Isolating Untrusted Software Extensions by Custom Scoping Rules

Philip W. L. Fong
Department of Computer Science
University of Calgary
Calgary, Alberta, Canada
pwlfong@ucalgary.ca

Simon Orr
Department of Computer Science
University of Regina
Regina, Saskatchewan, Canada
orrsim11@cs.uregina.ca

August 19, 2009

Abstract

In a modern programming language, *scoping rules* determine the visibility of names in various regions of a program [15]. In this work, we examine the idea of allowing an application developer to customize the scoping rules of its underlying language. We demonstrate that such an ability can serve as the cornerstone of a security architecture for dynamically extensible systems.

A run-time module system, ISOMOD, is proposed for the Java platform to facilitate software isolation. A core application may create namespaces dynamically and impose arbitrary name visibility policies (i.e., scoping rules) to control whether a name is visible, to whom it is visible, and in what way it can be accessed. Because ISOMOD exercises name visibility control at load time, loaded code runs at full speed. Furthermore, because ISOMOD access control policies are maintained separately, they evolve independently from core application code. In addition, the ISOMOD policy language provides a declarative means for expressing a very general form of visibility constraints. Not only can the ISOMOD policy language simulate a sizable subset of permissions in the Java 2 security architecture, it does so with policies that are robust to changes in software configurations. The ISOMOD policy language is also expressive enough to completely encode a capability type system known as Discretionary Capability Confinement. In spite of its expressiveness, the ISOMOD policy language admits an efficient implementation strategy. Name visibility control in the style of ISOMOD is therefore a lightweight access control mechanism for Java-style language environments.

Keywords: Access control, name visibility control, scoping rules, language-based security, Java.

1 Introduction

In a modern programming language, *scoping rules* determine the visibility of names in various regions of a program [15]. In this work, we examine the idea of allowing an application developer to customize the scoping rules of its underlying language. We demonstrate that such an ability can serve as the cornerstone of a security architecture for dynamically extensible systems.

In modern day computing, a successful software system must anticipate the evolution of software requirements. This gives rise to a family of software systems known as *dynamically extensible systems*, in which the functionality of a core application can be augmented dynamically by loading, linking, and invoking program code units that were not originally part of the core application. Dynamically extensible systems find their uses in mobile code platforms, scriptable applications, and systems with plug-in architectures. In each case, trusted and untrusted code units are loaded and executed in the same run-time environment. The challenge of *secure cooperation* is to protect these mutually suspicious code units from one another while they are executing within the same run-time environment.

An established paradigm for addressing the challenge of secure cooperation is language-based security [30]. Specifically, untrusted code units are encoded in a safe language, and subsequently executed in a secure run-time environment, the protection mechanisms of which are implemented by programming language technologies such as type systems, program rewriting and execution monitoring.

Most existing language-based approaches to access control are based on the classical notion of *interposition* [1, 32, 33, 35]. A direct implementation of this idea is to interpose monitoring code at the entry points of security relevant system services. At run time, authorization decisions are made by examining invocation arguments or execution history. In the Java Virtual Machine (JVM) [22] and the Common Language Infrastructure (CLI) [9], a direct implementation of this approach is found. Specifically, *stack inspection* [33], the main access control mechanism of the JVM and the CLI, is essentially a form of interposition. Direct interposition, however, is difficult to maintain. Security checks are scattered over the entire host system. Fixing a vulnerability requires the availability of host system source code. Worst still, as security checks are hard-coded into the host system, evolution in security requirements or software configuration is not easily addressed without reprogramming the host system itself.

A second language-based approach to implement interposition is by *load-time binary rewriting* [10, 32, 33, 35] Specifically, monitoring code is *weaved* into untrusted code at load time. Although this so called *Inlined Reference Monitor (IRM)* approach [32] is equal in expressive power to direct interposition [17], the former has clear software engineering advantages over the latter. In particular, the late binding of security checks allows security code to evolve separately from the rest of the system, thereby addressing the software engineering concerns raised in the previous paragraph. Unfortunately, independent reports have confirmed that the injected code incurs significant run-time overhead [32, 33]. For example, in [32], up to 70% slow down was observed if domain-specific optimization was not introduced.

There is an obvious engineering dilemma in the design of interposition-based access control mechanisms. By embedding security checks in the host system, direct interposition fails to support the evolution of security requirements and software configurations in a graceful manner. Maintaining the security policy offline, IRM has the advantage of anticipating software evolution, but it incurs significant runtime overhead.

The research question investigated by this work is as follows. *Is interposition (direct or IRM-based) always necessary for access control in the context of dynamically extensible systems?* Interposition is motivated by the need for execution monitoring [29], in which the dynamic state and the execution history of a system are examined when authorization decisions are made. In many cases, one simply wants to completely turn off a system service. (This is evident in the large number of target-less `BasicPermissions` defined in the Java 2 security architecture [16].) In other cases, the safety property [29] to be enforced is memory-less, and the avoidance of the confused deputy problem [18] is not a significant concern. In such contexts, execution monitoring can be replaced by a lighter-weight enforcement mechanism that does not exhibit the engineering dilemma presented by interposition.

This work examines a seldom studied point in the design space of language-based access control, specifically, *name visibility control*. The intuition is that, if the name of the entry point for a system service is not visible to an untrusted code unit, then the service is essentially inaccessible to the code unit. Therefore, access control can be achieved by specifying what names are visible, to whom they are visible, and to what extent they are visible. These aspects of name visibility are specified by the *scoping rules* of a programming language. Such rules are usually fixed for modern block-structured programming languages. In order to use name visibility control as an access control mechanism, two ingredients are necessary: (1) a policy language for specifying custom scoping rules that constrain the visibility of names, and (2) a protection mechanism that allows an application to impose custom scoping rules over a local namespace in which untrusted code resides. In the programming language literature, a facility that is responsible for managing the visibility of names across namespaces [19] is called a *module system*.

The goal of this research is to investigate the degree to which name visibility control can serve the purpose of access control when full-fledged execution monitoring is not necessary. To this end, a module system, ISOMOD, is proposed as a practical security architecture for dynamically extensible Java applications (Section 2). Because ISOMOD exercises name visibility control only at load time, and does not inject any monitoring code into classfiles, loaded code runs at full speed. Furthermore, because ISOMOD access control policies are maintained separately, they evolve independently from core application code.

An intriguing finding of this study is that *a rich family of access control policies can be expressed as name visibility constraints*. The ISOMOD policy language provides a declarative means for expressing a very general form of visibility constraints (Section 3). Not only can the ISOMOD policy language simulate a sizable subset of permissions in the Java 2 security architecture (Section 4.1), it can do so with policies that are robust to changes in software configurations (Section 4.2). The ISOMOD policy language is also expressive enough to completely encode a capability type system known as Discretionary Capability Confinement [12, 13] (Section 4.3). In spite of its expressiveness, the ISOMOD policy language admits an efficient implementation strategy (Sections 5 and 6).

In short, ISOMOD avoids the technical difficulties of interposition by trading off an acceptable level of expressiveness. Therefore, name visibility control in the style of ISOMOD is a lightweight alternative to interposition for language-based access control.

2 The ISOMOD Security Architecture

ISOMOD employs *name visibility control* as the sole mechanism for access control. We begin our discussion with a review of the Java class loading mechanism from the perspective of name visibility control.

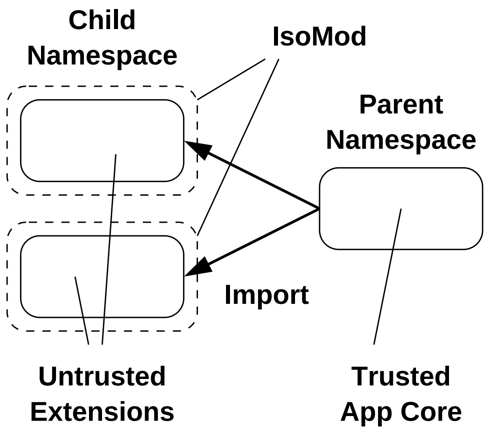

Figure 1: Hierarchical Namespaces and IsoMod

2.1 Delegation-style Class Loading

In programming language terminology, a Java class loader is the *mirror* [5] of a run-time namespace. An instance of the `ClassLoader` class is employed by the JVM to load the object code of classes, and to define the classes in the namespace mirrored by the `ClassLoader` instance. The object code of a class is transported through a network or stored in a file system in an intermediate representation known as a *classfile*. *Class definition* [22, Section 5.3.5] is the process by which a classfile is converted into a `Class` object in the JVM. Programmers may define a custom subclass of the `ClassLoader` class, thereby overriding the procedure by which the JVM locates classfiles (e.g., loading classfiles from the World Wide Web), or introducing a classfile preprocessing step prior to class definition (e.g., injecting instrumentation code). Once a class C is defined by a class loader L, L is said to be the *defining class loader* [21] of C.

Hierarchical organization of namespaces is enabled by the delegation model of class loading [21], in which the names visible in a parent namespace is imported implicitly into a child namespace (Figure 1). Specifically, the set of class names visible in a namespace L is the union of (1) the set of class names visible in the parent of L plus (2) the set of class names that are defined locally by L. How is this effect achieved algorithmically? Associated with every class loader L is another class loader, called the *delegation parent* of L. The class loaders thus form a *delegation hierarchy*, at the root of which is the *bootstrap class loader*. To look up the `Class` object of a given class name in a class loader L, the delegation parent of L is first consulted. If a `Class` object of the specified name is defined by the delegation parent, then that `Class` is returned. Otherwise, the delegation parent of that delegation parent is consulted, and so on. When none of the delegation ancestors of L defines a `Class` of the given name, then L will load and define that class (if it has not already done so). This class is then returned as the result of class look up.

A class may refer to external entities such as other classes or their fields and methods. These external references are resolved in the same namespace in which the referring class is defined (i.e., via the defining class loader of the referring class). In this way, static scoping is enforced.

2.2 An Architecture for Name Visibility Control

In a dynamically extensible software system, the trusted application core is defined in a parent namespace, while child namespaces are created for defining untrusted software extensions (Figure 1). Core application services are exposed to the extension code by implicitly importing names from the core application namespace to the extension namespace. ISOMOD is a run-time module system designed for isolating untrusted software extensions. It does so by controlling the visibility of names in the namespaces in which untrusted software extensions reside. Specifically, an ISOMOD namespace enforces two kinds of control: (1) restricting the visibility of names that are imported from the parent namespace, and (2) restricting the visibility of locally defined names. When a name is placed under visibility control, an ISOMOD namespace may (a) control which locally defined class can "see" the name, and (b) present an alternative, restricted view of the entity to which the name is bound. Every ISOMOD name space is endowed with a custom *name visibility policy*, which specifies visibility restrictions to be imposed on the names visible in the namespace. When appropriately constructed, an ISOMOD policy may be used to selectively hide core application services from untrusted extensions (Section 4.1 and 4.2), or impose collaboration protocols among classes defined in the extension namespace (Section 4.3). A major contribution of this work is the design of a policy language that can express a rich family of access control policies as fine-grained visibility constraints.

An ISOMOD namespace is an instance of a user-defined class loader class. An ISOMOD class loader performs extra checks on a classfile before converting it into a `Class` object. Specifically, class definition is only authorized when no external accesses in the classfile are denied by the policy. This *late enforcement* (i.e., load time) of visibility control distinguishes ISOMOD from traditional module systems, in which visibility control is enforced only at compile time. It is this feature that makes the ISOMOD module system into a viable protection mechanism.

An ISOMOD namespace may be constructed at run-time by an application core from an ISOMOD policy. This *late binding* of access control policy to code not only supports the separate maintenance of code and policy, but also supports the presentation of different views of the same application core to different extensions.

3 The ISOMOD policy language

The ISOMOD policy language provides a declarative and expressive means to specify the access control policy of an ISOMOD name space. An *access* is composed of three elements: (1) a *subject*, (2) an *object*, and (3) an *access right*. An object is also called a *target* to avoid confusion in the context of object-oriented programming. ISOMOD controls access to three kinds of targets: (a) declared types, (b) fields, and (c) methods. A *declared type* is either a class or an interface. For brevity, the word "class" is used as a synonym of "declared type". Every target is identified by a name visible in the ISOMOD namespace. A target can be accessed by exercising a fixed set of access rights as outlined in Figure 2. A subject is either (a) a declared type whose name is defined in the ISOMOD namespace, or (b) a method declared in such a class.

An ISOMOD policy is made up of a finite number of *policy clauses* (or *access control rules*), each of which has the following general syntax:

O (**allows**|**denies**) $\{r_1, \ldots, r_k\}$ [**to** *S*] [(**when**|**unless**) c]

Access Right	**Description**
Declared type target C / Declared type subject D	
extend	D extends C
implement	D implements C
Declared type target C / Method subject N	
catch	N handles exception type C
cast	N casts a reference to C
instanceof	N checks if a reference is C
new	N creates an instance of C
reflect	N gets the `Class` object of C
new-array	N creates an array of C
Similarly, cast-array, instanceof-array, reflect-array.	
Field target F / Method subject N	
get	N reads F
put	N writes F
Method target M / Method subject N	
invoke	N invokes M
override	N overrides M

Figure 2: Access Rights

Unary Predicates		
final(C)	*abstract*(C)	*interface*(C)
public(C)	*package-private*(C)	
Binary Predicates		
subclass(C, D)	*superinterface*(C, D)	*assignable*(C, D)
extends(C, D)	*implements*(C, D)	

Figure 3: Built-in Predicates on Declared Types

In general, a policy clause tells whether or not a target O grants (resp. denies) access rights $r_1, \ldots, r_k$ to a subject S. When the optional *to*-phrase is omitted, the rights are granted (resp. denied) categorically. An optional condition c may also be supplied to specify when the policy clause is applicable (not applicable). The condition c is a first-order predicate in O and S. The ISOMOD policy language predefines a number of built-in connectives, predicates and functions for expressing complex applicability conditions. ISOMOD also provides a simple mechanism for policy programmers to define application-specific predicates and functions. A sample of built-in predicates is given in Figure 3. A sample of built-in functions is given in Figure 4.

Prior to the definition of a declared type [22, Section 5.3], its classfile is examined by the ISOMOD class loader for conformance to the corresponding ISOMOD policy. To this end, the set of all accesses in which the classfile (or one of its declared methods) is a subject is first collected. Each access is then checked according to the authorization algorithm outlined in Figure 5: the policy clauses are examined in the order they appear in the policy, and the authorization decision of the first applicable policy clause is then adopted. (A default authorization decision can be specified by the user of ISOMOD to handle the case when no policy clause applies.) In this process, if any access is denied by the policy, the definition of the declared type will not be authorized.

formal-parameters(M)	a list containing the formal parameters of method M
return-type(M)	the return type of method M
field-type(F)	the field type of field F
package(C)	the package to which declared type C belong
equals(O_1, O_2)	equality test

Figure 4: Built-in Functions on Declared Types, Methods, and Fields

```
To decide if access $\langle S, O, r \rangle$ is granted by policy $P$:
  for each rule $R$ in policy $P$ do
    if $R$ is relevant to $\langle S, O, r \rangle$ then
      if $c$ is true then
        if $R$ is a when-rule then
          if $R$ is an allow-rule then
            return grant;
          else // $R$ is a deny-rule
            return deny;
      else // $c$ is false
        if $R$ is an unless-rule then
          if $R$ is an allow-rule then
            return grant;
          else // $R$ is a deny-rule
            return deny;
  return user-specified default;
```

Figure 5: Operational Semantics for Authorization

Simple as it is, the ISOMOD policy language is capable of expressing a rich family of access control policies, a topic to which we will now turn.

4 Sample Applications

The goal of this section is to demonstrate the utility of the ISOMOD policy language through examples. The first example demonstrates how to selectively hide system services (Section 4.1). The second example illustrates how to systematically control the acquisition of references, and to do so in a way that accommodates evolving software configurations (Section 4.2). Finally, the ISOMOD policy language is employed to completely encode a capability policy language, thereby demonstrating the expressiveness of the policy language (Section 4.3).

4.1 Selective Hiding of System Services

ISOMOD can be used to enforce many of the `BasicPermissions` defined in the Java 2 platform [16]. For example, the `getClassLoader` permission controls whether untrusted code may acquire a `ClassLoader` reference from the platform library. The effects of denying this permission can be simulated by the IsoMod policy below:

policy getClassLoader
  default allow
method `ClassLoader.getParent`
  denies { invoke }
method `ClassLoader.getSystemClassLoader`
  denies { invoke }
method `Class.getClassLoader`
  denies { invoke }
method `Class Class.forName(String,boolean,ClassLoader)`
  denies { invoke }

The policy begins with a header that identifies the policy name and asserts that the default authorization decision is to allow access (i.e., when no policy clause applies). Next come the policy clauses, which disallow invocation of all methods declared in the Java platform library that returns a `ClassLoader`. Notice that one may either specify a method target solely by its name (e.g., `getClassLoader`), or by both its name and its type signature (e.g., `forName`[1]).

The related `createClassLoader` permission controls whether untrusted code may create new instances of the `ClassLoader` class. In the Java 2 platform, security checks are embedded in the constructors of `ClassLoader`, `SecureClassLoader` and `URLClassLoader` for ensuring that the caller possesses the said permission. Denying the `createClassLoader` permission can be simulated with the following policy clause:

method $C.M$
  denies { invoke }
    when *constructor*(M) **and** *subclass*$(C,$ `ClassLoader`$)$

Notice that this policy clause is more general than the ones aforementioned: it is applicable to any constructor M of a class C that is either `ClassLoader` or one of its subclasses (i.e., the predicate *constructor* tests if a method is a constructor, and the binary relation *subclass* is the reflexive transitive closure of the *extends* relation). Specifically, constructor invocation is denied. This rules out all means of creating `ClassLoader` instances.

The following is an alternative policy clause that achieves the same effect.

class C
  denies { new }
    when *subclass*$(C,$ `ClassLoader`$)$

Rather than controlling the invocation of `ClassLoader` constructors, this policy clause directly disallows the creation of new `ClassLoader` instances.

Most `BasicPermissions` defined in the Java 2 platform can be expressed declaratively by IsoMod. There is, however, a clear software engineering advantage to the IsoMod approach. Consider what is required in implementing and maintaining a Java 2 `BasicPermission`. One

[1]The `forName` method is denied because untrusted code may pass in a `null` `ClassLoader` reference to access the bootstrap `ClassLoader`.

has to inspect the entire Java 2 platform library to identify all points of attack, and then interpose monitoring code at each point. When a vulnerability is found, library source code has to be modified. In the ISOMOD example above, an exhaustive audit of the platform library is still necessary, yet the maintenance path is far superior: the policy is expressed declaratively and maintained independently: e.g., in a policy file separate from the library source code.

The ISOMOD approach provides a way to enforce fine-grained access control policies not expressible by the Java 2 permission system. Suppose we are to prevent untrusted code from using the *Reflection API* to invoke methods, access fields and arrays, and create new object instances, but we want to permit the examination of class interfaces. The existing permissions defined in Java 2 are not sufficient for expressing this highly selective policy: the entire Reflection API must be turned on or off as a whole. However, there is no problem constructing ISOMOD policy clauses to selectively hide the following reflection services: (1) *method invocation*: `Method.invoke`; (2) *field access*: the `Field.get/set` family of methods; (3) *array access*: the `Array.get/set` family of methods; (4) *object instantiation*: `Class.newInstance`, `Constructor.newInstance`, `Array.newInstance`, `Proxy.newProxyInstance`; (5) *subtyping*: `Proxy.getProxyClass`.

4.2 Systematic Control of Reference Acquisition

In the `createClassLoader` example discussed in Section 4.1, we could have formulated the following rule to deny the instantiation of new `URLClassLoader` instances:

method `URLClassLoader.newInstance`
  denies { invoke }

We did not impose this policy clause because such a restriction is not part of the semantics of the `createClassLoader` permission. Yet, this observation reveals a general challenge in policy formulation. Suppose we want to eliminate all means by which untrusted code may acquire a `ClassLoader` instance (that is, either by retrieving an existing instance or by creating a new one). An exhaustive audit of the platform library must be conducted to ensure all means of leaking `ClassLoader` references are accounted for. Not only is this an error-prone approach, it does not account for many useful configuration management practices: What if non-standard platform extension libraries are installed? What if ISOMOD is used for isolating dynamically downloaded plug-ins of an extensible application? Platform extensions and application classes may expose additional means of leaking `ClassLoader` references. To ensure that the access control policy is bullet proof, even a minor perturbation of the software configuration will necessitate a re-audit of the software infrastructures. Such a practice is too costly to be feasible.

A major contribution of ISOMOD is that it offers an expressive and declarative policy language that addresses the aforementioned configuration management challenge in access control. We demonstrate this feature by producing an ISOMOD policy that systematically restricts the acquisition of `ClassLoader` references. To this end, we begin by exhaustively enumerating all means, excluding those involving array types, by which a reference of declared type A may *acquire* a reference of type C:

1. A declared type A *generates* a reference of type C when one of the following occurs: (a) A creates an instance of C; (b) A casts a reference to type C; (c) an exception handler in A with catch type C catches an exception.

2. A declared type B *shares* a reference of type C with declared type A when one of the following occurs: (a) A invokes a method declared in B with return type C; (b) A reads a field declared in B with field type C; (c) B writes a reference into a field declared in A with field type C.

3. A declared type B *grants* a reference of type C to declared type A when B invokes a method declared in A, passing an argument via a formal parameter (including the pseudo-parameter `this`) of type C.

Based on the analysis above[2], we formulate the following policy clauses to prevent untrusted code from acquiring a `ClassLoader` reference:

```
policy acquireClassLoader
  default allow
class $C$
  denies { new, cast, catch }
    when subclass($C$, ClassLoader)
field $C.F$
  denies { get, put }
    when subclass(field-type($F$), ClassLoader)
method $C.M$
  denies { invoke }
    when subclass(return-type($M$), ClassLoader)
method $C.M$
  denies { invoke }
    when exists $A$ in parameter-types($M$) : subclass($A$, ClassLoader)
```

The first policy clause eliminates all means of *generating* `ClassLoader` references. The second and third policy clauses eliminate all means of *sharing* `ClassLoader` references. The last policy clause eliminates all means of *granting* `ClassLoader` references. Built-in functions such as *field-type*, *return-type* and *parameter-types* are employed to specify fine-grained accessibility criteria. The use of existential quantification (**exists**) is also demonstrated.

The policy above systematically restricts the acquisition of `ClassLoader` instances. Neither policy reformulation nor source code auditing is necessary even if the configuration of the underlying system has evolved.

4.3 Discretionary Capability Confinement

This section demonstrates how ISOMOD can be used for enforcing a general-purpose capability type system, *Discretionary Capability Confinement (DCC)* [12, 13]. A lightweight, statically enforceable type system, DCC supports the use of abstractly-typed object references as capabilities in a Java-like object-oriented programming language. A *capability* [8] is an object reference qualified by a set of access rights, where, the latter specify in what ways the underlying object can be accessed through the reference. Capabilities can be modeled in a language-based environment through a *capability type system*, in which every object reference is statically assigned a *capability*

[2]For brevity, the analysis does not account for array types, but such an extension is straightforward.

($\mathcal{DCC}1$) Unless $B \triangleright A$, A shall not invoke a static method declared in B.

($\mathcal{DCC}2$) (i) A can generate a reference of type C only if $C \triangleright A$; (ii) B may share a reference of type C with A only if $(C \triangleright A) \vee (A \bowtie B)$.

($\mathcal{DCC}3$) If $A.m$ invokes $B.n$, and C is the type of a formal parameter of n, then $(C \triangleright B) \vee (A \bowtie B) \vee (B \triangleright m \wedge C \triangleright m)$.

($\mathcal{DCC}4$) A method m may invoke another method n only if $n \triangleright m$.

($\mathcal{DCC}5$) If A is a subtype of B, then $B \triangleright A$.

($\mathcal{DCC}6$) Suppose $B.n$ is overridden by $B'.n'$. Then (i) $n' \triangleright n$; (ii) if the method return type is C, then $(C \triangleright B) \vee (B \triangleright B')$; (iii) if C is the type of a formal parameter, then $(C \triangleright B') \vee (B \bowtie B')$.

($\mathcal{DCC}7$) If A is a subtype of B, then $B :\triangleright A$.

($\mathcal{HMS}1$) $\top :\blacktriangleright \mathcal{D}$.

($\mathcal{HMS}2$) If $\mathcal{D} :\blacktriangleright \mathcal{E}$, then $\mathcal{D} \blacktriangleright \mathcal{E}$.

($\mathcal{HMS}3$) If $(\mathcal{D} :\blacktriangleright \mathcal{E}) \wedge (\mathcal{D}' \blacktriangleright \mathcal{E})$, then $(\mathcal{D} \blacktriangleright \mathcal{D}') \vee (\mathcal{D}' \blacktriangleright \mathcal{D})$.

Figure 6: DCC Type Constraints

type that restricts access to the underlying object. In a sense, a capability type presents a restricted view of the object it types. In a Java-like object-oriented programming language, an object reference with a static interface type (or abstract class type) can be seen as a capability, because the typed reference only exposes a restricted view of the underlying object. This approach to modeling capabilities suffers from two problems: capability leakage and capability theft [31]. DCC is a minimal perturbation to Java for controlling capability propagation. In the following, the expressiveness of ISOMOD is demonstrated through a complete encoding of the DCC type system in the policy language. The focus here is ISOMOD and not DCC. Interested readers may consult [13] for more details of DCC.

In DCC, the space of declared types (i.e., class and interface) is partitioned into a finite number of *confinement domains*, so that every declared type belongs to exactly one confinement domain. We write $l(C) = \mathcal{D}$ when declared type C is assigned to confinement domain $\mathcal{D}$. The confinement domains are partially ordered by a *dominance relation* $\blacktriangleright$. We say that domain $\mathcal{D}$ *dominates* domain $\mathcal{E}$ when $\mathcal{E} \blacktriangleright \mathcal{D}$. Together, domain membership and dominance induce a natural pre-ordering of declared types: if $l(B) = \mathcal{E}$, $l(A) = \mathcal{D}$, and $\mathcal{E} \blacktriangleright \mathcal{D}$, then we write $B \triangleright A$, and say that B *trusts* A. The intuition behind these definitions is that, if C trusts A, then A may freely acquire a reference of static type C. Otherwise, C is said to be a *capability* for A. Capability acquisition is carefully restricted in DCC. We also write $A \bowtie B$ when $A \triangleright B$ and $B \triangleright A$ hold simultaneously. We postulate that there is a *root domain* $\top$ which is dominated by every domain.

To control capability granting, associated with every method m is a domain label $l(m)$, called the *capability granting policy* of m. Intuitively, the capability granting policy $l(m)$ dictates what capabilities may be granted by m, and to which declared types m may grant a capability. (We write $m \triangleright n$, $m \triangleright A$ and $A \triangleright m$ for the obvious meaning.)

A second partial ordering $:\blacktriangleright$ on domains is postulated. We say that $\mathcal{D}$ *strongly dominates* $\mathcal{E}$ when $\mathcal{E} :\blacktriangleright \mathcal{D}$. As we shall see, strong dominance controls whether subtyping is allowed across domain boundaries. This helps to establish mutually exclusive roles. As before, we write $B :\triangleright A$ when $l(B) = \mathcal{E}$, $l(A) = \mathcal{D}$, and $\mathcal{E} :\blacktriangleright \mathcal{D}$.

Figure 6 enumerates the type constraints of DCC as specified in [13]. The complete set of DCC type constraints has been successfully encoded as an IsoMod policy, which is displayed in Appendix A. Behind the policy of Appendix A is a number of assumptions. As in [13], we assume that domain membership and capability granting policies are embedded in Java classfiles via the JDK 1.5 metadata facility. Domains are represented by specially annotated interfaces, and the dominance and strongly dominance relations are encoded, respectively, by the subinterfacing relation and JDK 1.5 annotations. Domain-specific functions and predicates have been defined to examine these annotations. In the following we will examine aspects of this encoding that illustrate further features of IsoMod.

Consider the following type constraint from Figure 6:

$(\mathcal{DCC}2)$ (i) A can generate a reference of type C only if $C \triangleright A$; (ii) B may share a reference of type C with A only if $C \triangleright A \vee A \bowtie B$.

In this constraint, the first clause denies the generation of capabilities, and the second clause denies the sharing of capabilities with reference types belonging to a different confinement domain. This constraint can be encoded as the following IsoMod policy clauses:

class C
  denies { catch, cast, new } **to method** $A.M$
    unless *trusts*(C, A)
method $B.N$
  denies { invoke } **to method** $A.M$
    unless *trusts*(*return-type*(N), A) **or** (*trusts*(A, B) **and** *trusts*(B, A))
field $B.F$
  denies { get } **to method** $A.M$
    unless *trusts*(*field-type*(F), A) **or** (*trusts*(A, B) **and** *trusts*(B, A))
field $B.F$
  denies { put } **to method** $A.M$
    unless *trusts*(*A*, *field-type*(F)) **or** (*trusts*(A, B) **and** *trusts*(B, A))

Two additional features of IsoMod are demonstrated in the above policy clauses. Firstly, IsoMod provides a syntax (i.e., **to**) for qualifying to which subject a policy clause applies. As capability acquisition is permitted for some subjects but not others, this discrimination enables fine-grained access control. Secondly, IsoMod supports user-defined predicates and functions for modeling domain-specific relations. For example, *trusts* is a user-defined predicate for representing the binary trust relation between declared types.

Let us consider another type constraint from Figure 6:

$(\mathcal{HMS}3)$ If $(\mathcal{D} :\blacktriangleright \mathcal{E}) \wedge (\mathcal{D}' \blacktriangleright \mathcal{E})$, then $(\mathcal{D} \blacktriangleright \mathcal{D}') \vee (\mathcal{D}' \blacktriangleright \mathcal{D})$.

This constraint is the soul of a property known as *hereditary mutual suspicion* [13], which enforces a strong form of separation of duty [6, 20], so that collusion between mutually exclusive roles is severely restricted. The constraint mandates that, given an arbitrary domain $\mathcal{E}$, some form of dominance relation must exist between a domain $\mathcal{D}$ strongly dominated by $\mathcal{E}$ and a domain $\mathcal{D}'$ dominated by $\mathcal{E}$. An IsoMod encoding of it is given below:

```
class $C$
  denies { extend } to class $\mathcal{E}$
    unless
      domain($\mathcal{E}$) implies
        for $\mathcal{D}$ in strongly-dominated($\mathcal{E}$) :
          for $\mathcal{D}'$ in dominated($\mathcal{E}$) :
            dominates($\mathcal{D}, \mathcal{D}'$) or dominates($\mathcal{D}', \mathcal{D}$)
```

Our goal is to check $(\mathcal{HMS}3)$ exactly once for every domain $\mathcal{E}$. To this end, we observe that, at the bytecode level, every declared type extends exactly one superclass, with `java.lang.Object` being the only, uninteresting exception. We therefore "schedule" the verification of $(\mathcal{HMS}3)$ to occur when $\mathcal{E}$ extends some dummy class C. The same technique is used in the encoding of $(\mathcal{HMS}1)$ and $(\mathcal{HMS}2)$ (see Appendix A).

Besides DCC, we have also completely encoded the class-based access control mechanism of Java [22, Section 5.4.4] (i.e., public, protected, private, etc) as an IsoMod policy (Section 6). These examples demonstrate the expressiveness and versatility of the IsoMod policy language.

5 Implementation Experience

IsoMod has been fully implemented in approximately 10,000 lines of pure Java code. This section reports the implementation experience. The technical challenges encountered and the solutions adopted to address these challenges are discussed. Enough details of the IsoMod load-time verification mechanism are given so that its design can be reused in future work involving load-time type checking.

5.1 Design Rationales

We begin by identifying the technical challenges our implementation strategy attempts to address.

1. **Efficiency.** Class loading and policy evaluation incur a significant link-time overhead, slow down application start-up, and should thus be minimized.

2. **Early enforcement.** Class definition [22, Section 5.3.5] is irrevocable. Policy enforcement must be complete before a classfile is converted into a `Class` object.

3. **Circularity.** Circular dependency between type interfaces may arise from forward references. Policy evaluation must handle circularity gracefully.

4. **Attribution correctness.** Policy violation should be attributed to the offending classfiles (i.e., subjects) rather than the offended classfiles (i.e., targets). Only the definition of the subjects should be denied.

Initially we considered three possible designs, and eventually adopted one that adequately addresses all of the aforementioned technical challenges. The difference between the three designs lies in when external dependencies are resolved. Specifically, before a class is defined, its corresponding classfile is *preloaded* and screened for policy violation. Policy checking may involve the examination of external classes, as the following example (taken from Section 4.2) illustrates:

method $C.M$
  denies { invoke }
    when *subclass*(*return-type*(M), `ClassLoader`)

Checking whether a declared type A conforms to this policy clause involves examining both C and *return-type*(M). These latter declared types may not have been preloaded yet. The three designs differ in how this situation is handled.

Design #1: Eager preloading. Preload C and *return-type(M)* right away. While this design is easy to implement, the resulting class loading overhead will likely be prohibitive.

Design #2: Constraint maintenance. Lazy preloading can be achieved by constraint maintenance. That is, the definition of A is authorized right away, but the policy clause is recorded as a proof obligation, the discharging of which is deferred until C and *return-type*(M) are preloaded at a later time. This design eschews eager preloading, but its implementation is complex. This design has been adopted by Sun's implementation of the JVM for enforcing Temporal Namespace Consistency [21]. The most serious problem with this approach is its inability to correctly attribute policy violation to the offender. When the checking of the proof obligation fails at the time C or *return-type*(M) is preloaded, the only thing a system could do is to deny the definition of C and *return-type*(M), an incorrect attribution of policy violation.

Design #3: Three-stage, lightweight preloading. Preloading is divided into three stages (Figure 7), which incrementally constructs and maintains a lightweight representation of the classfile being preloaded. In the first stage, references to the classfile's supertypes are resolved, and all "extend" and "implement" accesses are checked. In the second stage, type references appearing in the type interface of a classfile are resolved, and then all "override" accesses are checked. In the third stage, type references appearing in the remaining external accesses are resolved, and then those accesses are checked. This design performs shallow preloading eagerly, but maintains lightweight type mirrors to anticipate deep preloading. This design implements a lazy preloading strategy, carefully breaks circularity, and correctly attributes policy violations to the offending classfiles. The preloading algorithm is detailed in Figure 7. This design is informed by previous work in modular bytecode verification in the presence of lazy, dynamic linking [14, 11].

The rest of this section provides details concerning Design #3, which was adopted in the implementation of IsoMod.

5.2 Type Mirrors

This section describes the lightweight representation of declared types created by IsoMod. This lightweight representation is called a *mirror* in the programming language literature [5]. More

Stage-I Preloading of Declared Type C

1. Retrieve the classfile of C.
2. Perform stage-I preloading on the supertypes of C. Circular subtyping is detected.
3. Check the "extend" and "implement" accesses of C.
4. Cache a lightweight representation of C, recording its type interface and the remaining external accesses.

Stage-II Preloading of Declared Type C

1. Perform stage-I preloading on C.
2. Perform stage-II preloading on the supertypes of C.
3. Perform stage-I preloading on the return types and parameter types of methods declared in C, and the field types of fields declared in C.
4. Check the "override" accesses of C.

Stage-III Preloading of Declared Type C

1. Perform stage-II preloading on C.
2. Perform stage-III preloading on the supertypes of C.
3. Perform stage-II preloading on those type references appearing in the targets of the remaining external accesses associated with C.
4. Check the remaining external accesses of C.
5. Authorize the definition of declared type C.

Figure 7: Preloading Algorithm

specifically, IsoMod maintains a mirror object for each declared type, method, field, and external access. The following presents the layout of these data structures and the incremental process by which they are constructed.

5.2.1 Structure of Mirrors

A *classfile* is a file format used by Java to represent object code in a file system. Each classfile describes a declared type (i.e., a class or an interface), including its methods, fields, and bytecode instructions. In order to perform verification, the preloading process converts a classfile into its respective mirror object, called a *declared type mirror*, or simply *type mirror*, the structure of which is described below. Notice that the following describes the data structures as they are fully constructed. The intermediate steps are left to Section 5.2.2.

Type mirror. A type mirror records the type interface of a declared type.

Class name: The name of this declared type

Super class**:* The type mirror representing the direct super class of this declared type

Interfaces**:* The type mirrors of the interfaces directly implemented by this declared type

Declared methods**:* The method mirrors of the methods (including constructors) declared in this declared type

Declared fields**:* The field mirrors of the fields declared in this declared type

External accesses*: The access mirrors corresponding to the external accesses made by this declared type (e.g., as reflected in the bytecode instructions)

Method mirror. A method mirror describes the signature of a method.

Method name: Name of this method

Return type**: The type mirror representing the return type of this method

Parameter types**: The type mirrors representing the parameter types of this method

Defining class*: The type mirror of the declared type in which this method is declared

Field mirror. A field mirror describes the signature of a field.

Field name: Name of this field

Field type**: The type mirror representing the type of this field

Defining class*: The type mirror of the declared type in which this field is declared

Access mirror. An access mirror represents an external access, which is made up of a triple $\langle subject, object, right\rangle$. Such a data structure is recorded to facilitate lazy verification.

Subject*: The subject of this external access, which can be either a type or method mirror

Object***: The object (i.e., target) of this external access, which can be either a type, method or field mirror

Right: A particular right exercised by this access (Figure 2)

5.2.2 Staged Construction of Mirrors

This subsection describes the incremental process by which the aforementioned mirrors are constructed. The mirror objects contain references to other mirror objects. These references are indicated by one or more asterisks (*, **, or ***) in Section 5.2.1. For example, a type mirror contains a reference to the mirror of its direct super class; an access mirror contains a reference to the mirror object representing the target of the external access. During the preloading process, some of the references are not fully resolved, because the targets of the references have not been constructed (or preloaded) yet. In these cases, a symbolic reference (e.g., a string) is recorded in place of the actual mirrors. The following explains when mirrors are constructed and how symbolic references are resolved.

Stage I. This stage initiates the construction of mirror objects for a given classfile. Another goal is to incrementally construct a *type mirror hierarchy* to facilitate *subtyping queries*. When a declared type is preloaded in Stage-I, mirrors of methods declared in that declared type are created. At this point, the bytecode of the method body is scanned, and all external accesses are recorded as access mirrors. If the same access occurs multiple times in the method body, only a single instance is recorded. The newly created method mirror is linked to the type mirror, and vice versa. Similarly, a field declared in the declared type has its respective field mirror constructed and properly linked. The direct supertypes (super class and super interfaces) are resolved into their respective type mirrors through recursive Stage-I Preloading (Figure 7, Stage I, Step 2). In short, all the mirror references annotated with a single asterisk (*) in Section 5.2.1 are resolved in this stage. By the end of Stage-I, the basic structure of the type mirror is established, its external accesses recorded, and the part of the type mirror hierarchy above this type mirror fully constructed.

Stage II. This stage is responsible for resolving the symbolic references appearing in method and field mirrors: i.e., the ones annotated with double asterisks (**) in Section 5.2.1. The goal of this stage is to make the type interface of a given type mirror fully usable. Specifically, the method and field mirrors of the type mirror are examined. Symbolic references corresponding to the return type and parameter types of a method mirror are resolved through Stage-I Preloading. The method mirror is updated, by linking the respective components to their mirror counterparts. The field types of field mirrors are updated in a similar fashion. Notice that Stage-II Preloading is performed on the mirrors of supertypes before it is performed on a given type mirror (Figure 7, Stage II, Step 2). This means, not only are the mirrors of methods and fields declared in the type mirror fully constructed in Stage II, the mirrors of those methods and fields *inherited* by the type mirror are also fully constructed by then. By the end of Stage-II, the type interface of the type mirror is fully constructed to facilitate *type interface queries*.

Stage III. This stage is responsible for resolving the targets of external accesses: i.e., those marked by triple asterisks (***) in Section 5.2.1. The goal is to resolve the symbolic references embedded in access mirrors, so that the external accesses can be authorized prior to class definition [22, Section 5.3.5]. To this end, the collection of access mirrors associated with a given type mirror are scanned. For every access mirror, each of its targets is examined. If the target is a type reference (e.g., as in the case of a “new” access), the reference is resolved by Stage-II Preloading. If the target is a method (resp. field) reference (e.g., as in the case of an “invoke” access), then such a reference will be of the form $c.m$ (resp. $c.f$), where c is a symbolic reference to a declared type, and m (resp. f) is a method signature (resp. field signature) consisting of a method name plus return type and parameter types (resp. field name plus field type). The type reference c is resolved into a type mirror C by Stage-II Preloading, after which the proper method mirror (resp. field mirror) is located in either C or one of C’s supertype mirrors. This resolution process closely parallels that of [22, Section 5.4.3]. This final step concludes the staged construction process for a type mirror.

5.3 Staged Verification

Not all external accesses are verified at once (Figure 7). The “extend” and “implement” accesses are discharged in Stage I, “override” in Stage II, and the rest in Stage III. Why is the verification process divided into three stages? And why is a given check scheduled to occur in a specific stage? These are the questions the present section attempts to answer. As we shall see, the three-stage design facilitates lazy preloading and gracefully handles circularity.

Stage I. Recall that this stage gathers subtyping information for a given declared type. Such subtyping information must be verified before it can be queried. Firstly, circular subtyping is detected by checking if a type mirror argument re-appears in the call chain of recursive Stage-I Preloading. Secondly, access mirrors corresponding to “extend” and “implement” rights are verified in this stage against the current policy. The completion of this stage ensures that subtyping queries are supported by properly verified subtyping information.

Notice as well that circular Stage-I Preloading is prevented because circular subtyping is ruled out.

Stage II. Recall that by the end of Stage II, the type interface of a type mirror is fully constructed to support type interface queries. The type interface of a declared type extends the type interfaces of its supertypes through inheritance and method overriding. To ensure type extension is proper, the "override" accesses are checked in this stage for policy compliance. This ensures that type interface queries are supported by a properly verified type interface.

Two kinds of recursive preloading are performed in this stage. Firstly, Stage-II Preloading is performed on the type mirrors of the supertypes. This does not lead to circular preloading because circular subtyping has already been ruled out in Stage I. Secondly, Stage-I Preloading is invoked on type references embedded in method and field mirrors. This does not lead to circular preloading because Stage-I Preloading never invokes Stage II Preloading.

Stage III. Recall that the third and final stage is responsible for resolving the symbolic references embedded in the targets of access mirrors, so that the latter can be discharged. Resolution is achieved by Stage-II Preloading. As a result, the following queries are supported:

- Type interface queries can be applied to the targets of access mirrors.
- Subtyping queries can be applied to the type mirrors embedded in the type interface of the above access targets.

Observe that these are the very queries that the IsoMod policy language is designed to support. The remaining access mirrors are therefore ready for verification.

Again, two kinds of recursive preloading is performed in this stage. Firstly, Stage-III Preloading is performed on the type mirrors of the supertypes. This does not lead to circular preloading because of the absence of circular subtyping. Secondly, Stage-II Preloading is invoked on type references embedded in the targets of access mirrors. This, again, does not lead to circular preloading because Stage-II Preloading never initiates Stage-III Preloading.

Efficiency. Observe from the above discussion that access mirrors are only checked when it is absolutely necessary to do so. To check an access mirror, type interface queries will be made against the target, and thus it undergoes Stage-II Preloading. Similarly, subtyping queries will be made against the type references embedded in the type interface of the access target, so Stage-II Preloading guarantees that such references are resolved via Stage-I Preloading. In this way, preloading is minimized to improve preloading efficiency.

Circularity. The staged preloading and verification process gracefully handles the two possible types of circularity. Subtyping circularity is managed in Stage I by ensuring that no circular subtyping occurs. Circularity arising from forward referencing is handled by stratification of the preloading process: Stage I never invokes Stage II and III; Stage II never invokes Stage III. This effectively breaks any possibility of circularity.

5.4 Policy Evaluation

When an access mirror is checked against an IsoMod policy, the policy clauses are examined in turn. IsoMod evaluates each of the policy clauses to determine if the clause is applicable to the access. Applicability is specified in the IsoMod policy language via the **when**/**unless** construct, which states a binary relation between the subject and the target. This binary relation is expressed in the form of a first-order formula with free variables denoting the subject and the target. Determining the applicability of a policy clause involves binding the subject and target variables, respectively, to the subject and target mirrors, and subsequently deciding if the mirror hierarchy that has been preloaded so far is a *model* for the formula. That is, the first-order formula is evaluated against the subtyping and type interface information that IsoMod has collected so far. This can be achieved easily by a syntax-directed evaluation of the abstract syntax trees of the formulas [15].

A natural question to ask is whether the implementation strategy described in this section delivers its performance promise, a topic to which we will now turn.

6 Performance Evaluation

This section reports the results of experiments conducted for profiling the performance characteristics of IsoMod. The goal of this study is threefold:

1. **Tractability.** To quantify the performance overhead incurred by executing an application through IsoMOD.

2. **Overhead composition.** To identify the major components of the IsoMod implementation which contribute to the overhead incurred.

3. **Overhead distribution.** To characterize the way in which the overhead incurred by IsoMod is distributed over the life cycle of an application.

These characteristics are important in validating IsoMod as a viable approach to software security. The Tractability Study in Section 6.2 measures the overhead incurred by using IsoMod. A quantifiable result was obtained through this study by comparing the running times of a test suite with and without IsoMod The analysis described in Section 6.3 further examined the results gathered from the Tractability experiment. The purpose of this analysis was concerned with discovering the makeup of IsoMod's overhead. The above analysis suggested that the majority of the overhead occurred during application startup. Section 6.4 describes an experiment designed to discover how a longer executing system would experience overhead as opposed to that of a shorter lived execution.

The results of the experiments and analysis aided in validating IsoMod as a viable approach to software security. The first experiment demonstrated that the overhead incurred by IsoMod was reasonable and manageable. The analysis of this experiment discovered IsoMod's overhead to be comprised of two major parts. The first part consisted of maintaining the type mirror hierarchy and the second was authorization of the access mirrors. The second experiment demonstrated that a longer executing system caused IsoMod's overhead to amortize away. That is, a longer executing system experienced less of an overhead encumbrance as opposed to that of a shorter lived system.

6.1 Methodology

The experiments were conducted on a system with the following characteristics:

- 3.0 GHz Pentium 4
- 1 Gigabyte of RAM
- Fedora Core 4 Linux Operating System
- Java Standard Edition, version 1.5

Accompanying software included BCEL (Byte Code Engineering Library), version 5.1.

We used a benchmarking suite consisting of six open source Java applications (Figure 8). These applications were chosen because they could be executed in batch mode without real-time interaction from users. The applications were tested under five configurations. The first configuration, *Control*, runs an application in a bare JVM. The other four configurations consisted of loading an application with an IsoMod class loader with one of four IsoMod policies. The policies used in the configurations consisted of:

NULL. Contains no policy clauses, and the default authorization decision is to allow all accesses. See Figure 9.

AllowAll. Contains policy clauses allowing access for every right occurring within the JVM. See Figure 10.

DCC. An IsoMod encoding of the Discretionary Capability Confinement capability type system. See Appendix A.

JAC. Encoding of the Java access control mechanism as specified in [22, Section 5.4.4]. See Figure 11.

Five trials[3] were repeated for each configuration to account for variability in the Operating System. Factors such as process scheduling and context switching may have contributed variations to the results. The average running time (in seconds) was recorded for each configuration.

In Section 6.4, the Java application JavaTar was used for testing under ten configurations. The configurations consisted of loading the application with the IsoMod class loader with the *DCC* policy and input data varying in size from 24KB to 495MB. The *DCC* policy was chosen for its complexity, which would incur significant overhead.

6.2 Tractability

This experiment quantified the overhead incurred by the IsoMod class loader. A classfile loaded by the IsoMod class loader is subjected to the verification process described previously. During this verification process, the complete collection of access mirrors is collected. These access mirrors are authorized against a policy. A classfile that passes all stages of verification is turned into a `Class` object and loaded into the JVM.

[3]Only five trials were used because the standard deviation and variance in the experimental results were found to be small.

Name	Description	Number of Classes
ZipDiff	Utility to compare the contents of two zip files	8
JavaTar	A TAR compression program written in Java	43
JavaCC	Sun's Compiler-Compiler	135
SableCC	A compiler-compiler	263
JRuby	A Java implementation of the Ruby language	473
Kawa	A Scheme interpreter written in Java	746

Figure 8: Benchmarking Suite

policy AllowAll
  default allow

Figure 9: *NULL* Policy

policy AllowAll
  default allow

class C
  allows { new, extend, implement, catch, class_cast, instanceof, array } **to class** D
    when *true*

method $C.m$
  allows { invoke, inherit, override } **to class** D
    when *true*

field $C.f$
  allows { get, put, inherit } **to class** D
    when *true*

Figure 10: *AllowAll* Policy

```
policy JAC
  default allow

class $C$
  allows { new, catch, class_cast, instanceof, extend, implement, array } to class $D$
    when $public(C)$ or $(pkg(C) = pkg(D))$

field $C.F$
  allows { get, put } to class $D$
    when
      $(public(C)$ or $(pkg(C) = pkg(D)))$
       and $((public(F)$ or $(protected(F)$ and $subclass(D, C)))$
         or $(((protected(F)$ or $pkg\text{-}private(F))$ and $(pkg(C) = pkg(D)))$ or
           $(private(F)$ and $(C = D))))$

method $C.M$
  allows { invoke } to class $D$
    when
      $(public(C)$ or $(pkg(C) = pkg(D)))$
       and $((public(M)$ or $(protected(M)$ and $subclass(D, C)))$
         or $(((protected(M)$ or $pkg\text{-}private(M))$ and $(pkg(C) = pkg(D)))$ or
           $(private(M)$ and $(C = D))))$
```

Figure 11: Java Access Control (*JAC*) Policy

The results of the experiment are shown in Figure 12. This figure describes the run-time overhead incurred by the IsoMod class loader for each application under the five configurations. From this figure it can be seen that the *NULL* configuration incurred the least amount of overhead, while *DCC* incurred the most. The other two configurations, *AllowAll* and *JAC* incurred almost equal amounts of overhead. The values of running time are shown in seconds and represent the total time for an execution.

A number of observations can be made about these results. First, the graph describes how much overhead is incurred when an application executes through the IsoMod class loader. Regardless of whether a policy is enforced or not, IsoMod incurs some overhead. This fact is demonstrated by the *NULL* configuration, where no policy clauses exist in the policy. Second, as the policies become more complex, the run-time overhead increases, as shown by the *DCC* configuration. With the introduction of more policy clauses, and more complex applicability conditions, more time must be given to authorize the accesses described within the mirror. Third, the total overhead for any application never exceeded three seconds. In the process of a longer executing application such as SableCC which executed for just over ten seconds, three seconds of additional overhead may be considered reasonable.

Overhead while executing through IsoMod cannot be avoided, but it has been demonstrated in the experimental results that the overhead is manageable and reasonable. By enforcing more complex policy clauses, such as *DCC*, the overhead increases. This particular result is expected as more conditional applicability checks must be made by the verification process. In our experiments with larger applications that execute longer, the overhead did not prove to be much of an encumbrance, since it never exceeded three seconds.

6.3 Composition of Overhead

Analysis of the above results revealed two major components comprising IsoMod's overhead:

1. **Mirror maintenance.** the process involved in constructing mirror objects described in a classfile. This process involved loading the classfile, constructing mirror objects for each method and field in the type interface. Supertypes undergo a similar construction. External accesses described in the classfile are examined, and corresponding access mirror objects are constructed and recorded. The overhead of this process can be easily measured by subtracting the running time of the *Control* configuration from the running time of the *NULL* configuration. This is because the *NULL* configuration contains no policy clauses. With no policy clauses, access mirrors are not subjected to applicability evaluation. The access mirrors are iterated through, but since no IsoMod policy clauses exist, all accesses will be immediately allowed. Since no access evaluation occurred, IsoMod only performed *Mirror Maintenance*.

2. **Access authorization.** this is the process of authorizing access mirrors against IsoMod policy clauses. Each access collected for a given classfile must be authorized against relevant policy clauses. These clauses can involve applicability of conditions of various complexity. Complex conditions require more time to evaluate when authorizing access mirrors. This process can be measured by taking the difference in the running time between the *NULL* configuration and a non-trivial configuration, say, *DCC*. The *NULL* configuration takes into

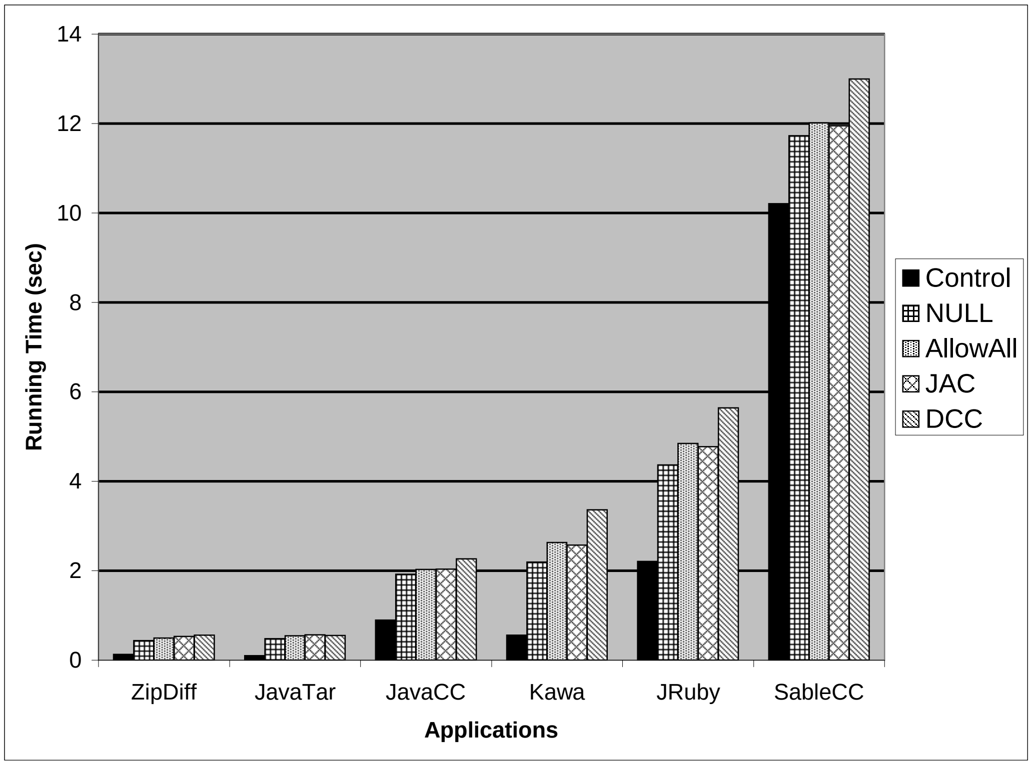

Figure 12: IsoMod Overhead

account the time spent on *Mirror Maintenance*. *DCC* includes complex access mirror applicability checks as well as *Mirror Maintenance*. By removing the time spent on the *NULL* configuration, we are left with the time spent on *Access Authorization*.

Figure 13 depicts the percentage of time spent for each component of the overhead over the complete cycle of execution. It can be seen that most of the time is spent on *Mirror Maintenance*. On average, *Mirror Maintenance* required 68% of the time, while only 32% was needed for *Access Authorization*.

Also shown in Figure 13 is the ratio between the number of authorizations checked during a full execution and the number of preloadings performed. The latter figure is the total number of Stage-I, Stage-II, and Stage-III classfile preloads actually performed during a full execution.

This ratio relates the number of authorizations to the number of declared types that were preloaded. This ratio is associated with the composition of the overhead in a direct manner. As the number of access mirrors that are authorized increases, then so does the percentage of time spent on performing this process. This particular aspect can be seen directly when looking at the JavaTar configuration up to the SableCC configuration. The ratio is increasing, as does the percentage of processing time spent on *Access Authorization*. This direct correlation indicates that the number of accesses being authorized will have an impact on how much time is spent on *Access Authorization*.

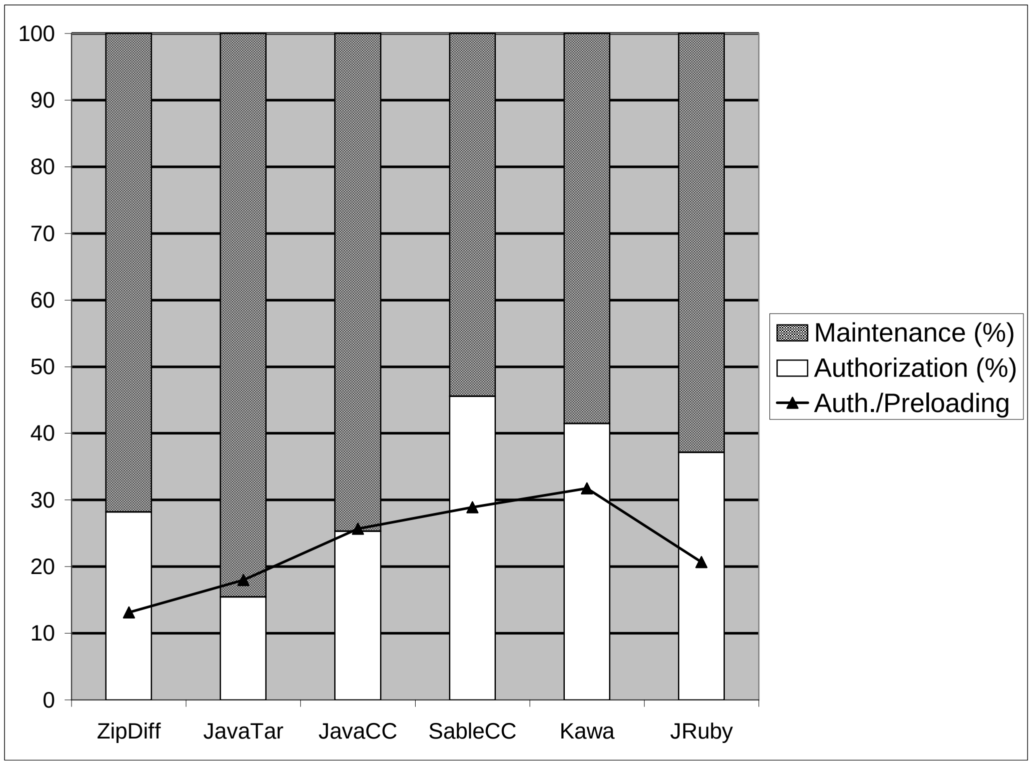

Figure 13: Composition of Overhead

6.4 Overhead Distribution

The goal of this experiment was to determine the percentage of the running time devoted to overhead during a long execution. Figure 14 describes *Penalty of Overhead*, as the percentage of running time given to IsoMod during a complete execution. The size of the input is given in logarithmic scale because the input size varied from 24KB up to 495MB.

The JavaTar application creates an archive of files. There are two main reasons for choosing JavaTar in this experiment. First, the applications execution time could be varied, by varying the number of files to be archived. Second, since JavaTar was only operating on multiple sets of data, the number of mirrors constructed and the number of access mirrors recorded would be identical for all configurations. Third, JavaTar's performance during the first experiment (Section 6.2) was found to have the highest percentage of run-time devoted to overhead (although the magnitude of the overhead was small). Executing JavaTar through IsoMod with the policy *DCC* was found to have 81% of the execution time dedicated to overhead.

The graph in Figure 14 clearly demonstrates that as the input size was increased, the percentage of run-time spent on overhead decreased. The results of the experiment clearly indicated how a longer executing application reacted when executed through IsoMod. Through each execution, the overhead incurred by IsoMod was constant. This was because the policy used was never changed. The difference in execution times arises from varying the size of the input data.

The results of this experiment were as expected. Because the number of mirrors constructed and number of access mirrors authorized remained constant for each configuration, the amount

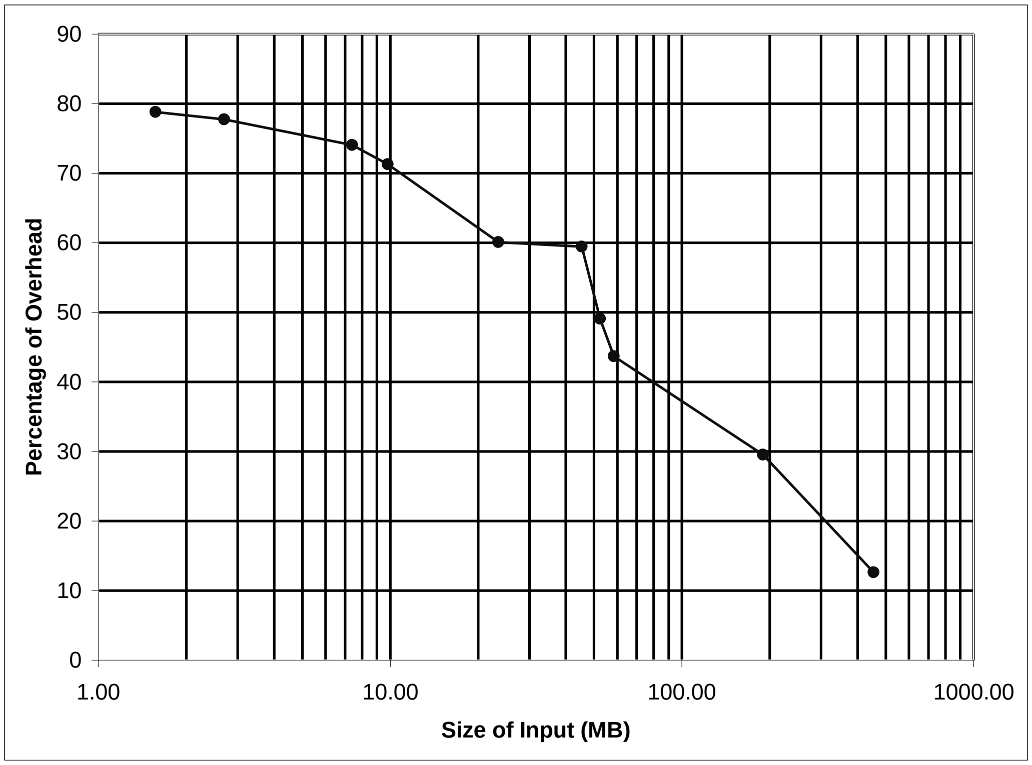

Figure 14: Penalty of Overhead

of overhead incurred by IsoMod remained constant and independent of the input data. This particular phenomenon of the overhead amortizing away would be even more visible in longer running applications, such as those found on servers. One could therefore enforce more complex policies on longer running application with a manageable overhead. The overhead incurred by enforcing these more complex policies would also be amortized away, as was demonstrated in this experiment.

Summary. Performance is one factor among many when determining if an approach to language-based access control is viable. Since these experiments have demonstrated that the overhead incurred by IsoMod is manageable and reasonable, they provide evidence of the usefulness of IsoMod as a language-based access control mechanism. The IsoMod class loader efficiently preloads and discharges accesses with small impact on the total execution time of an application. The most complex policy tested, *DCC*, incurred at most three seconds of overhead, which is reasonable. The authorization of accesses against policies was found to be minimal when compared to that of mirror maintenance. Because so little time is spent on authorization, more complex policies could be enforced with minor impact on the overhead. These results paved the way for exploring how IsoMod behaved in a longer executing system. An application using the IsoMod class loader that executed for a longer period of time had a lower percentage of time devoted to overhead. IsoMod only incurred overhead during application startup, and this overhead amortizes away during a long run.

7 Concluding Remarks

7.1 Limitations

Enforcement mechanisms that are based solely on static analysis, of which IsoMod is an example, are provably less powerful than those that also employ execution monitoring [17]. For example, policies in which authorization decision is a function of invocation arguments or execution history are not enforceable by IsoMod. This fact is demonstrated by IsoMod only being able to capture a subset of the Java 2 permissions, for which runtime information is not needed for enforcement. Our goal, however, is not to match the expressiveness of execution monitoring, but rather to find a lightweight alternative to interposition when full-fledged execution monitoring is not necessary.

The analysis of reference acquisition in Section 4 does not account for the acquisition of references through such Java platform facilities as reflection, (de)serialization and object cloning. To ensure that IsoMod functions properly one should formulate policy clauses to either completely disable such facilities, or selectively control their visibility. This can be easily achieved by the techniques demonstrated in Sections 4.1 and 4.2.

7.2 Related Work

As surveyed in Section 1, language-based software isolation has been achieved mostly by interposition-based mechanisms in the past. Early language-based systems such as Scheme 48 [26], Safe-Tcl [24] and SPIN [4] adopt *namespace management* as a primary protection mechanism. Two component mechanisms are involved. Firstly, dynamic linking dispatches monitoring code when system services are invoked. This is simply another form of interposition. A Java incarnation of linking-based interposition is described in [34]. Secondly, rudimentary name visibility control is employed to hide certain names in a namespace. None of the policy clauses in Sections 4.2 and 4.3 can be enforced in this manner. We have thus demonstrated that name visibility control can in fact be much more expressive than conventional belief.

The study of module systems has a long history [19]. We highlight some recent developments on the Java front. JavaMod [2] is a module system for Java-like languages. The interaction between modularity and subtyping is carefully articulated. Bauer *et al* [3] extend the Java package facility to obtain a module system that supports the decoration of import statements with linking obligations, which are in turn implemented as digital signatures. MJ [7] is a module system designed to control the complexity of configuration management in Java platforms. Liu and Smith [23] describe a module system that supports the declaration of bi-directional interfaces. Designed primarily for access control, IsoMod is unique in two ways: (1) name visibility constraints can be imposed dynamically; (2) fine-grained name visibility constraints can be expressed in the IsoMod policy language to control not only what names are visible, but also to whom and to what extent they are visible.

This work has been informed by the recent work in *encapsulation policies* [28, 27]. Specifically, the designer of a class A associates to A a fixed number of access control policies, each presenting a different view of A. A client class B then selects a policy through which B interacts with A. Three points of comparison are worth noting. Firstly, because the client decides which policy to adopt, the scheme cannot be used for protection. Secondly, policies are formulated on a per-class basis, the universally quantified access control rules described in Section 4 cannot be expressed. Thirdly, IsoMod defines a wider collection of access rights, thereby differentiating finer levels of visibility.

7.3 Future Work

A number of future directions are suggested by this work. Firstly, although IsoMod provides a means for enforcing a given name visibility policy, it does not prescribe what policy to enforce in a given security context. IsoMod could be extended into a full-fledged authorization system in the style of Java Authentication and Authorization Service (JAAS). Doing so would fill the aforementioned gap.

When a complex IsoMod policy is given, it can be a non-trivial matter to understand if the policy is achieving what the author intends. Therefore, another direction is to construct a formal semantic model of IsoMod. With such a semantic model, it would be possible to determine if a given safety property is enforced by an IsoMod policy.

A third direction has to do with the pragmatics of using IsoMod. This work provides an expressive policy language for defining scoping rules. With such expressiveness, it is possible for programmers to construct improper scoping rules, resulting in scoping disciplines that are incompatible with the typing disciplines of modern object-oriented programming languages. For example, how can one guarantee that a set of scoping rules honours the Principle of Subsumption [25]? Future work is necessary to address this challenge.

A fourth direction would be to incorporate name resolution control into IsoMod, in a way that is secure, expressive and declarative, so that IsoMod can be extended to support a general form of namespace management. Name resolution control could be added when a class is preloaded. Certain methods would be "tagged" as sensitive. Whenever one of these sensitive methods is encountered, the name is resolved to a more secure method which will perform additional run-time security checks.

A final direction is concerned with the automatic construction of IsoMod policies. Policy engineering from scratch can be a tedious process. A remedy is to have a tool that will scan sample code units, and infer a policy that serves as the starting point of further policy engineering.

7.4 Summary

Through the design of the IsoMod module system and its policy language, we demonstrated that name visibility control can serve as a lightweight access control mechanism that avoids the technical difficulties of interposition when full-fledged execution monitoring is not necessary. We further demonstrated that a rich family of access control policies can be encoded as name visibility constraints (aka scoping rules), the enforcement of which can be performed efficiently. Name visibility control in the style of IsoMod is therefore a viable access control mechanism for dynamically extensible systems.

Acknowledgments

This work was supported in part by an NSERC Discovery Grant.

References

[1] Martín Abadi and Cédric Fournet. Access control based on execution history. In *Proceedings of the 10th Annual Network and Distributed System Security Symposium (NDSS '03)*, San Diego, California, USA, February 2003.

[2] Davide Ancona and Elena Zucca. True modules for Java-like languages. In *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP '01)*, pages 354–380, Budapest, Hungary, June 2001.

[3] Lujo Bauer, Andrew W. Appel, and Edward W. Felten. Mechanisms for secure modular programming in Java. *Software - Practice & Experience*, 33(5):461–480, April 2003.

[4] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility, safety and performance in the SPIN operating system. In *Proceedings of the 5th ACM Symposium on Operating Systems Principles*, pages 267–284, Copper Mountain, CO, USA, December 1995.

[5] Gilad Bracha and David Ungar. Mirrors: Design principles for meta-level facilities of object-oriented programming languages. In *Proceedings of the 19th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '04)*, pages 331–344, Vancouver, BC, Canada, October 2004.

[6] David D. Clark and David R. Wilson. A comparison of commercial and military computer security policies. In *Proceedings of the 1987 IEEE Symposium on Security and Privacy (S&P '87)*, pages 184–194, May 1987.

[7] John Corwin, David F. Bacon, David Grove, and Chet Murthy. MJ: A rational module system for Java and its applications. In *Proceedings of the 18th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '03)*, pages 241–254, Anaheim, CA, USA, October 2003.

[8] Jack B. Dennis and Earl C. Van Horn. Programming semantics for multiprogrammed computations. *Communications of the ACM*, 9(3):143–155, March 1966.

[9] ECMA. *Standard ECMA-335: Common Language Infrastructure (CLI)*, 2nd edition, December 2002.

[10] David Evans and Andrew Twyman. Flexible policy-directed code safety. In *Proceedings of the 1999 IEEE Symposium on Security and Privacy (S&P '99)*, pages 32–45, Oakland, California, USA, May 1999.

[11] Philip W. L. Fong. Pluggable verification modules: An extensible protection mechanism for the JVM. In *Proceedings of the 19th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '04)*, pages 404–418, Vancouver, BC, Canada, October 2004.

[12] Philip W. L. Fong. Discretionary capability confinement. In *Proceedings of the 11th European Symposium on Research in Computer Security (ESORICS'06)*, volume 4189 of *Lecture Notes in Computer Science*, pages 127–144, Hamburg, Germany, September 2006. Springer.

[13] Philip W. L. Fong. Discretionary capability confinement. *International Journal of Information Security*, 7(2):137–154, April 2008.

[14] Philip W. L. Fong and Robert D. Cameron. Proof linking: Modular verification of mobile programs in the presence of lazy, dynamic linking. *ACM Transactions on Software Engineering and Methodology*, 9(4):379–409, October 2000.

[15] Daniel P. Friedman and Mitchell Wand. *Essentials of Programming Languages*. MIT Press, 3rd edition, 2008.

[16] Li Gong, Gary Ellison, and Mary Dageforde. *Inside Java 2 Platform Security*. Addison-Wesley, 2nd edition, 2003.

[17] Kevin W. Hamlen, Greg Morrisett, and Fred B. Schneider. Computability classes for enforcement mechanisms. *ACM Transactions on Programming Languages and Systems*, 28(1):175–205, January 2006.

[18] Norm Hardy. The confused deputy: or why capabilities might have been invented. *Operating Systems Review*, 22(4):36–38, October 1988.

[19] Robert Harper and Benjamin C. Pierce. Design considerations for ML-style module systems. In Benjamin C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 8, pages 293–346. MIT Press, 2005.

[20] Ninghui Li, Ziad Bizri, and Mahesh V. Tripunitara. On mutually-exclusive roles and separation of duty. In *Proceedings of the 11th ACM Conference on Computer and Communications Security (CCS '04)*, pages 42–51, Washington DC, USA, October 2004.

[21] Sheng Liang and Gilad Bracha. Dynamic class loading in the Java virtual machine. In *Proceedings of the 13th ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA '98)*, pages 36–44, Vancouver, BC, Canada, October 1998.

[22] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison Wesley, 2nd edition, 1999.

[23] Yu David Liu and Scott F. Smith. Modules with interfaces for dynamic linking and communication. In *Proceedings of the 18th European Conference on Object-Oriented Programming (ECOOP '04)*, Oslo, Norway, June 2004.

[24] John K. Ousterhout, Jacob Y. Levy, and Brent B. Welch. The Safe-Tcl security model. In G. Vigna, editor, *Mobile Agents and Security*, volume 1419 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.

[25] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[26] Jonathan A. Rees. A security kernel based on the lambda-calculus. A. I. Memo 1564, MIT, 1996.

[27] Nathanael Schärli, Andrew Black, and Stéphan Ducasse. Object-oriented encapsulation for dynamically typed languages. In *Proceedings of the 19th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '04)*, pages 130–149, Vancouver, BC, Canada, October 2004.

[28] Nathanael Schärli, Stéphan Ducasse, Oscar Nierstrasz, and Roel Wuyts. Composable encapsulation policies. In *Proceedings of the 18th European Conference on Object-Oriented Programming (ECOOP '04)*, Oslo, Norway, June 2004.

[29] Fred B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3(1):30–50, February 2000.

[30] Fred B. Schneider, Greg Morrisett, and Robert Harper. A language-based approach to security. In *Informatics: 10 Years Back, 10 Years Ahead*, volume 2000 of *Lecture Notes in Computer Science*, pages 86–101. Springer-Verlag, 2000.

[31] Lawrence Snyder. Formal models of capability-based protection systems. *IEEE Transactions on Computers*, 30(3):172–181, March 1981.

[32] Úlfar Erlingsson and Fred B. Schneider. IRM enforcement of Java stack inspection. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy (S&P '00)*, pages 246–255, Berkeley, CA, USA, May 2000.

[33] Dan S. Wallach, Andrew W. Appel, and Edward W. Felten. SAFKASI: A security mechanism for language-based systems. *ACM Transactions on Software Engineering and Methodology*, 9(4):341–378, October 2000.

[34] Dan S. Wallach, Dirk Balfanz, Drew Dean, and Edward W. Felten. Extensible security architecture for Java. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)*, pages 116–128, Saint Malo, France, October 1997.

[35] Ian Welch and Robert J. Stroud. Using reflection as a mechanism for enforcing security policies on compiled code. *Journal of Computer Security*, 10(4):399–432, 2002.

A An IsoMod Policy for DCC

```
policy DCC
  default allow

// ($\mathcal{DCC}1$)

method $B.N$
  denies { invoke } to method $A.M$
    unless (not $static(N)$) or $trusts(B, A)$
```

```
// ($\mathcal{DCC}2$)

class $C$
  denies { catch, cast, new } to method $A.M$
    unless trusts$(C, A)$
method $B.N$
  denies { invoke } to method $A.M$
    unless trusts(return-type$(N), A)$ or
          (trusts$(A, B)$ and trusts$(B, A)$)
field $B.F$
  denies { get } to method $A.M$
    unless trusts(field-type$(F), A)$ or
          (trusts$(A, B)$ and trusts$(B, A)$)
field $B.F$
  denies { put } to method $A.M$
    unless trusts(A, field-type$(F)$) or
          (trusts$(A, B)$ and trusts$(B, A)$)

// ($\mathcal{DCC}3$)

method $B.N$
  denies { invoke } to method $A.M$
    unless
      (trusts$(A, B)$ and trusts$(B, A)$) or
      (for $C$ in parameter-types$(N)$ :
        trusts$(C, B)$ or
        (trusts$(B, M)$ and trusts$(C, M)$))

// ($\mathcal{DCC}4$)

method $B.N$
  denies { invoke } to method $A.M$
    unless trusts$(N, M)$

// ($\mathcal{DCC}5$)

class $B$
  denies { extend, implement } to class $A$
    unless trusts$(B, A)$


// ($\mathcal{DCC}6$)

method $B.N$
```

```
    denies { override } to method $A.M$
        unless trusts$(M, N)$
method $B.N$
    denies { override } to method $A.M$
        unless trusts$($return-type$(N), B)$ or
                (trusts$(A, B)$ and trusts$(B, A))$
method $B.N$
    denies { override } to method $A.M$
        unless
            (trusts$(A, B)$ and trusts$(B, A))$ or
            (for $C$ in parameter-types$(N)$ : trusts$(C, A))$
```

// $(\mathcal{DCC}7)$

```
class $B$
    denies { extend, implement } to class $A$
        unless strongly-trusts$(B, A)$
```

// $(\mathcal{HMS}1)$

```
class $C$
    denies { extend } to class $\mathcal{E}$
        unless
            domain$(\mathcal{E})$ implies
            strongly-dominates$(\mathcal{E}$, org.aegis.dcc.Root$)$
```

// $(\mathcal{HMS}2)$

```
class $C$
    denies { extend } to class $\mathcal{E}$
        unless
            domain$(\mathcal{E})$ implies
            for $\mathcal{D}$ in strongly-dominated$(\mathcal{E})$ :
                dominates$(\mathcal{E}, \mathcal{D})$
```

// $(\mathcal{HMS}3)$

```
class $C$
    denies { extend } to class $\mathcal{E}$
        unless
            domain$(\mathcal{E})$ implies
            for $\mathcal{D}$ in strongly-dominated$(\mathcal{E})$ :
                for $\mathcal{D}'$ in dominated$(\mathcal{E})$ :
                    dominates$(\mathcal{D}, \mathcal{D}')$ or dominates$(\mathcal{D}', \mathcal{D})$
```