

Type Safe Dynamic Linking for JVM Access Control

Christian Skalka

Department of Computer Science

University of Vermont



Outline of Presentation

1. Review of JVM access control
2. Static enforcement and the problem with dynamic linking
3. Summary and contributions of our approach
4. Details of our approach
5. Summary of formal results
6. Conclusion

Background: JVM Access Control

Java JDK1.2 SecurityManager provides a PL-based security mechanism for defining and enforcing *access control* policies:

- All classes are signed by the class *owner* (principal)
- Local policy maps owners to sets of authorized privileges in *access control lists* (ACLs)
- Functions are made privileged by an explicit check for specific privileges `r` (`check(r)`), e.g. `check(FileWritePrivilege)`.

Background: JVM Access Control

- Any `check(r)` enforces authorization for `r` via a dynamic check (*stack inspection*).
- If `check(r)` occurs in a method `m`, then use of `m` requires authorization for `r`.
- If `m` requires authorization for `r`, then so does any method that calls `m`.

```
class Bar extends Runnable {  
    Object run(Object o) {check(r1); return o }  
}
```

```
(new Bar()).run(new Object())    requires authorization for r1
```

Static Enforcement of Stack Inspection

It is well known that stack inspection security can be statically enforced, with significant efficiency benefits, earlier detection of errors.

“Static Enforcement of Security with Types”, Skalka & Smith ICFP00

- Enforcement via types (Skalka et al.)
- Enforcement via control-flow analysis (Jensen et al.)
- Advanced language models (Higuchi et al., Skalka)

So well-studied that it's now a *canonical example*. But...

Dynamic Linking

The JVM provides *bytecode verification* to support safe dynamic linking of non-local code.

- Dynamic *extensibility* is motivation for JDK security model.
- Dynamic linking disallows whole program static analysis.
- Therefore, current analyses incompatible with practical requirements of the JVM.
- *Our goal*: to extend bytecode verification with analysis to enforce access control at *link time*.

Our Technique: *Reification* of Type Constraints

Basic idea: maintain a partial type approximation of linked code. Be abstract about unlinked references. Extend approximation (“reify”) as new code is linked.

- Types of methods and classes are represented as *constrained types* T/C
 - T is the type of the method or class
 - C is a constraint that “embodies” the type
- for program references to unlinked methods and classes, T is abstract and C is trivial: t/\mathbf{true} .
- as code is linked, the type is reified by filling in the details of constraints through extended bytecode verification: t/D , where D is inferred

Features of Our System

- *Reification soundness*: unlinked references are “given a pass”, but type errors will be detected before their execution (upon linkage).
- Reification has same precision and complexity as inference.
- Bytecode level analysis (not in this presentation).
- Support for OO features: inheritance and dynamic dispatch.
 - Different method versions are assigned effects independently (contrast with Higuchi and Ohori 2007).

Details of Type Form

Our types characterize properties of classes as deduced from code, including *authorization properties*. For a class `ClassName` with methods $m_1..m_n$:

$$\text{ClassName} : \forall v_1..v_k [C]. [T \ \text{ClassName}]$$

- $v_1..v_k$ are *polymorphic type variables*
- C constrains the class method types
- T is the type of methods in the class:

$$T = m_1 : T_1 .. m_n : T_n$$

Details of Type Form: Method Types

Method type form: $T_1..T_j \xrightarrow{R} T$

- $T_1..T_j$ are expected argument class types
- T is the returned class type
- R describes the *authorization requirements* of the method
 - R is a *row type*; intuitively, R will specify a resource as “present” (Pre) iff it is required for authorization
 - E.g. if r is required for authorization, then R is constrained to be of the form $\{r : \text{Pre}; \dots\}$.

Example

```
class Bar extends Runnable {  
  Object run(Object o) {check(r1); return o }  
}  
class Baz extends Runnable {  
  Object run(Object o) {return o }  
}
```

Baz : $\forall v_5[\mathbf{true}].[\text{run} : \text{Unit} \xrightarrow{\{v_5\}} \text{Unit} \text{ Baz}]$

Bar : $\forall v_3, v_4[(r_1 : \text{Pre}; v_3) = v_4].[\text{run} : \text{Unit} \xrightarrow{\{v_4\}} \text{Unit} \text{ Bar}]$

Where $\text{unitt} \triangleq [\text{Object}]$.

Trickiness: Dynamic Dispatch

Dynamic dispatch presents a challenge— what is the authorization requirement?

```
class Foo extends Object {  
  Object m(Object o, Runnable x) {return x.run(o) }  
}
```

Polymorphism allows flexibility to range over multiple versions of run; letting:

$$T \triangleq (\text{Unit}, [\text{run} : \text{Unit} \xrightarrow{\{v\}} \text{Unit Runnable}]) \xrightarrow{\{v\}} \text{Unit}$$

we obtain a type for Foo that is specialized only as needed:

$$\text{Foo} : \forall v[\mathbf{true}]. [m : T \text{ Foo}]$$

Abstract variables stand in for types of unknown code.

Instantiation in Different Contexts

Consider the following code snippets from the same program:

```
(new Foo()).m(new Object(), new Bar())           (1)
```

```
(new Foo()).m(new Object(), new Baz())           (2)
```

These two instances of `Foo` will be assigned distinct type instances:

$$[m : T \text{ Foo}][v_{(1)}/v] \qquad [m : T \text{ Foo}][v_{(2)}/v]$$

Computation of a solution to the constraints induced by `Bar` and `Baz`'s versions of `run` will obtain *only*:

$$r_1 : \text{Pre}; v_3 = v_{(1)} \qquad v_4 = v_{(2)}$$

Dynamic Linking and Reification

Extend this idea: use abstract type variables t to represent types of unlinked classes C .

Type variables t provide entry points into type approximation of object instances of C , filled in by link-time bytecode verification.

Execution model: $(CT, e, \Gamma) \rightarrow (CT', e', \Gamma')$

- CT is the linked class table
- e is the running program
- Γ is the inferred type of the class table, with abstract types for unlinked classes C mentioned in e and CT .

Dynamic Linking and Reification

Pre-linkage type of C: $\Gamma(C) = \forall t^C[\mathbf{true}].[t^C C]$

Linking evaluation rule:

$$\frac{\Gamma \vdash_W C : \sigma \quad \Gamma' = \text{reify}(\Gamma, \sigma)}{(CT, e, \Gamma) \rightarrow (CT[\text{class } C \text{ extends } \dots], e, \Gamma')}$$

where $\sigma \triangleq \forall t^C, v_1, \dots, v_n[D].[t^C C]$

Reification accomplishes the following:

- C's binding in Γ is updated with inferred constraint D .
- Every instance of $\Gamma(C)$ in Γ is updated with D ...
 - *Each instance update must provide fresh instance of D .*

Example

`(new Foo()).m(new Object(), new Bar())` (1)

Linked class table:

Class Bar extends Runnable{...}
Class Baz extends Runnable{...}

Foo class type scheme:

$$\forall t^{\text{Foo}}[\mathbf{true}].[t^{\text{Foo}} \text{Foo}]$$

Type instance associated with the Foo object in (1):

$$[t_{(1)}^{\text{Foo}} \text{Foo}]$$

Example

`(new Foo()).m(new Object(), new Bar())` (1)

Linked class table:

Class Bar extends Runnable{...}
Class Baz extends Runnable{...}

Foo class type scheme:

$$\forall t^{\text{Foo}}[\mathbf{true}].[t^{\text{Foo}} \text{Foo}]$$

Type instance associated with the Foo object in (1):

$$[t_{(1)}^{\text{Foo}} \text{Foo}]$$

Example

`(new Foo()).m(new Object(), new Bar())` (1)

Linked class table:

Class Bar extends Runnable{...}
Class Baz extends Runnable{...}

Foo class type scheme:

$\forall t^{\text{Foo}}[\mathbf{true}].[t^{\text{Foo}} \text{Foo}]$

Type instance associated with the Foo object in (1):

$[t_{(1)}^{\text{Foo}} \text{Foo}]$

Example

`(new Foo()).m(new Object(), new Bar())` (1)

Linked class table:

```
Class Bar extends Runnable{...}
Class Baz extends Runnable{...}
Class Foo extends Object{...}
```

Inferred Foo class type scheme:

$$\forall v, t^{\text{Foo}} [(m : T) = t^{\text{Foo}}]. [t^{\text{Foo}} \text{Foo}]$$

Reified type constraints associated with the Foo instance in (1):

$$[(m : T[v_{(1)}/v]) \text{Foo}] = [t_{(1)}^{\text{Foo}} \text{Foo}] \quad r_1 : \text{Pre}; v_3 = v_{(1)}$$

Example: Instantiation in Different Contexts (Pre-Linking)

`(new Foo()).m(new Object(), new Bar())` (1)

`(new Foo()).m(new Object(), new Baz())` (2)

Foo class type scheme (pre-linking):

$$\forall t^{\text{Foo}}[\mathbf{true}].[t^{\text{Foo}} \text{Foo}]$$

Type instances associated with the Foo objects in (1) and (2) (pre-linking):

$$\begin{aligned} & [t_{(1)}^{\text{Foo}} \text{Foo}] \\ & [t_{(2)}^{\text{Foo}} \text{Foo}] \end{aligned}$$

Example: Instantiation in Different Contexts (Post-Linking)

$(\text{new Foo}()).\text{m}(\text{new Object}(), \text{new Bar}())$ (1)

$(\text{new Foo}()).\text{m}(\text{new Object}(), \text{new Baz}())$ (2)

Inferred Foo class type scheme:

$$\forall v, t^{\text{Foo}}[(m : T) = t^{\text{Foo}}].[t^{\text{Foo}} \text{Foo}]$$

Type constraints associated with the Foo objects in (1) and (2):

$$[(m : T[v_{(1)}/v]) \text{Foo}] = [t_{(1)}^{\text{Foo}} \text{Foo}]$$

$$[(m : T[v_{(2)}/v]) \text{Foo}] = [t_{(2)}^{\text{Foo}} \text{Foo}]$$

Induced constraints:

$$r_1 : \text{Pre}; v_3 = v_{(1)} \qquad v_4 = v_{(2)}$$

Summary of Results

- *Link-time type safety*: if Γ contains sound typings for e and CT , then evaluation of (CT, e, Γ) will not result in authorization failure.
- Our analysis is conservative with respect to existing Java type analysis.
- Purely static version of our analysis is more precise than existing bytecode analyses for access control.
- Link-time analysis has the same precision and complexity as purely static version.

Conclusion

Contribution in a nutshell: extension to bytecode verification for link-time enforcement of JVM access control, with flexibility for realistic applications.

Future work: generalizing link-time analysis over arbitrary effect systems.

<http://www.cs.uvm.edu/~skalka>