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; for example, in Java*:

```
private Identity[] signers

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

This reference leak circumvents JDK1.2 security mechanism!

due to Princeton Secure Internet Programming Group

Object confinement: strategies

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

- Previous *type-based* 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:

- The *active* region of code is associated with a *current domain*
- 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:

$$[\mathrm{read}() = \ldots, \mathrm{write}(x) = \ldots]$$

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

$$[\mathrm{read}() = \ldots, \mathrm{write}(x) = \ldots] \cdot d$$

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* φ, which specify the per-domain access rights to the object:

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

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

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

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

pop **examples**

Assume the following definition:

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

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

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

Let d be the current domain:

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

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

For example, letting:

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

The following casts have the described results:

- $o \restriction (d, \{\mathrm{read}\})$ yields a read-only file object
- $o \restriction (\partial, \{\varnothing\})$ 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 pop obtained by transformation $(\!| e |\!)$:

- $(\!| e |\!)$ 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 $(\!| e |\!)$ safely evaluates to $(\!| v |\!)$. If e has runtime errors, then so does $(\!| e |\!)$. If e diverges, then $(\!| e |\!)$ diverges.

Transformational Approach

Correctness of term transformation $(\!| e |\!)$ 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 $(\!| e |\!) : \tau$ then $e : \tau$
- Since $(\!| e |\!) : \tau$ 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: pml

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

- *Row types* precisely describe the contents of identifier sets:

  $$\{m_1, \ldots, m_n\} : \{m_1+, \cdots, m_n+, \varnothing\}$$

  and membership check operations:

  $$\ni m \quad : \quad \forall\beta.\{b+, \beta\} \to \{b+, \beta\}$$

- *Conditional constraints* are used to accurately describe the results of other set operations, i.e. intersection, union, difference

**Skalka and Smith, "Set Types and Applications", TIP02*

Transforming pop: pml

For example, the type of the intersection operation $\wedge$ is:

$$\begin{array}{rcl} \wedge & : & \forall \beta_1\beta_2\beta_3[C].\{\beta_1\} \to \{\beta_2\} \to \{\beta_3\} \\ & & \text{where } C = \quad \mathrm{if}\ - \leq \beta_1\ \mathrm{then}\ \varnothing \leq \beta_3 \\ & & \qquad\qquad \wedge\ \mathrm{if}\ + \leq \beta_1\ \mathrm{then}\ \beta_2 \leq \beta_3 \end{array}$$

The pml type system comes equipped with:

- Type safety result
- Efficient type inference algorithm*

**Pottier, "A Versatile Constraint-Based Type Inference System"*

The pop-to-pml transformation (highlights)

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

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

A simplified definition of object transformation is as follows:

$$\begin{gathered} [\![[m_1(x) = e_1, \ldots, m_n(x) = e_n] \cdot d \cdot \varphi]\!]_{d'} \\ = \\ \{\mathrm{obj} = \{m_1 = \lambda x.[\![e_1]\!]_d, \ldots, m_n = \lambda x.[\![e_n]\!]_d\},\ \mathrm{ifc} = \hat{\varphi}\} \end{gathered}$$

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

$$\begin{aligned} [\![e_1.m(e_2)]\!]_d \quad = \quad & \text{let}\, c_1 = [\![e_1]\!]_d \,\text{in} \\ & c_1.\mathrm{ifc}.d \ni m; \\ & (c_1.\mathrm{obj}.m)([\![e_2]\!]_d) \end{aligned}$$

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 $[\tau_1] \cdot \{\tau_2\}$:

- τ_1 the types of methods
- τ_2 the type of the interface
- Direct pop types have an *interpretation* as (are syntactic sugar for) pml types

$$
\begin{array}{lcl}
o & \triangleq & [\mathrm{read}() = \ldots, \mathrm{write}(x) = \ldots] \cdot d \cdot \{d \mapsto \{\mathrm{read}, \mathrm{write}\}, \partial \mapsto \{\mathrm{read}\}\} \\
o & : & [\mathrm{read} : \mathrm{unit} \to \tau, \mathrm{write} : \tau \to \mathrm{unit}] \cdot \{d : \{\mathrm{read}, \mathrm{write}\}, \partial : \{\mathrm{read}\}\}
\end{array}
$$

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

$o.\mathrm{write}(v)$ *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 via the following package:

```
class c1 {
  public:
    m(x)= x;
}
```

```
class c2 {
  public:
    m( )= a
  private:
    a = new c1
 }
```

We can model objects in class `c1` as:

$$o_1 \triangleq [m(x) = x] \cdot c_1 \cdot \{c_2 \mapsto \{m\}, \partial \mapsto \{m\}\}$$

The class `c1` itself can be modeled as an *object factory*:

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

Using pop

Note that proper casting makes these objects *useless* outside c_2:

$$(\mathrm{fctry}_{c_1}.\mathrm{new}() \mid (\partial, \varnothing)) \;\longrightarrow\; ([m(x) = x] \cdot c_1 \cdot \{c_2 \mapsto \{m\}\,, \partial \mapsto \varnothing\})$$

Objects in class `c2` can thus be encoded as follows:

$$\begin{aligned} o_2 \quad \triangleq \quad & \text{let}\, a = \mathrm{ref}\ (\mathrm{fctry}_{c_1}.\mathrm{new}() \mid (\partial, \varnothing))\ \text{in} \\ & \quad [m() = !a] \cdot c_2 \cdot \{\partial \mapsto \{m\}\} \end{aligned}$$

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

Conclusion

Major points:

- The pop *language*, containing features for modeling object confinement mechanisms
- A *use-based* approach allowing a more fine-grained specification of confinement properties
- A *type system* for pop, enhancing security and performance of the language
  - Developed via *transformational approach*

Conclusion: future work

Future work:

- More realistic OO language model: *inheritance*
  - How are interfaces inherited?
- Dealing with garbage collection of useless objects
- Empirical comparison of use- and communication-based approaches
  - Implementation issues? Suitability for patterns of use?

`http://www.cs.jhu.edu/~ces/work.html`