

# Static Use Based Object Confinement

Christian Skalka and Scott Smith  
The Johns Hopkins University

## Object confinement: what is it?

*Object confinement* is concerned with the *encapsulation*, or protection, of object references

- code boundaries define usage *domains*
  - classes, packages
  - code ownership
- sensitive references restricted to certain domains

Object confinement systems provide more expressive *specification*, and more reliable *enforcement*, of reference flow among domains

## Object confinement: motivations

Beyond good programming practice, object confinement is a *security* issue; current Java mechanisms insufficient. For example\*:

```
private Identity[] signers

public Identity[] getSigners( ) {
    return signers;
}
```

Reference leak circumvents JDK1.2 security mechanism!

\**Bokowski and Vitek, "Confined Types*

## Object confinement: strategies

Our focus: *type-based* approaches to *static* enforcement of confinement.

- previous typed approaches: *communication*-based
  - Bokowski and Vitek, “Confined Types”
  - Clarke, Potter and Noble, “Ownership Types for Flexible Alias Protection”

These approaches enforce security at the point of communication across boundaries:

- for any object message send  $o.m(o')$ , the domain associated with  $o'$  must be accessible to the domain associated with  $o$

## Use-based object confinement

Our approach is *use*-based. We focus on how references are used within domains:

- for any object message send  $o.m(o')$ , the current code domain must be authorized for the use of  $o$ 's method  $m$

This approach has distinct benefits:

- a more *fine-grained* security specification
  - allows for more or less restrictive views, rather than all-or-nothing
- supports protocols where *untrusted intermediaries* are used, e.g. tunneling

## The pop system

To provide a theoretical foundation for our approach to object confinement, we develop the pop system, comprising an *OO language* core:

- object annotations for specifying confinement policies
  - object *domain* specifications
  - object *usage* specifications
- *run-time checks* enforce security policies

The language is low-level and flexible, can model a variety of higher-level systems: class and package definitions, code ownership systems...

## The pop system

The pop system also includes a *type discipline* for *static* enforcement of object confinement security:

- static enforcement of security means run-time checks can be eliminated, allowing *optimizations*
- static enforcement of security allows quicker detection of threats
- types enhance *readability* of policies
- type system for pop developed using advanced techniques, exploits well-founded previous work

## The pop language: objects

The pop language includes a familiar language of objects:

$$[\text{read}() = \dots, \text{write}(x) = \dots]$$

In addition to method definitions, objects are assigned *domain labels*  $d$ :

$$[\text{read}() = \dots, \text{write}(x) = \dots] \cdot d$$

The *meaning* of domains is flexible, and open to interpretation; e.g. domain labels may specify a code owner, or a package name, etc.

## The pop language: object interfaces

Objects are also endowed with *interfaces*  $\varphi$ , which specify the per-domain access rights to the object:

$$[\text{read}() = \dots, \text{write}(x) = \dots] \cdot d \cdot \varphi$$

Interfaces are mappings from domains to sets of object method names, and include a default domain  $\partial$ :

$$[\text{read}() = \dots, \text{write}(x) = \dots] \cdot d \cdot \{d \mapsto \{\text{read}, \text{write}\}, \partial \mapsto \{\text{read}\}\}$$

These interfaces are checked at run-time to ensure that any object use is authorized

## pop examples

Assume the following definition:

$$o \triangleq [\text{read}() = \dots, \text{write}(x) = \dots] \cdot d \cdot \{d \mapsto \{\text{read}, \text{write}\}, \partial \mapsto \{\text{read}\}\}$$

Let  $d' \neq d$  be the current domain:

- $o.\text{write}(v)$  will *fail*,  $o.\text{read}()$  will succeed

Let  $d$  be the current domain:

- $o.\text{write}(v)$  will succeed,  $o.\text{read}()$  will succeed

## The pop language: casting

The pop language also includes a *casting* mechanism, that allows object access rights to be *removed* (run-time enforcement of downcasting):

- $o \downarrow (d, \iota)$  modifies the interface associated with  $o$  to map  $d$  to  $\iota$

For example, letting:

$$o \triangleq [\text{read}() = \dots, \text{write}(x) = \dots] \cdot d \cdot \{d \mapsto \{\text{read}, \text{write}\}, \partial \mapsto \{\text{read}\}\}$$

The following casts have the described results:

- $o \downarrow (d, \{\text{read}\})$  yields a read-only file object
- $o \downarrow (\partial, \{\emptyset\})$  yields an object *unuseable* outside  $d$

## Types for pop

We develop a static type discipline that predicts dynamic behavior wrt confinement specifications:

- types reflect object interfaces, usage requirements
- developed using *transformational approach*, allowing reuse of existing type safety results, implementations

## Transformational Approach

Type system for expressions  $e$  in  $\text{pop}$  obtained by transformation  $\langle e \rangle$ :

- $\langle e \rangle$  is a term in a familiar *target language* pre-equipped with sound type system, including inference algorithm
- Transformation preserves semantics:

**Theorem:** If  $e$  safely evaluates to  $v$ , then  $\langle e \rangle$  safely evaluates to  $\langle v \rangle$ . If  $e$  has runtime errors, then so does  $\langle e \rangle$ . If  $e$  diverges, then  $\langle e \rangle$  diverges.

## Transformational Approach

Correctness of term transformation  $\langle e \rangle$  yields a source language type system “for free”— without further proof effort:

- Sound *indirect* type system for expressions  $e$  obtained from target type system: if  $\langle e \rangle : \tau$  then  $e : \tau$
- Since  $\langle e \rangle : \tau$  can be inferred, compose transformation and type inference to infer  $e : \tau$
- Method yields insight into semantics and/or desired structure of *direct* types for source language, eases proof development

## Transforming pop

We transform pop into pml, a functional language with *records*, *sets*, and an accurate type system\*

The transformation of interfaces  $\varphi$  is denoted  $\hat{\varphi}$ , and uses records with sets as field values in the image:

$$\{d_1 \mapsto \iota_1, \dots, \widehat{d_n \mapsto \iota_n}, \partial \mapsto \iota\} = \{d_1 = \iota_1, \dots, d_n = \iota_n, \partial = \iota\}$$

Objects are transformed roughly as follows:

$$\begin{aligned} & \llbracket [m_1(x) = e_1, \dots, m_n(x) = e_n] \cdot d \cdot \varphi \rrbracket_{d'} \\ & = \\ & \{\text{obj} = \{m_1 = \lambda x. \llbracket e_1 \rrbracket_d, \dots, m_n = \lambda x. \llbracket e_n \rrbracket_d\}, \text{ifc} = \hat{\varphi}\} \end{aligned}$$

\*Skalka and Smith, “Set Types and Applications”, TIP02

## Transforming pop

Method selects are encoded so that access rights are verified in the transformation:

$$\llbracket e_1.m(e_2) \rrbracket_d = \text{let } c_1 = \llbracket e_1 \rrbracket_d \text{ in} \\ c_1.\text{ifc}.d \ni m; \\ (c_1.\text{obj}.m)(\llbracket e_2 \rrbracket_d)$$

**NB:** The pml type system is sufficiently accurate to statically verify set membership checks

## Types for pop

Type systems for pop *easily* developed on the basis of the transformation into pml:

- sound indirect type system immediately obtained as composition of pop-to-pml transformation and pml type system
- a direct system developed on foundation of pml type system
  - direct type safety for pop easily obtained, by proving a simple correspondance between pop and pml type judgements

**NB:** no complicated *subject reduction* proof necessary to prove type safety!

## Direct pop types

We define direct type terms specifically adapted for pop, with object types of the form  $[\tau_1] \cdot \{\tau_2\}$ :

- $\tau_1$  the types of methods
- $\tau_2$  the type of the interface
- direct pop types have an *interpretation* as (are syntactic sugar for) pml types

$o \triangleq [\text{read}() = \dots, \text{write}(x) = \dots] \cdot d \cdot \{d \mapsto \{\text{read}, \text{write}\}, d' \mapsto \{\text{read}\}\}$

$o : [\text{read} : \text{unit} \rightarrow \tau, \text{write} : \tau \rightarrow \text{unit}] \cdot \{d : \{\text{read}, \text{write}\}, d' : \{\text{read}\}\}$

$o.\text{write}(v) : \text{unit}$       if  $d'$  is current (static) domain

$o.\text{write}(v)$                       *not well-typed* otherwise

## Using pop

The pop system is sufficiently flexible to model a number of confinement mechanisms with strengthened security.

Notably, pop can encode class definitions with strengthened `private` modifiers; recall:

```
private Identity[] signers

public Identity[] getSigners( ){
    return signers;
}
```

## Using pop

The essential problem is expressed in the following code:

```
class  $c_1$  {  
    public :  
         $m(x) = x$   
}  
  
class  $c_2$  {  
    public :  
         $m() = a$   
    private :  
         $a = \text{new } c_1$   
}
```

We can model objects in class  $c_1$  as:

$$o_1 \triangleq [m(x) = x] \cdot c_1 \cdot \{\partial \mapsto \{m\}\}$$

The class  $c_1$  itself can be modeled as an *object factory*:

$$\text{fctry}_{c_1} \triangleq [\text{new}() = o_1] \cdot d \cdot \{\partial \mapsto \{\text{new}\}\}$$

## Using pop

Objects in class  $c_2$ , and the class itself, can be encoded as follows:

$$o_2 \triangleq \text{let } a = \text{ref } (\text{fctry}_{c_1}.\text{new}() \uparrow (\partial, \emptyset)) \text{ in} \\ [m() = !a] \cdot c_2 \cdot \{\partial \mapsto \{m\}\}$$

- casts ensure that objects stored in `private` instance variables are *unuseable* outside scope of the object
- any *leaked* reference is a *useless* reference

The class  $c_2$  is encoded similarly to the encoding of  $c_1$ , as an object factory