Pluggable Verification Modules: An Extensible Protection Mechanism for the JVM

Philip W. L. Fong

Technical Report CS-2003-11
November 24, 2003

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

ISBN 0-7731-0463-1

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

November 24, 2003

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 [16, 17], 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 [14]. To evaluate the generality and utility of the extensible protection mechanism, an augmented type system JAC (Java Access Control) [26] has been successfully implemented as a PVM.

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 [8, 37], scriptable applications, and software systems with plug-in architectures [4, 12, 31]. 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. Such an approach forms the security foundation of the Java Virtual Machine (JVM) [29], 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 [26, 6, 5], annotation languages [22, 1], and other forms of static analyses [27]. 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. What has been checked in the code producer's *verification environment* may no longer hold in the code consumer's verification environment. A malicious code producer may verify the target program in a liberal verification environment and then falsely claim that the program is safe. The consequence of not checking the miscertified program against the verification environment on the code consumer side may be devastating. 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 explains why it is rare to see the mentioned works materialize into link-time protection mechanisms for the JVM. That is, until now.

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 [16, 17], 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 [14]. To evaluate the generality and utility of the extensible protection mechanism, an augmented type system JAC (Java Access Control) [26] has been successfully implemented as a PVM.

This paper is organized as follows. Section 2 reviews the previously proposed Proof Linking architecture, which forms the theoretical basis for the PVM mechanism. The Aegis VM and its implementation effort is described in Section 3. Section 4 and 5 detail the design of the PVM facility implemented in the Aegis VM. The utility of PVM is demonstrated in Section 6, in which the implementation of the JAC type system as a PVM is described. Related works are discussed in 7. Section 8 points out future research directions.

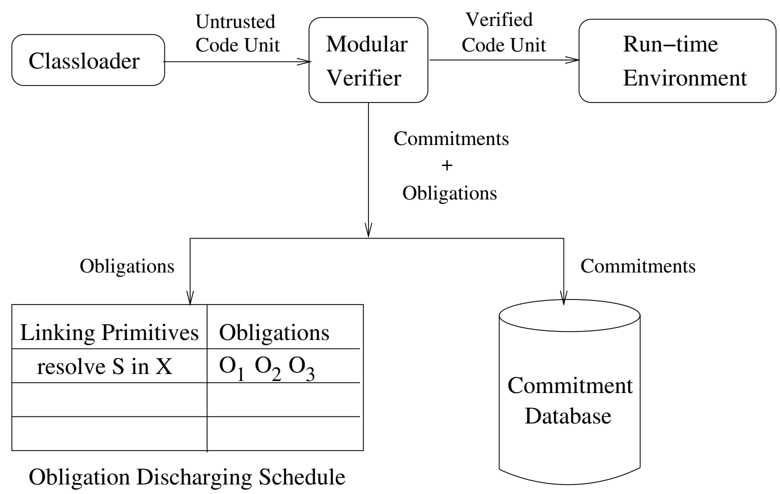

Figure 1: Modular Verification

2 The Proof Linking Architecture

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 compatiblity between this type interface and a given type environment. Cardelli succinctly called the two tasks *intrachecking* and *interchecking* [7]. Unfortunately, the two tasks are not cleanly separated in a typical implementation of the bytecode verification procedure [29]. Specifically, in the course of intrachecking a classfile, classloading is frequently initiated by the bytecode verifier in order to bring in the type interface of other classfiles for interchecking purposes. The result 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 static analyzer have to possess intimate knowledge of the VM internal, any undisciplined classloading performed by the analyzer for the sake of interchecking may also perturb the soundness of the dynamic linking semantics. Simply put, such a verification architecture is not designed to support extensibility.

An alternative verification architecture, Proof Linking [16, 17], was proposed to address the need of modularity in the JVM. Intrachecking and interchecking are cleanly separated. Intrachecking of classfiles is performed by a *modular verifier*, which infers for each classfile a *verification interface* composed of *proof obligations* and *commitments* (Figure 1). Commitments are assertions established by the verification procedure, 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. Interchecking thus in-

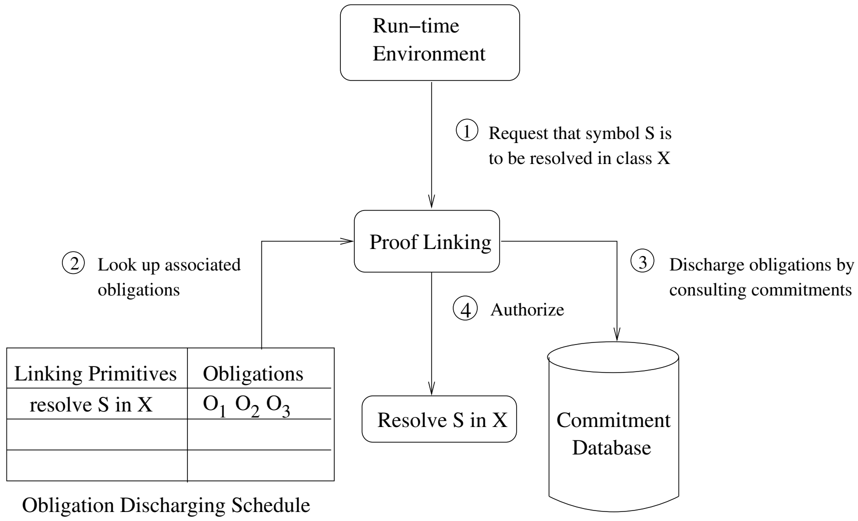

Figure 2: Incremental Proof Linking

volves 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.

Incremental interchecking is achieved through a process called *proof linking* (Figure 2). 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 [16, 17], incremental proof linking is modeled abstractly using *deductive database concepts* [30], whereby proof obligations are queries, commitments are facts stored in a database, and logic programs are formulated to express interchecking logics such as type rules. 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.

3 The Aegis VM

The Proof Linking architecture was originally proposed as a means to improve the maintainability and comprehensiblity of the JVM bytecode verification procedure. The main contribution 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. This architectural property is exploited in the PVM mechanism, which allows arbitrary *modular verifiers* to be "plugged" into the Aegis VM for screening untrusted classfiles. Without having to perform any interchecking and classloading, each modular verifier 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 4.

A *generic proof linking mehanism* 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. 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. Details of the generic proof linking mechanism and user-defined verification domains can be found in Section 5.

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 [14]. Five 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 is already in the publicly accessible CVS repository, and will be incorporated into the next release.

The PVM facility and the generic proof linking mechanism are designed to meet the following goals:

1. **Generality.** The PVM facility and the generic proof linking mechanism should be applicable to a wide range of static analyses.

2. **Efficiency.** Since linking events occur frequently in a Java platform, the generic proof linking mechanism must be efficient enough so that the overhead introduced by obligation discharging is acceptable.

3. **Utility.** The effort required of a programmer to incorporate an application-specific analysis into the dynamic linking process of a JVM should be significantly reduced when performed through the PVM facility.

Design highlights are outlined in Sections 4 and 5 to illustrate how the above design goals are achieved. Consult [15] for low-level implementation details.

4 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 plug-in mechanism. This section outlines the design of the PVM mechanism.

4.1 PVM Life Cycle

The Aegis VM provides a command line option for users to specify the path of a PVM. Multiple PVMs may be specified in the command line. When the Aegis VM bootstraps, each of the specified PVMs are loaded. Every PVM exports a C-string identifying the *verification domain* to which it belongs. As we shall see in the the following, this verification domain identifier will be used by the generic proof linking mechanism for interpreting the verification interfaces generated by this PVM.

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 each PVM, whereby application-specific intrachecking is conducted (Section 4.2). Successful intrachecking generates a *verification interface* (Section 4.3), 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.

4.2 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 [36, 32]), 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. The Aegis VM provides accessor functions for traversing ASTs, control flow graphs and type states. A verification function can exploit these to speed up verification.

4.3 Verification Interface

The verification function constructs a verification interface for each classfile passing intrachecking. The Aegis VM provides constructors for building verification interfaces, each of which is composed of four components — commitments (4.3.1), proof obligations (4.3.2),

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

Figure 3: Abstract Syntax of Proof Obligations

an auxiliary symbol table (4.3.3), and an obligation discharging schedule (4.3.4). Verification interfaces generated by a PVM are defined with respect to the verification domain to which the PVM belongs. The Aegis VM stores a verification interface along with the corresponding class definition, indexing according to its verification domain. Unloading of a class automatically cleans up the memory resources occupied by the associated verification interfaces.

4.3.1 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.

4.3.2 Proof Obligations

The verification function formulates proof obligations to capture external dependencies of a target classfile. Every proof obligation is a ground query composed of a *predicate identifier* and zero or more *arguments*. The abstract syntax of a proof obligation is given in Figure 3.

A fixed number of predicate symbols are defined for each verification domain (Section 5.1). Every predicate symbol is uniquely identified within a verification domain by a 16-bit unsigned integer. This number is referenced as the predicate identifier of an obligation.

Aspects of the target classfile type interface can be named as obligation arguments. Specifically, the `this`, `super`, `interface`, `field` and `method` syntax are used for naming the target class, its immediate superclass, immediate superinterfaces, declared fields and methods respectively. The *index* field identifies the specific member 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. An import symbol can only be named in obligations attached to the corresponding resolution primitive of the symbol (Section 4.3.4).

The obligation argument types described so far refer to VM data structures that are defined independent of the 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., *java.lang.Throwable* is needed for checking if a class can be thrown as 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 5.1 for more details.

As the Aegis VM has to explicitly track proof obligations, a compact obligation encoding has been derived [15], whereby an obligation with k arguments can be encoded with $k+1$ 32-bit machine words.

4.3.3 Auxiliary Symbol Table

Obligation argument types described so far does not permit class symbols appearing in the type signature of a constant pool field, method, or interface method reference to be named as obligation argument. The PVM facility provides a way for referring to these *auxiliary symbols*. Specifically, a verification function may construct an *auxiliary symbol table* in the verification interface for identifying method argument types, method return types, and field types as potential obligation arguments. Such auxiliary symbols may be named using the `auxiliary-symbol` syntax (Figure 3). The *index* field identifies a specific member of the 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 5.4).

4.3.4 Obligation Discharging Schedule

Because of lazy, dynamic linking, obligation discharging proceeds in an incremental mannar. 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* [29, 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.

When the classfile representation of a class C is examined, the verification function may attach obligations to any of the above linking primitives[1]. 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.

5 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.

5.1 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[2]. This function will be dispatched when a corresponding obligation is to be discharged. A *predicate dispatching table*, analogous to a virtual function table, is exported.

Global classnames. An array of classnames is exported to specify the names of global classes in the verification domain. It is assumed that the bootstrap classloader will be used for loading these classes.

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

[1] By design, only the above family of linking primitives are accessible to the verification function when it examines class C. This arrangement guarantees that the correctness condition Safety is satisfied [16].

[2] The formulation of the native predicate functions must correspond to a monotonic logic in order for the correctness condition Monotonicity to be satisfied [16]. It is the responsibility of the obligation library developer to take care of this requirement.

5.2 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[3]. A brief summary of the API facilities is given below. A complete list of helper functions in the API can be found in [15].

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.

5.3 Life Cycle of an Obligation Library

The Aegis VM has a command line option that allows users to specify the path of an obligation library. Multiple obligation libraries may be specified, thereby providing the Aegis VM with vocabularies for multiple verification domain. When the Aegis VM starts up, all the obligation libraries specified in the command line 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 discharing. Obligation libraries are properly cleaned up and unloaded before the Aegis VM shuts down.

5.4 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

[3]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 [16].

2. Each obligation argument is resolved into a corresponding pointer to a VM data structure or a global constant of the current verification domain. The arguments are placed in a temporary argument array. 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.

6 Java Access Control

JAC (Java with Access Control) [26] 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. Due to its simplicity and its relevance to access control, this verification domain is chosen as an example application of the PVM, whereby the utility of the PVM facility is assessed.

6.1 Motivation

To understand how the `readonly` qualifier 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.

6.2 The JAC Type System

Originally designed for typing Java source programs at compile-time, the JAC type system is recast here as an augmented type system for the JVM bytecode language. The JAC type system defines two types, namely, `readonly` and $\perp$. The bottom type $\perp$ 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.

6.2.1 Subtyping

The bottom type $\perp$ is a subtype of `readonly`, and as such the conversion of $\perp$ 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: $A \rightarrow B <: A' \rightarrow B'$ if $A' <: A$ and $B <: B'$.

6.2.2 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 $\perp$ type.

6.2.3 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 endorcement primitive is executed.

Resolution of a constant pool (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.

6.2.4 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 complete list can be found in [15]. The effect of a bytecode instruction is presented in a notation popularized by [29]. 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 $\langle$*fieldref*$\rangle$ is a reference field with a `readonly` import type, then v is `readonly`. If $\langle$*fieldref*$\rangle$ is a reference field, and o has a `readonly` type, then v is `readonly`. Otherwise, v is $\perp$.

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 $\langle$*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 $\langle methodref \rangle$ be $\langle B_0, B_1, \ldots, B_k \rangle \to B$. Then $A_i <: B_i$ for $0 \leq i \leq k$, and $B <: A$.

6.3 Embedding JAC Type Interface

To make JAC enforceable at link time, every classfile must carry a JAC type interface. A compact encoding has been designed [15] for embedding a JAC type interfaces into classfiles through the classfile attribute facility [29, 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 arguments have type $\perp$. The provision of assuming a default type interface for classfile not carrying a JAC attribute renders it possible to reuse legacy classfiles not compiled for JAC typechecking. This is particularly handy in the case of the standard Java class library — hundreds of system classes can be reused as is.

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.

6.4 Pluggable 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 *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.

6.5 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. If the classfile carries a JAC attribute, then the embeded 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. An iterative dataflow analysis algorithm is applied to the bytecode methods. Type constraints in Section 6.2.4 are verified. 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. Consequently, this step can be skipped entirely for classfiles with no JAC attribute.

3. Proof obligations are generated. 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:

   `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. That is, although intraprocedural typechecking may be optimized away in special cases, interprocedural typechecking must never be bypassed.

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

   `safe-method-override this`

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

6.6 Example

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.

`Bob.sum :  readonly` $\rightarrow \perp$

When equipped with the JAC PVM and obligation library, the Aegis VM will reject 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.

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`.

`Bob.sum :  readonly` $\rightarrow \perp$

When the class `Bob` is defined, the verification function of the JAC PVM will be invoked. 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 dispatched 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.

Now, 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.

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 InstantiationException,
               ClassNotFoundException,
               IllegalAccessException {
        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`.

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

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

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

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` $\rightarrow \perp$) is a subtype of the default import type ($\perp \rightarrow \perp$).

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

$$\texttt{Charlie.sum} : \texttt{readonly} \rightarrow \perp$$

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.

7 Related Works

The correctness of the Proof Linking architecture, especially its interaction with the lazy, dynamic linking, has been studied rigoriously [16]. The correctness proof has been generalized to account for multiple classloaders [17].

The study of type-safe linking was pioneered in the work of Cardelli [7], which was followed by works such as typed object files for TAL [20] and the comprehensive type system of Duggan [11].

Built on their prior experience in formalizing various aspects of Java's bytecode verifier and dynamic linking model [21, 10, 33, 9], Qian *et al* [34] 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 [28, 29]. 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* [18] 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 [35]. The framework was subsequently extended to account for flow-sensitive type qualifiers [19]. 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 an 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 as a means for implementing separation of concern, aspect-oriented programming [23, 24] can be seen as a high-level program extension mechanism. Specifically, aspect-oriented programming system allows the weaving of *aspect code* into programmer-specified *join points*, thereby modifying the behavior of the underlying program. Behavioral reflection [13] and intercessory metaobject protocols [25] 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, thereby achieving the effect of software extension.

8 Future Works

This paper reports the design and implementation of an *extensible protection mechanism* for the Java Virtual Machine (JVM), in which application-specific static analyses can be safely incorporated into the dynamic linking process of a JVM as *Pluggable Verification Modules (PVMs)*. The plug-in mechanism has been successfully implemented in the open source Aegis VM. As the extension API is orthogonal to the rest of the VM architecture, an interesting endeavor is to reproduce the same extension API in other JVM implementations, thereby making PVMs interoperable with multiple JVM implementations.

To facilate 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 for the Aegis VM to summarize the dataflow structure of a bytecode method in a SSA-based representation [3] along the line of Jimple [38] or SafeTSA [2].

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 [28]. An interesting direction is to generalize the idea of Qian *et al* [34], 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.

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*, pages 187–197, Orlando, Florida, May 2002.

[2] 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*, pages 137–147, Snowbird, Utah, May 2001.

[3] John Aycock and Nigel Horspool. Simple generation of static single-assignment form. In *Proceedings of the 9th International Conference on Compiler Construction*, volume 1781 of *Lecture Notes in Computer Science*, pages 110–124, Berlin, Germany, April 2000.

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

[5] Chandrasekhar Boyapati, Barbara Liskov, and Liuba Shrira. Ownership types for object encapsulation. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 213–223, New Orleans, Louisiana, January 2003.

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

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

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

[9] 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.

[10] 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.

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

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

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

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

[15] Philip W. L. Fong. *Proof Linking: A Modular Verification Architecture for Mobile Code Systems*. PhD thesis, School of Computing Science, Simon Fraser University, Burnaby, BC, Canada V5A 1S6, 2003. Upcoming.

[16] 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), October 2000.

[17] 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*, pages 53–66, Monterey, California, April 2001.

[18] 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*, pages 192–203, Atlanta, Georgia, May 1999.

[19] 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*, pages 1–12, Berlin, Germany, June 2002.

[20] 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, January 1999.

[21] Allen Goldberg. A specification of java loading and bytecode verification. In *Proceedings of the 5th ACM Conference on Computer and Communications Security*, pages 49–58, San Francisco, California, November 1998.

[22] 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*, pages 231–245, Seattle, Washington, November 2002.

[23] 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*, volume 1241 of *Lecture Notes in Computer Science*, Finland, June 1997. Springer-Verlag.

[24] 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*, volume 2072, pages 327–353, Budapest, Hungary, June 2001.

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

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

[27] Larry Koved, Marco Pistoia, and Aaron Kershenbaum. Access rights analysis for Java. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 359–372, Seattle, Washington, November 2002.

[28] 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, October 1998.

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

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

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

[32] 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*, pages 70–78, San Antonio, Texas, January 1999.

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

[34] 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*, pages 325–336, Minneapolis, Minnesota, October 2000.

[35] 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., August 2001.

[36] 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.

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

[38] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a Java bytecode optimization framework. In *Proceedings of CASCON'99*, Toronto, Ontario, November 1999.