Confined Types

Boris Bokowski
Freie Universität Berlin,
GMD-FIRST Berlin, Germany
bokowski@acm.org

Jan Vitek
Purdue University
jv@cs.purdue.edu

Abstract

Sharing and transfer of object references is difficult to control in object-oriented languages. Unconstrained sharing poses serious problems for writing secure components in object-oriented languages. In this paper, we present a set of inexpensive syntactic constraints that strengthen encapsulation in object-oriented programs and facilitate the implementation of secure systems. We introduce two mechanisms: *confined types* to impose static scoping on dynamic object references and, for technical reasons, *anonymous methods* which are methods that do not reveal the identity of the current instance (`this`). Confined types protect objects from use by untrusted code, while anonymous methods allow standard classes to be reused from confined classes. We have implemented a verifier which performs a modular analysis of Java programs and provides a static guarantee that confinement is respected. We present security related programming examples.

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 mingle with authorized code raise the stakes much higher than more traditional systems as 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 [21]. The lack of ownership information [32] 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 state of affairs creates a tension between security and program efficiency. Placing dynamic security checks by default in the prologue of each and every method is not realistic for performance reasons. Instead, security conscientious environments offer dynamic security checks which may be explicitly interspersed in the program logic by the developer [11, 37]. Unfortunately nothing short of full-fledged program verification can ensure some check has not been omitted somewhere in the program, potentially compromising the security of the entire system.

Reusability creates its own set of problems for security. Similar to the synchronization problems of concurrent programming, inheriting from classes that do not implement the same security policy may create security anomalies. A class may be perfectly safe in one context, and open a security hole when inherited in another context. Of course, from the point of view of the library designer, it is often not feasible to design classes that are secure in all contexts and even if one could do that, the overall performance of the component would be unacceptable as the price of security would have to be paid even in contexts that do not need it.

To a large extent the problem for implementing secure systems in Java is one of defining interfaces between protection domains. As we stand now, a Java 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 [13] — by curtailing visibility of methods and fields — and also restrict the scope of types, object-oriented languages do not provide strong encapsulation [30]. 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. Given some definition of domain, we give the following definition of *confinement*: a type is said to be confined in a domain if and only if all references to instances of that type originate from objects of the domain. 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. The difference is most visible when subtyping is considered. For all practical purposes, confined types enforce static scoping of dynamic object references.

Our proposal for achieving confinement in Java depends on static constraints on the definition and use of objects. Thus there is no extra runtime overhead and confinement is guaranteed at compile time — avoiding the need to worry about "confinement breach" exceptions. We extend Java with two additional modifiers, one for classes and one for methods, and enforce some restrictions on programs. While certain programming tasks may be clumsier, we argue that these restrictions are mild and that reasoning about security is much simpler.

Confined types have been implemented with `CoffeeStrainer` [2], a framework for static checking of structural constraints on Java programs. In this implementation we have chosen the Java `package` as the unit of protection. Packages are well suited for this task as they group related classes and they already provide basic access control features. Our annotations do not affect program semantics, thus a valid program with confinement annotations behaves identically as the same program with no annotations. The verification of the validity of these annotation is modular and is thus able to accommodate dynamic loading of classes.

Road map: An overview of language-based security mechanisms is given in Section 2. For a large class of security problems, these mechanisms are not sufficient. Section 3 details a well-known security defect in the Java Development Kit (JDK) which is our motivating example. Anonymous methods are introduced in Section 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. Section 6 introduces a complete programming example with confined types. Section 7 compares confined types to other approaches. Finally, Section 8 discusses design choices, implications on genericity, and benefits that confined types offer for other areas than security.

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 [6, 14, 25].

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 has been advocated for three main reasons. Firstly, language semantics can help to reason about program behavior and thus to prove security properties. Secondly, 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.

We review some of the main programming language approaches to security and discuss their merits in the context outlined above.

2.1 Language Safety

Safe programming languages guarantee that the execution of programs proceeds according to the language semantics. This means that, for example, types are not misinterpreted and data is not mistaken for executable code. In Java, safety depends on four techniques: *bytecode verification* to ensure that programs are well-formed, *strong typing* to guarantee that values are used according to their definition, *automatic memory management* to prevent errors such as deleting a live object, and *memory protection* to prevent array and stack operations from overflowing [45].

While safety is not security, these mechanisms are an essential foundation for language-based security.

2.2 Information Flow

Over the last 20 years an abundant body of work has been devoted to information flow control. Multilevel security policies [8] originally conceived for military applications are based on the notion that all data is labeled with security levels and that principals may only access data for which they have security clearance. The objective of information control techniques is to obtain a form of *non-interference* — a property which, informally, means that the values of low level security variables may not depend on high level security variables [43, 26, 42]. This requires checking all channels of communication that may create information flows (these include implicit channels such as conditional expressions and loops, as well as the more exotic timing and probabilistic channels). To date, these techniques are still not used in practice. Part of the problem stems from inherent restrictions: to achieve non-interference in a multi-threaded language Volpano and Smith [36] had to forbid the guard of loops to depend on high security variables. This restriction is quite stringent, yet even then probabilistic channels remain. A more fundamental problem with information flow control is that it assumes a homogeneous software system in which security labels are set once and for all, and all subsystems agree on

[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.*

the labels and on their meaning. In a distributed system assembled from heterogeneous components these assumptions do not hold. In our case, there are as many security policies as there are protection domains (e.g. applets), each of which may decide on its own labeling scheme. Furthermore, there is *mutual* distrust among the different components. Some of these problems have been addressed in a sequential subset of Java [28, 29], but extending the approach to the full language is still an open problem. To summarize, information flow provides a sound notion of what a secure system is, but current technology remains too restrictive for widespread usage.

2.3 Access Control

Discretionary access control mechanisms fail to provide the same strong guarantees as information flow control, but are less constraining and thus more usable in practice. Access control mechanisms entail security checks before any potentially dangerous operation 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*; types can be used to limit the operations that can be invoked on an object [33]. 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 [22, 33].

Dynamic access control: Java provides dynamic access control mechanisms based on call stack inspection. The purpose of this check is to verify that the current method was invoked (transitively) by a method with appropriate privileges [11]. Another dynamic scheme which has been proposed is to use objects as capabilities [23] by interposing a restricted proxy object between the user and the target ([12], see also [15, 44, 40]).

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 were originally conceived for software engineering purposes rather than for security and they fail to provide a sufficient solution to access control.

We now turn to an example to demonstrate the kind of problems which we want to address.

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 [35]. 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, an applet 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 other 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.

3.1 The Security Breach in Detail

In Figure 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.

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

Figure 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 2 makes the copy 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 [42] 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

```
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 2: Ad-hoc fix of security problem

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.

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 to `SecureIdentity` and declaring it *confined*. 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 [12, 15]. 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 3 outlines our solution.
The `getSigners()` method is similar to Figure 2. 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 how confined types help in developing secure code. They draw a strong demarcation line between internal representation objects and external interface objects.

```
confined class SecureIdentity ...{
    ...
    // the original Identity implementation
    ...
}

public class Identity {
    SecureIdentity target;
    Identity(SecureIdentity t) { target = t; }
    ...// public operations on identities;
}

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 3: Signatures with confined types.

We now define confinement in more detail. We begin the presentation with the definition of anonymous methods which will prove to be essential for modular checking of confinement.

4 Anonymous Methods

An *anonymous method* is a method that does not depend on the identity of the current instance to compute its value. The behavior of the method is entirely determined by its arguments and the value of the receiver object's fields. In order not to reveal the object's identity, an anonymous must not introduce new aliases to the current instance, nor perform any identity-dependent operations. Although anonymous methods are essential to allow confined types to inherit methods from unconfined parents, they also have interesting properties in their own right which may be useful in other contexts [4].

In Java technical terms, an anonymous method is a non-native 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, it is not allowed to perform reference comparisons using `this`[2].

Figure 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 pre-

[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.

```
  class Example {
      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 4: Anonymous methods.

serve anonymity. Line (4) reveals `this` to 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 (**anon**), and check for each such declared method whether it conforms to the definition of anonymity. 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()`, `hashCode()`[3], and `finalize()`.

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

[3]By default, `hashCode()` and `identityHashcode()` return the object's address in the heap; this introduces a dependence on the object's identity. We therefore require the implementation to calculate the code without using the object's address.

$\mathcal{A}1$	The reference `this` can only be used for accessing fields and calling anonymous methods of the current instance.
$\mathcal{A}2$	Anonymity declarations must be preserved when overriding methods.
$\mathcal{A}3$	The constructor called from an anonymous constructor must be anonymous as well.
$\mathcal{A}4$	Native methods may not be declared anonymous.

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 form of double dispatching [9]. We have mentioned that the default implementation of `hashCode()` must be changed[4], this 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. Changing code from reference semantics to value semantics has deep implications [24] and is not as efficient.[5]

To obtain a better sense of the impact of anonymity declarations on programming style, we analyzed JDK 1.1 to find out how many existing methods meet the above mentioned criteria ($\mathcal{A}1$, $\mathcal{A}2$, $\mathcal{A}3$, and $\mathcal{A}4$). The data has been 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 Table 1, 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	329 (94%)	1042 (83%)

Table 1: Anonymous methods in existing code.

Anonymous methods are closely related to Boyland's concept of borrowed receiver [4]. Boyland defines a reference to be borrowed by a method if the method can not store the

[4]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 hashing.

[5]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);
```

reference and thus does not introduce any static aliases. We have considered relaxing the anonymity restrictions to allow reference equality and the use of the unmodified `hashCode` method. With that definition the anon methods in Table 1 would go up to 332 and 1047 for `util` and `awt` respectively.

Section 5.4 explains our use of anonymous methods. We now turn to the definition of confined types.

5 Confined Types

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 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.

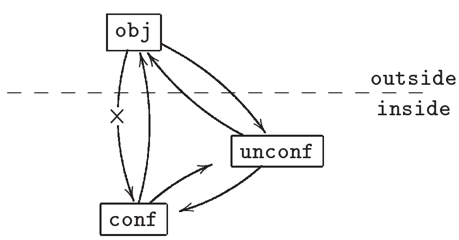

Figure 5: References between packages.

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

5.1 Overview of the Problem

Before listing the confinement constraints, it is helpful to consider all constructs with which object references may be transferred from a package `inside` to another package `outside`.

We start with reference transfers that originate from code of 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 Figure 6. 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 5.2 gives constraints on class and interface declarations. Section 5.3 presents constraints that prevent widening. Section 5.4 discusses constraints that deal with hidden widening. Based on the constraints introduced so far, Section 5.5 explains why reference transfers originating in the inside package cannot occur. Finally, Section 5.6 presents the remaining constraints that address reference transfers originating in outside packages.

The constraints can be checked statically. Our implementation is based on CoffeeStrainer, a system for statically checking structural constraints on Java programs [2]. An important design goal for these constraints was that only the

```
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[] {c};
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 { }

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 6: Transferring references.

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 containing confined types cannot be extended by untrusted programs. In practice, this constraint may have to be checked at load time.

5.2 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.

$\mathcal{C}1$	A confined class or interface must not be declared public or protected, and must not belong to the unnamed global package.
$\mathcal{C}2$	Subtypes of a confined type must be confined and belong to the same package as their confined supertype.

$\mathcal{C}1$ 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. $\mathcal{C}2$ 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.

5.3 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 an `Object` variable, 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.

$\mathcal{C}3$	Widening of references from a confined type to an unconfined type is forbidden in assignments, method call arguments, return statements, and explicit casts.

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 $\mathcal{C}3$.

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 8.2 discusses the impact of confined types on genericity.

5.4 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 the 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 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*.

$\mathcal{C}4$	Methods invoked on a confined object must either be defined in a confined class or be anonymous methods.
$\mathcal{C}5$	Constructors called from the constructors of a confined class must either be defined by a confined class or be anonymous constructors.

$\mathcal{C}4$ constrains inherited methods, in the case of overriden methods, i.e., if a method defined in a superclass is overriden 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. Moreover, note that interfaces do not play a role here since they do not introduce code.

Anonymous methods ease the restrictions that would otherwise be imposed on inheritance. Without them, it would be unsafe to invoke any inherited method of a confined object.

5.5 Preventing Transfer from the Inside

In our list points `r1` to `r6` involve transfers that originate in the inside package. Based on the constraints introduced so far, points `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 $\mathcal{C}1$ and $\mathcal{C}2$), 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 $\mathcal{C}3$.

Similarly, point `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 point `r4` — calling a method in an unconfined supertype — are not ruled out completely; rather, constraints $\mathcal{C}4$ and $\mathcal{C}5$ 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 can 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 $\mathcal{C}1$ and $\mathcal{C}2$ 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 which is rather obscure: 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 point `r6`. The class `java.lang.Thread` also requires special treatment as one of its static methods returns a reference to the currently executing thread object. We require that:

$\mathcal{C}6$	Subtypes of `java.lang.Throwable` and `java.lang.Thread` may not be confined.

5.6 Preventing Transfer from the Outside

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

$\mathcal{C}7$	The declared type of public and protected fields may not be confined.
$\mathcal{C}8$	The return type of public and protected methods may not be confined.

Fields whose declared types are confined types should not be accessible from outside the package, *i.e.*, confined field may not be public or protected ($\mathcal{C}7$), preventing object reference

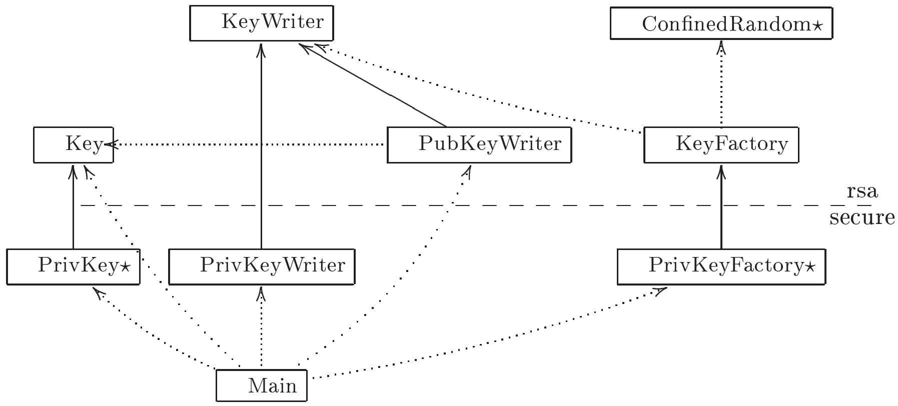

Figure 7: Relationships between package `rsa` and package `secure`. Full arrows indicate subtyping relations. Dashed arrows indicate implementation dependencies. Confined types are indicated with a ⋆.

transfer according to point `r7`. Although the confined type itself is not accessible from outside the package, confinement is not enforced in other packages. Thus, if a field of a confined type was accessible, it would be possible for the outside package to widen the reference to an unconfined supertype.

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 ($\mathcal{C}8$). Thus, point `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 (point `r10`) cannot occur because confined classes are not accessible from outside.

6 Example: Public-Key Cryptography

Public-key cryptography is one of the essential tools for security in distributed systems. The implementation of public key cryptography must therefore be secure. Furthermore, it should be possible to reuse this implementation in different contexts without endangering security.

More specifically, our 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 [34]. Further, we would like to offer the guarantee to clients of the RSA package that the object that represent their private keys remain confined to their application, and that under no circumstance another applet 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 easily be another client package (even simultaneously on the same JVM) which confines its public key class.

The package `rsa`, Figure 8, 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 9, 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, we will detail them in the following paragraphs. Figure 7 illustrates the relationships between the two packages.

Relevant portions of the implementations of both packages are given in Figure 8 and Figure 9.

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 ($\mathcal{C}4$).

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.

```
package rsa;

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


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

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


private confined class ConfinedRandom
     extends Random { }

public interface KeyWriter {
     anon public void setValues(
          BigDecimal m, BigDecimal e);
}


public class KeyFactory {

     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 8: Package containing RSA algorithm

```
package secure;

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


confined class PrivKey
     extends Key { }


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;
     }
}


confined class PrivKeyFactory
     extends KeyFactory { }


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 9: Confining a type in a different package

The class `KeyFactory` does not set the internal values of Key objects directly. Rather, it uses 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 $\mathcal{C}3$. 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 $\mathcal{C}3$, 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 8.2.

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 $\mathcal{C}7$, while `publicKey` can be `public`. In `main()`, then, a `Factory` object is created and `genKeyPair()` is invoked on it, providing two instances of `PubKeyWriter` and `PrivKeyWriter`, respectively.

7 Related Work

The original impetus for the work presented here comes from difficulties of implementing secure and reliable systems in Java. Some of these difficulties can be attributed to aliasing [41, 40]. Confined types follow up on work on flexible alias protection [30] in which we tried 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.

7.1 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 [19, 18, 1, 30, 10, 20, 7].

In [30], we proposed flexible alias protection 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 [7], 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 [18] and Almeida's Balloons [1] have similar aims. An Island or Balloon is an owner object that protects its internal representation from aliasing. The main difference to [30] 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 been formally validated. Almeida did implement an abstract interpretation algorithm for deciding whether a class meets his balloon invariants. But his approach requires whole-program analysis. The constraints present in this paper can be checked modularly, one class at a time.

The Sandwich types of Genius, Trapp, and Zimmermann [10] 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 [20] proposed an informal extension of the Eiffel programming language with ownership annotations that are tracked and monitored at run-time. Confined type are static, a choice better suited to security as errors are caught earlier.

Table 2 compares the proposals discussed above. Partial encapsulation allows selective protection of components. Enforcement of constraints can either be done at compile-time (*static*) or at run-time (*dynamic*). Verification can require analysis of the entire program (*whole-program*) or be modular at the class level (*class*). Granularity of protection can be either: at the *object* level, meaning that individual objects are protected, at the *class* level, meaning that all instance of a class are treated as a single encapsulation domain, and finally at the *package* level, meaning that all instances of all classes belonging to the same package are grouped in a single domain.

7.2 Security

Confined types depart from the work on information flow control [17, 28, 42]. 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, or more precisely, object references. Further, confined types are as much about integrity as secrecy.

The elegant paper of Leroy and Rouaix [22] 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. They propose a technique

	Language	Inheritance	Encapsulation	Enforcement	Modularity	Granularity
Islands [18]	Smalltalk-80	Yes	Full	Static	Class	Object
Balloons [1]	Toy	Yes	Full	Static	Whole-program	Object
Flexible Alias [30]	Toy	No	Partial	Static	Class	Object
Sandwich [10]	Toy	No	Full	Static	Whole-program	Class
Kent & Maung [20]	Eiffel	Yes	Partial	Dynamic	–	Object
Confined types	**Java**	**Yes**	**Partial**	**Static**	**Class**	**Package**

Table 2: Comparison of alias control techniques.

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. They chose a simple functional language (an idealization of Caml), our work can be viewed as an extension of theirs to object-oriented languages.

Another recurrent theme is the use of objects as *capabilities* or guards [12, 16, 15]. Different variants of this scheme boil down to the facade pattern [9] 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.

8 Discussion

8.1 Design alternatives

Unlike flexible aliasing protection [30], our proposal protects entire packages. This flat protection model can be limiting. First, the objects we want to protect need not all be in the same package. Second, it is not possible to compose larger systems out of components.

We have considered different designs allowing a class to be confined to a group of classes which need not be in the same package. For example, we could define the notion of a reference protection domain, then each class would be declared to belong to some domain. The following declaration bundles three classes in a protection domain.

```
domain java.security.Identity, java.lang.class,
     java.security.SecureIdentity;
```

The `SecureIdentity` class is still defined as confined, but now it will be visible only to the other two classes in the domain. The drawback of external domains is that we cannot use package visibility to define methods that may only be used by classes in the same domain.

This idea could be extended further to hierarchical protection domains. This requires named domains. Next we define two domains, one is the aforementioned domain, the second is a larger domain encompassing all security classes.

```
domain IdentityDomain is
     java.security.Identity, java.lang.class,
     java.security.SecureIdentity;

domain SecurityDomain is
     java.security.*, IdentityDomain;
```

While possible, hierarchical domains are pushing towards more complex models such as flexible alias protection [30, 7]. The cost in complexity may outweigh the gains.

8.2 Confined Types and Genericity

As has already been noted in sections 5.3 and 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 [31, 5], 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 [27, 3], in which different instantiations of parameterized types are translated to different classes or interfaces.

When looking at the example presented in Section 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 [38] 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.

8.3 Software Engineering

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. The external interface of the framework would then consist of unconfined types that usually do not contain functional code but make up a facade [9] through which the framework must be used. Based on this architecture, 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 such as guards and capabilities (see Section 7.2), 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.

8.4 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, static analysis of the package code 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 [39].

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 [10].

9 Conclusion

Software security is a difficult problem. This paper introduces 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 sharper.

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.

We have implemented a confinement verifier for Java using CoffeeStrainer [2]. The verification procedure is modular as classes are analyzed individually. Our extensions are transparent. Annotated classes can be compiled by the standard Java compiler (The concrete syntax used is to tag methods by the special comment `/*:anon*/`, and derive confined types from the empty interface `ConfinedType`). Our implementation is available from:

`http://www.inf.fu-berlin.de/~bokowski/ConfinedTypes`

Acknowledgments

The authors whish to thank John Boyland, James Noble and Paulo Sergio Almeida for their comments. The second author did part of the research at the University of Geneva, in the Object System Group.

References

[1] P. S. Almeida. Balloon types: Controlling sharing of state in data types. In M. Aksit and S. Matsuoka, editors, *ECOOP'97—Object-Oriented Programming, 11th European Conference*, volume 1241 of *Lecture Notes in Computer Science*, pages 32–59, Jyväskylä, Finland, 9–13 June 1997. Springer.

[2] B. Bokowski. Coffeestrainer: Statically-checked constraints on the definition and use of types in Java. In *Proceedings of ESEC/FSE'99*, Toulouse, France, Sept. 1999.

[3] B. Bokowski and M. Dahm. Poor man's genericity for Java. In *JIT Proceedings*. Springer-Verlag, Frankfurt, Germany, Nov. 1998.

[4] J. Boyland. Deferring destruction when reading unique variables. Technical report, University of Wisconsin – Milwaukee, Mar. 1999.

[5] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In *OOPSLA Proceedings*. ACM Press, Vancouver, BC, Oct. 1998.

[6] J. Chase, H. Levy, M. Baker-Harvey, and E. Lazowska. Opal: A single address space system for 64-bit architectures. In *Proceedings of the Fourth Workshop on Workstation Operating Systems*, pages 80–85, 1993.

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

[8] D. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, May 1976.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

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

[11] L. Gong. Java security architecture (JDK 1.2). Technical report, JavaSoft, July 1997. Revision 0.5.

[12] L. Gong. Guarding objects. In G. Vigna, editor, *Mobile Agents and Security*, volume 576 of *LNCS*, pages 1–23, Berlin, Germany, Aug. 1998. Springer.

[13] J. Gosling, B. Joy, and G. L. Steele. *The Java Language Specification*. The Java Series. Addison-Wesley, Reading, MA, USA, 1996.

[14] R. Grimm and B. N. Bershad. Security for extensible systems. In *Proceedings of 6th Workshop on Hot Topics in Operating Sytems*, pages 62–66, Cape Cod, Massachusetts, May 1997.

[15] D. Hagimont, J. Mossière, X. R. de Pina, and F. Saunier. Hidden software capabilities. In *16th International Conference on Distributed Computing System*, Hong Kong, May 1996. IEEE CS Press.

[16] C. Hawblitzel, C.-C. Chang, G. Czajkowski, D. Hu, and T. von Eicken. Implementing Multiple Protection Domains in Java. Technical Report 97-1660, Cornell University, Department of Computer Science, 1997.

[17] N. Heintze and J. G. Riecke. The SLam calculus: Programming with secrecy and integrity. In *Proceedings of the 25th POPL*, Jan. 1998.

[18] J. 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, Nov. 1991. Published as ACM SIGPLAN Notices, volume 26, number 11.

[19] J. Hogg, D. Lea, A. Wills, D. de Champeaux, and R. Holt. The Geneva convention on the treatment of object aliasing. *OOPS Messenger*, 3(2), Apr. 1992.

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

[21] W. Landi. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems*, 1(4), Dec. 1992.

[22] X. Leroy and F. Rouaix. Security properties of typed applets. In *Conference Record of POPL '98: The 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 391–403, San Diego, California, 19–21 Jan. 1998.

[23] H. Levy, editor. *Capability Based Computer Systems*. Digital Press, 1984.

[24] G. Lopez, B. Freeman-Benson, and A. Borning. Constraints and object identity. In *ECOOP Proceedings*, LNCS 821, pages 260–279. Springer-Verlag, Bologna, Italy, July 1994.

[25] S. Lucco, O. Sharp, and R. Wahbe. Omniware: A Universal Substrate for Web Programming. *World Wide Web Journal*, 1(1):359–368, Dec. 1995.

[26] J. McLean. Security models. In J. Marciniak, editor, *Encyclopedia of Software Engineering*. Wiley & Sons, 1994.

[27] A. Myers, J. Bank, and B. Liskov. Parameterized types for Java. In *POPL Proceedings*. ACM Press, Paris, France, Jan. 1997.

[28] A. C. Myers. Jflow: Practical static information flow control. In *Proceedings of the 26th ACM Symposium on Principles of Programming Languages (POPL 99)*, 1999.

[29] A. C. Myers and B. Liskov. Complete, safe information flow with decentralized labels. In *Proceedings of the 1998 IEEE Symposium on Security and Privacy, Oakland, California*, pages 186–197, 1998.

[30] J. Noble, J. Potter, and J. Vitek. Flexible alias protection. In *Proceedings of ECOOP'98*, Brussels, Belgium, July 20 - 24 1998.

[31] M. Odersky and P. Wadler. Pizza into Java: Translating theory into practice. In *Proc. 24th ACM Symposium on Principles of Programming Languages*, January 1997.

[32] J. Potter, J. Noble, and D. Clarke. The ins and outs of objects. In *Australian Software Engineering Conference*, Adelaide, Australia, November 1998. IEEE Press.

[33] J. G. Riecke and C. A. Stone. Privacy via Subsumption. In *Fifth Workshop on Foundations of Object-Oriented Languages*, 1998.

[34] R. Rivest, A. Shamir, and L. Aldeman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Commun. ACM*, 21(2), 1978.

[35] Secure Internet Programming Group. http://www.cs-.princeton.edu/sip/news/april29.html. 1997.

[36] G. Smith and D. Volpano. Secure information flow in a multi-threaded imperative language. In *Conference Record of POPL '98: The 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 355–364, San Diego, California, 19–21 Jan. 1998.

[37] J. Tardo and L. Valente. Mobile agent security and Telescript. In *IEEE CompCon*, 1996.

[38] K. K. Thorup and M. Torgersen. Unifying genericity – combining the benefits of virtual types and parameterized classes. In *ECOOP Proceedings*. Springer-Verlag, Lisbon, Portugal, June 1999.

[39] F. Tip, C. Laffra, P. F. Sweeney, and D. Streeter. Size matters: Reducing the size of java class file archives. Technical report, IBM Research Report RC 21321, Oct. 1998.

[40] J. Vitek and C. Bryce. Secure mobile code: the JavaSeal experiment. Manuscript, 1999.

[41] J. Vitek, M. Serrano, and D. Thanos. Security and communication in mobile object systems. In D. Tsichritzis, editor, *Objects at Large*. University of Geneva, 1997.

[42] D. Volpano and G. Smith. A type-based approach to program security. *Lecture Notes in Computer Science*, 1214, 1997.

[43] D. Volpano and G. Smith. Confinement properties for programming languages. *SIGACT News*, 29(3):33–42, Sept. 1998.

[44] D. Wallach, D. Balfanz, D. Dean, and E. Felten. Extensible Security Architectures for Java. In *Proceedings of the 16th Symposium on Operating System Principles*, 1997.

[45] F. Yellin. Low level security in Java. In *Fourth International Conference on the World-Wide Web*, MIT, Boston, Dec. 1995.