Encapsulating Objects with Confined Types

Kevin O’Neill
CS 711
September 15, 2003

A Security Breach in Java

- Suppose we have a Java package that implements a security architecture.
- Each object of class `Class` has a list of signers, principals under whose authority the class acts.
- Consider two methods:
  - One returns an array of principals in the system.
  - Another allows a class to get the list of principals that signed it.

```java
public class Class {
    private Identity[] signers;
    public Identity[] getSigners() {
        return signers;
    }
}
```

- Oops! The method returns a reference to the system’s internal array!
- Now a caller armed with such a reference, as well as the list of principals, can get whatever access rights they want
- This was a problem in JDK 1.1.1

An Ad-hoc Fix:

```java
public class Class {
    private Identity[] signers;
    public Identity[] getSigners() {
        Identity[] pub = new Identity[signers.length];
        for (int i=0; i<signers.length; i++)
            pub[i] = signers[i];
        return pub;
    }
}
```

Is this fix good enough?

- The better `getSigners()` fixes this particular example, but what’s to stop it from happening again?
- No standard mechanism seems to apply:
  - Type abstraction isn’t relevant.
  - Restricting use of `Identity` objects doesn’t help; an attacker only needs references to them.
  - Information flow isn’t relevant.
  - We can’t do dynamic checks of every array update in Java!

Confined Types

- Could we ensure that references to `Identity` objects can’t leak outside of some protection domain?
  - In particular, the package that the class belongs to?
- Imagine two types:
  - `SecureIdentity`, which cannot be leaked
  - `Identity`, a clone for external use only
    - not a subclass or superclass, so there’s no confusion
  - Can we check that `SecureIdentity` is confined to its package?
The General Problem

- Unlimited sharing of object references can lead to problems.
  - If an object doesn’t know who might have references, every method might be called by an adversary.
- Security checks are a problem:
  - Explicit security checks are tedious, but automatic ones are very slow.
- Class restrictions don’t help:
  - We could cast an object to Object and ship it out on the sly!

One Solution: Confined Types

- Introduced by Bokowski & Vitek, 1999
  - “A machine-checkable programming discipline that prevents leaks of sensitive object references.”
- Confined types do not require a change to the language:
  - They enforce “static scoping of dynamic object references”.
- CoffeeStrainer checks code at compile-time, and checked code goes straight to a standard Java compiler.
  - So no extra runtime overhead.

The Big Picture

It’s kind of like information flow, except that the only “flow” we’re concerned about is references to confined objects.

How to check confinement?

- Prevent all inappropriate reference transfers:
  - Don’t let “this” ever leak out of the package.
- Be really uptight about inheritance:
  - Prevent “widening”, or casts from a confined type to an unconfined type.
  - Use anonymous methods to ease restrictions on inheritance.

Bad reference transfers

```java
package inside;
public class C extends outside.B {
    void putReferences() {
        C c = new C();
        outside.B.c1 = c;
        outside.B.storeReference(c);
        outside.B.c3s = new C[] {c};
        badParentMethod(); // stores “this”
        badSubclassMethod();
        throw new BadException();
    }
    static C f = new C();
    static void C m() { return new C(); }
}
```

Widening

- “Bad widening” occurs when a reference to a confined type is widening to an unconfined supertype.
- Examples:
  - Assignments where the LHS is a supertype of the assigned expression
    - [Doesn’t this require an explicit cast?]
  - A method call where the declared parameter is a supertype
  - A return statement where the declared result type is a supertype
  - A cast expression:
    - Object o = (Object) myConfinedObject;
Hidden Widening

- Hidden widening may occur if a method inherited from an unconfined superclass is invoked on a confined object.
- But we can’t rule out inheritance completely, obviously.
- So we require that methods invoked on a confined object be either:
  - Defined in a confined class, or
  - Anonymous.

Anonymous Methods

- Do not depend on the identity of the current instance, i.e., based entirely by its arguments and fields.
- Non-native methods that use this only for accessing fields or calling other anonymous methods on itself.
- The definition is recursive:
  - To find anonymous methods, we label non-anonymous methods and iterate until a fixpoint is reached.

Example

```java
class Example {
    int count;
    int anon okMethod( A arg ) {
        alsoOkMethod( arg.foo() );
        return count;
    }
    Example notOkMethod( A arg ) {
        arg.bar( this );
        arg.o = this;
        alsoNotOkMethod( arg );
        if (this == arg) ..
        return this;
    }
}
```

With Anonymous Methods...

- It’s okay to inherit methods from an unconfined superclass, as long as all the methods are anonymous.
- Anonymous methods can’t leak confined object references to the outside.
- Anonymous methods are the norm:
  - E.g., 94% of methods in java.util and 83% in java.awt are anonymous.

Finally, today’s paper

- Encapsulating Objects with Confined Types (Grothoff, Palsberg, Vitek, 2001)
- Extends the original paper by simplifying the confinement rules and doing a constraint-based confinement analysis.
- Checks confinement rules for a large-scale Java benchmark suite.
  - Thesis: All package-scoped classes in Java programs should be confined.

Simpler Confinement Rules

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td>All methods invoked on a confined type must be anonymous.</td>
</tr>
<tr>
<td>C2</td>
<td>A confined type cannot be public.</td>
</tr>
<tr>
<td>C3</td>
<td>A confined type cannot appear in the type of a public (or protected) field or the return type of a public (or protected) method of a non-confined type.</td>
</tr>
<tr>
<td>C4</td>
<td>Subtypes of a confined type must be confined.</td>
</tr>
<tr>
<td>C5</td>
<td>A confined type cannot be widened to a non-confined type.</td>
</tr>
</tbody>
</table>

Figure 4: Confinement rules.
### The Main Simplification

- ALL methods invoked on confined types must be anonymous.
- Is this a reasonable simplification?
- Confined types within a package may want to pass references around...

### Inferring Anonymity and Confinement

- They use a constraint-based analysis.
  - Like for type inference and flow analysis.
- Analysis proceeds in two steps:
  1. Generate a system of constraints from program text.
  2. Solve the constraint system.
- A solution to the constraint system says which methods are anonymous and which classes are confined.

### Constraints

- Constraints are all ground Horn clauses.
- They take the following form:
  
  \[
  A :== \text{not-anon(methodId)} \\
  T :== \text{not-conf(ClassId)} \\
  C:== A \mid T \mid T \Rightarrow A \mid A \Rightarrow A \mid \\
  A \Rightarrow T \mid T \Rightarrow T
  \]

### Solving the Constraint System

- Confinement and anonymity rules are used to generate Horn clauses, based on program text.
- Solving the system to answer queries of the form “not-conf(ClassId)” can be done in linear time.
  - (Presumably in the length of the program text.)
- \text{Kacheck/J} does bytecode analysis to infer confinement for a large Java benchmark suite.

### The Purdue Benchmark Suite

- Includes 33 Java programs and libraries of various size, purpose, and origin.
- 46,165 classes and 1,771 packages.
- Main thesis: package-scoped classes should be confined.

### Results

- Of the package-scoped classes in the PBS, 25% are confined.
- In 6 of the 33 programs, > 40% were confined.
- Manual inspection of code indicates that programming style is essential to confinement.
- Don’t forget: the confinement tests here are fairly conservative because of the simple confinement rules.
### Typical Confinement Violations

<table>
<thead>
<tr>
<th>Violation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Anonymity violations</td>
<td>Methods in AWT library register the current object for notification</td>
</tr>
<tr>
<td>Widening to superclass</td>
<td></td>
</tr>
<tr>
<td>Sloppy access modifiers (public)</td>
<td></td>
</tr>
<tr>
<td>Widening in containers</td>
<td>Vectors and hashtables take arguments of type <code>Object</code></td>
</tr>
<tr>
<td></td>
<td>Java needs parametric polymorphism!</td>
</tr>
<tr>
<td></td>
<td>Adding generics would give 30% confinement, up from 25%</td>
</tr>
</tbody>
</table>

### Thoughts/Summary

- Confinement is an important property for high-security software.
- Kacheck/J infers confinement in a fast and scalable way.
- The errors that confinement prevents are probably too subtle for mainstream software engineering, especially for non-secure applications.