Encapsulating Objects with Confined Types

Christian Grothoff Jens Palsberg Jan Vitek
S^3 Lab, Department of Computer Sciences, Purdue University
{grothoff,palsberg,jv}@cs.purdue.edu

ABSTRACT
Object-oriented languages provide little support for encapsulating objects. Reference semantics allows objects to escape their defining scope. The pervasive aliasing that ensues remains a major source of software defects. This paper introduces `Kacheck/J` a tool for inferring object encapsulation properties in large Java programs. Our goal is to develop practical tools to assist software engineers, thus we focus on simple and scalable techniques. `Kacheck/J` is able to infer *confinement* for Java classes. A class and its subclasses are confined if all of their instances are encapsulated in their defining package. This simple property can be used to identify accidental leaks of sensitive objects. The analysis is scalable and efficient; `Kacheck/J` is able to infer confinement on a corpus of 46,000 classes (115 MB) in 6 minutes.

1. INTRODUCTION
Object-oriented languages rely on reference semantics to allow sharing of objects. Sharing occurs when an object is accessible to different clients; an object is aliased when it is accessible from the same client through different access paths. 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 each other in a manner not anticipated by designers of those objects, and updates in one sub-system can affect apparently unrelated sub-systems, undermining the reliability of the program.

While object-oriented languages provide linguistic support for protecting access to variables, methods, and even entire classes, they fail to provide any systematic way of protecting objects. A class may well declare some variable private and yet return the contents of that variable from a public method. In other words, object-oriented languages protect the state of individual objects, but cannot guarantee the integrity of systems of interacting objects. They lack a notion of an *encapsulation boundary* that would ensure 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. We deliberately ignore more powerful escape analyses [2, 3, 9] which are sensitive to small source code changes and return results that may be difficult to interpret. Of course, the tradeoff is that our analysis will sometime deem an object as 'escaping' when a more precise analysis would discover that this is not the case.

We have chosen to investigate *confined types* [5] as they give rise to a form of encapsulation that is both simple to understand and that can be checked with little cost. The basic idea underlying confined types is the following:

Objects of a confined type are
encapsulated in their defining package.

Thus, if a class is confined, instances of that class and all of its subclasses cannot be manipulated by code belonging to other packages. In terms of aliasing, confinement allows aliases within a package but prevents them from spreading to other packages as illustrated graphically in Figure 1.

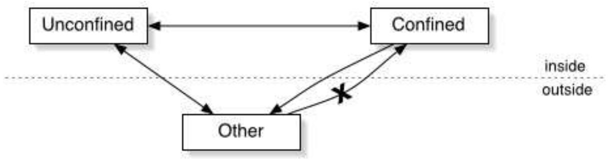

Figure 1: Objects in package `outside` cannot hold references to objects encapsulated in package `inside`.

The definition of confinement in [5] requires explicit annotations and thus pre-supposes that software is designed with confinement in mind. In this work we take a different approach: `Kacheck/J` infers confinement in existing Java packages. We begin with the following controversial thesis:

Thesis: All package-scoped classes in Java programs should be confined.

Furthermore, we show that a majority of large Java applications were written such that confinement would hold for package-scoped classes. In other words, confinement is a natural property to expect of package-scoped classes and one that should be enforced by compilers. To validate our hypothesis we gathered a large number of Java programs (46,000 class files—to the best of our knowledge the largest such benchmark suite) and implemented the Kacheck/J tool to infer confinement properties of these classes. The results of our analysis show that without any change to source programs, 3,998 classes (or 25% of the package-scoped classes) are confined. Furthermore, we found that if one adds features to Java that address the lack of generic container types, then the number of confined types can be increased to over 4,800. Finally, we were surprised to discover that with appropriate tool support, the number of confined classes can be well over 14,500 for that same benchmark suite (or 32% of all classes). Even though we can agree that there are valid uses of package-scoped classes that break confinement, we feel that these uses should be flagged and treated specially rather than the converse.

While more powerful program analysis may yield higher numbers of confined classes, especially if a whole-program approach is taken, our current numbers are already surprisingly high. Another pleasant surprise is that these results can be obtained efficiently. The average time to analyze a class file is less than 8 ms (or about 350 s for the whole benchmark suite) for a tool entirely written in Java running on stock hardware.

The contributions of this paper are:

1. A simpler and less restrictive set of confinement rules than in [5] (Section 3).
2. A constraint-based confinement analysis (Section 4).
3. A presentation of the Kacheck/J confinement inference tool (Section 5).
4. Confinement results for a large-scale benchmark suite (Section 6).
5. A discussion of refactorings aimed at improving confinement as well as proposals for better language support for confinement (Section 6).

2. AN EXAMPLE OF CONFINEMENT

In modern object-oriented programming languages such as Java, confinement can be achieved by a disciplined use of built-in access control mechanisms and some simple coding idioms. We will give a simple motivating example and use it to discuss our analysis.

Consider the class HashtableEntry used within the implementation of Hashtable in the Sun JDK's java.util package. The access modifier for this class is set to default access, which, in Java, means that the class is scoped by its defining package. HashtableEntry instances are used to implement linked lists of elements which have the same hash code modulo table size. They are a prime example of an internal data structure which is only relevant to one particular implementation of a hashtable and that should not escape the context of Hashtable and definitely not of the defining package java.util. Yet how can we be sure that code outside of the package cannot get access to a HashtableEntry object?

Since HashtableEntry is package-scoped we need not worry that outside code will create instances of the class. But in case a public method were to return a HashtableEntry object or a public field held a reference to such an object, outside code would be able to cast that reference to Object and either store it or use it as an argument. The implementation of Hashtable itself could cast a HashtableEntry object to some public superclass, and then expose a reference to the object. It is likely that a programmer would consider such a scenario to be the result of a programming error, and a good programmer would be careful and prevent such confinement breaches. One can view this as an escape problem: can references to instances of a package-scoped class escape their enclosing package? If not, then the objects of such a class are said to be *encapsulated* in the package. In the example at hand, HashtableEntry is indeed encapsulated as programmers have carefully avoided exposing them to code outside of the java.util package.

Kacheck/J discovers potential confinement violations and returns a list of confined types for each package analyzed by the tool. For instance, in the above example, the expected result of the analysis would be that HashtableEntry is confined to the package java.util, while Hashtable is not since it has been declared public. The analysis relies on access modifiers of classes, fields and methods, along with results of a simple intra-procedural analysis of the bytecode of all methods defined in the enclosing package (this part of the analysis performs confinement checks). Furthermore, for package-scoped classes, the code of inherited methods is also analyzed (this part of the analysis performs the so-called anonymity checks). Figure 2 illustrates the checks performed by the tool.

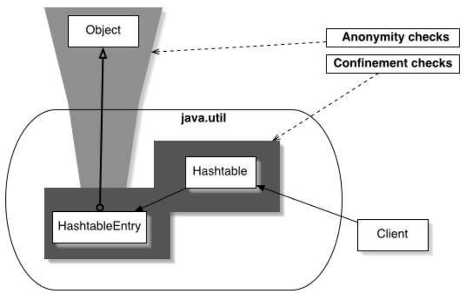

Figure 2: Analysis overview. All classes in the enclosing package, java.util in this case, are checked for confinement. Parent classes of confined classes (e.g. Object) are checked for anonymity. Client code need not be checked.

The analysis is modular since only one package needs to be considered at a time; this turns out to be a key feature for scalability. Furthermore, since client code is not required when checking confinement, it is possible to use Kacheck/J on library code.

In fact, our analysis infers that the class HashtableEntry is not confined because the method clone is invoked on one of its instances. The Hashtable's clone method clones all of the entries in the table. The problem with clone is that it returns a copy of the receiver cast to Object.

Manual inspection of the code reveals that each invocation of `HashtableEntry.clone()` is immediately followed by a cast to `HashtableEntry`. Thus instances of the class do not escape. But our analysis is not precise enough to discover that. A simple and efficient fix is to refactor the code by replacing `HashtableEntry.clone()` with a new method `clone_` that returns a `HashtableEntry`. This refactoring is simple enough and has the advantage of removing unnecessary type casts.

Simplifying Assumptions

`Kacheck/J` operates under some simplifying assumptions which we detail here.

Sealed packages

We assume that all classes of a package are known at analysis time. This assumption is important for the analysis results as a new class may break confinement of pre-existing classes (e.g. creating a `HashtableEntry` and returning it from a public method). In Java, user code may load new classes which declare themselves a member of a package. There are several possible approaches here. For example, packages loaded from a jar file may be declared sealed [14, 16], in which case no user class can be added to that package. Another solution would be to add support for incremental confinement analysis as part of bytecode verification.

Reflection

The analysis assumes that reflection does not violate language access control. In other words, it assumes that the semantics of private, protected and default access modifiers are respected by the reflection mechanisms. This assumption can be violated by changing the settings of the Java Security Manager. This may result in additional confinement breaches.

Native code

Native methods are not checked by `Kacheck/J` and may breach confinement. We assume that native methods defined in the current package do not directly breach confinement, while we make no assumptions about the behavior of native methods defined in other packages. Manual inspection of a large number of native methods indicates that this assumption is reasonable. Furthermore, we assume that native code in other parts of the system does not violate the semantics of the language by ignoring access control declarations.

3. CONFINED TYPES

The goal of confinement is to satisfy the following soundness property:

> **Soundness:** An object of confined type is encapsulated in the scope of that type.

In [5], the granularity of confinement is the package. Thus, no instance of a confined type may escape the package in which that type is defined. We say that instances of a confined class are encapsulated in their defining package.

Confinement is enforced by two sets of constraints. The first set of constraints, *confinement rules*, apply to the enclosing package, the package in which the confined class is defined. These rules track values of confined types and ensure that they are neither exposed in public members, nor widened to non-confined types. The second set of constraints, so-called *anonymity rules*, applies to methods inherited by the confined classes, potentially including library code, and ensures that these methods do not leak a reference to the distinguished variable `this` which may refer to an object of confined type.

In this section, we adapt the rules of Bokowski and Vitek to inference of confinement. The new rules are both simpler and less restrictive (i.e., more classes can be shown confined), while remaining sound. As in the original paper, the rules presented here do not require a closed-world assumption. Confinement inference is performed at the package level. The rules assume that all classes in a package are known and, for confined classes, that their superclasses are available.

3.1 Anonymity Rules

Enforcing confinement relies on tracking the spread of encapsulated objects within a package and preventing them from crossing package boundaries. We have chosen to track encapsulated objects via their type. Thus, a confinement breach will occur as soon as a value of a confined type can escape its package. Since we track types, widening a value from a confined type to a non-confined type is a violation of the confinement property.

Anonymity rules apply to inherited methods which may (but do not have to) reside in classes outside of the enclosing package. The goal of this set of rules is to prevent a method from leaking a reference to the distinguished `this` pointer. The motivation for these rules is that if `this` refers to an encapsulated object, returning or storing it amounts to hidden widening. Thus, we say that a method is *anonymous* if the following three rules hold.

$\mathcal{A}1$	An anonymous method cannot widen `this` to a non-confined type.
$\mathcal{A}2$	An anonymous method cannot be `native`.
$\mathcal{A}3$	Methods invoked on `this` must be anonymous.

Figure 3: Anonymity rules.

The first rule prevents an inherited method from storing or returning `this` unless the static type of `this` also happens to be confined. The second rule ensures that `native` methods are never anonymous. While rules $\mathcal{A}1$ and $\mathcal{A}2$ are direct anonymity violations, the rule $\mathcal{A}3$ tracks transitive violations. The call mentioned in rule $\mathcal{A}3$ depends on the dynamic type of `this` (the target of the call). Thus, anonymity of methods is determined in relation to a specific type.

3.2 Confinement Rules

Confinement rules are applied to all classes of a package. A class is *confined* if it satisfies the five rules of Figure 4.

$\mathcal{C}1$	All methods invoked on a confined type must be anonymous.
$\mathcal{C}2$	A confined type cannot be public.
$\mathcal{C}3$	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.
$\mathcal{C}4$	Subtypes of a confined type must be confined.
$\mathcal{C}5$	A confined type cannot be widened to a non-confined type.

Figure 4: Confinement rules.

Rule $\mathcal{C}1$ ensures that no inherited method invoked on a confined type will leak the `this` pointer. This rule does not preclude a confined type from *inheriting* non-anonymous methods, as long as they are never called. Rule $\mathcal{C}2$ prevents public classes from being confined. Rule $\mathcal{C}3$ ensures that no exposed member (private or protected) is of a confined type. This applies to all non-confined types in the package. Rule $\mathcal{C}4$ prevents non-confined classes (or interfaces) from extending confined types. Finally, rule $\mathcal{C}5$ prevents values of confined type from being cast to non-confined types.

Exceptions are a case of widening which is not explicitly listed in these rules. Instead, we consider that `throw` widens its argument to the class `Throwable`, which is declared public and thus violates rule $\mathcal{C}5$.

Our confinement rules do not forbid packages from having native code, but rule $\mathcal{A}2$ explicitly states that native methods are not anonymous. The motivation for this design choice is that while the developer of a package may be expected to manually inspect native code in the current package, it would be difficult to check native code of parent classes belonging to standard libraries. Furthermore, uses of `this` that violate $\mathcal{A}1$ are usually not perceived as bad behavior for native code. Essentially, we assume that native code within the enclosing package is, to some extent, trusted.

4. CONSTRAINT-BASED ANALYSIS

We use a constraint-based program analysis to infer method anonymity and confinement. Constraint-based analyses have previously been used for a wide variety of purposes, including type inference and flow analysis. Constraint-based analysis proceeds in two steps:

1. Generate a system of constraints from program text.
2. Solve the constraint system.

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

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

A constraint not-anon(methodId) asserts that the method methodId is *not* anonymous; similarly, not-conf(classId) asserts that the class classId is *not* confined. The remaining four forms of constraints denote logical implications. For example, not-anon(`A.m()`) ⇒ not-conf(`C`) is read "if method `m` in class `A` is not anonymous then class `C` will not be confined."

We generate constraints from the program text in a straightforward manner. The example of Figure 5 illustrates the generation of constraints. For each syntactic construct, we have indicated in comments the associated rule from Section 3. Figure 6 details the constraints that are generated for that example. A complete description of the constraints generated from Java bytecode is given in Appendix A.

```
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 getD() {
            return new D().p(); //  (C1)
        }
}
public class C {                //  (C2)
        public D getD() {       //  (C3)
            return new D();
        }
        public D d = new D();   //  (C3)
}
class D extends B {             //  (C4)
        A getA() {
            this.t(this);       //  (C5)
            a = new D();        //  (C5)
            return new D();     //  (C5)
        }
}
```

Figure 5: Example program.

All our constraints are ground Horn clauses. Our solution procedure computes the set of clauses not-conf(`classId`) that are either immediate facts or derivable via logical implication. This computation can be done in linear time.

Control Flow Analysis

The rule $\mathcal{C}1$ poses a control flow problem as it mandates that only methods that are actually invoked on a confined type need to be anonymous. Any conservative control flow analysis can be used to yield a set of candidate methods. We have chosen to perform a simple flow insensitive analysis that is practical and precise enough for our purposes.

Case	Constraint	Explanation
$(\mathcal{A}1)$	not-conf(`A`) $\Rightarrow$ not-anon(`A.m()`)	`this` widened to `A`
$(\mathcal{A}2)$	not-anon(`A.o()`)	`o` is `native`
$(\mathcal{A}3)$	not-anon(`A.m()`) $\Rightarrow$ not-anon(`B.p()`)	`B.p()` calls `m()` with `this` being the receiver object
$(\mathcal{C}1)$	not-anon(`D.p()`) $\Rightarrow$ not-conf(`D`)	`p()` invoked on a `D`-object
$(\mathcal{C}2)$	not-conf(`C`)	class `C` declared to be public
$(\mathcal{C}3)$	not-conf(`C`) $\Rightarrow$ not-conf(`D`)	public method `C.getD()` has return type `D`; public field `C.d` has type `D`
$(\mathcal{C}4)$	not-conf(`D`) $\Rightarrow$ not-conf(`B`)	`D` extends `B`
$(\mathcal{C}5)$	not-conf(`A`) $\Rightarrow$ not-conf(`D`)	`D` widened to `A`

Figure 6: The constraints generated from the example in Figure 5.

Since, by definition, confined types cannot be invoked from outside of their defining package and cannot be widened to non-confined types, the analysis only needs to record methods invoked on instances of a confined type. Thus, only invocations of the type `x.m()`, where the type of `x` is confined, need to be retained. This forms the root set for the control flow analysis. Transitive calls from within a confined method in this root set (e.g. `this.m()`) are recorded by anonymity rule $\mathcal{A}3$. The type of `x` in `x.m()` is determined as the union of the most general type inferred during bytecode verification with all subtypes of that type that are ever widened to it.

The analysis does not attempt to perform dead-code detection, so while the method that includes an invocation such as `a.m()` may be dead, we will nevertheless add `m` to the root set. This simplifies the analysis but costs some precision. Doing dead code detection would also lead to analysis results that are much more sensitive to changes in the source program. We strongly believe that the results of confinement inference should be stable in the face of trivial changes to the source program and that any changes should have only local effects.

5. IMPLEMENTING KACHECK/J

Although the confinement and anonymity rules have been described as source level constraints, we have chosen to implement `Kacheck/J` as a bytecode analyzer. The main advantage of working at the bytecode level is the large number of class files freely available. The implementation of `Kacheck/J` leverages the Open Virtual Machine project's bytecode verification framework.

In OVM, bytecode verification has been implemented using the flyweight pattern. For each of the 200 bytecode instructions defined in the Java Virtual Machine Specification, the OVM verifier creates an `Instruction` object that is responsible for computing the effect this instruction will have on an abstract state. Verification is a simple fixed-point iteration. The verification starts with an initial state which includes the instruction pointer, operand stack and variables. The verifier follows all possible control flows within the method.

This flyweight approach allows us to use the OVM bytecode verifier as a static analysis engine. We generate constraints by subclassing only 9 of the 200 `Instruction` objects. These special purpose instructions perform some simple checks and record basic facts about the program execution. For instance, the `areturn` instruction checks if `this` is used as return value, and if so, it reports that `this` is widened to the return type of the method. The `invoke` instructions record dependencies like the use of `this` as an argument or when a method is invoked on `this`.

Overall, the following changes were applied to the verifier:

- In non-static methods, local variable 0 (`this`) is tracked.
- Uses of `this` are recorded.
- All widenings are recorded.
- Types of thrown exceptions are recorded.

Widenings are captured by intercepting subtype checks. Anonymity checks only require slight modifications to the code that simulates the nine instructions: a check is added to record operations on `this`. See the Appendix A for details.

The flow analysis computes the implication chains for each potentially confined type T_1, such that

$$T_2 \Rightarrow (A \Rightarrow)^* A \Rightarrow T_1$$

is collapsed to

$$T_2 \Rightarrow T_1.$$

The constraints of the form T and $T \Rightarrow T$ are solved immediately while they are recorded.

The code specific to confined types (including verbose reporting of violations) is about 5,600 lines. The code reused from OVM (including class loading) is about 25,000 lines of code. The current version of the OVM is about 74,000 lines of code.

Example

Figure 7 gives an example of a chain of constraints that results in classes being not confined. Mind that the tool reorders parts of the solving process, while here only the final chain of constraints is explained.

The method `P.nonAnon()` is not anonymous because it widens `this` to `java.lang.Object`, which is a non-confined class (because it is public). This will generate a constraint of type $C \Rightarrow A$:

$$\textsf{not-conf}(\texttt{Object}) \Rightarrow \textsf{not-anon}(\texttt{P.nonAnon()})$$

The invocation of `nonAnon` in `nonAnonInd` with `this` as the receiver generates a constraint of the type $A \Rightarrow A$:

$$\begin{aligned} & \textsf{not-anon}(\texttt{P.nonAnon()}) \\ \Rightarrow \quad & \textsf{not-anon}(\texttt{B.nonAnonInd()}) \end{aligned}$$

The method `nonAnonInd`() is invoked on `C`. By rule $\mathcal{C}1$ a constraint of the type $A \Rightarrow C$ is generated:

$$\textsf{not-anon}(\texttt{B.nonAnonInd()}) \Rightarrow \textsf{not-conf}(\texttt{C})$$

As `C extends B`, a constraint of the type $C \Rightarrow C$ is generated by rule $\mathcal{C}4$:

$$\textsf{not-conf}(\texttt{C}) \Rightarrow \textsf{not-conf}(\texttt{B})$$

Solving this constraint system will result in `B` and `C` being non-confined (and `P` and `X` cannot be confined either because they are `public`).

```
public class P {
    public Object nonAnon() {
         return this;                 // (1)
    }
}
class B extends P {
    public Object nonAnonInd() {
        return this.nonAnon();        // (2)
    }
}
class C extends B {                   // (3)
}
public class X {
    public Object invocation() {
         return new C().nonAnonInd(); // (4)
    }
}
```

Figure 7: Sample constraint chain.

6. RESULTS

`Kacheck/J` has been evaluated on a large data set. This section gives an overview of the benchmark programs and presents the results of the analysis. We also discuss extensions of `Kacheck/J`, coding idioms for confinement and improved language support.

6.1 The Purdue Benchmark Suite

The Purdue Benchmark Suite (PBS) consists of 33 Java programs and libraries of varying size, purpose and origin. The entire suite contains 46,165 classes (or 115 MB of bytecode) and 1,771 packages. To the best of our knowledge the PBS is the largest such collection of Java programs. Most of the benchmarks are freely available and can be obtained from the `Kacheck/J` web page.

Name	Description	
Aglets	Mobile agent toolkit	ag
AlgebraDB	Relational database	db
Bloat	Purdue bytecode optimizer	bl
Denim	Design tool	de
Forte	Integrated dev. environment	fo
GFC	Graphic foundation classes	gf
GJ	Java compiler	gj
HyperJ	IBM composition framework	hj
JAX	Packaging tool	ja
JDK 1.1.8	Library code (Sun)	j1
JDK 1.2.2	Library code (Sun)	j2
JDK 1.3.0	Library code (IBM)	j3
JDK 1.3.1	Library code (Sun)	j4
JavaSeal	Mobile agent system	js
Jalapeno 1.1	Java JIT compiler	jp
JPython	Python implementation	jy
JTB	Purdue Java tree builder	jb
JTOpen	IBM toolbox for Java	jt
Kawa	Scheme compiler	kw
OVM	Java virtual machine	o4
Ozone	ODBMS	oz
Rhino	Javascript interpreter	rh
SableCC	Java to HTML translator	sc
Satin	Toolkit from Berkeley	sa
Schroeder	Audio editor	sh
Soot	Bytecode optimizer framework	so
Symjpack	Symbolic math package	sy
Tomcat	Java servlet reference impl.	tc
Toba	Bytecode-to-C translator	to
Voyager	Distributed object system	vy
Web Server	Java Web Server	ws
Xerces	XML parser	xe
Zeus	Java/XML data binding	ze

Figure 8: The Purdue Benchmark Suite (PBS v1.0).

Figure 9 gives an overview of the sizes, in number of classes, for each program or library that is part of the PBS. Appendix B provides additional data about the benchmarks. Our largest benchmarks, over 2,000 classes each, are Forte, JDK 1.2.2, JDK 1.3.*, Ozone, Voyager and JTOpen. Ozone and Forte are applications, while the others are libraries. The number of package-scoped classes is indicated in light gray for each application. This number is an upper bound for the number of confined classes; public classes can not be confined.

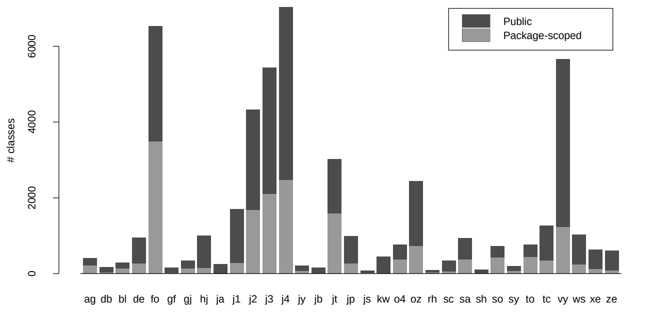

Figure 9: Benchmark characteristics: program sizes.

Figure 10 relates the proportion of package-scoped members to package-scoped classes. Package-scoped members are fields and methods that are declared to have either private or default access. Most coding disciplines encourage the use of package-scoped methods and package-scoped classes. Not surprisingly, programs that were designed with reuse in mind, such as libraries and frameworks, are better-written than one-shot applications. For instance, the Aglet workbench and JTOpen, both libraries, exhibit high degrees of encapsulation. Forte is noteworthy because even though it is an application, it has over 50% package-scoped classes and members. Compilers and optimizers written in an object-oriented style, such as Bloat, Toba and Soot, have high numbers of package-scoped classes because of the many classes used to represent syntactic elements or individual bytecode instructions. At the other extreme, we have applications like Jax and Kawa which have almost no package-scoped classes. It is also worth noting the increase in encapsulation between different versions of the JDK. The percentage of package-scoped classes doubled between JDK1.1.8 and JDK1.3.1, while the absolute number of classes tripled.

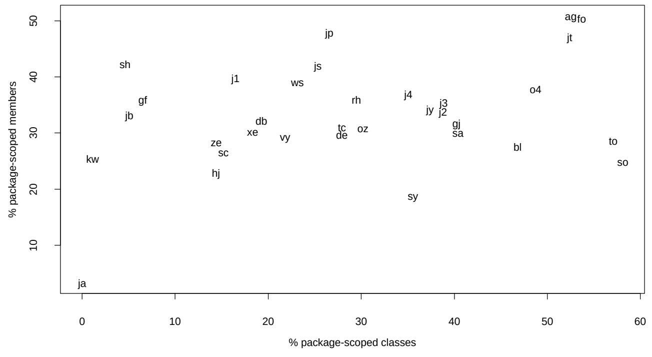

Figure 10: Benchmark characteristics: member encapsulation.

Coding style has an impact on confinement. While the relation between package-scoped classes and confined types is obvious, there is a more subtle connection between package-scoped members and confined types: public and protected methods can return potentially confined types. So it is reasonable to expect that programs with low proportions of package-scoped members will also have comparatively fewer confined types.

6.2 Confined Types

Running `Kacheck/J` over the PBS yields 3,998 confined classes, 25% of the package-scoped classes are confined. Figure 11 shows confined classes in percentage of all classes. The numbers are broken down per program with confined inner classes in light gray. Raw numbers are given in Appendix B.

There are 6 programs where more than 40% of the package-scoped types are confined (db, gf, jy, jb, jp, o4). It is interesting to note that these programs have very little in common: they are a mix of libraries (gf), frameworks (o4) and applications (db, jy, jb, jp). Their ratio of package-scoped classes and their sizes vary widely. Indeed, manual inspection of the programs indicates that programming style is essential to confinement. For example, in early versions of OVM and `Kacheck/J`, unit tests were systematically stored

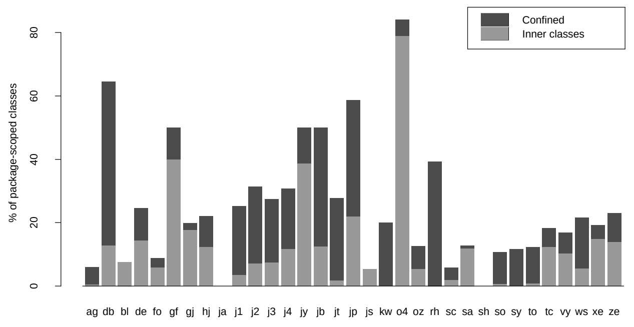

Figure 11: Confined types.

in a sub-package of the current package. Some methods and classes were declared public only to allow testing of the code. This in turn prevented many classes from being confined. The large number of confined inner classes in OVM (o4) comes from the objects representing bytecode instructions nested in an instruction set class. For Jalapeno, the high confinement ratio (153 classes out of 994) is partially the result of the single package structure of the program.

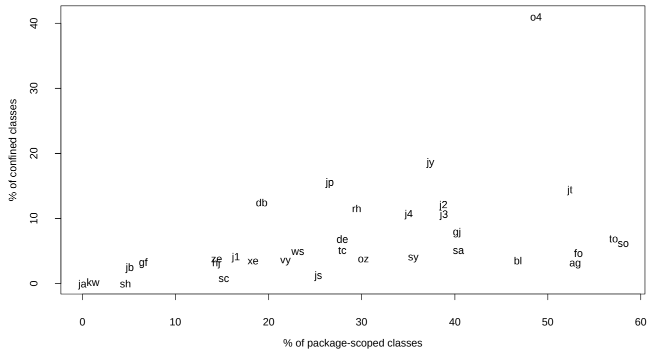

Figure 12: Confinement and package-scoping.

Quite predictably, programs with very few package-scoped classes (e.g. ja, kw, sh, gf) end up with few confined classes. Figure 12 shows the relationship between package-scoped classes and confined classes. The variability in this figure is quite high. For instance, libraries like Aglets (ag) which have very high ratios of package-scoped members and classes still perform quite poorly with only 13 classes being confined out of 410. Why does this happen? There can be two explanations: either the classes are really confined and our analysis is simply not powerful enough to discover that this is the case, or our original assumption that package-scoped classes are naturally confined is wrong. The first case leads to the question of how to improve our analysis. The second case raises the question of whether we can refactor the code to make them confined. To answer these questions, we start with a discussion of confinement violations.

6.3 Confinement Violations

Confinement breaches are caused by a small number of widely used programming idioms. For any violation `Kacheck/J` returns a textual representation of the implication chain that caused the violation. We give examples of the main causes for classes not being confined.

6.3.1 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);     ... }  }
```

6.3.2 Widening to superclass

Widening to a superclass is among the most frequent kind of confinement breach. For instance, Kacheck/J signals the following widening in the Aglet benchmark:

```
com/ibm/aglets/tahiti/SecurityPermissionEditor:
- illegal widening to:
  -  com/ibm/aglets/tahiti/PermissionEditor
```

PermissionEditor is an abstract superclass of the non-public SecurityPermissionEditor. PermissionEditor is the part of the interface that is exported outside the package.

6.3.3 Widening in Containers

A large number of violations comes from the use of container classes in Java. Data structures such as vectors and hashtables always take arguments of type Object, thus any use of a container will entail widening to the most generic super type. For instance, Kacheck/J reports that NativeLibrary, an inner class of ClassLoader, is not confined.

```
java/lang/ClassLoader$NativeLibrary:
     Illegal Widening 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. The internals of class loaders should really be encapsulated. 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 requires inspection of the source code and only remains true only until the next patch is applied to the class. This example shows the usefulness of tools such as Kacheck/J as they can direct the attention of software engineers towards potential security breaches or software defects.

6.3.4 Anonymous Inner Classes

This violation occurs frequently when inner classes are used to implement call-backs. For example in Aglets the MouseListener class is public. Thus, the following code violates confinement of the anonymous inner class.

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

Similar situations occur with package-scoped classes that implement public interfaces. They are package-scoped to protect their members, but are exported outside of the package.

6.4 Confinement with Generics

In Java, vectors, hashtables and other containers are omnipresent. Every time an object is stored in a container, its type is widened to Object leading to a widening violation for the object's class. If Java supported proper parametric polymorphism, the large majority of the violations would disappear (there can be a few heterogenous data structures, but they seem be the exception).

In order to try to assess the impact of generics, without rewriting all of the programs in the PBS, we modified Kacheck/J to ignore widening violations linked to containers. This is done by ignoring all widenings to Object that occur in calls to methods of classes java.util. Figure 13 gives the percentages of confined classes without generic violations; we call these classes Generic-Confined (GC). The light gray bars show the original number of confined classes. The dark grey bars show the effect of adding genericity. The number of confined types increases by 875 (over all programs in the PBS).

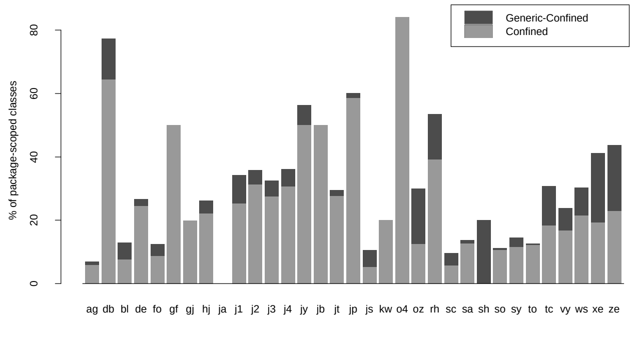

Figure 13: Generic-confined types.

These results should be viewed with caution because they can represent an overestimate of the potential gains since we do not guarantee that the container instances are package-scoped.

6.5 Inferring Access Modifiers

The low number of confined classes in some of the benchmarks is surprising. Looking at the access modifiers of classes in these benchmarks, the reason is immediately clear. For example, in Kawa, out of 443 classes, only 5 are package-scoped. Similarly, many benchmarks contain methods and or fields that are declared as public and thus prevent certain types from being confined. Are these access modes the tightest possible, or are they sometimes randomly chosen? To answer this question we infer the tightest access modes during analysis and then use the inferred modes for confinement checking. Figure 14 shows the result of this analysis. Classes that become confined with modifier inference are called *Confinable* (CA). With mode inference, the number of confinable classes jumps to 13,064 for the entire PBS. Furthermore if we combine confinable and generics, we obtain 14,591 Generic-Confinable classes.

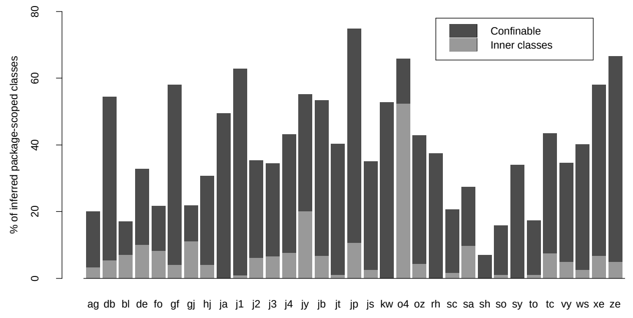

Figure 14: Confinable types.

Figure 15 relates the results of this new analysis to the original number of package-scoped classes. It is quite telling to see that Jax and Kawa, which were applications with the lowest number of confined classes suddenly have about 40% of their classes confinable.

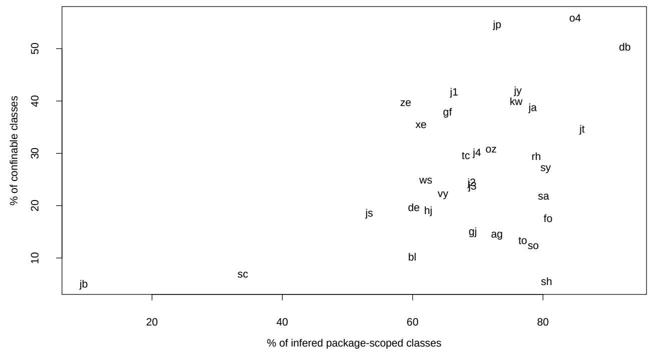

Figure 15: Confinable types and package-scoping.

Of course, using this option on library code may yield an overestimate of the potential gains as some classes that are never used from within the library can be made package-scoped, even though client code requires access to these classes. Nevertheless, the results give a good indication of the potential gains.

6.6 Hierarchical Packages

Our last experiment involves changing the semantics of the Java package mechanism. Currently, Java has a flat package namespace; that is to say, even though package names can be nested, there is no semantics in this nesting. This creates a dilemma between data abstraction and modularity. Good design practice suggests that applications be split into packages according to functional characteristics of the code. On the other hand, creating packages forces certain classes to become public even if those classes should not be used by clients of the program. From a confinement perspective, we could say more packages result in fewer confined classes. One extreme is Jalapeno, which is structured as a single package. This diminishes the usefulness of the confinement property.

To evaluate the impact of the package structure on confinement, we modified `Kacheck/J` to use a hierarchical package model. The general idea is that package-access would be extended to neighbor packages. We introduce a definition of scope that we call *n-package-scoped*. n-package-scoped limits access to classes in packages that are less than n nodes in the tree of package names away from the defining package. For example, the class `java.util.HashtableEntry` would be visible for `java.lang.System` for $n = 2$. The unnamed package is defined to have distance ∞ from all other packages, making a n-package-scoped class `a.A` invisible for `b.B` regardless of the choice of n.

Figure 16 shows the cumulative improvements yielded by increasing the proximity threshold n. With $n = 9$ most programs are treated as a single package and the benefits are 3,691 additional confined classes. The largest increase in confined classes comes from the Voyager benchmark with 813 new confined classes. The most important increment is at $n = 3$ with 2,679 additional confined classes. This threshold value allows classes to access package-scoped members (and classes) of sibling classes.

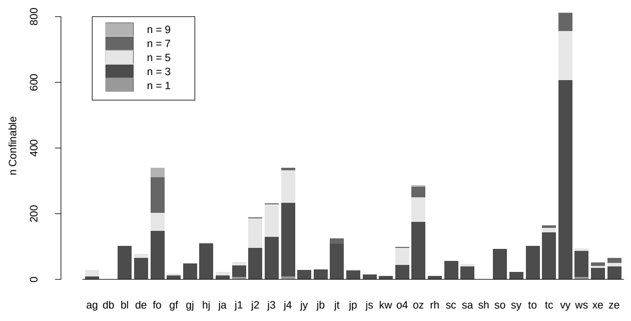

Figure 16: Hierarchical packages.

6.7 Coding for Confinement

Our results clearly point to containers as 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 containers 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 the adapter pattern to wrap an unconfined object around each confined object that must be stored in a container.

A confined implementation of a 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 for example 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`.

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 may be worse.

```
public interface Entry {
    public boolean equal(Entry e);
    public int hashCode();   }
public class Hashtable {
    public void put(Entry e) {...}
    public Entry get(Entry e) {...}  }
class MyEntry implements Entry {
    ConfinedKey key;
    ConfinedValue val;
    public boolean equal(Entry e) {...}
    public int hashCode() {...}  }
```

Figure 17: Example Hashtable interface.

6.8 Runtime Performance

All benchmarks were performed on a Pentium III 800 with 256 MB of RAM running Linux 2.2.19 with IBM JDK 1.3. Except for the JDK tests (j1, j2, j3, j4) all running times include loading and analyzing required parts of the Sun JDK 1.3.1 libraries. The longest running time is that of JDK 1.3.1 which consists of 7,037 classes and is analyzed in 41 seconds. On average, `Kacheck/J` needs 7.5 ms per class. Figure 18 summarizes the cost of confinement checking, detailed timings are in the appendix.

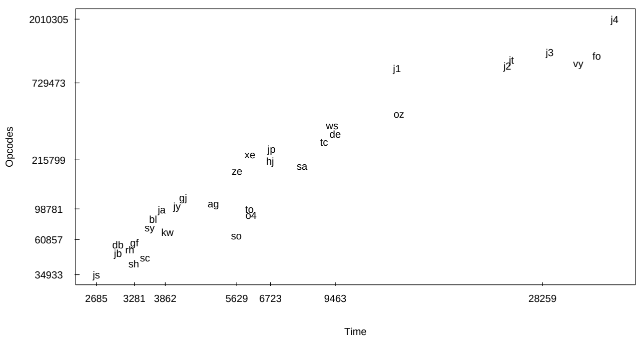

Figure 18: Running times in ms (log10 scale).

7. RELATED WORK

Reference semantics permeate object-oriented programming languages, and the issue of controlling aliasing has been the focus of numerous papers in the recent years [12, 11, 8, 1, 15, 10, 13, 7]. We will discuss briefly the most relevant work.

```
public class Parent {
   protected Parent nonAnonymousMethod() {
       return this;  // violation of A1
   }  }
class NotConf extends Parent {
   Parent violation() {
      return nonAnonymousMethod();
                 // hidden widening
   }  }
```

Figure 19: Confinement violation $\mathcal{C}1$.

Bokowski and Vitek [5] introduced the notion of confined types. In their paper, confined types are explicitly declared. The implication is that software must be designed and implemented with confinement in mind. Their paper discussed an implementation of a source-level confinement checker based on Bokowski's CoffeeStrainer [4]. `Kacheck/J` *infers* confinement from existing Java code. The main difference between that work and the present paper lies in the definition of anonymity. The most interesting confinement breach is hidden widening of confined types to public types that can occur with inherited methods (rule $\mathcal{C}1$).

Consider the example of Figure 19. Intra-procedural analysis would not reveal that `(new NotConf()).violation()` will widen `NotConfined` to `Parent`. So, Bokowski and Vitek chose to rely on explicit anonymity declarations and added an additional anonymity constraint:

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

Thus, once a method is declared anonymous, all overriding definitions of that method have to abide by the constraints. When inferring anonymity, the rule $\mathcal{A}4$ is not necessary. The goal of $\mathcal{A}4$ was to ensure that anonymity of a method is independent from the result of method lookup. If anonymity of methods is inferred, dynamic binding can be taken into account.

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

Figure 20: Anonymity need not be preserved in all subtypes.

Figure 20 shows a confined class `C` that extends a class `A`. The method `A.m()` meets all anonymity criteria except for rule $\mathcal{A}4$. The violation of that rule occurs in class `B`, because `B` extends `A` and redefines `m()` with an implementation that returns `this`. The key point to notice here is that the anonymity violation cannot occur if the dynamic type of `this` is `A`. We say the method `A.m()` is anonymous *in relation* to `C`, but not in relation to `B`.

Another difference between the old and the new anonymity rules is that we allow widening of the `this` reference to other confined types. The old rules forbid returning `this` or using `this` as an argument completely. The new rules allow such cases, if the type of the return value or the argument is again a confined type. An example is shown in figure 21, which is a minimal variation of figure 19. In this case the new rules would allow both classes to be confined.

```
class Parent {
   protected Parent anonymousMethod() {
       return this; // not a violation of A1
   }  }
class Confined extends Parent {
   Parent noViolation() {
      return anonymousMethod();
                 // widening, but no escape
   }  }
```

Figure 21: Confinement!

In [15], flexible alias protection is presented as a means to control potential aliasing amongst components of an aggregate object (or *owner*). Aliasing-mode declarations specify constraints on the 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. Clarke, Potter, and Noble [7] have formalized representation containment by means of ownership types.

Hogg's Islands [11] 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 [15] 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 present an abstract interpretation algorithm to decide if a class meets his balloon invariants, but it was also not implemented so far. Balloon types are similar to confined types in that they only require an analysis of the code of the balloon type and not of the whole program.

Boyland, Noble and Retert [6] introduced capabilities as a uniform system to describe restrictions imposed on references. Their system can model many of the different modifiers used to address the aliasing problem, such as immutable, unique, readonly or borrowed. They also model a notion of anonymous references, which is different from the one used in this paper. Their system of access rights cannot be used to model confined types, mainly because it lacks support for modeling package-scoped access.

Kent and Maung [13] 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 escape analyses such as the ones proposed by Blanchet [2] and others [3, 9] provide much more precise results than our technique, but come at a higher analysis cost. They often require whole program analyses, and are sensitive to small changes in the source code. More than anything, their results can be hard to interpret for a programmer; knowing that an object escapes may not be enough to have an idea how to re-engineer the code to avoid such an occurrence.

8. CONCLUSION

We have presented the `Kacheck/J` tool for inferring confinement in Java programs and used the tool to analyze over 46,000 classes. The results of the analysis are surprisingly high, about 25% of all package-scoped classes and interfaces are confined. Furthermore, we discovered that many of the confinement violations are caused by the use of container classes and thus might be solved by extending Java with genericity, this would increase confinement to 30%. The biggest surprise was the number of violations due to badly chosen access modifiers. After inferring tighter access modifiers, 45% of all package-scoped classes were confined. We expect that these numbers will rise even further once programmers start to write code with confinement in mind..

Confinement is an important property. It bounds aliasing of encapsulated objects to the defining package of their class, and helps in re-engineering object-oriented software by exposing potential software defects, or at least making, often subtle, dependencies visible. We have demonstrated that inferring confined types is fast and scalable. `Kacheck/J` is available from

`http://gecko.cs.purdue.edu/kacheck/`

Acknowledgments

This work is supported by grants from DAAD, Lockheed-Martin, CERIAS, and NSF (CCR–9734265). The authors wish to thank Ben Titzer and Theodore Witkamp for developing the modifier inference package, and Mario Südholt as well as the members of IFIP WG2.4 for helpful comments. Some of the programs from the Purdue Benchmark Suite originate from the Ashes suite, we wish to thank the Sable research group at McGill for making these available.

9. 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] John Boyland, James Noble, and William Retert. Capabilities for aliasing: A generalisation of uniqueness and read-only. In *ECOOP'01 — Object-Oriented Programming, 15th European Conference*, number to appear in Lecture Notes in Computer Science, Berlin, Heidelberg, New York, 2001. Springer.

[7] 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.

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

[9] 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.

[10] 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.

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

[12] 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.

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

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

[15] 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.

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

APPENDIX

A. CONSTRAINT GENERATION

In this section we present which opcodes generate which constraints for confined types.

InvokeStatic

- If `this` occurs in the argument list, record widening of `this` to the type T of the matching argument in the current method m. This generates the constraint: $C \Rightarrow A$ where C is $\textsf{not-conf}(T)$ and A is $\textsf{not-anon}(m)$.
- For each argument a of inferred type T that is an object, record the corresponding declared type T' of the parameter. This generates constraints $C' \Rightarrow C$ where C' is $\textsf{not-conf}(T')$ and C is $\textsf{not-conf}(T)$.

Areturn, Putfield, Putstatic, Aastore

- If the variable that is returned or stored is `this`, record widening of `this` to the declared type T' (the return type, type of the field or the type of the array). This generates a constraint $A \Rightarrow C$ where C is $\textsf{not-conf}(T')$ and A is $\textsf{not-anon}(m)$ with m being the current method.
- If the variable that is used is an object but not `this` and has inferred type T, record widening to the corresponding declared type T'. This generates constraints $C \Rightarrow C'$ where C is $\textsf{not-conf}(T')$ and C is $\textsf{not-conf}(T)$.

InvokeInterface, InvokeVirtual, InvokeSpecial

- If `this` occurs in the argument list, record widening of `this` to the type T of the matching argument in the current method m. This generates the constraint: $C \Rightarrow A$ where C is $\textsf{not-conf}(T)$ and A is $\textsf{not-anon}(m)$.
- If the call is of the form `this.n()`, calling a method n from method m on `this`, record method invocation distinguishing between invokevirtual, invokeinterface and invokespecial. This generates the constraint $A \Rightarrow A'$ where A is $\textsf{not-anon}(n)$ and A' is $\textsf{not-anon}(m)$.
- If the call is not on `this` but of the form $a.n()$, record an invocation on type T where T is the inferred type of a. This generates the constraint $A \Rightarrow C$ where A is $\textsf{not-anon}(n)$ and C is $\textsf{not-conf}(T)$.
- For each argument a of inferred type T that is an object, record the corresponding declared type T' of the parameter. This generates constraints $C \Rightarrow C'$ where C is $\textsf{not-conf}(T')$ and C is $\textsf{not-conf}(T)$.

Athrow

- If the variable that is thrown is `this`, record widening of `this` to `Throwable`. This generates a constraint $C \Rightarrow A$ where C is $\textsf{not-conf}(\texttt{Throwable})$ and A is $\textsf{not-anon}(m)$ with m being the current method. Because the condition $\textsf{not-conf}(\texttt{Throwable})$ is always true, a primitive constraint A can be used, too.
- If the thrown variable is an object but not `this` and has inferred type T, record widening to `Throwable`. This generates a constraint $C \Rightarrow C'$ where C is again always true ($\textsf{not-conf}(\texttt{Throwable})$) and C' is $\textsf{not-conf}(T)$.

Call Propagation

A call to method m on a type T must generate additional constraints for all subtypes S_i of T that are widened to T.

B. BENCHMARK DATA

Benchmark	Classes			Pkgs	Opcodes	Confinement				Time
	All	Public	Inner			C	GC	CA	GCA	(ms)
Aglets	410	193	133	18	107846	13	15	60	66	4979
AlgebraDB	161	130	9	6	51218	20	24	81	97	3009
Bloat	282	150	127	17	84212	10	17	29	39	3623
Denim	949	684	271	63	288140	65	71	187	211	9463
Forte	6535	3053	3769	192	1123362	306	437	1149	1346	37565
GFC	153	143	8	15	58003	5	5	58	58	3284
GJ	338	202	189	12	105323	27	27	51	52	4245
HyperJ	1007	862	70	26	211269	32	38	193	212	6711
JAX	255	255	0	9	97932	0	0	99	104	3790
JDK 1.1.8	1704	1423	29	80	917132	71	96	712	744	13103
JDK 1.2.2	4338	2655	1365	130	958619	527	603	1062	1173	23463
JDK 1.3.0	5438	3326	1780	176	1180406	581	685	1297	1476	29336
JDK 1.3.1	7037	4569	2043	213	2010305	756	891	2126	2344	41304
JPython	214	134	35	7	103094	40	45	90	107	4107
JTB	158	150	1	6	48900	4	4	8	8	3009
JTOpen	3022	1439	557	52	1048704	438	467	1049	1113	23950
Jalapeno 1.1	994	730	132	29	255436	155	159	543	549	6770
JavaSeal	75	56	19	9	34933	1	2	14	17	2685
Kawa	443	438	100	6	68733	1	1	177	177	3910
OVM	763	391	539	26	89975	313	313	427	428	6072
Ozone	2442	1705	490	112	447984	93	221	754	920	13245
Rhino	95	67	1	5	51752	11	15	28	33	3201
SableCC	342	290	47	8	45621	3	5	24	28	3470
Satin	938	559	455	48	194985	48	52	206	218	7955
Schroeder	108	103	7	2	41422	0	1	6	7	3270
Soot	721	302	79	6	65137	45	47	90	92	5622
Symjpack	194	125	0	11	73465	8	10	53	89	3559
Toba	762	327	79	11	98993	53	55	102	104	6020
Tomcat	1271	916	221	93	286368	65	109	377	448	8918
Voyager	5667	4430	1305	294	996077	208	295	1268	1442	34082
Web Server	1024	787	52	60	370664	51	72	255	301	9308
Xerces	622	508	125	35	233919	22	47	221	279	6038
Zeus	604	517	74	39	180437	20	38	237	278	5640
Total	46165	30277	13555	1771	10917301	3998	4873	13064	14591	347567

Figure 22: Statistics for the benchmarks. C is Confined, GC is Generic-Confined, CA is Confinable and GCA is Genrice-Confinable.