Encapsulating Objects with Confined Types [⋆]

Christian Grothoff[1] Jens Palsberg[1] Jan Vitek[2]

[1] UCLA Computer Science Department
University of California, Los Angeles
christian@grothoff.org, palsberg@ucla.edu
[2] Department of Computer Science
Purdue University, West Lafayette
jv@cs.purdue.edu

Abstract. Object-oriented languages provide little support for encapsulating objects. Reference semantics allows objects to escape their defining scope, and the pervasive aliasing that ensues remains a major source of software defects. This paper presents `Kacheck/J`, a tool for inferring object encapsulation properties of 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* – the property that all instances of a given type are encapsulated in their defining package. This simple property can be used to identify accidental leaks of sensitive objects, as well as for compiler optimizations. We report on the analysis of a large body of code and discuss language support and refactoring for confinement.

1 Introduction

Object-oriented languages rely on reference semantics to allow sharing of objects. Sharing occurs when an object is accessible to different clients, while aliasing occurs when an object is accessible from the same client through different access paths. Sharing and aliasing are both powerful tools and sources 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, thus undermining the reliability of the program.

While object-oriented languages provide linguistic support for protecting access to fields, methods, and entire classes, they fail to provide any systematic way of protecting objects. A class may well declare some field private and yet expose the contents of that field by returning it 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 their scope.

The goal of this paper is to report on experiments with a pragmatic notion 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 [6,7,8,22] which are sensitive to small code changes and may be difficult to interpret, as well as more powerful notions of ownership [1,4,5,11,12,13,16,17,21,32,35].Of course, the tradeoff is that we will sometime deem an object as 'escaping' when a more precise analysis would discover that this is not the case. In particular, we have chosen to investigate *confined types* introduced by Bokowski and Vitek in [39] 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, sealed package.

[⋆] The paper is an extended version of a paper with the same title in Proceedings of OOPSLA'01, ACM Conference on Object-Oriented Programming Systems, Languages and Applications, pages 241–253, Tampa Bay, Florida, October 2001.

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

The original definition of confinement required explicit annotations and thus pre-supposes that software is designed with confinement in mind. In some sense, the underlying assumption was that confinement is an unusual property that may require substantial changes to the original program. In this work we take a different point of view. We claim that confinement is a natural property of well designed software systems. We validate our hypothesis empirically with a tool that *infers* confinement in Java programs. We gathered a suite of forty-six thousand Java classes and analyzed them for confinement. Our results show that, without any change to the source, 24% of the package-scoped classes (exactly 3,804 classes or 8% of all classes) are confined. Furthermore, we found that by using generic container types, the number of confined types could be increased by close to one thousand additional classes. Finally, with appropriate tool support to tighten access modifiers, the number of confined classes can be well over 14,500 (or over 29% of all classes) for that same benchmark suite. While a more powerful program analysis may yield higher numbers of confined classes, especially if a whole-program approach is taken, our current numbers are already high and can be obtained efficiently as the average time to analyze a class file is less than eight milliseconds.

In a related effort, Zhao, Palsberg and Vitek [42] have shown that the confinement rules are sound for a simple object calculus inspired by Featherweight Java [30] in which sharing is impossible. This was achieved by recasting the confinement rules into a type system. They also showed the soundness of an extension of the confinement rules to generic types; we will discuss that extension later in this paper.

Since their introduction confined types have been applied in several different contexts. Clarke, Richmond, and Noble have shown that minor changes to the confinement rules presented here can be used to check the architectural integrity constraints that must be satisfied by Enterprise Java Beans applications [15]. Zhao, Noble and Vitek have applied the same ideas to Real-time Specification for Java to ensure static safety of scoped memory usage [41]. Herrmann introduced confinement as a software engineering mechanism for a new programming language [27]. Skalka and Smith have studied a somewhat different notion of confinement within the context of programming language security [38]. In their work the main goal is to control not the flow of references to objects but rather which methods are invoked on those objects.

This paper makes the following contributions and improvements on previous work on confinement:

– We simplify and generalize the confinement rules presented in the original paper on confined types [10].

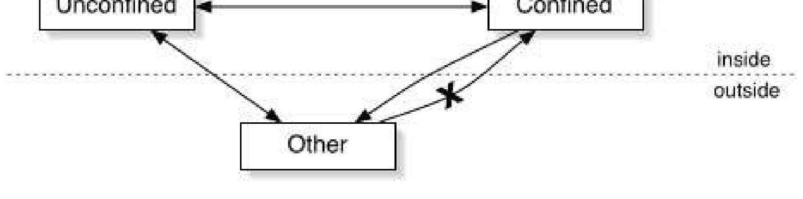

Fig. 1. Confinement is a property of object references. The diagram illustrates confinement in a tiny program with two packages called `inside` and `outside` and three classes `inside.Confined`, `inside.Unconfined` and `outside.Other`. Arrows denote allowed reference patterns. If the class `Confined` obeys the confinement rules, then objects defined in package `outside` cannot hold references to instances of the class `Confined`, the class is said to be encapsulated in the `inside` package.

- We present an efficient constraint-based confinement inference algorithm.
- We give an overview of the implementation of `Kacheck/J`, our confinement inference tool.
- We give results of the confinement analysis of a large corpus of programs.
- We discuss refactorings aimed at improving confinement as well as better language support.

Paper overview The paper is organized as follows. Section 2 starts with a look at an example of confinement in practice with a class taken from the Java standard library. Section 3 introduces confined types and the associated confinement rules. Section 4 presents our constraint-based analysis algorithm. Section 5 discusses the implementation of the inference tool. Section 6 gives result of the analysis of the benchmark suite. Section 7 discusses refactoring and language support. In Section 8 we present an example from the Freenet benchmark [20,19], where `Kacheck/J` is used to first discover that a class is not confined. The code is then refactored such that the class becomes confined. Section 9 gives an overview of related work. Section 10 concludes. The complete constraint generation rules are given in Appendix A. Appendix B gives additional benchmark data.

2 A Practical Example of Confinement

In statically typed 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 illustrate our analysis. Consider the class `HashtableEntry` used within the implementation of `Hashtable` in the `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 that table and definitely not of its defining package. Yet how can we be sure that code outside of the package cannot get access to an entry object?

Since `HashtableEntry` is a package-scoped class we need not worry that outside code will create instances of the class. However, the implementation of the hash table class itself could cast an entry object to some public superclass, and then expose a reference to the object. Alternatively, in the case where a public method were to return an entry object or a public field held a reference to such an object, outside code might obtain a reference to it (possibly causing an unexpected memory leak, say in a weak hash map). The outside code could also use that reference as an argument (which might have security implications if the object was representing ownership of

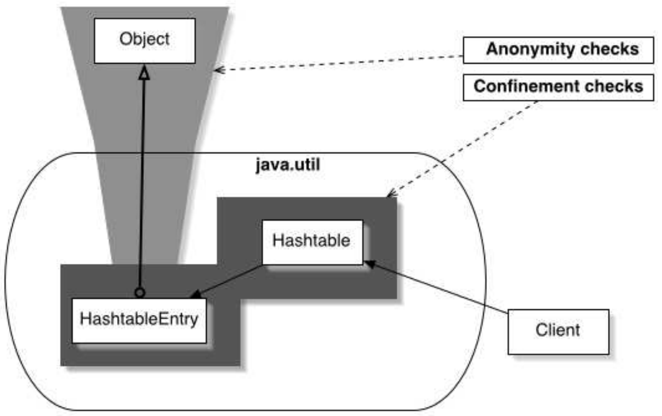

Fig. 2. Analysis overview. All classes in the enclosing package, `java.util` in this case, are checked for confinement. Parent classes of confined classes (e.g. `Object`) are checked for anonymity. Client code need not be checked.

a permission), or cast it to some public parent class and invoke methods on it (which may be problematic, in particular if the methods are overridden in the subclass and were not intended to be reachable from outside of the package).

It is likely that a programmer would consider these scenarios to be the result of a programming error, and a good programmer would take care to prevent such confinement breaches. One can view this issue 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.

Confinement can be checked by a simple program analysis which relies on access modifiers of classes, fields and methods and performs a context and flow insensitive analysis of the code of the confining package. We have implemented a tool called `Kacheck/J` which discovers potential confinement violations and returns a list of confined types for each package analyzed. 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. In order to determine this, the tool must analyze the body of all classes declared in the package `java.util` package, as well as all parent classes of confined classes. Figure 2 illustrates the checks performed by the tool. The analysis is modular since only one package (and the parent classes of confined types) 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 our tool on library code.

It turns out that contrary to our expectation, our analysis infers that the class `HashtableEntry` is not confined because the method `clone()` is invoked on one of its instances. The problem is that clone returns a copy of an entry which is typed as `Object`. Manual inspection of the code reveals that each invocation of this methods is immediately followed by a cast to `HashtableEntry`. Thus instances of the class do not actually escape; this is a typical pattern in a language without adequate support for genericity. Our analysis is not precise enough to discover the idiom—this is part of the price we pay for simplicity. One could consider extending the analysis to catch such idioms and we leave that for future work.

3 Confined Types

The goal of confinement is to satisfy the following soundness property: *An object of confined type is encapsulated in the scope of that type.* Notice that *scope* is a static notion whereas *confinement* controls run-time flow of objects. The idea of confined types is to make the static scope define a bound on where an object can flow. In this work we have set the granularity of confinement to be the package (other granularities have been studied in [15,41]; only minimal changes to the rules are required). Thus, no instance of a confined type may escape the package in which that type is defined. In order to ensure that the analysis is modular and sound in the presence of dynamic loading we must ensure that new code does not show up inside of the encapsulation boundary after the analysis. In Java, this can easily be achieved by requiring that the encapsulating package is sealed [33,40]. Henceforth, when we say that instances of a confined class are encapsulated in their defining package we require that the package is sealed.

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. We use the term widening to denote both:

– *static widening* from C to B: an expression or a statement that requires a check that C is a subtype of B, and
– *hidden widening* to B: an expression or a statement which requires that the type of the distinguished variable `this` is a subtype of B.

A typical example of static widening is an assignment `x=y`, where `x` is of type B and `y` is of type C; Java requires that C is a subtype of B. A typical example of hidden widening is an assignment `x=this`, where `x` is of type B; the dynamic type of the `this`-object cannot be determined locally, so we say that the assignment results in a hidden widening from that dynamic type of `this` to B.

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 `this` which may refer to an object of confined type.

In this section, we adapt the rules of Bokowski and Vitek to infer 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.

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.

3.1 Anonymity Rules

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 aliasing 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.

We say that a method is *anonymous* if it satisfies the three rules in Fig. 3. 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 a method is determined in relation to a specific type. One can use a conservative flow analysis to determine a set of possible target methods, or one can rely on the static type to determine possible targets.

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

Fig. 3. Anonymity rules.

Figure 4 gives an example of a problematic piece of code where a non-anonymous method allows presumably encapsulated objects to escape their container, possibly leaking private information. The interesting thing to note here is that for all assignments in the code the static types match exactly. In particular, the widening of the type of the presumably encapsulated object happens in the `escape` method when the static type of `this` is `C`. Detecting such hidden widenings is the purpose of the anonymity rules. Explicit (static) widenings, that is assignments where the static types of the variables involved are different, are covered by rules described in the next section.

An alternative approach would be to simplify the rules (as taken in [15]) and to disallow confined types to extend types other than `Object`. Anonymity rules are then no longer needed: the only place where hidden widening can occur with that limitation is `Object`. The only violation in `Object` is `clone()` which is then handled with a specific rule. However, while this approach

```
public abstract class C {
   public abstract int getSecret();
   public C escape() {
      return this;
   }
}
class Internal extends C {
   public int getSecret() {
      return 42;
   }
   public C notAnonymous() {
      return escape();
   }
}
public class Container {
   Internal i = new Internal();
   public C exposeAccidentally() {
      return i.notAnonymous();
   }
}
```

Fig. 4. Example of hidden widening in a non-anonymous method resulting in a confinement breach. A client outside of the package could execute `container.exposeAccidentally().getSecret();` to obtain the secret, despite the fact that `Internal` is package-scoped and there is no static widening of `Internal` to `C`.

significantly simplifies the rules for confinement, it also severely restricts the set of classes that can be confined. In this paper we focus on the design with anonymous methods.

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 5. Rule $\mathcal{C}1$ ensures that no inherited method invoked on a confined type will leak the `this` pointer. Together with the anonymity rules this rule prevents hidden widening. Note that the rule does not preclude a confined type from *inheriting* (or even declaring) non-anonymous methods, as long as they are never called. Rule $\mathcal{C}2$ prevents public classes from being confined. This is necessary since code outside of the package must not be able to instantiate a confined type. Rule $\mathcal{C}3$ ensures that no exposed member (public 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. This rule is primarily a design choice from the point of view that if a confined type encapsulates internal information, that information should also not be leaked as part of a subtype. In [42] it was shown that leaking references to confined types from a package can be prevented without this rule. Finally, rule $\mathcal{C}5$ prevents static widening of references of confined type to non-confined types.

3.3 Discussion and Special Cases

Exceptions are a case of widening which is not explicitly listed in our 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

$\mathcal{C}1$	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.

Fig. 5. Confinement rules.

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. In other words, with respect to anonymity, we make the safe choice that a native method cannot be anonymous; it can do whatever it wants. With respect to confinement, we trust the native methods to not violate the confinement rules. The reason for this design decision is that native code that does not conflict with the Java type system may still violate the anonymity rules. However, confinement violations can happen anywhere in native code, thus if we do not want to analyze or rule out all native code, we must trust that native code does not violate confinement. We have manually inspected some of the native code in GNU Classpath, and we found that anonymity violations do happen. We did not find any confinement violations in the native code that we inspected.

In Java, `System.arraycopy` is often used to copy elements from one array to another. While the signature of this special native method takes arguments of type `Object` and thus calls to this method would constitute a widening to a non-confined type, this method is used frequently enough to warrant a special treatment in `Kacheck/J`. The tool treats calls to `System.arraycopy` as a widening from the inferred source-array type to the inferred destination array type. This is safe if the language implements `System.arraycopy` correctly.

Another optimization in `Kacheck/J` is the treatment of static widenings of `this`. Static widenings of `this` are covered by both rules for anonymity ($\mathcal{A}1$) and for confinement ($\mathcal{C}5$). But while rule $\mathcal{A}1$ will only have an impact on confinement if the anonymous method is actually invoked ($\mathcal{C}1$), rule $\mathcal{C}5$ would always make the statically widened type non-confined. While this makes no difference in many cases, this does have an impact on some types if the code in which the widening takes place is dead. In some sense, $\mathcal{A}1$ implicitly contains a limited flow-sensitive dead code analysis, while $\mathcal{C}5$ does not. The `Kacheck/J` tool can be made to relax the rule $\mathcal{C}5$ to not include static widenings of `this` (since those would be caught by rule $\mathcal{A}1$ if the code is not dead). An example for this is shown in Figure 6. If the optimization is enabled, the liveness of the `dead()` method determines whether `Conf` is confined. This illustrates how relaxing $\mathcal{C}5$ makes the analysis more fragile in the sense that small changes in the code are more likely to change the set of confined classes.

```
class Conf {
   public Object dead() {
      return this;
   }
}
```

Fig. 6. Static widenings of `this` can be ignored. This can eliminate confinement violations in dead code.

In a related effort, Zhao, Palsberg and Vitek [42] have shown that the confinement rules are sound for a simple object calculus inspired by Featherweight Java [30] in which sharing is impossible. In that paper, the three anonymity rules are consolidated into just one rule, namely "*the this reference is only used to select fields and as receiver in invocation of other anonymous methods.*" That can be done because the calculus does not have native methods or assignment statements.

Potanin et al. [37] have presented an alternative means to check package confinement, by reduction to Java generics

Clarke, Richmond, and Noble have shown that minor changes to the confinement rules presented here can be used to check the architectural integrity constraints that must be satisfied by Enterprise Java Beans applications [15]. One main difference between their rules and ours is that they don't use a notion of anonymous methods. To a first approximation, we can understand their rules as the result removing $\mathcal{A}1$–$\mathcal{A}3$ from our rules and changing $\mathcal{C}1$ to "All methods invoked on a confined type must be defined in a confined type." Clarke, Richmond, and Noble [15] make a few further restrictions on the confinement rules that are appropriate for the domain of Enterprise Java Beans. In contrast to our analysis, their analysis enables different classes to appear as confined and as unconfined in different parts of the application (i.e., in different beans). The overall result is an analysis which works at a different level of granularity than ours and offers confinement per bean, rather than per package. The experimental results of Clarke, Richmond, and Noble [15] demonstrate that their analysis works well in the domain of Java Beans.

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, as usual, 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:

$$A ::= \mathsf{not\text{-}anon}(\text{methodId})$$
$$T ::= \mathsf{not\text{-}conf}(\text{classId})$$
$$C ::= A \mid T \mid T \Rightarrow A \mid A \Rightarrow A \mid A \Rightarrow T \mid T \Rightarrow T$$

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 7 illustrates the generation of constraints. For each syntactic construct, we have indicated in comments the associated rule from Section 3. Figure 8 details the constraints that are generated for that example. A complete description of the constraints generated from Java bytecode is given in Appendix A. 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 [23] in the number of constraints, which, in turn, is linear in the size of the program.

4.1 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.

Methods of confined classes cannot be invoked from outside of their defining package since confined types are by definition not public ($\mathcal{C}2$) and cannot be widened to non-confined types

($\mathcal{C}5$). So, for anonymity violations that are relevant to a given type, the analysis only needs to consider invocations of methods on instances of that type and its subtypes. Subtypes must be included since confined types may be widened to other confined types.

Our analysis performs a fixed-point iteration starting with the assumption that all non-public classes could potentially be confined. The analysis then records invocations of the type `x.m()`, where the type of `x` is in the current candidate set for confinement. These invocations form the root set for the control flow analysis. Calls of the form `this.m()` that are reachable from this root set are recorded in accordance with anonymity rule $\mathcal{A}3$. The set of types of `this` that are used for resolving virtual method calls is the static type of `x`, as inferred during bytecode verification, and all subtypes of that type that are ever found to be widened to it. Naturally, such widenings (rules $\mathcal{A}1$ and $\mathcal{C}5$) may be detected at any time during the flow analysis, which is the reason why a fixed-point computation is necessary. When the fixed-point computation terminates and all invocation chains for all applicable confinement candidates have been traversed, the remaining types for which no anonymity violations were found are declared confined.

The analysis does not attempt to perform dead-code detection, so while the method that includes an invocation such as `x.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 lead to analysis results that are much more sensitive to changes in the source program.

5 Implementing Confinement Inference: 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 that there are a large number of class files freely available to apply our tool to. The implementation of `Kacheck/J` leverages the XTC static analysis framework which was developed as part of the Ovm JVM. In XTC, bytecode verification is implemented using the Flyweight pattern [25]. For each of the 200 bytecode instructions defined in the Java Virtual Machine Specification, the XTC 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-

```
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)
   }
}
```

Fig. 7. Example program.

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` as receiver
($\mathcal{C}1$)	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`

Fig. 8. The constraints generated from the example in Figure 7.

point iteration. The verification starts with an initial state which includes the instruction pointer, operand stack and variables. The verifier follows all possible flows of control within the method.

By instrumenting the transfer functions of only 9 of the 200 `Instruction` objects we can use XTC's abstract interpretation engine to generate constraints. The instrumentation performs some simple checks and record basic facts about the program execution. For instance, the code for 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 and uses of `this` are recorded.
– All static widenings are recorded; thrown exceptions are considered widened to `Throwable`.

Widenings are captured by intercepting subtype checks done by the verifier. Anonymity checks only require slight modifications to the transfer functions that correspond to the nine instructions: a check is added to record operations on `this`. See Appendix A for details. The flow analysis computes the implication chains for each potentially confined type T, such that

$$(T' \Rightarrow A_1) \wedge (A_1 \Rightarrow A_2) \wedge \ldots\ (A_{n-1} \Rightarrow A_n) \wedge (A_n \Rightarrow T)$$

is collapsed to

$$T' \Rightarrow T.$$

The code specific to confined types (including verbose reporting of violations) is about 2,200 lines. The code reused from XTC (including reading and writing of Java 5.0 class files) is about 30,000 lines of code.

5.1 Example

Figure 9 gives an example of a chain of constraints that results in classes being not confined. Although the tool reorders parts of the solving process, we will in the following explain only the final chain of constraints. Notice first that `Object` is a non-confined class, so a constraint of the type C is generated by rule $\mathcal{C}2$:

$$\textsf{not-conf}(\texttt{Object})$$

The method `P.nonAnon()` widens `this` to `Object`. This will generate a constraint of type $C \Rightarrow A$ by rule $\mathcal{A}1$:

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

The invocation of `nonAnon` in `nonAnonInd` with `this` as the receiver generates a constraint of the type $A \Rightarrow A$ by rule $\mathcal{A}3$:

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

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`).

5.2 Simplifying Assumptions

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

Reflection The analysis assumes that reflection is not used to circumvent 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. The results obtained from `Kacheck/J` are only valid if native methods do not violate any of the confinement rules. Furthermore, we assume that native code in does not violate the semantics of the language by ignoring access control declarations. Manual inspection of a number of native methods indicates that these assumptions are reasonable. We do not assume that native methods satisfy the anonymity rules. Note though that we manually inspected the native methods and did not find any that violate the rules for anonymous methods.

6 Analysis 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. The first goal of the evaluation is to

```
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)
   }
}
```

Fig. 9. A confinement violation.

show that confinement is a common property in actual code (Section 6.2). The second goal is to identify common reasons why certain types are not confined and thereby gauge the limitations of our technique (Section 6.3). Studying reasons for nonconfinement also points out possible places where slight modifications to the analysis would dramatically increase opportunities for confinement. We present three such modifications in sections **??**, 6.5 and 6.6. In order to give evidence that confinement is a stable property that a programmer might want to declare in the program text, Section 6.7 studies how confinement properties of types change during the lifetime of a particular application. Annotating types as confined would not be practical if confinement was a fragile property and annotations would need to be changed frequently. Finally, in order for confinement to be useful in practice we will demonstrate that checking or inferring confinement scales and can be done quickly. Section 6.8 shows that `Kacheck/J` can rapidly analyze huge benchmarks.

6.1 The Purdue Benchmark Suite

The Purdue Benchmark Suite (Figure 10) 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.

Figure 11 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 cannot be confined.

Figure 12 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. From JDK1.1.8 to JDK1.3.1, the absolute number of classes tripled, and yet the percentage of package-scoped classes doubled. The reason is largely that most of the JDK1.1.8 code implements the simple, public core classes of the Java runtime (java.*), whereas JDK1.3.1 has substantial amounts of code that the main application does not interface with directly.

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 3,804 confined classes; 24% of the package-scoped classes and 8% of all classes are confined. Figure 13 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.

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

Fig. 10. The Purdue Benchmark Suite (PBS v1.0).

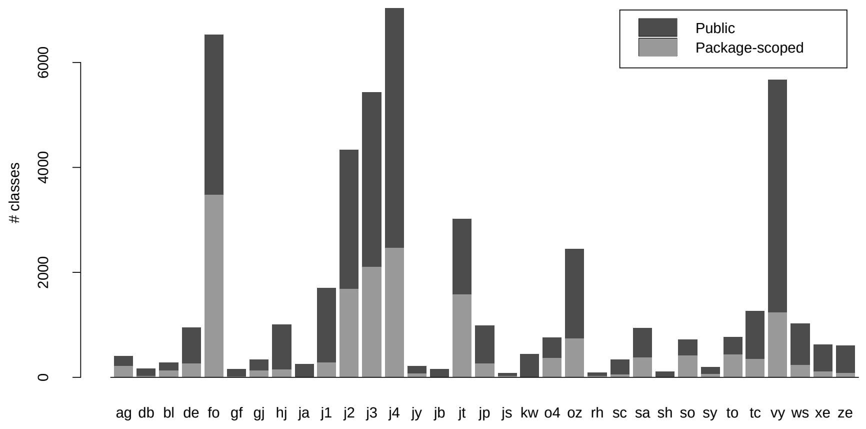

Fig. 11. Benchmark characteristics: program sizes.

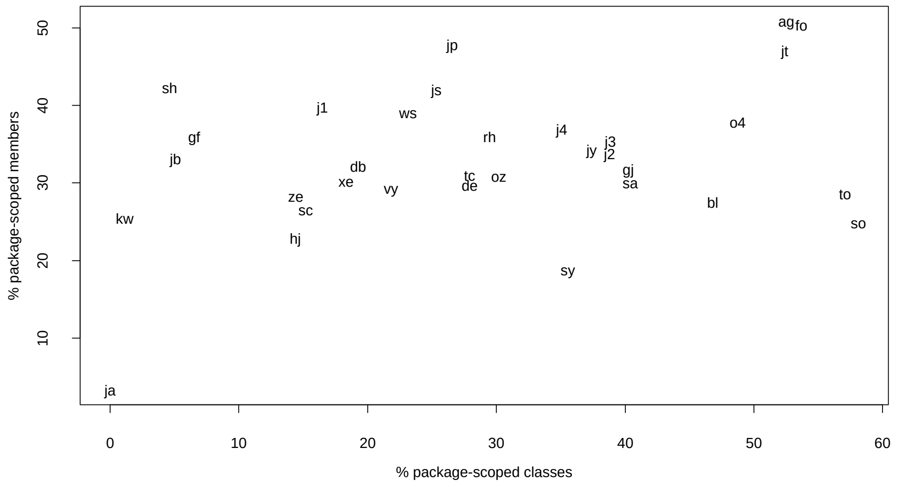

Fig. 12. Benchmark characteristics: member encapsulation.

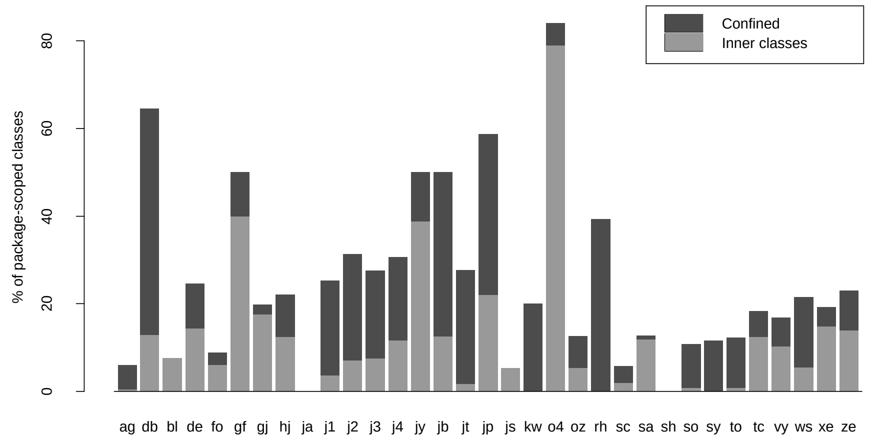

Fig. 13. Confined types.

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 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 of 16% (155 classes out of 994) is partially the result of the single package structure of the program.

Predictably, programs with very few package-scoped classes (e.g. ja, kw, sh, gf) end up with few confined classes. Figure 14 shows the relationship between package-scoped classes and confined classes. Notice that the fraction of package-scoped classes varies considerably from benchmark to benchmark. 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 (3%) being confined out of 410. Why does this happen? To answer that question, we start with a discussion of confinement violations.

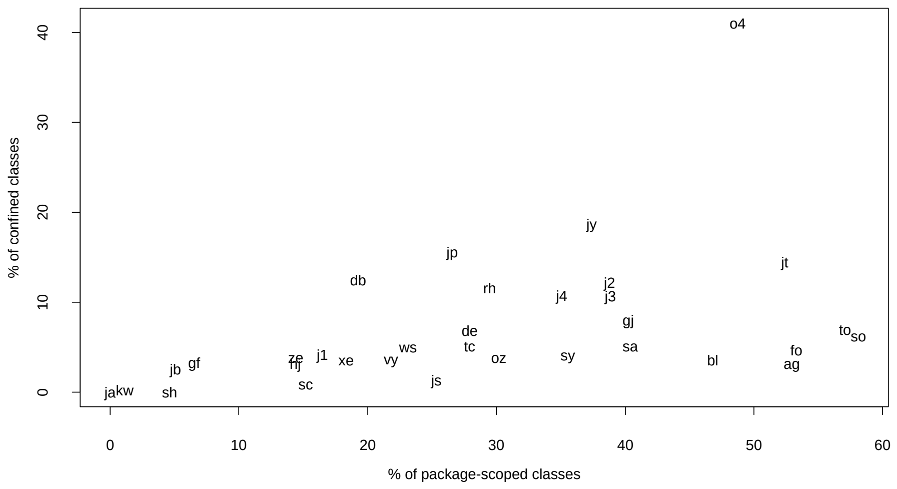

Fig. 14. Confinement and package-scoping.

6.3 Confinement Violations

It is difficult to quantify confinement violations into categories based on the constraints, mostly because many of the constraints work in concert. For example, widening one class to another ($\mathcal{C}5$) may violate confinement because the other class is not confined because, in turn, a non-anonymous method ($\mathcal{A}1$) is invoked ($\mathcal{C}1$) on it. And the reason for why the method is not anonymous could be because a third class is public ($\mathcal{C}2$). Notice that the confinement violation for the original class involves four different constraints. Rather than trying to quantify confinement violations, this section attempts to describe the causes for non-confinement based on a few characteristic examples.

Most 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.

Anonymity Violations The top three anonymity violations 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 cp, Object cn, int i) {
   synchronized ( getTreeLock() ) { ...
      e = new ContainerEvent( this, COMPONENT_ADDED, comp);
      ...
   }
}
```

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
```

The `PermissionEditor` class is an abstract superclass of the non-public `SecurityPermission-Editor`. `PermissionEditor` is the part of the interface that is exported outside the package.

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
```

`Kacheck/J` works with bytecodes, not source code. At the bytecode level of Java, generic types are absent. One might implement a `Kacheck/J`-like tool at the source level which handles generic types in a non-trivial way. We leave that for future work; inspiration might come from the paper by Zhao, Palsberg, and Vitek [42] which presented rules for handling confinement of generic types.
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.

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.

```
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.

Summary Even though confinement violations are often the result of a chain of events, there are two rules which by themselves eliminate most opportunities for confinement and thus deserve further consideration. The confinement rules that are the cause for the largest number of non-confined types overall are $\mathcal{C}2$ (class is public), followed by $\mathcal{C}5$ (instance widened to non-confined type). How dramatic the effect of these rules is shown in the following sections, where small modifications are made which limit the scope of these rules, resulting in a significant increase in the number of confined types. In Section 6.4 widening of confined types is discounted whenever it happens in conjunction with containes (eliminating many common applications of rule $\mathcal{C}5$). In Section 6.5 the access modifiers are inferred, making many classes and methods package-scoped that used to be public. Both variations result in a sharp increase in the number of confined types.

6.4 Confinement with Generics

In Java, vectors, hashtables and other containers are pervasive. 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 heterogeneous 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 15 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 from 3804 (8%) to 4862 (10%) over all programs in the PBS. These results should be viewed with caution because they could represent an overestimate of the potential gains since we do not guarantee that the container instances are package-scoped.

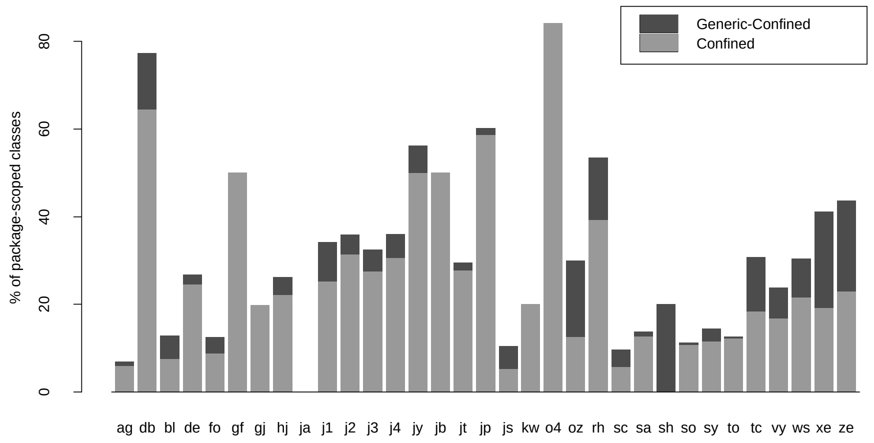

Fig. 15. Generic-confined types.

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 (1%) are package-scoped. Similarly, many benchmarks contain methods and/or fields that are declared as public and thus prevent certain types from being confined. That raises the question of whether the access modes are the tightest possible or whether they are more permissive than necessary. To answer the question we infer the tightest access modes during analysis and then use the inferred modes for confinement checking. This analysis is performed by the Java Access Modifier Inference Tool (JAMIT), which is also available on our webpage.

JAMIT infers the tightest legal access modes by looking at all accesses to a given member or type. It then checks what the most restrictive access modifier is that would permit all accesses according to Java's visibility rules. The analysis takes subtyping into account; subtypes can view protected members and overriding methods can only relax access modifiers. More importantly, in order to preserve overriding the access modifier in the parent may need to be relaxed to package-scoped (if all overriding subtypes reside in the same package) or protected.

Figure 16 shows the result of running `Kacheck/J` on code for which access modifiers were strengthened using JAMIT. Classes that become confined with modifier inference are called *Confinable* (CA). With mode inference, the number of confinable classes jumps from 3804 (8%)

to 12,880 (26.1%) for the entire PBS. Furthermore if we combine confinable and generics, we obtain 14,591 (29.6%) Generic-Confinable classes.

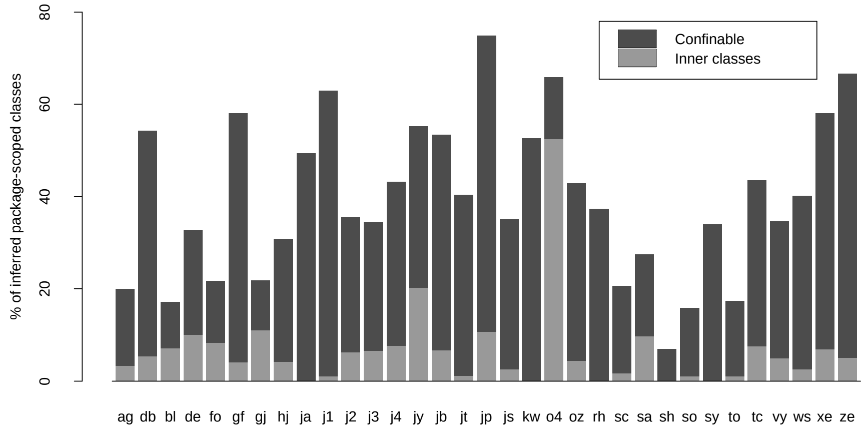

Fig. 16. Confinable types.

Figure 17 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 numbers of confined classes suddenly have about 40% of their classes confinable. 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.

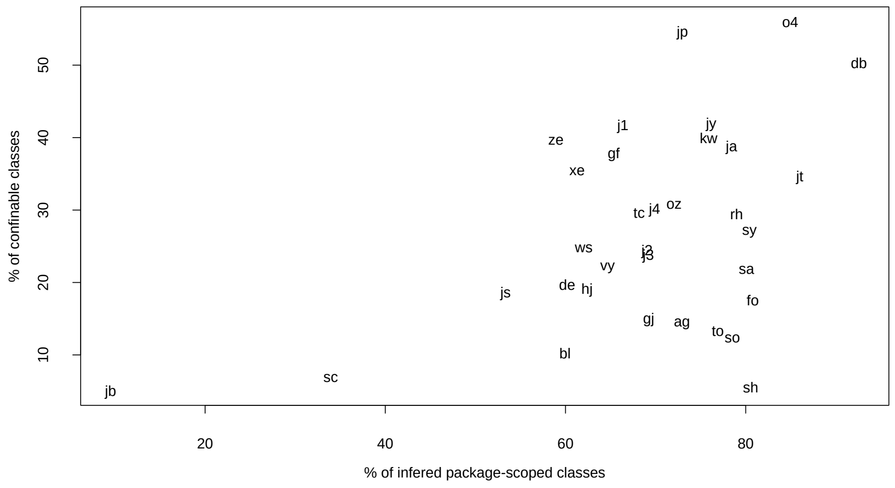

Fig. 17. Confinable types and package-scoping.

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 18 shows the cumulative improvements yielded by increasing the proximity threshold n. With $n = 9$ most programs are treated as a single package, increasing the number of confined types from 3,804 to 7,495. The largest increase in confined classes comes from the Voyager benchmark where the number of confined classes increases from 208 to 1021.

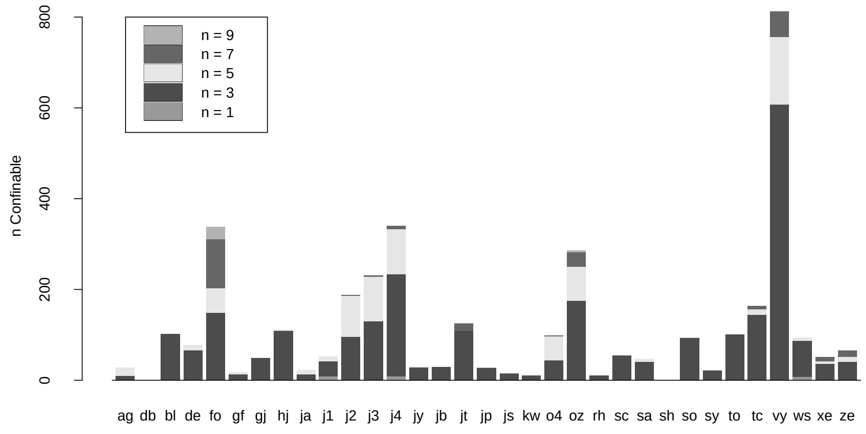

Fig. 18. Confinement with hierarchical packages.

6.7 Evolution of Confinement

For the working software engineer, it may be of interest to know whether confinement is preserved when software evolves. If a class is confined in one software version, then it would be helpful to know whether the class will likely be confined also in the next version. If the answer is yes, then confinement can be viewed as a meaningful, fundamental property of a type, and not just a coincidence of the arrangement of the code. To shed light on this issue, we present a study of the confined types in 14 versions of TomCat, ranging from version 3.0 to an early snapshot of 5.0. The results are unambiguous. Even with dramatic changes to the code base that involve adding and removing hundreds or thousands of classes, only very few existing classes suddenly become confined or stop being confined. Almost all confined classes stay confined (or are removed from the code base), and almost all non-confined classes stay non-confined (or are removed).

Fig. 19 shows the differences in the numbers of confined types between versions. The upward arrows indicate the number of types that are new in a particular version of the code. The top of the upward arrows is anchored at the number of confined types for the specific version. The dashed arrows that go down diagonally from that point indicate the number of types that used to be confined and that have been removed from the codebase. The fact that in almost all places both arrows meet in exactly the same point shows that it is rare that confined types become non-confined and vice versa. The height of the bars at the bottom also illustrates this; the height of the bar is the number of types that are live, were live in the previous version and changed from confined to non-confined or vice versa. The graph shows that while the overall changes to the code are quite significant, the number of types that change their confinement property is marginal (with a total of 6 changes from version 3.0 to 5.0, with the total number of confined types in the different versions in between ranging from 46 to 104 with an average of 68). This stability of the confinement property over time supports the thesis that confinement would be a reasonable annotation for a type.

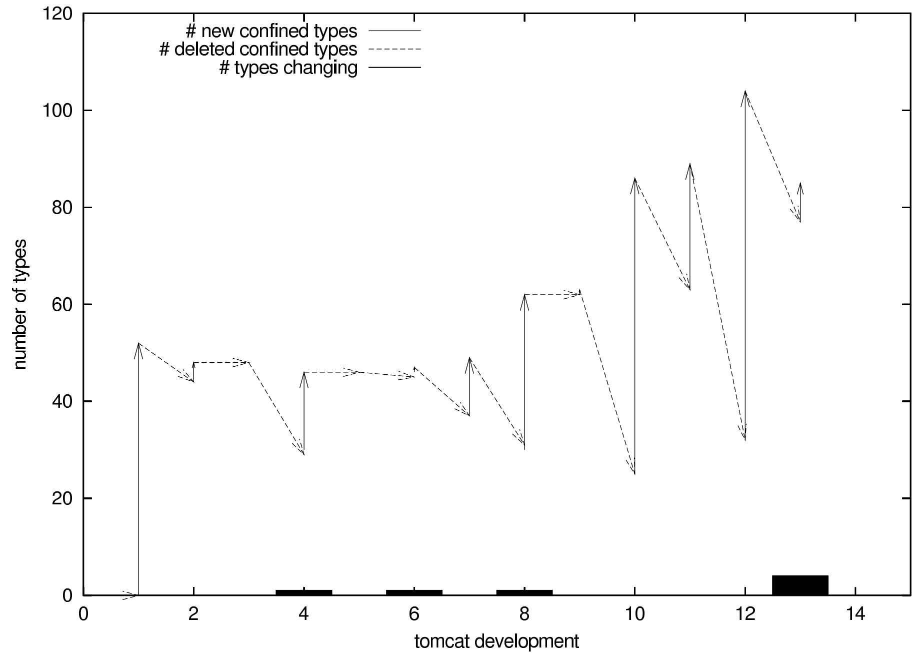

Fig. 19. Number of confined types in different versions of the TomCat benchmark. The top of the solid arrows marks the number of confined types in each version. The dashed arrows refer to the number of confined types that were already present and confined in the previous version of the code. The bars at the bottom represent the number of types that change confinement (become confined or are no longer confined) and exist in the current and the previous version of the code.

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 milliseconds per class. Figure 20 summarizes the cost of confinement checking, detailed timings are in the appendix.

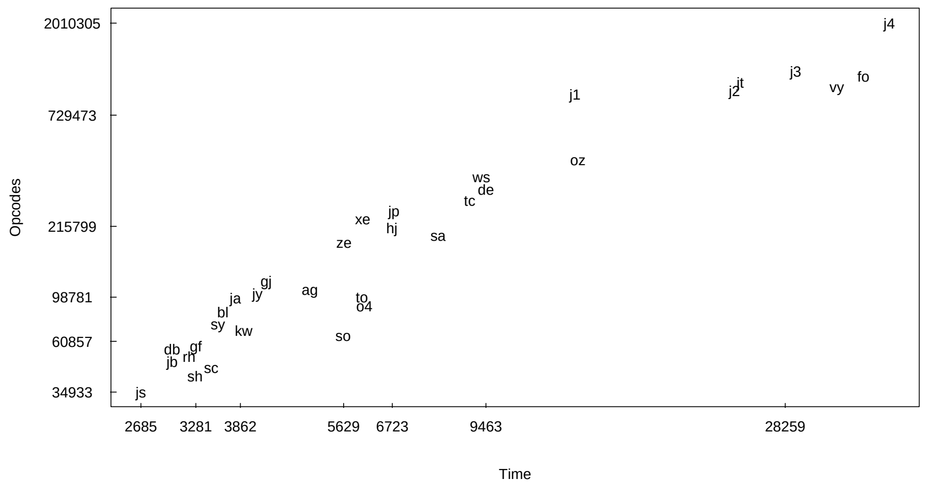

Fig. 20. Running times in ms (log-log scale).

7 Containers and Language Extensions

7.1 Coding for Confinement

Our results clearly point to containers as one source of confinement violations. We considered using generic extensions of Java to increase confinement. Unfortunately, the homogeneous translation strategies adopted by Java implies that at the bytecode level, code written with generics is translated back to code that uses `Object` and casts. One might be able to uncover patterns of bytecode compiled from generics and use that to improve the analysis; however, `Kacheck/J` makes no attempt to do that and thus cannot verify that classes stored in generic containers remain confined. Heterogeneous translation strategies would 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 `equal(Entry e)` and `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.

Zhao, Palsberg and Vitek [42] suggested an alternative that involves extending confinement to generic types and annotating bytecode with confinement assertions. In addition to the existing rules presented so far, they require the rules given in Figure 22. The rules $\mathcal{C}5$ and $\mathcal{C}6$ combined correspond to the subtyping partial order that prevents reference widening for Generic ConfinedFJ. $\mathcal{C}7$ corresponds to the extra requirement in the definition of well-formed generic types. Unlike in the base system, $\mathcal{C}8$ is necessary since we are not certain which method may be called before a generic class is instantiated.

7.2 Improved Language Support

Java can be extended to support confined types in several ways. Such extensions can be more or less intrusive on the syntax and semantics. We will consider two approaches:

```
public interface Entry {
   boolean equal( Entry e);
   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() { ... } }
```

Fig. 21. Example Hashtable interface.

1. explicit annotations for confined classes and anonymous methods, and
2. explicit annotations for confined classes but not for anonymous methods.

Using the meta-data facilities of Java 5 it is easy to add such annotations to Java code. Figure 23 shows how to specify the `Confined` and `Anonymous` annotations. In order to allow for running a static checker on the bytecode, the confinement property is preserved for the class files. The annotations are not needed at runtime. The rule that subclasses of confined types should also be confined is made explicit by the `inherited` annotation.

Explicit annotations of classes and methods In Bokowski and Vitek's original proposal for confined types [39], both confined classes and anonymous methods had explicit modifiers, in the following style:

```
@Confined class C extends B {
   @Anonymous int m() {
      return this.n();
   }
}
```

The constraints of Bokowski and Vitek are stricter than the constraints presented in this paper. In particular, Bokowski and Vitek require that the anonymity of a method is preserved in all subclasses and that the static receiver of a virtual call must be anonymous. In contrast, the constraints checked by `Kacheck/J` only require the unique dynamic targets to be anonymous. Thus the more modular checking will result in fewer confinement opportunities.

The explicit annotation `@Anonymous` for anonymous methods simplifies checking the confinement constraint $\mathcal{C}1$. That rule can be checked by (1) ensuring that every method invoked on a

$\mathcal{C}6$	A generic type or type variable cannot be widened to a type containing a different set of type variables.
$\mathcal{C}7$	A confined type cannot replace a public type variable in the instantiation of a generic type.
$\mathcal{C}8$	Overriding must preserve anonymity of methods.

Fig. 22. Alternative confinement rules

```
@java.lang.annotation.Retention(java.lang.annotation.RetentionPolicy.CLASS)
@java.lang.annotation.Inherited
public @interface Anonymous {}
@java.lang.annotation.Retention(java.lang.annotation.RetentionPolicy.CLASS)
@java.lang.annotation.Inherited
public @interface Confined {}
```

Fig. 23. Defining annotations for confinement and anonymous methods in Java 5. The presented code defines two annotations (`@Anonymous` and `@Confined`) which according to the given *retention policy* will be compiled into the `.class` files, but which will not be available for introspection at runtime. The *inherited* declaration ensures that the annotations are automatically applied to all subtypes.

confined type is declared as `@Anonymous` and (2) by checking the constraints given by Bokowski and Vitek [39] for anonymous methods. Having the programmer specifically specify methods as anonymous also makes it easier for the programmer to reason about confinement, just like confinement can be checked in a modular way the confinement violations are more localized since this approach avoids having to follow chains of anonymity violations across multiple methods.

Explicit annotations of classes, but not of methods There are many more anonymous methods than confined classes. Thus, the burden on the programmer to annotate code can be lightened considerably by only requiring explicit annotation of classes. Moreover, the resulting inference of anonymous methods can be done according to the rules presented in this paper. This inference is scalable and more precise than annotations for anonymous methods. Annotating existing code with a `@Confined` modifier can be done automatically with the results from `Kacheck/J`. The latest version of `Kacheck/J` for Java 5 allows both automatically annotating bytecode with `@Confined` metadata attributes (for use by other analyses that need confinement information) as well as checking that all types that are annotated to be `@Confined` in the source are actually confined (for verification of confinement assertions provided by the programmer).

8 Refactoring for Confinement

In this section we detail how `Kacheck/J` can aid the process of first discovering that a class is not confined and then refactoring the program such that the class becomes confined.

8.1 The example program

We will use an example which stems from the Freenet application [20,19]. The example was found by inspecting the Freenet source code and discovering that class `DoublyLinkedListImpl` has an inner class which probably should be confined. For clarity, we will work with a much condensed version of class `DoublyLinkedListImpl` and two of its clients. We condensed the code mainly by removing methods and code sections irrelevant to our quest for making a class confined.

Our example program is shown in Figure 24. The example program contains, in the left column, a rudimentary interface `DoublyLinkedList` and an implementation `DoublyLinkedListImpl` of doubly linked lists. The example program also contains, in the right column, two pieces of client code, called `IntervalledSum` and `LoadStats`, that use doubly linked lists. Notice that class `DoublyLinkedListImpl` has an inner class `ItemImpl` (which was called `Item` in the Freenet source code). Class `ItemImpl` is used to represent the state of objects of class `DoublyLinkedListImpl`. If we want to encapsulate the state of objects of class `DoublyLinkedListImpl`, then class `ItemImpl` should be confined.

The Freenet code was written by many different authors. The multiple authorship may explain the two inconsistent uses of class `DoublyLinkedListImpl`: one client re-implements the `Item` interface from scratch, whereas another client extends the `ItemImpl` code. The Freenet code contains more than just these two uses of `DoublyLinkedList`; the two clients in Figure 24 are simple yet representative samples.

```
package freenet.support;

public interface DoublyLinkedList {
  public interface Item {
    Item getNext();
    Item setNext(Item i);
    Item getPrev();
    Item setPrev(Item i);
  }
  int size();
  java.util.Enumeration elements();
  void push(Item i);
}
public class DoublyLinkedListImpl
  implements DoublyLinkedList {
  protected int size;
  protected Item head, tail;
  public int size() { return size; }
  public java.util.Enumeration elements() {
    return new java.util.Enumeration() {
      protected Item next = head;
      public boolean hasMoreElements() {
        return next != null;
      }
      public Object nextElement() {
        if (next == null) throw new
          java.util.NoSuchElementException();
        Item result = next;
        next = next.getNext();
        return result;
      }
    };
  }
  public void push(Item j) {
    // ...
    ++size;
  }
  public static class ItemImpl
    implements Item {
    private Item next, prev;
    public Item getNext() { return next; }
    public Item setNext(Item i) {
      Item old = next; next = i; return old;
    }
    public Item getPrev() { return prev; }
    public Item setPrev(Item i) {
      Item old = prev; prev = i; return old;
    }
  }
}
```

```
package freenet.support;

import freenet.support.DoublyLinkedList.Item;

public class IntervalledSum  {
  private final DoublyLinkedList l
    = new DoublyLinkedListImpl();
  public void report(double d) {
    l.push(new Report(d));
  }
  static class Report implements Item {
    double value;
    private Item prev, next;
    Report(double value) { this.value = value; }
    public Item getNext() { return next; }
    public Item setNext(Item i) {
      Item r = next; next = i; return r;
    }
    public Item getPrev() { return prev; }
    public Item setPrev(Item i) {
      Item r = prev; prev = i; return i;
    }
  }
}
```

```
package freenet.node;

import freenet.support.*;

public class LoadStats {
  private final DoublyLinkedListImpl lru
    = new DoublyLinkedListImpl();
  private final java.util.Map table
    = new java.util.HashMap();
  public void storeTraffic(byte[] n, long r) {
    LoadEntry le = new LoadEntry(n, r);
    table.put(le.fn, le);
    lru.push(le);
  }
  class LoadEntry
    extends DoublyLinkedListImpl.ItemImpl {
    private final Object fn;
    private final long qph;
    private LoadEntry(byte[] b, long qph) {
      this.fn = b;
      this.qph = qph;
    }
  }
}
```

Fig. 24. Doubly linked lists and two of their clients

8.2 Refactoring: remove simple confinement violations

In Figure 24, the class `ItemImpl` is public and therefore it is not confined by definition. However, confining `ItemImpl` is probably a good idea since its state is the internal representation of the `DoublyLinkedList`. Having clients outside of the package manipulate `ItemImpl`-objects might easily break invariants of the `DoublyLinkedList` implementation, such as the size of the list or the `head` and `tail` fields.

In order to confine `ItemImpl`, we must remove all violations of the confinement rules. Let us first consider rule $\mathcal{C}4$ which requires that subtypes of a confined type must be confined. This clearly conflicts with the subclassing of `ItemImpl` by `LoadEntry`. This problem can be solved using the "Replace Inheritance with Delegation" refactoring pattern [24]. Instead of extending `Item` a field `value` is added to the `Item` class. We use generics in order to give the field an appropriate type. Using this design also removes the code duplication in `Report`, which no longer needs to implement `Item`. Since `ItemImpl` is now going to be the only implementation of the `Item` interface, the split between implementation and interfaces is quite useless, so in order to simplify the code we remove the interfaces and eliminate the `Impl` from the names of the classes of the implementation. Finally, the access modifier of `Item` (formerly `ItemImpl`) is changed from public to default (in order to satisfy confinement rule $\mathcal{C}2$). The result of the first refactoring is the program in Figure 25.

```
package freenet.support;
public class DoublyLinkedList<T> {
  private int size;
  private Item<T> head, tail;
  public int size() { return size; }
  public java.util.Enumeration elements() {
    return new java.util.Enumeration() {
      protected Item next = head;
      public boolean hasMoreElements() {
        return next != null;
      }
      public Object nextElement() {
        if (next == null) throw new
          java.util.NoSuchElementException();
        Item result = next;
        next = next.next;
        return result;
      }
    };
  }
  public void push(T j) {
    // ...
    ++size;
  }
  static class Item<T> {
    Item next, prev;
    public final T value;
    Item(T val) { this.value = val; }
  }
}
```

```
package freenet.support;

import freenet.support.DoublyLinkedList;

public class IntervalledSum  {
  private final DoublyLinkedList<Report> l
    = new DoublyLinkedList<Report>();
  public void report(double d) {
    l.push(new Report(d));
  }
  static class Report {
    double value;
    Report(double value) { this.value = value; }
  }
}
```

```
package freenet.node;

import freenet.support.*;

public class LoadStats {
  private final DoublyLinkedList<LoadEntry> lru
    = new DoublyLinkedList<LoadEntry>();
  private final java.util.Map table
    = new java.util.HashMap();
  public void storeTraffic(byte[] nr, long rph) {
    LoadEntry le = new LoadEntry(nr, rph);
    table.put(le.fn, le);
    lru.push(le);
  }
  class LoadEntry {
    private final Object fn;
    private final long qph;
    private LoadEntry(byte[] b, long qph) {
      this.fn = b;
      this.qph = qph;
    }
  }
}
```

Fig. 25. The code after the first refactoring

8.3 Refactoring: remove widening violations

If we run Kacheck/J on the program in Figure 25 we will get the result that class Item is still not confined. The problem is that method nextElement widens Item to Object (upon return). We can refactor the program in Figure 25 to remove the violation.

The result of the second refactoring is the program in Figure 26. As a result of the refactoring, Item is confined and clients can no longer easily break invariants of the DoublyLinkedList container.

8.4 Refactoring: Summary

In our experience the biggest hurdle in refactoring code for confinement is to find candidates where such a refactoring would truely improve the code. The primary obstacle are Java's containers, which could theoretically be addressed by checking confinement at the source level. Nevertheless, in practice many classes can be easily confined by flattening the hierarchy and possibly wrapping references to instances in another class. However, while it is often easy to achieve confinement, refactoring code blindly simply to maximize confinement may result in unnatural datastructures with too many layers of abstraction.

9 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 [2,3,18,21,26,28,29,31,36]. We will discuss briefly the most relevant work.

```
package freenet.support;
public class DoublyLinkedList<T> {
  private int size;
  private Item<T> head, tail;
  public int size() { return size; }
  public java.util.Enumeration<T> elements() {
    return new java.util.Enumeration<T>() {
      protected Item next = head;
      public boolean hasMoreElements() {
        return next != null;
      }
      public T nextElement() {
        if (next == null) throw new
          java.util.NoSuchElementException();
        Item<T> result = next;
        next = next.next;
        return result.value;
      }
    };
  }
  public void push(T j) {
    // ...
    ++size;
  }
  static class Item<T> {
    Item next, prev;
    public final T value;
    Item(T val) { this.value = val; }
  }
}
```

Fig. 26. The code after the second refactoring

Bokowski and Vitek [10] introduced the notion of confined types. In their paper, confined types are explicitly declared. Their paper discussed an implementation of a source-level confinement checker based on Bokowski's CoffeeStrainer [9]. The main difference between that work and the present paper lies in the definition of anonymity. In both cases anonymity rules are used to detect confinement breaches from hidden widening of confined types to public types that can occur with inherited methods (rule $\mathcal{C}1$). However, the rules given by Bokowski and Vitek are much stronger than strictly necessary.

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

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

Fig. 27. Confinement violation $\mathcal{C}1$.

Consider the example of Figure 27. Notice that class Parent is public so it cannot be confined. Intra-procedural analysis would not reveal that the expression `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.

```
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
```

Fig. 28. Anonymity need not be preserved in all subtypes.

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. Figure 28 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 29, which is a minimal variation of Figure 27 (Parent is no longer public). 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
}  }
```

Fig. 29. Two confined classes.

Noble, Vitek, and Potter [36] presented flexible alias protection 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. Hogg's Islands [28] and Almeida's Balloons [2,3] have similar aims. An Island or Balloon is an owner object that protects its internal representation from aliasing. The main difference from [36] 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 [14] 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 [31] proposed an informal extension of the Eiffel programming language with ownership annotations that are tracked and monitored at run-time. Barnett et al. [5] used a simple notion of ownership as the basis for an approach to specifying and checking properties stated as pre- and post-conditions for methods and object invariants; in their system the checking of ownership is itself a proof obligation. Also Müller [34] used ownership in support of verification but in this case checked by a type system. In the field of static program analysis, a number of techniques have been developed. Static escape analyses such as the ones proposed by Blanchet [6,7] and others [8,22] 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.

Clarke, Potter, and Noble [16,18] formalized representation containment by means of ownership types. Their seminal paper has sparked much interest and many papers have explored ownership types since then. Ownership types enforce that all paths from the root of an object system must pass through an object's owner. The paper of Clarke, Potter, and Noble [18] allowed just three annotations, `rep`, `norep`, and `owner` for specifying ownership, while later papers have introduced additional or alternative annotations [1,17,32,35]. Ownership types are inherently more flexible than confined types, while experiments with inferring ownership types, for example using the approch of Aldrich, Kostadinov, and Chambers [1] indicate that confined types lead to more scalable inference. Ownership types have been used as the basis for specifying a variety of properties via types, such as the absence of data races and deadlocks [11,12].

Most of the approaches mentioned above use operational semantics to reason about alias protection and ownership. Banerjee and Naumann [4] used denotational semantics to prove a representation-independence theorem, that is, a result about whether a class can safely be replaced by another class, independently of the program in which the class occurs. They use a syntactic notion of confinement, like we do, in which the protection domain is an instance rather than a package. Their notion of confinement is more restrictive than ours and it leads to a powerful theorem about classes.

10 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 number of confined types found by the analysis are surprisingly high, about 24% 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://ovmj.org/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.

References

1. Jonathan Aldrich, Valentin Kostadinov, and Craig Chambers. Alias annotations for program understanding. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Appplications (OOPSLA)*, pages 311–330, November 2002.
2. Paulo Sérgio Almeida. Balloon types: Controlling sharing of state in data types. In Mehmet Aksit and Satoshi Matsuoka, editors, *ECOOP'97—Object-Oriented Programming, 11th European Conference*, volume 1241 of *LNCS*, pages 32–59, Jyväskylä, Finland, 9–13 June 1997. Springer-Verlag.
3. Paulo Sérgio Almeida. Type-checking balloon types. *Electrical Notes in Theoretical Computer Science*, 20, 1999.
4. Anindya Banerjee and David A. Naumann. Representation independence, confinement and access control. In *Proceedings of POPL'02, SIGPLAN–SIGACT Symposium on Principles of Programming Languages*, pages 166–177, 2002.
5. Michael Barnett, Robert DeLine, Manuel Fähndrich, K. Rustan M. Leino, and Wolfram Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, 2004. Preliminary version in Proceedings of Fifth Workshop on Formal Techniques for Java-like Programs, 2003.
6. 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.
7. Bruno Blanchet. Escape analysis for Java: Theory and practice. *ACM Transactions on Programming Languages and Systems*, 25(6):713–775, 2003.
8. 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.
9. Boris Bokowski. CoffeeStrainer: Statically-checked constraints on the definition and use of types in Java. In *Proceedings of ESEC/FSE'99*, pages 355–374, Toulouse, France, September 1999.
10. Boris Bokowski and Jan Vitek. Confined types. In *Proceedings 14th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'99)*, pages 82–96, Denver, Colorado, USA, November 1999.
11. Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Appplications (OOPSLA)*, pages 211–230, November 2002.
12. Chandrasekhar Boyapati, Alexandru Salcianu, William Beebee, and Martin Rinard. Ownership types for safe region-based memory management in real-time Java. In *ACM Conference on Programming Language Design and Implementation*, pages 324–337, June 2003.
13. John Boyland. Alias burying: Unique variables without destructive reads. *Software—Practice and Experience*, 31(6):533–553, 2001.
14. 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 2072 in Lecture Notes in Computer Science, pages 2–27, Berlin, Heidelberg, New York, 2001. Springer.
15. Dave Clarke, Michael Richmond, and James Noble. Saving the world from bad Beans: Deployment-time confinement checking. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Appplications (OOPSLA)*, pages 374–387, Anaheim, CA, November 2003.
16. David Clarke. *Object Ownership and Containment*. PhD thesis, School of Computer Science and Engineering, University of New South Wales, Sydney, Australia, 2001.
17. David Clarke and Tobias Wrigstad. External uniqueness. In *10th Workshop on Foundations of Object-Oriented Languages (FOOL)*, New Orleans, LA, January 2003.
18. 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.
19. Ian Clarke, Scott G. Miller, Theodore W. Hong, Oskar Sandberg, and Brandon Wiley. Protecting free expression online with freenet. *IEEE Internet Computing*, 6(1):40–49, 2002.
20. Ian Clarke, Oscar Sandberg, Brandon Wiley, and Theodore W. Hong. Freenet: A Distributed Anonymous Information Storage and Retrieval System. In *Workshop on Design Issues in Anonymity and Unobservability*, number 2009 in Lecture Notes in Computer Science, pages 46–66. Springer-Verlag, 2000.
21. David Detlefs, K. Rustan M. Leino, and Greg Nelson. Wrestling with rep exposure. Technical report, Digital Equipment Corporation Systems Research Center, 1996.

22. 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.
23. William F. Dowling and Jean H. Gallier. Linear-time algorithms for testing the satisfiability of propositional horn formulae. *Journal of Logic Progamming*, 1(3):267–84, October 1984.
24. Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.
25. Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, 1994.
26. 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, pages 194–214, Kyoto, Japan, March 1998. Springer Verlag.
27. Stephan Herrmann. Object Teams: Improving modularity for crosscutting collaborations. In *Objects, Components, Architectures, Services, and Applications for a Networked World*, number 2591 in Lecture Notes in Computer Science, pages 248–264. Springer-Verlag, 2003.
28. John Hogg. Islands: Aliasing Protection in Object-Oriented Languages. In *Proceedings of the OOPSLA '91 Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 271–285, November 1991. Published as ACM SIGPLAN Notices, volume 26, number 11.
29. John Hogg, Doug Lea, Alan Wills, Dennis de Champeaux, and Richard Holt. The Geneva convention on the treatment of object aliasing. *OOPS Messenger*, 3(2):271–285, April 1992.
30. Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, May 2001.
31. Stuart Kent and Ian Maung. Encapsulation and Aggregation. In *Proceedings of TOOLS PACIFIC 95 (TOOLS 18)*, pages 227–238. Prentice Hall, 1995.
32. K. Rustan M. Leino and Peter Müller. Object invariants in dynamic contexts. In *Proceedings of ECOOP'04, 16th European Conference on Object-Oriented Programming*, pages 491–516, 2004.
33. Sun Microsystems. Support for extensions and applications in the version 1.2 of the Java platform. http://java.sun.com/products/jdk/1.2/docs/guide/extensions/spec.html, 2000.
34. Peter Müller. *Modular Specification and Verification of Object-Oriented Programs*. PhD thesis, FernUniversität Hagen, 2001. Also as LNCS 2262, Springer-Verlag, 2002.
35. Peter Müller and Arnd Poetzsch-Heffter. Universes: A type system for controlling representation exposure. In A. Poetzsch-Heffter and J. Meyer, editors, *Programming Languages and Fundamentals of Programming*. Fernuniversität Hagen, 1999.
36. James Noble, Jan Vitek, and John Potter. Flexible alias protection. In Eric Jul, editor, *ECOOP'98—Object-Oriented Programming*, volume 1445 of *Lecture Notes In Computer Science*, pages 158–185, Berlin, Heidelberg, New York, July 1988. Springer-Verlag.
37. Alex Potanin, James Noble, Dave Clarke, and Robert Biddle. Featherweight generic confinement. In *Workshop on Foundations of Object-Oriented Languages*, 2004.
38. Christian Skalka and Scott F. Smith. Static use-based object confinement. *International Journal on Information Security*, 4(1-2):87–104, 2005. Preliminary version in Proceedings of Foundations of Computer Security, volume 02-12 of *DIKU technical reports*, pages 117–126.
39. Jan Vitek and Boris Bokowski. Confined types in Java. *Software Practice and Experience*, 31(6):507–532, 2001.
40. Ayal Zaks, Vitaly Feldman, and Nava Aizikowitz. Sealed calls in Java packages. In *OOPSLA '2000 Conference Proceedings*, ACM SIGPLAN Notices, pages 83–92. ACM, October 2000.
41. Tian Zhao, James Noble, and Jan Vitek. Scoped types for real-time Java. In *Proceedings of 25th IEEE Real-Time Systems Symposium*, pages 241–251, 2004.
42. Tian Zhao, Jens Palsberg, and Jan Vitek. Type-based confinement. *Journal of Functional Programming*, 16(1):83–128, 2006. Preliminary version, entitled "Lightweight confinement for Featherweight Java", in Proceedings of OOPSLA'03, ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, pages 135-148, Anaheim, California, October 2003.

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 $\mathsf{not\text{-}conf}(T)$ and A is $\mathsf{not\text{-}anon}(m)$.
– 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 $\mathsf{not\text{-}conf}(T')$ and C is $\mathsf{not\text{-}conf}(T)$.

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 component type of the array). This generates a constraint $C \Rightarrow A$ where C is $\mathsf{not\text{-}conf}(T')$ and A is $\mathsf{not\text{-}anon}(m)$ 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 $\mathsf{not\text{-}conf}(T')$ and C is $\mathsf{not\text{-}conf}(T)$.

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 $\mathsf{not\text{-}conf}(T)$ and A is $\mathsf{not\text{-}anon}(m)$.
– 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 $\mathsf{not\text{-}anon}(n)$ and A' is $\mathsf{not\text{-}anon}(m)$.
– 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 $\mathsf{not\text{-}anon}(n)$ and C is $\mathsf{not\text{-}conf}(T)$.
– 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 $\mathsf{not\text{-}conf}(T')$ and C is $\mathsf{not\text{-}conf}(T)$.

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 $\mathsf{not\text{-}conf}(\texttt{Throwable})$ and A is $\mathsf{not\text{-}anon}(m)$ with m being the current method. Because the condition $\mathsf{not\text{-}conf}(\texttt{Throwable})$ 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 ($\mathsf{not\text{-}conf}(\texttt{Throwable})$) and C' is $\mathsf{not\text{-}conf}(T')$.

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

Fig. 30. Statistics for the benchmarks. C is Confined, GC is Generic-Confined, CA is Confinable and GCA is Genrice-Confinable.

Benchmark	Classes			Pkgs	Opcodes
	All	Public	Inner		
Aglets	410	193	133	31	107846
AlgebraDB	161	130	9	9	51218
Bloat	282	150	127	20	84212
Denim	949	684	271	84	288140
Forte	6535	3053	3769	231	1123362
GFC	153	143	8	22	58003
GJ	338	202	189	14	105323
HyperJ	1007	862	70	29	211269
JAX	255	255	0	21	97932
JDK 1.1.8	1704	1423	29	90	917132
JDK 1.2.2	4338	2655	1365	133	958619
JDK 1.3.0	5438	3326	1780	177	1180406
JDK 1.3.1	7037	4569	2043	213	2010305
JPython	214	134	35	11	103094
JTB	158	150	1	8	48900
JTOpen	3022	1439	557	75	1048704
Jalapeno 1.1	994	730	132	29	255436
JavaSeal	75	56	19	18	34933
Kawa	443	438	100	13	68733
OVM	835	416	590	41	111161
Ozone	2442	1705	490	122	447984
Rhino	95	67	1	8	51752
SableCC	342	290	47	10	45621
Satin	938	559	455	70	194985
Schroeder	108	103	7	13	41422
Soot	721	302	79	9	65137
Symjpack	194	125	0	14	73465
Toba	762	327	79	14	98993
Tomcat	1271	916	221	105	286368
Voyager	5667	4430	1305	312	996077
Web Server	1024	787	52	76	370664
Xerces	622	508	125	45	233919
Zeus	604	517	74	42	180437
Total	49259	31741	14163	2120	11987191

Fig. 31. Statistics for the benchmarks

Benchmark	Conf	Confinable	GenConf	GenConfinable
Aglets	13	60	15	66
AlgebraDB	20	81	24	97
Bloat	10	29	17	39
Denim	65	187	71	211
Forte	306	1149	437	1346
GFC	5	58	5	58
GJ	27	51	27	52
HyperJ	32	193	38	212
JAX	0	99	0	104
JDK 1.1.8	71	712	96	744
JDK 1.2.2	527	1062	603	1173
JDK 1.3.0	581	1297	685	1476
JDK 1.3.1	756	2126	891	2344
JPython	40	90	45	107
JTB	4	8	4	8
JTOpen	438	1049	467	1113
Jalapeno 1.1	155	543	159	549
JavaSeal	1	14	2	17
Kawa	1	177	1	177
OVM	119	243	302	428
Ozone	93	754	221	920
Rhino	11	28	15	33
SableCC	3	24	5	28
Satin	48	206	52	218
Schroeder	0	6	1	7
Soot	45	90	47	92
Symjpack	8	53	10	89
Toba	53	102	55	104
Tomcat	65	377	109	448
Voyager	208	1268	295	1442
Web Server	51	255	72	301
Xerces	22	221	47	279
Zeus	20	237	38	278
Total	3804	12880	4862	14591

Fig. 32. Number of confined and confinable classes

Benchmark	Time (ms)		
	real	user	sys
Aglets	4979	4540	160
AlgebraDB	3009	2860	70
Bloat	3623	3530	90
Denim	9463	8020	300
Forte	37565	29870	1380
GFC	3284	3070	90
GJ	4245	3960	60
HyperJ	6711	6160	220
JAX	3790	3580	100
JDK 1.1.8	13103	11750	450
JDK 1.2.2	23463	19270	750
JDK 1.3.0	29336	25760	760
JDK 1.3.1	41304	39730	850
JPython	4107	3890	90
JTB	3009	2810	80
JTOpen	23950	21720	800
Jalapeno 1.1	6770	6270	230
JavaSeal	2685	2490	50
Kawa	3910	3440	180
OVM	6072	5270	200
Ozone	13245	11190	480
Rhino	3201	2920	70
SableCC	3470	3130	110
Satin	7955	6310	270
Schroeder	3270	2730	90
Soot	5622	5190	250
Symjpack	3559	3270	100
Toba	6020	5550	270
Tomcat	8918	7790	330
Voyager	34082	25960	1090
Web Server	9308	8060	250
Xerces	6038	5560	220
Zeus	5640	4960	150
Total	347567	303330	10670

Fig. 33. Time required for the analysis