Capabilities as Alias Control: Secure Cooperation in Dynamically Extensible Systems

Philip W. L. Fong Cheng Zhang

Technical Report CS-2004-3
April 14, 2004

Department of Computer Science
University of Regina
Regina, Saskatchewan, S4S 0A2
Canada

ISBN 0-7731-0479-8
ISSN 0828-3494

Capabilities as Alias Control: Secure Cooperation in Dynamically Extensible Systems

Philip W. L. Fong Cheng Zhang
Department of Computer Science
University of Regina
Regina, Saskatchewan, Canada S4S 0A2
{pwlfong, zhang20c}@cs.uregina.ca

April 14, 2004

Abstract

Secure cooperation is the problem of protecting mutually suspicious code units within the same execution environment from their potentially malicious peers. A statically enforceable capability type system is proposed for the JVM bytecode language to provide fine-grained access control of shared resources among peer code units. The design of the type system is inspired by recent advances in alias control type systems for object-oriented programming languages. The exercise of access rights and the propagation of capabilities are given a uniform interpretation as alias creation events. Each capability type assigns to a reference a dataflow trajectory, prescribing the set of aliases that is allowed to be created from the reference. An orthogonal and complementary type system for controlling object creation and downcasting is also designed to avoid a class of capability spoofing attacks. The combined type system successfully addresses a number of classical protection problems recast in a programming language context. This work therefore demonstrates the need and the feasibility of a language-based approach to enforce application-level security among peer code units.

1 Introduction

Secure cooperation [35] is the problem of protecting mutually suspicious code units within the same execution environment from their potentially malicious peers. Genuine cooperation is predicated on the establishment of trust between collaborating code units, so that access to shared resources can be precisely controlled. Secure cooperation is therefore an enabling infrastructure for dynamically extensible software systems such as mobile code language environments [7, 40], scriptable applications, and software systems with plug-in architectures [5, 13, 33].

1.1 Limitations of Existing Language-Based Approaches to Secure Cooperation

The language-based approach has become a leading security paradigm in the development of dynamically extensible software systems. Strongly typed language environments supporting dynamic loading of code units, such as the JVM [29] and the CLR [11], are prototypical platforms for hosting dynamically extensible applications. Language-based protection mechanisms such as stack inspection [44, 20], SASI/IRM [41, 42], Model-Carrying Code [38], and history-based access control [12, 1, 17] take the perspective of a software system protecting its resources and privileged services against untrusted software extensions. Essential as it is for infrastructure protection, such a bipartite perspective does not address the need of protection for peer code units that are suspicious of each other.

A bipartite, or more generally, a hierarchical perspective of secure cooperation sees the underlying software system as a collection of application layers. The emphasis is on the protection of an application layer from being abused by an adjacent client layer. In the context of extensible systems, such a perspective protects an application core from untrusted software extensions. Yet, the protection interest of an extensible system developer may go further than what a hierarchical perspective can offer. For instance, the developer may wish to impose specific communication protocols among collaborating software extensions. Likewise, the developer may need to promote structure and resource sharing between mutually suspicious software extensions by assuring them that abuse will not occur. Existing literature on language-based protection is relatively silent on this need of peer-to-peer security.

1.2 A New Approach: Capabilities as Alias Control

In this work, a novel capability type system is proposed for the JVM bytecode language. The design goal is to provide a fine-grained access control mechanism for capturing application-level security. Specifically, every object reference can be protected by a capability type, thereby allowing peer code units to precisely control the way shared structures are accessed. The caller of a method may control the way arguments are to be accessed by the callee, and, likewise, the callee may control the way the return value is to be accessed by the caller. When coupled with subtyping rules, an application core may impose communication patterns on collaborating concrete classes through the definition of abstract classes (or interfaces) in terms of capability types.

Inspired by recent advances in alias control type systems for object-oriented programming languages [24, 31, 3, 34, 8, 43, 6, 2], our type system offers a fresh interpretation of the notion of capability, which is traditionally understood as a reference plus a set of access rights [10, 9]. In a language-based environment, method invocation and field setting inevitably create aliases. Controlling alias creation therefore provides an effective means for restricting access to class members. Such an insight allows us to reinterpret a capability as a reference plus a dataflow trajectory, prescribing the set of aliases that is allowed to be created from the reference. This reinterpretation produces an extremely fine-grained access control mechanism for language-based systems. An orthogonal and complementary type system for controlling object creation and downcasting is also designed to avoid a class of capability spoofing attacks. The combined type system successfully

```
interface Prisoner {
  void send(Prisoner p, Guard g);
  void receive(Mail m);
}

interface Guard {
  void deliver(Mail m, Prisoner p);
}

public final class Mail {
  public Mail(string m) { msg = m; }
  public string read() { return msg; }
  private string msg;
}
```

Figure 1: The Prisoner Mail System Problem

addresses a number of classical protection problems [9] recast in a programming language context.

2 The Prisoner Mail System Problem

We motivate the discussion of our capability type system by examining a toy problem originally proposed in an early work of Ambler and Hoch [4] for studying protection in programming languages. This so called Prison Mail System Problem is simplified and recast here in an object-oriented flavor without diluting the essense of its original challenges. The protection challenges presented by the revised toy problem are then categorized according to a scheme inspired by the seminal paper of Cohen and Jefferson [9].

In the Prisoner Mail System are three types of objects — `Prisoner`s, `Guard`s and `Mail`s (Figure 1).

1. Instances of the `Prisoner` interface are forbidden from direct communication with each other. All message exchanges must be mediated by the Prison Mail System.

2. Instances of the `Guard` interface are responsible for delivering messages.

3. Instances of the `Mail` class are message carriers.

Classes implementing the `Prisoner` and `Guard` interfaces are dynamically loaded software extensions, and their integrity are not to be trusted. While the `Guard`s are ever suspicious of conspiracies, the `Prisoner`s resent any form of censorship. To their mutual agreement, the Prison Mail System application core imposes the following mail delivery protocol: The application core

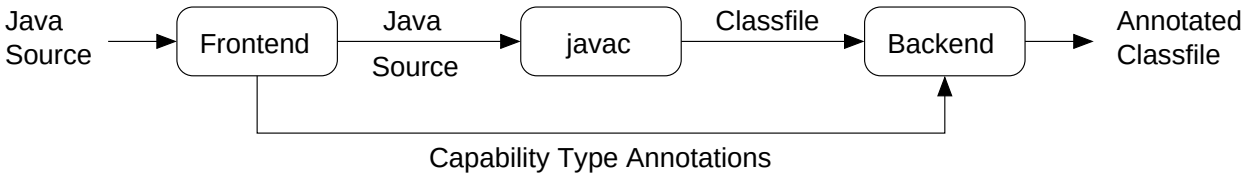

Figure 2: Processing of Type Annotations

randomizes the schedule of mail delivery and the assignment of `Guard` responsible for delivery. It schedules a delivery by invoking the `send` method of a `Prisoner` object, specifying which fellow `Prisoner` the sender is allowed to correspond with, and which `Guard` is to be responsible for the delivery. The `Prisoner.send` method will then create a `Mail` object, and pass it along with the addressee to the `deliver` method of the assigned `Guard`. `Guard.deliver` will in turn invoke the `receive` method of the receiving `Prisoner`, passing the `Mail` object as the argument. This completes one mail delivery.

The following security constraints must be enforced.

1. **Safe Invocation.** `Prisoner`s want to be assured that `Mail` messages passed to the `deliver` method are not read by the mediating `Guard` objects.

2. **Capability Amplification.** When a `Mail` is delivered to a receiving `Prisoner`, the previous restriction on read access should be lifted so that the embedded message can be consumed.

3. **Limiting Propagation of Capability.** `Mail` objects that are in transit must not be leaked to any party other than a `Prisoner`.

4. **Mediated Communication.** The sending `Prisoner` may not contact the receiving `Prisoner` directly. All communications must be mediated by the assigned `Guard`.

5. **Flexible Control of Capability Storing.** The sending `Prisoner` may not store away the addressee reference for future use. The receiving `Prisoner`, however, may save the `Mail` for future reading.

As we shall see in Section 4, all these protection problems are fully addressed by our capability type system.

3 A Capability Type System

This section provides a high-level overview of our capability type system. The type system is intended to be used for annotating Java source files. Type annotations will be extracted by the compiler frontend, and subsequently injected into the classfiles generated by the compiler (Figure 2). Type checking, inter- and intra-modular, will be conducted by the JVM at link time, against classfiles, at the bytecode level. This is to ensure dynamic linking is type-safe with respect to our

capability type system. The following description therefore focuses on typing Java classfiles at the bytecode instruction level. Syntactic issues such as the concrete syntax for annotating Java source programs with capability types, or the encoding scheme for embedding type annotations in classfiles, do not concern us in this paper.

3.1 An Intuitive Description

We begin the discussion of our capability type system with an informal account that highlights the intuitions behind the technicalities that follow.

Capability Types. A reference to an instance of class A offers indiscriminate access to the public interface of the object. When an object reference is passed from one context to another (i.e., argument passing or method value returning), the owner of the reference may want to selectively disable certain operations from being applied to the object in the receiver context (e.g., the sending `Prisoner` desires that the `deliver` method does not invoke the `read` method on the `Mail` argument). Essentially, the owner may want to present an alternative *view* of class A that is more restrictive than its public interface. More than that, depending on the context, different views may need to be presented for different receivers (e.g., mediating `Guard` vs receiving `Prisoner`). Such a need is traditionally filled by the use of capabilities. A capability is a reference plus a set of access rights, prescribing what operations can be performed on the underlying reference. In the context of a language-based environment, one may model capabilities with a type system by assigning to each object reference a type that prescribes access rights. Well-typed programs are those that only exercise rights permitted by the typing discipline. Our capability type system is an instance of this general approach.

The Priority of Method Invocation. In a language-based extensible system, the prime liability is code execution. Undisciplined execution of untrusted code is to be avoided. In the context of Java, this boils down to controlling method invocation. The chief goal of our capability type system is therefore the regulation of method invocation pattern.

Capabilities for Argument Passing. The core insight behind our type system is that method invocation coincides with alias creation events. Specifically, when a method is applied to its arguments, aliases of the arguments are created through the binding of actual arguments to formal parameters. Controlling the creation of aliases caused by argument binding effectively restricts method invocation. A capability can therefore be interpreted as a reference plus a set of alias control constraints, prescribing the argument binding events that may occur to the underlying reference. A capability type is thus a compact specification of such constraints.

Controlling Capability Propagation. In our type system, a capability type constrains not only a single aliasing event, but rather it specifies the set of all future aliasing events that may occur to a reference. Specifically, every capability type corresponds to some finite automaton, specifying the set of all call chains the reference may traverse. This effectively outlines a dataflow trajectory for the object reference, and provides precise control on the way a capability may be propagated.

Controlling Capability Sharing. A capability type may also specify if the underlying reference can be stored into fields, which again coincides with an alias creation event. Such a feature may be used to control if a capability can be shared after it is passed as an argument to a method. This in turns constrains structure sharing.

$$
\begin{aligned}
A, B &\in \textbf{JavaReferenceTypes} \\
M &\in \textbf{JavaMethodTypes} \\
F &\in \textbf{JavaFieldTypes} \\
m^{A,M} &\in \textbf{JavaMethodSignatures}^{A,M} \\
f^{A,F} &\in \textbf{JavaFieldSignatures}^{A,F}
\end{aligned}
$$

$$
\begin{aligned}
M &::= (A_1, \ldots, A_k)B \\
&\quad\mid (A_1, \ldots, A_k)\textbf{void} \\
F &::= A
\end{aligned}
$$

Figure 3: Abstract Syntax for Java Types

3.2 Assumptions and Notations

Figure 3 defines syntactic categories related to the standard Java type system. They will be assumed in the following discussion. In this work, a Java reference type (A, B) is either a class or interface type in Java (e.g., `Mail`, `Prisoner`). We assume in the following that the types of fields, formal parameters and method return values are all reference types; primitive (e.g., `int`) and array types (e.g., `Mail[]`) are ignored for notational economy. Similarly, although our scheme applies equally well to static methods and fields, we consider only instance methods and fields in this paper. A Java method type (M) is a list of parameter types, not including that of the implicit formal parameter `this`, plus a return type (e.g., (`Prisoner`, `Guard`)**void**). A Java field type (F) is simply a Java reference type (e.g., `string`). A Java method signature $m^{A,M}$ with Java method type M is defined for Java reference type A if a method with that signature is declared in A or one of its supertypes. For example, "`void send(Prisoner, Guard)`" is a method signature defined for Java reference type `Prisoner`. Its Java method type is (`Prisoner`, `Guard`)**void**. A Java field signature $f^{A,F}$ with Java field type F is defined for a Java reference type A if a field with this signature is declared in A or one of its supertypes. For example, "`string msg`" is a field signature defined for the Java reference type `Mail`, and has a Java field type `string`.

3.3 Capability Types

A capability type $\mathcal{T}^A$ defines a set of sequences of aliasing events that may occur to an underlying Java reference type A. The abstract syntax of capability types is given in Figure 4, the exposition of which is given below. A capability type is constructed from *primitive capabilities* and *capbility type constructors*.

$$\begin{aligned} \mathcal{C}^A &\in \textbf{PrimitiveCapabilities}^A \\ \mathcal{T}^A, \mathcal{U}^A, \mathcal{V}^A &\in \textbf{CapabilityTypes}^A \\ \mathcal{X}^A &\in \textbf{CapabilityTypeVariables}^A \end{aligned}$$

$$\mathcal{T}^A ::= \top^A \mid \bot^A \mid \mathcal{C}^B \rightarrow \mathcal{T}^A \mid [\mathcal{T}^A] \mid \mathcal{T}_1^A \sqcap \mathcal{T}_2^A \mid \mathcal{X}^A$$

Figure 4: Abstract Syntax for Capability Types

3.3.1 Primitive Capabilities

A primitive capability $\mathcal{C}^A$ for a Java reference type A is a *named* subset of method signatures defined for Java reference type A. A primitive capability $\mathcal{C}^A$ specifies the signatures $m^{A,M}$ of a set of methods that can be applied to an object reference (possibly through virtual method invocation). In short, it denotes a set of argument binding events. Primitive capabilities are *not* capability types.

Example. The following set named READ defines a primitive capability for the `Mail` class.

$$\text{READ} = \{\texttt{string read()}\}$$

It represents the event of passing an object reference as an argument to the `read` method declared in `Mail`. Since the method has only `this` as its formal parameter, the primitive capability represents the right to invoke `read` on a `Mail` reference.

3.3.2 Capability Type Constructors

A capability type can be built by recursively applying the following type constructor.

Top. *Top* ($\top$) is the *most* restrictive capability type. No aliasing of the underlying object reference is permitted.

Bottom. *Bottom* ($\bot$) is the *least* restrictive capability type. Arbitrary aliasing is permitted of the underlying object reference.

Propagation. If $\mathcal{C}^B$ is a primitive capability for some Java reference type B, and $\mathcal{T}^A$ is a capability type for Java reference type A, then the *propagation type* $\mathcal{C}^B \rightarrow \mathcal{T}^A$ is a capability type for Java reference type A. Intuitively, one may read $\mathcal{C}^B \rightarrow \mathcal{T}^A$ as "*grant* $\mathcal{T}^A$ *to* $\mathcal{C}^B$", meaning that the underlying object reference can be passed as an argument to methods with signatures belonging to primitive capability $\mathcal{C}^B$. Moreover, the result of binding the argument reference to the corresponding formal parameter is that the reference will acquire capability type $\mathcal{T}^A$ inside the body of the invoked method. Note that the propagation type also applies to the binding of a reference to the formal parameter `this` of an instance method.

Example. A `Mail` reference with capability type

$$\text{READ} \rightarrow \bot$$

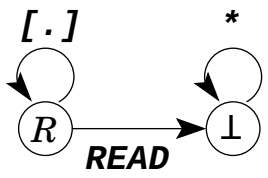

Figure 5: LTS for capability type $\mathcal{R}$

can be passed to the formal parameter `this` of the `read` method. In short, the `Mail` reference is `read`able. Inside the `read` method, the `this` parameter can be accessed without restriction ($\bot$).

Sharing. If $\mathcal{T}^A$ is a capability type for Java reference type A, then the *sharing type* $[\mathcal{T}^A]$ is also a capability type for Java reference type A. Intuitively, an object reference with capability type $[\mathcal{T}^A]$ can be stored into a field of Java reference type A. Moreover, the stored reference will acquire the capability type $\mathcal{T}^A$.

Example. A `Mail` reference with the capability type

$$[\text{READ} \rightarrow \bot]$$

can be saved into a field and subsequently retrieved for reading.

Choice. If $\mathcal{T}_1^A$ and $\mathcal{T}_2^A$ are capability types for Java reference type A, then the *choice type* $\mathcal{T}_1^A \sqcap \mathcal{T}_2^A$ is also a capability type for Java reference type A. Intuitively, the resulting capability denotes the right to exercise either $\mathcal{T}_1^A$ or $\mathcal{T}_2^A$, but not both. The $\sqcap$ operator is commutative, associative, and idempotent.

Example. A `Mail` reference with the capability type

$$(\text{READ} \rightarrow \bot) \sqcap [\text{READ} \rightarrow \bot]$$

can either be read rightaway, or be saved into a field for future reading.

Abstraction. As we shall see below, recusive definition of capability types is supported. This feature requires the use of type variables to name capability types. An occurrence of a type variable $\mathcal{X}^A$ names the capability type that defines the type variable. With recursive definitions, every capability type in fact specifies a *labelled transition system (LTS)* [14], each transition of which is labelled by either a primitive capability $\mathcal{C}^B$ or a sharing type constructor $[\,\cdot\,]$.

Example. A `Mail` reference with a capability type satisfying the following recursive definition

$$\mathcal{R} = (\text{READ} \rightarrow \bot) \sqcap [\mathcal{R}]$$

can be read rightaway, or be saved into a field for both future reading and further sharing. The LTS for this capability type is shown in Figure 5.

3.3.3 Subtyping

Subtyping permits the binding of more capable object references to variable names with less capable types. Formally, given capability types $\mathcal{T}_1^{A_1}$ and $\mathcal{T}_2^{A_2}$, $\mathcal{T}_1^{A_1}$ is a subtype of $\mathcal{T}_2^{A_2}$, denoted $\mathcal{T}_1^{A_1} <: \mathcal{T}_2^{A_2}$, if (1) the Java reference type A_1 is a subtype of the Java reference type A_2, and (2) there is a homomorphism from the LTS represented by $\mathcal{T}_2^{A_2}$ to the LTS represented by $\mathcal{T}_1^{A_1}$. As usual, the subtyping relation ($<:$) is reflexive and transitive.

$$\mathcal{M}^{A,M} \in \textbf{MethodAnnotations}^{A,M}$$
$$\mathcal{F}^{F} \in \textbf{FieldAnnotations}^{F}$$

$$\begin{array}{rcll}
\mathcal{M}^{A_0,M} & ::= & \mathcal{T}_0^{A_0}(\mathcal{T}_1^{A_1},\ldots,\mathcal{T}_k^{A_k})\mathcal{T}^B & \text{if } M = (A_1,\ldots,A_k)B \\
& | & \mathcal{T}_0^{A_0}(\mathcal{T}_1^{A_1},\ldots,\mathcal{T}_k^{A_k})\textbf{void} & \text{if } M = (A_1,\ldots,A_k)\textbf{void} \\
\mathcal{F}^{F} & ::= & \mathcal{T}^{A} & \text{if } F = A
\end{array}$$

Figure 6: Abstract Syntax for Capability Types of Reference Type Members

3.4 Typing Members of Reference Types

The members (i.e., fields or methods) of a Java reference type can be typed in our capability type system. The abstract syntax for capability types for reference type members is given in Figure 6. The capability annotation for a method defined for Java reference type A_0 with Java method type $M = (A_1,\ldots,A_k)B$ is of the form $\mathcal{T}_0^{A_0}(\mathcal{T}_1^{A_1},\ldots,\mathcal{T}_k^{A_k})\mathcal{T}^B$, where $\mathcal{T}_0^{A_0}$ is the capability type of `this`, $\mathcal{T}_i^{A_i}$ the capability type of formal parameter i, and $\mathcal{T}^B$ the capability type of the return value. The capability annotation for a field of Java reference type $F = A$ is simply a capabilitiy type $\mathcal{T}^A$. Subtyping of method types follows the usual contravariant rule (i.e., $\mathcal{U}_0^{A_0}(\mathcal{U}_1^{A_1},\ldots,\mathcal{U}_k^{A_k})\mathcal{U}^A <: \mathcal{V}_0^{B_0}(\mathcal{V}_1^{B_1},\ldots,\mathcal{V}_k^{B_k})\mathcal{V}^B$ if $\mathcal{V}_i^{B_i} <: \mathcal{U}_i^{A_i}$ and $\mathcal{U}^A <: \mathcal{V}^B$).

3.5 Capability Type Interfaces

Every Java class (or interface) is endowed with a capability type interface, the abstract syntax of which is provided in Figures 7. A capability type interface $\mathcal{S}^A$ for a class (or interface) A is composed of three sections, namely, (1) *primitive capability definitions*, (2) *capability type variable definitions*, and (3) *capability type annotations*.

Primitive capability definitions. Every primitive capability definition associates a set of method signatures defined for Java reference type A to a primitive capability $\mathcal{C}^A$. Primitive capabilities defined for the supertypes of A are implicitly inherited by A.

Capability type variable definitions. Each definition binds a capability type $\mathcal{T}^A$ to a type variable $\mathcal{X}^A$. (Mutually) recusive definition of capability type variables is supported, so long as the recursive definition is properly guarded [14]. Variables defined in the capability type interfaces of supertypes are implicitly inherited.

Capability type annotations. A capability type annotation for a field $f^{A,F}$ assigns a capability type $\mathcal{T}^F$ to the field. The underlying Java type of the field must match the underlying reference type of the capability type. A capability type annotation for method $m^{A.M}$ assigns a capability type to every formal parameter and also the return value (in the case of non-void method). Again, the

$$
\begin{array}{rcl}
\mathcal{P}^A & \in & \textbf{PrimitiveCapabilityDefinitions}^A \\
\mathcal{Q}^A & \in & \textbf{CapabilityTypeVariableDefinitions}^A \\
\mathcal{R}^A & \in & \textbf{CapabilityTypeAnnotations}^A \\
\mathcal{S}^A & \in & \textbf{CapabilityTypeInterfaces}^A
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{S}^A & ::= & \textbf{class}\ A\ \{\ \mathcal{P}_1^A \ldots \mathcal{P}_m^A\ \mathcal{Q}_1^A \ldots \mathcal{Q}_n^A\ \mathcal{R}_1^A \ldots \mathcal{R}_k^A\ \} \\
\mathcal{P}^A & ::= & \textbf{capability}\ \mathcal{C}^A = \{m_1^{A,M_1}, \ldots, m_k^{A,M_k}\} \\
\mathcal{Q}^A & ::= & \textbf{define}\ \mathcal{X}^{\mathcal{A}} = \mathcal{T}^{\mathcal{A}} \\
\mathcal{R}^A & ::= & \textbf{method}\ m^{A,M} : \mathcal{M}^{A,M} \\
 & \mid & \textbf{field}\ f^{A,F} : \mathcal{F}^F
\end{array}
$$

Figure 7: Abstract Syntax for Capability Type Interfaces

underlying Java type of the annotation must match the Java type of the method.

Method overriding must follow the usual subtyping requirement. That is, if the annotations of the overriding method and the overridden method are $\mathcal{M}_1^{A_1,M}$ and $\mathcal{M}_2^{A_2,M}$ respectively, then $\mathcal{M}_1^{A_1,M} <: \mathcal{M}_2^{A_2,M}$.

Methods that are not annotated explicitly inherit their annotations from supertypes, or else, if no such annotation is available, then a default annotation is assumed. Specifically, the default annotation for a field is simply $\perp$, where the default annotation for a method assigns $\perp$ uniformly to all formal parameters and the return value.

3.6 Type Checking

Type rules must be in place for checking if the implementation of a Java reference type conforms to a given capability type interface. Our type system controls the creation of aliases caused by passing arguments and setting fields. Accordingly, type rules should be in place for bytecode instructions *putfield*, *getfield*, *putstatic*, *getstatic*, *invokevirtual*, *invokespecial*, *invokeinterface* and *invokestatic*. We give an informal account of the type rules for *putfield* and *invokevirtual*. The rest are analogous[1].

putfield $f^{A,F}$

Operand Stack:

$\ldots, o, v \quad \longrightarrow \quad \ldots$

[1]The treatment of *invokespecial* is in fact nontrivial, for it is the instruction by which class constructors and private methods are invoked. Details can be found in an upcoming technical report.

Operation: Store the value v into the field $f^{A,F}$ of object instance o.

Type Constraints: The capability type of v is $\mathcal{U}^A$ and $f^{A,F}$ is annotated with capability type $\mathcal{V}^B$ then it must be true that $\mathcal{U}^A <: [\mathcal{V}^B]$.

invokevirtual $m^{A,M}$

Operand Stack:

$\ldots, o, a_1, a_2, \ldots, a_k \quad \longrightarrow \quad \ldots, v$

Operation: Invoke instance method $m^{A,M}$, with arguments a_1, a_2, ..., a_k, on object instance o. Any return value v is pushed into the operand stack.

Type Constraints: Suppose that the capability types of o, a_1, ..., a_k and v are $\mathcal{U}_0^{A_0}$, $\mathcal{U}_1^{A_1}$, ..., $\mathcal{U}_k^{A_k}$ and $\mathcal{U}_{k+1}^{A_{k+1}}$ respectively, and that $m^{A,M}$ has annotation $\mathcal{V}_0^{B_0}(\mathcal{V}_1^{B_1}, \ldots, \mathcal{V}_k^{B_k})\mathcal{V}_{k+1}^{B_{k+1}}$, then it must be the case that $\mathcal{U}_i^{A_i} <: \{m^{A,M}\} \rightarrow \mathcal{V}_i^{B_i}$ for $0 \leq i \leq k$, and $\mathcal{V}_{k+1}^{B_{k+1}} <: \mathcal{U}_{k+1}^{A_{k+1}}$.

4 Solving Protection Problems

In this section, we will look at how the above capability type system may be applied to address a number of classical protection problems recast in the programming language context. According to [9], "a *protection problem* is simply a description of some class of restricted behaviors. A protection problem can be solved in a *protection system* if the system provides some set of mechanisms which, when invoked, guarantee that the behavior of the system will be appropriately restricted." We present a solution to the Prison Mail System Problem, and discuss how the solution addresses the five protection problems highlighted in Section 2.

Our solution to the Prison Mail System Problem consists of the capability type interfaces for `Mail`, `Prisoner` and `Guard` as presented in Figure 8.

Safe Invocation. The right to access the `Mail.read` is captured in the primitive capability READ. The `Mail` parameter of `Guard.deliver` is not granted this right, because its capability annotation is RECV $\rightarrow$ [READ $\rightarrow \perp$], meaning that the `Mail` argument received by `Guard.deliver` may only be passed to the `Prisoner.receive` method. `Guard.deliver` is therefore forbidden to read the content of `Mail`.

Capability Amplification. When `Guard.deliver` passes the `Mail` reference to `Prisoner.receive`, capability amplification occurs. Specifically, as an argument of `Prisoner.receive`, the `Mail` reference acquires the capability type [READ $\rightarrow \perp$], which allows the reference to be stored and subsequently be used for accessing `Mail.read`. The `Mail` message can therefore be consumed by the receiving `Prisoner`.

Limiting Propagation of Capability. Since the `Mail` argument of `Guard.deliver` has capability type RECV $\rightarrow$ [READ $\rightarrow \perp$], propagation of the reference to other `Guard` is not permitted.

Mediated Communication. When the `Prisoner.send` method is invoked by the application core, it receives the identity of the addressee as a `Prisoner` argument. This argument has

```
class Mail {
 capability READ = { string read() }
 method Mail(string) : ⊥ (⊥) void
 method string read() : ⊥ () ⊥
 field string msg : ⊥
}

class Prisoner {
 capability SEND = { void send(Prisoner, Guard) }
 capability RECV = { void receive(Mail) }
 method void send(Prisoner, Guard)
   : ⊥ (DLVR → RECV → ⊥, DLVR → ⊥) void
 method void receive(Mail)
   : ⊥ ([READ → ⊥]) void
}

class Guard {
 capability DLVR = { void deliver(Mail, Prisoner) }
 method void deliver(Mail, Prisoner)
   : ⊥ (RECV → [READ → ⊥], RECV → ⊥) void
}
```

Figure 8: Solution to the Prison Mail System Problem

capability type DLVR $\rightarrow$ RECV $\rightarrow \perp$, and as such it can only be passed to `Guard.deliver` and subsequently to `Prisoner.receive`. The sending `Prisoner` is therefore disallowed from invoking `Prisoner.receive` directly. Mediation through `Guard.deliver` is mandatory.

Flexible Control of Capability Storing. Notice also that, having capability type DLVR $\rightarrow$ RECV $\rightarrow \perp$, the `Prisoner` argument received by `Prisoner.send` cannot be stored into a field. Access to the addressee is therefore transient. In contrary, the `Mail` argument passed to `Prisoner.receive` has capability type [READ $\rightarrow \perp$], and as such it can be stored by the receiving `Prisoner` for future consumption.

In summary, our capability type system is expressive enough to address all the protection problems exemplified in the Prison Mail System[2].

5 Preventing Capability Spoofing

The above capability type system does not prevent a class of capability spoofing attacks. For instance, a `Guard` object may create a collaborating `Prisoner` who impersonates the receiving Prisoner, and subsequently leaks a readable reference of `Mail` to the malicious `Guard`. Downcasting may also be exploited for the same purpose. Capability spoofing may also be launched indirectly through the invocation of static methods or exposing access to static fields. To prevent capability spoofing, an orthogonal type system for controlling object creation and downcasting is designed to complement our capability type system. Details of the mentioned attack and this complementary type system will be given in an upcoming technical report. The main ideas are outlined below.

5.1 Subsystem Annotations

The Java reference type hierarchy is decomposed into a *tree of subsystems*. Every Java reference type is *owned* by a unique subsystem. A Java reference type A is a *member* of a subsystem S if A is owned by S or a descendant subsystem of S. System classes such as `Object` and `String` are owned by the *root subsystem.* The following restrictions are imposed.

Subtyping. If a Java reference type A is a subtype of another Java reference type B, and B is owned by subsystem S, then A must be a member of S.

Static members access. A static member declared in Java reference type B may be accessed from a Java reference type A only if B is owned by a subsystem of which A is a member.

Object instantiation. A Java class A may create an instance of another Java class B only if B is owned by a subsystem of which A is a member.

Downcasting. Downcasting of an object reference to Java reference type B can be performed in a Java class A only if B is owned by a subsystem of which A is a member.

[2]Due to space constraints, we are not able to demonstrate how the choice operator ($\sqcap$) can be employed to solve the Confused Deputy problem [22]. Details can be found in an upcoming technical report.

Enforcing these restrictions involves the annotation of classfiles with subsystem ownership information, and the specification of type rules for JVM bytecode instructions *new*, *checkcast*, *instanceof*, *getstatic*, *putstatic* and *invokestatic*, both of which are straightforward.

5.2 The Prison Mail System Revisited

Subsystem annotations can be applied to avoid capability spoofing in the Prison Mail System application. Specifically, three subsystems are defined: *AppCore*, *PrisonerSys* and *GuardSys*. The subsystems *PrisonerSys* and *GuardSys* are children of the *AppCore* subsystem, which in turn is a child of the root subsystem. The `Prisoner` and `Guard` interfaces are owned by the *PrisonerSys* and *GuardSys* subsystems respectively, while the `Mail` class is owned by subsystem *AppCore*. Because *PrisonerSys* and *GuardSys* are sibling subsystems, instances of `Guard` are not permitted to create or downcast instances of `Prisoner`, and vice versa. This effectively removes the possibility of capability spoofing.

There is, however, one subtlety. How can Java code owned by *AppCore* creates instances of `Prisoner` and `Guard` in the first place? As in any extensible Java application, software extensions are always loaded and instantiated by the Java Reflection API, which is accessible only to trusted classes (e.g., the application core), and is not accessible to untrusted code (e.g., dynamically loaded extensions).

6 Implementing Capability Types

This section presents an implementation strategy we plan to undertake to realize our capability type system. The goal of this section is to convince readers that such an implementation is feasible.

Frontend. The frontend component in Figure 2, which extracts capability type interfaces from Java source files, can be implemented in the framework of the `javadoc` tool of the Java SDK. Custom doclets and taglets will be designed, so that capability type interfaces can be embedded in Java source files as comments.

Backend. The Prelude library [15] is a set of C functions for preprocessing Java classfiles. It can be employed to build the backend component of Figure 2 for injecting capability type annotations into Java classfiles.

Link-time Typechecker. The Aegis VM [15] is an open source JVM supporting a Pluggable Verification Module architecture [16, 18]. Static analyzers can be incorporated into the dynamic linking process of the Aegis VM with ease. Intrachecking can be performed with a typical iterative dataflow analysis algorithm, while interchecking can be performed in the framework of proof linking [19, 18]. Prior experience [16, 18] with employing the Pluggable Verification Module architecture to implement the JAC type system [27] demonstrates the feasbility of such an approach. We do however anticipate a nontrivial technical challenge: checking for subtyping of capability types amounts to solving a subgraph isomorphism problem. Without the support of empirical data, one cannot be sure of how intensive such computation is in practice. An interesting approach to address the problem is to adopt the spirit of Proof-Carrying Code [33, 32], and request the code producer (i.e., the backend) to precompute the homomorphisms involved in all subtyping checks.

7 Related Work

Secure cooperation [35] and its variations such as the Mutual Suspicion Problem [37], the Confused Deputy [22], the Safe Invocation Problem [35], and Layered Protection [18] have been studied in the security literature.

The limitations of various language-based protection mechanisms such as stack inspection, execution monitoring and code rewriting have been formally studied in recent years [36, 28, 21, 20, 17].

The notion of capabilities was first proposed by Dennis and Van Horn [10]. An archetypical capability-based operating system kernel is Hydra [9]. A capability-based security kernel for Java is J-Kernel [25], which is implemented as a class library, and relies on a combination of bytecode rewriting, dynamic checks and avoidance of structure sharing to enforce protection. Our type system is statically enforceable, and supports secure structure sharing.

Previous type systems for modelling access control are deeply influenced by stack inspection, and thus usually take a hierarchical perspective on protection [39, 20, 23]. To the best knowledge of the authors, our capability type system is the first of its kind to adopt a peer-to-peer perspective on access control.

Side effects make it difficult to reason about the behavior of a program. Alias control type systems [24, 31, 3, 34, 8, 43, 6, 2] were originally proposed to control the proliferation of aliases in object-oriented programming systems. The intent is that selective elimination of aliasing reduces the scope of side effects. Vitek and Bokowski's work on confined types [43] is a first attempt of applying alias control to address security issues. Our work, however, is the first to offer a uniform reinterpretation of capabilities as alias control.

8 Concluding Remarks

Summary. We have presented a capability type system designed for addressing the protection needs of dynamically extensible software systems. Not only have we proposed a novel protection mechanism in which an application core can impose communication protocols among untrusted software extensions, our capability type system also offers a fresh reinterpretation of capability in terms of alias control.

Future Work. A number of future directions are suggested by this work. Firstly, we would like to study the soundness of our capability type system formally in the framework of Featherweight Java [26]. Secondly, our capability type system can be seen as a lightweight partial specification language. We would therefore like to provide tool support for developers of extensible systems to articulate and validate capability type interfaces. Thirdly, we would like to employ our capability type system to demonstrate that there is a close connection between access control and software architecture. Our conjecture is that the notion of communication integrity [30] in software architecture can be understood as protection problems [9], and thus cross-pollination between the study of software architecture and that of software security should be attempted.

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*, San Diego, California, USA, February 2003.

[2] Jonathan Aldrich, Valentin Kostadinov, and Craig Chambers. Alias annotations for program understanding. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 311–330, Seattle, Washington, USA, November 2002.

[3] Paulo Sérgio Almeida. Balloon types: Controlling sharing of state in data types. In *Proceedings of the 11th European Conference for Object-Oriented Programming*, Jyvaskyla, Finland, June 1997.

[4] Allen L. Ambler and Charles G. Hoch. A study of protection in programming languages. In *Proceedings of an ACM Conference on Language Design for Reliable Software*, pages 25–40, Raleigh, North Carolina, March 1977.

[5] Brian Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gun Sirer, David Becker, Marc Fiuczynski, Craig Chambers, and Susan Eggers. Extensibility, safety and performance in the SPIN operating system. In *Proceedings of the 15th ACM Symposium on Operating System Principles*, pages 267–284, Copper Mountain, Colorado, December 1995.

[6] John Boyland, James Noble, and William Retert. Capabilities for sharing: A generalisation of uniqueness and read-only. In *Proceedings of the 15th European Conference for Object-Oriented Programming*, volume 2072 of *Lecture Notes in Computer Science*, pages 2–27, Budapest, Hungary, June 2001.

[7] Antonio Carzaniga, Gian Pietro Picco, and Giovanni Vigna. Designing distributed applications with mobile code paradigms. In *Proceedings of the 19th International Conference on Software Engineering*, pages 22–32, Boston, Massachusetts, May 1997.

[8] David Clarke, John Potter, and James Noble. Ownership types for flexible alias protection. In *Proceedings of the 13th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 48–64, Vancouver, BC, Canada, October 1998.

[9] Ellis Cohen and David Jefferson. Protection in the Hydra operating system. In *Proceedings of the 5th ACM Symposium on Operating System Principles*, pages 141–160, Austin, Texas, USA, November 1975.

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

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

[12] Guy Edjlali, Anurag Acharya, and Vipin Chaudhary. History-based access control for mobile code. In *Proceedings of the 5th ACM Conference on Computer and Communications Security*, pages 38–48, San Francisco, California, USA, November 1998.

[13] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole Jr. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium on Operating System Principles*, Copper Mountain, Colorado, December 1995.

[14] Wan Fokkink. *Introduction to Process Algebra*. Springer, 2000.

[15] Philip W. L. Fong. The Aegis VM Project. `http://aegisvm.sourceforge.net`.

[16] Philip W. L. Fong. Pluggable verification modules: An extensible protection mechanism for the JVM. Technical Report CS 2003-11, Department of Computer Science, November 2003.

[17] Philip W. L. Fong. Access control by tracking shallow execution history. In *Proceedings of the 2004 IEEE Symposium on Security and Privacy*, Oakland, California, USA, May 2004.

[18] Philip W. L. Fong. *Proof Linking: Modular Verification Architecture for Mobile Code Systems*. Ph.D. dissertation, School of Computing Science, Simon Fraser University, January 2004.

[19] 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.

[20] Cédric Fournet and Andrew D. Gordon. Stack inspection: Theory and variants. *ACM Transactions on Programming Languages and Systems*, 25(3):360–399, May 2003.

[21] Kevin W. Hamlen, Greg Morrisett, and Fred B. Schneider. Computability classes for enforcement mechanisms. Technical Report TR 2003-1908, Computer Science Department, Cornell University, August 2003.

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

[23] Tomoyuki Higuchi and Atsushi Ohori. A static type system for JVM access control. In *Proceedings of the 8th ACM SIGPLAN International Conference on Functional Programming*, pages 227–237, Uppsala, Sweden, 2003.

[24] John Hogg. Islands: Aliasing protection in object-oriented languages. In *Proceedings of the ACM Conference on Object-oriented Programming, Systems, Languages, and Applications*, pages 271–285, Phoenix, Arizona, November 1991.

[25] Chris Howblitzel, Chi-Chao Chang, Grzegorz Czajkowski, Deyu Hu, and Thorsten von Eicken. Implementing multiple protection domains in java. In *Proceedings of the 1998 USENIX Annual Technical Conference*, New Orleans, Louisiana, USA, June 1998.

[26] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, May 2001.

[27] Günter Kniesel and Dirk Theisen. JAC — access right based encapsulation for Java. *Software — Practice and Experience*, 31(6):555–576, May 2001.

[28] Jay Ligatti, Lujo Bauer, and David Walker. Edit automata: Enforcement mechanisms for run-time security policies. *International Journal of Information Security*, 2003. To appear.

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

[30] David C. Luckham and James Vera. An event based architecture definition language. *IEEE Transactions on Software Engineering*, 21(9):717–734, September 1995.

[31] Naftaly Minsky. Towards alias-free pointers. In *Proceedings of the 10th European Conference for Objected Oriented Programming*, volume 1098 of *Lecture Notes in Computer Science*, pages 189–209, Linz, Austria, July 1996. Springer.

[32] George C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages*, pages 106–119, Paris, France, January 1997.

[33] George C. Necula and Peter Lee. Safe kernel extensions without run-time checking. In *Proceedings of the Second Symposium on Operating System Design and Implementation (OSDI'96)*, pages 229–243, Seattle, Washington, October 1996.

[34] James Noble, Jan Vitek, and John Potter. Flexible alias protection. In *Proceedings of the 12th European Conference for Object-Oriented Programming*, volume 1445 of *Lecture Notes in Computer Science*, pages 158–185, Brussels, Belgium, July 1998. Springer.

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

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

[37] Michael D. Schroeder. *Cooperation of Mutually Suspicious Subsystems in a Computer Utility*. Ph.D. thesis, Massachusetts Institute of Technology, September 1972.

[38] R. Sekar, V. N. Venkatakrishnan, Samik Basu, Sandeep Bhatkar, and Daniel C. DuVarney. Model-carrying code: A practical approach for safe execution of untrusted applications. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 15–28, Bolton Landing, NY, USA, October 2003.

[39] Christian Skalka and Scott Smith. Static enforcement of security with types. In *Proceedings of the 5th ACM SIGPLAN International Conference on Functional Programming*, pages 34–45, Montreal, Québec, Canada, September 2000.

[40] Tommy Thorn. Programming languages for mobile code. *ACM Computing Surveys*, 29(3):213–239, September 1997.

[41] Úlfar Erlingsson and Fred B. Schneider. SASI enforcement of security policies: A retrospective. In *Proceedings of the 1999 New Security Paradigms Workshop*, pages 87–95, Caledon Hills, Ontario, Canada, September 1999.

[42] Úlfar Erlingsson and Fred B. Schneider. IRM enforcement of Java stack inspection. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy*, pages 246–255, Berkeley, California, USA, May 2000.

[43] Jan Vitek and Boris Bokowski. Confined types in Java. *Software — Practice and Experience*, 31(6):507–532, May 2001.

[44] 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.