Link-Time Enforcement of Confined Types for JVM Bytecode

Philip W. L. Fong

Technical Report CS-2004-12
December 10, 2004

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

ISBN 0-7731-0505-0
ISSN 0828-3494

Link-Time Enforcement of Confined Types for JVM Bytecode

Philip W. L. Fong
Department of Computer Science
University of Regina
Regina, Saskatchewan, Canada S4S 0A2
Tel: (306) 585-5228
Fax: (306) 585-4745
Email: pwlfong@cs.uregina.ca

First Draft: December 10, 2004
Revised: January 24, 2005

Abstract

Confined types enforce a form of encapsulation stronger than what the standard Java type system offers. It is known that confined types effectively prevent accidental reference leaks that could lead to security breaches in object-oriented programs. Because existing formulations of confined types target the Java source language, they can only be enforced by the code producer at compile time. As such these formulations fail to qualify as practical language-based protection mechanisms for the Java platform.

This paper presents a capability-based formulation of confined types for JVM bytecode, and reports the first implementation to enforce confined types at link time, on the code consumer side. This novel formulation of confined types is backward compatible, modular, and interoperable with lazy dynamic linking. The paper also demonstrates how this bytecode-level formulation of confined types can be applied to facilitate a form of secure cooperation between mutually suspicious code units.

1 Introduction

Security is an essential attribute of dependable systems. The Java platform, a language environment designed with security in mind, has been considered to be a platform of choice for developing and deploying networked and extensible software systems with demanding security requirements. At the core of the Java security infrastructure is a strong type system, which provides non-bypassable *encapsulation boundaries* for controlling access to privileged services and sensitive data. To appreciate the pivotal role of type soundness and encapsulation in the Java security enterprise, one only needs to recall that the soundness of the Java type system guarantees that no type confusion will occur, and thus the security manager will be properly encapsulated, and consequently the rest of the Java protection infrastructure can function as designed.

Both the Java source language and the Java Virtual Machine (JVM) [20] provide access control qualifiers (e.g., `public`, `protected`, etc) for enforcing the usual notion of data encapsulation. The Java platform, however, offers no provision for enforcing the stronger notion of *reference encapsulation*. This lack of programmatic support for preventing accidental reference leaking has led to a security breach in the `java.security` package of JDK 1.1.

The idea of confined types [31, 15, 34] was proposed as a lightweight annotation system for supporting reference encapsulation in the Java platform. It has been shown convincingly that proper adoption of confined types in Java could have prevented the aforementioned security breach [31]. Unfortunately, because existing formulations of confined types target the Java source language, they can only be enforced by the code producer at compile time. In the context of the JVM, in which code units bind via dynamic linking [19], program verification that is performed against source code, or administrated only by the code producer, cannot be trusted [23]. Consequently, these source-level formulations of confined types do not allow us to leverage reference encapsulation as a practical language-based protection mechanism [28] for the Java platform.

This paper presents the first formulation of confined types for JVM bytecode, which can be administrated by the code consumer at link time. This formulation of confined types offers a number of unique features:

1. The typing discipline is structured as a *capability type system*, thereby significantly simplifying the formulation and enforcement of the typing discipline.
2. The formulation is *backward compatible*, in the sense that (a) classfiles decorated with confined type annotations fully conform to the standard classfile format prescribed in the JVM specification [20], and, more importantly, (2) no retro-annotation of the existing classes in the Java platform class library is necessary. Consequently, the proposed formulation is fully compatible with the standard Java platform classes.
3. The type checking procedure is *modular*, meaning that type checking can be conducted one classfile at a time, thereby allowing a type checker to be integrated into the lazy dynamic linking environment of the Java platform, in which the code units that make up an application may not be completely loaded at run time.
4. The validation of intermodular dependencies is *staged* according a schedule consistent with the lazy dynamic linking semantics of Java, so that eager classloading is completely eschewed, and thus unnecessary performance overhead is avoided.

This paper also reports the first implementation of confined types as a link-time, language-based protection mechanism. Such an implementation is made possible by an extensible protection mechanism, Pluggable Verification Modules (PVMs) [10, 11], available in an open source JVM, the Aegis VM [9]. PVM is a general framework for introducing third-party bytecode verification procedures safely into the dynamic linking process of the JVM. The adoption of PVM as an enabling technology significantly reduces the engineering effort and the trusted computing base of our implementation.

The benefit of a bytecode-level formulation of confined types is assessed in the context of dynamically extensible software systems, in which both trusted and untrusted code coexist in the same execution environment. This paper demonstrates how safe dynamic linking can be coupled with bytecode-level confined types to facilitate a form of secure cooperation [27, 29], in which mutually suspicious code units are protected from their potentially malicious peers. Such an application would not have been possible if confined types are enforced only at compile time.

This paper is organized as follows. The notion of confined types is introduced in the next section. A reformulation of confined types as a type system for JVM bytecode is presented in Section 3. The implementation of this formulation of confined types as a PVM is discussed in Section 4. A case study demonstrating the utility of confined types for JVM bytecode in supporting secure cooperation is discussed in Section 5. The paper closes with related work, future work and conclusion.

2 Confined Types for Java

This section provides an overview of confined types as an augmented type system for the Java source language. Consult [31] for more details.

2.1 Motivation: Security Breach Caused By Reference Leaking

The notion of confined types is motivated by the aforementioned reference leaking bug as presented in [31]. In Java 1.1, every `Class` object carries an array of signers that represent the principals under which the class acts.

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

A defective implementation of Java 1.1 naively returns the signers array owned by `Class` as the value of the `getSigners` method. Malicious caller who obtains this array will then be able to tweak the contents of the array to amplify the access rights of any class. A simple fix of the security breach is given below:

```
public class Class {
  private Identity[] signers;
```

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

This fix, however, is not a sound engineering solution to the problem, because the fix still relies on programmers to exercise extreme care, the lack of which is essentially the cause of the security breach in the first place. Instead, one desires a solution that completely rules out the possibility of committing the careless mistake all together. To this end, a new type qualifier can be introduced to Java to declare that a reference type is *confined*, meaning that object references of that type (or arrays of that type) are not allowed to escape from the package in which the reference type is declared. Exploiting this, one may rename the `Identity` class to `ConfinedIdentity`, and declare it as `confined`:

```
confined class ConfinedIdentity { ...}
```

Now, the typing discipline confined types will ensure that `ConfinedIdentity` objects never escape from their defining packages. Programmers are thus forced by the typing discipline to develop code that does not leak `ConfinedIdentity` references:

```
public class Identity {
  SecureIdentity rep;
  Identity(SecureIdentity si) { rep = si; }
  ...
}

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

The confinement type rules are summarized in the following. The particular formulation of confined types presented in this section is a rational synthesis of the various formulations presented in [31, 15, 34].

2.2 Anonymous Methods

An inherited method (or constructor) may be invoked on a confined reference. Since the supertype in which the inherited method is declared may not be confined, the inherited method

$\mathcal{A}1$	The pseudo-parameter `this` of an anonymous method must only be used for accessing fields and as the receiver in the invocation of anonymous methods. More specifically, `this` must not be passed as an argument to a non-anonymous method, returned as a method return value, or stored into a field.
$\mathcal{A}2$	Anonymous methods shall only be overridden by anonymous methods.
$\mathcal{A}3$	Anonymous methods must not be `native`.

Figure 1: Anonymity Rules

$\mathcal{C}1$	A confined type must not be `public`.
$\mathcal{C}2$	Subtypes of a confined type must be confined.
$\mathcal{C}3$	The type of `public` or `protected` fields and the return type of `public` or `protected` methods must not be confined.
$\mathcal{C}4$	A confined type must not be widened to an unconfined type.
$\mathcal{C}5$	Methods invoked on a confined object must either be non-`native` methods declared in a confined type or be anonymous methods.

Figure 2: Confinement Rules

may leak the confined receiver reference it obtains via the `this` pseudo-parameter. It is therefore desirable to provide a means for a supertype to promise to its subtypes that an instance method never leaks the `this` pseudo-parameter. A confined subclasses can thus safely invoke inherited methods that are annotated as such. To this end a method may be annotated as being *anonymous*. Such a method promises not to leak the pseudo-parameter `this` through field setting and value returning. Figure 1 outlines the type rules regarding anonymity.

2.3 Confined Types

Instances of confined types are encapsulated in their defining package. The type rules for enforcing confinement is given in Figure 2. $\mathcal{C}1$ guarantees that code units defined outside of a package is not allowed to create instances of a confined type. $\mathcal{C}2$ assures that downcasting and dynamic binding does not bypass confinement. $\mathcal{C}3$ makes sure that confined objects are not leaked via field access and method return. $\mathcal{C}4$ requires that transitive data flow observes confinement boundaries[1]. $\mathcal{C}5$ mandates proper interfacing between code in confined types and methods inherited from unconfined types.

[1] The formulation of this rule in [34] is apparently more restrictive: "Confined types can only be widened to other types confined in the same package." The discrepancy is, however, immaterial. Confined types are package private, and thus their subtypes always belong to the same package. In addition,[15] correctly observes that a redundant rule in [31] concerning exceptions is properly subsumed by this rule.

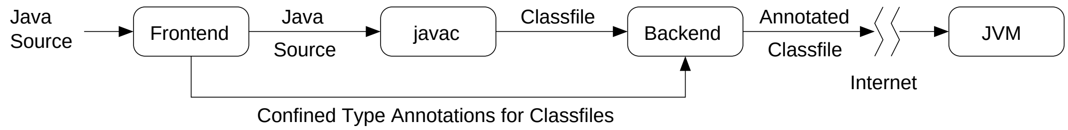

Figure 3: Processing of Confined Type Annotations

3 Confined Types for JVM Bytecode

3.1 Assumptions and Notations

This section presents a formulation of confined types for JVM bytecode. To appreciate the contributions of this paper, it is instrumental to understand that there are *three* type systems involved in the present discussion: (1) the standard Java type system, (2) the source-level confined type system as described in the previous section, and (3) the bytecode-level confined type system to be presented in this section.

Firstly, there is the standard Java type system. We use A and B to denote a Java reference type, which is a class, interface, or array type in the standard Java type system. Methods and fields are denoted by m and f respectively. A constant pool reference[2] is denoted by its referent delimited by angled brackets ($\langle\cdot\rangle$). For example, $\langle A \rangle$ denotes a constant pool class reference that resolves to the Java reference type A. Similarly, $\langle A.m \rangle$ and $\langle A.f \rangle$ denote respectively a constant pool method reference that resolves to method m declared in A and a constant pool field reference that resolves to field f declared in A.

Secondly, there is the source-level confined type system. We assume that Java source files are properly annotated with confinement information: a Java class or interface can be annotated as being confined, and an instance method can be annotated as being anonymous. Consult [31] for a discussion of how such source-level annotations can be achieved.

Thirdly, there is the bytecode-level confined type system. We envision a programming environment (Figure 3) in which confinement information embedded in Java source files is extracted by a compiler frontend, translated into classfile annotations, and subsequently injected into the classfiles generated by the standard `javac` compiler. Type checking will be conducted by the JVM at link time, against classfiles, at the bytecode level. It is the design and implementation of this bytecode-level confined type system that this paper claims to be contributions.

3.2 Confined Types as Capabilities

The bytecode-level formulation of confined types takes the form of a *capability type system* [4]. A capability is traditionally understood as an object reference plus a set of access rights [8, 7]. In a capability type system, every object reference is statically assigned a capability type, which prescribes what operations can be performed on the value. A capability type may impose on the underlying value a set of operational restrictions that constrains the way the value may be accessed. Three capability types are defined in our confined type system:

[2]The symbol table of a classfile is called a constant pool [20].

- The bottom type ($\perp$) is the least restrictive capability type. It is used for typing both unconfined object references and primitive type data (e.g., `int`, `boolean`, etc). Values typed as $\perp$ have the same semantics as in standard Java, and as such they can be freely stored into any field, passed as an argument to any method, or returned as values to method invocations.
- The `confined` type is used for typing those fields, method parameters (including the `this` pseudo-parameter) and return values for which the underlying Java reference type is confined. Object references typed as `confined` can only be transferred to locations that are marked explicitly as `confined` or `anonymous` (see below).
- The `anonymous` type is used for typing the `this` pseudo-parameter of an anonymous method. Object references typed as `anonymous` can only be passed via the `this` pseudo-parameter to an anonymous method, or used for accessing fields.

The three capability types form a subtyping hierarchy as shown below:

$$\perp <: \texttt{confined} <: \texttt{anonymous}$$

where the *subtyping relation* $<:$ is reflexive, transitive and antisymmetric. As in a typical capability type system, less restrictive capability types are subtypes of more restrictive ones. This permits the assignment of less restrictively typed object references to more restrictively typed variables, while forbidding assignment in the opposite direction.

In this capability-based formulation of confined types, confined-ness and anonymity are viewed as capabilities of object references rather than a property of a class or a method. As we shall see in Sections 3.4 and 3.5 respectively, this design choice has greatly simplified the data flow analysis involved in intraprocedural type checking, and reduces the amount of classloading overhead in inter-modular type checking.

3.3 Confined Type Interface and Backward Compatibility

Confined type interface. Associated with each Java reference type A is a *confined type interface* $\langle \mathcal{E}_A, \mathcal{I}_A \rangle$, where $\mathcal{E}_A$ is a set of *export type assertions*, and $\mathcal{I}_A$ a set of *import type assertions*. The set $\mathcal{E}_A$ assigns a confined type annotation to each symbol *exported* by the classfile of A: this includes the reference type A itself, and the fields and methods declared in the classfile. Similarly, the set $\mathcal{I}_A$ assigns a confined type annotation to each symbol *imported* by the classfile of A: this includes the class, field, and method references in the constant pool of the classfile. An export type assertion, that is, a member of $\mathcal{E}_A$, must take one of the following forms:

$A : T$ assigns a confined type T to the Java reference type A, that is, the reference type defined by the classfile. For example, $A : \texttt{confined}$ declares A to be confined, while $A : \perp$ declares otherwise.

$f : T$ assigns a confined type T to the field f declared in the classfile. For example, $f : \texttt{confined}$ declares that values stored in f has a `confined` capability.

$m : T_0(T_1, T_2, \ldots, T_k)T$ assigns confined types to the formal parameters and the return value of a method m declared in the classfile. Specifically, T_0 is assigned to the `this` pseudo-parameter, T_1, T_2, $\ldots T_k$ to the parameters on the formal parameter list, and T to the return value. The intention is that assigning `anonymous` to T_0 declares that m is an anonymous method. Notice that, for uniformity purposes, T_0 is present even for static methods. In that case, T_0 is $\perp$ (see below).

A number of well-formedness constraints must be observed by a confined type interface:

- The capability type $\perp$ must be used for annotating fields, formal parameters, and return values with primitive types (i.e., `int`, `boolean`, etc). The placeholder (i.e., T_0) for `this` in the annotation of a static method must be $\perp$.
- When the capability type `confined` is used for annotating fields, formal parameters (including the `this` pseudo-parameter of an instance method), and return values, the annotated entity must have a Java reference type belonging to the same package as A[3].
- The capability type `anonymous` must only be used for annotating the `this` pseudo-parameter of an instance method.

An import type assertion, that is, a member of $\mathcal{I}_A$, must take one of the following forms:

$$\langle B \rangle : T \qquad \langle B.f \rangle : T \qquad \langle B.m \rangle : T_0(T_1, T_2, \ldots, T_k)T$$

where T, T_0, T_1, ... are confined types. These assertions assign confined type annotations to the referents of constant pool references. Well-formedness of import type assertions is defined analogous to their export counterparts.

Backward compatibility. As pointed out earlier (Figure 3), classfiles must be properly annotated to carry a confined type interface in order for the confined type system to be enforceable at link time. This would suggest that all classfiles belonging to the standard Java platform will have to be explicitly annotated, a tedious process that should be avoided. To this end, a classfile is allowed to be left unannotated, and thus assumes a *default* confined type interface: the export type annotation of the underlying class and the import type annotation of every class reference is $\perp$, so is that for a field or a field reference; similarly, the default annotation of a method or a method reference is such that the return value and all formal parameters (including the pseudo-parameter `this`) have type $\perp$. Since this definition of default annotation is consistent with the standard Java type system, classfiles that do not carry explicit confined type annotations behave in accordance with the standard Java semantics. The provision of assuming a default type interface for an unannotated classfile renders it possible to reuse legacy classfiles not compiled for confined type checking. This is particularly handy in the case of the standard Java platform class library — hundreds of system classes can be reused as is. The current design of confined type interface is therefore backward compatible to legacy Java code.

3.4 Modular Intrachecking

Code safety is in general a whole-program notion: the safety of a classfile depends not only on properties that can be established by examining the classfile alone, but also on the compatibility

[3]This constraint is instrumental for preserving the soundness of this bytecode-level formulation of confined types. The type system of the Java source language is restrictive enough that such a constraint is not necessary in a source-level formulation of confined types. The JVM bytecode language, however, has a more type system, and thus necessitates the inclusion of this constraint into the bytecode-level formulation of confined types.

of the established properties with the runtime environment into which the classfile is linked. In the context of type checking, the two tasks correspond to the validation of the internal consistency of a type interface for a given code unit, and the validation of the compatibility between this type interface and a given type environment. Cardelli succinctly calls the two tasks *intrachecking* and *interchecking* [5].

One desirable property for a typing discipline is the provision for modular type checking. That is, intrachecking and interchecking should be cleanly separated, so that intrachecking of a code unit can be performed in the absence of other code unit. The design of the confined type interface renders type checking fully modular, meaning that intrachecking can be performed without consulting the confined type interface of external classfiles.

Intrachecking of a class must be conducted prior to class preparation [20, Section 5.4.2]. This may be as early as when a class is defined [20, Section 5.3.5], or as late as during standard bytecode verification [20, Section 5.4.1]. Two sets of checks are involved: (1) validating the integrity of the confined type interface (Sect. 3.4.1); (2) validating if the body of each bytecode method conforms to the method's export type assertion (Sect. 3.4.2).

3.4.1 Integrity of Confined Type Interface

Besides trivial checks of well-formedness (Section 3.3), the following checks apply when a class A is intrachecked.

- If $A : \texttt{confined} \in \mathcal{E}_A$, then A must not be public (i.e., $\mathcal{C}1$ in Figure 2).
- If $f : \texttt{confined} \in \mathcal{E}_A$, then f must not be public or protected (i.e., $\mathcal{C}3$ in Figure 2).
- If $m : T_0(T_1, T_2, \ldots, T_k)T \in \mathcal{E}_A$, where $T = \texttt{confined}$, then m must not be public or protected (i.e., $\mathcal{C}3$ in Figure 2).
- If $m : T_0(T_1, T_2, \ldots, T_k)T \in \mathcal{E}_A$, and m is a native method, then $T_0 = T_1 = \ldots = T_k = T = \perp$ (i.e., $\mathcal{A}3$ in Figure 1).

These are straightforward reformulation of source-level type rules (Figure 1 and 2) in terms of confined type assertions.

3.4.2 Analysis of Bytecode Methods

The export type assertion of a bytecode method is valid only if every program point in the method body can be consistently assigned a *confined type state*. The confined type state for a program point is in turn an assignment of a confined type to every location in the local variable array and the operand stack. A consistent confined type state assignment is a solution to the following constraint system:

1. The export type assertion of a bytecode method imposes typing constraints on the starting program point: the operand stack should be empty; the local variable array should be initialized with confined type annotations prescribed in the formal parameter list of the export type assertion.
2. Every JVM bytecode instruction imposes typing constraints on the confined type states at the program points immediately before and after and instruction.

Such a constraint system can be solved efficiently using a standard work-list algorithm for data flow analysis [24].

The typing constraints of JVM bytecode instructions are presented in this section. The effect of executing a bytecode instruction is presented in a notation popularized by [20]. For example, the *iadd* instruction pops two integers off the top of the operand stack, and subsequently push their sum. This can be illustrated as follows.

$$\ldots, i_1, i_2 \quad \longrightarrow \quad \ldots, i_3$$

where integer i_3 is the sum of i_1 and i_2. Most of the JVM bytecode instructions manipulate primitive type values, which trivially assume the capability type $\perp$, and thus retain their standard Java semantics. For example, confined type states before and after an *iadd* instruction must be constructed so that $i_1 : \perp$, $i_2 : \perp$, and $i_3 : \perp$. The typing constraints for most of the bytecode instructions are trivial variants of this theme. The remaining of this section presents the typing constraints of bytecode instructions that are non-trivial exceptions to the standard pattern. The goals are to illustrate how the design of confined type interface facilitate modular intrachecking, and to points out subtleties not apparent in the high level description of the previous sections. A more complete list of type rules can be found in Appendix A.

Object creation. An instance of a confined Java reference type acquires its `confined` capability when it is created. From that point on, it can either be propagated within its defining confinement domain, or be promoted to the more restrictive `anonymous` capability type when it is pass as `this` to an anonymous method. Because of the design of the subtyping hierarchy, at no point shall such an instance acquire the $\perp$ capability type.

new $\langle B \rangle$

> **Operand Stack:** $\ldots \quad \longrightarrow \quad \ldots, v$
>
> **Operation:** Create a new instance v of Java reference type B.
>
> **Type Constraints:** Suppose $\langle B \rangle : T \in \mathcal{I}_A$ and $v : T_v$. Then $T <: T_v$.

A caveat is that string literals are created to have bottom type $\perp$ because the `java.lang.String` class has a default confined type interface.

ldc/ldc_w $\langle$***utf8-literal***$\rangle$

> **Operand Stack:** $\ldots \quad \longrightarrow \quad \ldots, v$
>
> **Operation:** Push a `String` literal v onto the operand stack.
>
> **Type Constraints:** If $v : T_v$ then $\perp <: T_v$. That is, the reference v may assume any type.

Basic interprocedural data flow. The following rules capture basic interprocedural data flow in a bytecode method. They are the main engine behind the enforcement of source-level type rules $\mathcal{A}1$, $\mathcal{C}4$ and $\mathcal{C}5$ (see Figures 1 and 2). Note how the source-level type rules are elegantly captured by the design of the subtyping hierarchy. Also, the provision of import type assertions enables one to perform intrachecking without consulting external classes.

getfield $\langle B.f \rangle$

> **Operand Stack:** $\ldots, o \quad \longrightarrow \quad \ldots, v$

Operation: Retrieve the value v of field $\langle B.f \rangle$ from object instance o.

Type Constraints: Suppose $\langle B.f \rangle : T \in \mathcal{I}_A$, and $v : T_v$. Then $T <: T_v$.

putfield $\langle B.f \rangle$

Operand Stack: $\ldots, o, v \longrightarrow \ldots$

Operation: Store the value v into the field $B.f$ of object instance o.

Type Constraints: Suppose $\langle B.f \rangle : T \in \mathcal{I}_A$, and $v : T_v$. Then $T_v <: T$.

invokevirtual $\langle B.m \rangle$

Operand Stack: $\ldots, o, a_1, a_2, \ldots, a_k \longrightarrow \ldots, v$

Operation: Invoke method $\langle B.m \rangle$ on object instance o, passing arguments a_1, a_2, $\ldots$, a_k. Any return value v is pushed into the operand stack.

Type Constraints: Suppose $\langle B.m \rangle : T_0(T_1, T_2, \ldots, T_k)T \in \mathcal{I}_A$. Suppose further that $o : T_o$, $a_i : T_{a_i}$, and $v : T_v$. Then $T_o <: T_0$, $T_{a_i} <: T_i$, and $T <: T_v$.

areturn

Operand Stack: $\ldots, o \longrightarrow$

Operation: Return object reference o from current method.

Type Constraints: Suppose $o : T_o$, and $m : T_0(T_1, \ldots, T_k)T \in \mathcal{E}_A$, where m is the current method. Then $T_o <: T$.

Subtle cases. Two subtle cases are considered here. The following rule ensures that source-level type rule $\mathcal{C}4$ (Figure 2) is observed when type casting occurs.

checkcast $\langle B \rangle$

Operand Stack: $\ldots, o \longrightarrow \ldots, v$

Operation: Attempt to cast object reference o to a reference v of type B.

Type Constraints: Suppose $\langle B \rangle : T \in \mathcal{I}_A$, and $o : T_o$ and $v : T_v$. Then $T_o <: T$ and $T <: T_v$.

When an exception is thrown, the confined type of the exception object is implicitly widened to the confined type of the exception handler's catch type. As there is no guarantee which handler will end up be catching an exception, one has to assume conservatively that the confined type of the catch type is $\perp$. A constraint is therefore imposed to force all exceptions to have the bottom type $\perp$.

athrow

Operand Stack: $\ldots, o \longrightarrow o$

Operation: Throw o as an exception.

Type Constraints: If $o : T_o$ then $T_o <: \perp$. That is, $o : \perp$.

Arrays. Special attention was paid to the handling of arrays. Specifically, an array object is considered to be a carrier of its components. Consequently, an array type is confined iff its component is confined. This avoids the leaking of confined objects through the propagation of carrier arrays.

anewarray $\langle B \rangle$

Operand Stack: $\ldots, n \quad \longrightarrow \quad \ldots, a$

Operation: Create an array a of component type B with n components.

Type Constraints: Suppose $\langle B \rangle : T \in \mathcal{I}_A$ and $a : T_a$. Then $T <: T_a$.

aaload

Operand Stack: $\ldots, a, i \quad \longrightarrow \quad \ldots, v$

Operation: Load the reference component v from array reference a at index i.

Type Constraints: If $a : T_a$ and $v : T_v$ then $T_a <: T_v$.

aastore

Operand Stack: $\ldots, a, i, v \quad \longrightarrow \quad \ldots$

Operation: Store reference value v into array reference a as the component at index i.

Type Constraints: If $a : T_a$ and $v : T_v$ then $T_v <: T_a$.

3.5 Incremental Interchecking and Lazy Dynamic Linking

Incremental interchecking. Interchecking involves the assurance of compatibility between confined type interfaces of code units that are dynamically linked together to form an application. The lazy dynamic linking of Java code implies that interchecking must be conducted in an incremental, carefully staged manner. Specifically, the following checks are scheduled to occur at various stages of the dynamic linking process:

1. **Class preparation.** When a class A is prepared [20, Section 5.4.2], the following checks are applied to ensure that the confined type interface of A is consistent with those of its supertypes:
   - Suppose A extends or implements another reference type B. Suppose further that $A : T \in \mathcal{E}_A$ and $B : T' \in \mathcal{E}_B$. Then $T' <: T$ (i.e., $\mathcal{C}2$ in Figure 2).
   - Suppose A declares a method m that overrides a method of the same signature in supertype B, so that $m : T_0(T_1, T_2, \ldots, T_k)T \in \mathcal{E}_A$ and $m : T'_0(T'_1, T'_2, \ldots, T_k)T' \in \mathcal{E}_B$. Then $T'_i <: T_i$ for $0 \leq i \leq k$ and $T <: T'$ (i.e., $\mathcal{A}2$ in Figure 1). In essence, the usual contravariant rule for method type subtyping is in play here [26].
2. **Constant pool resolution.** When a constant pool entry is resolved in class A, the following checks are applied to assure that the export type assertion of the resolved target is consistent with the import type assertion of the constant pool entry.
   - Suppose the constant pool entry being resolved is $\langle B \rangle$. Suppose further $\langle B \rangle : T \in \mathcal{I}_A$, and $B : T' \in \mathcal{E}_B$. Then both $T <: T'$ and $T' <: T$. That is, $T = T'$.
   - Suppose the constant pool entry being resolved is $\langle B.f \rangle$. Suppose further $\langle B.f \rangle : T \in \mathcal{I}_A$, and $f : T' \in \mathcal{E}_B$. Then both $T <: T'$ and $T' <: T$. That is, $T = T'$.
   - Suppose the constant pool entry being resolved is $\langle B.m \rangle$. If $\langle B.m \rangle : T_0(T_1, T_2, \ldots, T_k)T \in \mathcal{I}_A$, and $m : T'_0(T'_1, T'_2, \ldots, T'_k)T' \in \mathcal{E}_B$, then $T_i <: T'_i$ for $0 \leq i \leq k$, and $T' <: T$. Again, the usual contravariant rule is in play here [26].

Preservation of laziness in dynamic linking. Suppose f is a field declared in class A, so that the Java field type of f is B. Suppose further that $B : T \in \mathcal{E}_B$. The intention, of course, is to have $f : T \in \mathcal{E}_A$, meaning that export type annotation faithfully reflects whether the Java field type of f is confined. Yet, it is entire conceivable that $f : T' \in \mathcal{E}_A$, where $T \neq T'$. That is, it is entirely possible for annotations to lie. Similar anomalies can be constructed for the formal parameters and return values of methods, and such constructions can work for both import type assertions and export type assertions. To fix thoughts, the Java types of fields, formal parameters and return values are called auxiliary symbols. For example, B is an auxiliary symbol in A. To rephrase the anomaly, the annotations of auxiliary symbols may not accurately reflect their confined-ness.

This seemingly dangerous state of affairs is in fact quite harmless. The reason is that the intrachecking type rules (e.g., *new*) dictate that a new instance of Java reference type B acquires a capability that accurately reflects the confined-ness of B. In order for this instance of B to successfully escape from its scope of instantiation, the import type assertions of constant pool entries in which B is an auxiliary symbol must be accurate in order for intrachecking to succeed. Similarly, in order for these constant pool entries to resolve successfully, the export type assertions of the resolved targets must match the import type assertions (which are known to be accurate), thereby forcing the export type assertions to be accurate. Consequently, any unsafe annotation of auxiliary symbols will be detected by type checking. This is the beauty of treating confined-ness as a capability rather than an intrinsic property of a Java reference type: it forces code producers not to lie if they want to have their code executed on the code consumer environment.

An alternative design were to explicitly enforce strict conformance between the actual confined-ness of auxiliary symbols and their annotations. This alternative design, however, necessitates the loading of classfiles corresponding to the auxiliary symbols at various stages of interchecking. These additional classloading activities will introduce overhead that is known to cause serious impact to the performance of an application [33].

The capability-based design adopted in this paper intentionally eschews eager classloading by a liberal annotation scheme. The scheme does not demand that annotations accurately reflect the confined-ness of auxiliary symbols. Yet, failure to do so causes either intrachecking or interchecking to fail, thereby protecting the integrity of dynamic linking. Consequently, laziness of dynamic linking is preserved without sacrificing type safety.

4 Implementation Under the PVM Framework

This section reports the first implementation[4] of confined types as a link-time protection mechanism for the JVM.

4.1 Embedding Confined Type Interfaces

To make confined types enforceable at link time, every classfile must carry a confined type interface. This can be achieved by embedding a confined type interface into classfiles through the classfile attribute facility [20, Section 4.7]. Specifically, import and export type assertions

[4]This implementation can be found in the CVS repository of the Aegis VM Project.

are embedded via a compact encoding in a `ConfinedTypes` class attribute. A classfile may carry at most one such attribute; the absence of the attribute in a classfile signals that it assumes a default confined type interface (Section 3.3). A simple Linux command-line tool has been developed to take the role of the backend component in Figure 3, providing a convenient means for manual annotation of classfiles. The classes examined in the case studies in Section 5 were all annotated using this tool.

4.2 Pluggable Verification Module

An implementation of the intrachecking and interchecking procedures described in Section 3 must be incorporated into the dynamic linking process of the JVM in order to enforce the bytecode-level formulation of confined types. The Aegis VM [9] is an open source JVM that offers an extensible protection mechanism called Pluggable Verification Modules (PVMs) [10, 11]. Unlike other implementations of the JVM, in which the link-time bytecode verification service is a fixture that cannot be extended conveniently, the PVM framework is based on a modular verification architecture whereby bytecode verification is a pluggable service that can be readily replaced, reconfigured and augmented. Third-party verification services can be safely incorporated into the dynamic linking process as a PVM. The verification service exported by a PVM will be invoked prior to the preparation of a classfile. The PVM may also schedule programmer-defined checks to occur at various points of the dynamic linking process. The PVM framework also provides reusable facilities for easing the construction of link-time static program analyzers.

In our case, the intrachecking procedure of Section 3.4 has been encapsulated in a PVM, while the interchecking steps of Section 3.5 have been scheduled to take place at the appropriate points of dynamic linking. Implementation began after the formulation in Section 3 was stabilized. The coding effort has been logged using a scheme akin to the Personal Software Process [17]. The first working version of the implementation consists of 2926 lines of moderately commented C code, finished in 35 man-hours spanning over approximately one month. A modest code complexity and rapid development cycle were achieved because the Aegis VM provides reusable facilities to ease the development of link-time verification services. The compactness of the resulting implementation also suggests that link-time enforcement of confined types involves only a modest increase in the size of the *trusted computing base (TCB)*, thereby preserving the tractability of TCB verification and validation.

5 Secure Cooperation

A fundamental security challenge in dynamically extensible software systems has been the facilitation of secure cooperation among mutually suspicious code units within the same application [27, 29]. This section illustrate how a class may exploit the bytecode-level formulation of confined types to impose confinement policies on data structures it shares with untrusted peer classes. Specifically, four tactics will be discussed:

1. **Protection by access contracts.** Confinement policies are embedded in confined type interfaces, specifying a contract between a class and its untrusted collaborators.

2. **Safe dynamic linking.** The interchecking type rules in Section 3.5 (constant pool resolution) ensure that the contract is honored by peer collaborators.
3. **Trust inspiration.** To inspire trust, collaborating code units must formulate matching confined type interfaces. The intrachecking type rules in Section 3.4 ensures that the collaborators indeed live up to their promises.
4. **Secure software extension.** Dynamic software extension is enabled in Java through a combination of dynamic loading and subtyping. The interchecking type rules in Section 3.5 (class preparation) ensure that dynamically loaded software extensions honor the confinement policies prescribed by their supertypes.

This section presents a case study that illustrates the above tactics.

5.1 Protection Through Import Type Assertions

Suppose an application class `Alice` is to share with `Bob` an instance of `Resource` that she owns. `Alice` does so by passing the `Resource` object as an argument to the method `share` exported by `Bob`, with the mutual understanding that `Bob` is not to leak this `Resource` reference outside of the defining package of `Alice`:

```
package domain;

confined class Resource { ...}

public class Alice {
  static Resource resource = new Resource();
  public static void main(String[] args) {
    Bob.share(resource);
  }
}
```

Suppose `Alice` cannot trust that `Bob` will uphold his end of the contract. To ensure that `Bob` does not leak the `Resource` reference accidentally or maliciously, the classfile of `Alice` can be annotated with a `ConfinedTypes` attribute that contains the following *import type assertion*:

$$\langle \texttt{Bob.share} \rangle : \bot(\texttt{confined})\bot$$

When the constant pool entry $\langle \texttt{Bob.share} \rangle$ is resolved in class `Alice`, interchecking will make sure that the resolved target has a matching export type assertion that promises to confine the argument reference. The Aegis VM will thus refuse to link `Alice` with any implementation of `Bob` that does not honor the import type assertion specified by `Alice`

5.2 Inspiring Trust by Export Type Assertions

Suppose the class `Bob` indeed provides a non-leaking implementation of the `share` method:

```
package domain;
public class Bob {
```

```
    static Resource resource;
    public static void share(Resource resource) {
      Bob.resource = resource;
    }
  }
```

To inspire trust, the classfile of `Bob` must be annotated properly. Specifically, the confined type interface of `Bob` must contain the following *export type assertion*:

$$\mathtt{share} : \bot(\mathtt{confined})\bot$$

When the class `Bob` is intrachecked, data flow analysis will be conducted on the body of the `share` method to ensure that it lives up to its promise.

To appreciate the robustness of trust inspiration, consider a version of `Bob` in which `share` exposes the `Resource` reference:

```
  package domain;
  public class Bob {
    public static Object leak;
    public static void share(Resource resource) {
      leak = resource;
    }
  }
```

The `share` method stores the `Resource` reference into a public field `leak`, thereby exposing the reference to access from outside of the `domain` package. Without further annotation, `Alice` will not link with `Bob` due to the incompatibility between the import type assertion of $\langle$`Bob.share`$\rangle$ in `Alice` and the export type assertion of `share` in the default confined type interface of `Bob`. Yet, `Bob` may *lie* by forging a confined type interface with an export type assertion that falsely claims that the `Resource` argument is `confined`:

$$\mathtt{share} : \bot(\mathtt{confined})\bot$$

When the Aegis VM performs intrachecking, the data flow analyzer will discover that reference leaking occurs in the body of `share`, and thus tag `Bob` as unsafe.

5.3 Secure Software Extension

Consider a more interesting example, in which the class `Alice` shares the `Resource` reference with a dynamically loaded software extension:

```
  package domain;
  public class Alice {
    Resource resource = new Resource();
    public static void main(String[] args) throws Throwable {
      Class C = Class.forName(args[0]);
      Bob b = (Bob) C.newInstance();
      b.share(resource)
    }
  }
```

In this example, `Bob` is an interface specifying the sharing protocol.

```
package domain;
public interface Bob {
  void share(Resource resource);
}
```

To protect `Alice`, the classfile of `Bob` is annotated to ensure that any implementation of the `share` method must confine the `Reference` argument. Specifically, `Bob` is endowed with the following export type assertion:

$$\texttt{share} : \bot(\texttt{confined})\bot$$

Suppose the class `Charlie` provides a non-compliant implementation of `Bob.share`:

```
package domain;
public class Charlie implements Bob {
  public static Resource leak;
  public void share(Resource resource) {
    leak = resource;
  }
}
```

If `Charlie` is not annotated, and thus assumes a default confined type interface, then the export type assertion of `share` will violate the method overriding rule scheduled to be checked at the time of class preparation (Section 3.5). Alternatively, if `Charlie` falsely claims the following export type assertion:

$$\texttt{share} : \bot(\texttt{confined})\bot$$

then intrachecking will detect the inconsistency. In both cases, this faulty implementation of `Charlie` will be rejected.

6 Concluding Remarks

6.1 Related Work

Language-based security. Language-based protection mechanisms employ programming languages technologies such as static analysis, program transformation, and type systems to address the security challenges of complex software systems [28]. Exemplary work includes software-based fault isolation [32], proof-carrying code [23], inlined reference monitors [30] and typed assembly languages [22]. Our long term research goal is to offer a language-based protection mechanism in which access control policies are defined via capability types. The present work is a helpful step in better understanding the structure of a capability type system for JVM bytecode.

Confined types. Confined types [31, 15, 34] is an example of alias control type systems [16, 21, 2, 25, 6, 4, 1, 3]. The common goal of these type systems is to control the proliferation of side effects due to aliasing in object-oriented programs. Confined types and the closely related ownership types [6] offer fresh interpretations of encapsulation, a notion with pertinence to language-based access control. Reformulating confined types to target JVM bytecode opens up a number of issues that are not addressed in the original source-level formulation: backward compatibility, modularity, and lazy dynamic linking. Our capability-based formulation turns confined types into a practical protection mechanism by addressing the above issues

Capability types. The idea of using typing disciplines to model capabilities has been around for some years (e.g., [4]). Existing capability type systems target source or hypothetical languages in a closed environment. A contribution of this paper is the formulation of a capability type system for JVM bytecode that takes into account the dynamic linking semantics of Java and the threat of untrusted software extensions.

Modular verification. The PVM framework [10] of the Aegis VM [9] is based on a modular verification architecture called Proof Linking [11]. The correctness of Proof Linking, especially its interaction with lazy dynamic linking, has been studied rigorously [12]. The correctness proof has been generalized to account for multiple classloaders [13]. The JAC type system [18] is another type system that has been implemented in the PVM framework [10]. The reformulation of confined types for the JVM bytecode is conceptually more involved than that of JAC.

6.2 Future Work

The present work is being extended in two directions. Firstly, the author is exploring a refinement of confined types called *discretionary object confinement* [14], in which confinement domains are stratified into a hierarchy of partially ordered trust levels. The boundary of a confinement domain is semi-permeable: objects can escape from one confinement domain to another if the trust relation permits it, or when the escape is explicitly granted by discretion. It turns out discretionary object confinement can be applied to build a minimalist capability system for the JVM. Secondly, the author is exploring a generic framework for defining capability type systems for JVM bytecode. The goal is to provide a meta language for programmers to specify their own capability type systems. The specification is then compiled into a link-time type checker that can be readily integrated into a JVM.

6.3 Conclusion

The first formulation of confined types for JVM bytecode is proposed to enable link time enforcement. The formulation is backward compatible, capability-based, modular, and interoperable with lazy dynamic linking. A prototype of the link-time type checker has been realized in the framework of Pluggable Verification Modules as a third-party verification service of the Aegis VM. The application of this formulation of confined types to achieve a form of secure cooperation has been illustrated in a case study.

A Confined Type Rules for JVM Bytecode Instructions

aaload

Operand Stack: $\ldots, a, i \quad \longrightarrow \quad \ldots, v$

Operation: Load the reference component v from array reference a at index i.

Type Constraints: If $a : T_a$ and $v : T_v$ then $T_a <: T_v$.

aastore

Operand Stack: $\ldots, a, i, v \quad \longrightarrow \quad \ldots$

Operation: Store reference value v into array reference a as the component at index i.

Type Constraints: If $a : T_a$ and $v : T_v$ then $T_v <: T_a$.

aconst_null

Operand Stack: $\ldots \quad \longrightarrow \quad \ldots, v$

Operation: Push a `null` reference v onto the operand stack.

Type Constraints: If $v : T_v$ then $\perp <: T_v$. That is, the `null` reference may assumes any type.

anewarray $\langle B \rangle$

Operand Stack: $\ldots, n \quad \longrightarrow \quad \ldots, a$

Operation: Create an array a of component type B with n components.

Type Constraints: Suppose $\langle B \rangle : T \in \mathcal{I}_A$ and $a : T_a$. Then $T <: T_a$.

areturn

Operand Stack: $\ldots, o \quad \longrightarrow$

Operation: Return object reference o from current method.

Type Constraints: Suppose $o : T_o$, and $m : T_0(T_1, \ldots, T_k)T \in \mathcal{E}_A$, where m is the current method. Then $T_o <: T$.

athrow

Operand Stack: $\ldots, o \quad \longrightarrow \quad o$

Operation: Throw o as an exception.

Type Constraints: If $o : T_o$ then $T_o <: \perp$. That is, $o : \perp$.

checkcast $\langle B \rangle$

Operand Stack: $\ldots, o \quad \longrightarrow \quad \ldots, v$

Operation: Attempt to cast object reference o to a reference v of type B.

Type Constraints: Suppose $\langle B \rangle : T \in \mathcal{I}_A$, and $o : T_o$ and and $v : T_v$. Then $T_o <: T$ and $T <: T_v$.

getfield $\langle B.f \rangle$

Operand Stack: $\ldots, o \quad \longrightarrow \quad \ldots, v$

Operation: Retrieve the value v of field $\langle B.f \rangle$ from object instance o.

Type Constraints: Suppose $\langle B.f \rangle : T \in \mathcal{I}_A$, and $v : T_v$. Then $T <: T_v$.

getstatic $\langle B.f \rangle$

Operand Stack: $\ldots \longrightarrow \ldots, v$

Operation: Retrieve value v from the static field $\langle B.f \rangle$.

Type Constraints: Suppose $\langle B.f \rangle : T \in \mathcal{I}_A$, and $v : T_v$. Then $T <: T_v$.

ldc/ldc_w ***⟨utf8-literal⟩***

Operand Stack: $\ldots \longrightarrow \ldots, v$

Operation: Push a `String` literal v onto the operand stack.

Type Constraints: If $v : T_v$ then $\perp <: T_v$. That is, the reference v may assume any type.

multianewarray $\langle B \rangle$**,** k

Operand Stack: $\ldots, n_1, n_2, \ldots, n_k \longrightarrow \ldots, a$

Operation: Create a multidimensional array a of array type B with space allocated for the first k dimensions.

Type Constraints: Suppose $\langle B \rangle : T_{\langle B \rangle} \in \mathcal{I}_A$. Then $T_{\langle B \rangle} <: T_a$.

new $\langle B \rangle$

Operand Stack: $\ldots \longrightarrow \ldots, v$

Operation: Create a new instance v of Java reference type B.

Type Constraints: Suppose $\langle B \rangle : T \in \mathcal{I}_A$ and $v : T_v$. Then $T <: T_v$.

putfield $\langle B.f \rangle$

Operand Stack: $\ldots, o, v \longrightarrow \ldots$

Operation: Store the value v into the field $B.f$ of object instance o.

Type Constraints: Suppose $\langle B.f \rangle : T \in \mathcal{I}_A$, and $v : T_v$. Then $T_v <: T$.

putstatic $\langle B.f \rangle$

Operand Stack: $\ldots, o, v \longrightarrow \ldots$

Operation: Store the value v into the static field $B.f$.

Type Constraints: Suppose $\langle B.f \rangle : T \in \mathcal{I}_A$, and $v : T_v$. Then $T_v <: T$.

invokeinterface $\langle B.m \rangle$

Operand Stack: $\ldots, o, a_1, a_2, \ldots, a_k \longrightarrow \ldots, v$

Operation: Invoke interface method $\langle B.m \rangle$ on object instance o, passing arguments a_1, a_2, …, a_k. Any return value v is pushed into the operand stack.

Type Constraints: Suppose $\langle B.m \rangle : T_0(T_1, T_2, \ldots, T_k)T \in \mathcal{I}_A$. Suppose further that $o : T_o$, $a_i : T_{a_i}$, and $v : T_v$. Then $T_o <: T_0$, $T_{a_i} <: T_i$, and $T <: T_v$.

invokespecial $\langle B.m \rangle$

Operand Stack: $\ldots, o, a_1, a_2, \ldots, a_k \longrightarrow \ldots, v$

Operation: Invoke private method, instance initializer, or superclass method $\langle B.m \rangle$ on object instance o, passing arguments a_1, a_2, …, a_k. Any return value v is pushed into the operand stack.

Type Constraints: Suppose $\langle B.m \rangle : T_0(T_1, T_2, \ldots, T_k)T \in \mathcal{I}_A$. Suppose further that $o : T_o$, $a_i : T_{a_i}$, and $v : T_v$. Then $T_o <: T_0$, $T_{a_i} <: T_i$, and $T <: T_v$.

invokestatic $\langle B.m \rangle$

Operand Stack: $\ldots, a_1, a_2, \ldots, a_k \longrightarrow \ldots, v$

Operation: Invoke static method $\langle B.m \rangle$ with arguments a_1, a_2, $\ldots$, a_k. Any return value v is pushed into the operand stack.

Type Constraints: Suppose $\langle B.m \rangle : \bot(T_1, T_2, \ldots, T_k)T \in \mathcal{I}_A$. Suppose further that $a_i : T_{a_i}$ and $v : T_v$. Then $T_{a_i} <: T_i$ and $T <: T_v$.

invokevirtual $\langle B.m \rangle$

Operand Stack: $\ldots, o, a_1, a_2, \ldots, a_k \longrightarrow \ldots, v$

Operation: Invoke method $\langle B.m \rangle$ on object instance o, passing arguments a_1, a_2, $\ldots$, a_k. Any return value v is pushed into the operand stack.

Type Constraints: Suppose $\langle B.m \rangle : T_0(T_1, T_2, \ldots, T_k)T \in \mathcal{I}_A$. Suppose further that $o : T_o$, $a_i : T_{a_i}$, and $v : T_v$. Then $T_o <: T_0$, $T_{a_i} <: T_i$, and $T <: T_v$.

References

[1] 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.

[2] 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.

[3] Adrian Birka and Michael D. Ernst. A practical type system and language for reference immutability. In *Proceedings of the 19th Annual ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'04)*, pages 35–49, Vancouver, B.C., Canada, October 2004.

[4] J. Boyland, J. Noble, and W. Retert. Capabilities for sharing: A generalization of uniqueness and read-only. In *Proc. of the 2001 Euro. Conf. on Object-Oriented Programming*, pages 2–27, Budapest, Hungary, July 2001.

[5] Luca Cardelli. Program fragments, linking, and modularization. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'97)*, pages 256–265, Paris, France, January 1997.

[6] 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.

[7] 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.

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

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

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

[11] Philip W. L. Fong. *Proof Linking: A Modular Verification Architecture for Mobile Code Systems*. PhD dissertation, School of Computing Science, Simon Fraser University, Burnaby, B.C., Canada, January 2004.

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

[13] Philip W. L. Fong and Robert D. Cameron. Proof linking: Distributed verification of Java classfiles in the presence of multiple classloaders. In *Proceedings of the USENIX Java Virtual Machine Research and Technology Symposium (JVM'01)*, pages 53–66, Monterey, California, USA, April 2001.

[14] Philip W. L. Fong and Boting Yang. Discretionary object confinement: A minimalist approach to capabilities for the JVM. Technical Report CS-2004-13, Department of Computer Science, University of Regina, Regina, SK, Canada, December 2004.

[15] Christian Grothoff, Jens Palsberg, and Jan Vitek. Encapsulating objects with confined types. In *Proceedings of the 16th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'01)*, pages 241–253, Tampa Bay, FL, USA, October 2001.

[16] John Hogg. Islands: Aliasing protection in object-oriented languages. In *Proc. of the ACM Conf. on Object-oriented Programming, Systems, Languages, and Applications*, pages 271–285, Phoenix, AZ, USA, Nov. 1991.

[17] Watts S. Humphrey. *Introduction to the Personal Software Process*. Addison Wesley, 1996.

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

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

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

[21] Naftaly Minsky. Towards alias-free pointers. In *Proceedings of the 10th European Conference for Objected Oriented Programming*, Linz, Austria, July 1996.

[22] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From system F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):527–568, May 1999.

[23] George C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages (POPL'97)*, pages 106–119, Paris, France, January 1997.

[24] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer, 1999.

[25] James Noble, Jan Vitek, and John Potter. Flexible alias protection. In *Proceedings of the 12th European Conference for Object-Oriented Programming*, pages 158–185, Brussels, Belgium, July 1998.

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

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

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

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

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

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

[32] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles (SOSP'93)*, pages 203–216, Asheville, North Carolina, USA, December 1993.

[33] Steve Wilson and Jeff Kesselman. *Java Platform Performance: Strategies and Tactics*. Addison Wesley, 2001.

[34] Tian Zhao, Jens Palsberg, and Jan Vitek. Lightweight confinement for featherweight Java. In *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'03)*, pages 135–148, Anaheim, California, USA, October 2003.