Encapsulating Objects with Confined Types

Christian Grothoff Jens Palsberg Jan Vitek
S^3 Lab, Department of Computer Sciences, Purdue University

April 6, 2001

Abstract

Object-oriented languages provide little support for encapsulating objects; reference semantics lets objects escape their defining scope. The pervasive aliasing that ensues remains a major source of software defects. This paper introduces the `Kacheck/J` tool for inference of object encapsulation properties in Java application code. The primary goal of `Kacheck/J` is to help software engineers build robust systems. For this purpose we emphasize simplicity and scalability. Simplicity is crucial for humans to be able to interpret the tool's output. Scalability is mandatory for dealing with software systems consisting of thousands of classes. The encapsulation properties that are currently supported by `Kacheck/J` are variants of *confined types*. We introduce a classification of confinement properties and of confinement breaches. The tool infers these properties automatically from Java bytecode by constraint-based analysis. The analysis is practical, a corpus of 20 959 classes files (50 MB) can be analyzed in less than 326 seconds.

1 Introduction

Most object-oriented languages adopt reference semantics to allow sharing of objects. Sharing occurs when an object is accessible to different clients; we say that it is aliased if it is accessible from the same client under two names. Sharing is both a powerful tool and a source of subtle program defects. A potential consequence of aliasing is that methods invoked on an object may depend on another in a manner not anticipated by designers of those objects. Thus updates to objects in one sub-system can affect apparently unrelated sub-systems, undermining the reliability of the software.

While object-oriented languages provide linguistic support for protecting access to variables and methods or even entire classes, they fail to provide any systematic way of protecting objects. A program may declare some instance variable as private and at the same time return an alias to its content from a method. In other words, object-oriented languages provide the means of protecting the state of single object, but cannot guarantee the integrity of systems of interacting objects. What is lacking is a notion of an encapsulation boundary that ensures that references to 'protected' objects do not escape.

The goal of this paper is to experiment with pragmatic notions of encapsulation in order to provide software engineers with tools to guide them in the design of robust

systems. To this end, we focus on simple models of encapsulation that can easily be understood by programmers. We deliberately ignore more powerful escape analyses which are sensitive to small source code changes [2, 3, 8] and which furnish results that are difficult to interpret. We have chosen to explore variants of *confined types* [5] as they are both simple and, as demonstrated in this paper, can be checked at little cost. Confined types were designed to prevent objects of these classes to escape from their defining package. The definition of confinement given in [5] relied on declarations presupposing that a program was designed with confinement in mind. In practice, it is more likely that confinement will come as an afterthought, once a software system is stable and ready to be released. Then, developers will need tools to help discover which classes are confined and give hints how to make a number of non-confined classes confined.

A disciplined use of the access control mechanisms of Java can prevent the users of a package from depending on unnecessary implementation details. One such mechanism is package scoped classes which may be accessed only from within their defining package. For example, consider the `PrivateKey` class, this class should probably not be declared public. Suppose for a moment that `PrivateKey` is package-scoped and that it is declared in package `P`. Can code outside package `P` get access to a `PrivateKey` object? Yes! This can happen if code in `P` creates a private key, casts it to some public superclass, and then sends the object out of the package. It is likely that a programmer will consider such a scenario to be the result of a programming error, and a good programmer will be on guard and try to avoid that it can happen. One can view this as an escape issue: can the objects of a non public class escape the enclosing package? If not, then such a class is said to be *encapsulated*.

> **Thesis:** In many cases, a software designer intend a nonpublic class to be encapsulated.

In other words, many classes are the "implementation details" of a package. In this paper we present support for this thesis, in the form of results of a large-scale analysis on existing Java code. We have implement `Kacheck/J`, a tool for inferring different encapsulation properties over Java bytecode files. We have found many non public classes that actually are confined and many more classes that could be made confined with the proper linguistic support or minimal refactoring. `Kacheck/J` provides tool support for re-engineering existing Java code to fit a stronger encapsulation discipline. The contributions of this paper are:

1. a presentation of the `Kacheck/J` confinement inference tool which can be used to guide re-engineering of existing Java code;
2. results of running `Kacheck/J` on existing Java code showing that confined types are all over the place; and
3. classification of different encapsulation properties and a discussion of ways to re-engineer classes to make them confined with little or no run-time overhead.

1.1 Related Work

Bokowski and Vitek [5] introduced the notion of confined types. In their paper, confined types are explicitly declared which means that software must be designed and implemented with confinement in mind. Until now, confined types have not been

widely adopted. Their paper discussed an implementation of a source-level confinement checker based on Bokowski's CoffeaStrainer [4]. We present a tool that can *infer* confinement from existing Java code. In other words, declarations are not needed. Furthermore, we have made several changes to the definition of confined type to accommodate inference. `Kacheck/J` works on Java bytecode and has been designed to be scalable. It shares some of the restrictions of the original confinement checker. Primarily, native methods are not analyzed, we assume that they do not breach encapsulation by accessing raw memory. We assume that packages cannot be extended without triggering a new round of verification. In the original paper a similar assumption held so that confining packages were implicitly sealed (c.f. [13, 15]).

Reference semantics permeate object-oriented programming languages, it is thus not surprising that the issue of controlling aliasing has been the focus of numerous papers in the recent years [11, 10, 7, 1, 14, 9, 12, 6]. We will discuss briefly the most relevant work.

In [14], flexible alias protection is presented as a means control potential aliasing amongst components of an aggregate object (or *owner*). Aliasing mode declarations specify constraints on sharing of references. The mode `rep` protects *representation objects* from exposure. In essence, `rep` objects belong to a single owner object and the model guarantees that all paths that lead to a representation object go through that object's owner. The mode `arg` marks argument objects which do not belong to the current owner, and therefore may be aliased from the outside. Argument objects can have different *roles*, and the model guarantees that an owner cannot introduce aliasing between roles. In [6], Clarke, Potter, and Noble formalize representation containment by means of ownership types.

Hogg's Islands [10] and Almeida's Balloons [1] have similar aims. An Island or Balloon is an owner object that protects its internal representation from aliasing. The main difference from [14] is that both proposals strive for full encapsulation, that is, all objects reachable from an owner are protected from aliasing. This is equivalent to declaring everything inside an Island or Balloon as `rep`. This is restrictive, since it prevents many common programming styles: it is not possible to mix protected and unprotected objects as done with flexible alias protection and confined types. Hogg's proposal extends Smalltalk-80 with sharing annotations but it has neither been implemented nor formally validated. Almeida did implement an abstract interpretation algorithm for deciding whether a class meets his balloon invariants. But his approach requires whole-program analysis.

Finally, Kent and Maung [12] proposed an informal extension of the Eiffel programming language with ownership annotations that are tracked and monitored at run-time.

In the field of static program analysis a number of techniques have been developed. Static analyses such as the ones proposed by Blanchet [2] and others [3, 8] provide much more precise results than our technique, but come at a higher analysis cost, they Aoften require whole program analyses, and are very sensitive to small changes in the source code. More than anything, their results are hard to interpret; knowing that an object escapes is often not enough to have a clue of how to re-engineer the code to avoid such an occurrence.

2 Confined Types and Anonymous Methods

The goal of any notion of confinement is to satisfy the following soundness property for some notion of scope:

Soundness: An object of confined type is encapsulated in its scope.

The first definition of confined types given by Bokowski and Vitek [5] assumed that confined objects are bound to the scope of their defining package. Confinement was enforced by two sets of constraints. The first set of constraints, *confinement constraints*, apply to the confining package and ensure that reference to confined objects cannot leak across package boundaries. The second set of constraints, so called *anonymity constraints*, applies to methods inherited by the confined classes, potentially including library code. These constraints are meant to ensure that methods inherited by a confined class do not leak a reference to `this` (*i.e.*, to a confined object).

2.1 A New Definition of Confinement

In this section, we recall the original definition and provide a variant of the anonymity constraints that is better suited to inference. Compared to the definition by Bokowski and Vitek, our definition allows more types to be confined, and thus it proves more objects to be encapsulated. We begin with anonymity constraints. These constraints apply to methods that are inherited by confined types. Their goal is to prevent a method from leaking a reference to a confined object to outside code through the hidden widening on the `this` reference. A method is said to be anonymous if the method satisfies the following three constraints.

$\mathcal{A}1$	An anonymous method cannot assign `this` to a field, use it as an argument or as a return value.
$\mathcal{A}2$	An anonymous method cannot be `native`.
$\mathcal{A}3$	A method which is anonymous in relation to a class `C`, can only use `this` to call methods m which are also anonymous in relation to `C`. The target method m is determined under the assertion that `this` is of type `C`.

Figure 1: Anonymity rules.

Confinement constraints apply to the code of a confining package and they prevent all forms of widening from a confined type to an unconfined type.

2.2 Better Anonymity rules

We now explain how the new definition of confinement differs from the one Bokowski and Vitek.

The most interesting confinement breach is hidden widening of confined types to public types that can occur with inherited methods (rule $\mathcal{C}1$). A simple example is given in Figure 3.

$\mathcal{C}1$	Only methods which are anonymous in relation to a confined type `C` or which are defined in a confined class can be invoked on `C`.
$\mathcal{C}2$	A confined class cannot extend `Thread` or `Throwable`.
$\mathcal{C}3$	A confined type cannot be a public class or interface.
$\mathcal{C}4$	A confined type cannot be the type of a public (or protected) field or the return type of a public (or protected) method.
$\mathcal{C}5$	Subtypes of a confined types must be confined.
$\mathcal{C}6$	A confined type cannot be widened to a non-confined type.

Figure 2: Confinement rules.

```
class Parent {
   Parent nonAnonymousMethod() {
       return this;
   }
}
class NotConfined extends Parent {
    Parent violation() {
       return nonAnonymousMethod();     // hidden widening
    }
}
```

Figure 3: Breaching confinement with a non anonymous method.

In the example a local analysis of the methods does not reveal that `NotConfined` is cast to `Parent`. To avoid having to analyze library code all over again for each confining package, Bokowski and Vitek relied on anonymity declarations. To ensure that library code need to be checked only once, they included an additional constraint:

$\mathcal{A}4$	Anonymity declarations must be preserved when overriding methods.

Thus once a method is declared anonymous, all redefinitions of that method in subclasses have to abide by the constraints.

When **inferring** anonymity, the rule $\mathcal{A}4$ can be improved upon significantly. It is not necessary to require that anonymity of a method be preserved in all subclasses. It is sufficient to require all methods that are invoked by an anonymous method to be anonymous (rule $\mathcal{C}1$). Discovering anonymous is a simple control-flow problem in which the dynamic type of `this` will be used to resolve the target of virtual method dispatch. The problem is simplified as there is no need to track messages send to any other receiver than `this`.

By relating methods with the dynamic type of the `this` reference we arrive at a definition of *anonymous methods in relation to* a class. As the type `this` is exactly known, the rule $\mathcal{A}4$ is no longer required, the rule $\mathcal{C}1$ already takes care of all relevant cases.

Figure 4 shows a confined class `C` that extends a class `A` that has a method `m()`. The method `A.m()` meets all anonymity criteria except for rule $\mathcal{A}4$. The violation of that rule occurs in class `B`, this class extends `A` and redefines `m` with an implementation that returns `this`. The key point to notice here is that the anonymity violation can never occur when the dynamic type of `this` is `A`. We say the method `P.m()` is anonymous *in relation* to `C`, but not in relation to `B`.

```
public class A {           // A is not confined
    Object m() {           // m() is anon in relation to C
        return null;       //     but not in relation to B
    }
    public Object n() {    // n() is anon in relation to B and C
        return new C().m();
    }
}
class B extends A {        // B is not confined
    Object m() {           // m() is not anon
        return this;
    }
}

class C extends A { }      // C is confined
```

Figure 4: Anonymity does not have to be preserved in all subtypes.

Figure 5 shows that this definition is still too strict. If the non-anonymous method is defined in the confined class itself, the `this` reference may be given away, as the static type of `this` is confined. Widening rules will prevent the reference from escaping. In the example, method `C.m()` is not anonymous by the old definition as it gives away the `this` reference. But as in this case exposing the `this` reference can breach encapsulation as the $\mathcal{C}$ rules ensure that `C` does not escape the package.

Thus we can improve $\mathcal{A}1$ as follows:

$\mathcal{A}1$'	A method that is anonymous in relation to a class `C` must either be declared in class `C` or it can neither assign `this` to a field, nor use it as an argument nor as a return value.

The new rules are sufficient as they only exclude cases that either can never occur in the control flow or where hidden widening cannot occur.

This revised definition improves the analysis significantly, allowing to infer a lot more classes as confined then it was possible with the declarations used by Bokowski and Vitek.

```
public class A {       // A is not confined
    public void n() {  // n() is anon in relation to C
        new C().m();
    }
}
class C extends A {    // C is confined
    C m() {            // m() is not anon
        return this;
    }
}
```

Figure 5: Example where this is given away in the confined class.

3 Constraint-Based Analysis

We use a constraint-based program analysis to determine which methods are anonymous and which types are confined. Constraint-based analyzes have previously been used for a wide variety of purposes, including type inference and flow analysis. The idea of constraint-based analysis is to do the analysis in two steps:

1. generate a system of constraints from the program text; and
2. solve the constraint system.

The solution to the constraint system is the desired information. For our application, constraints are of the following forms:

$$\begin{array}{lcl} A & ::= & \mathsf{not\text{-}anon}(\text{methodId}, \text{classId}) \\ T & ::= & \mathsf{not\text{-}conf}(\text{classId}) \\ C & ::= & A \mid T \mid A \Rightarrow A \mid A \Rightarrow T \mid T \Rightarrow T \end{array}$$

A constraint $\mathsf{not\text{-}anon}(\text{methodId}, \text{classId})$ denotes that the method methodId is *not* anonymous in relation to the class classId. A constraint $\mathsf{not\text{-}conf}(\text{classId})$ denotes that the class classId is *not* confined. The remaining three forms of constraints denote logical implications.

We generate constraints from the program text in a straightforward manner. The example program in Figure 6 illustrates all parts of the syntax from which constraints are generated. The numbers given in the comments in Figure 6 and in the following table refer to the rules in section 2. From the program in Figure 6, we generate the following constraints:

case	constraint	explanation
($\mathcal{A}1$)	$\mathsf{not\text{-}anon}(\texttt{A.m()}, *)$	illegal use of `this`
($\mathcal{A}2$)	$\mathsf{not\text{-}anon}(\texttt{A.o()}, *)$	`o` is `native`
($\mathcal{A}3$)	$\mathsf{not\text{-}anon}(\texttt{A.m()}, \texttt{B}) \Rightarrow \mathsf{not\text{-}anon}(\texttt{B.p()}, \texttt{B})$ $\mathsf{not\text{-}anon}(\texttt{A.m()}, \texttt{E}) \Rightarrow \mathsf{not\text{-}anon}(\texttt{B.p()}, \texttt{E})$	`p` calls `m` with `this` being either a `B`-object or an `E`-object
($\mathcal{C}1$)	$\mathsf{not\text{-}anon}(\texttt{E.p()}, \texttt{E}) \Rightarrow \mathsf{not\text{-}conf}(\texttt{E})$	`p` invoked on a `E`-object
($\mathcal{C}2$)	$\mathsf{not\text{-}conf}(\texttt{C})$	class `C` extends `Thread`
($\mathcal{C}3$)	$\mathsf{not\text{-}conf}(\texttt{D})$	class `D` declared to be public
($\mathcal{C}4$)	$\mathsf{not\text{-}conf}(\texttt{E})$	public method `getE` has return type `E`, public field `c` has type `C`
($\mathcal{C}5$)	$\mathsf{not\text{-}conf}(\texttt{E}) \Rightarrow \mathsf{not\text{-}conf}(\texttt{B})$	`E` extends `B`
($\mathcal{C}6$)	$\mathsf{not\text{-}conf}(\texttt{A}) \Rightarrow \mathsf{not\text{-}conf}(\texttt{E})$	`E` widened to `A`

In some of the constraints, we use the abbreviation $\mathsf{not\text{-}anon}(\texttt{A.m()}, *)$ to denote the set of constraints $\mathsf{not\text{-}anon}(\texttt{A.m()}, \texttt{X})$ for all classes X in the program.

All our constraints are ground Horn clauses. Our solution procedure computes the set of clauses $\mathsf{not\text{-}conf}(\texttt{classId})$ that are either immediate facts or derivable via logical implication. This computation can be done easily, in linear time.

```
public class A {
        A a;
        public A m() {
             a = this;                    //  (A1)
             new B().t(this);             //  (A1)
             return this;                 //  (A1)
        }
        native void o();                  //  (A2)
}
class B extends A {
        void t(A a) {}
        A p() {
             return this.m();             //  (A3)
        }
        public A getE() {
             return new E().p();          //  (C1)
        }
}
class C extends Thread {                  //  (C2)
}
public class D {                          //  (C3)
        public E getE() {                 //  (C4)
             return new E();
        }
        public C c = new C();             //  (C4)
}
class E extends B {                       //  (C5)
        A getA() {
             this.t(this);                //  (C6)
             a = new E();                 //  (C6)
             return new E();              //  (C6)
        }
}
```

Figure 6: Example program.

4 Implementation

Though the criteria for confined types have been described on the source level, our analysis is performed on Java bytecode. The code extends a bytecode verifier written for the OVM project, a GPLed implementation of a Java Virtual Machine in Java.

First `Kacheck/J` loads all classes (libraries and program) using the OVM loading mechanism. Then `Kacheck/J` runs a modified version of OVM's bytecode verifier to create the constraint system.

The verification process in OVM can be described with the flyweight pattern. For each of the 200 bytecode instructions defined in the Java Virtual Machine Specification the verifier creates an `Instruction` object that is responsible to compute the actions of this instruction. Starting with an initial state of the StackFrame (which includes the instruction pointer, stack height and types of the local variables) the verifier follows all possible control flows. The `Instruction` objects operate on a StackFrame yielding all next possible states of the interpreter that might follow the execution of that instruction.

This flyweight approach allows us to use the verifier to check for confined types by changing the simulation method of only 9 of the 200 `Instruction` objects. Instead of just computing the next possible StackFrames some extra checks are added. For example the `areturn` instruction now has to check that a `this` Reference is not used as the argument (or report this as a violation of anonymity for that method). The `invoke` instructions record violations like the use of `this` as an argument or dependencies if the method is invoked on `this`.

Overall, the following changes were applied to the verifier:

- in non-static methods the incoming reference to `this` in local variable 0 must be marked special and tracked during the control flow.
- violations and uses of `this` (return, invoke on, argument to) must be recorded during the analysis of the control flow.
- widening of private types (set field, return, use in arguments) must be recorded.
- record if confined types are used with `throw`.

The violations caused by implicit widening can be recorded in the subtyping facility that usually verifies if the assignments in the bytecode are legal. The violations concerned with anonymous methods only require slight modifications to the code that simulates the instructions: a check if the reference used happens to be `this`.

First the constraint solver determines the potentially confined classes using the collected constraints of type $\mathsf{not\text{-}conf}(C)$. For each of the potentially confined classes C it then implements a demand driven computation of the anonymous methods m in relation to C starting with the constraints $\mathsf{not\text{-}anon}(m, C) \Rightarrow \mathsf{not\text{-}conf}(C)$. Finally dependencies between the confined types (constraint $\mathsf{not\text{-}conf}(A) \Rightarrow \mathsf{not\text{-}conf}(B)$) are considered.

The code specific to confined types (including constraint solving) is about 3,000 lines. The code reused from OVM (including class loading) is about 15,000 lines of code. The current version of the OVM is about 44,000 lines of code.

5 Results

We have run `Kacheck/J` on the following 15 benchmarks programs:

JDK 1.1.8	library code
JDK 1.2.2	library code
JDK 1.3	library code
GJ	Generic Java compiler
Soot	Bytecode optimizer framework
Toba	bytecode-to-C translator
Jpat	protein analysis tool
Kawa	Scheme to bytecode compiler
Rhino	Javascript interpreter
Schroeder	Audio editor
Symjpack	Symbolic executor of mathematical expressions
SableCC	Java source to HTML translator
Aglet	Mobile agent development toolkit
HyperJ	IBM composition framework
`Kacheck/J`	confinement checker

Some of our benchmark programs were provided by the Sable Research group and are part of their Ashes Suite. Figure 7 shows general characteristics of the benchmark programs.

Benchmark	classes		packages	size (KB)
	all	public		
JDK 1.1.8	1704	1419	80	8776
JDK 1.2.2	4338	2648	130	10008
JDK 1.3	5438	3327	176	12877
GJ	1638	1130	40	3017
Soot	1442	604	7	2276
Kawa	3160	3155	8	4578
Toba	762	327	12	1378
Rhino	164	122	7	749
Schroeder	120	112	3	299
Javasrc	196	126	14	634
Symjpack	15	2	2	82
SableCC	342	290	9	566
Aglets	410	193	19	1015
HyperJ	921	793	155	2873
`Kacheck/J`	311	113	21	576352

Figure 7: Statistics for the benchmarks

5.1 Confined Types in Practice

The numbers of confined classes found in the benchmark programs are given in Figure 8. The differences between applications are stunning. For example in Soot nearly every third class is confined. On the other hand in Kawa only 2 of 3160 classes are confined. On the average, about 8.6% of all classes are confined and 27.4% of all non-public classes are confined.

Benchmark	confined classes	violations			
		hierarchy	widening	anonimity	member
JDK 1.1.8	37	18	203	41	52
JDK 1.2.2	415	19	1215	125	124
JDK 1.3	507	25	1521	140	171
GJ	93	0	412	5	14
Soot	453	0	382	1	41
Kawa	2	0	3	0	0
Toba	40	0	389	1	83
Rhino	21	0	17	1	3
Schroeder	3	0	5	3	0
Javasrc	14	0	53	3	58
Symjpack	10	0	3	0	12
SableCC	3	0	49	0	2
Aglets	104	4	91	42	4
HyperJ	22	0	97	15	21
`Kacheck/J`	18	0	169	0	18

Figure 8: Number of confined classes and reasons for non-confinement

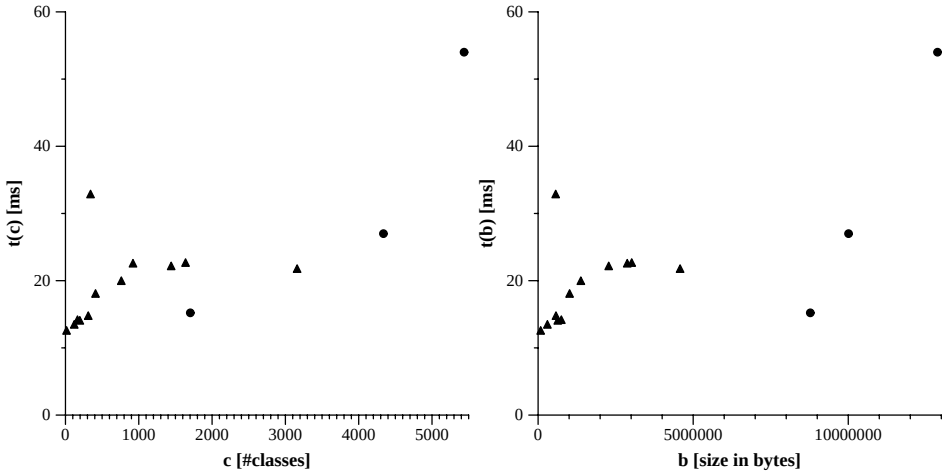
Performance All benchmarks were performed on a Pentium III 800 with 256 MB of RAM running Linux 2.2.18 with IBM JDK 1.3. The CPU timings show best user time over three consecutive runs of the program.

Except for the JDK tests all benchmarks times include loading the JDK 1.2.2 libraries, but `Kacheck/J` analyzes library code only to evaluate method anonymity. In practice only a small part of the library is analyzed.

The following graphs plot the number of classes/bytecode size against analysis time. Circles represent the JDK benchmarks which are special since they do not have load library code. The variability in analysis time is due to `Kacheck/J`'s treatment of bytecode verification. In some pathological cases analysis of a single method can dominate the entire program analysis time.

Benchmark	time (s)	average per class (ms)
JDK 1.1.8	15.2	8.9
JDK 1.2.2	27.0	6.2
JDK 1.3	54.0	9.9
GJ	22.7	13.8
Soot	22.2	15.4
Kawa	21.8	6.9
Toba	20.0	26.2
Rhino	14.2	86.5
Schroeder	13.5	112.5
Javasrc	14.1	71.9
Symjpack	12.6	840.0
SableCC	32.9	96.2
Aglets	18.1	44.1
HyperJ	22.6	24.5
`Kacheck/J`	14.8	47.6

Figure 9: Time required for the analysis



5.2 Confinement Violations

A rough categorization of confinement violations is as follows:

Hierarchy	The class is a subclass of Thread or Throwable.
Widening	The type is widened to a non confined type.
Anonymity	A method invoked on the class is not anonymous.
Member	A non-private field or method return type exposes a confined type

The data summarizing violation in the benchmarks is shown in Figure 8. Looking at the reasons why classes are not confined, it is obvious that most violations are caused by widening to non-confined classes. Invocations of non anonymous methods on a class play only a minor rule in the overall number of violations. Hierarchy violations occur almost exclusively in the JDK (note that in our current implementation classes that extend Throwable are caught during bytecode analysis and are marked as widening violations).

Some very frequent violations are caused by well known programming idioms. We give examples some of the most frequent cases.

Anonymous inner class cast to public type: This violation occurs very frequently when inner classes are used to implement call-backs. For example in the Aglet benchmark `(com/ibm/aglet/util/AddressBook.java)` the `MouseListener` class is public. Thus the following code violates confinement of the inner class.

```
MouseListener mlistener = new MouseAdapter() {
     public void mouseEntered(MouseEvent e) {
          ....
     }
};
```

Anonymity violations: The top three anonymity violations (accounting for 133 non confined classes) in the entire JDK come from methods in the AWT library which register the current object for notification. The method `addImpl` is representative:

```
protected void addImpl(Component comp, Object constraints, int index) {
   synchronized (getTreeLock()) {
    ...
       ContainerEvent e = new ContainerEvent(this,
                     ContainerEvent.COMPONENT_ADDED,
                     comp);
    ...
 }
```

Widening to (abstract) parent: Widening violations are the most frequent kind of confinement breach. For instance the tool signals the following widening in the Aglet benchmark:

```
com/ibm/aglets/tahiti/SecurityPermissionEditor:
     Illegal Widening:
      widened to com/ibm/aglets/tahiti/PermissionEditor
```

`PermissionEditor` is an abstract superclass of `SecurityPermissionEditor`. This part of the interface is then exported outside the package (in this case in a method call, implicit widening of the argument).

Widening to containers: For instance, the tool reports that a key part of the `ClassLoader` class is not confined.

```
java/lang/ClassLoader$NativeLibrary:
    Illegal Widening:
     widened to java/lang/Object
```

The error occurs because an instance of `NativeLibrary` is stored in a vector.

```
systemNativeLibraries.addElement(lib);
```

As such, this violation may indicate a security problem. Quick inspection of the code reveals that the vector in which the object is stored is private.

```
private static Vector systemNativeLibraries = new Vector();
```

After a little more checking it is obvious that the vector does not escape from its defining class. But this required inspection of the source code and remains true only until the next patch is applied to the class. Examples like this one are a good motivation for confined types.

6 Discussion

6.1 Increasing Confinement

The large number of widening violations occurring in the benchmarks suggests that we should focus on this category of error when re-engineering software to increase confinement opportunities. Most of the widening violation we have observed occur because the target of the widening is a public class. These breaches roughly break down in two groups: breaches due to casts to a public super type and the special case of casts to `Object` that occur when confined types are stored in containers.

We introduce the following classification:

Unconf	Classes that cannot be made confined.
Confinable	Classes that could possibly be made confined.
Confined	Classes that satisfy the confinement rules.
GenConfinable	Classes that are not confined only because they are stored in a container.

Unconf classes are classes that cannot possibly be confined because they are actually used outside of their defining package. Confinable classes are class that are not confined because they have been declared public, but otherwise meet all the confinement criteria. GenConfinable classes are confinable classes that are not confined only because they are stored in a container.

Confinable classes are interesting because all it takes to make the classes confined is to remove the public keyword in the class declaration. Figure 6.1 shows that there is a large number of types that may be made confined. The analysis infers confinability by

ignoring class access modifiers and trying to make all classes confined. When ever the tool find a use of a class outside of its defining package the class is marked as Unconf. Of course, making library classes confined may affect future clients, so the numbers shown here should be considered an upper bound.

Benchmark	number of classes		
	all	confined	confinable
JDK 1.1.8	1702	37	488
JDK 1.2.2	4338	415	1015
JDK 1.3	5438	507	1362
GJ	1638	93	409
Soot	1442	453	862
Kawa	3160	2	707
Toba	762	40	153
Rhino	164	21	80
Schroeder	120	3	8
Javasrc	196	11	48
Symjpack	15	10	12
SableCC	342	3	43
Aglets	410	104	152
HyperJ	921	22	176
`Kacheck/J`	311	18	38

Figure 10: Number of confined and confinable classes

Containers in Java are usually defined in the JDK `java.util` package and hold objects of type `Object`. As `Object` is public and the methods of the containers deal with `Object`, widening occurs in every use of these containers. In Figure 6.1 we show what happens if we do not count widening of potentially confined types to public types in calls to methods of `java.util` classes. The column GenConfinable list the number of classes (including public classes) that would be confined if violations with `java.util` are discounted.

The numbers are, again, an upper bound as we do not check that the container in which confined objects do not escape their confining package.

6.2 Coding for Confinement

Our results clearly point to containers is one source of confinement violations. We considered using generic extensions of Java (such as GJ) to increase confinement, unfortunately the homogeneous translation strategies adopted by most of these extensions imply that at the bytecode level, code written with GJ is translated back to code that uses the standard Java container classes. Thus it is not possible for `Kacheck/J` to verify that classes stored in generic types remain confined. Heterogeneous translation strategies have the drawback of causing code duplication. Fortunately, it is possible to achieve the desired result with some coding techniques. The basic idea is to use a

Benchmark	number of classes		
	all	confined	util-confinable
JDK 1.1.8	1702	37	527
JDK 1.2.2	4338	415	1129
JDK 1.3	5438	507	1499
GJ	1638	93	466
Soot	1442	453	862
Kawa	3160	2	707
Toba	762	40	153
Rhino	164	21	84
Schroeder	120	3	18
Javasrc	196	11	60
Symjpack	15	10	12
SableCC	342	3	47
Aglets	410	104	161
HyperJ	921	22	188
`Kacheck/J`	311	18	38

Figure 11: Number of confined and util-confinable classes

adapter pattern to wrap an unconfined object around each confined object that must be stored in a container.

A Hashtable Example: A confined implementation of Hashtable could provide an interface `Entry` with two methods `boolean equal(Entry e)` and `int hashCode()`. In the package that contains the confined class `C` the programmer would define an implementation `RealEntry` of `Entry` with a package-scoped constructor that takes the key and value (where e.g. the value has the type of the confined class) and package-scoped accessor methods. The `Hashtable` itself would only be able to access the `public` methods defined in `Entry`. As long as these method do not reveal the confined class, the confinement of the class can be verified.

The cost of this change would be the creation of the extra `Entry` object that might not be required by other implementations of `Hashtable`. On the other hand, to access a key-value pair this implementation only requires one cast (`Entry` to the `RealEntry` to access key and value, where the default implementation requires a cast on key and value. For other containers the tradeoffs maybe worse.

7 Conclusion

The number of confined types in normal Java code is surprisingly high. Many of violations occur together with the use of `java.util` classes and thus might be resolved extending Java with genericity or by changing the coding style. We proved that inferring confined types is fast, scalable and provides results that are easy to interpret.

References

[1] Paulo Sérgio Almeida. Balloon Types: Controlling sharing of state in data types. In *ECOOP Proceedings*, June 1997.

[2] Bruno Blanchet. Escape analysis for object oriented languages. application to Java. In *OOPSLA'99 ACM Conference on Object-Oriented Systems, Languages and Applications*, volume 34(10) of *ACM SIGPLAN Notices*, pages 20–34, Denver, CO, October 1999. ACM Press.

[3] Jeff Bogda and Urs Hölzle. Removing unnecessary synchronization in Java. In *OOPSLA'99 ACM Conference on Object-Oriented Systems, Languages and Applications*, volume 34(10) of *ACM SIGPLAN Notices*, pages 35–46, Denver, CO, October 1999. ACM Press.

[4] Boris Bokowski. CoffeeStrainer: Statically-checked constraints on the definition and use of types in Java. In *Proceedings of ESEC/FSE'99*, Toulouse, France, September 1999.

[5] Boris Bokowski and Jan Vitek. Confined Types. In *Proceedings 14th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'99)*, Denver, Colorado, USA, November 1999.

[6] David G. Clarke, John M. Potter, and James Noble. Ownership types for flexible alias protection. In *OOPSLA '98 Conference Proceedings*, volume 33(10) of *ACM SIGPLAN Notices*, pages 48–64. ACM, October 1998.

[7] D. Detlefs, K. Rustan, M. Leino, and G. Nelson. Wrestling with rep exposure. Technical report, Digital Equipment Corporation Systems Research Center, 1996.

[8] Alain Deutsch. Semantic models and abstract interpretation techniques for inductive data structures and pointers. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 226–229, La Jolla, California, June 21–23, 1995.

[9] Daniela Genius, Martin Trapp, and Wolf Zimmermann. An approach to improve locality using Sandwich Types. In *Proceedings of the 2nd Types in Compilation workshop*, volume LNCS 1473, Kyoto, Japan, March 1998. Springer Verlag.

[10] John Hogg. Islands: Aliasing protection in object-oriented languages. In *OOPSLA Proceedings*, November 1991.

[11] John Hogg, Doug Lea, Alan Wills, Dennis deChampeaux, and Richard Holt. The Geneva convention on the treatment of object aliasing. *OOPS Messenger*, 3(2), April 1992.

[12] S.J.H. Kent and I. Maung. Encapsulation and Aggregation. In *Proceedings of TOOLS PACIFIC 95 (TOOLS 18)*. Prentice Hall, 1995.

[13] Sun Microsystems. Support for extensions and applications in the version 1.2 of the Java platform. 2000.

[14] James Noble, John Potter, and Jan Vitek. Flexible alias protection. In *Proceedings of ECOOP'98*, volume 1543 of *LNCS*, Brussels, Belgium, July 20 - 24 1998. Springer-Verlag.

[15] Ayal Zaks, Vitaly Feldman, and Nava Aizikowitz. Sealed calls in Java packages. In *OOPSLA '2000 Conference Proceedings*, ACM SIGPLAN Notices. ACM, October 2000.