5 Extended Example: Confined Types

This chapter, which is based on [Bokowski, Vitek 1999], presents a non-trivial example of using CoffeeStrainer in the context of building secure software. We present a set of static constraints that strengthen encapsulation in object-oriented programs and facilitate the implementation of secure systems. We introduce two new concepts: *confined types* to impose static scoping on dynamic object references and *anonymous methods* which do not reveal the identity of the current instance (`this`). Confined types protect objects from use by untrusted code, while anonymous methods allow reuse of standard classes.

In the next section, we set the stage by introducing the problem domain of writing secure software in general, and sketching the idea behind confined types. An overview of existing language-based security mechanisms is given in Section 5.2. These mechanisms are not sufficient. Section 5.3 details a well-known security defect in the Java Development Kit which is our motivating example. Anonymous methods are introduced in Section 5.4. While independent from confined types, they are essential to allow a traditional programming style and in particular code reuse. Confined types are presented in Section 5.5. Section 5.6 introduces a complete programming example with confined types. In Section 5.7, the implementation of confined types in terms of constraint methods is presented. Section 5.8 compares confined types to related approaches. Section 5.9 discusses design choices, implications on genericity, and benefits that confined types offer for other areas than security. Finally, Section 5.10 summarizes the benefits of confined types.

5.1 Introduction

Writing secure code is hard. The steady stream of security defects reported in production code attests to the difficulty of the task. Software systems, such as the Java virtual machine, that permit untrusted code to interact with authorized code make it more difficult to ensure security than more traditional systems because trust boundaries become thinner and fuzzier.

In object-oriented programming it is difficult to control the spread and sharing of object references. This pervasive aliasing makes it nearly impossible to know accurately who owns a given object, that is to say, which other objects have references to it [Landi 1992]. The lack of ownership information [Potter et al. 1998] imposes a defensive programming style: since every method may have been called by an adversary, appropriate security checks that verify the caller's authority must be performed at method entry.

This creates a tension between security and program efficiency. Placing dynamic security checks in the prologue of each method is not realistic for performance reasons. Instead, security conscientious environments offer dynamic security checks that can be called explicitly by the programmer [Gong 1997, Tardo, Valente 1996]. These checks are interspersed in the program logic and thus nothing short of full-fledged program verification can ensure that no check has been omitted somewhere with the potential of compromising the security of the entire system.

Reusability, namely inheritance, creates its own set of problems for security. Similar to the *inheritance anomaly* known from concurrent programming [Matsuoka, Yonezawa 1993, Löhr 1992, Briot et al. 1998], inheriting from classes that do not implement the same security policy may create security anomalies. Of course, from the point of view of the library designer, it is not possible to implement classes that are secure in all contexts. Even if one could do that, the overall performance of the library would be unacceptable in secure contexts.

To a large extent the problem for implementing security is one of defining interfaces between protection domains. In Java, one virtual machine may manage objects of many different protection domains — code loaded from different sources — but imposes no clear boundaries between these domains. So, from the security engineer's viewpoint, there is no well-identified place where to put security checks.

One solution to this dilemma is to separate objects that are internal to a protection domain from external objects. Internal objects implement the behavior of the application without concern to security, while external objects are the interface between protection domains and must implement the security policy. Such a separation of concerns simplifies the life of the application programmer as the core of the system can be written without security checks. Moreover, it improves security as a smaller set of classes, the interface objects, become the focal point for security analysis.

Current object-oriented languages do not provide the means to enforce such a distinction between objects. While access modifiers can restrict how certain object types are manipulated [Gosling et al. 1996] — by curtailing visibility of methods and fields — and also restrict the scope of types, object-oriented languages do not provide strong encapsulation [Noble et al. 1998]. They typically cannot control the scope of object references. Thus references to sensitive parts of an application may leak to other protection domains.

We propose *confined types* as an aid for writing secure code. Confined types are meant to be used for preventing internal objects from escaping their protection domain, which we choose to be a single Java `package`. Packages are well suited for this task as they group related classes and they already provide basic access control features. Confinement is defined as follows: a type is said to be confined in a package if and only if all references to instances of that type originate from objects of the package. Confined types differ from existing access control features in that they prevent references to instances from leaving a domain rather than restricting access on a class level. In effect, confined types enforce static scoping of dynamic object references. Confined types are declared by inheriting from an empty marker interface, and anonymous methods are marked using a special Javadoc comment. Our annotations do not affect program semantics, thus a valid program with confinement annotations behaves identically to the same program with no annotations. Based on these annotations, we enforce some restrictions. While certain programming tasks may be clumsier, we argue that these restrictions are mild and that reasoning about security is much simpler.

5.2 Security in programming languages

Security is increasingly becoming a software issue as the mechanisms used to implement security policies are cheaper and more flexible in software than in hardware [Chase et al. 1993, Lucco et al. 1995, Grimm, Bershad 1997].

In a computer system, *principals* are the entities whose actions must be controlled. Principals invoke operations on objects[1]. The context within which a principal executes is called a *protection domain*. Access to resources within the same protection domain is not checked, while cross-domain operations must be authorized by a *security policy*.

Implementing security policies at the programming language level is reasonable. Language semantics can help to reason about program behavior and thus to prove security properties. Type systems and static analysis algorithms can reduce the run-time cost of security. Finally, protection domains can be made extremely lightweight and allow fine-grained interactions.

Usually, safety at the programming language level relies on *access control*. Access control mechanisms entail security checks before any potentially dangerous operations to verify that the current program has the authority to perform that action. Schemes such as *capabilities* and *access control lists* have been used to implement access control. A good example of the use of access control lists is the Unix file system.

Static access control: Object-oriented languages provide two basic means for controlling access to objects. The first is access modifiers such as the Java `private`, `public`, and `protected` modifiers that restrict the visibility of attributes and classes. The second is type abstraction; subtyping can be used to limit the operations that can be invoked on an object [Riecke, Stone 1998]. In Java, type abstraction is not useful since using the `instanceof` operator and reflection make it quite easy to retrieve the type of an object, which is not the case in some other systems [Leroy, Rouaix 1998, Riecke, Stone 1998].

Dynamic access control: Java provides dynamic access control mechanisms based on call stack inspection. That is, a dynamic check verifies that the current method was invoked (transitively) by a method with appropriate privileges [Gong 1997]. Another dynamic scheme is to use objects as capabilities [Levy 1984], this can be done as proposed in [Gong 1998] by interposing a restricted proxy object between the user and the target (see also [Hagimont et al. 1996, Vitek, Bryce 1999, Wallach et al. 1997]).

To sum up, the protection mechanisms that have been proposed so far are not perfect. On the one hand, dynamic checks are error-prone as it is easy to forget one check and there is no guarantee that all *potentially* dangerous operations that can be invoked by untrusted code are protected by access checks. On the other hand, static protection mechanisms are weak and were originally conceived for software engineering purposes rather than for security. We now turn to an example to demonstrate the problem which we want to address.

[1]Here, the notion of object is more general than in object-oriented programming. In the security literature an object may be a datum, a file, a hardware device, etc.

5.3 The class signing example

In Java, each class object (instance of class `Class`) stores a list of signers, which contains references to objects of type `java.security.Identity`, representing the principals under whose authority the class acts. This list is used by the security architecture to determine the access rights of the class at runtime. A serious security breach was found in the JDK 1.1.1 implementation which allowed untrusted code to acquire extended access rights [Secure Internet Programming Group 1997]. The breach was due to a reference to the internal list of signers leaking out of the implementation of the security package into an untrusted applet.

The scenario is as follows. Assume a malicious applet loaded from the net. Without any trusted signatures, its access rights are strongly limited. But the JDK does allow the applet to get its own list of signers. Furthermore, it can find out about all the principals known to the system by calling a method of `java.security.IdentityScope`. The method that returned the list of signers of a class (implemented by a Java array object) accidentally returned a reference to the system's internal array. Since arrays are mutable data structures, the applet can then proceed to update the array to include signatures of principals known to the system and obtain access rights it should not have, thus opening the system to more serious attacks.

We first present a program fragment which exhibits the security problem described above, and then give a solution using confined types.

5.3.1 The security breach in detail

In Figure 5.1, the array `signers` is the system's internal array that contains references to instances of the class `Identity` (the principals). Modifying this array is definitely a dangerous operation but there are no provisions in the implementation of the array class for checking the authority of the caller in an update. The security breach is caused by the `getSigners()` method which returns a reference to the `signers` object.

```
package java.lang;

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

Figure 5.1: Signatures without confined types

The attacker need only call `getSigners()` to be able to freely update the system's signature array. A simple fix is to return a shallow copy of the internal array. Figure 5.2 makes the copy

```
package java.lang;

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

Figure 5.2: An ad-hoc fix of the security problem

explicit. While this solves the particular problem, nothing guarantees that similar defects are not present in other parts of the package.

What is interesting about this example is that none of the standard Java protection mechanisms seem to help. Access modifiers and type abstraction are not relevant here. Restricting the use of the `Identity` objects would do no good as the attack does not interact with `Identity` objects, it only needs to acquire references to them and copy those references. Information flow control [Volpano, Smith 1997] does not apply either, since we do want to allow applets to read the signature information and to see identities known to the system. Finally, inserting dynamic checks in the array update operation, which is the point where the security policy is actually broken, is unrealistic as *all* array updates performed in the JVM would incur the cost of a dynamic check.

We now give a solution that guarantees that none of the key data structures used in code signing escape the scope of their defining package.

5.3.2 Class signing with confined types

To prevent software defects such as the one outlined above, we propose to ensure that references to identity objects are confined to the `java.security` package. This is achieved by renaming the `Identity` class as `SecureIdentity` and declaring it *confined*, using the empty marker interface `ConfinedType` that contains appropriate static constraints. Intuitively, the meaning of confinement is that references to instances of a confined class, or to instances of any of its subclasses, cannot be disclosed to, or accessed by, other packages. That is to say, only the classes defined in package `java.security` may interact with `SecureIdentity` objects. In order to preserve the functionality of the original interface, we define a new class `Identity` which can be seen outside of the `security` package. This class implements the public methods of `SecureIdentity` and has a private reference to a `SecureIdentity` instance. `Identity` plays the role of a guard and encapsulates the real identity object [Gong 1998, Hagimont et al. 1996]. The `Identity` class is purely for external use, it is neither a subclass nor a superclass of `SecureIdentity` and thus cannot be confused with a `SecureIdentity` object within the `security` package. Any attempt to return a `SecureIdentity` object to an outside package will be caught at compile-time as a violation of confinement. Figure 5.3 outlines our solution.

```
class SecureIdentity implements ConfinedType
{
  ...
  // the original Identity implementation ...
}

public class Identity {

  SecureIdentity target;

  Identity(SecureIdentity t) {
    target = t;
  }

  ...
  // public operations on identities;
}

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

Figure 5.3: Signatures with confined types

The `getSigners` method is similar to Figure 5.1. The important difference is that the type of the internal array `signers` is different from the type of the array that is being returned. The confinement constraints extend to arrays, thus if a type `A` is confined, then the array type `A[]` is confined as well. The `getSigners` method allocates an unconfined array to which newly created objects of type `Identity` are copied. If `getSigners` tried to return its internal array, a confinement breach error would be signaled.

This solution preserves the functionality of the original program, in fact outside code need not be aware of the existence of confined types. But from a security engineering point of view, attention is directed to the `Identity` class as it can be accessed by untrusted components, and may thus (if deemed necessary) include dynamic security checks.

This example shows that confined types help in developing secure code as they draw a strong demarcation line between internal representation objects and external interface objects. We now define confinement more precisely, starting with anonymous methods which are essential for a usable notion of confinement since they allow confined types to inherit methods defined in unconfined supertypes.

5.4 Anonymous methods

An *anonymous method* is a method that does not reveal the current instance's identity to others which means it does not introduce new aliases to the current instance, nor perform any identity-dependent operations. Anonymous methods are needed for confined types; however, they have interesting properties in their own right and may be useful in other contexts [Boyland 1999].

This section defines anonymous methods and presents static constraints that are required to ensure that methods are anonymous. The actual constraint methods implementing these constraints will be explained in Section 5.7.

In Java technical terms, an anonymous method is an instance method that may use `this` *only* for accessing the fields of the current instance and for calling other anonymous methods on itself. Thus, the anonymous method keeps its implicit `this` parameter secret by not assigning `this` to a variable, nor providing `this` as a method argument, nor returning `this` as the method's return value. Additionally, is not allowed to perform reference comparisons using `this`[2].

```
class Example implements HasAnonymousMethods {

  int count;
  int /**@anon*/ ok(A arg) {
1   alsoOk(arg.foo());
2   return count ;
  }
  void /**@anon*/ alsoOk(int i) {
3   count = i + count ;
  }
  Example notOk(A arg) {
4   arg.bar(this);
5   arg.o = this;
6   notOk(arg);
7   if(this == arg) ...
8   return this;
  }
}
```

Figure 5.4: Examples for anonymous methods

Figure 5.4 presents a valid class `Example` with two anonymous methods (`ok`, `alsoOk`) and a non-anonymous method (`notOk`). Lines (1 - 3) show examples of anonymity-preserving code, while (4 - 8) show examples that do not preserve anonymity. Line (4) reveals this to

[2]As a rule of thumb, the keyword `this` should not be used at all in anonymous methods, except to access fields hidden by a parameter or local variable of the same name.

method `bar`. (5) stores `this` in a field of `arg`. Line (6) calls a non-anonymous method (don't mind the infinite recursion). Line (7) uses `this` for reference comparison. Finally, (8) returns `this`.

Because the definition of anonymous methods is recursive, we require anonymous methods to be declared as such explicitly, and check for each such declared method whether it conforms to the definition of anonymity. To declare a method anonymous, the tagging Javadoc comment `/**@anon*/` is used, and the containing class or interface must be a subtype of `HasAnonymousMethods`. In addition to the constraint regarding the use of `this`, there is another constraint regarding anonymity of overridden methods: anonymity is a property that potential callers rely on, methods in subclasses that override an anonymous method must therefore be anonymous as well.

We regard constructors as a special case of instance methods. Accordingly, constructors may be declared anonymous as well, and the same constraints that apply to instance methods apply to constructors. In Java, the first statement of each constructor is a call to another constructor, which may be in the same class, or in the direct superclass of the current class. Without an explicit call, the constructor of the superclass is called implicitly. An anonymous constructor must thus ensure that explicit and implicit calls are made only to anonymous constructors. The `Object` constructor, the only one that does not call another constructor, is anonymous by definition, as are several other commonly used methods in Object: `wait`, `notify`, `notifyAll`, and `finalize`. The method `hashCode`, which by default returns the object's address in the heap, reveals the object's identity. It is possible, however, to implement `hashCode` without using the object's address, which would allow it to be declared confined as well.

The following list summarizes the constraints that apply to anonymous methods and constructors:

A1 The reference `this` can only be used for accessing fields and calling anonymous methods of the current instance.

A2 Anonymity declarations must be preserved when overriding methods.

A3 The constructor called from an anonymous constructor must be anonymous as well.

Clearly some programming styles are restricted with anonymous methods. It is important to assess how restrictive our proposal actually is and whether common programming idioms would become too cumbersome to be practical or too inefficient. For instance, the visitor pattern breaks anonymity to implement a double dispatching [Gamma et al. 1995]. We have mentioned that the default implementation of `hashCode` must be changed[3], which comes at a price in runtime performance that remains to be evaluated. The use of the reference equality operator is restricted as well, instead value comparison must be used. But changing code from reference semantics to value semantics has deep implications [Lopez et al. 1994] and is not as efficient[4].

[3]An anonymous `hashCode` method is an advantage for persistence and for JVM implementers as objects can be freely moved around the store and between main memory and secondary storage without affecting code that relies on hash codes.

[4]The inefficiency could be somewhat mitigated by program analysis. The following code fragment does not violate anonymity because reference equality implies value equality:

```
if (this == that) return true; else return equals(that);
```

To obtain a better sense of the impact of anonymity declarations on programming style, we analyzed `java.util` and `java.awt`, two representative packages of the Java development kit (JDK 1.1) to find out how many existing methods meet the above mentioned criteria (**A1**, **A2**, and **A3**). The data was collected by iterating a static analysis detecting anonymity violations. In each iteration, methods flagged by the analysis were declared as `non-anon`. The process was repeated until the fixpoint was reached. The results, summarized in Figure 5.5, are encouraging. Without changes to existing code, between 83% and 94% of the methods are already anonymous. With some care a portion of those `non-anon` methods could be re-written to become anonymous.

Package	`java.util`	`java.awt`
classes + interfaces	28 + 3	63 + 7
all methods	351	1246
anon methods	330 (94%)	1042 (83%)

Figure 5.5: Anonymous methods in existing code.

Anonymous methods are closely related to the concept of borrowed receiver presented by Boyland in [Boyland 1999], where the goal is to to implement unique variables without destructive reads. His proposals also allows arguments to be borrowed. One important difference with anonymous methods is that Boyland allows the use of reference equality and `hashCode`. The definition of anonymous methods could be adapted to allow this as well. On the other hand, with the current definition, an object which has only anonymous methods can be moved in memory without affecting the semantics of the program. Section 5.5.3 explains our use of anonymous methods. We now turn to the definition of confined types.

5.5 Confined types

In this section, we describe the concept of confined types and the constraints that ensure the required properties of confined types. The actual constraint methods that implement the constraints listed in this section will be explained in Section 5.7.

A *confined type* is a type whose instances may not be referenced or accessed from outside a certain protection domain. Confined types are introduced by annotating class or interface definitions with the keyword `confined`. Instances of confined types are called *confined objects*. In Java, packages are an obvious choice of protection domains as packages have already some protection mechanism built into the language in the form of access modifiers. Instances of confined classes may thus only be referenced or accessed from within a single package. Since confined objects cannot be referenced from outside their confined class' package, we can unambiguously refer to a confined object's *confining package*, meaning the package in which the object's class is defined *and* the package in which all the code that can potentially manipulate the object is located. We can also refer to the package of a confined type since all classes (or interfaces) that extend (implement) a confined class (interface) must belong to the same package.

Figure 5.6 summarizes the relationships between an object `obj` in package `outside` and the objects `conf` and `unconf` from package `inside`. A reference from `obj` to the confined object is not allowed, but all other references, including from `conf` to objects outside of the package are.

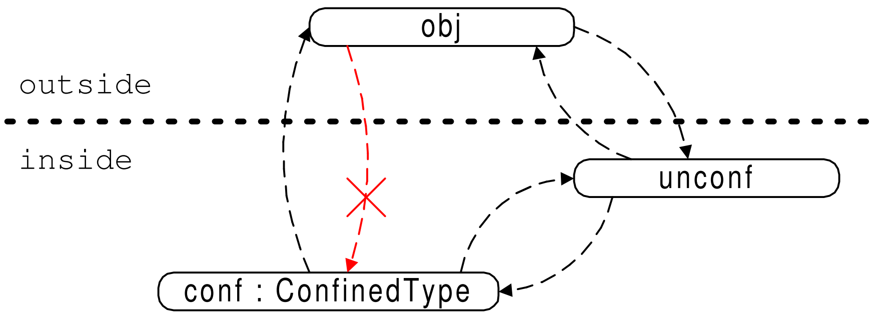

Figure 5.6: References between objects in inside and outside packages.

It is important to understand that we are not trying to prevent information to leak through covert channels [Lampson 1973], just stop references to confined objects from being transferred out of their confining package.

Overview of the problem

To be able to define constraints that ensure confinement, we must first analyze all possible data flows with which object references may be transferred from one package to another. Without loss of generality, we only consider data flows with which an object reference can be transferred from a package `inside` to a package `outside`. The analysis has two parts: In the first part, we assume that the control flow is in package `inside`, which actively hands out the reference; in the second part, we assume the control flow to be in package `outside`, which tries to retrieve a reference from package `inside`.

We start with reference transfers that originate from package `inside`. The possible targets in package `outside` fall into three categories: fields, method and constructor parameters (including the implicit parameter `this`), and parameters of catch clauses. Taking into account that object references can be stored in arrays, we distinguish six cases for transfers from the inside:

r1 Package `inside` assigns a reference to one of its objects to a field in package `outside`

r2 Package `inside` calls a method or constructor defined in package `outside` passing a reference to one of its objects as an argument,

r3 Package `inside` wraps an object reference into an array (or multiple nested arrays) and uses points r1 or r2 for transferring the array reference,

r4 Calling a method or constructor defined in a class in package `outside` from a subclass of that class in package `inside` (the implicit parameter this is transferred),

r5 Calling a method defined in a class in package `outside` from a superclass of that class in package `inside` (the implicit parameter `this` is transferred),

r6 Package `inside` throws an exception which is handled by a `catch` clause defined in package `outside` (the exception object is transferred).

We now list reference transfers that originate in package `outside`. The possible sources in package `inside` fall into three categories: fields, method return values, and references to newly instantiated objects using the operator `new`. Again, taking into account that object references can be stored in arrays, we distinguish four cases for reference transfers originating in package `outside`:

r7 Package `outside` reads a field of package `inside` containing a reference to an instance of a class defined in package `inside`,

r8 Package `outside` calls a method of package `inside` that returns an object reference to an instance of a class defined in package `inside`,

r9 Package `outside` uses points r7 or r8 to obtain a reference to an array (or multiple nested arrays), into which package `inside` has wrapped an object reference,

r10 Package `outside` instantiates an object of a class defined in package `inside` using the `new` operator.

These points are illustrated in Figures 5.7 (package `inside`) and 5.8 (package `outside`). Each line labeled **`r1`** to **`r10`** demonstrates a reference transfer.

We now introduce the constraints that prevent reference transfers. The presentation proceeds as follows: Section 5.5.1 gives constraints on class and interface declarations. Section 5.5.2 presents constraints that prevent widening. Section 5.5.3 discusses constraints that deal with hidden widening. Based on the constraints introduced so far, Section 5.5.4 explains why reference transfers originating in the inside package cannot occur. Finally, Section 5.5.5 presents the remaining constraints that address reference transfers originating in outside packages.

The constraints can be checked statically and are implemented and enforced using CoffeeStrainer. The actual constraint methods are described in Section 5.7. It is important to note that a main design goal for the constraints was that only the classes of the confining package should have to be checked. Other packages may remain unchecked, with the exception of anonymous methods, because the standard Java access control checks are sufficient to protect packages with confined types from other packages. We assume for this that packages that have been checked can either be sealed and protected from extension by untrusted code (using digital signatures and a class loader checking the signatures), or that the constraints on confining packages are re-checked at load-time.

5.5.1 Confinement in declarations

The first two constraints restrict the declaration of classes and interfaces. The goal is to ensure that confined types are only visible in their package and to guarantee that subtyping preserves confinement.

```
package inside;

public class C extends outside.B {

    void putReferences() {
      C c = new C();
r1    outside.B.c1 = c;
r2    outside.B.storeReference(c);
r3    outside.B.c3s = new C[] fcg;
r4    calledByConfined();
r5    implementedInSubclass();
r6    throw new E();
    }
    void implementedInSubclass() {
    }
r7  static C f = new C();
r8  static void C m()
    {
      return new C();
    }
r9  static C[] fs = new C[]{new C()};
r10 public C() { }
}

public class E extends RuntimeException {
}
```

Figure 5.7: Transferring references, package `inside`

```
package outside;

public class B {

r1  static inside.C c1;
r2  static void storeReference(inside.C c2) {
      // store c2
    }
r3  static inside.C[] c3s;
r4  void calledByConfined() {
      // store this
    }
    static void getReferences() {
r7    inside.C c7 = inside.C.f;
r8    inside.C c8 = inside.C.m();
r9    inside.C[] c9s = inside.C.fs;
r10   inside.C c10 = new inside.C();
      D d = new D();
      try {
        d.putReferences();
r6    } catch (inside.E ex) {
        // store ex
      }
    }
}

class D extends inside.C {

r5   void implementedInSubclass() {
       // store this
     }
}
```

Figure 5.8: Transferring references, package `outside`

C1 A confined class or interface must not be declared `public` or `protected`, and must not belong to the unnamed global package.

C2 Subtypes of a confined type must be confined in the same package as their confined supertype.

C1 ensures that confined types have `private` or package-local access. Confined types cannot belong to the unnamed global package, as this package is "open" to extensions. C2 guarantees that if a confined class (or interface) is extended (implemented) then the extending class (interface) is also confined and belongs to the same package. Thus, the confinement property extends transitively to all subtypes of a confined type.

5.5.2 Preventing widening

To prevent references to confined objects from escaping their package, reference widening from a confined type to an unconfined supertype cannot be allowed. Clearly, the root of the type hierarchy, `java.lang.Object`, is not confined. Thus, if a confined reference can be widened and stored in a variable of type `Object`, then the confined object may leak out of its package.

In Java, reference widening may occur in either of:

- an assignment, if the declared type of the left hand side of the assignment is a supertype of the assigned expression's static type,
- a method call, if the declared type of a parameter is a supertype of the corresponding argument expression's static type,
- a `return` statement, if the declared result type of the method is a supertype of the result expression's static type,
- a cast expression, if the type casted to is a supertype of the casted expression's static type.

Widening must be prevented if it entails losing the confinement property of an object reference. The following constraint enforces confinement.

C3 Widening of references from a confined type to an unconfined type in assignments, method call arguments, `return` statements, and explicit casts is forbidden.

As noted in Section 3, Java arrays are a way to leak references as well. Consequently, the constraint takes arrays into account as well. For a confined type `A`, we regard the array type `A[]` to be a confined type as well, called a *confined array type*, so that they are a special case of C3.

In general, confined objects may not be stored in unconfined collections (of which arrays are just one example). Although this restricts common programming styles, the signed classes example showed that it is exactly this kind of potential leakage which is easy to overlook. Thus, we think it is worth the effort to provide special-purpose confined collections (or arrays) rather than trading security for the reuse of collection classes. Section 5.9.1 discusses the impact of confined types on genericity.

5.5.3 Preventing hidden widening

In addition to the obvious widening of the previous section, implicit or *hidden* widening occurs whenever a method inherited from an unconfined superclass is invoked on a confined object. Upon entry in the inherited method the implicit parameter `this` which refers to current instance is widened from the confined type to the unconfined supertype.

Clearly, hidden widenings should not be ruled out completely, as this would make it impossible to derive confined classes from non-trivial unconfined classes. But allowing confined classes to extend unconfined classes without restrictions is dangerous. The reference to the current instance may leak out if a method in the superclass transfers it to any other object. However, anonymous methods of Section 5.4 are safe since they do not leak `this`. We can now give the constraints that ensure the safety of hidden widenings. We say that methods *defined* by a class are the new methods introduced in that class, all other methods are *inherited*. Note that interfaces do not play a role here since they do not introduce code.

C4 Methods invoked on a confined object must either be defined in a confined class or be anonymous methods.

C5 Constructors called from the constructors of a confined class must either be defined by a confined class or be anonymous constructors.

Constraint C4 ensures that methods called on a confined reference are either defined in a confined class or anonymous. In the case of overridden methods, i.e., if a method defined in a superclass is overridden in a confined subclass, it is safe to execute the method as it preserves confinement. Similar to methods, constructors of unconfined superclasses that are called by the constructors of a confined class need to be anonymous. This applies to instance field initializers and instance initialization blocks as well, as these might also leak out a reference to the object.

We should emphasize that these constraints need only be checked within the defining package of the confined type as it is not possible to invoke methods of confined types of another package. Also, note that methods and constructors defined by confined classes need not be anonymous.

The role of anonymous methods is to allow code reuse by easing the restrictions that would otherwise be imposed on inheritance. Without them, it would be unsafe to invoke any inherited method of a confined object. Thus anonymous methods are used only in ordinary classes.

5.5.4 Preventing transfer from the inside

In the list of possible reference transfers from the inside package to an outside package, items r1 to r6 involve transfers that originate in the inside package.

Based on the constraints introduced so far, items r1 and r2 – assigning to a field in an outside package, and passing parameters to a method in an outside package – are not allowed for confined types. Since neither a confined type itself nor one of its subtypes is accessible from the outside package (due to constraints C1 and C2), the type of the field or parameter can only be an unconfined supertype of the confined type. But then, transferring the reference would require reference widening which is ruled out by constraint C3.

Similarly, item r3 – wrapping references to confined objects in an array and transferring the array reference by assigning it to a field or passing it as a parameter – is not possible, because arrays of confined types are confined as well.

Reference transfers according to item r4 – calling a method in an unconfined supertype – are not ruled out completely; rather, constraints C4 and C5 require the called methods (resp. constructors) to be anonymous, as discussed in Section 4. Thus, it is possible to transfer references, but only to code that neither discloses the reference to a non-anonymous method nor depends on the reference.

Item r5 – transferring `this` to a subclass by calling a method which is implemented in the subclass – cannot transfer a confined reference to an outside package, because constraints C1 and C2 make sure that all subclasses of a confined type must reside in the same package as the confined type.

With Java exceptions, there is another opportunity for transferring references: If an exception of a certain type is thrown, it may be caught with a catch clause whose formal parameter is of a supertype of the actual exception that was thrown. As we don't see important uses where exception objects should be confined to a package, we just disallow subtypes of `java.lang.Throwable` to be confined types, thus disallowing reference transfers according to item r6. We require that:

C6 Subtypes of `java.lang.Throwable` may not be confined.

5.5.5 Preventing transfer from the outside

Reference transfers from package `inside` to package `outside` (r7 – r10) which involve transfers that originate in package `outside` have not yet been addressed. The new constraints are:

C7 The declared type of public and protected fields may not be confined.

C8 The return type of public and protected methods may not be confined.

If a field's declared type is a confined type, it should not be accessible from outside the package, i.e., no field whose declared type is confined may be `public` or `protected` (C7), preventing object reference transfer according to item r7. The reason for this constraint is that although the confined type itself is not accessible from outside the package, a `public` or `protected` field of that type may exist in package `inside`, which allows package-external access to references stored in the field. Although the confined type itself is not accessible in outside packages, such references can still be used through unconfined supertypes of the confined type.

By similar reasoning, methods which return confined type should not be accessible from outside the package, i.e., no method returning a confined type should be `public` or `protected` (C8). Thus, item r8 is prevented as well. Again, note that confined array types are a special case of the general constraint, so fields of confined array types and methods returning confined arrays must have private or package-local access, preventing r9. Instantiating a confined class from outside (item r10) cannot occur because confined classes are not accessible from outside.

The constraints that have been specified in this and the previous section can be implemented using static constraint methods. The actual implementation will be explained in Section 5.7.

5.6 Example: public-key cryptography

This section gives an example of how confined types can be used in practice to build secure software. Using this example, it can be seen what implications follow from the constraints specified in the previous two sections.

The domain chosen for the example, public-key cryptography, is one of the essential tools for security in distributed systems. The basic idea of public-key cryptography is to employ an asymmetric scheme where a pair of keys – one private, the other public – is used to encrypt and decrypt information in a way that a message encrypted by the private key can only be decrypted using the public key and vice versa. Two important uses are, first, encryption of messages by a sender who uses the public key of the recipient, making sure that only the recipient can decode the message using his private key, and second, signing a message by a sender who uses his own private key, allowing recipients to verify the signature by successful decryption using the sender's public key. The security of this scheme hinges on the property that the private key practically cannot be derived from the public key, and that the private key is never disclosed.

Therefore, implementations of public key cryptography must be secure. In this section, we present an example implementation of the RSA algorithm [Rivest et al. 1978] which is reusable in different contexts without endangering security. Another goal is to ensure that the random number objects used in the generation of public-private key pairs should not be accessible outside of the implementation of the RSA algorithm because the generated keys could be guessed from the random number generator object's state. Finally, and most importantly, we would like to offer the guarantee to clients of the RSA package that the objects that represent their private keys remain confined to their application, and that under no circumstance other untrusted code be granted access to a private key.

The solution we present in this section uses confined types to achieve the desired security properties. It is noteworthy that the result is achieved with little effort on the part of the client (the users) of the RSA library. We structure the code in two packages:

- Package `rsa`: a reusable public-key cryptographic library.
- Package `secure`: one particular user of the rsa package.

The classes that we want to protect are `ConfinedRandom`, the random numbers used to generate keys, and `PrivKey`, the actual private keys. The first class belongs to the `rsa` implementation and the second is owned by the client of the library, the `secure` package. Thus `ConfinedRandom` is confined in package `rsa`, while `PrivKey` is confined in `secure`.

Public keys are implemented by the `Key` class and do not have to be confined as we assume that clients may want to pass them around to other packages. Of course, there could be another client package (even simultaneously on the same JVM) which confines its public key class.

The package `rsa`, Figure 5.10, provides a class `Key` that encapsulates RSA encryption. Class `KeyFactory` generates a key pair `(pub, priv)` such that a message encrypted with the public key can be decrypted using the private key and vice versa, i.e., `pub.crypt(priv.crypt (m))` returns `m`. The implementation of `KeyFactory` relies on class `ConfinedRandom` for generating the keys.

The package `secure`, Figure 5.11, introduces classes `PrivKeyFactory` and `PrivKey` to, respectively, generate and represent private keys. A class `Main` is given to demonstrate how keys are used. There are several other classes in the implementation, which we will detail in the following paragraphs. Figure 5.9 illustrates the relationships between the two packages. Full arrows indicate subtyping relations, and dashed arrows indicate implementation dependencies. Confined types are marked using gray boxes.

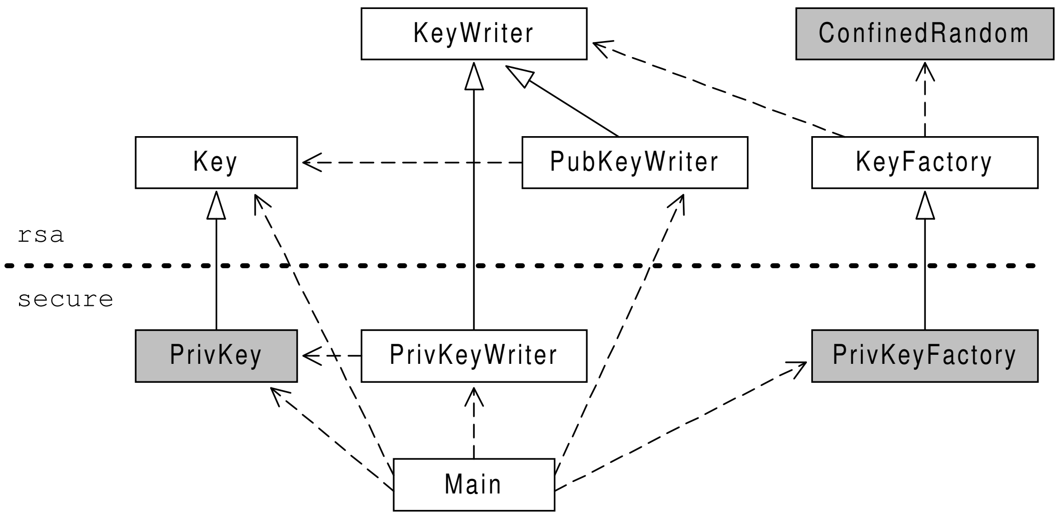

Figure 5.9: Inheritance and usage relationships between package `rsa` and package `secure`.

In class `Key`, the fields `mod` and `exp` are `public`. Although this allows to access sensitive information from the outside, an object reference is required to read the fields' values. The idea is to subclass `Key` in another package and to make this subclass confined. Accordingly, the method `crypt` is declared `anon` as otherwise this method could not be called on a confined object (C4).

Often confined types require only a trivial implementation, as can be seen in class `ConfinedRandom`. This is an example of making an unconfined class confined in another package by subclassing. The class `ConfinedRandom` is used in class `KeyFactory` for the field `randomGenerator`. This field is declared `private` so that only the class `KeyFactory` has to be checked by the programmer for potential leakage of a reference to the random generator object or leakage of its internal state.

The class `KeyFactory` does not set the internal values of `Key` objects directly. Rather, it uses

```
package rsa;

import java.math.BigDecimal;
import java.util.Random;

public class Key implements HasAnonymousMethods {
  public BigDecimal mod;
  public BigDecimal exp;

  /**@anon*/ public String crypt(String msg) {
    /* return (msg^^exp)%mod */
  }
}

private class ConfinedRandom extends Random implements ConfinedType { }

public interface KeyWriter extends HasAnonymousMethods {
  /**@anon*/ public void setValues(BigDecimal m, BigDecimal e);
}

public class KeyFactory implements HasAnonymousMethods {
  private ConfinedRandom randomGenerator =
    new ConfinedRandom(System.currentTimeMillis());

  /**@anon*/ public void genKeyPair(KeyWriter pub, KeyWriter priv) {
    // set internal values of both key objects,
    // using random generator...
  }
}

public class PubKeyWriter implements KeyWriter {
  private Key key;

  public PubKeyWriter(Key k) { key = k; }

  /**@anon*/ public void setValues(BigDecimal m, BigDecimal e) {
    key.mod = m;
    key.exp = e;
  }
}
```

Figure 5.10: Package containing RSA algorithm

```
package secure;

import rsa.*;
import java.math.BigDecimal;

class PrivKey extends Key implements ConfinedType { }

private class PrivKeyWriter implements KeyWriter {
  private PrivKey key;
  PrivKeyWriter(PrivKey k) { key = k; }
  /**@anon*/ public void setValues(BigDecimal m, BigDecimal e) {
    key.mod = m; key.exp = e;
  }
}

class PrivKeyFactory extends KeyFactory implements ConfinedType { }

public class Main {
  private static PrivKey privateKey = new PrivKey();
  public static Key publicKey = new Key();

  public static void main(String[] args) {
    PrivKeyFactory keyFactory = new PrivKeyFactory();
    keyFactory.genKeyPair(new PubKeyWriter(publicKey),
                          new PrivKeyWriter(privateKey));
    // use keys for encryption
    // and decryption...
  }
}
```

Figure 5.11: Confining a type in a different package

the interface KeyWriter which normally would not appear in a design without confined types. The reason for this is that both Key and KeyFactory will be subclassed and made confined in another package. If KeyFactory referenced Key directly, the confined subclass of Key could not be used with KeyFactory or a subclass of it because at some place a reference widening to the original type Key would be needed, which is forbidden by C3. Class PubKeyWriter trivially implements the interface KeyWriter.

Note also that PrivateKey does not define any new methods or fields. However, a new implementation of KeyWriter is needed for accessing the internal values of the confined type PrivKey. Due to constraint C3, which prevents widening from PrivKey to Key, the previously defined class PubKeyWriter cannot be used. The similarity of the new implementation PrivKeyWriter to PubKeyWriter suggests that genericity would help here; this is discussed in Section 5.9.1.

Similar to PrivKey, a confined subclass SecKeyFactory is derived from KeyFactory. The interesting point here is that the superclass has access, and uses, a confined class (namely ConfinedRandom), but our restrictions guarantee that these values can not be leaked to the subclass.

In class Main, a private and a public key object is created. Note that private or package-local access for field privateKey is required by C7, while publicKey can be public. In main, then, a SecFactory object is created and genKeyPair is invoked on it, providing two instances of PubKeyWriter and PrivKeyWriter, respectively.

5.7 Implementing confined types

Implementing the constraints defined for confined types (Section 5.5) and anonymous methods (5.4) is straightforward. In this section, we present and explain the constraint methods that make up the complete implementation of confined types and anonymous methods. Two empty marker interfaces containing constraint methods are needed: ConfinedType, which marks confined types, and HasAnonymousMethods, the interface that marks classes containing anonymous methods. We will start by explaining the constraint methods of ConfinedType.

The constraint methods of ConfinedType, all of them interface constraints, are shown in three figures. Figure 5.12 contains constraint methods implementing C1, C2, and C6, Figure 5.13 contains constraints methods which deal with widening (C3), and the remaining constraint methods implementing C4, C5, C7, and C8 are shown in Figure 5.14.

For convenience, a helper method isConfined is defined that returns true if its argument is a confined type. The definition constraint methods checkClass and checkInterface which are called for all confined types delegate the required checks to a common helper method checkUserType. This method implements two of the constraints for confined types (C1 and C6). Each assignment of a string value to the variable rationale can be read as a comment for the check that follows the assignment. The constraint C2, which requires that subtypes of confined types be confined as well, is not reflected in a constraint method because the type constraint methods in ConfinedType already apply to all subtypes of confined types.

```
package confined;

/**
 *@constraints
 * public static boolean isConfined(AType t) {
 *   return ((AReferenceType)t).isSubtypeOf(this);
 * }
 * private boolean checkUserType(AUserType ut) {
 *   rationale = "subtypes of java.lang.Throwable may not be confined (C6)";
 *   if(ut.isSubtypeOf(barat.Barat.getThrowableInterface())) return false;
 *   rationale = "a confined type must not be declared public or protected (C1)";
 *   if(ut.isPublic() || ut.isProtected()) return false;
 *   rationale = "confined types cannot be in the unnamed global package (C1)";
 *   return ut.qualifiedName().indexOf('.')==-1;
 * }
 * public boolean checkClass(Class c) {
 *   return checkUserType(c);
 * }
 * public boolean checkInterface(Interface c) {
 *   return checkUserType(c);
 * }
 * // no need to check C2 because constraints already apply to
 * // all subtypes of interface "ConfinedType"
 *
```

Figure 5.12: Implementation of ConfinedType (Part I – implementing C1, C2, C6)

```
* public boolean checkUseAtAssignmentOperand(Assignment a) {
*   rationale = "illegal assignment widening from confined to unconfined (C3)";
*   return isConfined(a.getLvalue().type());
* }
* public boolean checkUseAtFieldInitializer(Field f) {
*   rationale = "illegal initializer widening from confined to unconfined (C3)";
*   return isConfined(f.getType());
* }
* public boolean checkUseAtLocalVariableInitializer(LocalVariable v)
* {
*   rationale = "illegal initializer widening from confined to unconfined (C3)";
*   return isConfined(v.getType());
* }
* public boolean checkUseAtConditionalIfTrue(Conditional c) {
*   rationale = "illegal widening in conditional from confined to unconfined (C3)";
*   return isConfined(c.type());
* }
* public boolean checkUseAtConditionalIfFalse(Conditional c) {
*   rationale = "illegal widening in conditional from confined to unconfined (C3)";
*   return isConfined(c.type());
* }
* public boolean checkUseAtMethodCallParameter(int i, AMethodCall mc) {
*   rationale = "illegal argument widening from confined to unconfined (C3)";
*   return isConfined(mc.getCalledMethod().getParameters()
*                                         .get(index).getType());
* }
* public boolean checkUseAtCastOperand(Cast c) {
*   rationale = "illegal cast from confined to unconfined type (C3)";
*   return isConfined(c.type());
* }
```

Figure 5.13: Implementation of ConfinedType (Part II – implementing C3)

The usage constraint methods needed to implement C3 are shown in Figure 5.13. Essentially, each of them checks, whenever a confined type is used in a Java construct that may involve widening, that the potentially widened type is a confined type as well.

```
 *
 * public boolean checkUseAtInstanceMethodCall(InstanceMethodCall mc) {
 *   AMethod calledMethod = mc.getCalledMethod();
 *   rationale = "methods invoked on a confined object must either be "
 *                 + "defined in a confined type or be anonymous methods (C4)";
 *   return isConfined(calledMethod.containingUserType())
 *          || HasAnonymousMethods.isAnonymous(calledMethod);
 * }
 * public boolean checkConstructor(Constructor c) {
 *   rationale = "constructors called from the constructors of a confined class"
 *                 + "must either be anonymous or defined in a confined class (C5)";
 *   return isConfined(c.constructorCall().calledConstructor()
 *                         .containingUserType())
 *          || HasAnonymousMethods.isAnonymous(c.constructorCall()
 *                                              .calledConstructor());
 * }
 * public boolean checkUseAtField(Field f) {
 *   rationale = "public and protected fields may not be of confined type (C7)";
 *   return !(f.isProtected() || f.isPublic());
 * }
 * public boolean checkUseAtResult(AMethod m) {
 *   rationale = "public and protected methods may not return confined type (C8)";
 *   return !(m.isProtected() || m.isPublic());
 * }
 */
public interface ConfinedType { }
```

Figure 5.14: Implementation of ConfinedType (Part III - implementing C4, C5, C7, C8)

The first constraint method in Figure 5.14, the usage constraint method `checkUseAtInstanceMethodCall`, makes sure that methods called on confined objects are either defined in a confined type, or anonymous (C4). The method `isAnonymous` is defined as part of the constraint code for the interface `HasAnonymousMethods` and will be explained below. Similarly, the definition constraint method `checkConstructor` checks that constructors called from the constructors of confined types are either defined in confined superclasses, or anonymous (C5). Finally, the remaining two usage constraint methods check that methods with confined return types and fields of confined types are neither declared public nor protected (C7 and C8).

Figure 5.15 shows the constraint methods needed for anonymous methods. The implementation of the constraints for anonymous methods makes use of tags. The helper method `isAnonymous(m)` returns `true` if the method `m` has been tagged with the Javadoc comment `/**@anon*/` *and* it is contained in a subtype of the empty interface `HasAnonymousMethods`. This latter check is important because there are constraints that need to be checked for methods tagged with `/**@anon*/` and the corresponding constraint methods must be attached to a certain class or interface, in this case to the interface `HasAnonymousMeth-`

ods itself. Based on `isAnonymous`, it is straightforward to express the constraints A1–A3 for anonymous methods into the four interface definition constraint methods `checkThis`, `checkConcreteMethod`, `checkAbstractMethod`, and `checkConstructor`.

```
package confined;
/**
 *@constraints
 * public static boolean isAnonymous(AMethod m) {
 *   return m.containingUserType().isSubtypeOf(thisInterface)
 *          && m.hasTag("anon");
 * }
 *
 * public boolean checkThis(This t) {
 *   // non-anon methods need not be checked:
 *   if(!isAnonymous(t.containingMethod())) return true;
 *   rationale = "'this' may be used only for field access or calls of anon methods (A1)";
 *   Node n = (Node)t.container();
 *   if((n instanceof InstanceMethodCall) && t.aspect().equals("instance")) {
 *     return isAnonymous(((InstanceMethodCall)n).getCalledMethod());
 *   } else {
 *     return n instanceof InstanceFieldAccess;
 *   }
 * }
 *
 * private boolean checkMethod(AMethod m) {
 *   AMethod overridden = m.getOverriddenMethod();
 *   if(overridden==null) return true;
 *   rationale = "overriding must maintain anonymity (A2)";
 *   return !isAnonymous(overridden) || isAnonymous(m);
 * }
 * public boolean checkAbstractMethod(AbstractMethod m) {
 *   return checkMethod(m);
 * }
 * public boolean checkConcreteMethod(ConcreteMethod m) {
 *   return checkMethod(m);
 * }
 *
 * public boolean checkConstructor(Constructor c) {
 *   rationale = "called constructor must be anonymous as well (A3)";
 *   Constructor calledCtor = c.getConstructorCall().getCalledConstructor();
 *   return calledCtor.containingClass()==Barat.getObjectClass()
 *          || isAnonymous(calledCtor);
 * }
 */
public interface HasAnonymousMethods {
}
```

Figure 5.15: Implementation of constraints for anonymous methods

5.8 Related approaches

In this section, we discuss work related to the concept of confined types. While the examples for using CoffeeStrainer in the previous chapters have been fairly simple, confined types are an elaborate concept that in itself deserves to be put into the context of related research.

The original impetus for the work presented in this chapter comes from difficulties of implementing secure and reliable systems in Java. Some of these difficulties can be attributed to aliasing [Vitek et al. 1997, Vitek, Bryce 1999]. Confined types follow up on work on flexible alias protection [Noble et al. 1998] which tries to control aliasing at the level of individual objects. Related work is divided between literature on alias control and security; we review both topics in the following two subsections.

5.8.1 Alias control

Reference semantics permeate object-oriented programming languages, it is thus not surprising that the issue of controlling aliasing has been the focus of numerous papers in the recent years.

In [Noble et al. 1998], *flexible alias protection* is proposed to control potential aliasing amongst components of an aggregate object (or *owner*). Aliasing mode declarations specify constraints on sharing of references. The mode `rep` protects *representation objects* from exposure. In essence, `rep` objects belong to a single owner object and the model guarantees that all paths that lead to a representation object go through that object's owner. The mode `arg` marks argument objects which do not belong to the current owner, these objects may be aliased from the outside. Argument objects can have different *roles*, and the model guarantees that an owner cannot introduce aliasing between roles. In [Clarke et al. 1998], Clarke, Potter, and Noble formalize representation containment by means of ownership types. Both papers have been presented in the context of a simple programming language without inheritance or subtyping. There is no obvious way to maintain containment in the presence of either. Confined types were designed to support both concepts.

Hogg's Islands [Hogg 1991] and Almeida's Balloons [Almeida 1997] have similar aims. An Island or Balloon is an owner object that protects its internal representation from aliasing. The main difference to [Noble et al. 1998] 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 as it prevents many common programming styles: it is not possible to mix protected and unprotected objects as done with flexible alias protection and confined types. Hogg's proposal extends Smalltalk-80 with sharing annotations but it has neither been implemented nor formally validated. Almeida did implement an abstract interpretation algorithm for deciding whether a class meets his balloon invariants. But his approach is not modular, it requires whole-program analysis. The constraints present in this paper can be checked modularly, one class at a time.

The Sandwich types of Genius, Trapp, and Zimmermann [Genius et al. 1998] are a compromise between flexible alias protection and balloons. The objects protected from aliasing are computed by inspection of the type graph of the whole program. The criterion for protection is when a type is only reachable from another (owner) type. The prototypical example is the class LIST_CELL which only appears in the implementation of LIST. The drawback of sandwich types is that they require global program analysis, and do not deal with inheritance and subtyping.

Finally, Kent and Maung [Kent, Maung 1995] proposed an informal extension of the Eiffel programming language with ownership annotations that are tracked and monitored at run-time. In contrast, the constraints for confined types are checked at compile-time, which has two main advantages: there is no run-time overhead, and security violations are detected earlier.

5.8.2 Security

Confined types depart from the work on information flow control [Volpano, Smith 1997, Heintze, Riecke 1998, Myers 1999]. We are not trying to protect the information content of objects, as shown by the class signing example of Section 3, rather we control the flow of language level objects, namely object references. Further, confined types are as much about integrity as secrecy.

A paper of Leroy and Rouaix [Leroy, Rouaix 1998] has similar goals as the work presented in this paper. The authors formalize the security properties of applets written in a strongly typed programming language. Further, they propose a technique based on type abstraction to guarantee that certain locations in the store will not be written by untrusted components. Leroy and Rouaix did not deal with subtyping or inheritance as they chose a simple functional language (an idealization of CAML). In contrast, our work extends theirs to object-oriented languages. Moreover, confined types have been implemented for a real programming language.

Another recurrent theme is the use of objects as *capabilities* or guards [Hagimont et al. 1996, Hawblitzel et al. 1997, Gong 1998]. Different variants of this scheme boil down to the facade pattern [Gamma et al. 1995] in which a facade object protects access to one or more targets. The facade implements the security policy for access to the targets. The proposals typically do not provide any strong security guarantees, as some reference to one of the targets may still be leaked to an adversary. Confined types strengthen this approach. If target objects are confined, then no reference can be revealed to outside code.

5.9 Discussion

In this section, we discuss relations between confined types and areas that are not related to security. We see some implications of confined types on genericity (Section 5.9.1), and applications for confined types in the areas of software engineering (Section 5.9.2) and program optimization (Section 5.9.3).

5.9.1 Confined types and genericity

As has already been noted in Sections 5.5.2 and 5.6, confined types could profit from parameterized types. Because parameterized types reduce the need for reference widening (e.g., when storing objects in collections), much more reuse would be possible if confined types were combined with parameterized types. Interestingly, we found that confined types may influence the ongoing discussion about how to incorporate genericity in Java because they do not fit equally well with all proposals that have been put forward so far. There are two observations:

The first observation concerns the translation scheme used to translate generic types to normal classes and interfaces so that they can be executed on unmodified Java virtual machines. With a homogeneous translation scheme [Odersky, Wadler 1997, Bracha et al. 1998], different instantiations of a parameterized type are translated to a single class or interface. Because parameterized types instantiated with a confined type then cannot be distinguished at runtime from those instantiated with unconfined types, references to confined objects could leak out by confusing them with references to unconfined objects. Thus, confined types fit better with proposals that have a heterogeneous translation scheme [Myers et al. 1997, Bokowski, Dahm 1998], in which different instantiations of parameterized types are translated to different classes or interfaces.

When looking at the example presented in Section 5.6, another observation for the discussion about genericity can be made: In the example, the two classes `Key` and `KeyFactory` had to be decoupled by the intermediate interface `KeyWriter`. Although this interface would not be needed in a conventional design, the decoupling was required for subclassing both `Key` and `KeyFactory` in package `secure`. This suggests that virtual types [Thorup, Torgersen 1999] might be a better fit for confined types, as they allow subclassing of a whole family of classes in such a way that use relationships between classes in the original family become use relationships between classes in the derived family.

5.9.2 Strong encapsulation

Confined types may be useful from a software engineering point of view as well. Confined types can be viewed as the representational components of a framework which cannot be accessed from the outside, resulting in a strong encapsulation of these components. The external interface of the framework would then consist of unconfined types that usually do not contain functional code but make up a facade [Gamma et al. 1995] through which the framework must be used. Based on this architecture, for example, a package designer may decide to change the interface of a confined type, knowing that the effects of that change are limited to the single package and will not break client code.

Note that unlike techniques like guards and capabilities (see Section 5.8.2), in which every possible access path to otherwise unprotected objects needs to be controlled, confined types take the opposite approach. First, any direct access to confined types is disallowed, and then facades may be used to grant access for certain uses.

5.9.3 Optimization

Confined types can help program optimization. As the scope of a confined type is limited to a package, aggressive optimizations can be applied within the package. For instance, when performing static analysis of the package's code, it is known that it contains *all* uses of that package's confined types. It may thus be possible to remove methods that are not called in the package, as they are dead code, and even modify the structure of confined objects or of the class hierarchy [Tip et al. 1998].

Restricting widening improves the precision of concrete type inference and thus helps generating better code for confined types.

Finally, Genius, Trapp, and Zimmermann have shown that aliasing restrictions can be used to improve locality of memory access and have obtained significant speed up on small scale programs [Genius et al. 1998].

5.10 Conclusion

Software security is a difficult problem. This chapter has introduced two new language mechanisms, confined types and anonymous methods, that can be used for controlling the dissemination of object references. This control eases the task of writing secure code, as the interface between components is clearer.

Confinement and anonymity are enforced by a set of syntactic constraints which can be verified statically. Thus, our proposal incurs no run-time overhead and all confinement violations are caught before running the program. The verification procedure is modular as classes are analyzed individually. Furthermore, as all CoffeeStrainer constraints, our extension is transparent, i.e. annotated classes can also be compiled by standard Java compilers.