A Module System for Isolating Untrusted Software Extensions

Philip W. L. Fong and Simon A. Orr
Department of Computer Science, University of Regina, Regina, Saskatchewan, Canada
{pwlfong,orrsim11}@cs.uregina.ca

Abstract

With the recent advent of dynamically extensible software systems, in which software extensions may be dynamically loaded into the address space of a core application to augment its capabilities, there is a growing interest in protection mechanisms that can isolate untrusted software components from a host application. Existing language-based environments such as the JVM and the CLI achieves software isolation by an interposition mechanism known as stack inspection. Expressive as it is, stack inspection is known to lack declarative characterization and is brittle in the face of evolving software configurations.

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 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. In short,* ISOMOD *avoids the technical difficulties of interposition by trading off an acceptable level of expressiveness. Name visibility control in the style of* ISOMOD *is therefore a lightweight alternative to interposition.*

1. Introduction

Secure cooperation is the problem of protecting mutually suspicious code units from one another while they are executing in the same run-time environment [22, 27]. It finds applications in dynamically extensible software systems such as mobile code platforms, scriptable applications, and software systems with plug-in architectures. The language-based approach to security [26] is an established paradigm for addressing the challenge of secure cooperation. 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. This paper proposes a language-based access control mechanism that is based solely on *name visibility control*.

Existing language-based approaches to access control are mostly based on the classical notion of *interposition* [30, 29, 32, 1]. A direct implementation of this idea is to interpose monitoring code at the entry points of system services. At run time, authorization decisions are made by examining invocation arguments or execution history. For example, stack inspection [30] as found in the Java Virtual Machine (JVM) [19] and the Common Language Infrastructure (CLI) [9] is implemented in this way. 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, 29, 30, 32] Specifically, monitoring code is *weaved* into untrusted code at load time. Although this so called inlined reference monitor (IRM) approach [29] is equal in expressive power to direct interposition [14], 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 [29, 30].

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 [25], 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 by the large number of target-less `BasicPermission`s defined in the Java 2 security architecture.) In other cases, the safety property [25] to be enforced is memory-less, and the avoidance of the confused deputy problem [15] 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.

In this paper, we explore a seldom studied point in the design space of language-based access control — *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. 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, we present the design of a practical security architecture for dynamically extensible Java applications that is built around a module system called ISOMOD (Sect. 2). In programming language literature, a *module system* is a facility responsible for managing the visibility of names across namespaces [16]. 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 interesting 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 (Sect. 3). Not only can the ISOMOD policy language simulate a sizable subset of permissions in the Java 2 security architecture (Sect. 4.1), it can do so with policies that are robust to changes in software configurations (Sect. 4.2). The ISOMOD policy language is also expressive enough to completely encode a capability type system known as Discretionary Capability Confinement [12] (Sect. 4). In spite of its expressiveness, the ISOMOD policy language admits an efficient implementation strategy (Sect. 5). 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.

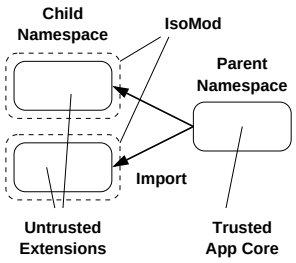

Figure 1. Hierarchical namespaces & ISOMOD

2. The ISOMOD Security Architecture

ISOMOD employs *name visibility control* as the sole mechanism for access control. We describe the Java class loading mechanism from the perspective of name visibility control. In programming language terminology, a Java class loader is the *mirror* [5] of a run-time namespace. Hierarchical organization of namespaces is enabled by the delegation model of class loading [18], in which the names visible in a parent namespace are imported implicitly into a child namespace (Fig. 1). Consequently, the set of names visible in a namespace is the union of (1) the set of names visible in its delegation parent and (2) the set of names that are defined locally. A class may refer to external entities such as other classes and their fields and methods. These external references are resolved in the same namespace in which the referring class is defined. In short, static scoping is enforced.

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 (Fig. 1). Core application services are exposed to the extension code through implicit name import 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 extension namespaces. Specifically, an ISOMOD namespace enforces 2 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 namespace is endowed with a custom *name visibility policy*, which specifies

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

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 (Sect. 4.1 and 4.2), or impose collaboration protocols among classes defined in the extension namespace (Sect. 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 type. 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* of visibility control distinguishes ISOMOD from traditional module systems, in which visibility control is enforced only at compile time. It is this feature that turns 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 namespace. An *access* is composed of three elements: (1) a *subject*, (2) an *object*, and (3) an *access right*.

Unary Predicates on Declared Types		
final	abstract	interface
public	package-private	
Binary Predicates on Declared Types		
subclass	superinterface	assignable
extends	implements	

Figure 3. Sample predicates

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 Fig. 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]

In general, a policy clause describes if a target O grants (denies) access rights $r_1, \ldots, r_k$ to a subject S. When the optional *to*-phrase is omitted, the rights are granted (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. Fig. 3 shows a sample of built-in predicates.

Prior to the definition of a declared type [19, Sect. 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 Fig. 4: 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.) If any access is denied by the policy, the definition of the declared type will not be authorized.

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.

```
To decide if access ⟨S, O, r⟩ is granted by policy P:
  for each rule R in policy P do
    if R is relevant to ⟨S, O, r⟩ 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 a allow-rule then
              return grant;
            else // R is a deny-rule
              return deny;
  return user-specified default;
```

Figure 4. Authorization semantics

4. Sample Applications

4.1. Selective Hiding of System Services

ISOMOD can be used to enforce many of the `BasicPermissions` defined in the Java 2 platform. 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 return 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]).

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

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 for creating `ClassLoader` instances.

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

```
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 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.

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. 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, 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 reaudit of the software infrastructure. Such a practice is simply unacceptable.

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 by which a reference of 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

We demonstrate how ISOMOD can be used for enforcing a general-purpose capability type system, *Discretionary Capability Confinement (DCC)* [12]. 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, the latter specifying in what ways the underlying object can

[2]For brevity, the analysis does not account for array types. 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 5. DCC Type Constraints

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 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) could be seen as a capability, because the typed reference only exposes a restricted view of the underlying object. This approach of modeling capabilities suffers from two problems: capability leakage and capability theft [28]. DCC is a minimal perturbation to Java for controlling capability propagation. In the following, we illustrate the expressiveness of ISOMOD by employing its policy language to encode the DCC type system. The focus here is ISOMOD and not DCC. Interested readers may consult [12] for more details of DCC.

In DCC, the space of declared types (e.g., class & 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 belongs 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 $l(m)$, 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.

Fig. 5 enumerates the type constraints of DCC as specified in [12]. We have successfully encoded all the DCC type constraints by an ISOMOD policy, which is displayed in Appendix A. Behind the policy of Appendix A is a number of assumptions. As in [12], 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 highlight some aspects of this encoding that illustrate further features of ISOMOD.

Consider the following type constraint from Fig. 5:

($\mathcal{DCC}2$) (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**

$$(\textit{trusts}(A, B) \textbf{ and } \textit{trusts}(B, A))$$

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 a second type constraint from Fig. 5:

$(\mathcal{HMS}3)$ 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* [12], which enforces a strong form of separation of duty [6, 17], 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 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 verify $(\mathcal{HMS}3)$ 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 [19, Sect. 5.4.4] (i.e., public, protected, private, etc) as an ISOMOD policy. These exercises 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 our implementation experience.

Challenges. We begin by describing the technical challenges our implementation strategy attempts to address:

Stage-I Preloading of Declared Type C
1. Retrieve the classfile of C.
2. Perform stage-I preloading on the supertypes of C. Circular preloading 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 the declaring classes of any fields or methods that are the targets of some remaining external accesses associated with C.
4. Check the remaining external accesses of C.
5. Authorize the definition of declared type C.

Figure 6. Preloading Algorithm

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 [19, Sect. 5.3.5] is irrevocable. Policy enforcement must be complete before a classfile is converted into a `Class`.
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.

Three-Staged, Lightweight Preloading. We adopted a design that adequately addresses the above technical challenges. Specifically, before a declared type is defined, its corresponding classfile is *preloaded* and screened for policy violation. Preloading is divided into three stages (Fig. 6). In the first stage, all "extend" and "implement" accesses are checked, and a lightweight representation of the preloaded classfile's type interface is constructed. In stage two, all "override" accesses are checked, and external type references appearing in the type interface are resolved. In the third stage, all remaining external accesses are checked. This design performs shallow preloading eagerly, but maintains lightweight type mirrors to anticipate deep preloading.

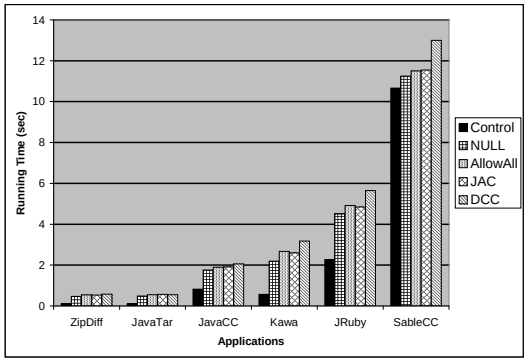

Figure 7. Overhead of ISOMOD

The lazy preloading strategy breaks circularity, and correctly attributes policy violations to the offending classfiles. The preloading algorithm is detailed in Fig. 6. This design is informed by previous work on modular bytecode verification in the presence of lazy, dynamic linking [13, 11].

Performance Evaluation. We conducted experiments to profile the performance of our implementation strategy. We measured the running time of 6 open source Java applications under 5 configurations on a stock PC (P4 3GHz). The first configuration, *Control*, is to run an application in a bare JVM. In the other four configurations, the applications are loaded by an ISOMOD class loader. These configurations differ in the ISOMOD policy being imposed. The *NULL* configuration imposes an empty policy that allows all accesses. The *AllowAll* configuration imposes a policy with policy clauses that match and allow every access. The *JAC* and *DCC* configurations impose the ISOMOD encoding of Java's access control mechanism and DCC respectively (see Sect. 4.3). Five trials were repeated for each configuration to account for variability, and the average running time (in seconds) is reported in Fig. 7. A number of observations can be made of Fig. 7. Firstly, the overhead of ISOMOD for a given application is the difference in running time between an ISOMOD configuration and the *Control* configuration. Notice that this overhead never exceeds 3.5 seconds in our experiments. This confirms that the technology is feasible for practical applications. Secondly, the difference in running time between the *NULL* and *Control* configurations roughly provides an estimate of the overhead contributed by the maintenance of type mirrors. The rest of the overhead can be attributed to policy evaluation. Thirdly, the overhead contributed by ISOMOD does not grow with the total running time of an application. This can be explained by noticing that ISOMOD only incurs overhead at the time of application start-up. For a long-running application (e.g., SableCC), the overhead will be amortized away.

6. Concluding Remarks

Limitations Enforcement mechanisms that are based solely on static analysis, of which ISOMOD is an example, are provably less powerful than those that employ execution monitoring [14]. For example, policies in which authorization decision is a function of invocation arguments or execution history are not enforceable by ISOMOD. Our goal 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.

Related Work As surveyed in Sect. 1, language-based software isolation has been achieved mostly by interposition-based mechanisms in the past. Early language-based systems such as Scheme 48 [22], Safe-Tcl [21] 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 [31]. Secondly, rudimentary name visibility control is employed to hide certain names in a namespace. None of the policy clauses in Sect. 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 [16]. 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 [20] 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* [24, 23]. 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 Sect. 4 cannot be expressed. Thirdly, ISOMOD defines a wider collection of access rights, thereby differentiating finer levels of visibility.

Future Work We are interested in extending ISOMOD into a full-fledged authorization system in the style of JAAS. Another direction is to study ISOMOD policy refinement and composition. We are also interested in employing ISOMOD to enforce communication integrity.

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

References

[1] M. Abadi and C. Fournet. Access control based on execution history. In *Proceedings of the 10th Annual Network and Distributed System Security Symposium*, San Diego, CA, USA, Feb. 2003.

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

[3] L. Bauer, A. W. Appel, and E. W. Felten. Mechanisms for secure modular programming in Java. *Software - Practice & Experience*, 33(5):461–480, Apr. 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, Dec. 1995.

[5] G. Bracha and D. 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*, pages 331–344, Vancouver, BC, Canada, Oct. 2004.

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

[7] J. Corwin, D. F. Bacon, D. Grove, and C. 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*, pages 241–254, Anaheim, CA, USA, Oct. 2003.

[8] J. B. Dennis and E. C. V. Horn. Programming semantics for multiprogrammed computations. *Communications of the ACM*, 9(3):143–155, Mar. 1966.

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

[10] D. Evans and A. Twyman. Flexible policy-directed code safety. In *Proceedings of the 1999 IEEE Symposium on Security and Privacy*, pages 32–45, Oakland, CA, USA, May 1999.

[11] P. 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*, pages 404–418, Vancouver, BC, Canada, Oct. 2004.

[12] P. W. L. Fong. Discretionary capability confinement. In *Proceedings of the 11th European Symposium on Research in Computer Security*, Hamburg, Germany, Sept. 2006.

[13] P. W. L. Fong and R. 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, Oct. 2000.

[14] K. W. Hamlen, G. Morrisett, and F. B. Schneider. Computability classes for enforcement mechanisms. *ACM Transactions on Programming Languages and Systems*, 28(1):175–205, Jan. 2006.

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

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

[17] N. Li, Z. Bizri, and M. V. Tripunitara. On mutually-exclusive roles and separation of duty. In *Proceedings of the 11th ACM Conference on Computer and Communications Security*, pages 42–51, Washington DC, USA, Oct. 2004.

[18] S. Liang and G. Bracha. Dynamic class loading in the Java virtual machine. In *Proceedings of the 13th ACM Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 36–44, Vancouver, BC, Canada, Oct. 1998.

[19] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison Wesley, 2nd edition, 1999.

[20] Y. D. Liu and S. F. Smith. Modules with interfaces for dynamic linking and communication. In *Proceedings of the 18th European Conference on Object-Oriented Programming*, Oslo, Norway, June 2004.

[21] J. K. Ousterhout, J. Y. Levy, and B. 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.

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

[23] N. Schärli, A. Black, and S. Ducasse. Object-oriented encapsulation for dynamically typed languages. In *Proceedings of the 19th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 130–149, Vancouver, BC, Canada, Oct. 2004.

[24] N. Schärli, S. Ducasse, O. Nierstrasz, and R. Wuyts. Composable encapsulation policies. In *Proceedings of the 18th European Conference on Object-Oriented Programming*, Oslo, Norway, June 2004.

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

[26] F. B. Schneider, G. Morrisett, and R. Harper. A language-based approach to security. In *Informatics: 10 Years Back, 10 Years Ahead*, volume 2000 of *LNCS*, pages 86–101. Springer, 2000.

[27] M. D. Schroeder. *Cooperation of Mutually Suspicious Subsystems in a Computer Utility*. Ph.D. thesis, MIT, 1972.

[28] L. Snyder. Formal models of capability-based protection systems. *IEEE Transactions on Computers*, 30(3):172–181, Mar. 1981.
[29] Úlfar Erlingsson and F. B. Schneider. IRM enforcement of Java stack inspection. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy*, pages 246–255, Berkeley, CA, USA, May 2000.
[30] D. S. Wallach, A. W. Appel, and E. W. Felten. SAFKASI: A security mechanism for language-based systems. *ACM Transactions on Software Engineering and Methodology*, 9(4):341–378, Oct. 2000.
[31] D. S. Wallach, D. Balfanz, D. Dean, and E. W. Felten. Extensible security architecture for Java. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 116–128, Saint Malo, France, Oct. 1997.
[32] I. Welch and R. 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*$(\mathcal{D}', \mathcal{D})$