# Pluggable Verification Modules: An Extensible Protection Mechanism for the JVM

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

## ABSTRACT

Through the design and implementation of a JVM that supports *Pluggable Verification Modules (PVMs)*, the idea of an *extensible protection mechanism* is entertained. Link-time bytecode verification becomes a pluggable service that can be readily replaced, reconfigured and augmented. Application-specific verification services can be safely introduced into the dynamic linking process of the JVM. This feature is enabled by the adoption of a previously proposed modular verification architecture, Proof Linking [23, 24], which decouples bytecode verification from the dynamic linking process, rendering the verifier a replaceable module. The PVM mechanism has been implemented in an open source JVM, the Aegis VM [21]. To evaluate the software engineering and security engineering benefits of this extensible protection mechanism, an augmented type system JAC (Java Access Control) [37] has been successfully implemented as a PVM.

## Categories and Subject Descriptors

D.2.0 [**Software Engineering**]: General—*protection mechanisms*; D.3.4 [**Programming Languages**]: Processors—*run-time environments*; D.3.2 [**Programming Languages**]: Language Classification—*object-oriented languages, macro and assembly languages*; D.2.11 [**Software Engineering**]: Software Architectures—*domain-specific architectures*; D.4.6 [**Operating Systems**]: Security and Protection—*access controls*

## General Terms

Security, Languages, Verification

## Keywords

Pluggable verification modules, proof linking, extensible protection mechanism, Java virtual machine, bytecode verification, Aegis VM, mobile code security, extensible systems

## 1. INTRODUCTION

As our society becomes increasingly aware of the need for secure computing infrastructures, the programming language community has invested in recent years an unprecedented interest in the interplay between software security and programming language environments. One emerging challenge arises from the growing popularity of *Dynamically Extensible Software Systems*, such as mobile code language environments [11, 53], scriptable applications, and software systems with plug-in architectures [5, 19, 44]. In such systems, executable extensions can be dynamically linked into the address space of a host software system, either to deliver a short lived service, or to augment the capability of the underlying host in a permanent manner. If adopted unchecked, malicious software extensions could compromise the security of the host. An effective protection approach is to mandate the use of a *safe language* for programming software extensions. Strongly typed object-oriented programming languages, with support for enforcing data encapsulation on the one hand, and support for constructing extensible application framework on the other, becomes a natural linguistic choice. Such an approach forms the security foundation of the Java Virtual Machine (JVM) [39], an archetypical platform for constructing extensible systems. Software extensions are compiled into strongly typed intermediate code units called *classfiles*, which are in turn typechecked by a *bytecode verifier* at the time of dynamic loading. Since bytecode verification is an integral part of the classloading semantics, typechecking is therefore nonbypassable.

Future applications of the JVM will likely demand additional forms of verification to provide enhanced levels of protection. To address this need, the attention of scholarship has turned to safety properties that go beyond simple "type safety". These *application-specific* safety properties are captured in augmented type systems [12, 55, 7, 37, 2], annotation languages [33, 1], and other forms of static analyses. One critique of these works is that the notion of safety is often formulated as a compile-time property, enforced by the code producer, at the level of the source language. In the context of the JVM, in which code units bind via dynamic linking, program verification that is performed against source code, or administrated only by the code producer, cannot be trusted. Unfortunately, given the *inherent complexity* of Java's dynamic linking process, and its *tight coupling* with the bytecode verifier, programming alternative static analyses into the existing bytecode verification procedure is an extremely taxing and error-prone exercise.

This partly explains why it is rare to see the mentioned works materialize into link-time protection mechanisms for the JVM. This analysis reveals a fundamental design flaw in the protection architecture of the JVM, in which the link-time static verification service is a fixture that cannot be extended conveniently. As new security needs arise, there exists no mechanism whereby alternative static analyses can be retrofitted into the dynamic linking process of the runtime system. This problem is in no way specific to the JVM: the critique applies equally to other safe language environments such as the Common Language Infrastructure (CLI) [18].

Through the design and implementation of a JVM that supports *Pluggable Verification Modules (PVMs)*, the idea of an *extensible protection mechanism* is entertained. Link-time bytecode verification becomes a pluggable service that can be readily replaced, reconfigured and augmented. Application-specific verification services can be safely introduced into the dynamic linking process of the JVM. This feature is enabled by the novel application of a previously proposed modular verification architecture, Proof Linking [23, 24], which decouples bytecode verification from the dynamic linking process. The main contributions of this paper are the proposal of Pluggable Verification Modules as a standard feature for JVM-like extensible software systems, and the report of a concrete design and implementation of such a facility in an open source Java Virtual Machine, the Aegis VM [21]. To evaluate the utility of the extensible protection mechanism, an augmented type system JAC (Java Access Control) [37] has been implemented as a PVM. This implementation exercise illustrates two aspects of utility: Firstly, it demonstrates that incorporating an application-specific static analysis into the link-time verification procedure of the Aegis VM through the PVM extension mechanism requires only a tractable amount of programming effort. Secondly, it demonstrates how Proof Linking, as implemented in the PVM facility, can be exploited to enable secure cooperation [49] among mutually distrusting code units.

This paper is organized as follows. Section 2 analyzes the architectural deficiencies of existing JVMs, and articulates how the Proof Linking architecture can be novelly applied to turn link-time static verification into a pluggable service. The concrete design of the PVM facility as implemented in the Aegis VM is detailed in Sections 3 and 4. The utility of the extensible protection mechanism is evaluated in Sections 5 and 6, in which the implementation of the JAC type system as a PVM is used to illustrate the software engineering and security engineering benefits of the PVM facility. Discussion, related work, future work, and conclusion can be found in Section 7.

## 2. PLUGGABLE VERIFICATION SERVICE VIA PROOF LINKING

This section analyzes the verification architecture of existing JVMs, and explains why extensibility is not properly supported in such an architecture (Section 2.1). It then reviews the previously proposed Proof Linking architecture (Section 2.2), and subsequently outlines a novel application of this modular verification architecture to turn link-time static verification into a pluggable service (Section 2.3). Lastly, the implementation status of this extensible protection mechanism is reported (Section 2.4).

### 2.1 Problem Analysis

The lack of modularity in the verification architecture of existing JVMs is the main obstacle to rendering bytecode verification a pluggable service. To see this, note that 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 of the established properties with the runtime environment into which the classfile is linked. In the context of typechecking, the two tasks roughly correspond to the inference of a type interface for a code unit, and the checking of the compatibility between this type interface and a given type environment. Cardelli succinctly called the two tasks *intrachecking* and *interchecking* [10]. Unfortunately, the two tasks are not cleanly separated from each other and from the rest of the VM in a typical implementation of the bytecode verification procedure [39]:

1. **Interleaving of Intrachecking and Interchecking:** In the course of intrachecking a classfile, classloading may be initiated by the bytecode verifier in order to bring in the type interface of other classfiles for interchecking purposes. The result of mixing some interchecking into intrachecking is a tight coupling between the bytecode verifier and the dynamic linking logic of the runtime environment. Under this verification architecture, if an application-specific static analysis is to be introduced into the dynamic linking process of the JVM, not only will the intrachecking component have to possess intimate knowledge of the VM internal, any undisciplined classloading performed by the component for the sake of interchecking may also perturb the soundness of the dynamic linking semantics.

2. **Delocalization of Interchecking:** Because of the incremental nature of dynamic loading, not all code units of a program is present at run time. The bulk of interchecking, therefore, has to be performed incrementally as new classes are linked into the VM. Under this verification architecture, application-specific interchecking logic has to be carefully grafted at strategic locations of the dynamic linking procedure. Not only is this a nontrivial software engineering undertake, the process is inherently error prone.

Simply put, the verification architecture of a typical JVM implementation is not designed to support extensibility. Any extension mechanism for supporting application-specific verification must involve a complete rethinking of the verification architecture.

### 2.2 The Proof Linking Architecture

The Proof Linking architecture [23, 24] was originally proposed as a means to improve the comprehensibility and maintainability of the standard bytecode verifier in the JVM. This has been achieved by modularizing the JVM verification procedure. To understand the Proof Linking architecture, consider the example depicted in Figure 1. Suppose class $C$ defines a method $M(S)$, the body of which contains an *invokespecial* bytecode instruction that delegates the call to $A.M(S)$. For this instruction to be properly typed, JVM semantics require that $C$ must be a subclass of $A$. This requirement is an example of interchecking.

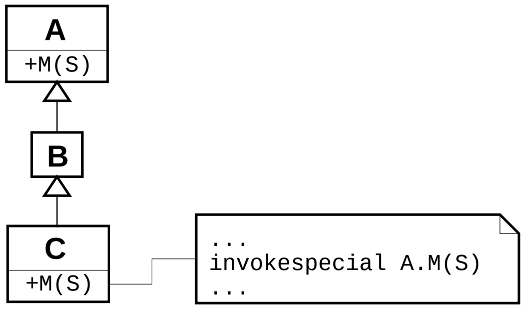

**Figure 1: A Typechecking Example**

Proof Linking cleanly separates intrachecking and interchecking. Intrachecking of classfiles is performed by a *modular verifier*, which infers for each classfile a *verification interface* composed of *proof obligations* and *commitments* (Figure 2). Commitments are assertions established by the modular verifier, while proof obligations are assumptions made during the process. Proof obligations and commitments are analogous respectively to the import and export parts of a module interface. For example, a modular verifier that implements intrachecking for the standard JVM type system will scan the body of $C.M(S)$, and, noticing the *invokespecial* instruction, generate a proof obligation `subclass(`$C$`,` $A$`)` to record the intermodular dependency. The modular verifier also generates the commitment `extends(`$C$`,` $B$`)` to indicate that class $C$ inherits directly from class $B$. Interchecking thus involves the discharging of proof obligations using commitments of loaded classes. Due to the incremental nature of dynamic loading and lazy, dynamic linking, a JVM program may not be completely loaded or linked, and thus not all proof obligations can be discharged right away. Consequently, also included in the verification interface is an *obligation discharging schedule*, which assigns to each proof obligation a *linking primitive* (i.e., a linking event), prescribing that the obligation should be discharged prior to the execution of the linking primitive. For example, the above proof obligation `subclass(`$C$`,` $A$`)` is scheduled to be discharged before the linking primitive "**resolve** $A.M(S)$ **in** $C$" is executed. Obligation discharging is therefore staged carefully to dovetail with the dynamic loading of classfiles and their commitments.

Incremental interchecking is achieved through a process called *proof linking* (Figure 3). Whenever a linking primitive is to be executed, the JVM attempts to discharge the associated proof obligations using commitments of classes that are already loaded. Execution of the linking primitive is only authorized if the check succeeds. In previous works [23, 24], incremental proof linking is modeled abstractly using *deductive database concepts* [40], whereby proof obligations are deductive queries, commitments are facts stored in a database, and logic programs are formulated to express interchecking logics such as type rules. For example, the following Horn clause expresses defines the `subclass/2` relation to be the reflexive transitive closure of `extends/2`.

```
subclass(X, X).
subclass(X, Y) :- extends(X, Z), subclass(Z, Y).
```

The obligation `subclass(`$C$`,` $A$`)` can be discharged if both the commitments `extends(`$C$`,` $B$`)` and `extends(`$B$`,` $A$`)` are already loaded before the linking primitive "**resolve** $A.M(S)$ **in** $C$" is executed. Correctness of the proof linking process is assessed through the examination of a *linking strategy*, which formally specifies the temporal dependencies between linking primitives as a partially ordered set. Three correctness conditions, namely, Safety, Monotonicity and Completion, have been established with the help of the PVS specification and verification system.

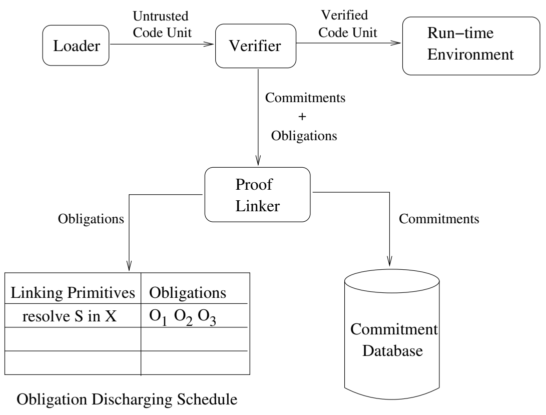

**Figure 2: Modular Verification**

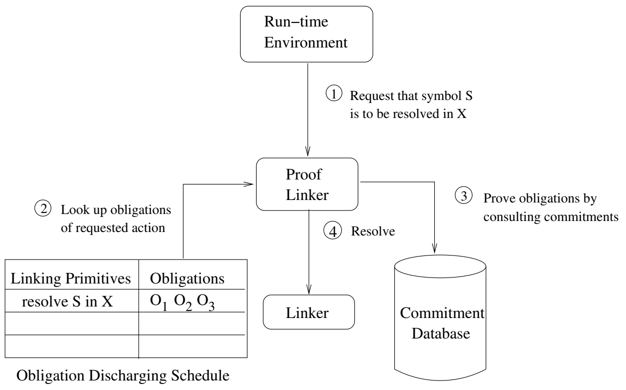
Run–time Environment
1 Request that symbol S is to be resolved in X
Proof Linker
2 Look up obligations of requested action
3 Prove obligations by consulting commitments
4 Resolve
Linking Primitives
Obligations
resolve S in X
O1 O2 O3
Linker
Commitment Database
Obligation Discharging Schedule

**Figure 3: Incremental Proof Linking**

## 2.3 A Solution Approach

One of the main contributions of this work is the novel employment of the Proof Linking architecture to create an *extensible protection mechanism* for the Aegis VM. Because Proof Linking cleanly separates intrachecking and interchecking through the use of a verification interface, intrachecking is decoupled from the dynamic linking process. No longer entangled with classloading logic, intrachecking code can be engineered as a separate component that interacts with the VM solely by generating verification interfaces. This architectural property is exploited to allow arbitrary *pluggable verification modules (PVMs)* to be dynamically linked into the JVM to augment the standard bytecode verifier. Every PVM implements an intrachecking procedure for an *application-specific verification domain*. Without having to perform any interchecking and classloading, each PVM infers a verification interface to capture intermodular dependencies. Details of the PVM facility and the design of a domain-independent representation of verification interface is described in Section 3.

A *generic proof linking mechanism* is built into the Aegis VM for supporting domain-independent interchecking. Specifically, proof obligations and commitments generated by PVMs are tracked by the Aegis VM, which discharges proof obligations according to the obligation discharging schedules embedded in verification interfaces. Consequently, the complexity of grafting interchecking code into the dynamic linking procedure is now replaced by a reusable proof linking engine. User-defined *verification domains* can be specified through the creation of *obligation libraries*, which are used by the generic proof linking mechanism for evaluating obligations specific to a verification domain. Details of the generic proof linking mechanism and obligation libraries can be found in Section 4.

The PVM extension mechanism offers a number of *software engineering benefits* to developers of an application-specific verification technology. Firstly, intrachecking code no longer needs to assume implementation-level knowledge of the VM internal. Instead, a PVM depends only on an abstract model of proof linking over which verification interfaces are defined. Secondly, interchecking is now a fully automated service provided by the generic proof linking mechanism, thereby freeing the developers from having to manually graft interchecking code into the dynamic linking procedure. Thirdly, Safety and Completion are guaranteed for free. The PVM extension facility is designed in such a way that user-defined PVMs and obligation libraries automatically satisfy the two aforementioned correctness conditions of Proof Linking. Developing well-mannered link-time static analyses under the PVM framework is therefore highly tractable. A detail demonstration of these software engineering benefits in the context of the JAC type system can be found in Section 5.

The PVM extension mechanism also offers a number of *security engineering benefits* to the administrators and users of a JVM platform. Firstly, the administrator is free to mix and match PVMs to create a custom-made JVM platform that addresses the specific security needs of an application area (e.g., embedded VM for mobile phones). When a security flaw is discovered in a PVM, it can be replaced as an individual component. Configuration management of the verification service is therefore streamlined. Secondly, application-specific access control policies can be easily expressed in terms of proof obligations and commitments. Enabled by the generic proof linking mechanism, safe dynamic linking allows an extensible application to erect protection boundaries between the application core and dynamically loaded extensions. An illustration of this latter security engineering benefit in the context of the JAC type system can be found in Section 6.

### 2.4 Implementation Status

The PVM facility and the generic proof linking mechanism have been fully implemented in the Aegis VM. The implementation effort has been administrated as an open source project [21]. Six development releases result in a VM that supports features including dynamic linking, access control, delegation style classloading, loading constraints, reflection, garbage collection, native method dispatching and all aspects of bytecode interpretation. The VM does not yet support multithreading. The Aegis VM currently runs on the GNU/Linux (x86) platform. It provides a realistic platform on which to test the feasibility of the Proof Linking architecture. Features described in this paper has been incorporated into release 0.2.0. Design highlights are outlined in Sections 3 and 4 to illustrate the utility of the approach. Consult [22] for low-level implementation details.

## 3. PLUGGABLE VERIFICATION MODULES

PVMs are dynamically loadable shared libraries on the GNU/Linux platform. Programmers may implement an application-specific analysis as a PVM, and subsequently use it to augment the Aegis VM through the PVM extension mechanism. This section outlines the design of this plug-in mechanism.

### 3.1 PVM Life Cycle

The Aegis VM can be configured with an arbitrary number of PVMs. The PVMs are loaded when the VM bootstraps. Every PVM exports an identifier specifying the *verification domain* to which it belongs. The Aegis VM uses this identifier to match the PVM with a corresponding obligation library, with which the verification interfaces generated by this PVM will be interpreted.

Next, the *initialization function* exported by each PVM is invoked. From this point on, the verification facilities of the loaded PVMs will be called into service whenever a class is to be defined. Specifically, prior to the definition of a class, the corresponding classfile representation is parsed into an abstract syntax tree (AST). A built-in dataflow analyzer is then employed to typecheck the bytecode methods in the AST. The results of dataflow analyses are passed along with the AST into the *verification function* of every loaded PVM, whereby application-specific intrachecking is conducted (Section 3.2). Successful intrachecking generates a *verification interface* (Section 3.3), which is then processed by the generic proof linking mechanism. Class definition is authorized only if all PVMs endorse the safety of the corresponding classfile representation.

When the Aegis VM shuts down, the *clean-up function* exported by each PVM is invoked before the PVM is unloaded.

### 3.2 Verification Function

The verification function implements the core functionality of a PVM. To reduce the overhead of user-defined intrachecking, and to facilitate PVM development, the AST of the target classfile and the results of typechecking bytecode methods are passed as arguments to the verification function. Specifically, the built-in dataflow analyzer of the Aegis VM generates (1) an explicit intraprocedural control flow graph for each bytecode method (control flow is implicit due to the presence of the notorious subroutine construct in the JVM bytecode language [52, 45]), and (2) a *type state* for each program point. Each type state describes (i) the depth of the operand stack, (ii) the type of each data item residing in the operand stack and local variable array, and (iii) the subroutine call chain leading to the program point. A verification function may reuse the above information to avoid analyzing a classfile from scratch. Experience from implementing the JAC type system has confirmed that the availability of this additional information greatly facilitates the development of PVMs.

```
obligation ::= predicate-identifier { symbolic-argument }*
symbolic-argument ::= this
                    | super
                    | interface index
                    | field index
                    | method index
                    | literal index
                    | import-symbol index
                    | auxiliary-symbol index
                    | global-class index
                    | global-constant index
```

**Figure 4: Abstract Syntax of Proof Obligations**

## 3.3 Verification Interface

The verification function constructs a verification interface for each classfile passing intrachecking. Every verification interface is composed of three components — proof obligations (3.3.1), commitments (3.3.2), and an obligation discharging schedule (3.3.3). The Aegis VM stores verification interfaces in class definitions. Unloading of a class automatically cleans up the memory resources occupied by the associated verification interfaces.

### 3.3.1 Proof Obligations

The verification function formulates proof obligations to capture external dependencies of a target classfile. Every proof obligation represents a condition to be checked at link time by the generic proof linking mechanism. The PVM facility provides an expressive, symbolic representation for verification functions to encode proof obligations. Under this representation, every proof obligation is a ground query composed of a *predicate identifier* and zero or more *symbolic arguments*. The abstract syntax of a proof obligation is given in Figure 4.

Every verification domain defines a fixed number of predicate symbols (Section 4.1). Each predicate symbol represents a boolean function that checks if some condition holds. The predicate identifier of an obligation is a numeric index specifying a predicate symbol in the verification domain to which the PVM belongs. Intuitively, the boolean function corresponding to the predicate symbol will be applied to the symbolic arguments when the obligation is discharged.

The verification function may name various components of the class being verified as the arguments of an obligation. Specifically, the `this`, `super`, `interface`, `field` and `method` argument syntax are used for naming the target class, its immediate superclass, immediate superinterfaces, declared fields and declared methods respectively. The *index* field identifies the specific candidate of a given argument type.

A `literal` argument names an `int`, `float`, `long`, `double`, or UTF-8 string literal in the constant pool. The *index* field refers to the index of the literal in the constant pool.

An `import-symbol` argument names the *resolved* target of a class, field, method, or interface method reference in the constant pool. The *index* field refers to the index of the symbolic reference in the constant pool.

Occasionally, the verification function may need to name an argument that corresponds to a class that cannot be referenced using the above argument syntax. Such a class reference is said to be an *auxiliary symbol*. Every verification interface provides its own symbol table for storing auxiliary class symbols. The `auxiliary-symbol` argument syntax identifies an entry of this auxiliary symbol table. When a class is prepared, all the class symbols mentioned in the auxiliary symbol table are loaded and cached, so that they can be readily retrieved at the time of obligation discharging (Section 4.4).

The obligation argument types described so far refer to run-time JVM data structures that are defined independently of any specific verification domain. Obligation argument types `global-class` and `global-constant` provide syntax for naming data structures specific to a verification domain. Specifically, every verification domain may identify a fixed number of Java classes to be instrumental to verification (e.g., the class *java.lang.Throwable* is needed for checking if a class can be thrown as an exception). These classes can be named as obligation arguments through the `global-class` syntax. Every verification domain may also define a fixed number of immutable, native data structure to represent domain constants. Such constants can be named using the `global-constant` syntax. See Section 4.1 for more details.

As the Aegis VM has to explicitly track proof obligations, a compact obligation encoding has been derived [22], whereby an obligation with $k$ arguments can be encoded with $k + 1$ 32-bit machine words.

### 3.3.2 Commitments

Static properties successfully established by the verification function for a target classfile are captured in commitments. To maximize optimization opportunities, the Aegis VM does not mandate a particular representation for commitments. Any appropriate data structures can be employed by a verification function to represent commitments for the verification domain of the PVM.

### 3.3.3 Obligation Discharging Schedule

Because of lazy, dynamic linking, obligation discharging proceeds in an incremental manner. Every proof obligation formulated by a verification function is explicitly scheduled to be discharged when a specific linking primitive is executed. Such an obligation is said to be *attached* to the target linking primitive. Specifically, the following family of linking primitives are defined for every class $C$:

**endorse** $C$**:** Endorsement of a class $C$ occurs when $C$ is *prepared* [39, Section 5.4.2].

**endorse** $C.F$**:** Endorsement of a field $C.F$ occurs prior to the first access of $F$, and after the endorsement of class $C$.

**endorse** $C.M$**:** Endorsement of a method $C.M$ occurs prior to the first invocation of $M$, and after the endorsement of class $C$.

**resolve** $S$ **in** $C$**:** This primitive coincides with the resolution of constant pool reference $S$. It occurs after the endorsement of class $C$, and after the endorsement of the referent of $S$.

In practice, proof obligations that require the checking of $C$ against the commitments of its supertypes or auxiliary symbols are attached to endorsement primitives, while those that validate import symbols are attached to resolution primitives.

The design of the PVM facility is intentionally constrained, so that a verification function may only attach proof obligations to "**endorse** $C$", "**endorse** $C.F$", "**endorse** $C.M$", and "**resolve** $S$ **in** $C$" when the classfile representation of class $C$ is intrachecked. This constraint guarantees that the correctness condition Safety is satisfied [23]. Specifically, the design guarantees that, when a linking primitive is executed, all proof obligations that may be attached to it are already generated.

## 4. DOMAIN-INDEPENDENT PROOF LINKING

A generic proof linking mechanism has been implemented so that Aegis VM can process the proof obligations and commitments generated by PVMs. At the heart of this facility is a mechanism that allows users to define new verification domain.

### 4.1 Obligation Libraries

Users may define an application-specific verification domain by developing an *obligation library*. As a dynamically loadable shared library on GNU/Linux, each obligation library supplies the definitions for predicate symbols, global classes and global constants of a verification domain. An obligation library developer has to program the following:

**Predicate functions.** A native boolean function is defined for each predicate symbol in the verification domain[1]. This function will be dispatched when a corresponding obligation is to be discharged. A *predicate dispatching table*, analogous to a virtual function table, must be exported by an obligation library.

**Global classnames.** An array of classnames must be exported to specify the names of global classes in the verification domain. These classes will be automatically loaded by the bootstrap classloader when the Aegis VM bootstraps.

**Global constants.** An array of native data structures must be exported as global constants for the verification domain.

### 4.2 Obligation Library API

To facilitate the evaluation of obligations, the Aegis VM provides an *obligation library API*, whereby native predicate functions can examine the run-time state of the VM and look up commitments. A brief summary of the API facilities is given in Figure 5. The obligation library API is carefully designed so that native predicate functions only have access to type interfaces and commitments of classes that are already loaded. This guarantees that the correctness condition Completion is satisfied [23].

[1]The formulation of the native predicate functions must correspond to a monotonic logic in order for the correctness condition Monotonicity to be satisfied [23]. It is the responsibility of the obligation library developer to take care of this requirement.

**Package interface interrogation.** Examine the name and classloader of a loaded package.

**Class interface interrogation.** Examine the access control flags, package, classloader, name, superclass, superinterfaces, declared fields, declared methods, and constant pool entries of a loaded class.

**Field interface interrogation.** Examine the access control flags, declaring class, name, and type signature of a field.

**Method interface interrogation.** Examine the access control flags, declaring class, name, type signature, and exception class names of a method.

**Subtyping tests.** Subclassing, subinterfacing, subtyping, etc.

**Contextual information.** Retrieve commitments of a class for the current verification domain; access global classes and global constants of the current verification domain.

**Figure 5: Facilities in the Obligation Library API**

### 4.3 Life Cycle of an Obligation Library

The Aegis VM can be configured with an arbitrary number of obligation libraries, thereby equipping the VM with vocabularies for multiple verification domains. When the Aegis VM starts up, all the configured obligation libraries are loaded and initialized. The global classes specified by each obligation library are loaded by the bootstrap classloader when the Aegis VM bootstraps. After this point, the native predicate functions are made available to the generic proof linking mechanism for obligation discharging. Obligation libraries are properly cleaned up and unloaded before the Aegis VM shuts down.

### 4.4 Obligation Discharging Sequence

Whenever a linking primitive is executed, the Aegis VM attempts to discharge all the attached proof obligations. To discharge a proof obligation, the following steps are followed:

1. The predicate identifier is used as an index to look up the corresponding native predicate function in the predicate dispatching table of the current verification domain. This is a constant-time operation

2. Each symbolic argument of the proof obligation is resolved into a corresponding pointer to a VM data structure or a global constant of the current verification domain. As argument resolution amounts to a constant-time look up operation, construction of the argument array takes time linear to the number of arguments involved.

3. The native predicate function is invoked with the argument array as input. A boolean value is returned to indicate the result of evaluation.

Execution of the linking primitive is authorized only when all attached proof obligations are successfully discharged.

This concludes the discussion of the PVM facility and the generic proof linking mechanism. We now turn to the assessment of their utility.

## 5. SOFTWARE ENGINEERING BENEFITS

This section assesses the utility of the PVM facility in supporting the development of link-time static verification procedures for the JVM. Due to its simplicity and its relevance to access control, the augmented type system JAC [37] is chosen as an example verification domain to be implemented on top of the PVM facility. Specifically, the JAC type system, which is originally designed for typing Java source programs at compile time (Section 5.1), is recast here as a type system for the JVM bytecode language, enforceable at link time (Section 5.2). An annotation scheme is designed for embedding JAC type interfaces into Java classfiles (Section 5.3). An obligation library is implemented for the JAC verification domain (Section 5.4). A PVM for the JAC type system is described (Section 5.5). Through the recounting of this implementation exercise, the reader may gain a better appreciation of the development effort involved in the introduction of a non-standard static analysis into the dynamic linking process of the JVM through the PVM facility. This in turn provides the basis for a fair assessment of the software engineering benefits of the PVM facility (Section 5.6).

### 5.1 JAC for the Java Source Language

JAC (Java with Access Control) [37] was proposed as an augmented type system for controlling the proliferation of side effects due to alias creation in object-oriented programs. Rather than preventing the creation of aliases, JAC prevents undesirable side effects from occurring when aliasing is unavoidable. Specifically, it allows a Java reference type to be qualified as being `readonly`, which effectively protects the transitive state of the reference from any write access. Unlike the C type qualifier `const`, which only protects the state of the object directly accessible from a `const`-qualified reference/pointer, the write protection of JAC extends to all objects reachable from a `readonly`-qualified reference in the underlying object graph.

To understand how transitive write protection works, consider the following Java linked-list class.

```
public class List {
  public int data;
  public List next;
  public List(int data, List next) {
    this.data = data; this.next = next;
  }
}
```

Notice that the instance variables of `List` are `public`, and as such they can be freely modified by client code. However, a `List` variable qualified as `readonly` cannot be used for modifying the transitive state reachable from the variable.

```
readonly List x = new List(1, new List(2, null));
x.data = 5;      // Error: Writing to immediate state
x.next.data = 6; // Error: Writing to transitive state
```

Objects reachable from a `readonly` reference are `readonly`. Furthermore, unqualified reference types can be converted to `readonly` ones, but not vice versa.

The original JAC type system is designed for typing Java source programs at compile time. As discussed above, code units that are checked to be type safe in the compilation environment may no longer be type safe when they are linked against those code units in the run-time environment. For access control type systems to become a viable protection mechanism for mobile code systems, they must be enforced at link time. To this end, the JAC type system is recast in this section as a type system for the JVM bytecode language. Furthermore, this bytecode incarnation of JAC differs from Kniesel and Theisen's work [37] in the following ways:

1. The original JAC syntax forces the return type of an instance method to share the same type qualifier with the type of `this`, the object instance to which method invocation is targeted. This restriction is purely a source-level syntactic constraint, and will not be enforced here. The bytecode version of JAC presented below is fully capable of qualifying the two types independently.

2. The original JAC type system has a `mutable` type qualifier for decorating instance variables, thereby selectively shielding the qualified fields from the transitive effect of write protection. The `mutable` qualifier is not modeled in this work. Extending the current work to include the `mutable` qualifier is a straightforward exercise.

3. The original work in [37] describes an extension of JAC to accommodate the generic type system of GJ [9], a variant of Java that supports genericity. This extension is beyond the scope of this paper.

### 5.2 JAC for the JVM Bytecode Language

The JAC type system is recast here as a type system for the JVM bytecode language. Similar to its source language counterpart, the bytecode version of the JAC type system defines two types, namely, `readonly` and $\bot$. The bottom type $\bot$ applies to *both* mutable object references *and* primitive values (i.e., `int`, `boolean`, etc). The `readonly` type applies to object references for which transitive states are protected.

#### 5.2.1 Subtyping

The bottom type $\bot$ is a subtype of `readonly`, and as such the conversion of $\bot$ to `readonly` is permitted. We write $A <: B$ if type $A$ is equivalent to or a subtype of $B$. Method subtyping follows the usual contravariant rule: $A \to B <: A' \to B'$ if $A' <: A$ and $B <: B'$.

#### 5.2.2 Type Interface

Associated with each Java classfile is a JAC type interface, which consists of an *export* part and an *import* part. Each part is a list of type assertions, relating symbols to their types. The export part describes type assertions for fields and methods declared in the classfile. The import part contains type assertions for field, method and interface method references in the constant pool. The type assertion of a field simply assigns a JAC type to the field. The type assertion of a method assigns a JAC type to the return value and to each formal parameter, including `this` in the case of an instance method. A type assertion is well-formed if primitive types in the standard Java type system are qualified by the $\bot$ type.

### 5.2.3 Interchecking

Subclassing is safe only if method overriding honors the usual subtyping rule. That is, if method $C.M : T$ overrides method $C'.M : T'$, then $T <: T'$. A similar requirement applies to subinterfacing. This check can be performed when the class endorsement primitive is executed.

Resolution of a constant pool method reference (or interface method reference) $C.M$ with import type assertion $C.M : T$ is type safe if the resolved target $C'.M$ is defined in a classfile that exports type assertion $C'.M : T'$ and $T' <: T$. Resolution of a constant pool field reference with import type assertion $C.F : T$ is type safe if the resolved target has an export type assertion $C'.F : T$. Notice that the typing requirements are different in the two cases.

### 5.2.4 Intrachecking

The export type assertion of a bytecode method is valid if every program point in the method body can be consistently assigned a *JAC type state*. A JAC type state is an assignment of a JAC type to every location in the local variable array and the operand stack. Every bytecode instruction imposes typing constraints on the JAC type states at the program points before and after the instruction. The typing constraints for a sample of bytecode instructions are presented below. A more complete list can be found in [22]. The effect of a bytecode instruction is presented in a notation popularized by [39]. For example, the *iadd* instruction pops two integers from the top of the operand stack, and push their sum back. 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 $i_2$.

***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:** Neither $a$ nor $v$ is `readonly`.

***getfield*** $\langle$***fieldref***$\rangle$

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

**Operation:** Load the value $v$ of the instance variable $\langle$*fieldref*$\rangle$ from object instance $o$.

**Type Constraints:** If the constant pool entry $\langle$*fieldref*$\rangle$ is a reference field with a `readonly` import type, then $v$ must be `readonly`. If $\langle$*fieldref*$\rangle$ is a reference field, and $o$ has a `readonly` type, then $v$ must be `readonly`. Otherwise, $v$ must be $\perp$.

***putfield*** $\langle$***fieldref***$\rangle$

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

**Operation** : Store the value $v$ into the instance variable $\langle$*fieldref*$\rangle$ of object instance $o$.

**Type Constraints:** The type of $o$ must not be `readonly`. If the constant pool entry $\langle$*fieldref*$\rangle$ is a reference field with an import type $\perp$, then $v$ must not have a `readonly` type.

***invokevirtual*** $\langle$***methodref***$\rangle$

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

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

**Type Constraints:** Let the type of $o, a_1, \ldots, a_k$ and $v$ be $A_0, A_1, \ldots, A_k$ and $A$, and the import type of the constant pool entry $\langle$*methodref*$\rangle$ be $\langle B_0, B_1, \ldots, B_k\rangle \rightarrow B$. Then $A_i <: B_i$ for $0 \leq i \leq k$, and $B <: A$.

## 5.3 Embedding JAC Type Interfaces

To make JAC enforceable at link time, every classfile must carry a JAC type interface. A compact encoding has been designed [22] for embedding JAC type interfaces into classfiles through the classfile attribute facility [39, Section 4.7].

A well-formed JAC attribute assigns no more than one type to an export symbol or an import reference. It is however, not necessary for all symbols to receive a type assignment. The symbols left untyped are said to have *default types.* In fact, a classfile may not even have a JAC attribute. In such a case, all import references and export symbols are assumed to have default types. The default type of a field is $\perp$; the default type of a method is such that the return value and all formal parameters have type $\perp$. This definition of default types is consistent with the interpretation of the standard Java type system in the context of JAC. The provision of assuming a default type interface for a classfile not carrying a JAC attribute renders it possible to reuse legacy classfiles not compiled for JAC typechecking, so long as client classes do not pass read-only references to the legacy classes. This is particularly handy in the case of the standard Java class library — hundreds of system classes can be reused as is. This simple choice of default type makes the experiments described in Section 6 tractable[2].

A command line utility was developed to facilitate the injection of JAC attributes into classfiles. The program takes a classfile and a high level JAC type interface specification as input, and generate a version of the input classfile with the corresponding JAC attribute embedded[3].

## 5.4 Obligation Library for JAC

An obligation library has been implemented for the JAC verification domain. The JAC obligation library exports the following predicates:

1. *Import safety predicates*:

   `safe-field-import` *field import-type*
   `safe-method-import` *method import-type*

   where *field* is a field, *method* a method, and *import-type* a UTF-8 literal representing an import type signature. The predicates checks if the export type of

[2]A more sophisticated design is to extend this version of JAC to incorporate polymorphism, and to automatically generate a principal type interface for each library class using a type inference algorithm. Not needed by the experiments in this paper, such a feature is left for future work.

[3]Alternatively, the specification and embedding of JAC type annotations can be performed in the framework of the new JDK 5.0 metadata facility [6].

*field*/*method* is compatible with *import-type*. Implementation of the two predicate functions involves the invocation of obligation library API functions to retrieve commitment data structures.

2. *Method overriding safety predicate*:

   `safe-method-override` *class*

   For each of the method declared in *class*, check that its export type is a subtype of the export type of every method it overrides. Implementation of this predicate function involves applying obligation library API functions to visit all superclasses and superinterfaces of *class*, and to retrieve their commitment data structures.

The obligation library also exports a global constant for representing default types. No global class is specified for the JAC verification domain.

## 5.5 PVM for JAC

A PVM has been implemented for JAC. When the verification function of the JAC PVM is invoked on a classfile, it performs the following verification steps:

1. **Parsing the type interface into a commitment data structure.** If the classfile carries a JAC attribute, then the embedded JAC type interface is checked for well-formedness. Otherwise, a default JAC type interface is assumed. In either case, a JAC-specific commitment data structure is generated to store the JAC type interface.

2. **Intrachecking bytecode methods.** An iterative dataflow analysis algorithm is applied to verify that the bytecode methods satisfy the type constraints specified in Section 5.2.4. Notice that, if all import references have default types, then there is no need to run the dataflow analysis on a method with default export type. In such a case, the analysis merely repeats what has already been checked by the built-in bytecode verification procedure. Consequently, intrachecking can be safely skipped for classfiles with no JAC attribute.

3. **Generating obligation attachments.** Firstly, a corresponding import safety obligation is attached to the resolution primitive of each import reference in the constant pool. For example, an obligation of the following form will be generated for a field reference in the constant pool:

   `safe-field-import` `import-symbol` $i$ `literal` $j$

   where $i$ is the constant pool index of the field reference, and $j$ is the constant pool index of the UTF-8 string storing the import type. If the import type of the field reference is not explicitly specified in the JAC attribute, then the reference has a default type, and the following obligation should be generated instead:

   `safe-field-import` `import-symbol` $i$
   `global-constant` 0

   where "`global-constant` 0" denotes the global constant representing default types. The formulation of import safety obligations for method references is similar.

   Notice that obligation attachments should still be generated for an import reference even if it has default type, since the export type of the reference target may not be compatible with the default import type. In other words, although intraprocedural typechecking may be optimized away in special cases (see step 2 for details), interprocedural typechecking must never be bypassed.

   Secondly, a method overriding safety obligation is attached to the class endorsement primitive of the target class.

   `safe-method-override this`

4. **Formulating verification interface.** A verification interface composed of the commitment data structure from step 1 and the proof obligations from step 3 is constructed. No auxiliary symbol is needed in this verification domain.

| PVM | Obl. Lib. | Shared | Total |
|---|---|---|---|
| 2082 | 490 | 75 | 2647 |

**Figure 6: LOC for the JAC Implementation**

## 5.6 Assessment

**Reduction of Code Complexity.** Figure 6 gives an estimate of the complexity of the JAC implementation described in Sections 5.4 and 5.5. Specifically, the number of lines of (moderately commented) C code for the JAC PVM, the corresponding obligation library, and common data structures shared between the two are displayed. Such a modest complexity is achieved because the Aegis VM provides reusable facilities to ease the development of verification services. These include, for example, construction of AST and type analysis results, API for interrogation of VM state, utilities for formulating and managing commitments and obligations, and, most importantly, automated interchecking through proof linking. In short, the Aegis VM provides a reusable development platform for developers of link-time verification services.

**Reduction of Cognitive Stain.** Notice from Section 5.5 that a PVM developer works with an abstract proof linking model rather than concrete details of the Aegis VM internal. Not only that, Safety and Completion are granted for free: violations of the two correctness conditions cannot even be expressed in a PVM. Such provisions make the construction of well-mannered verification services a cognitively tractable task.

## 6. SECURITY ENGINEERING BENEFITS

Using the JAC type system as an example, this section illustrates that features built into the PVM facility, including user-defined intrachecking, and safe dynamic linking through proof linking, can be exploited to impose access control constraints among mutually distrusting code units within the same application.

## 6.1 Protection Through Import Type Annotations

Suppose an application class `Alice` needs to compute the sum of all integers in a `List` it creates. The task is delegated to another class `Bob`, which provides a `sum` method that computes the sum of all elements in a given `List`.

```
public class Alice {
  public static void main(String[] args) {
    List L =
      new List(1, new List(2, new List(3, null)));
    System.out.println(Bob.sum(L));
  }
}
```

Suppose `Alice` cannot trust that `Bob` is side-effect free. To ensure `Bob` does not accidentally or maliciously modify the values stored in the `List` argument, the classfile of `Alice` can be annotated with a JAC attribute containing the following *import* type assertion.

$$\texttt{Bob.sum} : \texttt{readonly} \rightarrow \bot$$

When the `Alice` class is defined at load time, the JAC PVM will attach a `safe-method-import` obligation to the resolution primitive of the import reference `Bob.sum`. The generic proof linking mechanism will thus refuse to link `Alice` with any implementation of `Bob` that does not honor this import type specification. Consequently, the transitive state of the `List` reference passed into `sum` will be write protected.

## 6.2 Inspiring Trust by Asserting Export Types

Suppose the class `Bob` indeed provides a side-effect free implementation of the `sum` method.

```
public class Bob {
  public static int sum(List L) {
    int acc = 0;
    while (L != null) {
      acc += L.data;
      L = L.next;
    }
    return acc;
  }
}
```

To inspire trust, the classfile of `Bob` will need to be annotated properly. Specifically, the following *export* type assertion is embedded into the classfile of `Bob`.

$$\texttt{Bob.sum} : \texttt{readonly} \rightarrow \bot$$

When the class `Bob` is defined at load time, the verification function of the JAC PVM will be invoked, and dataflow analysis is conducted on the body of the `Bob.sum` method so as to ensure that the implementation indeed lives up to its promise. In this case, the JAC PVM successfully verifies the export type assertion of the method, and class definition is therefore granted. Next, when the import reference `Bob.sum` is resolved in `Alice`, the proof obligation `safe-method-import` will be discharged to make sure that the export type of `sum` in Bob is compatible with its corresponding import type in `Alice`. Again, the check will succeed, and resolution will be granted.

To appreciate the robustness of trust inspiration, consider a version of `Bob` in which the `sum` method silently corrupts the `List` argument.

```
public class Bob {
  public static int sum(readonly List L) {
    int acc = 0;
    while (L != null) {
      acc += L.data;
      if (L.next == null)  // corrupt last node
        L.data = 0;
      L = L.next;
    }
    return acc;
  }
}
```

The `sum` method perturbs the integer datum stored in the last node of the `List` argument, corrupting its transitive state. Without further annotation, `Alice` will not link with `Bob` due to the incompatibility between the default export type of `Bob.sum` and its expected import type in `Alice`. Yet, the classfile of `Bob` could be annotated with a JAC attribute that falsely claims that the `sum` method is side-effect free.

$$\texttt{Bob.sum} : \texttt{readonly} \rightarrow \bot$$

When the Aegis VM attempts to verify this version of `Bob` with the JAC PVM, the dataflow analyzer will fail to confirm the consistency of the export type assertion, and class definition will fail. In either case, write protection is guaranteed.

## 6.3 Secure Software Extension

Consider a more realistic example, in which the class `Alice` dynamically loads a user-specified extension to carry out the summation operation.

```
public class Alice {
  public static void main(String[] args)
    throws Throwable {
    List L = new List(1, new List(2,
                          new List(3, null)));
    Class C = Class.forName(args[0]);
    Bob b = (Bob) C.newInstance();
    System.out.println(b.sum(L));
  }
}
```

In this example, `Bob` is defined as an interface specifying the invocation convention of the summation service.

```
public interface Bob {
  int sum(List L);
}
```

To protect `Alice`, the classfile of `Bob` is annotated to ensure that any implementation of the `sum` service must treat the `List` argument as `readonly`. Specifically, `Bob.sum` has the following export type in `Bob`.

$$\texttt{Bob.sum} : \texttt{readonly} \rightarrow \bot$$

Notice, however, that there is no need to annotate `Alice`, because a default import type for `Bob.sum` is assumed.

$$\texttt{Bob.sum} : \bot \rightarrow \bot$$

When the interface method reference `Bob.sum` is resolved in `Alice`, the corresponding `safe-method-import` obligation will be discharge successfully since the export type of the resolved target (`readonly` → ⊥) is a subtype of the default import type (⊥ → ⊥).

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

```
public class Charlie implements Bob {
  public int sum(List L) {
    int acc = 0;
    while (L != null) {
      if (L.next == null)  // corrupt last node
        L.data = 0;
      acc += L.data;
      L = L.next;
    }
    return acc;
  }
}
```

If `Charlie` is not annotated, then the default export type of `Charlie.sum` will violate the subtyping constraint required for type safe method overriding. The obligation `safe-method-override` will thus fail to discharge when `Charlie` is prepared. Alternatively, if `Charlie` falsely exports the following type assertion

`Charlie.sum : readonly` → ⊥

then the JAC PVM will detect the inconsistency when the `Charlie` class is defined. In both cases, this faulty implementation of `Bob` will be rejected.

## 6.4 Assessment

A fundamental security challenge in dynamically extensible software systems has been the facilitation of secure cooperation among mutually distrusting code units within the same application. This problem and its variations have been known by many names: the problem of mutual suspicion [50] safe invocation [49], confused deputy [30], layered protection [22], etc. Although the examples in this section are specific to the JAC type system, they do illustrate how the PVM facility and the generic proof linking mechanism may serve as an *enabling technology* by which application-specific solutions for secure cooperation can be implemented:

1. **Protection by Access Contracts.** Access control constraints may be formulated as commitments, specifying a contract between a class and its potentially malicious collaborators (e.g., JAC import types).

2. **Safe Dynamic Linking.** Appropriate obligations must be formulated to enforce the contract at link time (e.g., `safe-method-import`). The semantics of the obligations are defined through the development of an obligation library.

3. **Trust Inspiration.** To inspire trust, collaborating code units must formulate matching commitments (e.g., JAC export types), and ensure that they live up to their promises. The latter check is performed at link time by a PVM.

4. **Secure Software Extension.** Dynamic software extension is enabled in Java through the combination of dynamic loading and subtyping. The current protection scheme can also handle dynamic software extension through the design of appropriate subtyping rules (e.g., `safe-method-override`).

| | PVM | Obl. Lib. | Shared | Total |
|---|---|---|---|---|
| JAC | 2082 | 490 | 75 | 2647 |
| Confined Type | 2257 | 579 | 90 | 2926 |

**Figure 7: Relative Complexity (in LOC) of the JAC and Confined Types Implementations**

Capability type systems [8] similar to JAC can be readily enforced in the PVM framework.

## 7. CONCLUDING REMARKS

### 7.1 Discussion

*Wider Applicability.* Although only JAC was examined in this paper, the PVM framework is designed to be a general extension mechanism for hosting a wide range of link-time verification services. Efforts are underway to develop PVMs and obligation libraries for confined types [55] and alias burying [7], the other two alias control type systems that appear with JAC [37] in the *SP&E Special Issue on Aliasing in Object-Oriented Systems*. Given our previous experience with JAC, a first working implementation[4] of confined types was obtained in 35 hours, with a complexity comparable to that of JAC (see Figure 7). Implementation of alias burying is still on-going. Such implementation exercises will further demonstrate the generality of the PVM framework, and reveal potential areas of improvement.

*Trusted Computing Base.* The administrator of the Aegis VM is responsible for configuring it with PVMs and obligation libraries, which are developed in native code, and are considered part of the trusted computing base (TCB). A potential objection to this design is that application-specific verifiers should be part of the application layer, expressed in JVM bytecode as a classloader. Such an alternative approach, *if viable*, would reduce the size of the TCB, and prevent buggy verifiers from crashing the JVM. Attractive as it is, such an approach is unfortunately *not* feasible in the general case. Firstly, such a proposal overlooks the need for interchecking application classes with built-in classes. The latter are defined by the bootstrap classloader rather than user-defined classloaders, and thus a verification module embedded in an user-defined classloader will have no way of generating proof obligations and commitments for those built-in classes. Secondly, and more importantly, the attachment and discharging of proof obligations at specific linking events cannot be modeled at the bytecode level. Introducing application-specific checks to strategic points of the dynamic linking process can only be achieved by modifying the internal behavior of the JVM. The existing design appears to be the only technically feasible approach in the general case. We however do not rule out the possibility that some verification domains (e.g., [15]) may admit an implementation embedded in a classloader.

An interesting approach to reduce programming errors in PVMs and obligation libraries is to design alternative PVM and obligation library APIs that support plugins coded in

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

a safe C dialect such as Cyclone [32]. The feasibility of this approach is currently being explored.

***Efficiency.*** Early experience with small example code does not indicate that the PVM facility causes any significant performance degradation. Because there is no existing Java code base that are heavily annotated with augmented types such as JAC, it is very difficult to obtain performance measurement with real Java applications. However, we predict that any performance impact caused by the introduction of the PVM framework is correlated to the degree of static and dynamic coupling of application classes: the tighter classes are coupled, the more linking primitives are executed, and thus more obligation discharging and verification sessions will be resulted. To better profile the performance characteristics of the PVM facility, we are currently designing an artificial application domain in which the degree of class coupling can be precisely manipulated.

***Self-Certifying Code.*** To reduce the performance overhead incurred by link-time verification, a code producer may annotate untrusted code with a program-analytic certificate that witness to the safety of the code. Upon receiving the annotated code, the link-time verification service checks statically if the certificate indeed establishes the safety property in question. Intuitively, certificate checking is more efficient than establishing code safety from scratch. This is the motivation behind the idea of Proof-Carrying Code (PCC) [44, 43] and J2ME pre-verification. The Proof Linking architecture and its PVM realization support the adoption of self-certifying code in a legacy environment, in which a Java application may contain *both* unannotated code and self-certifying code. Specifically, a PVM may be constructed to perform efficient certificate checking for self-certifying code, and resort to full verification for unannotated code. In both cases, the PVM generates a verification interface (i.e., proof obligations and commitments) for the classfile in question. Because of the modular verification architecture, proof linking processes proof obligations and commitments consistently in both cases despite the difference in how intrachecking is performed (i.e., certification checking vs full verification). Consult [24, 22] for an extensive discussion of how Proof Linking supports the interoperability of various distributed verification protocols.

## 7.2 Related Work

The correctness of the Proof Linking architecture, especially its interaction with lazy, dynamic linking, has been studied rigorously [23]. The correctness proof has been generalized to account for multiple classloaders [24]. Previous work focuses on how the Proof Linking architecture may improve the comprehensibility and maintainability of the standard bytecode verifier [23], and how it can facilitate distributed verification [24]. This paper takes the research program a significant step further by (1) providing a complete implementation of the Proof Linking architecture in the Aegis VM, (2) applying the Proof Linking architecture to build an extensible protection mechanism, in which the link-time verification service becomes pluggable, and (3) evaluating the utility of the extension mechanism in a concrete verification domain.

The lack of extensibility in mainstream VM implementations has motivated a number of design alternatives. The XVM [31] allows applications to load code expressing application-specific policies for customizing the run-time compilation strategy, object placement within the heap, and thread scheduling. The Jupiter VM [16] is based on a building block architecture, in which pluggable resource factories can be used to control run-time mechanisms such as thread model, memory allocation strategy, and object locality. The OVM [46] employs an object-oriented application framework to provide a reflective, customizable intermediate language. The Aegis VM is the first to address the need of extensibility on the security front by offering a configurable link-time verification service within a formally verified architecture.

The study of type-safe linking was pioneered in the work of Cardelli [10], which was followed by works such as typed object files for TAL [28] and the comprehensive type system of Duggan [17].

Built on their prior experience in formalizing various aspects of Java's bytecode verifier and dynamic linking model [29, 14, 47, 13], Qian *et al* [48] proposed a formal specification of the Java classloading model, taking into account of both bytecode verification and the on-going maintenance of loading constraints. In their specification, bytecode verification is modeled as a modular primitive. Interchecking and classloading is avoided by the formulation of *subtype constraints* to capture intermodular dependencies, a strategy similar to the formulation of proof obligation. The subtype constraints are maintained and verified lazily in the same way as the type equivalence constraints mandated by Liang and Bracha [38, 39]. Two points of comparison are observed. While proof obligations can be arbitrary queries, Qian *et al* focus only on Java subtyping constraints. Type consistency is modeled as a constraint problem over semilattices. In contrast, the generic proof linking model can be applied to a wide spectrum of verification domains. Furthermore, subtype constraints are *maintained on-the-fly*, whereas proof obligations are scheduled to be discharged prior to the execution of their target linking primitives. This scheduling element introduces an additional dimension of complexity into the Proof Linking architecture.

Foster *et al* [26] developed a general framework for adding user-defined type qualifiers to a language. The framework supports qualifier polymorphism, and handles qualifier inferences separately from the standard type system. The framework has been successfully applied to detect format string vulnerabilities [51]. The framework was subsequently extended to account for flow-sensitive type qualifiers [27]. The inference algorithm has been implemented in a tool CQUAL, which allows programmers to annotate C programs with application-specific type qualifiers, and subsequently checks for type-safety statically. Although the work of Foster *et al* shares with PVM the same goal of enabling users to incorporate application-specific verification into a programming language system, the two works differ in several aspects. Firstly, while the work of Foster *et al* represents a type-theoretic study of user-defined type qualifiers, PVM is a plug-in architecture aimed at supporting a wide-range of static verification tasks. Generality is achieved through a customizable proof linking mechanism, in which verification interfaces are represented as proof obligations and commitments. Secondly, while CQUAL is a compile-time analysis tool, the PVM facility is a link-time protection mechanism. The explicit modeling of linking primitives and the formulation of obligation discharging schedules are essential for enforcing safety in a lazy, dynamic linking environment.

Extensibility is achieved in this work through modularization. Alternatively, software adaptation could be conducted in a more systematic manner through the application of advanced programming constructs. Originally proposed as an alternative to encapsulation for implementing separation of concern, aspect-oriented programming [34, 35] can be seen as a high-level program extension mechanism. Specifically, aspect-oriented programming systems allow the weaving of *aspect code* into programmer-specified *join points*, thereby modifying the behavior of the underlying program. Behavioral reflection [20] and intercessory metaobject protocols [36] allow operations such as method invocation to be intercepted. When an interception occurs, a metaobject will be notified of the event via some kind of method call back facility. Programmers can customize the semantics of the metaobject to achieve the effect of software extension.

## 7.3 Future Work

As the extension APIs for PVMs and obligation libraries are orthogonal to the rest of the VM architecture, an interesting endeavor is to reproduce the same extension APIs in other JVM implementations, thereby making PVMs and obligation libraries interoperable with multiple JVMs.

To facilitate program analysis, the Aegis VM passes to the PVM verification function the AST and type analysis results of the target classfile. The verification function can utilize the control flow and typing information to speed up intrachecking. Further speedup may be achievable if the dataflow structure of bytecode methods can be made explicit, and is passed along with the classfile AST to the verification function. One promising direction is to develop reusable PVMs that summarize the dataflow structure of a bytecode method in a SSA-based representation [4] along the line of Jimple [54] or SafeTSA [3]. To support this, an extension of the PVM API is planned to allow PVMs to share analysis results with each other.

The author is exploring the design of a capability type system in which high-level access control constraints can be expressed to enable secure cooperation in Java bytecode programs [25]. Such a type system will exploit the PVM facility as a development platform. Another direction of interest is to explore the embedding of Jif (Java Information Flow) [42, 41] in the PVM framework.

Proof Linking generalizes the link-time access control checks performed in a standard JVM. An extensible protection mechanism is obtained by making these access control checks customizable. Another set of safety checks performed by the JVM are loading constraints, which are essentially equivalence constraints over binding of class symbols from different namespaces [38]. An interesting direction is to generalize the idea of Qian *et al* [48], and make loading constraints customizable: users may introduce application-specific constraint systems over the binding of class symbols, and maintain binding consistency with pluggable constraint solvers. This flexibility could yield an extensible protection mechanism for which the subtype constraint system of Qian *et al* becomes a special case.

Both the the PVM facility and the extensible loading constraint system suggested in the previous paragraph are special-purpose extension mechanisms. An existing check is identified, and customizability is introduced through some kind of special-purpose plug-in mechanism. An alternative is to consider the application of general-purpose software adaptation mechanisms, such as Aspect-Oriented Programming, to extend the protection mechanism of a JVM. In this approach, customizable join points are documented and publicized as an Extension Programming Interface. Customization code is then weaved into these join points as security aspects. Such an approach may reduce the probability of programming error, and thus simplify the process of transforming an existing protection mechanism into an extensible one.

## 7.4 Conclusion

A extensible protection mechanism, Pluggable Verification Modules, has been designed and implemented for the JVM. Enabled by the Proof Linking architecture, the extension mechanism turns link-time verification into a pluggable service. The software engineering and security engineering benefits of the extension framework have been demonstrated. This suggests the notion of pluggable verification services is more than just an isolated technological novelty, but rather it represents a paradigm for building safe language environments. The notion of safety changes as the protection needs of a language environment evolve. A well-designed language environment should anticipate evolution by building into itself an extension mechanism that supports the augmentation of its verification service. Much like a pipeline architecture (i.e., scanner, parser, optimizer, code generator, etc) is a standard architecture for compilers, the author argues that a modular verification architecture such as Proof Linking should be a standard design for safe language environments.

## 8. ACKNOWLEDGEMENTS

This work was supported in part by an NSERC Discovery Grant. The author is grateful for the valuable comments from Warren Burton, Robert Cameron, James Cordy, Tiko Kameda, and the anonymous reviewers.

## 9. REFERENCES

[1] Jonathan Aldrich, Craig Chambers, and David Notkin. ArchJava: Connecting software architecture to implementation. In *Proceedings fo the 24th International Conference on Software Engineering (ICSE'02)*, pages 187–197, Orlando, Florida, USA, May 2002.

[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 (OOPSLA'02)*, pages 311–330, Seattle, Washington, USA, November 2002.

[3] Wolfram Amme, Niall Dalton, Jeffery von Ronne, and Michael Franz. SafeTSA: A type safe and referentially secure mobile-code representation based on static single assignment form. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation (PLDI'01)*, pages 137–147, Snowbird, Utah, USA, May 2001.

[4] John Aycock and Nigel Horspool. Simple generation of static single-assignment form. In *Proceedings of the 9th*

*International Conference on Compiler Construction (CC'00)*, volume 1781 of *Lecture Notes in Computer Science*, pages 110–124, Berlin, Germany, April 2000.

[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 (SOSP'95)*, pages 267–284, Copper Mountain, Colorado, USA, December 1995.

[6] Joshua Bloch. JSR 175: A metadata facility for the Java programming language. `http://www.jcp.org/en/jsr/detail?id=175`.

[7] John Boyland. Alias burying: Unique variables without destructive reads. *Software — Practice and Experience*, 31(6):533–553, May 2001.

[8] John Boyland, James Noble, and William Retert. Capabilities for sharing: A generalisation of uniqueness and read-only. In *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP'01)*, Budapest, Hungary, June 2001.

[9] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: adding genericity to the Java programming language. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'98)*, pages 183–200, Vancouver, British Columbia, Canada, October 1998.

[10] 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.

[11] 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 (ICSE'97)*, pages 22–32, Boston, Massachusetts, USA, May 1997.

[12] David G. Clarke, John M. 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 (OOPSLA'98)*, pages 48–64, Vancouver, British Columbia, Canada, October 1998.

[13] Alessandro Coglio and Allen Goldberg. Type safety in the JVM: Some problems in JDK 1.2.2 and proposed solutions. In *Proceedings of the 2nd ECOOP Workshop on Formal Techniques for Java Programs*, Sophia Antipolis and Cannes, France, June 2000.

[14] Alessandro Coglio, Allen Goldberg, and Zhenyu Qian. Toward a provably-correct implementation of the JVM bytecode verifier. In *Proceedings of the OOPSLA'98 Workshop on the Formal Underpinnings of Java*, October 1998.

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

[16] Patrick Doyle and Tarek S. Abdelrahman. A modular and extensible JVM infrastructure. In *Proceedings of the USENIX 2nd Java Virtual Machine Research and Technology Symposium (JVM'02)*, San Francisco, California, USA, August 2002.

[17] Dominic Duggan. Type-safe linking with recursive DLL and shared libraries. *ACM Transactions on Programming Languages and Systems*, 24(6):711–804, November 2002.

[18] ECMA. Common language infrastructure (CLI). Standard 335, ECMA, December 2002.

[19] 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 (SOSP'95)*, Copper Mountain, Colorado, USA, December 1995.

[20] Jacques Ferber. Computational reflection in class based object oriented languages. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'89)*, pages 317–326, New Orleans, Louisiana, USA, October 1989.

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

[22] Philip W. L. Fong. *Proof Linking: A Modular Verification Architecture for Mobile Code Systems.* PhD thesis, School of Computing Science, Simon Fraser University, Burnaby, B.C., Canada V5A 1S6, 2004.

[23] 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.

[24] 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.

[25] Philip W. L. Fong and Cheng Zhang. Capabilities as alias control: Secure cooperation in dynamically extensible systems. Technical Report CS-2004-3, Department of Computer Science, University of Regina, Regina, Saskatchewan, Canada S4S 0A2, 2004. ISBN:0-7731-0479-8.

[26] Jeffrey S. Foster, Manuel Fähndrich, and Alexander Aiken. A theory of type qualifiers. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation (PLDI'99)*, pages 192–203, Atlanta, Georgia, USA, May 1999.

[27] Jeffrey S. Foster, Tachio Terauchi, and Alex Aiken. Flow-sensitive type qualifiers. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Lanuage Design and Implementation (PLDI'02)*, pages 1–12, Berlin, Germany, June 2002.

[28] Neal Glew and Greg Morrisett. Type-safe linking and modular assembly language. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'99)*, pages

250–261, San Antonio, Texas, USA, January 1999.

[29] Allen Goldberg. A specification of Java loading and bytecode verification. In *Proceedings of the 5th ACM Conference on Computer and Communications Security (CCS'98)*, pages 49–58, San Francisco, California, USA, November 1998.

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

[31] Timothy L. Harris. Extensible virtual machines. Technical Report 525, University of Cambridge Computer Laboratory, Cambridge, UK, December 2001.

[32] Trevor Jim, Greg Morrisett, Dan Grossman, Michael Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of C. In *Proceedings of the USENIX Annual Technical Conference*, pages 275–288, Monterey, California, USA, June 2002.

[33] Sarfraz Khurshid, Darko Marinov, and Daniel Jackson. An analyzable annotation language. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'02)*, pages 231–245, Seattle, Washington, USA, November 2002.

[34] Cregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Christina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP'97)*, volume 1241 of *Lecture Notes in Computer Science*, Finland, June 1997. Springer-Verlag.

[35] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey palm, and William Griswold. An overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP'01)*, volume 2072, pages 327–353, Budapest, Hungary, June 2001.

[36] Gregor Kiczales, Jim Des Rivieres, and Daniel Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

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

[38] 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.

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

[40] J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 2nd edition, 1987.

[41] Andrew C. Myers. JFlow: Practical mostly-static information flow control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'99)*, pages 228–241, San Antonio, Texas, USA, January 1999.

[42] Andrew C. Myers and Barbara Liskov. Complete, safe information flow with decentralized labels. In *Proceedings of the 1998 IEEE Symposium on Security and Privacy (S&P'98)*, Oakland, California, USA, May 1998.

[43] 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.

[44] 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, USA, October 1996.

[45] R. O'Callahan. A simple, comprehensive type system for Java bytecode subroutines. In *Proceedings of the 26th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'99)*, pages 70–78, San Antonio, Texas, USA, January 1999.

[46] K. Palacz, J. Baker, C. Flack, C. Grothoff, H. Yamauchi, and J. Vitek. Engineering a customizable intermediate representation. In *Proceedings of the Workshop on Interpreters, Virtual Machines and Emulators (IVME'03)*, San Diego, California, USA, June 2003.

[47] Zhenyu Qian. Standard fixpoint iteration for Java bytecode verification. *ACM Transactions on Programming Languages and Systems*, 22(4):638–672, July 2000.

[48] Zhenyu Qian, Allen Goldberg, and Alessandro Coglio. A formal specification of Java class loading. In *Proceedings of the 15th Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'00)*, pages 325–336, Minneapolis, Minnesota, USA, October 2000.

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

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

[51] Umesh Shankar, Kunal Talwar, Jeffrey S. Foster, and David Wagner. Detecting format-string vulnerabilities with type qualifiers. In *Proceedings of the 10th USENIX Security Symposium*, Washington, D.C., USA, August 2001.

[52] Raymie Stata and Martin Abadi. A type system for Java bytecode subroutines. *ACM Transactions on Programming Languages and Systems*, 21(1):90–137, January 1999.

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

[54] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a Java bytecode optimization framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research (CASCON'99)*, Mississauga, Ontario, Canada, November 1999.

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