m\ in\ D \wedge \\
&\qquad P \vdash rev\ (take\ n\ ST)\ [\leq]\ Ts))
\end{aligned}
$$

In all other cases:

$app_i\ i\ (ST, LT) = False$

Effect (wrt types!)

$$eff_i :: instr \Rightarrow ty_i \Rightarrow ty_i$$

$$eff_i\ (\texttt{Load}\ n)\ (ST,LT) = (let\ OK\ T = LT_{[n]}\ in\ (T \cdot ST, LT))$$
$$eff_i\ (\texttt{Store}\ n)\ (T \cdot ST, LT) = (ST, LT[n := OK\ T])$$
$$eff_i\ (\texttt{Push}\ v)\ (ST,LT) = (let\ \lfloor T \rfloor = typeof\ v\ in\ (T \cdot ST, LT))$$
$$eff_i\ \texttt{Pop}\ (T \cdot ST,LT) = (ST, LT)$$
$$eff_i\ (\texttt{Getfield}\ F\ D)\ (T \cdot ST, LT) = (snd\ (field\ P\ D\ F) \cdot ST, LT)$$
$$eff_i\ (\texttt{Putfield}\ F\ D)\ (T_1 \cdot T_2 \cdot ST, LT) = (ST, LT)$$
$$eff_i\ (\texttt{New}\ C)\ (ST,LT) = (Class\ C \cdot ST, LT)$$
$$eff_i\ (\texttt{Checkcast}\ C)\ (T \cdot ST,LT) = (Class\ C \cdot ST, LT)$$

Effect (wrt types!)

$$eff_i\ \texttt{IAdd}\ (Integer \cdot Integer \cdot ST, LT) = (Integer \cdot ST, LT)$$

$$eff_i\ \texttt{CmpEq}\ (T_1 \cdot T_2 \cdot ST, LT) = (Boolean \cdot ST, LT)$$

$$eff_i\ (\texttt{Goto}\ b)\ (ST, LT) = (ST, LT)$$

$$eff_i\ (\texttt{IfFalse}\ b)\ (Boolean \cdot ST, LT) = (ST, LT)$$

$$eff_i\ (\texttt{Invoke}\ M\ n)\ (ST, LT) =$$

$$let\ Class\ C = ST_{[n]};\ (D, Ts, T_r, b) = method\ P\ C\ M\ in$$

$$(T_r \cdot drop\ (n + 1)\ ST, LT)$$

$$eff_i\ \texttt{Return}\ (T \cdot ST, LT) =\ \text{don't care}$$

Lifting applicability and effect

Remember: $ty_i' = ty_i\ option$

$None :: ty_i'$ means “unreachable”

$app :: instr \Rightarrow ty_i' \Rightarrow bool$

$app\ instr\ None = True$

$app\ instr\ \lfloor\tau\rfloor = app_i\ instr\ \tau$

$eff :: instr \Rightarrow ty_i' \Rightarrow ty_i'$

$eff\ instr\ None = None$

$eff\ instr\ \lfloor\tau\rfloor = \lfloor eff_i\ instr\ \tau\rfloor$

Successor instructions

$succs :: instr \Rightarrow pc \Rightarrow pc$

$succs\ (\texttt{Goto}\ b)\ pc = \{nat(int\ pc + b)\}$

$succs\ (\texttt{IfFalse}\ b)\ pc = \{pc + 1, nat\ (int\ pc + b)\}$

$succs\ \texttt{Return}\ pc = \{\}$

In all other cases:

$succs\ i\ pc = \{pc+1\}$

Orderings

$\sqsubseteq$

$$\tau \sqsubseteq \tau'$$

τ describes a smaller set of JVM states than τ'

Examples:

- $([\mathit{Class\ C}], []) \sqsubseteq ([\mathit{Class\ D}], [])$ if $P \vdash C \preceq^* D$
- $([], [\mathit{OK\ Integer}]) \sqsubseteq ([], [\mathit{Err}])$

Construction of $\sqsubseteq$ on $\mathit{ty}_i' = (\mathit{ty\ list} \times \mathit{ty\ err\ list})\mathit{option}$:
by recursion over the type structure

Base case: $\sqsubseteq$ on ty is subtype relation $\leq$

$\sqsubseteq$ on pairs and lists

Given: $\sqsubseteq :: 'a \Rightarrow 'a \Rightarrow bool$

$\sqsubseteq :: 'b \Rightarrow 'b \Rightarrow bool$

On pairs: $\sqsubseteq :: 'a \times 'b \Rightarrow 'a \times 'b \Rightarrow bool$

$$(a,b) \sqsubseteq (a',b') \equiv (a \sqsubseteq b \wedge a' \sqsubseteq b')$$

On lists: $\sqsubseteq :: 'a\ list \Rightarrow 'a\ list \Rightarrow bool$

$$xs \sqsubseteq ys \equiv (|xs| = |ys| \wedge (\forall i{<}|xs|.\ xs_{[i]} \sqsubseteq ys_{[i]}))$$

$\sqsubseteq$ *on option*

Given: $\sqsubseteq :: 'a \Rightarrow 'a \Rightarrow bool$

Extension: $\sqsubseteq :: 'a\ option \Rightarrow 'a\ option \Rightarrow bool$

$$None \sqsubseteq x = True$$
$$\lfloor a \rfloor \sqsubseteq \lfloor b \rfloor = a \sqsubseteq b$$

$\Longrightarrow$ *None* is bottom element

$\sqsubseteq$ *on err*

Given: $\sqsubseteq :: 'a \Rightarrow 'a \Rightarrow bool$

Extension: $\sqsubseteq :: 'a\ err \Rightarrow 'a\ err \Rightarrow bool$

$$x \sqsubseteq Err = True$$
$$OK\ a \sqsubseteq OK\ b = a \sqsubseteq b$$

$\Longrightarrow$ *Err* is top element

Example

$$
\begin{aligned}
\mathit{None} \sqsubseteq\ & \lfloor([\mathit{Class\ C}], [\mathit{OK\ Boolean}, \mathit{OK\ Integer}])\rfloor \\
\sqsubseteq\ & \lfloor([\mathit{Class\ C}], [\mathit{OK\ Boolean}, \mathit{Err}])\rfloor \\
\sqsubseteq\ & \lfloor([\mathit{Class\ C}], [\mathit{Err}, \mathit{Err}])\rfloor \\
\sqsubseteq\ & \lfloor([\mathit{Class\ Object}], [\mathit{Err}, \mathit{Err}])\rfloor
\end{aligned}
$$

Rememeber the fixed method context

C_M	::	*cname*	the class the method is defined in
Ts	::	*ty list*	parameter types
T_r	::	*ty*	return type
mxs	::	*nat*	maximum stack size
mxl_0	::	*nat*	number of local variables (w/o parameters)
mxl	::	*nat*	number of registers ($mxl = 1 + \|Ts\| + mxl_0$)
is	::	*instr list*	instructions

Well-typed method

The method is well-typed w.r.t. τs iff

- $is \neq [] \wedge |\tau s| = |is|$
- $\forall\, ST\ LT.\ \lfloor (ST, LT) \rfloor \in set\ \tau s \longrightarrow |ST| < mxs \wedge |LT| = mxl$
- $\lfloor ([], OK\ (Class\ C_M) \cdot map\ OK\ Ts\ @\ replicate\ mxl_0\ Err) \rfloor \sqsubseteq \tau s_{[0]}$
- $\forall\, p < |is|.\ \forall\, q \in succs\ is_{[p]}\ p.\ q < |is|$
- $\forall\, p < |is|.\ app\ is_{[p]}\ \tau s_{[p]}$
- $\forall\, p < |is|.\ \forall\, q \in succs\ is_{[p]}\ p.\ eff\ is_{[p]}\ \tau s_{[p]} \sqsubseteq \tau s_{[q]}$

Well-formed method

The method declaration $M : Ts \to T_r = (mxs, mxl_0, is, [])$ in class C_M is well-formed w.r.t. τs iff

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

Well-formed program

A JVM program *P* is well-formed w.r.t.

$$\Phi :: cname \Rightarrow mname \Rightarrow ty_i' \; list$$

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!).

Type safety of the JVM

Conformance

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

Let P be well-formed w.r.t. Φ.

1. Define conformance between JVM state and P, Φ.
2. Show that execution of P preserves conformance.
3. Show that if the state conforms to P, Φ, *check-instr* is true.

Conformance of stack and registers

- Stack:

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

- Registers:

  $P,h \vdash [v_1,\ldots,v_m]\ [:\leq_\top]\ [U_1,\ldots,U_n] \equiv$
  $m = n \wedge P,h \vdash v_1 :\leq_\top U_1 \wedge \ldots \wedge P,h \vdash v_n :\leq_\top U_n$

  where

  $P,h \vdash v :\leq_\top Err$
  $(P,h \vdash v :\leq_\top OK\ T) = (P,h \vdash v :\leq T)$

Conformance

- Frame (stk,loc,C,M,pc) conforms to P,Φ:
  - $P \vdash C$ *sees* $M : \ldots \to \ldots = (\ldots,is,\ldots)$ *in* C
  - $(\Phi\ C\ M)_{[pc]} = \lfloor(ST, LT)\rfloor$
  - $P,h \vdash stk\ [:\leq]\ ST\ \wedge\ P,h \vdash loc\ [:\leq_\top]\ LT$
  - if it is not the top frame: $is_{[pc]} = \texttt{Invoke}\ M_0\ n_0$ and M_0 is the method in the frame above.
- JVM state:

  $P,\Phi \vdash (\_, h, frs)\ \surd$ iff
  $P \vdash h\ \surd$ and every frame in *frs* conforms to P,Φ.

Type safety

Thm (Preservation of conformance) If *P* is well-formed w.r.t. Φ and $P,\Phi \vdash \sigma\ \surd$ and *exec* $P\ \sigma = \lfloor \sigma' \rfloor$ then $P,\Phi \vdash \sigma'\ \surd$.
Proof by case distinction over the instructions.

Thm If *P* is well-formed w.r.t. Φ and $P,\Phi \vdash \sigma\ \surd$ and the computation of *exec* $P\ \sigma$ calls *exec-instr* $\ldots$ then *check-instr* $\ldots$ (with the same parameters) is true.
Proof by case distinction over the instructions.