PROOF LINKING: A MODULAR VERIFICATION ARCHITECTURE FOR MOBILE CODE SYSTEMS

by

Philip W. L. Fong

B.Math (Computer Science), University of Waterloo, 1993

M.Math (Computer Science), University of Waterloo, 1995

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY
in the School
of
Computing Science

SIMON FRASER UNIVERSITY
January 11, 2004

APPROVAL

Name: Philip W. L. Fong

Degree: Doctor of Philosophy

Title of thesis: Proof Linking: A Modular Verification Architecture For Mobile Code Systems

Examining Committee: Dr. Anoop Sarkar
Chair

Dr. Robert D. Cameron, Senior Supervisor

Dr. Warren Burton, Supervisor

Dr. Tiko Kameda, Supervisor

Dr. Uwe Glässer, SFU Examiner

Dr. James R. Cordy, External Examiner
School of Computing
Queen's University

Date Approved:

Abstract

This dissertation presents a critical rethinking of the Java bytecode verification architecture from the perspective of a software engineer. In existing commercial implementations of the Java Virtual Machine, there is a tight coupling between the dynamic linking process and the bytecode verifier. This leads to delocalized and interleaving program plans, making the verifier difficult to maintain and comprehend. A modular mobile code verification architecture, called Proof Linking, is proposed. By establishing explicit verification interfaces in the form of proof obligations and commitments, and by careful scheduling of linking events, Proof Linking supports the construction of bytecode verifier as a separate engineering component, fully decoupled from Java's dynamic linking process. This turns out to have two additional benefits: (1) Modularization enables distributed verification protocols, in which part of the verification burden can be safely offloaded to remote sites; (2) Alternative static analyses can now be integrated into Java's dynamic linking process with ease, thereby making it convenient to extend the protection mechanism of Java. These benefits make Proof Linking a competitive verification architecture for mobile code systems. A prototype of the Proof Linking Architecture has been implemented in an open source Java Virtual Machine, the Aegis VM (`http://aegisvm.sourceforge.net`).

On the theoretical side, the soundness of Proof Linking was captured in three correctness conditions: Safety, Monotonicity and Completion. Java instantiations of Proof Linking with increasing complexity have been shown to satisfy all the three correctness conditions. The correctness proof had been formally verified by the PVS proof checker.

To my beloved wife, Rainbow, who taught me
the meaning of beauty and authenticity.

Acknowledgments

Funding for this research was provided by an NSERC scholarship and an NSERC grant.

I would like to thank my supervisory committee and my examiners. Your input to this work has greatly improved its quality.

I would like to thank Rob Cameron, my senior supervisor, for his support and guidance over the years. Thank you for having confidence in me and encouraging me in countless occasions. You have inspired me to strive for a holistic research agenda, in which theory blurs into practice.

I would like to thank Prem Devanbu, my internship supervisor at AT&T. As a role model you have shown me that a scholar can be learned and yet humble, serious in scholarship and yet humorous in character. You have shown me that doing world-class research can be fun.

I would like to thank Qiang Yang, my former supervisor. Although I no longer work in the same area as you do, I always remember you as the one who introduced me to the world of scholarship. For this I am forever grateful.

A special group of people has become a constant source of joy and support, without which my PhD career would have become much dryer: Tinnie Cham, Cecilia Chan, Benny Chin, Helen Gan, Kevin Hung, Edward Liu, Elsie Liu, and Stephen Siu. I also want to thank Fenton Ho, Kai Kwok and Jasmine Wong: your companionship throughout this journey has been one of the most precious experiences I have ever had. I am also in debt to the heroic support of my parents and siblings, who is never tired of lending a helping hand whenever I needed it.

I especially want to express my deepest gratitude to my wife, Rainbow, who demonstrated extraordinary loyalty and love throughout the course of my study. In every way this work would not have been possible without your support and encouragement.

Above all, I thank God, the author and vision of my academic life, for allowing me to serve Him through scholarship.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

Recent years have witnessed a significant growth of interest in mobile code, particularly in the form of active contents (e.g., web-browser applets), also known by some as code-on-demand [31]. A key factor in this growth has been the development of suitable security mechanisms for the protection of host computer systems against the threat of executing untrusted code. As a distributed system architecture, a mobile code system usually involves two (or more) processes, namely, a *code producer process* (e.g., a web server process such as `httpd`) and *a code consumer process* (e.g., a web browser). Code migration occurs when the producer process sends to the consumer process an open program (e.g., a Java applet), which describes side effects to be produced on the consumer side (e.g., accessing a local file). Upon arrival at the destination, the program is then dynamically linked into the consumer process's address space, with its open variables bound to resources owned by the consumer process. Execution of the linked program thus produces the desired side effects on the consumer process. Such an arrangement gives rise to serious security threats. If there is no control on the kind of mobile programs that can be executed in the consumer process, then arbitrary side effects might compromise data confidentiality, system integrity and resource availability.

The Java programming language [14] and its associated support technologies have achieved considerable success through a strong protection mechanism implemented within the Java Virtual Machine (JVM). As Java bytecode is downloaded from an untrusted origin, the JVM subjects it to a verification step [130, Chapter 5] in order to ensure that it cannot affect the host machine in an undesirable way. Specifically, the bytecode verifier performs dataflow and structural analyses to guarantee that untrusted bytecode can be linked into the JVM without producing type confusion. As some authors have pointed out, Java's access control mechanism, namely, the security manager, is protected by the type system [49]. So long as downloaded bytecode obeys the typing rules of Java, the security manager should be tamperproof.

This work represents a critical rethinking of the existing verification architecture of the JVM from the point of view of a software engineer sensitive to the specific security requirements of mobile code systems. A thorough introduction to the issues motivating this research is given in the following sections.

1.1.1 Stand-alone Verification Module

Relying on a link-time verifier to protect a host computer system has the problem that the verifier itself may be flawed. If so, designers of malicious code may well be able to exploit the vulnerability to produce type confusion. In fact, several security breaches have been discovered in major Java implementations [49, 135, 107, 108, 157]. These vulnerabilities may be attributed, in part, to the inherent complexity of bytecode verification, involving both dataflow and structural analyses.

Additional complexity in verifier implementation may arise through the combination of verification in an incremental process with lazy, dynamic linking. This complexity becomes manifest in two problematic architectural features of the Sun JVM:

1. **Interleaved logic.** The Sun implementation of JVM interleaves bytecode verification and loading. Java programs are composed of classes, each being loaded into the JVM separately. In the middle of verifying a class X, a new class Y

may need to be loaded in order to provide enough information for the verification of class X to proceed. For example, in order for the verifier to make sure that a method may throw an exception of class *ArithmeticException*, it must check whether *ArithmeticException* is a subclass of the class *Throwable*. As a result, the loader has to be invoked to bring in *ArithmeticException* and all its superclasses. Moreover, since the loader cannot trust the bytecode of *ArithmeticException* (and its superclasses) to be well-formed, part of the verification work must be carried out by the loader. As a result, verification and loading logic are interleaved in the Sun JVM.

2. **Delocalized implementation.** The Sun bytecode verifier has a four-pass architecture. Pass one is the verification logic performed by the loader. Passes two and three, performed by the bytecode verifier at class preparation time [130, Section 5.4.2], check for the well-formedness of bytecode files and carry out dataflow analysis to type check methods of the underlying classes. Pass four is invoked at run time, whenever symbolic references need to be resolved. Consequently, security checks are scattered throughout the run-time system, again adding complexity to the task of analyzing the verification logic.

In the program understanding literature, it is well known that interleaving and delocalized program plans lead to programs that are difficult to comprehend [169, 126]. This so-called "scattershot security" [135] adds considerable complexity to the task of implementing, validating and maintaining a reliable verifier.

Nevertheless, one may understand the rationale for current JVM architectures by considering the need to accommodate a lazy, dynamic linking strategy. Such a strategy seeks to defer expensive computations that may never be needed. For example, a class may be parsed but not further analyzed when only its interface is needed (pass one). Subsequently, its internal structure may be checked when code is linked in (passes two and three), but external references may be left unresolved in the event they are not needed. Finally, these external dependencies may be resolved individually as necessary at run time (pass four). Although such a strategy is not required by the JVM specification, the performance advantages should be easy to understand, particularly for classes with strong static coupling but weak dynamic coupling.

The above analysis reveals a software engineering challenge that is common to all dynamically-linked languages with both security and efficiency concerns. In particular, for mobile code systems which incorporate a protection mechanism based on link-time, static analysis, one has to determine how loading, verification, and linking interact with each other so that the following goals are achieved simultaneously.

1. **Laziness**: loading, verification, and linking can be deferred as long as possible.
2. **Safety**: all necessary verification checks are performed before any code is executed.
3. **Comprehensibility**: the resulting system architecture can easily be understood and thus verified.

As described previously, an *ad hoc* implementation of *laziness* dramatically increases the interleaving and delocalization of program plans within the system. This degrades *comprehensibility*, which may in turn lead to the loss of *safety*. A well-designed mobile code architecture should achieve the goals of safety and comprehensibility by localizing all the security-related code into a *stand-alone verification module* free of loading and linking logic. In particular, it should allow one to *specify*, *craft*, *understand*, and *evaluate* the mobile code verifier as an individual engineering component, independent of the loading and linking procedures.

1.1.2 Distributed Verification

As mentioned above, the verification of incoming, untrusted bytecode is performed by the JVM at link time. I call this protection mechanism, in which a static code verification procedure is invoked dynamically by the runtime environment, *proof-on-demand*. Proof-on-demand is conceptually simple. It allows the JVM to take full responsibility for assuring type safety even in the presence of dynamically generated code — a feature essential for implementing protection mechanisms based on dynamic code rewriting [59, 201, 202, 215, 220, 167, 168]. However, proof-on-demand imposes a considerable computational burden on the JVM. The *link-time overhead* is significant enough that some authors hyperbolically compare it to a denial-of-service attack [135, p. 110]. Compounding these concerns, the architectural complexity of link-time bytecode verification also adds significantly to the JVM's *memory footprint* [193, Sec. 5.3.1].

Future computational platforms will likely include a vast array of small information appliances that have limited computational resources and demanding response-time requirements. Downloaded mobile code will continue to be popular to provide short-lived system extensions[1]. With its stability and widespread acceptance, the Java platform — and specifically realizations thereof based on the Connected, Limited Device Configuration (CLDC) specification [193] and the Connected Device Configuration (CDC) specification [194] — will likely become a major infrastructure for hosting mobile programs in small devices[2]. In these contexts, however, the high resource requirements and architectural complexity of proof-on-demand implementations may become intolerable. The CLDC specification has hence rejected the proof-on-demand approach.

To address these issues, some or all of the verification burden may be offloaded to parties other than the mobile code hosting environment. This gives rise to a *distributed verification system*, in which a mobile code runtime environment shares some or all of its verification burden with certain remotely located facilities. Each facility interacts with the hosting environment by means of a *verification protocol*. A distributed verification system may in fact employ distinct verification protocols for different code units provided that an overall framework for protocol interoperability is defined. An individual verification protocol is thus a fixed scheme that orchestrates the communication and division of labor among the parties involved in the distributed

[1] For example, consider the mobile code language WMLScript [218] for the Wireless Application Protocol.

[2] Consider, for example, the Java TV API, the Web Services API, Mobile Media API, just to name a few.

verification of a code unit. For example, proof-on-demand is a trivial verification protocol that assigns the entire verification burden to the host environment.

In a distributed verification system, the party hosting the mobile code runtime environment is called the *code consumer*. The party responsible for construction and distribution of mobile programs is the *code producer*. Code producers and consumers interact in various ways to define a verification protocol. As an alternative to proof-on-demand, two families of verification protocols have been proposed in the related literature.

1. **Self-Certifying Code.** The first protocol family involves annotating the untrusted code to make it self-certifying. This approach is exemplified in the work on *proof-carrying code* [144, 143, 44]. The protocol proceeds as follows. (i) The code consumers, or possibly an authority representing them, publish a safety policy in the form of a verification-condition generator. Given any mobile program, the generator computes a verification condition that must be shown to be true if the code is to be accepted as safe by consumers. (ii) To distribute a program, a code producer computes the verification condition from the code, proves the condition, and then attaches the proof to the program code when it is distributed. (iii) Upon receiving a mobile program, a consumer recomputes the verification condition, and then checks if the attached proof indeed establishes the verification condition. Execution is granted if proof checking succeeds. Since proof checking is often substantially easier than proof generation, this protocol induces less link-time overhead than proof-on-demand. Furthermore, since proof generation may now be performed once and for all on the producer side, difficult-to-prove safety properties may consequently become affordable.

   In application to Java, the essential idea behind proof-carrying code is that the code producer can annotate a mobile program with static analysis results, so that a consumer may use the annotations to avoid performing a full bytecode verification. This idea has been applied to the verification of Java bytecode in various forms [165, 119, 44], and has further been incorporated in the stack map method of the CLDC specification [193, Sec. 5.3].

2. **Signature-based Methods.** A second family of distributed verification protocols is based on a very efficient and well-understood mechanism, namely, signature checking. Execution is granted to code that is signed by a trusted party.

   A major objection to these protocols is that, unlike a proof (or other kinds of annotations), the semantics of a signature may not be well defined. Thus, there may be no protection against the possibility that signing authorities miscertify. Moreover, celebrity is required in the certification of mobile programs, making it hard for non-established developers to inspire trust.

   These objections are nicely addressed by a protocol which I call *proof delegation* [54, 55]. The protocol proceeds as follow. (i) The code consumers, or more likely an authority representing them, publish a safety policy in the form of a static program analyzer that checks if a given mobile program is safe. The analyzer is encapsulated in a trusted coprocessor, for example, having the form factor of a PCMCIA card or a PCI card [101]. Attempts to physically tamper with the encapsulated analyzer or to extract the private encryption key in the hardware will render the hardware dysfunctional, or perhaps clear its memory [63]. The hardware is then distributed to code producers. (ii) To distribute software, a code producer submits mobile programs to the trusted program analyzer, which verifies the safety of the code, and digitally signs it. (iii) Upon arrival at a consumer site, the signature attached to the program code will be authenticated. The bytecode verification of the proof-on-demand protocol is replaced by a simple and efficient signature-checking primitive. Using trusted coprocessors, proof delegation physically binds the signature to the formal properties enforced by a static program analyzer, thereby giving a well-defined semantics to the signature[3].

In order to support distributed verification protocols in a mobile code system as complex as the JVM, two further issues must be addressed:

[3] When combined with a public key management infrastructure, signature-based verification protocols also enable a very flexible configuration management solution, in which software releases known to be flawed can be disabled remotely [54, 55].

1. **Conditional Certification.** When a Java classfile is verified remotely, it is only checked against the classes on the *producer side*. However, Java type safety is a link-time notion, and a classfile is safe only if it is checked against the loaded classes on the *consumer side*. For example, suppose a classfile is safe only if it can be shown that class X is a subclass of class Y. Suppose further that this relation does not hold on the consumer side. A malicious code producer could forge a class hierarchy in which the subclassing relation holds, and then use it to trick a remote verifier into certifying the classfile in question. Trusting the certificate generated by the remote verifier, the consumer executes the code in the classfile, and type confusion happens as a result. To protect against such attacks, a conditional semantics for certificates is needed. That is, a conditional certificate guarantees that a classfile is safe *if specified external dependencies are further validated on the consumer side at link time*. This issue is especially pressing in the case of signature-based verification protocol: even though the external dependencies may be computed on the fly by the code consumer, the computation itself may have a complexity comparable to that of self-certifying code, thereby nullifying the efficiency advantage of a signature-based verification protocol.

2. **Protocol Interoperability.** A Java developer may use some off-the-shelf components, and write "glue" code to orchestrate their interaction. A possible scenario may be that the prefabricated components are already certified using efficient signature-based protocols, while the home-grown connection code is certified by CLDC-style stack maps. A JVM hosting this program will not only need to be fluent in both signature-based and on-demand protocols, but also need to *combine the two different kinds of certificate (signatures and stack maps) when assessing the safety of the whole program*. What is needed, then, is a mechanism to hide the details of a code unit's certificate, and examine only its *certification interface*, which offers us a safe mechanism for combining certificates.

The current architecture of the JVM offers no support for addressing the two issues above. Neither the Java classfile format [130, Chapter 4] nor the Java Archive (JAR) [192] file format offers provision for expressing the conditional semantics of a certification. In essence, the existing mobile code transport infrastructure in the Java platform lacks a way to express an explicit certification interface. Without such an interface, it is difficult to work through the complex interdependencies between verification and dynamic linking in order to support the interoperability of verification protocols.

1.1.3 Augmented Type Systems

Future systems will likely see additional forms of run-time verification to provide enhanced levels of protection. As the pervasiveness of mobile code hosting environments increases, so too do the vulnerabilities and the potential consequences of these vulnerabilities. To counteract this, attention will turn to safety properties that go beyond simple "type safety" in ensuring system security. These *application-specific* safety properties are usually formulated as augmented type systems on top of the base type system of a mobile code language. While the literature about such augmented type systems is vast, two particularly interesting bodies of work are summarized here:

1. **Information Flow Control.** The US military's *multilevel security model*, in which documents are classified into a finite set of security levels such as *unclassified*, *restricted*, *confidential*, *secret*, and *top secret*, is an incarnation of the more general security model proposed by Bell and La Padula [20, 124]. Under such a model, a subject may only read objects with classification level no higher than its clearance, but may only write to objects with classification level no lower than its clearance. Information is always unidirectionally flowing from low classification source to high classification destination. Denning [50, 52, 51] first applied this idea to the control of information flow in high level programming languages through static analysis. Subsequent developments have been constantly reported [171], among which the work of Volpano and Smith [213, 209, 208, 186, 210, 187, 211, 212, 207, 184, 185] has recently attracted considerable attention from the mobile code community. They defined

an augmented type system on a prototypical high level imperative programming language, so that programmers may decorate a variable by a discrete security level. They have proven a form of *noninterference* property [81], so that, in a well-typed program, the values of more sensitive variables never "interfere" with the values of less sensitive variables. Realization of such a type system in a mobile code programming language will address a significant aspect of confidentiality in mobile code systems.

2. **Alias Control.** Reasoning about side effects is difficult in object-oriented systems in which writable aliases could be created in an unrestrained manner [100]. Augmented type systems have been proposed to control the effect of aliasing. They achieve this by adopting one or both of the following strategies:

    - **Alias prevention:** Alias creation is avoided by either using unique types [17, 138, 27, 6] or placing constraints on the connectivity of the object graph [100, 8, 40, 6].
    - **Access control:** Side effects resulted from aliases is controlled by either tagging aliases to be read-only [100, 122, 182] or imposing other forms of access control [121, 150, 28].

    As these type systems are effectively access control constraints, they could be applied to enforce safety policies [206, 26, 25].

One critique of the research mentioned above is that the notion of type safety is often formulated as a compile-time property, administrated by the code producer, performed against source programs at the time of code generation. However, as mentioned before, in a mobile code environment in which code units bind via lazy, dynamic linking, type safety is in fact a link-time notion. Code units that are checked to be type safe within the compilation environment may no longer be type safe when they are linked against the code units found in the mobile code hosting environment. For augmented type systems to become a viable mobile code protection mechanism, type compatibility between individual code units must be enforced at link time. Unfortunately, given the *inherent complexity* of the lazy, dynamic linking process, and its *tight coupling* with the static type verification component, the programming cost is likely to be prohibitive if one is to augment the existing type checking procedure of a production mobile code environment such as the JVM. This explains why it is very rare to see any of the mentioned work implemented in a realistic mobile code system. In summary, the lack of modularity in the verification procedure prohibits a mobile code system from being extended to incorporate alternative protection mechanisms that are based on link-time static analysis.

1.2 Thesis Statement

Despite the apparent diversity of the three problems discussed in the previous section, they share a common crux: the lack of modularity in the verification architectures of mobile code systems. Consider the motivation behind the adoption of Pluggable Authentication Modules (PAM) for modularizing authentication services. So long as an application is written into the PAM API, system administrators can conveniently replace its out-dated authentication technology by a more modern one (e.g., replacing the use of the `/etc/passwd` file by a shadow password file), adopt an alternative authentication protocol (e.g., strong authentication with Kerberos or one-time passwords), or even authenticate with alternative semantics (e.g., biometric authentication). Thanks to modularity, these improvements can be injected into an application without changing a single line of its code. What PAM does to authentication services, a "*Pluggable Verification Module*" architecture can do to verification services. Modularization allows the verification service to evolve independently from the mobile code hosting environment, provides freedom in the adoption of alternative verification protocols, and supports the introduction of new verification semantics.

This dissertation advocates the adoption of a language-independent architecture for building dynamically-linked mobile code systems in order to address the issues of stand-alone verification modules, distributed verification, and augmented type systems. By design of this architecture, the verification logic of the run-time environment is localized in a stand-alone module fully decoupled from loading and linking, while the laziness of dynamic linking is preserved. To achieve this, the verifier eschews the loading of classes to validate external dependencies. Instead, it converts each dependency into a *proof obligation*, which constitutes the safety precondition for endorsing that dependency. Each proof obligation is scheduled to be discharged when the *linking primitive* responsible for materializing the said dependency is executed by the run-time environment. The run-time environment is responsible for tracking and discharging proof obligations, and for scheduling the execution of linking primitives according to a fixed *linking strategy*. I coin the term *Proof Linking* to refer to this modular verification architecture.

The goal of this study is to bring concrete evidence, in the context of the Java programming language environment, to the following theses:

TS1 *(Modeling Adequacy and Soundness)*: The Proof Linking architecture can be instantiated to adequately model the semantic complexity of a production mobile code system, and to do so in a provably sound manner.

TS2 *(Implementation Feasibility)*: The Proof Linking architecture can be feasibly realized to provide support for stand-alone verification modules, distributed verification and augmented type systems.

1.3 Dissertation Overview

The rest of this dissertation is structured as follows:

Chapter 2 *Viewer Discretion*

This chapter surveys the security requirements and protection mechanisms for existing mobile code systems. It paints an intellectual backdrop for subsequent chapters, and identifies works that this research converse with. This chapter can be read independently of the rest of the dissertation.

Chapter 3 *The Proof Linking Architecture*

This chapter describes the key components of the generic Proof Linking architecture. It specifies three formal correctness conditions, namely, Safety, Monotonicity and Completion, to which any language-specific instantiation of Proof Linking should conform. A brief description of how the architecture addresses the issues of stand-alone verification modules, distributed verification and augmented type systems is also given. A number of concrete research problems are identified.

Chapter 4 *Lazy, Dynamic Linking*

This chapter presents a Java instantiation of the Proof Linking architecture that accounts for the complexity of lazy, dynamic linking. The three correctness conditions introduced in Chapter 3 are shown to be preserved in this instantiation.

Chapter 5 *Multiple Classloaders*

This chapter introduces another level of complexity into the Proof Linking model presented in Chapter 4, namely, that of the presence of multiple classloaders. Again, the correctness conditions were shown to be preserved.

Chapter 6 *The Aegis VM*

This chapter presents a realization of Proof Linking in an open source Java Virtual Machine, the Aegis VM. It describes the design and implementation of an extensible protection mechanism, Pluggable Verification Module, which embodies the Proof Linking architecture. The data structures and programming interfaces for handling proof obligations and commitments are described.

Chapter 7 *Application: Java Access Control*

The generality and utility of the framework presented in Chapter 6 is evaluated in a specific verification domain, namely, that of the augmented type system JAC [122].

Chapter 8 *Conclusion*

This chapter compares and contrasts Proof Linking with related works, summarizes the contributions and limitations of this research, and highlights any future research directions.

Chapter 2

Viewer Discretion

2.1 Introduction

Mobile code computation is a relatively new paradigm for structuring distributed systems [31]. Mobile programs migrate from remote sites to a host, and interact with the resources and facilities local to that host. This new mode of distributed computation promises great opportunities for electronic commerce, mobile computing, and information harvesting. There has been a general consensus that security is the key to the success of mobile code computation. In this chapter, the issues surrounding the protection of a host from potentially hostile mobile programs are surveyed.

Decades of research in operating systems and computer security have provided significant experience and insight into the nature of system security. Yet, the advent of mobile code systems has presented new security challenges that push the boundaries of previous security technologies. In this chapter, the distinctive security challenges of mobile code computation are captured in three security requirements, namely, the establishment of *anonymous trust* (establishing trust with programs from unfamiliar origin), *layered protection* (establishing protection boundaries among mutually-distrusting components of the same process), and *implicit acquisition* (coping with the implicit nature of mobile code acquisition).

This chapter also surveys existing approaches to protection in mobile code systems. Mobile code system protection mechanisms are classified into four categories, namely, *discretion*, *verification*, *transformation*, and *arbitration*. Each category is evaluated by the degree to which they address the security requirements of mobile code computation.

The chapter begins with an introduction to mobile code systems (Section 2.2). Section 2.3 describes the distinct security challenges of mobile code computation. Section 2.4 surveys four major categories of protection approaches in existing mobile code systems, namely, discretion, verification, transformation, and arbitration.

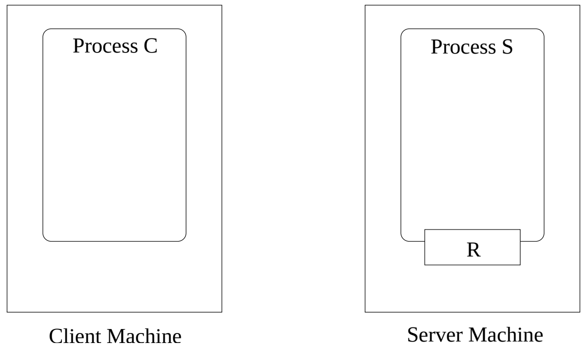
2.2 The Nature of Code Mobility

2.2.1 Code Mobility

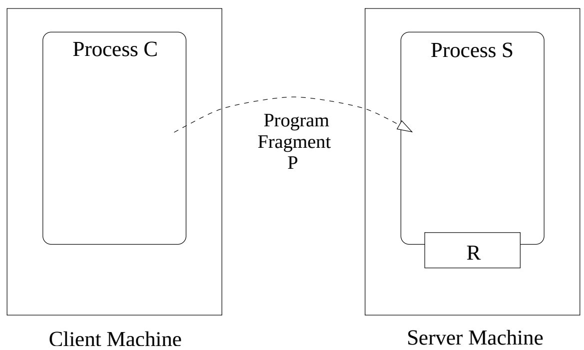
Mobile code is an architectural paradigm for structuring distributed software systems [31]. In general, one can always construct distributed systems by writing socket code using a client-server architecture. What is interesting about mobile code systems is the notion of *code mobility*: instead of simply passing data messages, communicating processes in mobile code systems exchange program code. Here, it is wise to distinguish between *strong mobility* and *weak mobility* [46]. Strong mobility is the mobility of *computation*: an entire thread of computation (or a group of threads), including both its code and its state (e.g., binding environment, stack, etc) are transported. Weak mobility describes the motion of code only. For instance, various forms of active contents (e.g., Java applets [14]) are examples of weak mobility.

Weak Mobility

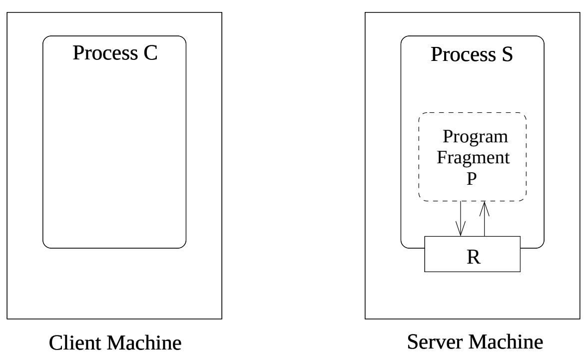
A weak mobility system has a typical architecture as depicted in Figure 2.1. A client process C (or the *code producer*) and a server process S (or the *code consumer*) run in two different machines. The server process S has access to some resource R on the server machine (Figure 2.1(a)). Resource R could be a data structure, some service



(a) Before



(b) Migration



(c) After

Figure 2.1: Weak Mobility

routines, or simply computer cycles. Attempting to access resource R in the server machine, the client process C sends a program fragment P to the server process S (Figure 2.1(b)). Upon arrival, code fragment P is dynamically linked into process S. S then invokes the code in P (Figure 2.1(c)), enabling P to access resource R on behalf of process C. The result of accessing R is communicated back to C if necessary. To make the above abstract description more concrete, think of S being a web browser process (e.g., `netscape`), with access to local windowing system and terminal display (resource R), and think of C being a web server process (e.g., `httpd`). The code fragment P could be a Java applet.

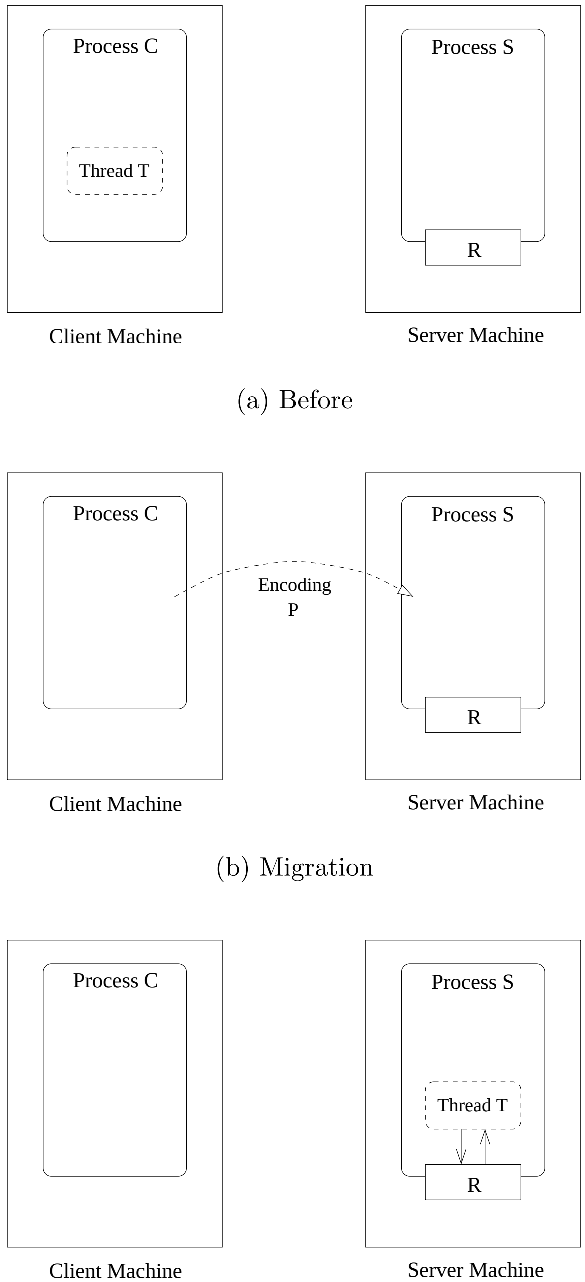
Carzaniga *et al* [31] identify two variants of the above architecture:

Code-on-Demand: The server process S initiates communication. It requests client C to send it program fragment P. Various forms of active contents, including Java applets [14], ActiveX controls [137], and JavaScript [147] are commercial examples of code-on-demand.

Remote Evaluation: The client process C initiates the communication. It sends code fragment P to server S, requesting S to evaluate P on its behalf. Stamos and Gifford [189] were the first to envision this architecture. MIME Tcl extension [24], active packets [98, 139, 99] and various forms of active/intelligent disks [3, 114, 163] are examples of remote evaluation.

Strong Mobility

Strong mobility is commonly known as mobile agents [31]. A typical mobile agent architecture is depicted in Figure 2.2. A group of coordinating threads T is originally running in a process C on the client machine, and accessing the resources therein (Figure 2.2(a)). At a certain point, thread group T attempts to access resource R that is owned by a process S running on a remote machine. To do so, thread group T requests its parent process C to transport it to remote process S. Process C suspends T, and then marshals T into an encoding P that encapsulates both the code T is executing and the dynamic state of T. The encoding P is then sent off to process S



(a) Before

(b) Migration

(c) After

Figure 2.2: Strong Mobility

(Figure 2.2(b)). Upon arrival, the encoding P is dynamically linked into process S, and the execution of T is resumed, thereby allowing T to interact with resource R (Figure 2.2(c)). T remains in S until there is a need to access remote resource again.

Another motivation for agent migration is to create opportunities for multiple agents to interact with each other at a rendezvous point. Typical applications include online auction [173, 226, 13, 95] and other e-commerce domains [134].

Generalization

Let us ponder again on the two architectures laid out in Figures 2.1 and 2.2. The motivation of exchanging code in both cases is for an external party to access the remote resource R located in the server machine. In every mobile code system, there is a computing platform, called a *host*, that offers external access to a selected set of computing resources that it owns. The resources are managed by a process called a *computing environment (CE)*, which defines a controlled interface between the resources and external parties who want to access the resources. In the previous example, process S fills this role of a computing environment.

When an external party attempts to access the facilities managed by the computing environment, it sends off a *mobile code unit* which corresponds to either a code fragment (as in the case of weak mobility) or an encoding of some threads (as in the case of strong mobility). The program code in the mobile code units will then be dynamically linked into the computing environment process. One or more threads will be started to carry out the computation prescribed by the downloaded mobile code unit, that is, to access the resources of interest on behalf of the external party. These threads might in turn start some other threads. Together these threads constitute a recognizable application program that carry out a single job. Such a group of threads is called an *execution unit*. An execution unit is a direct analogue of a process in a traditional operating system, just as the computing environment is an analogue of the operating system itself. From a security perspective, no execution unit should be allowed to access host resources except in conformance to the protocol imposed by a computing environment. Notice that a host may serve multiple computing environments, while each computing environment may in turn host one or more execution units.

Consider Java as an example. The Java Virtual Machine (JVM) is the computing environment of Java. As an interpreter, it creates a *sandbox* in which interaction between Java threads and the underlying operating system can be controlled. Multiple JVMs can coexist at a single host. Migration occurs when a Java classloader retrieves mobile code units, called Java classfiles, from, say, a remote web server process. The classfiles are verified and then dynamically linked into the JVM. Finally, the JVM spawns off an applet thread that invokes the linked code. An applet is therefore an execution unit in Java.

2.2.2 Motivation of Code Mobility

Motivation for adopting the mobile code paradigm has been surveyed in great detail in the literature [120, 35, 36, 31, 199]. The following list summarizes several representative applications:

1. **Real-time interaction with remote resources:** Most computing resources, like databases, file systems, or even physical displays, are not transportable. If such resources are located at a remote site, then computation that requires *real-time interaction* with the resources has to happen where the resources reside. Code mobility allows one to prescribe the location of computation, so as to make real-time interaction possible. For example, active contents like Java applets prescribe interactive presentation that is to be rendered on the browser side.

2. **Reduction of Communication Traffic:** Mobile computers (e.g., hand-held computers or intelligent mobile phones) usually interact with servers through unreliable, low-bandwidth, high-latency, high-cost networks. Mobile programs become an attractive alternative because network traffic can be reduced by migrating the client program to the server side, thus avoiding the potential cross-network communication bottlenecks.

3. **Customization and extension of server capabilities:** Valuable hardware resources are usually managed by server software (e.g., an operating system). Such server software usually defines access policies that are extremely general, and tend to ignore the specific needs of individual clients. Recently, various proposals have been made to allow application-specific extension code to be downloaded dynamically into server software, so as to customize its access policies to meet the specific needs of clients. Typical examples include extensible operating systems [22, 58, 110, 178], active networks [198, 34], active/intelligent disks [3, 114, 163], and many others.

4. **Avoiding distribution of state:** In traditional client-server applications, the state of computation is distributed among servers and clients. As a consequence, it is difficult to maintain consistency of the distributed states, and to articulate the correctness of the computation. Mobile code systems localize computation states in a single process. They offer a better abstraction that makes the crafting of distributed software a more manageable task.

Chess *et al* [36] fairly pointed out that any application that can be crafted under the mobile code paradigm can also be structured as a client-server application. However, mobile code systems offer many engineering advantages that its client-server counterparts may lack. Recently, Carzaniga *et al* [31] propose an abstract model for evaluating the potential benefit of adopting various mobile code paradigms. The result suggests that the advantages pay off only for certain kinds of application domains.

2.2.3 Code Mobility, Dynamic Linking, and Binding Environment

The essence of code mobility is the ability to transport some resource-accessing mobile code units from one host to another, and then execute those code units on the resource-bearing host. In order for the resource-accessing mobile code unit to do so, it must contain some free variables that refer to the resources it is interested in. Therefore, upon arrival, all mobile code units must go through a phase of integrating into the computing environment, and that integration usually takes the form of dynamic linking, that is, linking the free variables in the mobile code units to the actual resources owned by the computing environment.

Viewing from a programming language perspective, mobile code units are basically code that contains free variables. A computing environment is in fact a *binding environment* that defines bindings for the free variables in mobile code units. Code migration, then, is the dynamic specification of the binding environment in which an open program is to be executed. This view, first advocated by Queinnec and De Roure [161], can be illustrated in the following way.

The signature of a typical metacircular Scheme interpreter[1] [2] has the following form:

```
(eval form env)
```

Given a Scheme form (an open program) and a binding environment, the `eval` interpreter computes the value of `form`, using the bindings in `env` whenever free variables in `form` are to be resolved. If we see a computing environment as a binding environment that supplies bindings for the free variables occurring in a mobile code unit, then code mobility can be viewed as the explicit identification of the binding environment in which a form is to be evaluated. It is this dynamic linking aspect that makes computation "occur at another site". In this manner, remote evaluation can be understood as the evaluation of a form in a remote environment:

```
(eval form network-locator-for-environment)
```

Here, `network-locator-for-environment` names the binding environment in which `form` will be evaluated. Code-on-demand can be understood similarly:

```
(eval network-locator-for-program local-environment)
```

[1] A metacircular Scheme interpreter is a Scheme interpreter written in the Scheme language itself.

Here, `network-locator-for-program` names the remote program that is to be brought in for execution.

Mobile agents are not easily understood in such a framework. In the studying of code sharing via the internet, Queinnec and De Roure [161] propose the following special form that captures various forms of code mobility:

```
(import (v1 v2 ...)  env form)
```

The `import` special form composes an environment, and then evaluates `form` in this environment. The target environment is composed by taking the current environment (the lexical environment `import` is in) and then overshadowing the definition of variables `v1`, `v2`, ... by those in the first-class environment `env`. If `env` represents a remote host, then the target environment will be composed of both the bindings in the remote host and the current state of computation. Strong mobility is thus given a very clean model.

Realizing that dynamic linking and dynamic specification of binding environment are the underlying reality of mobile code computing is crucial for the following reason: there is a very subtle analogy between a protection domain and a name space (a binding environment).

2.2.4 Viewer Discretion: The Security Challenge

PostScript files [196] could be seen as a form of mobile programs [199]. As a stack-based, page-description language, it offers a very compact description of documents. When PostScript documents are sent to printers, the documents are interpreted to generate printout. This setup off-loads some of the document rendering burden from the printing hosts to the printers. Because of its simplicity, one does not usually expect this incarnation of remote evaluation to go wrong.

As PostScript gradually gained its popularity as a portable document description language, it became a standard medium for document distribution on the internet. Recognizing it as a MIME type, web browsers automatically spawn off a viewer process as a Postscript file is downloaded. In 1995, the CERT coordination center discovered that some older versions of postscript viewers allow Postscript programs to access the local file system in an unsafe manner [32]. This presents a potentially serious threat to the web browsing system, for the vulnerability can be abused by malevolent programmers "intentionally embedding commands within an otherwise harmless image so that when displaying that image the PostScript viewer may perform malicious file creations or deletions." [32]

With the emergence of *rogue applets* [123, 135] and other forms of malicious active contents, users of mobile code systems are now more aware of the security threats associated with mobile code computation. A rogue mobile code unit may overwrite valuable data on local disks, covertly transmit private information to another party, hang the hosting browsers, or masquerade as another trusted application.

Unfortunately, mobile code units may originate from unfamiliar sources, making it difficult for users to determine if a given code unit should be granted execution rights. A naive response would to be turn off all mobile code capabilities. This option, however, is not desirable due to the fact that increasingly software infrastructures are built around mobile code technologies. The crux of the problem then is not one of avoiding mobile code technologies, but of protecting users from unsafe mobile programs.

Security is also the most challenging aspect in the crafting of mobile code hosting environment. For one, it is difficult to articulate exactly what it means for a mobile code system to be secure. The next section delineates the distinct security needs of mobile code systems.

2.3 The Distinct Security Requirements of Mobile Code Systems

There are two classes of security issues in mobile code computing:

- *Host security* is concerned with the protection of a host from untrusted mobile programs, and with the avoidance of mutual interference among execution units.

- *Application security* is concerned with the assurance of correctness and confidentiality for the computation that is delegated to a remote host. When an untrusted host carries out a computation on behalf of a client, the host may maliciously corrupt or expose the internal state of the client's execution units. Application security goes by other names like *code security* [210].

This chapter is mainly devoted to the exploration of issues concerning host security. Readers interested in application security may consult [61, 204, 106, 105, 39].

Viewing from the perspective of host security, mobile code systems share many similarities with an operating system. In particular, security issues arise in both kinds of systems when they attempt to address the need of *multiprogramming*, and specifically, the need for safe resource sharing. A multiprogramming system allows multiple execution units to coexist on a host. It is desirable to ensure that no single execution unit has total monopoly of the host's resources. As a result, individual execution units must be guarded from interfering with each other, and from undesired exploitation of the host's resources and facilities. To this end, various protection mechanisms are built into an operating system. Silberschatz and Galvin [180, p 431] understand the entire business of protection as one of *providing safe sharing of name spaces*:

> *Protection was originally conceived as an adjunct to multiprogramming operating systems, so that untrustworthy users might safely share a common logical name space, such as a directory of files, or share a common physical name space, such as memory.*

Given the common security goal of safe resource sharing, it is then important to question why the existing protection mechanisms in operating systems cannot be directly transplanted to mobile code systems to address similar security needs. In other words, we need to articulate in clear and precise terms an answer to the following question:

Question of Distinction: What makes the security needs of mobile code computing different from those of traditional multiprogramming operating systems?

This section attempts to answer the above question.

Sections 2.3.1–2.3.2 give a brief survey of the security requirements and protection mechanisms in traditional multiprogramming systems. Details can be found in standard textbooks on operating systems [180], computer architecture [97], computer security [23] and network security [113]. Readers already familiar with these topics may skip the mentioned sections. Section 2.3.3 describes three distinctive challenges in mobile code security. Section 2.3.4 then articulates why software-based solutions to such challenges should be sought.

2.3.1 Ingredients of Host Security

There are three aspects to the security of a computing system:

Integrity: System resources should be protected from unauthorized modification, deletion, or other means of tampering.

Confidentiality: Sensitive information should be protected from leaking through unauthorized channels. Confidentiality goes by other names such as *privacy* or *secrecy*.

Availability: The computing system should be protected from interference that affects its normal operation and availability of service.

In order to establish and evaluate the security of a computing system, one has to refine the above criteria, and lay out exactly what the security requirements are in concrete terms. A *policy* is a well-defined, consistent, and implementable statement of the security one expects the system to enforce. It is an administrative decision about what should constitute a breach of security.

There are several well-known types of *threat* that may compromise integrity, confidentiality or availability.

Corruption. Execution units may modify or erase important data. Others may tamper with the internal state of the system, rendering the system state incoherent.

Typical examples of such attacks include various forms of computer virus that cause damage to host's data. Such attacks compromise the integrity of the system.

Leakage. Execution units may actively release sensitive information to an outside party. Others may engage in data processing activities from which malicious third parties can infer information that is supposed to be classified (i.e., covert channels). For example, an unguarded Java applet may access personal financial records on an unsuspecting PC's hard drive, and then send the information back to the attacker via a socket connection. Such attacks violate system confidentiality.

Denial of service. Execution units may monopolize shared resources like the terminal screen (creating a window so huge that it covers everything on your terminal screen), CPU time (raising its own priority above all other threads), threading services (killing all threads other than itself), etc. Such attacks compromise the availability of the system.

Masquerading. A rogue execution unit may masquerade as a legitimate application by faking the user interface of the latter, thus fooling the users into entrusting it with critical resources (e.g., stealing CPU time for factoring an integer) and data (e.g., asking users for their passwords). Execution units may also pretend to originate from a trusted origin (e.g., various spoofing attacks). An execution unit may even fool the type system by appearing to be of another type (e.g., type confusion in Java), thus gaining access to the internal state of the system. Masquerading is a very subtle form of attack that could potentially lead to the compromising of integrity, confidentiality, and availability.

These types of threat have long been of concern in traditional computing environments. Mobile code systems such as Java are also subject to threats of these kinds [123, 135, 64, 154].

2.3.2 Traditional Protection Mechanisms

Two relevant protection mechanisms in traditional operating systems are *memory protection* and *access control*[2].

Memory Protection

Memory protection seeks to ensure that execution units interfere neither with the execution states of other units nor with the state of the global host. In general, this requires that control can flow outside of the execution unit only through some well-defined interface provided by the computing environment. In short, the goal of memory protection is to avoid the malfunctioning of one execution unit from contaminating other execution units or even the host itself. Memory protection also goes by other names in the mobile code literature, namely, *low level security* [225] or simply *safety* [112].

In traditional operating systems, memory protection is achieved by three mechanisms in combination:

1. Processes are placed in separate address spaces. At run-time, every address reference, be it a data reference or a control transfer, is checked by the hardware to see if the referenced location belongs to the address space of the running process. If not, a memory fault will be generated to halt the process and return control to the operating system.

2. The CPU offers two mode of execution, namely, the kernel mode and the user mode. User processes are always executed in the user mode. Instructions that set the boundary of address space are protected, and can only be executed in the kernel mode. User processes are then protected from redefining the boundary of their address spaces.

[2] In fact, competent operating systems offer other kinds of protection mechanisms, including CPU protection, instruction protection, I/O protection, etc. In a mobile code system, execution units usually execute inside a user process, and thus are already protected from monopolizing the CPU(s), executing privileged instructions, and directly accessing any I/O channels. Therefore, these issues will not be touched on in the remainder.

3. Control flowing outside of the address space must pass through a special interface of the operating system kernel. This is usually achieved by providing a set of predefined system calls accessible by a special TRAP instruction. When a system call is invoked, the CPU switches to kernel mode, and control is transferred to the operating system, which in turn processes the system call on behalf of the user process.

Instead of a simple dichotomy of kernel and user modes, some operating systems provide multiple, concentric *rings* of security levels. Code running in the more trusted rings may access data and code in the less trusted ring, but not vice versa. Less trusted code may transfer control to the more trusted code via special entry points called *gates*. Generally, these arrangements require special hardware and operating system support.

Access Control

In traditional operating systems, resources of the host are modeled as *objects*, while user processes are modeled as *subjects*. A subject can perform *operations* on an object. The permission to perform a certain operation on an object is said to be an *access right*. Security policies are expressed as an assignment of rights to subjects. A *protection domain* is a collection of access rights. A user process acquires its access rights by being associated to a protection domain.

Access rights in a multiprogramming system can be expressed as an *access matrix*. Every row in the matrix represents a protection domain, while every column represents an object. By labeling an entry in the matrix with access operations, one effectively defines what rights are given to a protection domain (row) for the accessing of an object (column). In traditional operating systems, an access matrix is usually implemented in one of two ways. The first approach is the *access control list*. Associated with every system resource is a list of ⟨subject, right⟩ pairs. Whenever a resource is accessed, its associated list is checked to see if the accessing subject is on the list and is granted appropriate access right. A second approach is that of *capability*. A capability is an unforgeable pointer to a system resource. By acquiring the pointer, a subject is then granted the right to access the object. In a sense, capability controls access through visibility — if a system resource is not even visible to a process, then there is no way the process can access it.

In traditional operating systems, access control presumes two related mechanisms: *authentication* and *authorization*. Authentication is the process of establishing the identity of a user. Authorization is the process of translating the result of authentication into decisions of what access right is granted to authenticated users. Under such a system, whether a user has right to perform an operation is largely dependent on his/her identity, or, more precisely, on the operating system's knowledge about him/her.

2.3.3 Distinctiveness of Mobile Code Security

What then distinguishes mobile code computing from multiprogramming in traditional operating systems? Why cannot the traditional protection mechanisms satisfy the security needs of mobile code computing? This work advocates the perspective that the mobile code phenomenon is distinctive in the following way:

> **Distinctiveness of Code Mobility:** Subject only to *time-bounded, automatable checking*, code originating from *any arbitrary source* may be executed in an environment that *exposes access to shared resources*.

Here, the three key phrases are "time-bounded, automatable checking", "any arbitrary source", and "exposes access to shared resources", details of each will be given in the following.

Anonymous Trust

Traditional discretionary access control [172] is based on trusted identities. A user is a known party. Based on one's trust with the user's identity, and based on notions like resource ownership, one then authorizes access to system resources. A direct translation of this idea to mobile code security is to attach to every mobile code unit a digital signature that signifies its origin. In this view, the source of the mobile code unit is treated as a user in a traditional operating system, and authorization of access rights is based on the familiarity of the code unit's origin. This approach works well when the mobile program is developed by a well-known brand name, or when it is sent from a trusted source. Yet, the approach breaks down when the foreign code is written by an unknown author and sent from an obscure origin. The very spirit of WWW computing is that any party can freely share information or active content with others who have access to the internet. It is conceivable that in the future an increasing number of useful mobile programs are going to be developed and distributed by parties unknown to the average users. Security that is based solely on identity cannot account for such phenomena. In such an approach, the only completely safe practice will be to throw the baby out with the bath water, and reject all untrusted code. This difficulty was first articulated by Ousterhout *et al* [153], and then found its full expression in a paper of Chess [37], in which he discusses the fallacy of the "*Identity Assumption*".

> *The most important assumption that mobile code systems violate is:*
>
> > *Whenever a program attempts some action, we can easily identify a person to whom that action can be attributed, and it is safe to assume that that person intends the action to be taken.*
>
> *For all intents and purposes, that is, every program that you run may be treated as though it were an extension of yourself.*

The above discussion illustrates a fundamental challenge in mobile code security:

> **Anonymous Trust:** How can users establish trust for a mobile code unit sent from an unknown origin and developed by an unknown party?

Given the above, security engineers should not rely solely on identity or origin information to authorize access. A competent security infrastructure should accept trustworthy mobile programs even if they are anonymous. Therefore, it is accepted as an axiom that mobile programs from *any* origin will be downloaded into a computing environment:

> **Axiom of Anonymity:** Code from any arbitrary origin will be downloaded.

Two implications follow from the Axiom of Anonymity. Firstly, mandatory access control is particularly relevant to mobile code security. Secondly, with the emergence of the web, *the model of software distribution has changed*. No longer can users establish trust solely by means of brand names and well known vendors, for everyone may now distribute software. It is necessary to put tools and techniques into the hands of this new breed of anonymous programmers so that they can construct software that *inspires* trust in hosts. Devanbu *et al* call such an endeavor *trusted software engineering* [55].

Layered Protection

Another fundamental aspect of mobile code computing is that a mobile code system defines a complete multiprogramming environment on top of the existing operating system. The mobile code computing environment may define its own computing model, maintain its own resources, provide its own set of services, and hence define its own security model. Consequently, it is usually not realistic to simply treat an execution unit as just another process in the operating system, running in just another protection domain. The desire for platform independence (Section 2.3.4) further discourages non-portable adherence to the security model of a particular operating system.

In addition, as one of the users in the underlying platform, a mobile code computing environment may *expose* some of the operating system resources to the visiting execution units. The mobile code security model must honor the security constraints imposed by the operating system.

A typical arrangement in existing mobile code systems is to have the computing environment running as an operating system process, while treating execution units as secondary threads executing within that process. For example, a web browser is a single process containing a Java Virtual Machine, in which Java applets are run. A Java web server is also a single process. Servlets run within the server process as secondary threads, thus avoiding costly CGI-based solutions. Since a process defines both the boundary for memory protection and the protection domain for access control, execution units are protected from exploiting resources outside of the computing environment.

Protection within a computing environment is less trivial. Mobile code units are linked into the computing environment process. Sharing the same address space and protection domain, the execution units and the computing environment process are indistinguishable from the point of view of the operating system, which must rely on process boundary to enforce protection. This creates an embarrassing situation, in which code fragments within a single application process do not trust each other[3]. As a result, the computing environment process has to enforce memory protection and access control[4] within the process itself.

The security model of the operating system and the security model of the mobile code system then form a parent-child relationship. Providing mechanisms for defining child protection domains inside a single process becomes the second fundamental challenge to mobile code security:

---

[3]It is revealing to read a passage in a standard operating systems text [180, p 112]:

> *However, unlike processes, threads are not independent of one another. Because all threads can access every address in the task, a thread can read or write over any other thread's stacks. This [multi-threading] structure does not provide protection between threads. Such protection, however, should not be necessary. Whereas processes may originate from different users, and may be hostile to one another, only a single user can own an individual task with multiple threads. The threads, in this case, probably would be designed to assist one another, and therefore would not require mutual protection.*

The introduction of mobile code computing environment definitely falsifies the above assumption.

[4]Again, one could point out the need to avoid the monopolization of the process's time slice by a secondary task. This is easily achieved if a competent scheduling mechanism is in place. Therefore we will leave the issue here. For those interested in having more control on Java's scheduling mechanism, consult [125, section 8.3].

> **Layered Protection:** How can protection be established among mutually-distrusting execution units coexisting in a single application process?

An alternative articulation of this problem is what Rees [162, p 16] called the *safe invocation problem*:

> *When a program is invoked, it inherits all of the privileges of the invoker. The assumption is that every program that a user runs is absolutely trustworthy. . . . It is remarkable that we get by without secure cooperation. We do so only because people who use programs place such a high level of trust in the people who write those programs.*

What Rees refers to by secure cooperation is the ability to invoke an unfamiliar routine without granting it all the invoker's rights. To achieve this, one needs to be able to hierarchically construct a child protection domain for executing untrusted code within the invoker's process.

Layered protection is a characteristic feature in single-address-space operating systems like OPAL [33] and Mungi [96], and extensible operating systems like SPIN [22], VINO [178], and Exokernel [58, 110]. In such operating systems, untrusted code may be (dynamically) introduced into a privileged protection domain (e.g., the kernel). One wants to avoid the untrusted code units from exploiting the resources in that domain. More recent works [38, 103, 195] from the Operating System community attempt to address the need for *intra-address-space protection mechanisms* motivated by software plug-ins, device drivers and data-driven security threats.

Implicit Acquisition

Code mobility defines a new model of *software acquisition*. Traditionally, acquisition of commercial-off-the-shelf software involves a slow, manual, and explicit process. Alternatives are reviewed and tested. Impact analysis is conducted. Deployment is planned and staged. System administrators know exactly what packages are installed on the system. Potential impact is announced to users.

Software acquisition is completely different in a mobile code system. As De Paoli *et al* [154] put it:

> *Conventional computing paradigms assume that programs are installed and configured once on any and every machine and that these programs only exchange data. This means that a user can make all possible checks over a new program before running it. This assumption, however, is no longer valid for open and mobile environments, such as Java and the Web.*

A mobile code unit may arrive even without the knowledge of the users. Simply by browsing a web page or opening an email will invoke the installation of active content. Acquisition is therefore implicit. It is actually a design goal that the entire acquisition process be invisible to the users. In such an acquisition process, only *automatable checks* are allowed, be it signature checking, program analysis, type-checking, etc. All such checking should take only a *limited* time to complete. It is this time constraint that makes code mobility *a completely new model for software acquisition*. The traditional acquisition process establishes trust gradually. Yet, with the time constraint, a computing environment is forced to establish the trustworthiness of a program without going through the traditional evaluation cycle. In fact, the time spent on trust establishment should be only a small fraction of the brief execution time of the execution unit. Therefore, the third fundamental challenge of mobile code security is the following:

> **Implicit Acquisition:** In the absence of an explicit acquisition process, how can trust be established automatically within a limited time frame?

2.3.4 Software-based Solutions

There is a misconception that hardware-based protection mechanisms are the only way to enforce memory protection [21]. Although some of the security requirements of mobile code systems can potentially be addressed by a hardware solution, the following sections focus on software-based solutions implementing logical separation. This perspective is adopted for the following reasons.

1. **Portability.** Execution units might migrate to a heterogeneous array of computing platforms. Adopting a hardware solution either limits the platforms on which a mobile program can be deployed, or else requires the mobile code units to be explicitly ported to interoperate with the peculiarities of the target platforms. It is therefore preferable to have mobile code protection mechanisms that can be implemented in general-purpose processors, on typical operating systems.

2. **Performance.** Numerous reports [214, 183, 216] confirm that memory protection schemes based on hardware-enforced address space boundaries incur significant performance penalties for cross-boundary procedure calls. Software-based memory protection mechanisms such as Software-based Fault Isolation [214], Proof-Carrying Code [144], and type-safe languages [22] are found to be significantly more efficient than hardware-based mechanisms. Even interpreted languages such as Java are also found to exhibit adequate performance for system programming [164]. All these suggest that software-based solutions can be as adequate as, and at times even more competitive than, hardware-based solutions.

3. **Ease of Experimentation.** A software solution is usually easier to implement and modify. It can then be easily distributed to or reproduced by the computing community, who in turn may iteratively review and improve the solution.

4. **Expressiveness.** Although memory protection naturally invites a hardware solution, it is not obvious how hardware protection can be employed to enforce high-level access control, especially when application-specific security is concerned.

The following discussion will therefore focus on software-based protection mechanisms for mobile code security.

2.4 Protection Mechanisms for Mobile Code Systems

This section outlines four software-based approaches for protection: *discretion*, *verification*, *transformation*, and *arbitration*. The approaches presented here are complementary. In fact, most of the protection mechanisms of existing mobile code systems can be understood as a combination of the four approaches.

2.4.1 Discretion

Discretion refers to protection mechanisms that rely on "tokens" of trust to make security decisions. In particular, it refer to the use of various authentication techniques [136, 177] for establishing trust. Every mobile code unit is associated with some digital signature(s). Whenever a foreign mobile code unit arrives at a host, its signature is authenticated, and a (mechanical) process of authorization will translate the result of authentication into access privileges. In such systems, signature authentication is assumed to be very efficient. Because of its inherent simplicity and the efficiency of signature authentication, discretion-based protection addresses the challenge of implicit acquisition very well. As a result, it has been studied as a general protection infrastructure [60, 104] and is implemented in many existing mobile code systems (e.g., Java [86, 84, 85, 87, 88, 18], Telescript [221, 197], Agent Tcl [90], ActiveX [137, 158], etc).

At the heart of the discretion approach is the *semantics* of the signature. What a signature means determines the kind of access rights granted. In the following, issues surrounding the assignment of meaning to a signature are surveyed.

Denotation of Signatures

A digital signature is an unforgeable token that denotes a security property of the signed code unit. Three potential denotations can be attached to signatures of mobile code units.

Identity/Origin Semantics: The signature of a mobile code unit identifies its author or origin. A computing environment maintains a mapping between known signatures and their associated rights. This method is a direct translation of the traditional discretionary access control found in many operating systems. As discussed in Section 2.3.3, schemes that are based on knowing the owners or authors of programs do not work well in the establishment of anonymous trust.

Authoritative Endorsement Semantics: Signing a mobile code unit means that the signing party endorses the unit as being "safe", usually in an informal sense. In such an approach, some trusted authority will be responsible for certifying mobile code units. A developer submits his/her mobile program to the certification authority before the program's publication. Usually, what it means to be "safe" is defined informally, if it is properly defined at all. In such approaches, a signature signifies nothing more than the endorsement of the mobile code unit by a certain party. Such endorsement has no formal semantics — it cannot be reduced to formally defined security properties. Endorsement is based on trust. Security is then a relative property depending on the extent to which the signing party is trustworthy.

Program-Analytic Semantics: The signature denotes a formal program-analytic property such as type safety or invariance of a particular assertion (program invariant). A signature is attached to the mobile code unit only when the corresponding formal property is found in the unit. The attachment of signature can result from three possibilities:

1. Code is trusted because it is *generated* by a trusted *compiler* [166, 145].

2. Code is trusted because it has been properly *rewritten* by a trusted program *transformer* [214, 178].

3. Code is trusted because it has been *certified* by a trusted program *analyzer* [55].

A program-analytic semantics may be more reliable than informal endorsement, because trust is placed on a formally defined, publicly available program certifying algorithm instead of mere human judgment [158]. Unfortunately, not many security properties have been formalized into program-analytic terms. Memory safety and confidentiality are the rare cases that has been formalized (see Section 2.4.2). Works on the translation of security properties into program-analytic terms will definite further the feasibility of this approach.

There are two inherent challenges to adopting a program-analytic semantics for digital signatures:

- Suppose a flaw is found in a widely accessible implementation of a program certifying algorithm, or, say, such an implementation no longer reflects the evolved security policy. In such cases, mechanism should be in place to revoke the key that generates the signature.

- Signatures must not be forgeable. That is, only a trusted program certifier may sign mobile code units. This problem is more complicated than it seems, especially when such trusted program certifiers are to be distributed into the hands of untrusted programmers. How can one be sure that the certifier is not tampered with?

See Section 2.4.2 for a discussion on how trusted hardware and key-management schemes can be employed to address the above challenges.

Multiple Trust Levels

An authorization procedure maps signatures to a space of meaning. The granularity of control depends on the size of the meaning space. A large meaning space results in finer-grained control. This is illustrated by the following example.

The ActiveX approach [137, 158] relies solely on authentication to enforce security. ActiveX controls are native mobile code. An ActiveX control is transported with a signature that identifies its origin. It is up to the user to decide if the signature is trusted, and, in cases when the foreign control is trusted, it will be granted full access to the host.

Authentication technologies, both cryptographic and non-cryptographic [136, 177], are well established. In addition to its inherent simplicity, the ActiveX approach results in minimal interference and allows execution units to have maximal capability. This represents a property of nearly all discretion-based protection mechanisms: signature checking reduces the need for run-time or link-time interference.

The ActiveX approach grants trusted code with unlimited access to the entire host environment. This represents an *"all-or-nothing"* model of security. In many cases, this inflexibility is absolutely unacceptable. For example, one might want to grant an execution unit rights to read local files as long as it is forbidden to open socket connection to remote sites (and thus disclose contents of local files). Therefore, when talking about security, there are actually different levels of trust. Based on the result of authentication, one might want to grant certain execution units more capabilities, while granting less to the less trusted execution units. A discretion-based protection mechanism must provide fine-grained authorization mechanisms.

The need for fine-grained access control is epitomized by the evolution of the sandbox model in Java [84, 85, 87]. In the 1.0.2 version of the Java Development Kit (JDK), all code is untrusted unless it is loaded from the local disk. Untrusted code is executed in a "*sandbox*" in which access to most system resources is prohibited. In JDK 1.1, signed applets are introduced. Applets associated with a trusted signature are given full access to the system, while the untrusted applets remain in the sandbox. In JDK 1.2, an extensive access domain architecture is incorporated within the security infrastructure, allowing the selective mapping of signatures to collections of permissions. Thus, one may define "*playgrounds*" of different access restrictions, and put applets of various trust levels into appropriate playgrounds.

2.4.2 Verification

A *firewall* [23] is a first degree approximation to the *static verification* approach. A firewall is a computer that sits between a local network and the rest of the global network. It filters packets as they go by, according to various criteria which can be configured. Packets that appear unsafe are rejected before they can do any harm to the protected network.

In the verification approach to mobile code security, security policies are formulated as program analytic properties. Incoming mobile code units must pass through a trusted program analyzer before reaching the computation environment. The trusted program analyzer, usually called a *verifier*, filters out potentially unsafe programs. The execution units that can reach the computation environment are guaranteed to satisfy certain security properties. Usually, there is no further need for the computation environment to provide any countermeasure to enforce the checked properties. Because of this, static verification is an attractive protection mechanism when the program property of interest is expensive to enforce dynamically (e.g., dynamic type checking).

Verification for Memory Protection

To date, the most successful application of the verification approach is for memory protection. This is illustrated in the following three examples.

Typed Intermediate Language. In the Java language [89], memory protection is achieved by the use of a safe intermediate language. Java source programs are compiled into *Java Virtual Machine (JVM)* bytecode [130]. The bytecode representation is specially designed to protect execution units from interfering with each other and from accessing the JVM's internal state. Firstly, the JVM bytecode language is strictly typed. Secondly, pointer arithmetic is not allowed. Therefore, bytecode instructions can only access memory in a type-safe manner. As a consequence, memory protection is reduced to type-checking. All untrusted Java classfiles must pass through a bytecode verifier before they are dynamically linked into the JVM. Since the JVM bytecode is unstructured, data-flow analysis must be used to make sure that the classfile is type-safe. In fact, dataflow analysis within the JVM is not limited to checking type safety, but is also used to check for other safety concerns such as operand stack overflow[5]. Therefore, runtime checks that would otherwise be needed to avoid operand stack overflow and ensure typesafety can be safely avoided.

Proof-Carrying Machine Code. Type systems can capture only certain kinds of security properties, and occasionally require a safe (intermediate) language. To enforce memory safety in a less structured language (e.g., native code), and to provide more expressive means for formulating safety properties, Hoare logic [45] can be used. Hoare logic is an axiomatic system for proving program correctness. Safety properties can be encoded as precondition/postcondition pairs, while verification reduces to the establishment of the postcondition given the validity of the preconditions.

In their work on proof-carrying code[6], Necula and Lee use Hoare logic to establish the memory safety of operating system kernel extensions [144] and native code ML library routines [143]. In the following, their work on kernel extension will be used for illustrating the idea of proof-carrying code.

Suppose a kernel maintains a table of descriptors. Each entry of the table is composed of two words: the first one is a tag specifying if the second data word is modifiable. The kernel allows users to install their application-specific access functions for the table entries. These access functions are passed pointers to a table entry when called. The functions must obey the following constraints:

[**C1**] The function must not access table entries other than the one passed as argument.

[**C2**] The tag word is read only.

[**C3**] The data word is writable if and only if the tag is non-zero.

[5] Every JVM call frame contains an operand stack for evaluating nested expressions.

[6] The contribution of Necula and Lee's work is twofold. One is demonstrating the feasibility of using Hoare logic as a practical verification formalism for mobile code security, and the other is the proposal of proof-carrying code as a verification protocol. The current discussion focuses on the first contribution. The second contribution will be discussed when the design space of verification protocols is explored in Section 2.4.2.

[**C4**] Privileged registers must not be modified (remember that the access functions are called by the kernel and thus are run in kernel mode).

The verification of the above policy is performed in the following steps:

1. The operational semantics of the underlying instruction set is formalized into proof rules.
2. The policy is encoded as precondition and postcondition.
3. The code for the access function is translated into an array of predicates describing the effect of executing the code.
4. The conjunction of the precondition, postcondition, and the predicates in the last step forms a *verification condition*.
5. The proof that the verification condition follows from the proof rules in step 1 is attached to the code unit.
6. Any party who would be interested in executing the code unit must check if the attached proof correctly establishes the verification condition.

Proving the memory protection policies described above turns out to be a very tractable task, especially for certain low level routines (e.g., network packet filters) in which no looping is involved. Though some of the above constraints can be readily enforced by type systems in an abstract intermediate language (e.g., Java bytecode), Hoare logic offers flexibility that allows one to deal directly with native code instead of using compiled high level language or interpreted bytecode, both introducing considerable run-time interference that cannot be tolerated in certain applications[7]. Moreover, the ability to deal with constraint [**C4**], in which the modifiability of one field is dependent on the value of another, demonstrates the expressiveness of Hoare logic. This benefit of expressiveness was demonstrated more fully in a later paper

[7]It is observed that PCC has significant performance advantages over its competitors like software-based fault isolation (Section 2.4.3).

[146], in which Hoare logic is used to enforce a liveness condition, resource usage limitation, and data abstraction in an information harvesting agent.

The down side of using a proof system as powerful as Hoare logic is its inherent undecidability. In Necula and Lee's experience, the added flexibility may result in difficulties when dealing with looping programs [144]. In those cases, one might have to generate a proof by hand. To limit this, they have experimented with automatically generating loop invariants by a certifying compiler [145]. In particular, starting with a small type-safe source language, one may build a compiler that generates machine code annotated with type specifications and loop invariants in the midst of aggressive compiler optimization (e.g., array-bound checking elimination). For simple memory- and type- safety properties, the above approach effectively automates the generation of loop invariants. Yet, for the general case, manual intervention cannot be avoided without the use of some domain-specific language [146, Section 9]. To avoid such manual intervention during run-time verification of mobile code, proofs may be generated ahead of time and transmitted with the mobile code. Section 2.4.2 describes this concept of proof-carrying code in more detail.

Typed Assembly Language While Java has to resort to an intermediate language in order to carry type information, and while Necula and Lee have to resort to a highly expressive logical proof to capture similar information for machine code, Morrisett *et al* [141, 140, 80] demonstrate that type checking can actually be performed on an assembly language. In particular, this is demonstrated by a typed assembly language (TAL) [141] which carries the type information of a rich, functional source language (a call-by-value variant of System F, the polymorphic λ-calculus augmented with products and recursion on terms). The significance of this work is threefold. Firstly, it demonstrates that type safety can be achieved without using an abstract intermediate language, which inevitably reduces run-time performance. Indeed, typed assembly code can be fully typechecked without reference to the original source program. Secondly, the typing construct imposes almost no restrictions on optimization. This makes the safety of the program independent of the compiler that generates the code. Thirdly, there is a type-preserving, effective procedure that can translate the source language to TAL. This contrasts with the incompleteness of verification under the more general approach of Necula and Lee [144].

In summary, one may see Java bytecode as a portable intermediate representation which allows type annotation to be attached for the sake of enforcing memory protection statically. Proof-carrying code, when applied solely to memory protection, captures typing information for a target language using a very expressive logic, thereby providing static typing without using an interpretive intermediate language. Finally, TAL demonstrates that static typing can actually be performed in a target language without the use of an overly expressive formalism.

Verification for Confidentiality

Program-analytic approaches to the enforcement of confidentiality have received a lot of attention, and are relatively well-understood. Building on Bell and La Padula's security model [20, 124], the work of Dorothy Denning [50, 52, 51] has laid the foundation for the study of secure information flow analysis. Developments have been constantly reported [171]. In particular, the work of Volpano *et al* [213, 209, 208, 186, 210, 187, 211, 212, 207, 184, 185] on using a type system to capture information flow has recently attracted considerable attention from the mobile code community. This section gives a brief survey of Denning's lattice model of information flow, and the basic idea of Volpano *et al*'s type system.

In the US military's *multilevel security* model, documents are *classified* into a finite set of sensitivity levels such as *unclassified*, *restricted*, *confidential*, *secret*, and *top secret*. Personnel are each granted a single *clearance level*, indicating a level of trust. Personnel with clearance *secret* may access all documents except those classified as *top secret*. The Bell-LaPadula model and its later generalization by Denning is a formalization of the above concepts. A security system is composed of a set S of subjects and a disjoint set O of objects. Each subject $s \in S$ is associated with a fixed security class $C(s)$, denoting its clearance. Likewise, each object $o \in O$ is associated with a fixed security class $C(o)$, denoting its classification level. The security classes are *partially ordered* by a relation $\leq$, and $\leq$ forms a *lattice*[8]. To prevent subjects with low clearance from accessing sensitive data, the following property is needed:

> **Simple Security Property** (No read up). A subject s may have *read* access to an object o only if $C(o) \leq C(s)$.

To prevent subjects with high clearance from releasing sensitive data to low-clearance subjects, the following property is needed:

> ***-Property** (No write down). A subject s who has read access to an object o may have write access to an object p only if $C(o) \leq C(p)$.

In summary, a subject may only read objects with classification level no higher than his clearance, but may only write to objects with classification level no lower than his clearance. Information must only flow unidirectionally from low classification sources to high classification destinations.

In the context of mobile programs, one may want to control the flow of information among variables and routines. For example, one may want to protect the content of a private variable from leaking into a globally accessible data area, or from flowing into a routine that writes to a socket. To illustrate this, let us define a very simple procedural language:

$$
\begin{array}{rrl}
\textit{(expressions)} & e ::= & x \mid n \mid e + e' \mid e = e' \mid e \ < \ e' \\
\textit{(commands)} & c ::= & e \ := \ e' \mid c; c' \mid \textbf{if } e \textbf{ then } c \textbf{ else } c' \mid \\
& & \textbf{while } e \textbf{ do } c \mid \textbf{letvar } x := e \textbf{ in } c
\end{array}
$$

Suppose we have two variables H and L, so that H contains some classified data and L is globally accessible. Information about the value stored in H can flow into L in at least two ways. Firstly, information flow may be *explicit*:

[8] A partial ordering is a binary relation which is transitive and antisymmetric. A lattice is a partial ordering in which every pair of elements possesses a least upper bound and a greatest lower bound. Consult Davey and Priestley [47] for more information.

$X := H + 1;$
$L := X - 1;$

Secondly, information flow from H to L may also be *implicit*. Conditional statements may convert information into control flow.

if $H > 0$ **then**	$L := 0;$
$\quad L := 1;$	**while** $H > 0$ **do**
else	$\quad H := H - 1;$
$\quad L := 0;$	$\quad L := L + 1;$

Information flow analysis attempts to statically uncover these kinds of information leakage. As noted by Podgurski and Clarke [155], information flow analysis is a kind of dependence analysis [66]. In the following, a type system in the style of [213] is presented.

To deal with explicit information flow, each expression is associated with a secure flow type, which represents the classification level of the data item. The lattice structure of the classification levels induces a natural subtyping relationship among the secure flow types: if type τ represents a classification level at least as high as that of type τ' then $\tau \geq \tau'$. An expression involving operands with distinct security types receives the least upper bound of the operands' types as its type. For example, if e and e' have security types τ and τ' respectively, and $\tau \leq \tau'$, then $e + e'$ can be assigned security type τ'. Each variable also has a type τ *var*, indicating that it holds contents with type no higher than τ. Explicit leaking is then prevented by requiring that assignment of the form $X := a$ is well-typed only if X has type τ *var* and a has type no higher than τ. To formally express this, we allow expression type τ to be coerced to any type τ' if $\tau \leq \tau'$, and then require that $X := a$ is well-typed if and only if X has type τ *var* and a has type τ. With this arrangement, the above example code that explicitly leaks information will not be well-typed.

To handle implicit information flow, every command is associated with a type τ *com*. Intuitively, a command has type τ *com* if every variable that is being assigned in the command has type τ' *var* where $\tau \leq \tau'$. That is, τ is a lower bound for the security levels of the variables being assigned in the command. The idea is that if a conditional or iterative construct involves a condition expression of type τ then commands in the body should not assign to variables with security levels lower than τ. To make this work, we need two more subtyping rules. For the variables, τ *var* $\leq$ τ' *var* if and only if $\tau \leq \tau'$. For the commands, the opposite must hold: τ *com* $\leq$ τ' *com* if and only if $\tau' \leq \tau$. Again, expressions can be freely coerced to their supertypes. The following type rules are then defined:

- An assignment statement of the form $X := a$ has type τ *com* if X has type τ *var* and a has type τ.
- A sequential composition of the form $c; c'$ has type τ *com* if both c and c' have type τ *com*.
- A conditional statement of the form **if** e **then** c **else** c' has type τ *com* if e has type τ while c and c' have type τ *com*.
- An iterative statement of the form **while** e **do** c has type τ *com* if e has type τ while c has type τ *com*.

Volpano *et al* prove that a type system such as the above satisfies variants of the simple security property and the *-property [213]. They also extend their type system to account for covert flow [208], procedures [209], and concurrency [186]. A *covert channel* is a mechanism that is not intended for communication but, nevertheless, may leak information. A good example is that a thread T may be constructed to conditionally enter an infinite loop depending on the value of a classified variable. Now another spying thread S may then time the execution of thread T, thereby obtaining information about the classified variable without exchanging data with T. To deal with procedures, they use a constrained type to capture the condition on which the procedure may be executed securely. Instead of a fixed type, a principal type containing subtype inequalities is computed. They have also studied the impact of various scheduling assumptions on the soundness of secure flow types in the presence

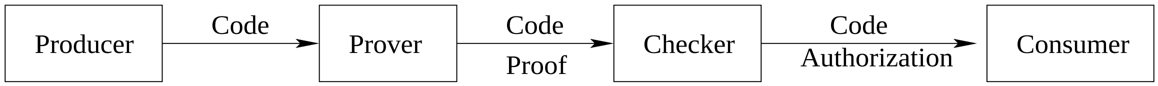


Figure 2.3: Participants of a Verification Protocol

of concurrency. It is shown that the original type system in [186] has to be properly restricted in order for soundness to be preserved (e.g., by restricting the kind of condition that can appear in conditional or iterative statements).

Information flow analysis is a very difficult problem. In a simple sequential language in which there is no malicious third-party observer, information flow analysis as described in this section is fully adequate. However, in a context typical to mobile programs, neither the sequential assumption nor the closed-world assumption hold. In order to deal with the complications of covert channels and malicious observers, analyses such as the above quickly become too conservative to be useful. Recent developments in this area attempts to provide more accurate analyses. See [171] for an up-to-date survey.

Verification Protocol

The verification approach requires only that security properties of the mobile code units are checked prior to their execution. It does not specify how the various parties involved in the verification process are to be orchestrated. The design space within the verification approach is in fact much wider than we have seen earlier. The term *verification protocols* is used to refer to schemes that specify how various concerned parties cooperate to carry out a verification procedure. A typical verification system consists of four parties (see figure 2.3):

Producer: The party who generates the mobile code unit.

Prover: The party who takes a mobile code unit, and generates a proof that the code satisfies a predefined set of security properties.

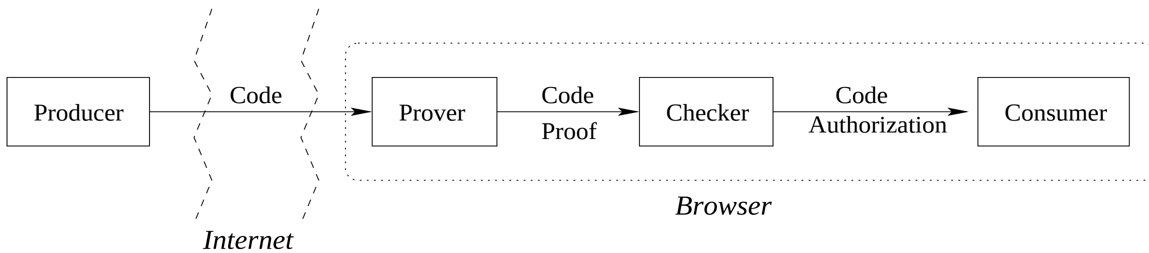


Figure 2.4: The Proof-on-Demand Protocol

Checker: The party who takes a mobile code unit and the proof generated by the prover, and translates it into a decision of whether execution/linking is authorized.

Consumer: The party who takes the mobile code unit and an authorization of execution, and executes the code.

A verification protocol specifies where the above participants are located, and how they orchestrate the verification process. In the following, three verification protocols will be surveyed: *proof-on-demand*, *proof-carrying code* and *proof delegation*.

Proof-on-Demand. The proof-on-demand protocol is exemplified in the Java mobile code system. Both the prover and the checker are on the host that execute the mobile code unit (Figure 2.4). Whenever an untrusted classfile is to be linked into the JVM, the bytecode verifier will be invoked on-the-fly to prove that the classfile is structurally intact and type-safe [129]. Since the JVM assumes both the prover and consumer role, there is no need to generate the proof explicitly, and thus proof checking reduces to the simple authorization of execution if verification succeeds.

Performing verification in the computing environment has the key advantage that it supports run-time code generation. Execution units may dynamically generate code that will be linked in on-the-fly. Having the verifier built inside the dynamic linker allows such dynamically generated code to be verified properly — a feature essential for implementing protection mechanisms based on dynamic code rewriting [59, 201, 202, 215, 220, 167, 168].

However, the proof-on-demand protocol has several drawbacks:

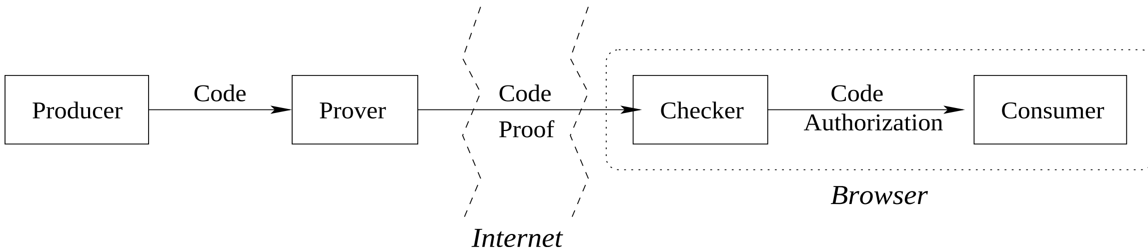


Figure 2.5: The Proof-Carrying Code Protocol

1. Compared to authentication-based approaches, proof-on-demand introduces extra link-time overhead to the computing environment. For example, in the case of Java, verification could involve complicated dataflow analysis, which introduces observable slow down to the dynamic linking process[9].

2. There is a close coupling between the verifier and the computing environment. Such close coupling introduces two software engineering problems:

   (a) When a bug is found in the verifier, as frequently happened in various industrial strength Java-enabled web browsers[10], one has to replace the entire computing environment (e.g., browser). This is usually difficult to guarantee because there are in an order of 10^7 browser users out there, and it is difficult to guarantee that they are all updated.

   (b) If the architecture of the computing environment is not well articulated, there will be many dependencies among the loader, linker, and verifier. Consequently, building a correct verifier is made more difficult.

Proof-Carrying Code. In the proof-carrying code (PCC) protocol [144, 143], the verifier is located at the producer side of the internet (Figure 2.5). Verification proceeds in the following steps:

1. The code consumer publicizes a safety policy in the form of some static program properties. The properties should be specified in a way so that a proof that the

[9]McGraw and Felten [135, p 110] observe that the slow down could be as unbearable as a denial-of-service attack.

[10]Consult [49, 135, 107, 108, 157] for a shamefully endless list of bugs found in various web browsers.

properties are present in a program can be formally expressed and mechanically verified.

2. If a code producer wants to ship a mobile code unit to the consumer, it must first acquire the safety policy.

3. On behalf of the producer, a prover then produces, manually or mechanically, a proof that the code unit to be shipped satisfies the safety policy mandated by the code consumer.

4. The proof is then attached to the code unit when it is shipped.

5. When the code unit and the attached proof arrive at the code consumer side, a mechanical proof checker will determine if the proof establishes the safety of the code unit. If so, it authorizes execution.

6. The consumer executes the code.

The idea of proof-carrying code is based on the intuition that proof-checking is more tractable than proof-generation. Firstly, since the safety of the mobile code unit is proven on the producer side instead of being repeated every time the code is loaded into the computing environment (as in the case of Java), the protocol effectively reduces link-time interference. Secondly, transferring the "burden of proof" to the code producer allows one to consider more expressive safety policy. The consumer does not have to worry that verifying such policy will introduce unbearable performance overhead to linking, for only proof checking is involved. With this setup, even manual proof generation is affordable assuming standardized policies for consumers. The resources the producer put into generating a safety proof for the code will be amortized over multiple uses of the code.

A potential problem of proof-carrying code has to be mentioned. In the worst case, the size of the proof may be exponential in the size of the program [144, 146]. This creates a problem for the transportation and checking of the generated proof. However, Necula and Lee observe that such "proof bloat" is a rare case.

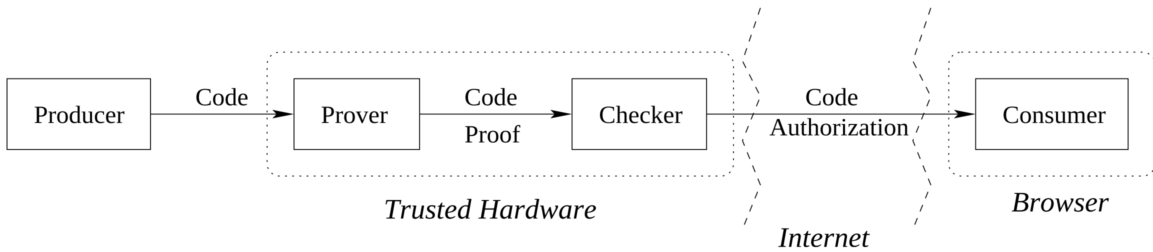


Figure 2.6: The Proof Delegation Protocol

Recent development of the idea of PCC includes lightweight bytecode verification [165, 119], typed assembly language [141], and certifying compiler for Java [44]. The common theme is to avoid the full expressiveness of proof-carrying code, while concentrating on a specific program analysis problem (e.g., type checking), in which small certificates can be generated mechanically by a certifying compiler.

Proof Delegation. Proof delegation[11] [54, 55] is motivated by a number of security and software engineering concerns, including the following:

1. **Performance**: Verification, be it proof generation or proof checking, introduces performance degradation. Can *either* or even *both* be removed from the consumer's burden?

2. **Disclosure**: Most of the verification protocols require the use of a high level (intermediate) language (proof-on-demand) or the attachment of program annotations (e.g., proof-carrying code). These representations may disclose information that the producer might consider proprietary.

3. **Configuration Management**: In all verification protocols, a verifier of some sort is involved. The implementation of such program verifier is usually complicated, and bugs are found in commercial implementations (footnote 10). When

[11] The term "proof delegation" was not coined in the original papers. The term is coined here simply because it reflects accurately where the approach lies within the wide spectrum of verification protocols.

verifier bugs are found, how easy is it to distribute the upgrades? Alternatively, how can customers be protected from using an "outdated" computing environment with known security vulnerabilities?

The proof delegation protocol consists of three components (figure 2.6):

1. **Trusted Hardware**: Beside publicizing the theorem to be proven, the consumer also distribute to the producer a trusted proof generator and a proof checker. The target theorem, the proof generator, and the proof checker are all installed into some trusted coprocessor, and the entire package is handed to the producer. Trusted coprocessors are usually realized in the form of hardware extensions to consumer equipment, for example, as PCI cards or PCMCIA cards. They contain a secret key, which is physically protected from tampering[12]. When a code producer wishes to distribute a mobile program, he submits the code to the trusted coprocessor. The embedded verification software in the trusted hardware checks if the code is safe, and then attaches a digital signature to the code using the secret key in the trusted coprocessor. The producer can then distribute the properly signed code.

2. **Authentication-enabled Computing Environment**: A computing environment that has authentication capability will then be able to check if a mobile code unit has been endorsed by a trusted coprocessor. If so, no link-time check will be necessary.

3. **Key Management Infrastructure**: When faults are found in an implementation of the verification algorithms, the key associated with the implementation can be revoked automatically by well-known key revocation techniques. From then on, mobile code units signed by the expired key (corresponding to the faulty implementation) will have to be re-certified. Browser owners have no need to upgrade their browsers.

[12]Attempting to physically tamper with the hardware and the firmware will trigger protection mechanisms that erase the secret key on board.

In effect, the verification effort is delegated to a trusted program analyzer. Firstly, on the customer side, proof generation and proof checking is replaced by efficient signature checking, thereby improving *performance*. Secondly, no program analytic annotation is needed for the authorization of execution, thereby avoiding *disclosure* of intellectual property. Thirdly, when a verifier implementation is found to be faulty and its corresponding key revoked, software upgrades are confined to the code production side, leaving code consumers (e.g., browser users) unaffected. This is much more manageable, for the number of producers is significantly smaller than the number of consumers. Moreover, mobile program developers have an obvious motivation for actively updating their certifying hardware. This arrangement addresses the *configuration management* need of keeping verification technology up-to-date.

The goal of trusted hardware is to establish a physically secure binding between a signature and a well-defined, program analytic semantics (Section 2.4.1). Such a binding makes the distribution of trusted program certification software possible. Without such binding, a malicious programmer may tamper with the certification software, generating signed mobile code units that are in fact faulty. Trusted hardware is thus designed to ensure that this tampering does not occur.

2.4.3 Transformation

A mobile code representation that is good for transportation (e.g., platform independent, compact for transport efficiency) may not be tailored for execution. In many mobile code systems, code units are transported in the form of virtual machine bytecode, and then, upon arrival to a host, the bytecode is transformed into native code for efficient execution. Such just-in-time (JIT) compilation [15] is now a core feature of mobile code systems like Java [130] and Omniware [133]. Link-time code generation also adds portability to mobile code systems [77]. Yet, dynamic code generation can also be seen as a protection mechanism. Mobile code units are expressed in a high level representation (e.g., a type-safe intermediate language as in Java) in which unsafe behavior cannot be expressed. Upon arrival at the host, the code units are translated into a format which is directly executable on the host machine. Since code generation is performed by a trusted compiler on the host, and since unsafe behavior cannot be expressed in the source code, the generated code can be considered safe.

In a similar vein, transformation can be used to tailor untrusted code into a more secure form. In contrast to dynamic code generation, unsafe behavior *can* be expressed in the migrated code. Upon arrival at the host, the code unit is statically analyzed, and extra protection code is injected at program points where security cannot be guaranteed.

Transformation for Memory Protection

As early as the 1970's, code rewriting has been applied to memory protection within a single address space [53]. Recently, the Omniware mobile code system [133] uses transformation to implement memory protection for untrusted mobile code units. Omniware mobile code units are transported as bytecode for the Omniware Virtual Machine (OmniVM) [4]. OmniVM is designed to resemble a RISC architecture, thus allowing efficient performance, simple implementation, and retargetability. OmniVM divides its address space into segments. In order to ensure that execution units access only those segments for which they have authorization, Software-based Fault Isolation (SFI) is used [214]. The basic idea of SFI is to rewrite untrusted mobile code units into versions containing no access to unauthorized segments. Each memory address is divided into two parts, namely, a segment identifier and an offset within the segment. Two possible rewriting rules can be formulated:

1. **Segment Matching**: For every memory reference, guard code is inserted before the instruction that initiates the reference. The inserted code dynamically checks that the referenced segment matches the current segment. A memory fault is raised if the check fails.

2. **Sandboxing**: For every memory reference, the segment identifier of the target address is dynamically overwritten by the identifier of the current segment.

The systematic application of either rule to every memory reference in a program guarantees that no interference occurs between disjoint segments.

Experience indicates that observable run-time overhead is caused by this approach because additional code is introduced by the transformation. Despite this overhead, native code that is instrumented this way can run at speeds comparable to the original code [214], although not as efficiently as a proof-carrying code version [144, 146].

In extensible operating systems VINO [178] and Exokernel [58, 110], users are allowed to dynamically download untrusted extension code into the kernel address space to modify the behavior of the operating systems. To avoid corruption of the kernel address space, untrusted extension code units are subject to SFI transformation before downloading.

2.4.4 Arbitration

Another way to protect a host is to protect it completely from "direct" contact with untrusted execution units. Whenever an untrusted execution unit requests the execution of an operation, a trusted party, *the arbitrator*, is called in to carry out the operation for the execution unit. By restricting the kind of operations visible to the execution unit, and by examining the client's run-time state, the arbitrator can carefully block out unsafe operations. The cost of such flexibility is usually a considerable run-time overhead.

Arbitration can be used to enforce both memory protection and access control. An *interpreter* is often used to enforce memory protection. *Interposition* is frequently used for enforcing access control. Each of them will be examined in turn.

Memory Protection by Interpreter

Using an interpreter has been a very popular way of implementing safe and portable computation. Mobile code languages like Java [89], Safe Tcl [153], Scheme48 [162], and Telescript [221], JavaScript [147], all involve the interpretation of some source or intermediate languages. The interpreter approach can achieve memory protection in two ways:

1. **Restricting expressiveness:** A safe intermediate representation can be defined for mobile code units. Due to language restrictions, certain unsafe operations cannot be expressed, while others can be statically checked for. Take the JVM bytecode representation [130] as example. Privileged native instructions cannot be expressed; there is no pointer arithmetic; the language is strictly typed; interactions with host resources are performed through a public application programming interface (API). As a result, memory interference can be avoided.

2. **Dynamic checking:** The execution unit interacts with the host CPU only through the arbitration of the interpreter. Consequently, the interpreter can screen out all potentially dangerous moves by run-time checking. For example, the JVM checks against null pointer dereferencing, out-of-bound array access, and illegal type-cast [130].

Access Control by Interposition

Interposition is the insertion of trusted arbitration code, usually in the form of a *reference monitor* [170] (Figure 2.7), between a protected service and the entry point of the service. In a traditional operating system setting, processes usually access system resources via a non-bypassable system call interface. Any attempt to access the protected resources are therefore subject to the monitoring of the trusted arbitration code before reaching the target service. Access control policies can be programmed into the arbitration code, which can flexibly screen out inappropriate access to the service. This section surveys several implementations of interposition in mobile code systems: *application wrappers*, *reference monitors*, *reference monitor inlining*, and *name resolution control*.

Application Wrappers. Application wrappers [83, 79, 78, 67] are software containers that control the interactions between untrusted programs and their execution environments. The goal is to retrofit arbitration code into a legacy software system in a non-intrusive manner.

Janus [83] is an application wrapper especially designed for protecting a host against insecure mobile code computing environments. The intuition behind the design of Janus is that an untrusted process cannot do much harm if its access to the underlying operating system is appropriately restricted. Using the process tracing facilities and the `/proc` virtual file system [62] in Solaris, Janus creates a user-level sandbox that monitors all system calls made by an untrusted process. Since legacy computing environments which have unreliable protection mechanisms (e.g., an old version of `ghostview` [32], `sh`, or a buggy, Java-enabled web browser) can be executed inside the Janus sandbox, the Janus monitor can effectively block out unsafe system access initiated by the execution units running inside the legacy computing environment. Users may even supply their own policy module to specify which system calls to allow, which one to deny, and for which a function must be called to determine what to do.

Janus represents a very practical solution to a very practical problem. It does not require modification to the kernel and the computing environment, and it effectively protects the host from any unreliable computing environment. However, even disregarding its platform-dependent nature, Janus does not satisfactorily address the layered protection problem. Firstly, Janus does not allow the computing environment to define a different protection domain for each execution unit. Secondly, the kind of security policy it may express is limited due to its ignorance of the semantics of the computing environment. For instance, when a JVM is running inside a Janus sandbox, the policy modules of Janus have no way of figuring out the internal state of the JVM, and must make their access control decision independent of the JVM state. In general, layered protection is adequately addressed only when interposing is a built-in feature of the computing environment instead of being a retrofitted patch of the operating system.

Reference Monitors. The Java incarnation of a reference monitor is composed of two mechanisms — the *security manager* and *stack inspection*. All access to operating system services are isolated in the standard Java API. Whenever a service routine is invoked, the API transfers control to a corresponding monitor method of the global

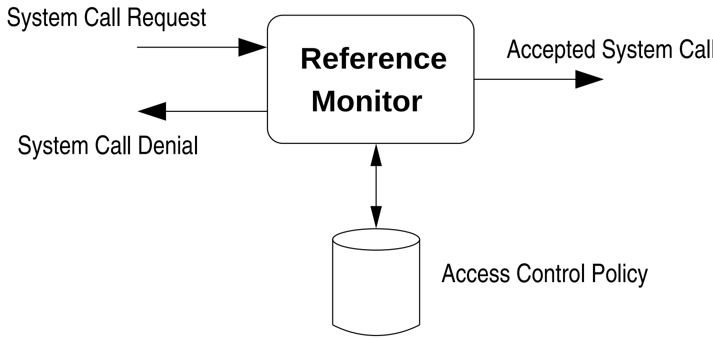


Figure 2.7: Reference Monitor

security manager object. The monitor method will inspect the Java run-time stack to determine if the call is safe. If the monitor method disallows the access, an exception will be thrown. Otherwise, control is returned to the service routine, carrying out the original request. The security authority may override these monitor methods of the security manager class in order to customize the security policy of the JVM.

The Java security model allows one to define intricate security policies. Stack inspection allows the security manager to decide with fine control what access will be granted. The drawbacks of this approach are as follows. Firstly, the security manager needs to implement complex stack inspection logic to differentiate between accesses initiated by different execution units. From a software engineering point of view, the construction and maintenance of this logic is both difficult and prone to error. Secondly, a procedural definition of security policy is hard to understand. A popular solution is to implement traditional access control lists in the arbitration code (as in Java [84, 85, 86, 87, 88] and Agent Tcl [90]). Subsequently, Netscape has attempted to extend the Java stack inspection mechanism by providing stack annotation which simplifies the logic for access right checking [216]. This extended version of stack inspection is later proven by Wallach, Appel and Felten [217, 215] to be equivalent to formal deduction in ABLP logic [1].

Reference Monitor Inlining. Code rewriting (Section 2.4.3) can be applied at load time to introduce monitoring code into an untrusted program. Here, the arbitration code does not reside at the entry points of privileged services, but instead is injected into the program itself to detect and avoid misuse of privileged services. Specifically, Java stack inspection has been implemented using this strategy [202, 215]. SFI has also been applied to enforce security policies expressed as security automata [201]. There have been a number of other efforts in applying load time code rewriting to enforce high level access control policies [59, 220, 167, 168].

Name Resolution Control In name resolution control, arbitration occurs at the time of dynamic linking. The relative simplicity of name resolution offers the potential of centralizing all security logic into a single mechanism.

Safe-Tcl [153] is a security-aware extension of the popular Tcl scripting language [152]. Protection is achieved by three mechanisms — *safe interpreters*, *aliases*, and *hidden commands*. Tcl is a command-based language, similar to other shell scripting languages. Access to operating system facilities are provided through a set of commands. Safe-Tcl defines a *padded cell security model*, in which every execution unit is executed in its own interpreter. All system services are available in a trusted, master interpreter. When an untrusted script is executed, it is sandboxed in a separate, untrusted, safe interpreter. A safe interpreter acts like a separate name space. Privileged commands can be *hidden* in the safe interpreter, thus blocking untrusted script from unauthorized access to system resources. Also, to obtain finer-grained control, a command may be *aliased*. Specifically, the name of a privileged command in the safe interpreter may be "overshadowed" by a trusted arbitration routine in the master interpreter. The arbitration routine decides at run-time if the access is granted. If the access is permitted, it delegates the original call to the overshadowed command in the master interpreter.

The padded cell model represents a form of interposition called *name resolution control*, in which the name resolution mechanism (e.g., dynamic linking) is exploited to provide selective exposure of privileged services. In essence, name resolution control is composed of two component mechanisms. Firstly, granting of capabilities is realized by *name visibility control*. The notion of a safe interpreter, which is essentially a namespace, coincides with that of a protection domain. A privileged service can be accessed only if it can be *named* in the safe interpreter. The assignment of each script to a separate name space, plus the possibility of name hiding, allows one to easily tailor a different access policy for each script. Secondly, *message interception* selectively binds names of privileged services to wrapper code that protects the entry points of those services. Here, accessibility is not controlled by visibility, but instead by dynamic checking of the possession of rights.

Scheme48 [162] is another early mobile code system that uses name resolution control as its primary protection mechanism. In Scheme, a procedure is a function closure, containing a lambda expression and a binding environment. When a procedure is applied, the only objects that are visible inside the lambda expression are the actual arguments and the values of the names in the lexical environment. Scheme48 allows programs to construct arbitrary binding environments, and then execute untrusted code inside these carefully-crafted environment. During the course of constructing such environments, privileged procedure names can be made invisible or be redefined to refer to arbitration routines.

Wallach *et al* [216] describe a way to implement name resolution control in the context of Java. In Java, a name space coincides with a classloader [127]. A class name in one classloader represents a different class than another class with the same name in a different classloader. The classloader was originally conceived for name space partitioning so that there will be no name conflict among separate execution units. Taking advantage of this design, one may create a subclass of the standard classloader class, in which all requests for name resolution are monitored. As a result, if a privileged name is to be hidden, the classloader can throw an exception when the name is resolved. Aliasing can be simulated by resolving the names of privileged classes to arbitration classes.

The extensible operating system SPIN [22, 181] also models protection domains by name spaces. All extension code in SPIN is written in the type-safe language Modula 3. Capabilities are directly modeled as pointers. Therefore, if a name is well-typed in a code unit, then the resource or service it refers to will be accessible. Typing thus provides a means of expressing *conditional visibility* of a symbol. Fine-grained protection is achieved by allowing users to manipulate name spaces. Name spaces can be created dynamically, and code units are executed within the confine of that name space, thus restricting its capabilities. An interesting feature is that name spaces can be extended by the `Combine` operation, which creates a union of two name spaces. This compares with the more flexible name space extension primitive `import` mentioned in Section 2.2.3. In general, a system that uses name resolution control for protection needs ways to construct and extend name spaces.

One potential problem with modeling protection domains using name spaces is that there is no way of revoking capability. The J-Kernel [94], is a Java security kernel that provides a capability revocation mechanism within a name-space-as-protection-domain framework.

Interposing Mechanisms. In the reference monitor inlining and name resolution control approaches, arbitration code is interposed by dynamic code rewriting and dynamic linking respectively. Other advanced programming language constructs having the potential of being adopted as interposing mechanisms include, for example, behavioral reflection [219], metaobject protocols [118], and aspect-oriented programming [116, 175]. Protection mechanisms based on behavioral reflection have begun to appear [220]. Although most of these mechanisms ultimately rely on load-time code rewriting [175, 220], they represent high-level language abstractions that are usually easier to work with than rewriting.

Interposing works by monitoring the execution of an untrusted execution unit. It is theoretically interesting to find out how much can be achieved by such a mechanism. Schneider [176] proposed a characterization of security policies enforceable by execution monitoring (EM). Specifically, an EM-enforceable policy prescribes access event sequences recognized by a Büchi automaton [9]. It is observed that Büchi-like security automata can only enforce safety properties, but not liveness properties. Viswanathan [205] points out that any reasonable characterization of execution monitoring must involve a computability constraint. Subsequently, Bauer, Ligatti and Walker [19, 128] proposed a characterization of increasingly general classes of security policies enforceable by insertion, suppression and editing automata, while Hamlen, Morrisett and Schneider [91] offers a characterization of security policies enforceable by code rewriting. These policy classes are provably more expressive than EM-enforceable policies. An open question raised by Bauer, Liatti and Walker is whether one can further classify the space of security policies by *constraining* the capabilities of the execution monitor. The question is partially addressed by Fong [69], who proposed a fine-grained characterization of subclasses of EM-enforceable security policies using an information-based approach. The characterization yields policy classes that contain naturally occurring security policies.

Chapter 3

The Proof Linking Architecture

This chapter presents a brief overview of Proof Linking as a language-independent verification architecture for mobile code systems. A set of formal correctness conditions are formulated for evaluating language-specific instantiations of the architecture. The possibility of applying Proof Linking to address the issues of stand-alone verification modules, distributed verification, and augmented type systems is also discussed.

3.1 Architectural Overview

A mobile program is assumed to be composed of one or more *code units* (modules, classes and so on), each of which may contain externally visible *members* (functions, methods, variables, and so on). Code units and their members are identified by symbolic names. A code unit and its members may contain symbolic references to other code units and their members. When a program is executed, its code units are loaded, verified, and the symbolic references are incrementally replaced by actual machine pointers.

A modular architecture for dynamic linking is postulated here. It is assumed that link-time activities like loading a code unit, verifying a code unit, and resolving a symbolic reference are all atomic *linking primitives*, in the sense that, although concurrent execution is allowed, no linking primitive attempts to invoke any other

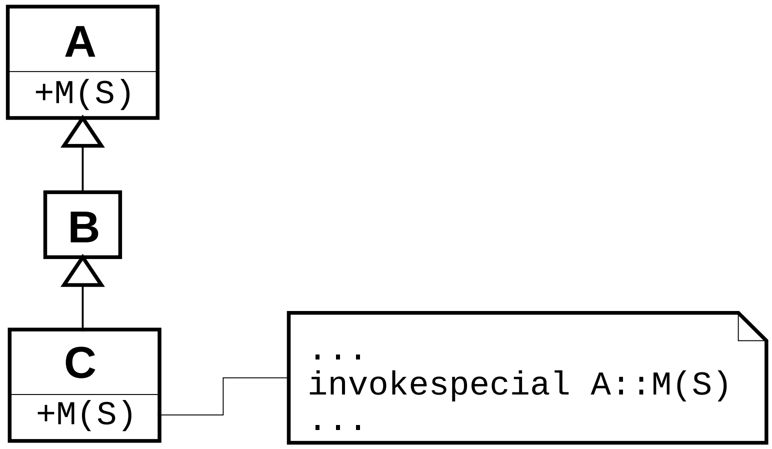


Figure 3.1: Running Example

during its processing, nor will one recursively invoke itself. This setup poses the following challenge:

> *Verification requires knowledge of other code units which might not have been loaded yet. How can the loading of these code units be avoided without affecting the integrity of the verification process?*

For illustrative purpose, consider the example as depicted in Figure 3.1. Suppose the Java class A defines a method $M(S)$. Suppose further that A has a direct subclass B, which in turn has a direct subclass C. Assume that class C overrides the method $M(S)$. Say the body of method $C{::}M(S)$ contains an *invokespecial* bytecode instruction that delegates the call to method $A{::}M(S)$. When method $C{::}M(S)$ is verified, the bytecode verifier has to check if class C is a subclass of class A in order to type check the *invokespecial* instruction. This fact cannot be confirmed by examining solely the body of $C{::}M(S)$, but instead class A and other superclasses of class C must be examined. This example illustrates a challenge that any modular verification architecture must address. In the Proof Linking architecture, this challenge is addressed by the decomposition of verification into two subtasks: *modular verification* and *proof linking*[1].

[1] Capitalization is used to differentiate between "*Proof Linking*" as a verification architecture and "*proof linking*" as a verification subtask

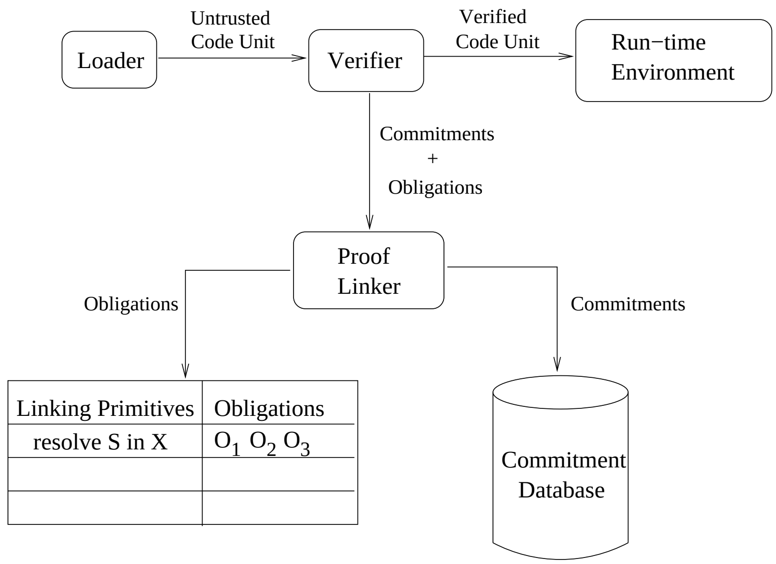


Figure 3.2: Modular Verification

3.1.1 Modular Verification

Verification is modularized by the formulation of *verification interfaces*. Figure 3.2 depicts the setup for modular verification in a prototypical mobile code system. Untrusted code units are subjected to static verification after loading. The verifier might need the knowledge of another code unit in order to decide if the current code unit should be endorsed. Instead of validating such external dependencies by recursively loading and/or verifying other code units, the modular verifier computes for the current code unit a conservative *safety precondition* summarizing the external dependencies that will guarantee safety of this code unit. The safety precondition is represented as a conjunctive set of database queries. In the running example (Figure 3.1), during the verification of classfile C, a modular Java bytecode verifier will generate the query `subclass(`*'C'*`,` *'A'*`)` as the *invokespecial* instruction is scanned. The verifier may end up generating many such queries. The conjunctive set of all queries formulated by a verification session becomes the safety precondition for endorsing the code unit in question.

The modular verifier also *schedules* each of the queries for evaluation. Each query describes a safety precondition of a certain linking primitive. For example, the query above, `subclass(`*'C'*`,` *'A'*`)`, is associated with the linking primitive "**resolve** A::$M(S)$ **in** C". In essence, this schedules the subclassing check for evaluation immediately prior to the resolution of the method symbol A::$M(S)$ in class C. Such a query is said to be the *proof obligation* for the associated linking primitive, representing a condition that must be met if the run-time system is to safely execute the corresponding linking primitive. A proof obligation is also said to be *attached* to its associated linking primitive. Proof obligations generated by the modular verifier are collected by the *proof linker*, which records in a global *obligation table* the mapping from linking primitives to their attached obligations.

In order for the run-time system to discharge proof obligations, the verifier also computes, for each code unit, a set of clauses called *commitments*. The commitments are ground facts that describe the interface properties of the code unit. For example, during the verification of the Java classfile C, the fact `extends(`*'C'*`,` *'B'*`)` is generated as one of the commitments. The generated commitments are collected by the proof linker, and subsequently asserted into a global *commitment database*. As we shall see below, the commitment database provides the set of facts against which proof obligations are evaluated.

3.1.2 Proof Linking

The process by which the run-time system cross-validates the results of verifying different code units is called *proof linking*. Figure 3.3 depicts the setup for proof linking. When the run-time system needs to execute a linking primitive, it sends the request to the proof linker. The proof linker looks up the obligations that have been attached to the linking primitive in question, and then posts them to the commitment database as deductive queries. If the queries are satisfied, the request is granted. Otherwise, a linking exception is raised to signal failure to proof link.

To make proof linking more expressive, arbitrary logic programs can be provided as an *initial theory* in the commitment database. For example, recursive definitions of the following program can be present in the commitment database to define subclassing as the transitive closure of the `extend/2` relation:

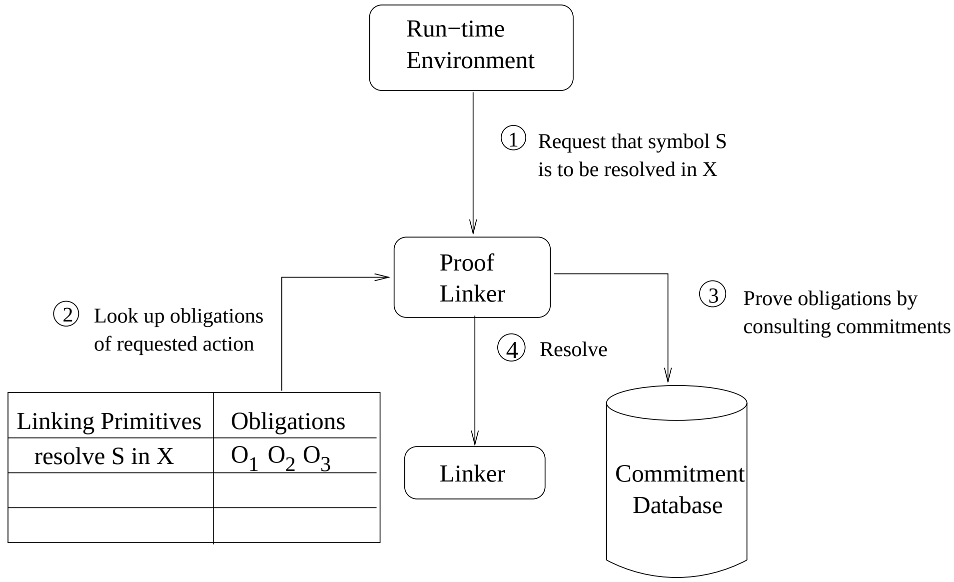


Figure 3.3: Proof Linking

subclass(X,X).

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

After the linking primitives "**verify** B" and "**verify** C" are processed, the commitment database may contain the following commitments:

extends(*'C'*, *'B'*).

extends(*'B'*, *'A'*).

When the linking primitive "**resolve** A::$M(S)$ **in** C" is to be executed, the JVM will look up its attached obligations, among which the query subclass(C,A) will be found. The proof linker then attempts to satisfy this query by consulting the facts and rules in the commitment database. The query succeeds and the linking primitive is executed (assuming that any other obligations are satisfied as well).

3.1.3 Remarks

Although a deductive database model has been used as a means of representing proof obligations and commitments, an actual system is not required to be implemented this way. As loading and linking in a mobile code system occur frequently, a declarative implementation would likely be unacceptably inefficient. Given queries and commitments of fixed signatures, and given a fixed initial theory, appropriate data structures and algorithms can be designed for the efficient assertion of commitments and discharging of obligations. For example, the commitments collectively defining the `extends` relationship can be represented in a space-efficient manner using an appropriate tree data structure, while the logic of the `subclass` relationship (i.e., transitive closure) may be implemented efficiently by a tree traversal algorithm. For more information about how proof linking can be implemented efficiently, consult Chapter 6.

There are however two reasons to model the proof linking process as a series of database updates and queries. Firstly, the database model provides an abstract framework to describe the general notion of incremental proof linking, without getting into the idiosyncrasies of individual mobile code systems. Secondly, and more importantly, it allows one to define a formal model of proof linking and its correctness conditions.

3.2 Correctness of Incremental Proof Linking

To assess the theoretical soundness and modeling adequacy of the incremental proof linking process in a complex dynamic linking environment such as the JVM, one needs a theoretical model upon which the semantics and correctness of the process can be articulated. Such a formal model of incremental proof linking is the topic of this section. Based on this model, the following three correctness conditions are formalized:

1. **Safety:** All obligations relevant to the safe execution of a linking primitive are generated and checked before that primitive is executed.

2. **Monotonicity:** Checked obligations may not be contradicted by subsequently asserted commitments.

3. **Completion:** All commitments that may be needed for satisfying an obligation are generated before the obligation is checked.

Note the parallel between the complete generation of obligations required by the Safety condition and the complete generation of commitments required by the Completion condition. There is also an interesting parallel between Monotonicity and Completion. The latter may be rephrased to state that once an obligation fails, no subsequently asserted commitment will enable it.

In summary, the Safety, Monotonicity and Completion conditions are intended to ensure that the checking of proof obligations and authorization of linking activities are deterministic processes even though the lazy, dynamic linking procedure is not. In essence, the correctness of proof linking is characterized by the correct scheduling of static verification steps over time.

The remainder of this section formalizes these notions as follows. Section 3.2.1 defines a formal model of the proof linking process. Section 3.2.2 presents a simple proof linking algorithm as the operational semantics of the proof linking model. Section 3.2.3 then goes on to formalize the Safety, Monotonicity and Completion conditions in terms of the terminology developed in the previous two subsections.

3.2.1 Building Blocks of Proof Linking

The proof linking process is modeled as a sequence of verification steps orchestrated by a proof linker. An abstract model of the proof linking process consists of four building blocks: (1) linking primitives, (2) an initial theory, (3) proof obligations and commitments, and (4) a linking strategy.

Linking Primitives

The proof linker could be seen as a reference monitor that mediates and protects linking activities that have security significance. The first building block of a proof linking model is obtained by the identification of linking primitives to be monitored by the proof linker. The precise set of linking primitives may vary depending on the kind of activities one would like to model, but in a prototypical mobile code system the following set would be defined for each code unit X:

load X**:** acquire code unit X.

verify X**:** verify code unit X.

resolve S **in** X**:** replace symbolic reference S in code unit X with an actual machine pointer.

use S **in** X**:** symbolic reference S in code unit X is used for the first time.

Associated with each linking primitive p are two *linking events*, namely, "**begin** p" and "**end** p", which respectively represent the initiation and termination of the primitive p. These events occur asynchronously as the run-time system executes various linking primitives. It is assumed that events are then queued up in some synchronized event queue, waiting to be examined by the proof linker. Intuitively, when the run-time system requests that a linking primitive p be authorized to execute, "**begin** p" will be generated. Similarly, the run-time system generates "**end** p" to inform the proof linker that p is properly terminated. The sequence of linking events that enters the event queue from the beginning of an execution session to some point of execution is said to be an *execution trace* of the run-time system in that period of time. An execution trace is *well-formed* if it is reasonably structured in the following sense. It is assumed that each linking primitive can only be executed at most once during the life time of the run-time environment. Consequently, each event may occur at most once in a well-formed execution trace. It is further assumed that event "**end** p" can occur in a well-formed execution trace only if there is a corresponding event "**begin** p" occurring strictly before it. Given a fixed set P of linking primitives, $traces(P)$ denotes the set of all possible well-formed execution traces made up of linking events corresponding to primitives from the set P.

Initial Theory

The second building block of a proof linking model is obtained by the specification of an *initial theory*. The initial theory represents the evaluation logic to be used for checking proof obligations. The initial theory can be an arbitrary logic program representing a decidable first-order theory. Decidability is required for proof linking to be properly defined. The precise formulation of the initial theory depends on the semantics of the verification task at hand. The Greek letter Γ is usually used for denoting an initial theory.

Proof Obligations and Commitments

The third building block of a proof linking model is obtained by the definition of proof obligations and commitments that could be generated by linking primitives. In general, every linking primitive may generate both proof obligations and commitments. Commitments are facts describing the information collected as a result of executing a linking primitive. Obligations are queries that are attached as safety preconditions to linking primitives, called *targets*. An obligation-target pair is called an *attachment*. Notice that the obligations can attach to any linking primitive. It is assumed that commitments and attachments generated by a linking primitive are available when the corresponding **end** event occurs.

It is postulated that, given a fixed set P of linking primitives, the functions $com(p)$ and $att(p)$ map a linking primitive p to the respective sets of commitments and attachments generated by p when the event "**end** p" occurs. Also, the notation $obl(q)$ is used for representing the set of obligations that are attached to a linking primitive q, that is, the set $\{\ o \mid \exists\, p \in P \,.\, \langle o, q\rangle \in att(p)\ \}$.

Linking Strategy

Given a set P of linking primitives, a linking strategy σ is a subset of $traces(P)$. Every implementation of a mobile code run-time environment defines a linking strategy. The strategy expresses the order in which linking events may be processed by the run-time system. An execution trace $\tau \in \mathit{traces}(P)$ is *σ-conforming* if $\tau \in \sigma$. To say that a run-time system implements a linking strategy σ is to say that the run-time system guarantees that all execution traces it generates are σ-conforming[2].

A linking strategy can be specified syntactically by formulating ordering constraints. Ordering constraints are notations specifying properties that must be met by all execution traces in the linking strategy being specified. For example, given linking primitives p and q, the notation $p < q$ represents the property that an execution trace τ satisfies one of the following conditions:

- "**begin** q" does not occur in τ;
- "**begin** q" occurs in τ after an occurrence of "**end** p".

Given the notation above, a linking strategy for a prototypical mobile code system can be specified as follows: given any code units X and Y, and a symbol S imported by code unit X from Y, the following holds:

1. **Natural Progression Property:**

$$\textbf{load } X < \textbf{verify } X < \textbf{resolve } S \textbf{ in } X$$

2. **Import-Checked Property:**

$$\textbf{verify } Y < \textbf{resolve } S \textbf{ in } X < \textbf{use } S \textbf{ in } X$$

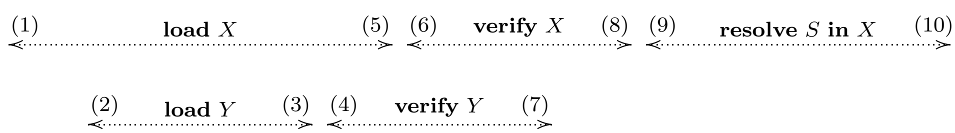
Assuming that code unit X imports symbol S from code unit Y, the following execution trace conforms to the strategy:

---

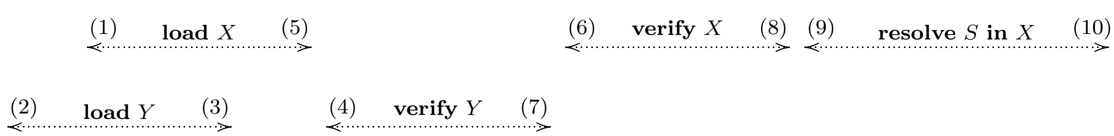
[2]Under this definition, a run-time system may implement a linking strategy by generating only a subset of conforming execution traces.

(1) **begin** "**load** X"
(2) **begin** "**load** Y"
(3) **end** "**load** Y"
(4) **begin** "**verify** Y"
(5) **end** "**load** X"
(6) **begin** "**verify** X"
(7) **end** "**verify** Y"
(8) **end** "**verify** X"
(9) **begin** "**resolve** S **in** X"
(10) **end** "**resolve** S **in** X"

The ordering of events corresponds to the following timeline:



Switching the relative ordering of events (1) and (2) results in a new execution trace that still conforms to the strategy. Further switches of (4) with (5) and (6) with (7) also maintain conformance and lead to an execution trace illustrated by the following timeline diagram:



Now, if the position of (8) with (9) are further switched, then the resulting execution trace would violate the **Natural Progression Property**. Similarly, moving event (7) after (9) would violate the **Import-Checked Property**.

In summary, a proof linking model is given by a 5-tuple $\langle P, \Gamma, com(\cdot), att(\cdot), \sigma\rangle$, where P is a set of linking primitives, Γ is an initial theory, $com(\cdot)$ and $att(\cdot)$ are commitment and attachment mappings, and σ is a linking strategy.

```
algorithm ProofLinker(P, Γ, com(·), att(·), σ):

DB ← Γ; Obligations[ ] ← ∅;
Ready ← ∅; Satisfied ← ∅; Failed ← ∅;
while (¬ run-time-env-terminated_σ()) do
  e ← get-next-event_σ();
  switch e of
  case "begin p":
    All-Obligations-Satisfied ← true;
    for all o ∈ Obligations[p] do
      if (DB ⊢ o) then
        Satisfied ← Satisfied ∪ { o };
      else
        Failed ← Failed ∪ { o };
        All-Obligations-Satisfied ← false;
      end if
    end for
    if (All-Obligations-Satisfied) then
      Ready ← Ready ∪ { p };
      authorize-execution(p);
    else
      deny-execution(p);
    endif
  case "end p":
    DB ← DB ∪ com(p);
    for all ⟨o, q⟩ ∈ att(p) do
      Obligations[q] ← Obligations[q] ∪ {o};
    end for
  end switch
end while
```

Figure 3.4: The Proof-Linker Model Algorithm

3.2.2 A Model Proof Linking Algorithm

The semantics of the proof linking process is defined operationally as an algorithm handling asynchronously generated linking events. Figure 3.4 presents a model proof-linking algorithm in which linking primitives are consumed from a global event queue. The proof linker maintains two global data structures, namely, a commitment database (`DB`) and an obligation table (`Obligations[]`). The commitment database is a decidable first-order theory containing both facts and rules. The obligation table maps each linking primitive to a set of database queries. Initially, the commitment database contains the initial theory (Γ), and the obligation table is empty (line 1). The proof linker consumes linking events in an order specified by the linking strategy (line 4). When the **begin** event of a linking primitive is removed from the event queue (line 6), its associated obligations are retrieved from the obligation table (line 8). The verification of these obligations is then attempted against the logic program in the commitment database (line 9). If all obligations are satisfied (line 16), then the linking primitive will be allowed to execute (line 18); otherwise, its execution will be denied (line 20), which means that the corresponding **end** event will not be returned when line 4 is executed in the future. Alternatively, when the **end** event of a primitive is removed from the event queue (line 22), the commitments and attachments for the primitive are collected. The commitments are added to the commitment database (line 23). The attachments are incorporated into the obligation table (lines 24–26). The proof linker repeats this process until the run-time environment terminates (line 3).

3.2.3 Formalization of Correctness Conditions

To formalize the correctness conditions of the proof linker, three auxiliary variables are introduced into the listing in Figure 3.4. "`Satisfied`" (lines 2 and 10) denotes the set of obligations that have already been checked at line 9, while "`Failed`" (line 2 and 12) collects obligations that have failed to check. "`Ready`" (lines 2 and 17) is the set of primitives that are ready for execution.

Fixing the initial theory Γ and the commitment and attachment mappings $com(\cdot)$ and $att(\cdot)$, the proof linker is said to be correct if the following conditions hold:

1. **Safety:** Before any primitive is executed, all obligations that may potentially be attached to it are generated and checked. Formally, the following invariant should hold at all times:

   $$\forall p \in \texttt{Ready} \ . \ \forall o \in \texttt{Obligations}[p] \ . \ o \in \texttt{Satisfied}$$

   To enforce this, one may require that, for any linking primitives p and q, $p < q$ if there is an obligation o such that $\langle o, q \rangle \in att(p)$.

2. **Monotonicity:** Obligations may not be contradicted by subsequently asserted commitments. The Monotonicity condition may be captured formally by asserting that the following invariant holds at all times:

   $$\forall o \in \texttt{Satisfied} \ . \ \texttt{DB} \vdash o$$

   In a deductive database model, Monotonicity results naturally from the application of Horn clause logic [131]. If the initial theory and generated commitments are required to be definite clauses (aka Horn clauses) and the obligations are constrained to be definite queries, then no contradiction will result from the assertion of commitments, thus ensuring that subsequent commitments do not contradict satisfied preconditions.

3. **Completion:** Conversely, obligation failure may not be subsequently contradicted by asserted commitments. A formal restatement is that the following invariant should hold at all times:

   $$\forall o \in \texttt{Failed} \ . \ \texttt{DB} \nvdash o$$

   This condition can be enforced as follows. A commitment c is said to *support* an obligation o if (1) there is a set $S \subseteq P$ of linking primitives so that

   $$\Gamma \cup \bigcup_{p \in S} com(p) \vdash o,$$

(2) $c \in com(p)$ for some $p \in S$, and (3) c is used as a premise in establishing the proof for o. For any linking primitives p and q, $p < q$ is required if there is a commitment $c \in com(p)$ and an obligation $o \in obl(q)$ such that c supports o. Thus, if an obligation o of primitive q is eventually provable, then generation of the commitments necessary for its proof must be complete when "**begin** q" is processed.

In general, the correctness of proof linking depends on (1) the linking strategy σ, (2) the initial theory Γ, and (3) the commitment and attachment mappings $com(\cdot)$ and $att(\cdot)$. In particular, the Safety and Completion conditions constrain the linking strategy to ensure that an obligation is checked neither too late nor too early, while the Monotonicity condition imposes syntactic constraints on the underlying logic used in expressing proof obligations and commitments.

Note that the correctness conditions do not impose a strict policy on the linking strategy. Either eager linking (linking every code unit at once) or lazy linking (linking a code unit only when its code is being executed)—or indeed any intermediate strategy—can be tailored to satisfy the correctness conditions. To maximize the opportunities for laziness, however, strategies with fewer ordering constraints are preferred so long as the correctness conditions hold.

These three correctness conditions will be used to judge if a language-specific instantiation of the Proof Linking architecture is sound.

3.3 Architectural Advantages

This section explores how the adoption of the Proof Linking architecture addresses the issues presented in Section 1.1, namely, that of stand-alone verification modules, distributed verification, and augmented type systems.

3.3.1 Stand-alone Verification Modules

It is generally desirable for the mobile code verification technology to evolve independently of the mobile code hosting technology. In the context of Java classfile verification, this would mean that the standard bytecode verifier is manufactured as a replaceable component that can be "plugged" into any virtual machine that supports pluggable verification services. It should be possible to validate the correctness of verification components independent of the rest of the mobile code hosting environment. Third party vendors can specialize in producing highly secure verification modules, while JVM vendors can concentrate their efforts on producing faster virtual machines. As a result, installation of a virtual machine of one brand does not preclude the adoption of a bytecode verifier of another brand. This software configuration model should yield higher quality and more secure mobile code hosting environments.

The Proof Linking architecture is a framework for identifying and reducing the coupling between a mobile code hosting environment and its verification component. Consequently, it may represent a good basis for the development of stand-alone verification modules in mobile code systems.

3.3.2 Distributed Verification

Conditional Certification

Modularization makes it feasible for mobile code verification to be performed remotely. The correctness conditions in Section 3.2.3 only require that the **verify** primitive correctly generates all commitments and obligations. It does not specify how such commitments and obligations are generated. Consequently, remote verification can proceed as follows:

Conditional certification. Untrusted code units are certified by a trusted verifier at a remote site. Instead of endorsing external dependencies that might have been invalidated at link time, the remote verifier captures the dependencies in proof obligations and commitments. Prior to shipping a code unit, the remote verifier attaches the corresponding proof obligations and commitments to each code unit. The annotated code unit is signed by the verifier using its private key, and is then distributed to consumers.

Proof linking. Upon acquiring the signed, annotated code unit, the run-time system performs a special **verify** primitive that (1) authenticates the signature, and then (2) processes the proof obligations and commitments as if they were generated locally. To the proof linker, this special **verify** primitive looks not different than a normal **verify** primitive, and will proof-link the remotely-verified classfile correctly.

This scheme nicely addresses the need for conditional certification. Even though the remote verifier is unable to validate the external dependencies of a code unit, it nevertheless can express them as proof obligations. The proof linking mechanism is invoked to discharge the proof obligations at run-time, when the necessary information has become available.

Protocol Interoperability

The Proof Linking architecture provides an infrastructure for the interoperability of the verification protocols such as proof-on-demand, proof-carrying code and proof delegation. Because linking primitives communicate solely by attaching proof obligations and asserting commitments, they are highly modular. An intelligent **verify** primitive may process certified code units to produce the appropriate proof obligations and commitments as if they are formulated locally. For code units that are not certified at all, the intelligent **verify** primitive may itself perform a complete verification to generate the necessary proof obligations and commitments. The proof linking mechanism checks and discharges obligations from each of theses sources in the same way, without any need to know the verification protocol used for a particular code unit. As a result, a mobile program could consist of a number of code units that are signed remotely, others that are proof carrying, and still others that are completely uncertified. As long as each verification protocol uses the same *verification interface* for proof obligations and commitments, interoperability is assured.

3.3.3 Augmented Type Systems

If an implementation of the Proof Linking architecture supports an open mechanism of attaching obligations to linking primitives and discharging obligations by examining commitments, then it would be very convenient to introduce an augmented type analysis into the overall linking process of the mobile code hosting environment. Specifically, additional versions of the **verify** primitives can be introduced into the linking strategy for conducting intra-modular type analysis for the augmented type system. Inter-modular type safety is enforced by formulating appropriate proof obligations and commitments and by processing them with the open proof linking mechanism. This extension mechanism will greatly cut down the cost of implementing alternative protection mechanisms that are based on static type analysis.

3.4 Research Problems

The Proof Linking architecture, as presented in the previous sections, is a conceptual architecture for structuring the verification service of a mobile code hosting environment. To what extent this architecture is applicable for structuring the verification service of a production mobile code system (**TS1**), and whether the architecture delivers the intended advantages in practice (**TS2**) are the topics of investigation for this research. To operationalize this study, the focus of this research is turned to the implementation of Proof Linking on a production mobile code hosting environment — the Java platform. The Java platform contains enough semantic complexity for testing the modeling adequacy and soundness of Proof Linking. Moreover, the realization of Proof Linking in the JVM represents a concrete contribution to the technical community by enabling the Java platform to support stand-alone verification modules, distributed verification, and augmented type systems. This endeavor involves addressing four research problems listed below.

3.4.1 Modeling Adequacy and Soundness

As an archetypical mobile code protection mechanism, Java bytecode verification is chosen for testing the modeling adequacy and soundness of the Proof Linking architecture. It is hypothesized that, on the one hand, the Proof Linking architecture is rich enough to model the semantic complexity of the Java run-time environment, and, on the other hand, there is a reasonable implementation of Java bytecode verification, in the form of a linking strategy and an appropriate selection of proof obligations and commitments, which satisfies the three correctness conditions outlined in Section 3.2.3, namely, that of Safety, Monotonicity, and Completion. Specifically, a proper instantiation of Proof Linking should account for two distinct complexities of the JVM — lazy, dynamic linking and multiple classloaders.

Lazy, Dynamic Linking

The first complication comes from the fact that the Java dynamic linking semantics is closely tied to its object model and its bytecode verification procedure. Specifically, loading of one class will initiate the loading of classes representing its supertypes, and, as mentioned before, verification also affects the loading schedule of classes. There are complex temporal dependencies among linking primitives that are not accounted for in the prototypical Proof Linking architecture presented in the previous sections. This motivates the first research problem.

> **Research Problem 1:** Can the Proof Linking architecture be correctly instantiated to model Java bytecode verification in the presence of Java's lazy, dynamic linking process?

To address this research problem, the proof linking model was extended to account for Java bytecode verification in the context of a Java-specific dynamic linking model [71, 73]. Specifically, the mentioned temporal dependencies were captured in a relatively lazy linking strategy. A set of proof obligations and commitments was designed to capture the verification interface of Java bytecode verification. This proof linking model was then proven to satisfy the three correctness conditions: Safety, Monotonicity, and Completion. See Chapter 4 for more details of these results.

Multiple Classloaders

The second modeling complication originates from the fact that the standard Java classloading semantics uses multiple classloaders to implement namespace partitioning [130, Chapter 5][127]. This introduces additional dependencies between symbol resolution and the notions of proof obligations and commitments. Specifically, the referents of symbols occurring in proof obligations and commitments might come from various namespaces that are created at run time, while the prototypical Proof Linking architecture in this chapter assumes that there is only one, static namespace. This complication is especially significant in the case of remote verification, in which the remote verifier has no way of addressing symbols that live in the run-time namespaces of the consumer JVM. This motivates a second research problem:

> **Research Problem 2:** Can the Proof Linking architecture be correctly instantiated to model Java bytecode verification in the presence of multiple classloaders?

To address this research problem, the proof linking model was extended to account for the existence of multiple classloaders in Java. The extension preserves the modularity of the verification architecture as well as the correctness of the proof linking process. It turns out that, most of the modeling apparatus can be reused with only minimal modification. See Chapter 5 for details of these results.

3.4.2 Implementation Feasibility

Previous sections promise that the realization of Proof Linking in a mobile code system should bring support for stand-alone verification modules, distributed verification and augmented type systems. This section describes the research problems surrounding the first and third applications of Proof Linking.

Stand-alone Verification Modules

As we will see in Chapters 4 and 5, Java bytecode typechecking introduces significant complexities into the Proof Linking architecture. Whether such complexities can be feasibly addressed in a realistic Java platform is a problem of interest.

> **Research Problem 3:** Is there a feasible implementation of the Proof Linking architecture for modularizing the standard Java bytecode verification process?

Specifically, one may have concerns about the architectural impact of modular verification and incremental proof linking on a JVM, especially in the following two aspects.

1. **Data structures.** It is anticipated that non-trivial data structures will be needed to support the incremental proof linking process. Introduction of complex data structures into an already complex linking process within the JVM may be questionable.

2. **Linking strategies.** The correctness conditions imposed on linking strategies affect the temporal ordering of linking activities. An easy to validate linking strategy for Java bytecode verification may perhaps not be the laziest one allowed by the JVM specification. The degree to which laziness is curtailed may proportionately affect the efficiency of the overall linking process.

To assess how the above two concerns can be addressed in a realistic implementation of the Proof Linking architecture, the Java instantiation of Proof Linking as developed in Chapters 4 and 5 was implemented in an open source JVM, the Aegis VM [68]. A discussion concerning the design of data structures and linking strategies for this implementation can be found in Chapter 6.

Augmented Type Systems

If the proof linking mechanism is generalized to process arbitrary proof obligations and commitments, then a verifier plug-in mechanism can be constructed to support the introduction of alternative link-time verification tasks other than Java bytecode verification.

> **Research Problem 4:** Is it feasible for Proof Linking to be implemented as a general-purpose, efficient, and usable mechanism for servicing user-defined, link-time static analyses?

The key challenges for this endeavor are three:

1. **Generality.** The studies proposed so far are focused on one very specific analysis, namely, that of Java bytecode verification. The generalized Proof Linking framework should be applicable to a wide range of static analyses.

2. **Efficiency.** Since linking events occur frequently in a Java platform, the generalized Proof Linking implementation 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 under the Proof Linking framework.

The real question of feasibility then becomes one of whether there are acceptable tradeoffs in balancing these and other design goals in the implementation of a pluggable verification architecture.

The above research problem has been addressed by two implementation efforts — the implementation of one possible design that is mindful of the issues of generality, efficiency and utility (Chapter 6), and the application of this implementation to realize a security-related type analysis (Chapter 7).

Chapter 4

Lazy, Dynamic Linking

This chapter[1] describes a first instantiation of the Proof Linking architecture for Java bytecode typechecking. Specifically, we will see how the Proof Linking architecture can be instantiated to handle the complexity originating from the lazy, dynamic linking semantics of Java classloading. This instantiation of Proof Linking will be shown to satisfy all the correctness conditions outlined in Section 3.2.3.

4.1 A Simplified View of Java Dynamic Linking

In Java, code units are classfiles, each of which contains the definition of a single class. The standard Java classloading semantics uses multiple classloaders to implement namespace partitioning. A loaded class is identified by both its classname and its defining classloader [130, Chapter 5][127]. Since the focus of this chapter is on analyzing how the deferred discharging of proof obligations interacts with a lazy, dynamic linking process, a simplified view of Java classloading is considered: every class is loaded by the same classloader (i.e., the bootstrap classloader). As a result of this simplification, classnames are sufficient for identifying code units. Complexities introduced by the presence of multiple classloaders are discussed in the next chapter.

[1]Results in this chapter originally appeared in [71, 73].

A symbolic reference in Java may refer to either a class[2] or a member of a class. Class references are simply classnames. Member references refer to either fields or methods. As a class may contain multiple members with the same identifier, both the identifier and the descriptor (i.e., type signature) of a member are generally needed to uniquely denote the member within a class. The descriptor of a field specifies its type, while that of a method specifies the type of both its formal parameters and its return value. A member M of a class X with descriptor S has a symbolic reference of the form $X{::}M(S)$.

4.2 A First Proof Linking Model

This section describes a proof linking model that captures Java verification passes 1 through 3, and integrates them within a single verification primitive[3]. Further improvement could be achieved by formulating commitments and obligations related to the checking of resolution errors — the fourth pass of verification [130, chapter 5]. As this exercise is a conceptually straightforward extension, and since it has less of an impact on enabling other verification protocols (see Section 1.1.2), it is omitted to facilitate presentation.

[2]Although the current discussion assumes a programming model in which all classes are *top-level* classes, it is sufficient for handling Java *nested classes*, which were introduced in Java 1.1 so that programmers can define classes as members of other classes, locally within a block of statements, or within an expression[14, Chapter 5]. However, nested classes are strictly source language constructs implemented entirely by a Java source compiler which transforms all Java 1.1 nested classes into Java 1.0 bytecode. Consequently, the presence of nested classes requires no change to the current scheme, which assumes a standard JVM bytecode semantics.

[3]Because pass 4 (resolution of constant pool entries) is not included in the modelling, and because some of the verification checks are performed entirely within the **verify** primitive, it is not necessary to capture all typing information (e.g., whether a member is abstract) in the current proof linking model. The initial theory, commitments and obligations formulated in Sections 4.2.3 and 4.2.4 reflect only the dependencies between the loader and the verifier.

load X	Acquire the definition of class X.
verify X	Perform bytecode verification on class X.
endorse X	Endorse class X for resolution.
endorse X::$M(S)$	Endorse class member X::$M(S)$ for resolution.
resolve Y **in** X	Resolve class symbol Y in class X.
resolve Y::$M(S)$ **in** X	Resolve member symbol Y::$M(S)$ in class X.

Figure 4.1: Linking Primitives for Modeling Java Typechecking in the Presence of Lazy, Dynamic Linking

4.2.1 Linking Primitives

Figure 4.1 shows the set of linking primitives for modeling Java typechecking in the simplified dynamic linking process. The **load** and **verify** primitives are defined as in Section 3.2.1. Additional linking primitives are introduced. Since class symbols and member symbols are resolved separately, the linking primitive that resolves method Y::$M(S)$ in class X is denoted by "**resolve** Y::$M(S)$ **in** X", and the simpler syntax of "**resolve** Y **in** X" is reserved for resolution of classes. Auxiliary primitives "**endorse** X" and "**endorse** X::$M(S)$" are also introduced, with the intuitive semantics of declaring the corresponding objects to be ready for linking. These primitives serve as placeholders to which one may attach obligations that should be checked before a symbol is resolved. As we shall see in the following, they are used for fine tuning the discharging schedule of proof obligations. Readers may safely identify them with class preparation [130, Section 5.4.2].

Java typechecking can be modeled by assuming that only the **verify** primitives generate proof obligations and commitments. Furthermore, obligations are only attached to the **endorse** and **resolve** primitives.

4.2.2 Linking Strategy

A relatively lazy linking strategy for the above proof linking model is articulated. The Natural Progression Property and the Import-Checked Property are first modified to accommodate the introduction of new primitives, and then further properties are added to capture the linking dependencies peculiar to Java.

1. **Natural Progression Property:**

   $$\textbf{load } X < \textbf{verify } X < \textbf{endorse } X < \textbf{resolve } Y \textbf{ in } X < \textbf{resolve } Y::M(S) \textbf{ in } X$$

2. **Import-Checked Property:**

   $$\textbf{endorse } Y < \textbf{resolve } Y \textbf{ in } X < \textbf{use } Y \textbf{ in } X$$

   and also

   $$\textbf{endorse } Y < \textbf{endorse } Y::M(S) < \textbf{resolve } Y::M(S) \textbf{ in } X < \textbf{use } Y::M(S) \textbf{ in } X$$

3. **Subtype Dependency Property:** To establish an obligation concerning a class, type information concerning its superclasses and superinterfaces might be needed. For example, to show that a class is properly defined, one has to show that none of its superclasses is declared final, and that all of its superinterfaces are properly defined interfaces. To address this need, it is necessary to make sure that the commitments regarding the superclasses and superinterfaces of a class are completely generated before the class is endorsed for linking. Consequently, it is required that, for all superclasses and superinterfaces Y of X,

   $$\textbf{verify } Y < \textbf{endorse } X$$

4. **Referential Dependency Property:** As mentioned in Section 3.4.1, bytecode verification affects the loading schedule of classes. Specifically, if the body of a method $X::M(S)$ refers to a class symbol Y, then it might be necessary to obtain the type information regarding class Y before the type safety of $X::M(S)$ can be endorsed. For example, if method $X::M(S)$ assigns a reference of type Y to a variable of type Z, then Java type rules require Z to be either a superclass or a superinterface of Y. Unless Y is a superclass of X, it is entirely possible that the superclasses and superinterfaces of Y are not verified yet. Consequently, the required supports for the obligation are not necessarily present at the time the obligation is checked, a violation of the Completion property. In such a case, Y is said to be *relevant* to the endorsing of $X::M(S)$. It is assumed that, statically

associated with the bytecode of each method $X{::}M(S)$ is a set of relevant classes Y, and that the following is required[4].

$$\textbf{endorse}\ Y < \textbf{endorse}\ X{::}M(S)$$

That is, all relevant classes (plus their superclasses and superinterfaces) are fully verified before the obligations attached to "**endorse** $X{::}M(S)$" are checked. Since the notion of relevance is statically defined, an implementation may enforce the above ordering constraint by identifying the relevant classes when "**verify** X" scans the bytecode of X. Equipped with this information, the run-time system can consistently endorse all relevant classes before endorsing a method of X.

4.2.3 Initial Theory

A first-order, decidable initial theory is defined for this proof linking model. Specifically, Figure 4.2 shows the set of precidate symbols[5], while Figure 4.3 shows the logic program that captures the axioms of the theory. It is instructive to point out once again, that the logic program in Figure 4.3 does not exhaust all the type rules of the Java bytecode language, for some of the type rules are checked at symbol resolution time (pass 4), which is outside of the scope of this model. That is why one does not find any rule describing, for example, abstract classes. More than that, some of the rules in the initial theory only capture a partial aspect of their counterpart in Java's type system. For example, the `accessible` predicate only captures the notion of accessibility for protected members, and permits access in the cases of public, package

[4] An alternative formulation is to require that:

$$\textbf{verify}\ Y < \textbf{endorse}\ X{::}M(S)$$

for all class Y that is either a relevant class of method $X{::}M(S)$ or a supertype of a class relevant to method $X{::}M(S)$. See footnote 3 on page 115 for a follow up of this variation.

[5] Proof obligations and commitments were formatted in a datalog-style notation in previous publications [71, 73]. They are reformatted here in a style with structures and operators. Doing so improves readability without altering the essence of the scheme.

`class(`X`)`: *X is a non-interface class.*

`interface(`X`)`: *X is an interface class.*

`non_final(`X`)`: *X is not declared to be a final class.*

`package(`P`,`X`)`: *Class X is defined in package P.*

`extends(`X`,`Y`)`: *Y is a direct superclass of X.*

`implements(`X`,`L`)`: *L is the list of all direct superinterfaces of X.*

`public_member(`$M(S)$`,`X`)`: *$M(S)$ is a public member of class X.*

`protected_member(`$M(S)$`,`X`)`: *$M(S)$ is a protected member of class X.*

`package_private_member(`$M(S)$`,`X`)`: *$M(S)$ is a package private member of class X.*

`private_member(`$M(S)$`,`X`)`: *$M(S)$ is a private member of class X.*

`implementable(`L`)`: *The list L contains only properly defined interface symbols.*

`subclassable(`X`)`: *X is a properly defined class.*

`subclass(`X`,`Y`)`: *X is either Y or one of its subclasses.*

`throwable(`X`)`: *X is either* 'java/lang/Throwable' *or one if its subclasses.*

`subinterface(`X`,`Y`)`: *X is either Y or one of its subinterfaces.*

`assignable(`X`,`Y`)`: *X is either Y or one of its subtypes.*

`accessible(`Y`::`$M(S)$`,`X`,`Z`)`: *Y::$M(S)$ is accessible from within class X via a reference of type Z.*

`transitively_implements(`X`,`Y`)`: *This is the "transitive closure" of the* `implements` *relation. This is a helper predicate for defining* `subinterface`.

`list_member(`X`,`L`)`: *The standard Prolog list membership predicate. This is a helper predicate for defining* `transitively_implements`.

Figure 4.2: Predicate Symbols of the Initial Theory Used for Modeling Java Typechecking in the Presence of Lazy, Dynamic Linking

```
implementable([]).
implementable([X | L]) :-
   interface(X),
   implements(X, I),
   implementable(I),
   implementable(L).

subclassable('java/lang/Object').
subclassable(X) :-
   class(X),
   non_final(X),
   implements(X, I),
   implementable(I),
   extends(X, Y),
   subclassable(Y).

throwable(X) :-
   subclass(X, 'java/lang/Throwable').

list_member(X, [X | _]).
list_member(X, [_ | L]) :-
   list_member(X, L).

transitively_implements(X, Y) :-
   implements(X, I),
   list_member(Y, I).
transitively_implements(X, Z) :-
   implements(X, I),
   list_member(Y, I),
   transitively_implements(Y, Z).

subinterface(X, X) :-
   interface(X).
subinterface(X, Z) :-
   subclass(X, Y),
   transitively_implements(Y, Z).

assignable(X, Y) :-
   subclass(X, Y).
assignable(X, Y) :-
   subinterface(X, Y).

accessible(Y::N(T), _, _) :-
   public_member(N(T), Y).
accessible(Y::N(T), X, _) :-
   protected_member(N(T), Y),
   package(P, X),
   package(P, Y).
accessible(Y::N(T), X, Z) :-
   protected_member(N(T), Y),
   subclass(Z, X),
   subclass(X, Y).
accessible(Y::N(T), _, _) :-
   package_private_member(N(T), Y).
accessible(Y::N(T), _, _) :-
   private_member(N(T), Y).
```

Figure 4.3: Axioms of the Initial Theory Used for Modeling Java Typechecking in the Presence of Lazy, Dynamic Linking

`class(`X`)`

`interface(`X`)`

`non_final(`X`)`

`package(`P`, `X`)`

`extends(`X`, `Y`)`

`implements(`X`, `L`)`

`public_member(`$M(S)$`, `X`)`

`protected_member(`$M(S)$`, `X`)`

`package_private_member(`$M(S)$`, `X`)`

`private_member(`$M(S)$`, `X`)`

Figure 4.4: Commitments Generated by **verify** X for Modeling Java Typechecking in the Presence of Lazy, Dynamic Linking

private and private members. The modeling is partial because only the type constraints enforced during the first three passes of bytecode verification are considered. The unmodeled aspects are, again, enforced during pass 4 of the verification process.

4.2.4 Proof Obligations and Commitments

In this proof linking model, only the "**verify** X" primitive generates commitments and obligations. Figure 4.4 enumerates the commitments that may be generated by "**verify** X": The modular bytecode verifier basically scans the loaded classfile, and generate commitments that describe the type interface of the class.

The following list[6] describes the proof obligations generated by "**verify** X", together with the primitives to which the generated obligations are attached:

1. **Obligation**: `subclassable(`Y`)`

   Target: **endorse** X

   Intention: This attachment guarantees that it is safe for the current class X to subclass from the direct superclass Y. The obligation states that the definition of Y does not involve circular subclassing and subinterfacing, subclassing from an interface, and subinterfacing from a class. In the Sun JVM implementation, these checks are performed in pass 1 of bytecode verification. The obligation also makes sure that no final class is subclassed (a pass 2 check in the Sun JVM).

2. **Obligation**: `implementable(`L`)`

   Target: **endorse** X

   Intention: This attachment guarantees that the list L of direct superinterfaces of the current class X are all properly defined interfaces (e.g., no circular implementation, etc). In the Sun JVM implementation, the check is performed in pass 1 of bytecode verification.

3. **Obligation**: `throwable(`Y`)`

   Target: **endorse** X::$M(S)$

   Intention: This attachment is generated when the body of a method X::$M(S)$ contains an *athrow* bytecode instruction. It guarantees that the class Y of the object being thrown as an exception is indeed a subclass of `java.lang.Throwable`. The class Y is statically identified as being relevant to X::$M(S)$. In the Sun JVM, this check is performed in pass 3 of bytecode verification as part of a dataflow analysis.

[6] The second edition of the JVM specification [130] simplifies the type rules of the Java bytecode language originally presented in the first edition [129]. This allows fewer obligations to be generated in the current proof linking model than the one presented in [73], which is already a simplification of the list reported in [71]. A further simplification was adopted by the implementation in Chapter 6.

4. **Obligation**: `subclass(`X`,`Y`)`

   Target: **resolve** Y::$N(T)$ **in** X

   Intention: The JVM specification [130, page 137] states that "each *invokespecial* instruction must name an instance initialization method, a method in the current class, or a method in a superclass of the current class." This attachment is generated for the third case. In the Sun JVM, the check is performed in pass 3 as part of a dataflow analysis.

5. **Obligation**: `assignable(`Y`,`Z`)`

   Target: **endorse** X::$M(S)$

   Intention: This attachment is generated when the body of method X::$M(S)$ contains one of the following bytecode instructions: *putstatic*, *putfield*, *getfield*, *invokestatic*, *invokevirtual*, *invokeinterface*, *invokespecial*, *areturn* and *aastore*. It is generated for the following reasons:

   (a) In the case of *invokestatic*, *invokevirtual*, *invokeinterface* and *invokespecial*, an actual argument of reference type Y must be assignment compatible[7] to its corresponding formal parameter of reference type Z.

   (b) In the case of *getfield*, *putfield*, *invokevirtual*, *invokeinterface* and *invokespecial*, the type Y of the object instance to which the instruction is applied must be assignment compatible to the class Z named in the instruction.

   (c) In the case of *areturn*, the type Y of the object instance that is returned by the instruction must be assignment compatible to the return type Z of X::$M(S)$.

   (d) In the case of *putfield* and *putstatic*, the type Y of the value that is stored by the instruction must be assignment compatible to the type Z of the target field.

[7] In the case of reference types, the notion of assignment compatibility coincide with that of invocation compatibility. Only the notion of assignment compatibility is captured in the current initial theory, so that unnecessary complication caused by redundancy can be avoided.

(e) In the case of *aastore*, the type Y of the value that is stored by the instruction must be assignment compatible to the component type Z of the target array.

In each case, Y is statically identified by the modular verifier as being relevant to X::$M(S)$. In the Sun JVM, this check is performed in pass 3 as part of a dataflow analysis.

6. **Obligation**: `accessible(`Y`::`$N(T)$`,`X`,`Z`)`

   Target: **resolve** Y::$N(T)$ **in** X

   Intention: This attachment is generated when the body of a method X::$M(S)$ contains one of the following: *getfield*, *putfield*, *invokevirtual* and *invokespecial*. It is generated to make sure that it is safe to access member Y::$N(T)$ from within the current class X through an object instance of type Z. The type Z is statically identified to be relevant to X::$M(S)$. In the Sun JVM, this check is performed in pass 3 as part of a dataflow analysis.

4.3 Correctness of the Proof Linking Model

Suppose a JVM can be implemented to generate the proof obligations and commitments described in the previous section, and enforce the ordering constraints of Section 4.2.2. It can be proven that such an implementation satisfies the three correctness conditions in Section 3.2.3.

1. **Safety:** Only "**verify** X" generates obligations. As specified in Section 4.2.4, the generated obligations are only attached to "**endorse** X", "**endorse** X::$M(S)$", "**resolve** Y **in** X", and "**resolve** Y::$M(S)$ **in** X". In each case, however, the proof linker processes these primitives only after the appropriate "**verify** X" primitive, in accord with the Natural Progression and Import-Checked properties.

2. **Monotonicity:** The initial theory and the commitments are all in Horn clause form. Obligations are expressed as definite queries. Monotonicity is thereby assured.

3. **Completion:** Consider the obligation `subclassable(`Y`)` attached to "**endorse** X" by "**verify** X". Supports of this obligation are asserted by all primitives of the form "**verify** Z", where Z is either Y or one of its superclasses or superinterfaces. It then suffices to show that "**verify** Z" < "**endorse** X". According to Section 4.2.4, the obligation is imposed only when Y is declared to be a direct superclass of X. The Subtype Dependency Property guarantees that Y and all its superclasses and superinterfaces are verified before X is endorsed. Therefore, the obligation is consistently established.

   Consider now the obligation `assignable(`Y`,`Z`)` attached to "**endorse** X::$M(S)$". Supports of the obligation are asserted by all primitives of the form "**verify** W", where W is Y or a superclass or superinterface thereof. As shown in Section 4.2.4, Y is relevant to X::$M(S)$ if the the obligation is to be asserted. It then follows from the Referential Dependency Property that the superclasses and superinterfaces of Y are already verified when the obligation is tested. Thus, all the supports are already present, and the obligation can be consistently established.

   Using similar arguments, one can establish that Completion holds for all obligations.

The above reasoning can be formalized and checked using the PVS specification and verification tool [188]. Details are given in Appendix A.

4.4 Summary

This chapter describes how the Proof Linking architecture can be instantiated in a way that captures the complexity of Java's lazy, dynamic linking semantics, and does so without violating the three correctness conditions specified in Section 3.2.3. This is achieved by introducing a new set of linking primitives (**endorse**) for staging the discharging of certain proof obligations, and also imposing two additional ordering constraints (Subtype Dependency Property and Referential Dependency Property) to synchronize the generation of commitments and discharging of proof obligations.

One limitation of the model is that it assumes a single namespace for all symbols. Whether the proof linking model can be extended to account for the complexity of multiple classloaders is the topic of the next chapter.

Chapter 5

Multiple Classloaders

In order to focus on the interplay between incremental proof linking and lazy, dynamic linking, the discussion in the previous chapter assumes a simplified JVM with only one classloader. The assumption is relaxed in this chapter[1], and analyze the interaction between incremental proof linking and lazy, dynamic linking in the setting of multiple classloaders. It turns out that a systematic, straightforward set of extensions to the original model is sufficient to make incremental proof linking work with multiple classloaders. This demonstrates that the Proof Linking Architecture is applicable to realistic mobile code environments and is orthogonal to namespace partitioning with multiple classloaders. Of particular interest is that the extended model is carefully designed to support the distributed verification of Java classfiles (see Section 3.3.2).

5.1 Enter Multiple Classloaders

In standard Java platforms, multiple namespaces can be created by defining multiple classloaders. A Java class is identified not only by its name, but by both its name and the classloader in which the class is defined. Formally, when a Java application attempts to load a class C with class name X using a classloader L_i, L_i may delegate

[1]Results in this chapter first appeared in [72, 74].

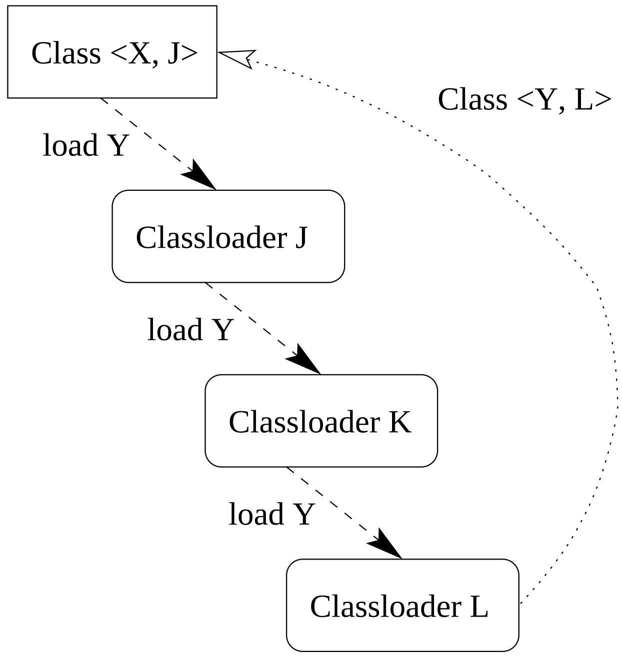


Figure 5.1: Java Delegation Style Classloading

the classloading task to another classloader, which, in turn, might delegate the task to yet another classloader. The classloader L_i is called the *initiating classloader* of class C. The classloader L_d that eventually loads and defines C is said to be its *defining classloader*. C is uniquely identified by the pair $\langle X, L_d \rangle$. We also write $X^{L_i} \mapsto \langle X, L_d \rangle$ to indicate the fact that class $\langle X, L_d \rangle$ is bound to the symbol X in the namespace associated with classloader L_i as a result of L_i initiating the loading of $\langle X, L_d \rangle$.

When a symbolic reference Y is resolved in a class $\langle X, J \rangle$, the classloader J will be used as the initiating classloader for class Y. Doing so guarantees that the loading of classes referenced in class $\langle X, L \rangle$ is consistently initiated by the same classloader that defines the class. The situation is depicted in Figure 5.1, in which the defining classloader J of loaded class $\langle X, J \rangle$ is used for initiating the loading of a class with name Y. Classloader J, delegates the loading request to classloader K, which in turn delegates the loading request to classloader L, the classloader that finally defines the class as $\langle Y, L \rangle$. As a result, the following name bindings materialize:

$$Y^J \mapsto \langle Y, L \rangle, \quad Y^K \mapsto \langle Y, L \rangle, \quad Y^L \mapsto \langle Y, L \rangle.$$

Details of Java's classloading mechanism are described elsewhere [127][130, Chapter 5].

The notation $\langle P, L\rangle$ is used to denote loaded package with package name P and defining classloader L.

5.2 An Extended Proof Linking Model

5.2.1 Overview of the Solution Approach

A naive attempt to account for the complexity introduced by multiple classloaders would be to replace each class reference Y appearing in commitments and obligations by a classname-classloader pair $\langle Y, K\rangle$, where K is the defining classloader of the referenced class. Unfortunately, this naive approach would not work. Suppose that the commitment or obligation mentioning symbol Y is generated when a class $\langle X, J\rangle$ is being verified. There is no guarantee that the class designated by the class symbol Y has already been properly loaded before class $\langle X, J\rangle$ is verified (recall that the goal is to avoid recursive classloading). In the adversarial case, the defining classloader for Y is simply not known yet, and so the naive approach will break down for the obvious reason. Fortunately, the initiating classloader for class symbol Y is already known: classloader J will be used for initiating the loading of class reference Y. Consequently, if the name resolution process can be explicitly mirrored in the discharging of proof obligations, then the previous proof linking model can be reused. The proof linking model as presented in the previous chapter will be extended according to the following strategy:

1. The modular verifier formulates commitments and obligations not in terms of loaded classes, but in terms of symbolic class references. This removes the necessity of identifying unknown defining classloaders.

2. Before commitments are asserted into the commitment database, and obligation attachments are recorded in the obligation table, they are tagged with the

load $\langle X, J\rangle$	Acquire the definition of a class with name X with the defining classloader J.
verify $\langle X, J\rangle$	Assess the safety of the bytecode in loaded class $\langle X, J\rangle$.
bind X^L **to** $\langle X, J\rangle$	Bind the class symbol X in the namespace of classloader L to the loaded class $\langle X, J\rangle$. That is, classloader L becomes an initiating classloader of X.
endorse $\langle X, J\rangle$	Endorse the loaded class $\langle X, J\rangle$ for resolution.
endorse $\langle X{::}M(S), J\rangle$	Endorse the loaded member $\langle X{::}M(S), J\rangle$ for resolution.
resolve Y **in** $\langle X, J\rangle$	Resolve the class symbol Y in loaded class $\langle X, J\rangle$.
resolve $Y{::}M(S)$ **in** $\langle X, J\rangle$	Resolve the member symbol $Y{::}M(S)$ in loaded class $\langle X, J\rangle$.

Figure 5.2: The Extended Set of Linking Primitives for Modeling Java Typechecking in the Presence of Multiple Classloaders

initiating classloaders of the symbolic class references. This provides the context in which to perform symbol resolution during the incremental proof linking process.

3. Name binding events are explicitly modeled as linking primitives generating binding commitments. This allows us to access the name binding information of the JVM.

4. Translation rules are introduced into the initial theory for explicit resolution of tagged symbolic references to loaded classes using binding commitments. This allows us to mirror JVM name resolution in the incremental proof linking process.

The above plan will be carried out step by step in the following sections.

5.2.2 Linking Primitives

Let us begin the discussion of the extended proof linking model by looking at its linking primitives, which can be found in Figure 5.2. Two changes to the original primitive set have been made:

1. The **load**, **verify**, **endorse** and **resolve** primitives have been adapted to refer to loaded classes rather than simple class names.

2. A new family of **bind** primitives has been introduced. They model the explicit binding of loaded classes to symbols defined in the local namespace of a classloader. When the JVM binds the loaded class $\langle X, J\rangle$ to the symbol X in an initiating classloader L, the primitive "**bind** X^L **to** $\langle X, J\rangle$" is executed. It is assumed that the JVM will execute at most one "**bind** X^L **to** $\langle X, J\rangle$" for each symbol X in classloader L. The special binding primitive "**bind** X^L **to** $\langle X, L\rangle$" represents the definition of class $\langle X, L\rangle$ by classloader L.

In addition, the **use** primitives are no longer considered in the extended proof linking model, as doing so does not add further insight into the discussion.

5.2.3 Static Type Rules

Let us begin with the static type rules formulated in the previous chapter. To reason about loaded classes in multiple namespaces instead of class names in a single namespace, the naive approach mentioned in Section 5.2.1 was first adopted, the result of which was then enriched according to the solution plan. To do so, class names are uniformly replaced by loaded class notations. For example, consider the subclassing rule mentioned in the previous chapter:

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

This rule is rewritten as follows:

```
subclass(⟨X, J⟩, ⟨X, J⟩).
subclass(⟨X, J⟩, ⟨Y, K⟩) :-
  extends(⟨X, J⟩, ⟨Z, L⟩),
  subclass(⟨Z, L⟩, ⟨Y, K⟩).
```

```
implementable([]).
implementable([⟨X, J⟩ | L]) :-
  interface(⟨X, J⟩),
  implements(⟨X, J⟩, I),
  implementable(I),
  implementable(L).

subclassable(⟨'java/lang/Object', ∅⟩).
subclassable(⟨X, J⟩) :-
  class(⟨X, J⟩),
  non_final(⟨X, J⟩),
  implements(⟨X, J⟩, I),
  implementable(I),
  extends(⟨X, J⟩, ⟨Y, K⟩),
  subclassable(⟨Y, K⟩).

subclass(⟨X, J⟩, ⟨X, J⟩).
subclass(⟨X, J⟩, ⟨Y, K⟩) :-
  extends(⟨X, J⟩, ⟨Z, L⟩),
  subclass(⟨Z, L⟩, ⟨Y, K⟩).

throwable(⟨X, J⟩) :-
  subclass(⟨X, J⟩, ⟨'java/lang/Throwable', ∅⟩).

transitively_implements(⟨X, J⟩, ⟨Y, K⟩) :-
  implements(⟨X, J⟩, I),
  list_member(⟨Y, K⟩, I).
transitively_implements(⟨X, J⟩, ⟨Y, K⟩) :-
  implements(⟨X, J⟩, I),
  list_member(⟨Z, L⟩, I),
  transitively_implements(⟨Z, L⟩, ⟨Y, K⟩).

subinterface(⟨X, J⟩, ⟨X, J⟩) :-
  interface(⟨X, J⟩).
subinterface(⟨X, J⟩, ⟨Y, K⟩) :-
  subclass(⟨X, J⟩, ⟨Z, L⟩),
  transitively_implements(⟨Z, L⟩, ⟨Y, K⟩).

assignable(⟨X, J⟩, ⟨Y, K⟩) :-
  subclass(⟨X, J⟩, ⟨Y, K⟩).
assignable(⟨X, J⟩, ⟨Y, K⟩) :-
  subinterface(⟨X, J⟩, ⟨Y, K⟩).

accessible(⟨Y::M(S), K⟩, _, _) :-
  public_member(⟨Y::M(S), K⟩, _)
accessible(⟨Y::M(S), K⟩, ⟨X, J⟩, _) :-
  protected_member(⟨Y::M(S), K⟩, ,)
  package(⟨P, L⟩, ⟨X, J⟩),
  package(⟨P, L⟩, ⟨Y, K⟩).
accessible(⟨Y::M(S), K⟩, ⟨X, J⟩, ⟨Z, L⟩) :-
  protected_member(⟨Y::M(S), K⟩, ,)
  subclass(⟨X, J⟩, ⟨Y, K⟩),
  subclass(⟨Z, L⟩, ⟨X, J⟩).
accessible(⟨Y::M(S), K⟩, _, _) :-
  package_private_member(⟨Y::M(S), K⟩, _)
accessible(⟨Y::M(S), K⟩, _, _) :-
  private_member(⟨Y::M(S), K⟩, _)
```

Figure 5.3: Axioms in the Initial Theory Used for Modeling Java Typechecking in the Presence of Multiple Classloaders

`class(`$\langle X, J \rangle$`)`
`interface(`$\langle X, J \rangle$`)`
`non_final(`$\langle X, J \rangle$`)`
`package(`$\langle P, L \rangle$`, `$\langle X, J \rangle$`)`
`extends(`$\langle X, J \rangle$`, `$\langle Y, K \rangle$`)`
`implements(`$\langle X, J \rangle$`, `I`)`
`public_member(`$M(S)$`, `$\langle X, J \rangle$`)`
`protected_member(`$M(S)$`, `$\langle X, J \rangle$`)`
`package_private_member(`$M(S)$`, `$\langle X, J \rangle$`)`
`private_member(`$M(S)$`, `$\langle X, J \rangle$`)`

Figure 5.4: Foundational Queries Used for Modeling Java Typechecking in the Presence of Multiple Classloaders

The reformulation[2] of rules is entirely mechanical. A list of all the reformulated rules is presented in Figure 5.3. The evaluation of the static type rules above requires ability to evaluate the query forms in Figure 5.4, a topic to which we will turn next.

5.2.4 Commitment Assertion

Suppose that a classfile with class name X is being verified, and that X extends a class with name Y. The verifier must assert a commitment specifying this subclassing relationship. However, because the actual verification of the classfile may happen remotely (Section 3.3.2), the defining classloaders for classes X and Y are not known at the time of verification. Consequently, the commitment cannot be phrased in terms of loaded classes. To address the above problems, three reformulations are introduced below.

Commitment Formulation and Tagging

First, the generation of commitments is separated into a two stage process, involving (1) commitment formulation and (2) commitment tagging. The modular verification procedure formulates the mentioned subclassing commitment in the following form:

[2] In fact, the original rule can be used here without change. A more stylized version is adopted here to draw attention to the difference in argument types of the predicate symbols.

```
class(this)

interface(this)

non_final(this)

package(P,this)

extends(this,Y)

implements(this,L)

public_member(M(S),this)

protected_member(M(S),this)

package_private_member(M(S),this)

private_member(M(S),this)

relevant(Y,M(S),this)
```

Figure 5.5: Commitments For Modeling Typechecking in the Presence of Multiple Classloaders

```
bind_class_list([],[]) @ ⟨X,J⟩.
bind_class_list([Y | H],[⟨Y,K⟩ | I]) @ ⟨X,J⟩ :-
    Y^J ↦ ⟨Y,K⟩,
    bind_class_list(H,I) @ ⟨X,J⟩.
```

Figure 5.6: Rules for Resolving a List of Class Symbols

```
class(⟨X, J⟩) :-
   class(this) @ ⟨X, J⟩.

interface(⟨X, J⟩) :-
   interface(this) @ ⟨X, J⟩.

non_final(⟨X, J⟩) :-
   non_final(this) @ ⟨X, J⟩.

package(⟨P, J⟩, ⟨X, J⟩) :-
   package(P, this) @ ⟨X, J⟩.

extends(⟨X, J⟩, ⟨Y, K⟩) :-
   extends(this, Y) @ ⟨X, J⟩,
   Y^J ↦ ⟨Y, K⟩.

implements(⟨X, J⟩, I) :-
   extends(this, H) @ ⟨X, J⟩,
   bind_class_list(H, I) @ ⟨X, J⟩.

public_member(M(S), ⟨X, J⟩) :-
   public_member(M(S), this) @ ⟨X, J⟩.

protected_member(M(S), ⟨X, J⟩) :-
   protected_member(M(S), this) @ ⟨X, J⟩.

package_private_member(M(S), ⟨X, J⟩) :-
   package_private_member(M(S), this) @ ⟨X, J⟩.

private_member(M(S), ⟨X, J⟩) :-
   private_member(M(S), this) @ ⟨X, J⟩.
```

Figure 5.7: Translation Rules for Resolving Symbols in Commitments

`extends(this,` Y`)`

Notice that symbolic references such as Y are used rather than actual loaded classes. The relative reference `this` represents the class being verified. Figure 5.5 shows a complete reformulation of the commitments originally found in Figure 4.4. The reformulation is straightforward and mechanical. Notice that a new commitment, "`relevant(`Y`,` $M(S)$`, this)`", has been introduced for identifying class Y to be relevant to the endorsing of method $M(S)$ in `this`. Such an explicit representation of relevance relationship is necessary for articulating linking strategies involving *conditional ordering constraints*, details of which to follow in Section 5.2.6.

Before the commitments are actually asserted into the commitment database by the "**verify** $\langle X, J\rangle$" primitive, they undergo a tagging process in which information is attached to each commitment so as to identify the initiating classloader that will be used for resolving the symbolic references occurring in the commitment. Specifically, whenever "**verify** $\langle X, J\rangle$" asserts a commitment p, it *systematically* tags the commitment as p `@` $\langle X, J\rangle$. For example, the subclassing commitment above will be asserted as:

`extends(this,` Y`) @` $\langle X, J\rangle$

Similar tagging is systematically applied to all the commitments in Figure 5.5 before they are actually asserted into the commitment database.

Binding Commitments

Second, execution of the **bind** primitive contributes binding information by asserting commitments. Whenever a "**bind** X^L **to** $\langle X, J\rangle$" primitive terminates, it asserts the commitment "$X^L \mapsto \langle X, J\rangle$". These facts will be used for explicit resolution of symbols in commitments and queries. In order to facilitate resolving lists of class symbols, the rule in Figure 5.6 is also added into the initial theory.

Translation Rules

Third, the initial theory is augmented with translation rules that express how subgoals expressed in terms of loaded class notations (Figure 5.4) may be satisfied using tagged commitments (Figure 5.5). For example, evaluating queries of the form `subclass(`$\langle X, J\rangle$`,` $\langle Y, K\rangle$`)` causes subgoals of the form `extends(`$\langle X, J\rangle$`,` $\langle Y, K\rangle$`)` to be generated. To satisfy these subgoals using tagged commitments, the following translation rule is used.

$$
\begin{array}{l}
\texttt{extends(}\langle X, J\rangle\texttt{,}\ \langle Y, K\rangle\texttt{)} \text{ :-} \\
\quad \texttt{extends(this,}\,Y\texttt{)}\ \texttt{@}\ \langle X, J\rangle, \\
\quad Y^J \mapsto \langle Y, K\rangle.
\end{array}
$$

Basically, the rule retrieves the corresponding tagged commitment, and then validates binding information by consulting binding commitments. A similar translation rule is required for each predicate that may be asserted as a commitment. The formulation of these rules is straightforward, and the complete set is given in Figure 5.7.

The three reformulations above address the need for modular and remote verification. The use of symbolic references in commitment formulation and the explicit modeling of name resolution in the incremental proof linking process allow a verifier to operate without explicitly identifying defining classloaders. Separating commitment

$\texttt{subclassable(}Y\texttt{)} \mathbin{\texttt{@}} \langle X, J\rangle$:-
  $Y^J \mapsto \langle Y, K\rangle$,
  $\texttt{subclassable(}\langle Y, K\rangle\texttt{).}$

$\texttt{implementable(}I\texttt{)} \mathbin{\texttt{@}} \langle X, J\rangle$:-
  $\texttt{bind\_class\_list(}I\texttt{,}\ H\texttt{)} \mathbin{\texttt{@}} \langle X, J\rangle$
  $\texttt{implementable(}H\texttt{).}$

$\texttt{throwable(}Y\texttt{)} \mathbin{\texttt{@}} \langle X, J\rangle$:-
  $Y^J \mapsto \langle Y, K\rangle$,
  $\texttt{throwable(}\langle Y, K\rangle\texttt{).}$

$\texttt{subclass(}Y\texttt{,}\ Z\texttt{)} \mathbin{\texttt{@}} \langle X, J\rangle$:-
  $Y^J \mapsto \langle Y, K\rangle$,
  $Z^J \mapsto \langle Z, L\rangle$,
  $\texttt{subclass(}\langle Y, K\rangle\texttt{,}\ \langle Z, L\rangle\texttt{).}$

$\texttt{assignable(}Y\texttt{,}\ Z\texttt{)} \mathbin{\texttt{@}} \langle X, J\rangle$:-
  $Y^J \mapsto \langle Y, K\rangle$,
  $Z^J \mapsto \langle Z, L\rangle$,
  $\texttt{assignable(}\langle Y, K\rangle\texttt{,}\ \langle Z, L\rangle\texttt{).}$

$\texttt{accessible(}Y\texttt{::}M(S)\texttt{,}\ X\texttt{,}\ Z\texttt{)} \mathbin{\texttt{@}} \langle X, J\rangle$:-
  $X^J \mapsto \langle X, J\rangle$,
  $Y^J \mapsto \langle Y, K\rangle$,
  $Z^J \mapsto \langle Z, L\rangle$,
  $\texttt{accessible(}\langle Y\texttt{::}M(S), K\rangle\texttt{,}\ X^J\texttt{,}\ Z^L\texttt{).}$

Figure 5.8: Translation Rules for Resolving Symbols in Obligations

generation into a two-staged process consisting of a commitment formulation stage and a commitment tagging stage also facilitates remote verification. Specifically, the commitment formulation stage can be performed remotely by a certification service, while the commitment tagging stage can be performed locally by the **verify** primitives of a JVM.

5.2.5 Obligation Attachment

As with the commitments, the **verify** primitive cannot identify the defining classloader for the classes appearing in obligations. Following a similar strategy, proof obligations are formulated in terms of symbolic class names, and subsequently tagged with the context in which they are to be evaluated. For example, the verifier may formulate an obligation of the following form:

$\texttt{subclass(}Y\texttt{,}\ Z\texttt{)}$

The obligation is subsequently tagged with an evaluation context before attachment:

$\texttt{subclass(}Y\texttt{,}\ Z\texttt{)}\ \texttt{@}\ \langle X, J\rangle$

Similar tagging is systematically applied to each obligation in Section 4.2.4.

Again, in order to evaluate the tagged obligations, one has to provide translation rules that transform tagged queries into queries in terms of loaded classes. For example, the following rule is required in the initial theory in order to handle all $\texttt{subclass(}\cdot\texttt{,}\ \cdot\texttt{)}$ queries:

$$
\begin{array}{l}
\texttt{subclass(}Y\texttt{,}\ Z\texttt{)}\ \texttt{@}\ \langle X, J\rangle \ \text{:-} \\
\quad Y^J \mapsto \langle Y, K\rangle, \\
\quad Z^J \mapsto \langle Z, L\rangle, \\
\quad \texttt{subclass(}\langle Y, K\rangle\texttt{,}\ \langle Z, L\rangle\texttt{).}
\end{array}
$$

The translation rule basically resolves all the symbols in the tagged context, and evaluates a corresponding query in terms of loaded classes. A similar translation rule is formulated for each of the obligations stated in Section 4.2.4. The complete set is presented in Figure 5.8.

Recall that, in the running example (Figure 3.1), the above subclassing obligation is attached to the primitive "**resolve** $Z{::}M(S)$ **in** $\langle X, J\rangle$", which is identified by the loaded class reference $\langle X, J\rangle$. If verification is performed remotely, then the defining classloader for the class being examined will not be available, and as a result, the remote verifier cannot completely identify the target primitives to which the obligation is attached. Fortunately, obligations are always attached to primitives that are operating on the class being verified (see Section 4.2.4). The remote verifier may thus formulate the target of attachment in terms of placeholders:

resolve $Z{::}M(S)$ **in** $\_\_\_$

and rely on the local "**verify** $\langle X, J\rangle$" primitive to fill in $\langle X, J\rangle$, as it does when tagging obligations.

5.2.6 Linking Strategy

The linking strategy presented in the previous chapter involves certain ordering constraints whose semantics depends on the static relationship among classes and/or methods. Specifically, the Subtype Dependency Property is defined in terms of the subtyping relationship among classes, whereas the semantics of the Referential Dependency Property is based on the static relevance relationship. The formulation is well defined because a static type environment is assumed (see Appendix A). When dynamic classloading is introduced, the definition of such *conditional ordering constraints* needs to be rigorously specified in order not to introduce semantic confusion. To this end, the notations used for expressing ordering constraints are extended. As usual, given linking primitives p and q, the syntactic constraint "$p < q$" requires that any execution of primitive q should be preceded by the completion of primitive p. Given further a query g, the *conditional ordering constraint* "$p < q$ if g" requires that execution of q must not begin if g is satisfiable and p has not yet completed. More precisely, the notation represents the property that an execution trace τ satisfies one of the following conditions:

- "**begin** q" does not occur in τ;
- "**begin** q" occurs in τ after an occurrence of "**end** p".
- "**begin** q" occurs in τ. Let S be the set of primitives whose **end** events occur in τ before "**begin** q". The following judgment does not hold:

  $$\Gamma \cup \bigcup_{r \in S} com(r) \vdash g$$

  That is, the condition g is not provable from the initial theory and the commitments that are asserted into the commitment database prior to the occurrence of "**begin** q".

The following ordering constraints are imposed on the linking primitives. Except for the newly introduced *Proper Resolution Property*, the rest are refinement to those ordering constraints found in the previous chapter (Section 4.2.2).

1. **Natural Progression Property:** The natural life cycle of a class $\langle X, J\rangle$ is reflected in the ordering below:

$$\textbf{load } \langle X, J\rangle < \textbf{verify } \langle X, J\rangle < \textbf{bind } X^J \textbf{ to } \langle X, J\rangle$$
$$< \textbf{endorse } \langle X, J\rangle < \textbf{resolve } Y \textbf{ in } \langle X, J\rangle < \textbf{resolve } Y{::}M(S) \textbf{ in } \langle X, J\rangle$$

2. **Proper Resolution Property:** The defining classloader of a loaded class is used for resolving the symbolic references of the class:

$$\textbf{bind } Y^J \textbf{ to } \langle Y, K\rangle < \textbf{resolve } Y \textbf{ in } \langle X, J\rangle$$

Delegation of classloading bottoms out when a classloader defines the requested class:

$$\textbf{bind } Y^K \textbf{ to } \langle Y, K\rangle < \textbf{bind } Y^J \textbf{ to } \langle Y, K\rangle$$

3. **Import-Checked Property:** Resolving a symbolic reference requires that the target object is well-defined:

$$\textbf{endorse } \langle Y, K\rangle < \textbf{resolve } Y \textbf{ in } \langle X, J\rangle \qquad \text{if } Y^J \mapsto \langle Y, K\rangle$$

$$\textbf{endorse } \langle Y, K\rangle < \textbf{endorse } \langle Y{::}M(S), K\rangle$$
$$< \textbf{resolve } Y{::}M(S) \textbf{ in } \langle X, J\rangle \;\; \text{if } Y^J \mapsto \langle Y, K\rangle$$

4. **Subtype Dependency Property:** To establish an obligation concerning a class, type information about its superclasses and superinterfaces might be needed. For example, to establish that the direct superclass Y^J of a loaded class $\langle X, J\rangle$ is subclassable (i.e., `subclassable(`Y`)` `@` $\langle X, J\rangle$), all superclasses and superinterfaces of $\langle X, J\rangle$ are required to be loaded, verified and bound before $\langle X, J\rangle$ is used. To address this need, it is required that

$$\textbf{bind } Y^L \textbf{ to } \langle Y, K\rangle < \textbf{endorse } \langle X, J\rangle \qquad \text{if } \texttt{subtypedependent(}Y^L\texttt{)}\ \texttt{@}\ \langle X, J\rangle$$

where the conditional query is handled by the following rules in the initial theory:

```
subtypedependent(X^J) @ ⟨X, J⟩.
subtypedependent(Y^L) @ ⟨X, J⟩ :-
    subtypedependent(Z^K) @ ⟨X, J⟩,
    Z^K ↦ ⟨Z, L⟩,
    extends(this, Y) @ ⟨Z, L⟩.

subtypedependent(Y^L) @ ⟨X, J⟩ :-
    subtypedependent(Z^K) @ ⟨X, J⟩,
    Z^K ↦ ⟨Z, L⟩,
    implements(this, I) @ ⟨Z, L⟩,
    list_member(Y, I).
```

5. **Referential Dependency Property:** Sometimes, verification of a class Y is needed before a method $\langle X{::}M(S), J\rangle$ can be safely endorsed. For example, if method $\langle X{::}M(S), J\rangle$ assigns a reference of type Y to a variable of type Z, then Java type rules require Z to be either a superclass or a superinterface of Y. Unless Y is a superclass of X, it is entirely possible that the superclasses and superinterfaces of Y are not verified yet. Consequently, the required supporting commitments for the obligation are not necessarily present at the time the obligation is checked, a violation of the Completion Condition. In such a case, Y is said to be *relevant* to the endorsing of $\langle X{::}M(S), J\rangle$. It is assumed that, verification of method $\langle X{::}M(S), J\rangle$ generates commitments `relevant(`Y`,` $M(S)$`, this) @` $\langle X, J\rangle$ for all relevant class symbols Y. The following is then required[3].

$$\textbf{endorse}\ \langle Y, K\rangle < \textbf{endorse}\ \langle X{::}M(S), J\rangle \qquad \text{if } \texttt{relevant(}Y\texttt{,}\ M(S)\texttt{, this) @}\ \langle X, J\rangle$$

[3] An alternative formulation of this requirement is the following ordering constraint.

$$\textbf{bind}\ Y^K\ \textbf{to}\ \langle Y, L\rangle < \textbf{endorse}\ \langle X{::}M(S), J\rangle \qquad \text{if } \texttt{referentialdependent(}Y^K\texttt{) @}\ \langle X, J\rangle$$

where the conditional query is handled by the following rules in the initial theory:

```
referentialdependent(Y^J) @ ⟨X, J⟩ :-
    relevant(Y, M(S), this) @ ⟨X, J⟩.

referentialdependent(Z^L) @ ⟨X, J⟩ :-
    relevant(Y, M(S), this) @ ⟨X, J⟩,
    Y^J ↦ ⟨Y, K⟩,
    subtypedependent(Z^L) @ ⟨Y, K⟩.
```

This variant of the Referential Dependency Property achieves the same goal as the original version. It was not adopted in previous works [72, 73] because of its convoluted formulation. Nevertheless, this slightly lazier variant was eventually implemented in the Aegis VM (Chapter 6). The Completion condition is preserved because the **endorse** primitive does not generate commitments.

1. `subclass(`C`,` A`)` `@` $\langle C, L_3\rangle$	/* **resolve** A::$M(S)$ **in** $\langle C, L_3\rangle$ */
1.1. $C^{L_3} \mapsto \langle C, L_3\rangle$	/* **bind** C^{L_3} **to** $\langle C, L_3\rangle$ */
1.2. $A^{L_3} \mapsto \langle A, L_1\rangle$	/* **bind** A^{L_3} **to** $\langle A, L_1\rangle$ */
1.3. `subclass(`$\langle C, L_3\rangle$`,` $\langle A, L_1\rangle$`)`	
1.3.1. `extends(`$\langle C, L_3\rangle$`,` $\langle B, L_2\rangle$`)`	
1.3.1.1. `extends(this,` B`)` `@` $\langle C, L_3\rangle$	/* **verify** $\langle B, L_2\rangle$ */
1.3.1.2. $B^{L_3} \mapsto \langle B, L_2\rangle$	/* **bind** B^{L_3} **to** $\langle B, L_2\rangle$ */
1.3.2. `subclass(`$\langle B, L_2\rangle$`,` $\langle A, L_1\rangle$`)`	
1.3.2.1. `extends(`$\langle B, L_2\rangle$`,` $\langle A, L_1\rangle$`)`	
1.3.2.1.1. `extends(this,` A`)` `@` $\langle B, L_2\rangle$	/* **verify** $\langle B, L_2\rangle$ */
1.3.2.1.2. $A^{L_2} \mapsto \langle A, L_1\rangle$	/* **bind** A^{L_2} **to** $\langle A, L_1\rangle$ */
1.3.2.2. `subclass(`$\langle A, L_1\rangle$`,` $\langle A, L_1\rangle$`)`	

Figure 5.9: Subgoals generated by evaluating `subclass(`C`,` A`)` `@` $\langle C, L_3\rangle$

That is, the commitments for all relevant classes (plus their superclasses and superinterfaces) are collected before the obligations attached to "**endorse** $\langle X$::$M(S), J\rangle$" are checked.

5.2.7 Putting It All Together

To illustrate how the scheme works, consider a refinement of the running example (Figure 3.1). Suppose class $\langle A, L_1\rangle$ defines a method $M(S)$. Suppose further that $\langle A, L_1\rangle$ has a direct subclass $\langle B, L_2\rangle$, which in turn has a direct subclass $\langle C, L_3\rangle$. Assume that $\langle C, L_3\rangle$ overrides the method $M(S)$. Say the loaded method $\langle C$::$M(S), L_3\rangle$ contains an *invokespecial* instruction that delegates the call to $\langle A$::$M(S), L_1\rangle$. The obligation `subclass(`C`,` A`)` `@` $\langle C, L_3\rangle$ will be attached to the primitive "**resolve** A::$M(S)$ **in** $\langle C, L_3\rangle$" (Section 4.2.4). When the obligation is discharged, the subgoals in Figure 5.9 will be generated. The original obligation is shown as the top-level goal, annotated with "**resolve** A::$M(S)$ **in** $\langle C, L_3\rangle$", the primitive to which the obligation is attached. Also, all the innermost subgoals are annotated with the primitives that assert their matching commitments.

The deduction is successful because the commitments required by the innermost subgoals are already asserted at the time the obligation is checked, that is, at the time "**resolve** A::$M(S)$ **in** $\langle C, L_3\rangle$" is executed. For example, subgoal 1.1 is satisfiable because, according to the Natural Progression Property, the primitive "**bind** C^{L_3} **to** $\langle C, L_3\rangle$" has already been executed. Also, subgoal 1.2 is satisfiable because

$$\begin{array}{llr} \textbf{bind } A^{L_3} \textbf{ to } \langle A, L_1\rangle & < \textbf{resolve } A \textbf{ in } \langle C, L_3\rangle & \text{(Proper Resolution)} \\ & < \textbf{resolve } A{::}M(S) \textbf{ in } \langle C, L_3\rangle & \text{(Natural Progression)} \end{array}$$

The rest of the subgoals are more interesting. Note that `subtypedependent(`B^{L_3}`)` `@` $\langle C, L_3\rangle$ is satisfiable before "**resolve** A::$M(S)$ **in** $\langle C, L_3\rangle$" is executed. By applying the Subtype Dependency Property and other ordering constraints, the following can be deduced.

$$\begin{array}{llr} \textbf{verify } \langle B, L_2\rangle & < \textbf{bind } B^{L_2} \textbf{ to } \langle B, L_2\rangle & \text{(Natural Progression)} \\ & < \textbf{bind } B^{L_3} \textbf{ to } \langle B, L_2\rangle & \text{(Proper Resolution)} \\ & < \textbf{endorse } \langle C, L_3\rangle & \text{(Subtype Dependency)} \\ & < \textbf{resolve } B{::}M(S) \textbf{ in } \langle C, L_3\rangle & \text{(Natural Progression)} \end{array}$$

That is, the commitments `extends(this,` B`)` `@` $\langle C, L_3\rangle$ and $B^{L_3} \mapsto \langle B, L_2\rangle$ (generated by "**verify** $\langle B, L_2\rangle$" and "**bind** B^{L_3} **to** $\langle B, L_2\rangle$" respectively) are already in place when the obligation is checked. Therefore, subgoals 1.3.1.1. and 1.3.1.2. are necessarily satisfiable. Similar reasoning applies to subgoals 1.3.2.1.1. and 1.3.2.1.2.

This example is really a skeleton for the proof of Completion, one of the three correctness criteria for proof linking. Detailed correctness justification of Java proof linking with multiple classloaders will be the topic of the next section.

5.3 Correctness

Recall that, given a well-defined linking strategy, proof linking is correct if the three correctness conditions can be established: Safety, Monotonicity and Completion (Section 3.2.3).

5.3.1 Consistency of the Linking Strategy

The strict partial order imposed by the linking strategy above is well-defined. To see this, consider the following linearization of the linking primitives:

1. "**load** $\langle X, J\rangle$" for all classnames X and classloaders J
2. "**verify** $\langle X, J\rangle$" for all loaded class $\langle X, J\rangle$
3. "**bind** X^J **to** $\langle X, J\rangle$" for all loaded class $\langle X, J\rangle$
4. "**bind** X^L **to** $\langle X, J\rangle$" for all loaded class $\langle X, J\rangle$ and classloader L such that $J \neq L$
5. "**endorse** $\langle X, J\rangle$" for all loaded class $\langle X, J\rangle$
6. "**endorse** $\langle X{::}M(S), J\rangle$" for all loaded member $\langle X{::}M(S), J\rangle$
7. "**resolve** Y **in** $\langle X, J\rangle$" for all class symbols Y and loaded class $\langle X, J\rangle$
8. "**resolve** $Y{::}M(S)$ **in** $\langle X, J\rangle$" for all member references $Y{::}M(S)$ and loaded class $\langle X, J\rangle$.

It is easy to check that this linearization satisfies all the ordering constraints imposed by the linking strategy in Section 5.2.6. The linking strategy is therefore consistent.

5.3.2 Safety and Monotonicity

Establishment of the Safety and Monotonicity properties follows the corresponding arguments for the single-classloader case (Section 4.3).

1. **Safety:** Only **verify** primitives generate obligations. A "**verify** $\langle X, J\rangle$" primitive only attaches obligations to "**endorse** $\langle X, J\rangle$", "**endorse** $\langle X{::}M(S), J\rangle$", "**resolve** Y **in** $\langle X, J\rangle$" and "**resolve** $Y{::}M(S)$ **in** $\langle X, J\rangle$", all of which are ordered after "**verify** $\langle X, J\rangle$" by the Natural Progression Property. Therefore, once a primitive begins execution, no additional unchecked obligation will be attached to it.

(α-0)	$X_0^{L_0} \mapsto \langle X_0, L_0\rangle$	**bind** $X_0^{L_0}$ **to** $\langle X_0, L_0\rangle$
(γ)	$X_n^{L_0} \mapsto \langle X_n, L_n\rangle$	**bind** $X_n^{L_0}$ **to** $\langle X_n, L_n\rangle$
(β-0)	`extends(this,` X_1`)` `@` $\langle X_0, L_0\rangle$	**verify** $\langle X_0, L_0\rangle$
(α-1)	$X_1^{L_0} \mapsto \langle X_1, L_1\rangle$	**bind** $X_1^{L_0}$ **to** $\langle X_1, L_1\rangle$
(β-1)	`extends(this,` X_2`)` `@` $\langle X_1, L_1\rangle$	**verify** $\langle X_1, L_1\rangle$
(α-2)	$X_2^{L_1} \mapsto \langle X_2, L_2\rangle$	**bind** $X_2^{L_1}$ **to** $\langle X_2, L_2\rangle$
(β-2)	`extends(this,` X_3`)` `@` $\langle X_2, L_2\rangle$	**verify** $\langle X_2, L_2\rangle$
(α-3)	$X_3^{L_2} \mapsto \langle X_3, L_3\rangle$	**bind** $X_3^{L_2}$ **to** $\langle X_3, L_3\rangle$
$\vdots$	$\vdots$	$\vdots$
(β-($n-1$))	`extends(this,` X_n`)` `@` $\langle X_{n-1}, L_{n-1}\rangle$	**verify** $\langle X_{n-1}, L_{n-1}\rangle$
(α-n)	$X_n^{L_{n-1}} \mapsto \langle X_n, L_n\rangle$	**bind** $X_n^{L_{n-1}}$ **to** $\langle X_n, L_n\rangle$

Figure 5.10: Leaves of the Proof Tree for the Obligation `subclass(`X_0`,` X_n`)` `@` $\langle X_0, L_0\rangle$

2. **Monotonicity:** The initial theory, commitments and obligations form a monotonic, Horn clause theory. Once an obligation is satisfied, it will not be contradicted by subsequently asserted commitments.

5.3.3 Completion

Completion has to be established on an obligation-by-obligation basis. Continuing with the running example in Section 5.2.7, let us consider an obligation `subclass(`X_0`,` X_n`)` `@` $\langle X_0, L_0\rangle$ that is attached to the primitive "**resolve** X_n::$M(S)$ **in** $\langle X_0, L_0\rangle$". The goal is to show that, if the predicate `subclass(`X_0`,` X_n`)` `@` $\langle X_0, L_0\rangle$ eventually becomes provable, then it is necessarily provable before the primitive "**resolve** X_k::$M(S)$ **in** $\langle X_0, L_0\rangle$" is executed.

Suppose that the obligation `subclass(`X_0`,` X_n`)` `@` $\langle X_0, L_0\rangle$ becomes provable at a certain point. Generalizing the proof tree found in Figure 5.9, the proof tree of the subclassing obligation contains the innermost subgoals shown in Figure 5.10. The subgoals are labeled as (α-i), (β-i) and (γ). The goal is to prove that the primitives that assert commitments satisfying these subgoals have all been executed prior to the execution of "**resolve** X_n::$M(S)$ **in** $\langle X_0, L_0\rangle$". As already explained in Section

5.2.7, the Proper Resolution Property guarantees that supporting commitment (γ) is already in place. Induction is applied to show that commitments (α-i) and (β-i) are already asserted when the obligation is checked.

Basis: Commitment (α-0) and (β-0) are already asserted because,

$$\begin{aligned} \textbf{verify } \langle X_0, L_0 \rangle &< \textbf{bind } X_0^{L_0} \textbf{ to } \langle X_0, L_0 \rangle && \text{(Natural Progression)} \\ &< \textbf{resolve } X_k{::}M(S) \textbf{ in } \langle X_0, L_0 \rangle && \text{(Natural Progression)} \end{aligned}$$

Induction Step: Assume that commitments (α-i) and (β-i) are already in place, for $0 \leq i < k$, where $k > 0$. The presence of these commitments enable the query `subtypedependent(`$X_k^{L_{k-1}}$`) @` $\langle X_0, L_0 \rangle$ to be satisfiable. The following can then be deduced.

$$\begin{aligned} \textbf{verify } \langle X_k, L_k \rangle &< \textbf{bind } X_k^{L_k} \textbf{ to } \langle X_k, L_k \rangle && \text{(Natural Progression)} \\ &< \textbf{bind } X_k^{L_{k-1}} \textbf{ to } \langle X_k, L_k \rangle && \text{(Proper Resolution)} \\ &< \textbf{endorse } \langle X_0, L_0 \rangle && \text{(Subtype Dependency)} \\ &< \textbf{resolve } X_n{::}M(S) \textbf{ in } \langle X_0, L_0 \rangle && \text{(Natural Progression)} \end{aligned}$$

Since the contributors "**bind** $X_k^{L_{k-1}}$ **to** $\langle X_k, L_k \rangle$" and "**verify** $\langle X_k, L_k \rangle$" for respectively (α-k) and (β-k) are already executed, the commitments are present when the obligation is checked.

This concludes the proof of Completion for one class of obligations. Completion can be established similarly for the rest of the obligations.

5.4 Summary

It has been shown in this chapter that the Java instantiation of the Proof Linking Architecture can be extended in a systematic manner to provide support for multiple classloaders. The proposed extension preserves the three correctness conditions. Particular attention is paid to make the extension applicable in the setting of distributed verification.

Successful extension of the Proof Linking Architecture to support Java delegation style classloader gives us strong evidence that the architecture can indeed be realized in a production mobile code system. The following chapter describes an implementation of the Proof Linking Architecture in an open source JVM.

Chapter 6

The Aegis VM

6.1 Introduction

This chapter[1] reports an implementation effort that serves to establish thesis statement **TS2**, namely, the implementation feasibility of the Proof Linking architecture in a production mobile code system for supporting stand-alone verification modules and augmented type systems.

To give concrete evidence to support the thesis statement **TS2**, the Proof Linking architecture has been fully implemented in an open source JVM, *the Aegis VM* [68][2]. The project is an on-going work. 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 support multithreading yet. With the features already implemented, the VM provides a realistic Java platform for implementation and evaluation of Proof Linking. Currently, the VM runs on the GNU/Linux (x86) platform, but efforts in porting the Aegis VM to platforms such as

[1]Results in this chapter and the next first appeared in [70].

[2]The features described in this chapter is publicly available in the project CVS repository, and will be integrated into the next release.

Darwin, Solaris and Windows are planned[3].

At the core of this exercise is the design and implementation of a generic mechanism for recording and discharging proof obligations, and for defining arbitrary initial theories that provide the vocabularies and semantics in terms of which proof obligations and commitments can be formulated. This *generic proof linking mechanism* will be described in Section 6.2. The generic proof linking mechanism is then employed to support two further features. Firstly, standard Java bytecode verification is implemented in this framework as a *stand-alone verification module*, in accordance with the blueprint prescribed in Chapters 4 and 5. Secondly, a generic *verifier plug-in mechanism*, Pluggable Verification Modules (PVMs), is implemented. This mechanism allows one to introduce well-mannered static program analyses into the dynamic linking process of the Aegis VM. Details of the stand-alone Java bytecode verifier and the PVM facility can be found in Section 6.3.

The design goals outlined in Section 3.4.2 were observed in the development of the generic proof linking mechanism and the PVM facility. They are restated here for reference.

1. **Generality.** The generic proof linking mechanism and the PVM facility 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.

At the system level, two potential problems are also noted in Section 3.4.2.

[3] A contributer has submitted a patch that ports the Aegis VM to the Cygwin platform. Cygwin is an open source POSIX layer on top of the Win32 platform.

1. **Data structures.** It is anticipated that non-trivial data structures will be needed to support the incremental proof linking process. Introduction of complex data structures into an already complex linking process within the JVM is questionable. Consequently, the data structure required to implement proof linking should have manageable complexity.

2. **Linking strategies.** The correctness conditions imposed on linking strategies affect the temporal ordering of linking activities. An easy-to-validate linking strategy for Java bytecode verification may perhaps not be the laziest one allowed by the JVM specification. The degree to which laziness is curtailed may proportionately affect the efficiency of the overall linking process. An implementation of Proof Linking should try to maximize the degree of laziness while preserving correctness.

In the following discussion, we will see how the Aegis VM was designed and implemented to meet the above design goals while avoiding the potential problems.

6.2 Incremental Proof Linking

This section describes the generic proof linking mechanism implemented in the Aegis VM. The key features of the design are summarized before a detail treatment is given.

6.2.1 Key Features

Obligation Discharging as Native Function Dispatching

A generic proof linking mechanism should support the attachment of arbitrary obligations to the set of linking primitives specified in Chapter 5. Each verification domain requires a different initial theory. A generic mechanism must be in place to capture the logic of evaluating obligations for a wide spectrum of verification domains. Also, since linking events occur frequently in a Java platform, the overhead of obligation evaluation must remain an acceptable percentage of the overall run time.

A declarative representation, such as the deductive database notation that has been used throughout this work, will satisfy the generality requirement, but will fail miserably in terms of efficiency. For this reason, a procedural approach is adopted for specifying obligation semantics. Specifically, an initial theory, including its vocabulary and axioms, is specified in terms of a dynamically loadable shared library of predicates implemented as C functions. A security engineer implements a library of native predicate functions following a specific convention, and then configures the Aegis VM to load the library into its address space at the time of start up. Whenever an obligation is to be discharged, the corresponding native function will be dispatched to evaluate the validity of the obligation. An Application Programming Interface (API) is defined to allow these native predicate functions to query the internal states of the Aegis VM (e.g., interrogating the type interface of a specific class) or to retrieve the commitments related to a class. Details of this design is given in Section 6.2.2.

Flexible Obligation Encoding

The end result of a modular verification session is either a simple failure, or a success together with a set of obligation attachments. Obligations are not discharged until the target linking primitives to which the they attach are executed. As such, they must be represented and stored explicitly in the Aegis VM. This is achieved by an efficient but expressive *obligation encoding scheme*. An encoded obligation is composed of an identifier that denotes a native function implementing its semantics, together with a list of actual arguments. Also designed is a rich *obligation argument encoding scheme* that lets a modular verifier name the components of a class interface or other global constants as obligation arguments. This includes, with respect to the class being verified, the immediate superclass and superinterfaces, declared methods, declared fields, constant pool entries, and so on. Prior to the discharging of an obligation, the Aegis VM will resolve these encoded arguments into the VM data structures to which they refer. The resolved arguments are then passed into the native predicate function implementing the obligation.

One particular kind of symbol that may be named as obligation arguments comprises the class symbols referenced in the body of a bytecode method. These appear in a classfile as substrings of an UTF8 entry in the constant pool. The availability of the referents for these auxiliary symbols is not guaranteed by the normal linking process of the Aegis VM. These are exactly the class symbols *relevant* to the verification of the bytecode method (Section 4.2.2). Relevant symbols are explicitly identified by a modular verifier. The Aegis VM implements the Referential Dependency Property (Section 4.2.2) by performing extra classloading in order to make sure that all the identified relevant symbols are available before they are needed for obligation discharging.

Details of obligation encoding can be found in Section 6.2.3.

Class-Centric Obligation Attachment

Once an obligation is formulated, it is attached to a linking primitive. One challenge is to make sure that obligation attachment works properly with the class unloading semantics of Java. According to the JVM Specification, a class is unloaded when its defining classloader becomes unreachable [130, Section 2.17.8]. At the point when a loaded class $\langle X, J \rangle$ is unloaded, all commitments and obligations tagged with $\langle X, J \rangle$ should be retracted from the commitment database. If commitments and obligations tagged with a loaded class are stored along with the class, then unloading a class automatically removes all the associated commitments and obligations. Consequently, obligation attachment points are built into the class structure to facilitate the tracking and discharging of proof obligations. Details can be found in Section 6.2.4.

Agnostic Representation for Commitments

To maximize the opportunity for optimization, the Aegis VM does not assume a particular representation for commitments. A **verify** primitive is free to produce any native representation of commitments. It is only assumed that such a commitment data structure is stored along with a loaded class. The obligation library API provides facilities for predicate functions to examine the content of commitment data structures. Details of commitment generation are described in Section 6.3, where modular verification is discussed.

6.2.2 Pluggable Obligation Libraries

POL Life Cycle

Programmers define a custom verification domain by developing a corresponding *pluggable obligation library (POL)*. Each POL is a dynamically loadable shared library on GNU/Linux, and supplies native functions that will be used for evaluating obligations in the initial theory of the verification domain. The Aegis VM has a command line option that allows users to specify the path of a POL[4]. Users may specify multiple POLs in the command line, thereby equipping the VM with vocabulary sets for multiple verification domains. When the Aegis VM starts up, all the POLs specified in the command line are loaded. Each library exports an *initialization function*, which is invoked after the POL is loaded. The initialization function initializes the POL by, for example, creating global data structures that will be used for obligation evaluation purposes. The VM then loads a number of classes on behalf of the POLs during its bootstrapping process, so that the said classes can be used in the course of obligation evaluation. After that point, the native functions in the library are made available to the generic proof linking mechanism for obligation discharging. Before the Aegis VM shuts down, each *clean up function* exported by the loaded POLs is invoked to clean up the libraries before they are unloaded.

POL API

Each POL must export a global variable named `pol_profile` with the following type:

```
struct pol_profile_t {
```

[4] A planned feature is to allow users to specify the same information using a configuration file.

```
  const char *id;
  pol_init_t init;
  pol_finish_t finish;
  uint16_t npreds;
  const pol_predicate_t *predicates;
  uint16_t nclasses;
  const char * const *global_classes;
  uint16_t nconstants;
  const void * const *global_constants;
};
```

This global variable describes the *profile* of the POL. The POL programmer must initialize the fields as follows:

- `id`: A C-string identifying the verification domain that this obligation library is intended to model. As we shall see in the following, a *pluggable verification module* relies on this identifier to match up the obligations it generates with their evaluation semantics.

- `init`: A function pointer referring to the initialization function of the POL. Specifically, the type of the pointer is given below:

  ```
  typedef int (*pol_init_t)(void);
  ```

- `finish`: A function pointer referring to the clean-up function of the POL. Specifically, the type of the pointer is given below:

  ```
  typedef int (*pol_finish_t)(void);
  ```

- `npreds`: The number of predicate functions exported by this POL, in a 16-bit unsigned integer.

- `predicates`: An array of `npreds` function pointers, each referring to the implementation of a predicate exported by the POL. More details concerning the type signature of predicates will be given below.

- `nclasses`: The size of the `global_classes` array, in a 16-bit unsigned integer.

- `global_classes`: An array of `nclasses` C-strings, each specifying the name of a class that the VM should preload during its bootstrapping process (using the bootstrapping classloader). The classes named in this array are used for evaluating obligations in the verification domain this POL models. As such, they are accessible from a POL through a standard API, and can also be named by a modular verifier as obligation arguments. For example, the class *'java.lang.Throwable'* is a global class of the standard Java typechecking POL, because it is instrumental in the evaluation of the predicate `throwable` (Figure 5.3).

- `nconstants`: The size of the `global_constants` array, in a 16-bit unsigned integer.

- `global_constants`: An array of immutable, native data structure representing the domain constants in the verification domain. Predicate functions may access them through standard API calls. A modular verifier may also name them as obligation arguments.

The main job of a POL programmer is to supply native functions that implement the semantics of obligation predicates. Each predicate listed in the `predicates` array must have the following type signature[5].

```
typedef bool (*pol_predicate_t)(POLEnv env,
                                uint16_t nargs,
                                const void *args[const]);
```

- `env`: Contextual information supplied by the Aegis VM. It is needed for retrieving commitments and other data structures that might be needed for proper evaluation of a predicate.

[5]The formal parameter `args` is typed in a syntax introduced in the C99 standard [102].

- `nargs`: The number of actual arguments being passed, in a 16-bit unsigned integer.
- `args`: An array of `nargs` generic C pointers, each pointing to an actual argument of the obligation.

The 16-bit index of a predicate function in the array `predicates` becomes the *predicate identifier* for that predicate. This numeric index is used by a modular verifier to identify a predicate while formulating obligations. Before discharging an obligation, the Aegis VM will look up all the actual arguments embedded in an obligation encoding. The actual arguments will be placed in an array, which will subsequently be passed into a corresponding predicate function via parameter `args`, along with the array size in parameter `nargs`. As modular verifiers are trusted components, a predicate function is not obligated to verify the type and number of the arguments. The nargs argument is supplied for implementing variable-size argument list and for debugging. As we shall see in the following section, arguments could be internal data structures of the Aegis VM, such as classes, methods, fields, and constant pool entries, or global classes and constants supplied through the POL profile, etc.

To facilitate the evaluation of obligations, the Aegis VM offers a *pluggable obligation library API* to allow obligation functions to examine the run-time state of the VM (e.g., relationship between classes), or, more specifically, retrieve the commitments associated with a class for the current verification domain. A brief summary of the API functions that an obligation function may access is given below. A complete list can be found in Appendix B.

1. *Package interface interrogation*: Given a package, examine its package name, classloader, etc.
2. *Class interface interrogation*: Given a class, examine its access control flags, package, classloader, class name, superclass, superinterfaces, fields, methods, constant pool entries, etc.

header		**argument list**			
pred	*nargs*	*arg-1*	*arg-2*	...	*arg-k*
16 bits	16 bits	32 bits	32 bits	...	32 bits

Figure 6.1: Encoding of a Proof Obligation

argument	
type	*index*
16 bits	16 bits

Figure 6.2: Encoding of an Obligation Argument

3. *Field interface interrogation*: Given a field, examine its access control flags, declaring class, field name, type signature, etc.

4. *Method interface interrogation*: Given a method, examine its access control flags, declaring class, method name, type signature, exception class names, etc.

5. *Subtyping relationship*: Subclassing, subinterfacing, subtyping, etc.

6. *Contextual information*: Retrieve commitments of a class, retrieve global classes or constants specified in the POL profile, etc.

With these helper functions, a POL predicate function determines whether an obligation is satisfied, returning a boolean value to indicate so.

6.2.3 Obligation Encoding and Relevant Symbols

Proof obligations are encoded for efficient storage and discharging. Every obligation is encoded with respect to a *base POL* and a *target class*. The base POL defines the verification domain to which the obligation belongs. It is the POL against which predicates, global classes and global constants are looked up. The target class is the one with which the obligation is tagged in Chapter 5. It provides a reference point from which class components can be addressed. Figure 6.1 depicts the encoding scheme for an obligation in the Aegis VM.

type	**index**
`PRE_ARG_Constant`	index of a constant pool entry in the target class
`PRE_ARG_Symbol`	index of a *relevant* symbol for the target class
`PRE_ARG_Relative`	a tag specifying either the target class (0) or its superclass (1)
`PRE_ARG_Interface`	index of an immediate superinterface for the target class
`PRE_ARG_Field`	index of a declared field in the target class
`PRE_ARG_Method`	index of a declared method in the target class
`PRE_ARG_GlobalClass`	index of a global class named in the `global_classes` array of the base POL's profile.
`PRE_ARG_GlobalConstant`	index of a global constant in the `global_constants` array of the base POL's profile.

Figure 6.3: Argument Type

- An encoded obligation begins with a *pred* field, which is the 16-bit unsigned integer identifying a POL predicate function. Recall that a predicate identifier is essentially the index of a native function in the `predicates` array of a POL's `pol_profile`.

- Next comes a *nargs* field, which is a 16-bit unsigned integer specifying the number of arguments for this obligation.

- Exactly *nargs* 32-bit arguments follow. Each argument *arg-i* is encoded as shown in Figure 6.2, as two 16-bit subfields. The encoding scheme for arguments is discussed below.

Each encoded argument is composed of two 16-bit unsigned integer fields (Figure 6.2). The *type* field is a tag specifying the type of the argument. The *index* field supplies additional information to identify the argument. Figure 6.2 summarizes the use of the *type*-*index* pair.

- Argument types `PRE_ARG_GlobalClass` and `PRE_ARG_GlobalConstant` allow global classes specified by the base POL and the global constants it exports to be named as obligation arguments.

- Argument types `PRE_ARG_Relative`, `PRE_ARG_Interface`, `PRE_ARG_Field` and `PRE_ARG_Method` provide a means to encode supertypes and members of the target class.

- The argument type `PRE_ARG_Symbol` is for encoding relevant symbols. When a modular verifier scans the bodies of bytecode methods, it will identify all the class symbols relevant to the verification of the methods. A symbol table of such relevant classes are returned as part of the result of a successful verification. An argument with type `PRE_ARG_Symbol` identifies a member of this *relevant symbol table*.

- The argument type `PRE_ARG_Constant` is for encoding referents of constant pool entries. Two types of constant pool entries can be encoded. Firstly, constant pool entries corresponding to `int`, `long`, `float`, `double`, and UTF8 literals can be encoded this way to represent constant arguments of the respective types. Secondly, the resolved target of a constant pool entry corresponding to an import reference (i.e., a class, field, method, or interface method reference) can also be named as an obligation argument in any of the following cases:

  1. The target primitive is "**resolve** Y **in** $\langle X, L\rangle$", and the constant pool entry is the class reference Y.
  2. The target primitive is "**resolve** $Y{::}M(S)$ **in** $\langle X, L\rangle$", and the constant pool entry is either the class reference Y or the member reference $Y{::}M(S)$.

  where $\langle X, L\rangle$ denotes the target class.

To make the above discussion more concrete, consider this example. Suppose a modular verifier attempts to formulate an obligation that will be evaluated by the predicate function at index 2 of the base POL's `predicates` array. Suppose further that 3 arguments are to be named in this obligation. The first is the 6th method

Linking Primitive	Corresponding VM Data Structure
endorse *class*	*class*
endorse *field*	*field*
endorse *member*	*member*
resolve *constant-pool-entry* **in** *class*	*constant-pool-entry*

Figure 6.4: Linking Primitives and Their Corresponding VM Data Structures

declared by the target class. The second is the 4th member of the relevant symbol table. The third is the 2nd global constant exported by the base POL. The obligation will be encoded as the following sequence of 16-bit quantities:

pred	*nargs*	*type-1*	*index-1*	*type-2*	*index-2*	*type-3*	*index-3*
2	3	`PRE_ARG_Method`	6	`PRE_ARG_Symbol`	4	`PRE_ARG_GlobalConstant`	2

6.2.4 Obligation Attachment Points

Encoded obligations are attached to a well-defined set of linking primitives. Each linking primitive operates on a specific data structure of the Aegis VM (Figure 6.4), making the data structure a natural attachment point for obligations. Specifically, obligations attached to a linking primitive is stored in the corresponding VM data structure. When the VM attempts to carry out a linking primitive, it will first discharge the obligations stored in the VM data structure. To reduce space consumption, obligations belonging to the same verification domain are stored together at each attachment point. This eliminates the need for each obligation to carry an extra field that identifies the verification domain to which the obligation belongs.

Notice that obligations attached in this way are removed from the VM along with the associated class data structures when the latter is unloaded. The VM memory management mechanism is thus conveniently exploited to manage the life time of proof obligations.

6.2.5 Obligation Discharging Sequence

Obligations are discharged according to the linking strategy presented in the last chapter. When a class is to be defined, its classfile representation (as obtained from a

load primitive) is parsed into an abstract syntax tree (AST). Subtyping information provided by the AST is used to initiate the loading of all supertypes, thus implementing the Subtype Dependency Property. The AST is then examined by all modular verifiers in turn. This corresponds to the execution of the corresponding **verify** primitive. Successful completion of each modular verifier results in a verification interface consisting of commitments, obligation attachments and a relevant symbol table. A permanent class data structure is finally created in the namespace corresponding to the defining classloader of the class (**bind** X^L **to** $\langle X, L\rangle$), and the AST is discarded. When the loaded class is subsequently *prepared* [130, Section 5.4.2], all the classes named in the relevant symbol table are explicitly loaded and defined, thus implementing the Referential Dependency Property[6]. The **endorse** primitives are then executed, beginning with class endorsement, then field and method endorsement. After this point, constant pool entries may be resolved as a result of bytecode execution (**resolve** primitives). To resolve a constant pool entry, the Aegis VM acquires the target pointer, makes it available for obligation argument resolution, and then discharge obligations attached to the **resolve** primitive before marking the constant pool entry as having been successfully resolved. This sequencing of events implements the Proper Resolution Property.

To discharge an encoded obligation, the following steps are followed.

1. The *pred* field of the obligation header (Figure 6.1) is used as an index to look up the corresponding predicate function in the current POL. This operation takes constant time.

2. A temporary array of *nargs* pointers is created for holding arguments.

3. Each obligation argument is resolved into a corresponding pointer to a VM data structure. The arguments are then placed into the temporary argument array. As argument resolution amounts to a simple look up operation of VM data

[6]To be precise, the variant of the Referential Dependency Property as described in footnote 3 on page 115 is implemented. As noted before, this does not affect the correctness of proof linking.

structures, no classloading is involved. Resolution of an argument can thus be carried out in constant time.

4. The predicate function is invoked with the argument array as input.
5. The result of evaluation is returned by the predicate function as a boolean value.

Since predicate lookup and argument resolution are both constant time operations, the overhead involved in the dispatching of an obligation is proportional to the number of obligation arguments. Such a modest overhead is quite reasonable.

6.2.6 Correctness Issues

Since the Aegis VM implements the linking strategy in Chapter 5, most of the correctness arguments transfer to the Aegis VM. As prescribed in the previous chapter, a modular verifier only attaches obligations to the **endorse** and **resolve** primitives of the target class. The Safety condition is therefore guaranteed. Completion is guaranteed by two features. Firstly, the POL API (Section 6.2.2) is carefully constrained to expose only those commitment information already proven to be available at the time of obligation discharging. Secondly, the argument encoding scheme (Section 6.2.3) and the linking strategy are designed in such a way that properly named obligation arguments are always resolvable. In particular, relevant classes are explicitly loaded before the execution of any primitive that may bear obligations containing their references. Similarly, the rules governing when constant pool entries may be named as obligation arguments (Section 6.2.3) guarantee that targets of import references are made available to the obligation argument resolution mechanism before the corresponding obligations are discharged. Such design decisions ensure the Completion condition is satisfied.

Monotonicity is not guaranteed by the Aegis VM. It is the responsibility of the POL programmer to make sure that the initial theory implemented in a POL satisfies the Monotonicity condition.

6.2.7 Comparison with the Sun Linking Strategy

As opposed to the Sun JVM implementation, which postpones bytecode verification until a class is prepared, the implementation strategy above performs full verification at the time of class definition. This is necessary for ensuring that commitments and relevant symbol tables are properly constructed before they are needed for obligation discharging. The Sun JVM performs one pass of verification at class definition time, postponing the second and third passes until class preparation. Despite this difference, the classloading order of Aegis VM is exactly the same as that of Sun. The reason is that, although verification is scheduled earlier in the Aegis VM than in the Sun implementation, *the loading schedule of relevant classes remains the same* — they are loaded when a class is prepared. Consequently, although Aegis VM performs eager verification, the linking strategy is nevertheless identical to that of Sun.

The linking strategy of Chapter 5 permits more laziness in classloading than is implemented in the Aegis VM. Specifically, the linking strategy of the previous chapter allows method endorsement to be postponed until a method is resolved for the first time, and assumes that relevant symbols are identified separately for each method. An implementation could potentially exploit these properties to delay the endorsement of a method and the loading of its relevant classes. If such a method is not actually invoked, then the loading of relevant classes can be completely avoided. Comparing to the unconditional loading of all relevant symbols performed by the Aegis VM at class preparation time, such an arrangement could potentially increase the performance of a JVM in a significant manner [222, Chapter 6]. This path is not followed so as to reduce memory consumption and maintain higher fidelity to the Sun linking strategy.

This concludes the discussion of the generic proof linking mechanism implemented in the Aegis VM. In the next section, we turn to the counterpart of incremental proof linking, namely, modular verification.

6.3 Modular Verification

This section describes how the generic proof linking mechanism is exploited to support stand-alone verification modules for Java bytecode typechecking, and presents a plug-in mechanism that allows users to augment the protection mechanisms of the Aegis VM by introducing alternative static analyses into the dynamic linking process. Again, key features of the design are summarized before a detail treatment is given.

6.3.1 Key Features

Stand-alone Java Bytecode Verifier

A stand-alone Java bytecode verifier is implemented in an open source C library called *Prelude* [68], which is distributed separately of the Aegis VM. The Prelude library provides facilities for parsing and typechecking Java classfiles. It also provides generic facilities for formulating verification interfaces composed of obligation attachments, commitments and relevant symbol tables. Because intermodular dependencies are all captured in proof obligations and commitments, bytecode verification can be conducted without consulting external classes. See Section 6.3.2 for details concerning the Prelude library. The Aegis VM employs the Prelude library to perform bytecode verification. Proof obligations and commitments generated by the Prelude library interoperates with the generic proof linking mechanism in the Aegis VM.

Pluggable Verification Modules

A plug-in mechanism is designed so that programmers may introduce alternative static analyses into the dynamic linking process of the Aegis VM. Modular static analyses are implemented as dynamically loadable shared libraries, called *pluggable verification modules (PVMs)*, according to a set of coding convention prescribed by a PVM API. The Aegis VM can be configured to load these libraries into its address space at the time of start up. Whenever a class is to be defined, its classfile representation is passed to each loaded PVM in turn for verification. Class definition is authorized only when all the PVMs endorse the safety of the classfile.

The PVM mechanism depends on the Prelude library in two ways. Firstly, the result of successful verification is a set of obligation attachments, a commitment data structure and a relevant symbol table. The Prelude library provides constructors for formulating these objects. Secondly, each PVM will receive as input arguments an abstract syntax tree (AST) and the dataflow analysis results produced by the Prelude classfile parser and bytecode typechecker respectively. This information can be exploited by a PVM to facilitate its own program analysis.

Details of the PVM mechanism can be found in Section 6.3.3

6.3.2 Stand-alone Java Bytecode Verifier

The Prelude library features facilities for parsing and validating the structural well-formedness of Java classfiles, performing dataflow analysis to assure the type safety of bytecode methods, and formulating verification interfaces in the form of proof obligations, commitments and relevant symbol tables.

- *Classfile parsing*: A family of C functions are in place for generating an AST from a Java classfile. The classfile representation is verified for well-formedness. Standard classfile attributes are recognized and parsed. Unrecognized attributes are preserved verbatim. Accessor functions are in place to facilitate traversal of the AST.

- *Bytecode typechecking*: A dataflow analyzer is implemented to perform standard typechecking on bytecode methods. The analyzer also recovers the intraprocedural control flow graph, which is originally implicit due to the presence of the notorious subroutine construct in the bytecode language [191, 151]. For each program point, a *type state* is also computed to report (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 that leads to the program point. Facilities are also in place to formulate the proof obligations specified in Chapters 4 and 5.

- *Verification interface formulation*: Data structures are implemented to support construction of verification interfaces composed of obligation attachments, commitment data structures and relevant symbol tables. These data structures are employed by the Prelude bytecode typechecker for recording verification interfaces in the standard Java bytecode verification domain. Yet, as the data structures are general enough, they are reused by the PVM mechanism for recording results of modular verification. Specifically, proof obligations are encoded in the scheme described in Section 6.2.3, while the representation of commitments are left to be domain specific.

Specific implementation issues concerning the bytecode typechecker are discussed below.

Meet Computation

Java typechecking involves dataflow analysis. In the Sun implementation of dataflow analysis, the meet of two classes Z_1 and Z_2 is their most specific common superclass [130, Section 4.9.2]. To compute this superclass at verification time, the Sun JVM immediately loads all superclasses of Z_1 and Z_2. The Prelude bytecode typechecker eschews recursive loading by representing the meet symbolically as $Z_1 \sqcap Z_2$, the semantics of which is that the reference could either be Z_1 or Z_2, and thus any operation on such a reference should be supported by the type interfaces of both Z_1 and Z_2. Obligations are then formulated in terms of these *meet expressions*[7]. Specifically, when the *composite obligation* `P(`$Z_1 \sqcap Z_2$`)` is to be imposed, the typechecker will generate both `P(`Z_1`)` and `P(`Z_2`)`. For example, if `subclass(`Y`,` $Z_1 \sqcap Z_2$`)` is found to be a safety precondition, then the two obligations `subclass(`Y`,` Z_1`)` and `subclass(`Y`,` Z_2`)` are

[7]Such a symbolic representation does not affect the termination of the data flow analysis because the number of class symbols that may appear in a method is finite, and the underlying lattice is consequently bounded [111].

generated. First reported in [71], this technique of formulating meet expressions and composite obligations were independently proposed by Coglio and Goldberg [41, 42], who later employed it to fix a bug found in the standard Java bytecode verification algorithm [130, Section 4.9.2].

Handling Arrays

Java arrays are classes, and there are special type rules for handling them. However, these rules are ultimately formulated in terms of type rules of ordinary classes. For example, an array of X is assignment compatible to `java.lang.Object`, `java.lang.Cloneable`, and an array of Y for which X is assignment compatible to Y. As a result, the Prelude bytecode typechecker can always translate an obligation involving array types into one or more (conjunctive) obligations that are free of array type.

Optimization

Notice that the linking strategy of the Aegis VM is more or less fixed, except for the ordering imposed by the Referential Dependency Property, which is defined in terms of a relevant symbol table. Specifically, if a class is not statically identified to be relevant to a method, less ordering, and thus more laziness, results. To optimize the linking process, the modular bytecode verifier may choose not to identify a class as relevant if it is redundant to do so. For example, if the modular verifier attempts to attach `subclass(`Y`,`Z`)` to "**endorse** X::$M(S)$", normally, Y will need to be statically identified as being relevant to X::$M(S)$. But in the case when $Y = Z$, one knows that `subclass(`Y`,`Z`)` is trivially provable. So, the obligation need not be generated in this case, and thus it is not necessary to identify Y as being relevant to X::$M(S)$. Similar optimizations have been adopted for obligations involving other special cases of type rules.

Simplification

Two simplifications have been employed to reduce the complexity of the implementation. Firstly, commitments are not explicitly generated for Java typechecking, for they merely carry type information that is already embedded in a Java classfile. Instead, the obligation predicate functions in the Java typechecking POL invokes the type interface interrogation functions in the POL API to obtain the same information. This reduces memory consumption, and improves obligation discharging efficiency. Although the commitment facility is not explicitly utilized for Java bytecode verification, it is crucial for other verification domains, as we shall see in Chapter 7.

Secondly, two of the obligations in Section 4.2.4, namely `subclassable` and `implementable`, are not generated. The reason is that they are assumed for all classfiles, and thus their explicit formulation as proof obligations is omitted to improve memory efficiency and implementation economy. Instead, equivalent checks are hard-coded into the classloading logic of the Aegis VM.

Obligation Library

A POL is implemented for evaluating proof obligations specific to the Java bytecode typechecking domain. The native predicate functions implement the core initial theory in Figure 5.3. Most of the predicate functions can be implemented trivially by the interrogation functions in the POL API. The obligation translation rules are implicitly embedded into the obligation argument resolution mechanism, while the commitment translation rules are implemented within the interrogation functions exported by the POL API. Also, the bytecode typechecker generates obligations that may name Java built-in classes as arguments. This small set of Java built-in classes are identified as global classes of the POL.

6.3.3 Pluggable Verification Modules

This section describes the design of the PVM facility as implemented in the Aegis VM.

PVM Life Cycle

PVMs are dynamically loadable shared libraries on the GNU/Linux platform. Programmers may implement a modular verifier as a PVM, and subsequently use it to augment the Aegis VM through the PVM plug-in mechanism. The Aegis VM has a command line option that allows users to specify the path of a PVM. Multiple PVMs may be specified in the command line. When the Aegis VM bootstraps, each PVM thus specified will be loaded. Each PVM exports the identifier of a verification domain to which it corresponds. The Aegis VM will search through all the loaded POLs to find a matching one (the `id` field of the POL profile structure). The predicate functions of the matching POL will be used for interpreting the proof obligations generated by the modular verifier of this PVM. The *initialization function* exported by the PVM will then be invoked. From this point on, the verification facilities of the PVM will be called into service whenever a class is to be defined. Specifically, when a class is defined, the classfile parsing facility of the Prelude library is employed to construct an abstract syntax tree (AST) for the classfile representation. The Prelude bytecode typechecker is then applied to run dataflow analysis on the bytecode methods of the AST. The AST and the dataflow analysis results (i.e., subroutine control flow graph and type states) are then passed into each loaded PVM for examination. In this way, the PVM does not need to analyze the target classfile from scratch. The obligation attachments, commitment data structures and the relevant symbol tables generated along the way are all collected in Prelude verification interface data structures, which are in turn processed uniformly by the generic proof linking mechanism. Before the Aegis VM shuts down, the *clean-up function* exported by each of the PVMs will be invoked in turn, and then the libraries are unloaded.

PVM API

A PVM must export a global variable named `pvm_profile` as its *profile*. The type of the variable is given below.

```
struct pvm_profile_t {
```

```
  const char *pol;
  pvm_init_t init;
  pvm_finish_t finish;
  pvm_verify_t verify;
};
```

The profile variable must be initialized as follows.

- `pol`: A C-string identifying the POL that should be used for interpreting the proof obligations generated by this PVM.
- `init`: A function pointer referring to the initialization function of this PVM. Specifically, the type of the pointer is given below:

  ```
  typedef int (*pvm_init_t)(void);
  ```

- `finish`: A function pointer referring to the clean-up function of this PVM. Specifically, the type of the pointer is given below:

  ```
  typedef int (*pvm_finish_t)(void);
  ```

- `verify`: A function pointer referring to the modular verifier exported by this PVM. More details will follow.

The core of a PVM is the function referenced by the `verify` field.

```
typedef const PREVI *(*pvm_verify_t)(JArena *arena,
                                     const PREClassFile *classfile,
                                     const PREAnalysis **analyses);
```

- `arena`: The memory arena [92] from which memory is allocated to build the resulting verification interface.
- `classfile`: The abstract syntax tree of the classfile being examined.

- `analyses`: The results of typechecking dataflow analyses for methods in `classfile`.

If verification is successful, the verifier function returns a *verification interface* (`PREVI`), which is composed of three elements:

1. a table of class symbols relevant to the verification of methods in this class,
2. a set of proof obligation attachments generated by the verifier (obligations are encoded in the scheme specified in Section 6.2.3),
3. a domain-specific commitment data structure.

The header file that defines the PVM API can be found in Appendix C.

6.4 Summary

This chapter describes an implementation of the Proof Linking architecture in an open source JVM, the Aegis VM. Modularization of the link-time verification step allows standard bytecode verification to be implemented as a stand-alone bytecode verification module called the Prelude library. Generalization of the incremental proof linking procedure yields a general purpose verifier plug-in mechanism, PVM, that supports the introduction of alternative static program analyses into the dynamic linking process of the VM.

In the next chapter, the utility of the PVM facility is evaluated in an experimental application of the facility to enforce an augmented type system.

Chapter 7

Application: Java Access Control

In this chapter, the generality and utility of the POL and PVM facilities are evaluated in a specific verification domain, namely, that of the augmented type system JAC proposed by Kniesel and Theisen [122]. The chapter begins with an examination of the JAC type system in its original form, which was designed for typing Java source programs. The JAC type system is then recast into a form that types Java bytecode (Section 7.1). We then look at how a link-time typechecker for the bytecode version of JAC can be implemented in the framework of the POL and PVM facilities (Section 7.2). Lastly, examples of Aegis VM in action, enforcing access control through the JAC POL and PVM, are given in Section 7.3. The goal of this discussion is to demonstrate that features designed into the POL and PVM facilities are sufficient for supporting the implementation of link-time typecheckers for augmented type systems.

7.1 A JAC Type System for Java Bytecode

7.1.1 Motivation

JAC (Java with Access Control) was proposed as an augmented type system for controlling the proliferation of side effects due to alias creation in object oriented programs [122]. As briefly discussed in Section 1.1.3, rather than preventing the creation of aliases, as is frequently done in many similar augmented type systems, 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/C++ qualifier `const`, which only protects the state of the object directly accessible from a `const`-qualified reference/pointer, JAC's write protection extends to all objects reachable from a `readonly`-qualified reference in the underlying object graph. This verification domain is chosen as a test case for the PVM facility because of its simplicity and its relevance to access control.

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 all instance variables in `List` are `public`, and as such they can be freely modified by client code.

```
List x = new List(1, new List(2, null));
x.data = 3;       // OK: Writing to the immediate state of x
x.next.data = 4;  // OK: Writing to the transitive state of x
```

However, a `List` variable qualified as `readonly` cannot be used for modifying the transitive state reachable from the variable.

```
readonly List y = x;
y.data = 5;       // Error: Writing to the immediate state of y
y.next.data = 6;  // Error: Writing to the transitive state of y
```

Notice that unqualified reference types can be converted to `readonly` ones, but not vice versa.

```
readonly List u;
u = x;  // OK: Converting from unqualified to readonly
u = y;  // OK: Readonly to readonly
List v;
v = x;  // OK: Unqualified to unqualified
v = y;  // Error: Readonly to unqualified
```

Notice also that references reachable from a `readonly` reference are all `readonly`.

```
v = x.next;  // OK: Transitive state of unqualified is unqualified
v = y.next;  // Error: Transitive state of readonly is readonly
```

The types of class/instance variables, method parameters and return values, can all be qualified as above. The meaning of such qualification is that only reference values with compatible qualification types can be stored in class/instance variables, passed as arguments, or returned as method values.

The original JAC type system is designed for typing Java source programs, and is enforced at compile time. As discussed in Chapter 1, code units that are checked to be type safe in the compilation environment may no longer be type safe when they are linked against the 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 [122] 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 annotation scheme 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 [122] describes an extension of JAC to accommodate the generic type system of GJ [29], a variant of Java that supports genericity. This extension is beyond the scope of this work.

7.1.2 The Type System

The bytecode version of the JAC type system consists of type rules for well-formedness and subtyping, classfile linking, and intraprocedural type consistency, each of which will be discussed in turn.

Type Assertions

The JAC type system has two types[1], namely, `readonly` and $\perp$. Every value in Java is typed as one of the two. The bottom type $\perp$ corresponds to unqualified reference types *and* primitive types (i.e., `int`, `boolean`, etc). The `readonly` type applies to references for which transitive states are protected.

There are two kinds of type assertions, one for fields (i.e., class or instance variables), the other for methods (i.e., instance and static methods, and instance and class initializers). A well-formed type assertion for a field must assign $\perp$ to a field that has a Java primitive type. A well-formed type assertion for a method consists of an assignment of a JAC type to every formal parameter, including `this` in the case of an instance method, and a JAC type to the return value, if there is one. Again, well-formedness further requires that $\perp$ be assigned to formal parameters with Java primitive types. The same constraint applies to method return types.

[1]To avoid confusion, we call the types in the JAC type system JAC types, and the types in the standard Java type system Java types. If the context is clear, especially in this section, the JAC types are simply called types.

Subtyping

The two JAC types are ordered as in the following subtype lattice:

$$\begin{array}{c}\texttt{readonly}\\ | \\ \perp\end{array}$$

The subtyping relationship permits the conversion of $\perp$ to `readonly`. We write $A <: B$ if type A is equivalent to or a subtype of type B. Method subtyping follows the usual contravariant rule: $A \to B <: A' \to B'$ if $A' <: A$ and $B <: B'$.

Typing Classes

Associated with each Java classfile is a *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 assigns a type assertion for every symbol exported by the class, including class and instance variables, and all declared methods and initializers (hereafter, these symbols are collectively called *export symbols*). The import part contains type assertions for symbols in the constant pool that are either a field reference, a method reference, or an interface method reference (hereafter, these three types of constant pool entries are collectively called *import references*). The import part corresponds to a type environment in formal type systems, and as such it entails each of the type assertions in the export part. In essence, the type interface is a composite *type judgment*.

Subclassing is type safe if instance method overriding honors the usual subtyping rule, that is, if method $C.M$ overrides $C'.M$, and the methods are typed by the assertions $C.M : T$ and $C'.M : T'$, then $T <: T'$. A similar requirement applies to subinterfacing. In essence, the type of an overriding method should be no more stringent than the overridden method.

Resolution of a constant pool (interface) method reference with import 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 $C.F$ with import type assertion $C.F : T$ is type safe if the resolved target $C'.F$ is defined in a classfile that exports type assertion $C'.F : T$. Notice that the typing requirement is different in the two cases.

Typing Methods

Each JVM stack frame has a local variable array and an operand stack. The local variable array contains the "registers" of a JVM, while the operand stack is used for evaluating nested expressions. When a new stack frame is created, the local variable array is initialized with the input arguments. Execution proceeds until the method returns, or when an exception escapes the method scope.

The type assertion for a method is valid if every program point in the body of the method can be consistently assigned a *type state*. A type state is an assignment of a JAC type to each location in the local variable array and the operand stack. Every JVM bytecode instruction imposes typing constraints on the type states at the program points before and after the instruction. The typing constraints for bytecode instructions are presented below.

Most of the bytecode instructions consume operands from the operand stack, and deposit results back into it. For example, the *iadd* instruction pops the two integers at the top of the operand stack, and pushes their sum back, as illustrated in below using a convention popularized by the JVM specification [130].

Operand Stack :

Before: $\ldots, i_1, i_2$

After: $\ldots, i_3$

where integer i_3 is the sum of i_1 and i_2. For a simple instruction such as *iadd*, the typing constraint is straightforward: i_1, i_2 and i_3 must all be typed as $\perp$. The type constraints for most of the instructions in the JVM bytecode instruction set are similarly straightforward. The following, however, are the ones that require special consideration.

- *bastore*, *castore*, *sastore*, *iastore*, *fastore*, *lastore*, *dastore*:

  Operand Stack :

  Before: $\ldots, a, i, v$

  After: $\ldots$

  Operation : Store primitive value v into array reference a as the component at index i. The seven instructions are for storing Java primitive types `boolean`/`byte`, `char`, `short`, `int`, `float`, `long`, and `double` respectively.

  Type Constraints :

  – The type of a must not be `readonly` — storing into the immediate state of a `readonly` array reference is not permitted.

- *aastore*:

  Operand Stack :

  Before: $\ldots, a, i, v$

  After: $\ldots$

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

  Type Constraints :

  – The type of a must not be `readonly`.

  – The type of v must not be `readonly`. Otherwise, a `readonly` element v would be stored into an unqualified array a, making it possible for a subsequent *aastore* to illegally remove the readonly qualification.

- *aaload*:

  Operand Stack :

  Before: $\ldots, a, i$

  After: $\ldots, v$

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

Type Constraints :

- The type of v must be identical to that of a.

- *athrow*:

  Operand Stack :
  Before: $\ldots, o$
  After: o

  Operation : The object reference o is thrown as an exception. The operand stack of the stack frame that catches o will be cleared, and then o will be pushed into that operand stack.

  Type Constraints :

  - The type of o must not be `readonly`. Otherwise, o may escape the current static scope, and the type information of o will not be recoverable in the static scope in which it is caught. Consequently, all exceptions are required to be of type $\perp$ for backward compatibility with the standard Java semantics.

- *areturn*:

  Operand Stack :
  Before: $\ldots, o$
  After: [empty]

  Operation : Return object reference o as a value of this method invocation.

  Type Constraints :

  - If the return type of the current method (as prescribed by its export type) is not `readonly`, then o must not have a `readonly` type.

- *getstatic* ⟨fieldref⟩:

  Operand Stack :
  Before: $\ldots$
  After: $\ldots, v$

Operation : Load the value v of class variable $\langle$fieldref$\rangle$. The field reference $\langle$fieldref$\rangle$ is a constant pool entry.

Type Constraints :

- If $\langle$fieldref$\rangle$ is a reference field with a `readonly` import type, then v is `readonly`. Otherwise, v is $\perp$.

- *getfield* $\langle$fieldref$\rangle$:

  Operand Stack :

  Before: $\ldots, o$

  After: $\ldots, v$

  Operation : Load the value v of the instance variable $\langle$fieldref$\rangle$ from object instance o. The field reference $\langle$fieldref$\rangle$ is a constant pool entry.

  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 $\perp$.

- *putstatic* $\langle$fieldref$\rangle$:

  Operand Stack :

  Before: $\ldots, v$

  After: $\ldots$

  Operation : Store the value v into the class variable $\langle$fieldref$\rangle$. The field reference $\langle$fieldref$\rangle$ is a constant pool entry.

  Type Constraints :

  - If the field reference $\langle$fieldref$\rangle$ has an import type $\perp$, then v must not have a `readonly` type.

- *putfield* $\langle$fieldref$\rangle$:

Operand Stack :

Before: $\ldots, o, v$

After: $\ldots$

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.

- *invokespecial* $\langle$methodref$\rangle$, *invokevirtual* $\langle$methodref$\rangle$:

Operand Stack :

Before: $\ldots o, a_1, a_2, \ldots, a_k$

After: $\ldots v$

Operation : Invoke method $\langle$methodref$\rangle$, with arguments $a_1, a_2, \ldots, a_k$, on the object reference o. Any return value v is pushed into the operand stack. The method reference $\langle$methodref$\rangle$ is a constant pool entry. The *invokevirtual* instruction uses dynamic binding to select the actual method being invoked, while the *invokespecial* instruction does not have a dynamic binding semantics, and can be used for invoking special methods such as instance initializers.

Type Constraints :

- Let the type of the i'th actual argument be T, and the import type of the method reference $\langle$methodref$\rangle$ is such that the i'th formal parameter has type T'. Then $T <: T'$.
- Let the type of the the object reference o be T, and the import type of the $\langle$methodref$\rangle$ is such that `this` has type T'. Then $T <: T'$.
- The value v must have a type identical to the return type prescribed by the import type of the method reference $\langle$methodref$\rangle$.

- *invokeinterface* $\langle$interface-methodref$\rangle$:

  Operand Stack :

  Before: $\ldots o$, a_1, a_2, $\ldots$, a_k

  After: $\ldots v$

  Operation : Invoke interface method $\langle$interface-methodref$\rangle$, with arguments a_1, a_2, $\ldots$, a_k, on the object reference o. Any return value v is pushed into the operand stack. The interface method reference $\langle$interface-methodref$\rangle$ is a constant pool entry.

  Type Constraints :

  - Let the type of the i'th actual argument be T, and the import type of the method reference $\langle$interface-methodref$\rangle$ is such that the i'th formal parameter has type T'. Then $T <: T'$.
  - Let the type of the the object reference o be T, and the import type of the $\langle$interface-methodref$\rangle$ is such that `this` has type T'. Then $T <: T'$.
  - The value v must have a type identical to the return type prescribed by the import type of the method reference $\langle$interface-methodref$\rangle$.

- *invokestatic* $\langle$methodref$\rangle$:

  Operand Stack :

  Before: $\ldots a_1$, a_2, $\ldots$, a_k

  After: $\ldots v$

  Operation : Invoke static method $\langle$methodref$\rangle$, with arguments a_1, a_2, $\ldots$, a_k. Any return value v is pushed into the operand stack. The method reference $\langle$methodref$\rangle$ is a constant pool entry.

  Type Constraints :

  - Let the type of the i'th actual argument be T, and the import type of the method reference $\langle$methodref$\rangle$ is such that the i'th formal parameter has type T'. Then $T <: T'$.

– The value v must have a type identical to the return type prescribed by the import type of the method reference $\langle$methodref$\rangle$.

7.1.3 Typechecking Procedure

Subclassing and symbol resolution To enforce type safety of subclassing, it suffices to make sure method overriding always satisfies the subtyping constraint. This check can be performed when the **endorse** primitive for the class in question is executed. Similarly, the typing constraint for symbol resolution can be checked when the corresponding **resolve** primitive is executed.

Typechecking methods To demonstrate the validity of a method type assertion, the standard iterative dataflow analysis algorithm [148] can be applied to verify if a type-consistent fix point for the type states can be found.

7.2 Read-Only Types in the Context of the Aegis VM

Having explored the essence of the JAC type system, and also eliciting the requirements of a JAC typechecker, attention is now turned to demonstrating that features designed into the POL and PVM facilities are sufficient for supporting the implementation of a link-time typechecker for the type system.

7.2.1 Embedding JAC Type Interface in Java Classfiles

To make JAC enforceable at link time, every Java classfile must carry a JAC type interface. The Java classfile format has an extension facility called attributes [130, Section 4.7]. Arbitrary annotations can be embedded in a classfile as an attribute. This section describes how a JAC type interface is encoded as a classfile attribute.

Type assertions must be encoded before they can be stored in an attribute. The JAC types `readonly` and $\perp$ are encoded as the ASCII characters 'R' and '.' (period) respectively. An encoded field type is a length-1 ASCII string containing an encoded JAC type. Method type encoding is not as straightforward. The JAC type signature of a method is encoded as an ASCII string in the following format:

"$t_0(t_1t_2\dots t_k)t$"

where

- t_0 is the encoded type of `this` for instance methods, and is '.' for static methods;
- t_1, t_2, ..., t_k are the encoded types of the formal parameters;
- t is the encoded return type for methods with a return value, and is '.' for void methods.

Such an encoding scheme yields a uniform encoding format for both instance and static methods. This is necessary because the constant pool does not carry information for one to differentiate a static method reference from an instance one.

A JAC type interface assigns a type encoding string to each export symbol and import reference. Such a type interface is embedded in a classfile as a *JAC attribute* with the following format:

```
struct JACMap {
  uint16_t id;
  uint16_t index;
};

struct JACTypeInterface {
  uint16_t attribute_name_index;
  uint32_t attribute_length;
  uint16_t padding;
```

```
  uint16_t constant_annotations_count;
  uint16_t field_annotations_count;
  uint16_t method_annotations_count;
  struct JACMap constant_annotations[constant_annotations_count];
  struct JACMap field_annotations[field_annotations_count];
  struct JACMap method_annotations[method_annotations_count];
};
```

The meaning of the fields is given below.

- `attribute_name_index`: Index of a UTF8 string in the constant pool. The UTF8 string must have value "JAC".
- `attribute_length`: Total number of bytes in this attribute, not counting the six bytes consumed by `attribute_name_index` and `attribute_length`.
- `padding`: Zero.
- `constant_annotations_count`: Number of import type assertions.
- `field_annotations_count`: Number of export type assertions for fields.
- `method_annotations_count`: Number of export type assertions for methods.
- `constant_annotations`: An array of type assertions, mapping the index of an import reference in the constant pool to the index of a UTF8 string in the constant pool. The string is the encoded type of the import reference.
- `field_annotations`: An array of type assertions, mapping the index of a field declaration in the classfile to the index of a UTF8 string in the constant pool. The string is the encoded export type of the field.
- `method_annotations`: An array of type assertions, mapping the index of a method declaration in the classfile to the index of a UTF8 string in the constant pool. The string is the encoded export type of the method.

A well-formed JAC attribute must assign no more than one type to an export symbol or an import reference. It is, however, not necessary for every export symbol or import reference to receive a type assignment. Entries left untyped are said to have *default types*. In fact, a classfile may not even have a JAC attribute. In this case, all export symbols and import references 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 $\perp$. The provision of assuming a default type interface for classfiles 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 classfiles can be reused without change.

A small utility was developed to facilitate the attachment of JAC attributes to classfiles. This C program takes a Java classfile and a high level JAC type interface specification file as input, and generates a version of the input classfile with the specified JAC attribute embedded.

7.2.2 Pluggable Obligation Library for JAC

A POL was implemented to provide facilities for discharging obligations formulated by the JAC PVM. As mentioned in the previous section, two kinds of conditions must be checked for at link time: (1) the import type of a constant pool entry must be compatible with the export type of the resolved target, and (2) method overriding must honor the subtyping constraint.

To facilitate proof obligation discharging, it is assumed that the verification interface generated by the modular JAC verifier will contain an optimized copy of the JAC attribute as its commitment data structure. The JAC POL exports the following predicates for enforcing the above two conditions:

1. *Import safety predicates*:

   `safe_field_import(`*field*`,` *import_type*`)`

   `safe_method_import(`*method*`,` *import_type*`)`

The first predicate function checks that the export type of field *field* is identical to the import type encoded in the UTF8 literal *import_type*. Similarly, the second predicate function checks that the export type of method *method* is identical to or a subtype of the import type encoded in the UTF8 literal *import_type*. The encoded export type of fields and methods can be retrieved from commitment data structures through the POL API. Type compatibility can then be checked by comparing the encoding of the two types involved.

2. *Method overriding safety predicate*:

   `safe_method_override(`*class*`)`

   For each of the method declared in *class*, check that it is a subtype of every method it overrides. This involves examining the type interfaces of superclasses and superinterfaces. The POL API provides functions for traversing the class hierarchy and retrieving JAC type interfaces as commitment data structures.

The POL also exports a global constant for representing default types. This default type constant may appear as an *import_type* argument in an import safety obligation. The predicates are implemented to account for such exceptional cases.

7.2.3 Pluggable Verification Module for JAC

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

1. If the classfile carries a JAC attribute, then the encoded type interface is checked for well-formedness. If no JAC attribute is found, then a default type interface is assumed. In either case, the JAC type interface of the classfile is cached as a commitment data structure.

2. The usual iterative dataflow analysis algorithm is applied to the body of each bytecode method. The type constraints of Section 7.1.2 are verified. The import type assertions are consulted for the effect of bytecode instructions.

Notice that, if all the import references have default types, then there is no need to run the dataflow analysis on a method with default export type[2]. A special case is that this step can be skipped entirely for classfiles with no JAC attribute.

3. A list of obligation attachments is generated. First, a corresponding import safety obligation is attached to each import reference in the constant pool. For example, an obligation of the following form will be generated for every field reference:

pred	*nargs*	*type-1*	*index-1*	*type-2*	*index-2*
`safe_field_import`	2	`PRE_ARG_Field`	i	`PRE_ARG_Constant`	j

Here, the *pred* field contains the predicate identifier of `safe_field_import`, i is the constant pool index of the field reference, and j is the constant pool index of an UTF8 literal containing the expected import type of the reference. If the import type of the field reference is not explicitly specified, then a default import type is assumed, and the following obligation will be generated instead.

pred	*nargs*	*type-1*	*index-1*	*type-2*	*index-2*
`safe_field_import`	2	`PRE_ARG_Field`	i	`PRE_ARG_GlobalConstant`	0

where the 0'th global constant is assumed to be an immutable data structure representing the default type. 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.

Second, a single method overriding safety obligation is attached to the current class.

pred	*nargs*	*type-1*	*index-1*
`safe_method_override`	1	`PRE_ARG_Relative`	0

[2]If a method has a default export type, then the initial type state will only contain $\perp$. Bytecode instructions only produce $\perp$ value if all the operands are $\perp$. This holds when all the import references have default type. As a result, the dataflow analysis will trivially succeed.

The sole argument refers to the target class (Figure 6.3).

4. A verification interface is generated for the target classfile. The verification interface includes the obligation attachments generated in step 3 and the commitment data structure from step 1. No relevant symbol is needed for this verification domain.

The JAC attribute encoding, and the POL and PVM described in this section enable the link-time enforcement of the JAC type system. The next section describes some example runs of this implementation.

7.3 Examples

Consider the List data structure in the running example:

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

Suppose an application class `Alice` needs to compute the sum of all integers in a `List` it creates. The task is delegated to 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 following *import* type assertion can be introduced to the classfile of `Alice`.

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

Specifically, the following encoded import type is assigned to the constant pool entry corresponding to the method reference `Bob.sum`.

“`.(R).`”

When equipped with the JAC POL and PVM, 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`.

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

When the class `Bob` is defined, the `verify` 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 `Bob.sum` is compatible with the import type of `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(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.

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

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`.

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

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

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

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 ($\texttt{readonly} \to \perp$) is a subtype of the default import type ($\perp \to \perp$).

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

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

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.

7.4 Summary

The PVM facility was applied to implement an augmented type system, namely, the access control type system JAC. This implementation exercise highlights the role of a verification interface in access control: proof obligations and commitments determine the manner in which a class is accessed.

Features offered by the generic proof linking mechanism and the PVM facility have been shown to be sufficient in supporting the link-time enforcement of JAC at the bytecode level. The provision of user-defined obligations, domain-specific global constants and class symbols, and pluggable verifiers yield a very general framework upon which a wide variety of static verification tasks can be programmed into the Aegis VM. As the Proof Linking architecture essentially decouples the verification logic from the dynamic linking process, implementing program-analytic protection mechanisms in the Aegis VM becomes a demonstrably tractable task.

Chapter 8

Conclusion

This chapter concludes the dissertation by offering a critical reflection of what has been achieved in the work, as well as directions for future research.

8.1 Discussion

Program safety is in general a whole-program notion: the safety of a code unit depends not only on properties that can be established by examining the unit alone, but also on the compatibility of the established properties with the runtime environment into which the code unit 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 the type interface and a given type environment. Cardelli succinctly called the two tasks *intrachecking* and *interchecking* [30]. Unfortunately, the two tasks are not cleanly separated in a typical implementation of the JVM bytecode verification procedure. As pointed out repeatedly in this dissertation, 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.

This work answers the need for modularity in mobile code verification. Intrachecking and interchecking are cleanly separated by the formulation of a *verification interface*. Through the inference of *proof obligations* and *commitments*, intrachecking of a classfile is carried out by a *modular verifier* in the absence of other closely-coupled classfiles. Interchecking thus involves the discharging of proof obligations using commitments of loaded classes, a process called *proof linking*. This setup is analogous to the notion of separate compilation, in which individual compilation units can be compiled separately because of an explicit module interface. Proof obligations and commitments can thus be seen as the interface that makes the *linking of safety proofs* a rational process even if each individual safety proof is generated separately.

Due to the incremental nature of lazy, dynamic linking, a mobile program may not be completely loaded or linked, and thus not all proof obligations can be discharged right away. Consequently, a notion of *obligation discharging schedule* is introduced to the verification interface. Every proof obligation is meant to protect a linking primitive that bears security significance. Execution of the linking primitive is only authorized if the proof obligation can be discharged successfully. The incremental proof linking process arbitrates the execution of linking primitives, while proof obligations are interposed at the entry points of linking services. Consequently, the proof linking facility can be seen as a *reference monitor for dynamic linking primitives*.

Articulating the correctness of incremental proof linking in a production mobile code environment turns out to be a non-trivial endeavor. Complex *temporal dependencies* among linking primitives are introduced by intermodular coupling, subtype relationships, and name binding constraints among separate namespaces. Such temporal dependencies are formally modeled through a *linking strategy*, which allows one to analyze the *timeliness* of obligation attachment (Safety) and commitment generation (Completion). Care must also be taken to ensure that logical inferences used to authorize linking primitives cannot be subsequently contradicted (Monotonicity).

The reformulation of Proof Linking to account for multiple classloaders in Java has been instructive. Firstly, it demonstrates that name resolution has to be explicitly mirrored in the obligation discharging process in the presence of multiple namespaces. Secondly, it shows that the *core initial theory* is *immutable* to the introduction of the added complexity. These findings informed the design of the POL mechanism, in which the complexity of name resolution is completely encapsulated in the Pluggable Obligation Library API, while leaving the tractable responsibility of crafting a core initial theory to the POL programmer.

Through the conception and implementation of *Pluggable Verification Modules*, an *extensible protection mechanism* has been developed. Link-time verification becomes a pluggable service that can be replaced, reconfigured and augmented. Application-specific verification services can be introduced into a mobile code system after the fact.

The significance of PVM as an enabling technology can only be understood through the appreciation of the need for enforcing safety properties at the bytecode level, at the time of dynamic linking. Program verification, when performed against source code, or when administrated by the code producer, cannot be trusted. What has been checked in the code producer's *verification environment* may not 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. Many augmented type systems previously proposed in the literature possess the potential to be used for enforcing application-specific security constraints. 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.

With the help of the PVM facility, such an application has been carried out for the JAC type system. Fine-grained access control policies can be programmed into the verification interface of a classfile to protect its transitive state from write access. Although write access is the only capability captured in JAC, the exercise suggests that PVM has opened up a possibility for enforcing more general forms of statically checkable access control policies at dynamic link time.

8.2 Related Works

This section offers a review of literature directly related to the research reported in this dissertation.

8.2.1 Verification Protocols

Proof-carrying code [144, 143, 146], proof-delegation [54, 55] and proof-on-demand are examples of distributed verification protocols. Proof Linking can be seen as an enabling technology for supporting interoperability of these verification protocols.

8.2.2 Formalization of Java Classloading and its Relationship to Bytecode Verification

Dean [48] pioneered the study of security issues surrounding the coexistence of static typing and dynamic linking. He postulated a *consistent extension* property, which states that typing judgments verified to be correct in the course of dynamic linking will not be contradicted subsequently by future classloading activities. He then used the PVS theorem prover to confirm that consistent extension is preserved by a simplified Java linking model, which accounts for user-initiated classloading. The main observation is that so long as the same class name is not defined more than once (in a given classloader[1]), then consistent extension can be achieved. This requirement was subsequently named as *temporal namespace consistency* [127]. The Monotonicity condition can be seen as a generalization of the consistent extension property for arbitrary verification domains.

The Java community was caught by complete surprise in 1997 when Saraswat from AT&T Research[2] posted an article on his homepage, announcing that Java is not

[1] Dean's work predates the Saraswat discovery [174], and so does not address complexity involving multiple classloaders.

[2] The author was an intern student at AT&T in that summer, fortunate enough to have the opportunity to learn firsthand from Saraswat about this flaw.

type-safe [174]. This anomaly reflects an omission in the Java/JVM specification itself, especially in the treatment of multiple classloaders. The cause of the problem has to do with the fact that different occurrences of a class name in the same lexical scope may be defined by different classloaders, and thus receive multiple, potentially contradicting definitions. This anomaly can be exploited in type-spoofing attacks, leading to run-time type confusion. The security hole was subsequently fixed through the imposition of *loading constraints*, which are essentially type equivalence constraints enforcing *namespace consistency among delegating classloaders* [127]. Since then the correctness of Java's classloading model, especially its relationship with the bytecode verifier, has been scrutinized under unprecedented vigilance by the formal verification community [109, 82, 160, 56, 200]. Among them the work of Qian, Goldberg and Coglio [160] shares many similarities with this work.

Built on their prior experience in formalizing various aspects of Java's bytecode verifier and its dynamic linking model [82, 43, 159, 41], Qian *et al* [160] proposed a formal specification of Java's 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. Recursive 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 [127]. Linking primitives are modeled as "*macro operations*", although the articulation of a linking strategy has not been explicit. Despite the many similarities in our treatment of modularization and dynamic linking, the following differences are observed:

- **Generality**: While a proof obligation can be an arbitrary query, 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 multiple verification domains involving a wide spectrum of initial theories.

- **Constraint Maintenance vs Query Satisfaction**: Qian *et al*'s subtype constraints are *maintained on-the-fly* very much like the type equivalence constraints in [127, 130]. In contrast, proof obligations are scheduled to be discharged prior to the execution of their respective linking primitives. This scheduling element introduces an additional dimension of complexity into the correctness proof of Proof Linking (Safety and Completion). By restricting oneself to only consider Java subtype constraints, on-the-fly constraint maintenance is conceptually simpler. Whether this simplicity translates to actual performance gain is a topic of interest.

- **Implementation and Embodiment**: The constraint system of Qian *et al* began as a *modeling apparatus* for formalizing the bytecode verification algorithm and its relationship to classloading. It was subsequently implemented in the constraint solving system SPECWARE as a *detached* bytecode verifier [43]. In contrast, the notion of proof obligations, commitments and incremental proof linking were *implemented* and *embodied* in an actual JVM for supporting regular bytecode typechecking and also Pluggable Verification Modules.

The standard Java bytecode verification algorithm, as outlined in the JVM specification [130, Section 4.9.2], computes the meet of two reference types as their most specific common superclass. This requires the immediate loading of the superclasses of the two reference types during the course of bytecode verification. To avoid recursive classloading, the Prelude bytecode typechecker defers the loading of superclasses by formulating *meet expressions* and *composite obligations* (Section 6.3.2). First reported in [71], this technique was independently employed by Coglio and Goldberg [41, 42] to fix a bug in the standard Java bytecode verification algorithm that originates from the subtleties of multiple classloaders.

For a sample of recent works on formalization of the JVM, consult [93, 149]. See also [190], which provides a mathematical specification of the Java language and the JVM (including the bytecode verification components) using Abstract State Machines. The formal models of the JVM thus obtained are executable, and have been formally verified to be safe.

8.2.3 Type-Safe Dynamic Linking

Cardelli [30] defined a formal model for type-safe, *static* linking in the simply-typed lambda calculus F_1 (without recursion). Linking is characterized as a series of substitutions that preserve type safety invariants. The **verify** primitive corresponds to Cardelli's *intra-checking*, while the **endorse** and **resolve** primitives could be seen as an incremental version of *inter-checking*. The approach of this work differs substantially from Cardelli's in the treatment of typing environments. In particular, Cardelli's notion of an import environment as an input to intra-checking is replaced by the notion of obligations produced as output. In essence, obligations represent a logical specification of all allowable typing environments for which a module intra-checks. This technique is key to the implementation of modular verification. Furthermore, to cope with lazy, dynamic linking, obligations are explicitly scheduled to be discharged incrementally: an obligation discharging schedule is an integral part of a verification interface. The second distinction in the treatment of typing environments is that Cardelli's notion of export environments is replaced by the set of commitments produced during module verification. In this case, however, the replacement is essentially a direct encoding of the typing environment in logical form. To mention a last point of comparison, the correctness of Cardelli's model is dependent, though implicitly, on a specific ordering of substitution steps [30, Lemma 3-3 & Section 6]. In the Proof Linking framework, the interaction between verification correctness and relative ordering of linking events are formalized explicitly as three correctness conditions, namely, Safety, Monotonicity, and Completion.

Building on the work of Cardelli, Glew and Morriset [80] proposed the typed object file as an extension to Typed Assembly Language (TAL) [142]. TAL object programs are annotated with type information, to be type checked at link time. The typed object file provides a means for safe, modular type checking of separately compiled code units. As the authors note, however, this approach does not naturally extend to lazy, dynamic linking. The deficiency was later remedied by Duggan [57] , who proposed a very general, type-safe module system, which accounts for recursive module dependencies, dynamic linking, dynamic loading, and shared libraries.

8.2.4 User-Defined Type Qualifiers

Foster *et al* [75] 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 [179]. The framework was subsequently extended to account for flow-sensitive type qualifiers [76]. 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 is essential for enforcing safety in a lazy, dynamic linking environment.

8.2.5 Extensibility and Modularity

With modularization comes extensibility. Proof Linking decouples the verification logic from dynamic linking, thereby rendering verification a pluggable service. A major contribution of this dissertation is the introduction of an extensible protection mechanism enabled by a modular architecture. Yet, modularization is not the only way to extensibility. Three closely related concepts – aspect-oriented programming, reflection and metaobject protocols, and load-time code rewriting, can be thought of as general-purpose program extension mechanisms.

- *Aspect-Oriented Programming*: Originally proposed as an alternative to encapsulation as a means for implementing separation of concern, aspect-oriented programming [116, 117] 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.

- *Reflection and Metaobject Protocols*: Behavioral reflection [65, 219] and intercessory metaobject protocols [118] 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 callback facility. Programmers can customize the semantics of the metaobject, thereby achieving the effect of software extension.

- *Load-Time Code Rewriting*: Load-time code rewriting has been employed to introduce security checks into mobile code systems [59, 201, 202, 215, 167, 168]. Dedicated classloaders can be programmed in Java to inject behavioral changes into untrusted code.

8.2.6 Programming Languages for Developing Verifiable Systems

Mobile code systems in general, and the Proof Linking architecture in particular, wrestle with a number of design issues once faced by early programming languages designed for developing verifiable systems (e.g., Euclid [156, 224] and GYPSY [10]).

- **Verification interface.** Euclid supports the specification of modules and routines through the use of Hoare-style pre- and post-assertions, and invariants [132]. Interchecking, that is, proof of usage validity, can be performed with the assertions in the absence of implementation. Intrachecking is then performed separately by assuming the pre-assertions and then inferring the post-assertions.

- **Proof obligations.** The Euclid compiler generates proof obligations in the form of legality assertions [223], which can then be inlined into the object code as run-time checks.

- **Programmer control over visibility, scope and aliasing.** To the best knowledge of the author, Euclid and GYPSY are the original sources for the idea of explicit access control and alias control in programming languages. Visibility of identifiers, including the kind of access permitted, can be modified by Euclid programmers at scope boundaries. Alias creation is forbidden in Euclid and GYPSY in order to simplify the proof rules of the languages. See [11] for an early study of protection in programming languages for mutually distrusting code units.

A unique feature of Proof Linking, which is absent from the design of these early languages, is the introduction of a scheduling element into the verification interfaces in order to deal with the reality of lazy, dynamic linking, even in the presence of multiple namespaces.

8.3 Contributions

8.3.1 Primary Contributions

The primary contribution of this dissertation lies in the definition of a modular verification architecture, Proof Linking, for mobile code systems. The soundness and adequacy of the architecture have been evaluated in the context of Java bytecode verification. Its implementation feasibility has been established by the availability of a reference implementation. The following list highlights the specific aspects of the primary contribution.

1. **The Proof Linking architecture**

   (a) Through the formulation of verification interfaces in the form of proof obligations and commitments, the Proof Linking architecture fully decouples static verification from dynamic linking, thereby enabling separate verification.

   (b) A novel scheduling element has been introduced into the verification interface so that obligation discharging can be appropriately staged to dovetail with the incremental process of lazy, dynamic linking.

2. **Modeling adequacy and soundness**

   (a) A novel notion of linking strategy has been employed to formalize the temporal dependencies among linking primitives, thereby making it possible to establish the correctness of Proof Linking in a formal framework.

   (b) Three correctness conditions for Proof Linking have been formalized — Safety, Monotonicity, Completion.

   (c) Increasingly realistic instantiations of Proof Linking for Java bytecode verification have been shown to satisfy the aforementioned correctness conditions.

3. **Implementation feasibility**

   (a) Under the Proof Linking architecture, the Java bytecode verifier can be specified, crafted, understood, and evaluated as a separate engineering component. Specifically, such a stand-alone bytecode verification module has been implemented in the open source Prelude library.

   (b) The Proof Linking architecture enables an extensible protection mechanism, Pluggable Verification Modules (PVM), in which well-mannered static program analyses can be introduced into the dynamic linking process of the JVM. A reference implementation of PVM has been realized in an open source JVM, the Aegis VM. The PVM facility of the Aegis VM is the first known technology of this sort.

   (c) The design of the Aegis VM and its PVM facility achieves a novel tradeoff balancing the simultaneous need for generality, efficiency and utility.

(d) The generality and utility of the PVM facility have been assessed favorably in an experimental application of the facility to enforce a published augmented type system, JAC.

8.3.2 Secondary Contributions

Listed below are contributions not directly related to the establishment of the thesis.

1. An extensive overview and taxonomy of mobile code protection mechanisms has been developed.

   (a) Three distinct security challenges for mobile code systems have been identified: anonymous trust, layered protection, and implicit acquisition.

   (b) Existing mobile code protection mechanisms have been categorized into four major approaches: discretion, verification, transformation, and arbitration.

   (c) The notion of distributed verification protocols has been proposed, and its design space has been analyzed.

2. The JAC type system has been recast into a version suitable for enforcing access control at the bytecode level, at link time. The techniques employed can be applied to other augmented type systems.

8.4 Limitations and Future Works

In this section, the limitations of this work are discussed along with possible extensions.

Generality The generality of the generic proof linking mechanism has been supported by only one experiment — the application of PVM to enforce the augmented type system JAC. To further assess the generality of the design, more experiments are needed. It will be interesting to demonstrate that PVM can be applied to a wide spectrum of verification domains, ranging from low level type systems such as those for alias control [121, 28, 6] to high level partial specification systems such as AAL [115], or even safety properties at the software architectural level [5, 7].

One specific question usually raised by observers of this work has been whether Proof Linking can handle polymorphic type systems. Examination of the JAC obligation set suggests that a positive answer is highly plausible. The crux is to design an efficient and compact encoding for polymorphic types that appear in a type interface. Polymorphic type inference can be programmed into a POL and a PVM. A polymorphic extension of the JAC type system for Java bytecode has been planned, successful implementation of which will offer a definite positive answer to the question.

Efficiency The generic proof linking mechanism has been designed with efficiency in mind. The degree to which this design delivers its promise has not been substantiated by experimental data. One future direction is to conduct empirical studies so as to understand the performance characteristics of the generic proof linking mechanism.

To reduce the cost of program analysis within the PVM, the Aegis VM passes to the PVM verification function the abstract syntax tree and type analysis results of the target classfile. The verification function can utilize the control flow and typing information to speed up its analysis. The question is whether one can do better than this. For example, the JAC PVM has to perform its own dataflow analysis to ensure there is a fix point for which all type constraints are satisfied. The process will greatly speed up 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 Prelude bytecode typechecker to summarize the dataflow structure of a bytecode method in an SSA-based representation [16] along the line of Jimple [203] or SafeTSA [12].

Distributed Verification Proof Linking has been claimed to be an enabling technology for verification protocol interoperability and conditional certification. The Java instantiation presented in Chapter 5 has been carefully designed to anticipate such an application. Yet, no empirical evidence has been provided to confirm its effectiveness in supporting distributed verification. More empirical work is needed before the claim can be substantiated.

Theoretical Considerations The formalization of the incremental proof linking process explicitly models linking primitives and their temporal dependencies. It is interesting to see if modeling apparatus such as linking strategies can be combined with the work of Cardelli [30] to yield a formal characterization of type-safe dynamic linking for a Java-like object-oriented programming language, in which complex temporal dependencies exist among linking events.

Another theoretical question is to understand the expressive power of the Proof Linking framework. Exactly what properties can be enforced by posting proof obligations? What is the effect of restricting the expressiveness of the underlying query language (to, say, a sublanguage of Horn-clause logic)?

Generalizing the formal specification and verification work reported in Appendix A to handle multiple classloaders involves the introduction of a dynamically constructed class hierarchy. The formal dynamic linking model proposed in [109] seems to fit well into the Proof Linking framework.

More Extensible Protection Mechanisms Proof Linking is a generalization of the link-time access control checks performed in a standard JVM. By making these checks customizable, one obtains an extensible protection mechanism for the JVM. Another set of checks performed by the JVM in the course of dynamic linking are loading constraints, which are essentially equivalence constraints over binding of class symbols from different namespaces [127]. An interesting direction is to generalize the idea of Qian *et al* [160], and make loading constraints customizable: users may introduce domain-specific constraint systems over the binding of class symbols, and maintain binding consistency with pluggable constraint solvers. This flexibility yields an extensible protection mechanism for which the subtype constraint system of [160] becomes a special case.

Both the extensible loading constraint system suggested in the previous paragraph, and the PVM 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 could then be weaved into these join points as security aspects. The standardization of extension mechanisms reduces the probability of programming error, and greatly simplifies the process of transforming an existing protection mechanism into an extensible one.

Access Control Types The JAC type system formalizes the control of write access in a type system. Implementation of JAC as a PVM demonstrates that access control type systems can indeed be enforced at link time, at the bytecode level. It is interesting to see if some of the access control type systems [121, 28] originally proposed for alias control can be *lifted* to the architectural level [5] for enforcing system-level security policies, thereby obtaining a purely program-analytic access control mechanism for the JVM.

8.5 Conclusion

The recent advent of global internetworking and the growing demand for dynamic software extensibility have made mobile code systems a reality. With new possibilities come new security risks. The unique security challenges presented by mobile code systems have generated exciting synergies between core Computer Science subdisciplines such as Computer Security, Programming Languages, Operating Systems and Software Engineering. The research reported in this dissertation addresses the need for modularity in the verification of mobile code fragments, examines the interaction of modular verification with lazy, dynamic loading and namespace partitioning, and exploits modularization to create an extensible protection mechanism. Inasmuch as a deeper understanding of modular verification in the context of mobile code security has been achieved, exciting new research directions have also been identified. It is therefore fitting to conclude this work with the words of Eliot:

> *We shall not cease from exploration*
> *And the end of all our exploring*
> *Will be to arrive where we started*
> *And know the place for the first time*
>
> **T. S. Eliot, *Four Quartets***

Appendix A

Correctness Proof by PVS

The proof of Safety, Monotonicity, and Completion as presented informally in Chapter 4 can be formally verified using a theorem prover. In particular, the above properties have been formally established using the PVS specification and verification system [188]. This appendix reports on the exercise, in order (1) to demonstrate that the verification of Safety, Monotonicity, and Completion can be performed rigorously with the help of a theorem prover, (2) to illustrate the specification and proof techniques that are found to be helpful in such an endeavor, and (3) to highlight the improved understanding of proof linking that may be gained as a result.

A.1 Intended Model

Before one can specify and prove theorems about the correctness of proof linking, one has to define an *intended model* for the first-order theory that is used in proof linking. Specifically, one has to define the intended meaning for predicate symbols "`extends`", "`subclass`", and so on. To this end, one must give a specification of the class hierarchy in a way that captures not only properties that the modular verifier enforces, but also the potential anomalies that could arise if proof linking is not performed properly.

For instance, a modular verifier can guarantee that the class *'java/lang/Object'* has no immediate superclass, that interfaces have *'java/lang/Object'* as their only direct superclass, and that all other classes have a unique direct superclass. However, confined to examine one code unit at a time, a modular verifier cannot rule out the possibility of circular subclassing (i.e., two classes being subclasses of each other) and subclassing from an interface class. Such anomalies must be made possible in our specification of the intended model. To capture these, the following are specified:

- `class` is a non-empty type.
- `java_lang_object` is a distinguished object of type `class`. All other `class` objects have type `(non_root_class?)`.
- The set `(non_root_class?)` is further partitioned into two subsets, `(interface?)` and `(proper_class?)`.
- The function below is defined to map a non-interface class to its unique, direct superclass.

  ```
  extends : [(proper_class?) -> class]
  ```

  Notice that circularity and subclassing from an interface class is thus allowed.
- As an "extension" to the function `extends`, the following predicate is defined to capture the fact that `java/lang/Object` has no superclass and that interfaces extend `java/lang/Object`.

  ```
  direct_super_class? : [class -> class -> bool]
  ```
- The following predicate is defined to be the transitive closure of `direct_super_class?`.

  ```
  super_class? : [class -> class -> bool]
  ```

Other notions like subinterfacing, subclassability, and so on are completely specified according to what the modular verifier enforces and allows. This part of the specification documents the capabilities and limitations of the modular verifier.

A.2 Strategy

To specify linking strategies, an abstract datatype `primitive` is defined. A PVS datatype declaration introduces constructors and accessors for each of the subtypes (e.g., the constructor for the linking primitive "**verify** X" is `PRIM_verify(X)`). A binary relation `before` over `primitive` is then defined to represent the partial ordering as specified in Section 4.2.2. Notice that the specification defines the linking strategy in terms of the intended model. For example, the Subtype Dependency Property requires `before(PRIM_verify(Y), PRIM_endorse_class(X))` to hold if `super_class?(X)(Y)` is true.

For the sake of clarity and specification economy, `before` is specified in the following manner. Each of Natural Progression Property, Import-Checked Property, Subtype Dependency Property and Referential Dependency Property are captured in a separate relation. Another binary relation `Precede` is then defined as a union of the four. Also, only the immediate precedence of primitives is specified in `Precede`. A binary relation `before` is then defined as the transitive closure of `Precede`.

When a new strategy is defined, it is imperative to check that it actually defines a strict partial ordering over the set of primitives. To illustrate this necessity, consider an alternative formulation of the Subtype Dependency Property, in which it is required that

$$\textbf{endorse}Y < \textbf{endorse}X$$

for any class X and its *direct* superclass or *direct* super interface Y. Despite the subtle difference, this formulation appears to have achieved everything one wants a Subtype Dependency Property to achieve, namely, forcing all superclasses and superinterfaces of X to be verified before X is endorsed. Unfortunately, such a formulation also introduces an inconsistency: the resulting strategy is not a strict partial ordering. Recall that the modular verifier cannot rule out circular subclassing. In the case when X and Y are subclasses of each other, the above formulation places "**endorse** X" and "**endorse** Y" before each other, an impossibility if `before` is to be a strict partial ordering. It is through this articulation process that current formulation of

Subtype Dependency Property has come to be adopted instead of the alternative shown here.

To prove that `before` is a strict partial ordering, one has to show that it is transitive (which is trivial since `before` is defined as a transitive closure of `Precede`) and irreflexive. The latter can be shown by, firstly, assigning an (integer) ordinal number[1] to each primitive and, secondly, showing that the ordering of ordinals preserves the ordering of `before`. Irreflexivity follows since no integer is less than itself.

A.3 Database

An abstract datatype `predicate` is defined to capture the signature of the predicate symbols used in proof linking. A `database` is represented as a `set` of `predicate`. Also defined is a mapping `model : [predicate -> bool]` which correlates a `predicate` to the relation it denotes. For example, `model(PRED_extends(`X`, `Y`))` maps to the value of `direct_super_class?(`X`)(`Y`)`.

Obligation attachments specified in Section 4.2.4 are encoded by a relation `may_attach?(`q`)(`o`)(`p`)` that evaluates to true when `primitive` q could potentially attach `predicate` o to `primitive` p as an obligation. (This relation captures the $att(\cdot)$ mapping defined on page 74.) We also define the shorthand `potential_obligation?(`p`)(`o`)` to mean ($\exists$ (q) : `may_attach?`(q)(o)(p)). (This relation formalizes the $obl(\cdot)$ mapping defined on page 74.) In addition, Figure 4.4 is mirrored by a relation `commit?(`p`)(`c`)` that evaluates to true when `primitive` p asserts `predicate` c as a commitment. (This relation captures the $com(\cdot)$ mapping defined on page 74.) Both `may_attach?` and `commit?` are defined in terms of the intended model. For example, `commit?(PRIM_verify(`X`))(PRED_extends(`X`, `Y`))` is true iff the relation `direct_super_class?(`X`)(`Y`)` holds in the intended model. This formally captures the condition under which the modular verifier generates a specific commitment or obligation. One can also sanity-check the definition of `commit?` by proving the following challenge using case analysis:

[1]PVS automatically defines an ordinal number for members of an abstract datatype.

$$\text{CONSISTENT\_COMMITMENT} : \text{LEMMA}$$
$$(\forall\ (p : \text{primitive}, c : \text{predicate}) : \text{commit?}(p)(c) \Rightarrow \text{model}(c))$$

A `state` is defined to be a `set` of `primitive`. Intuitively, a `state` describes the set of primitives that are already terminated at a certain point of the proof linking process. A `state_database` is then defined to be a mapping from a `state` to a `database` that contains all the predicates committed by members of the given `state`. Also defined is the function `STATE_before : [primitive -> state]`, which maps a primitive to the state containing all the primitives that are *guaranteed* to have terminated before the initiation of the given primitive. As a result, the expression `state_database(STATE_before(`p`))` gives the database containing all commitments that are guaranteed to be available prior to the execution of a `primitive` p.

Query evaluation and the initial theory are captured by an inductively defined relation `provable? : [[predicate, database] -> bool]`, which captures whether a `predicate` is provable in a given `database`. For non-recursive queries, `provable?` simply checks if the predicate is an element of the database. For recursive queries, `provable?` unfolds the query inductively. For instance, `provable?(PRED_subclass(`X`,` Y`),` DB`)` is true iff the following is true:

$$\begin{aligned}
&X = Y \vee \\
&\quad (\exists\ (Z : \text{class}) : \\
&\qquad \text{provable?}(\text{PRED_extends}(X, Z), \text{db}) \wedge \\
&\qquad \text{provable?}(\text{PRED_subclass}(Z, Y), \text{db}))
\end{aligned}$$

To sanity-check the definition, and to prepare for proving completion, a generalization of the `CONSISTENT_COMMITMENT` lemma is verified:

$$\begin{aligned}
&\text{SOUNDNESS} : \text{THEOREM} \\
&\quad (\forall\ (o : \text{predicate}) : \\
&\qquad (\exists\ (S : \text{state}) : \text{provable?}(o, \text{state_database}(S))) \Rightarrow \text{model}(o))
\end{aligned}$$

The above theorem says that, once an obligation is shown to be provable in a database, it means that the relation it denotes is true in the intended model. The theorem is established by induction, of which the `CONSISTENT_COMMITMENT` lemma serves as a base case.

All the definitions ready, we are now in the position to establish the correctness conditions.

A.4 Correctness Proofs

The discussion of the Safety condition, which can be checked by straightforward case analysis, is skipped for brevity. Monotonicity is captured by the following theorem:

$$
\begin{array}{l}
\text{MONOTONICITY}: \ \textsc{theorem} \\
\quad (\forall\ (\text{DB1}:\ \text{database}, \text{DB2}:\ \text{database}): \\
\qquad (\text{DB1} \subseteq \text{DB2}) \Rightarrow \\
\qquad\quad (\forall\ (o:\ \text{predicate}):\ \text{provable?}(o, \text{DB1}) \Rightarrow \text{provable?}(o, \text{DB2})))
\end{array}
$$

The theorem says that, if an obligation is provable in a database, it stays provable even if more clauses are added into the database. This theorem can be proven by induction on the relation `provable?`. Strictly speaking, the above proof is not necessary because the use of Horn clauses trivially guarantees Monotonicity (Section 3.2.3). Yet, the ability to establish Monotonicity without relying on the syntactic characteristics of the logic suggests that a more general form of logic may be used as long as the MONOTONICITY theorem can be proven.

The Completion condition can be represented in the following theorem:

$$
\begin{array}{l}
\text{COMPLETION}: \ \textsc{theorem} \\
\quad (\forall\ (p:\ \text{primitive}, o:\ \text{predicate}): \\
\qquad \text{potential\_obligation?}(p)(o) \Rightarrow \\
\qquad\quad (\exists\ (S:\ \text{state}):\ \text{provable?}(o, \text{state\_database}(S))) \Rightarrow \\
\qquad\qquad \text{provable?}(o, \text{state\_database}(\text{STATE\_before}(p))))
\end{array}
$$

To understand the above formulation, recall that Completion requires that, if an obligation check fails, then subsequently asserted commitments cannot serve to establish it. Conversely, if an obligation can ever be established, it must be satisfied at the time of checking. Formally, if an obligation o can be attached to a primitive p, and if o is provable in some database, then it must be provable immediately prior to the execution of p.

Proof of Completion is considerably more challenging than establishing Safety and Monotonicity. In order to establish the above theorem, the completeness of the query evaluation procedure is first established by induction and by applying the `MONOTONICITY` theorem:

$$
\begin{array}{l}
\text{COMPLETENESS}: \ \textsc{lemma} \\
\quad (\forall\ (p: \ \text{primitive}, o: \ \text{predicate}): \\
\qquad \text{potential\_obligation?}(p)(o) \Rightarrow \\
\qquad\quad \text{model}(o) \Rightarrow \text{provable?}(o, \text{state\_database}(\text{STATE\_before}(p))))
\end{array}
$$

The COMPLETENESS theorem says that, if an obligation o can be attached to a linking primitive p, and if the relation denoted by o is true in the intended model, then o must become provable before p is executed. It is easy to see that the `SOUNDNESS` theorem and the `COMPLETENESS` theorem together imply `COMPLETION`.

The idea behind the above proof is to show that the dynamically evolving commitment database is always consistent with a static typing model. Since the relations defined in the static model are invariant, and since the commitments always mirror these static relations, subsequently asserted commitments never contradict each other. Establishing consistency between sets of clauses by finding a shared model is a standard technique in formal logic.

Appendix B

Pluggable Obligation Library API

This appendix contains the C header file that every Pluggable Obligation Library must include. The file captures the POL API. The version shown here is slightly edited to improve presentation.

```
/*
    libaegisvm - The Aegis Virtual Machine for executing Java bytecode
    Copyright (C) 2003  Philip W. L. Fong

    This library is free software; you can redistribute it and/or
    modify it under the terms of the GNU Lesser General Public
    License as published by the Free Software Foundation; either
    version 2.1 of the License, or (at your option) any later version.

    This library is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
    Lesser General Public License for more details.

    You should have received a copy of the GNU Lesser General Public
    License along with this library; if not, write to the Free Software
    Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
```

```
*/

#ifndef __AE_POL_H__
#define __AE_POL_H__

/*
 * Name of the global variable containing the profile of a POL.
 */

#define POL_PROFILE_SYMBOL "pol_profile"

/*
 * Oblique types for referring to internal data structures of VM.
 */

typedef const void *POLEnv;
typedef const void *POLPackage;
typedef const void *POLClassloader;
typedef const void *POLClass;
typedef const void *POLField;
typedef const void *POLMethod;

/*
 * Type signatures for predicate functions, initialization function,
 * and clean-up function.
 */

typedef bool (*pol_predicate_t)(POLEnv env,
uint16_t nargs,
const void *args[const]);

typedef int (*pol_init_t)(void);
```

```
typedef int (*pol_finish_t)(void);

/*
 * Type interface for POL profile
 */

typedef struct pol_profile_t pol_profile_t;

struct pol_profile_t {
  const char *id;
  pol_init_t init;
  pol_finish_t finish;
  uint16_t npreds;
  const pol_predicate_t *predicates;
  uint16_t nclasses;
  const char * const *global_classes;
  uint16_t nconstants;
  const void * const *global_constants;
};

/*
 * Package interface interrogation
 */

const char *pol_package_name(POLPackage package);

/*
 * Class interface interrogation
 */

const char *pol_class_utf8_constant(POLClass class, uint16_t index);
ji4 pol_class_integer_constant(POLClass class, uint16_t index);
```

```
jf4 pol_class_float_constant(POLClass class, uint16_t index);
ji8 pol_class_long_constant(POLClass class, uint16_t index);
jf8 pol_class_double_constant(POLClass class, uint16_t index);
bool pol_class_is_public(POLClass class);
bool pol_class_is_final(POLClass class);
bool pol_class_is_super(POLClass class);
bool pol_class_is_interface(POLClass class);
bool pol_class_is_abstract(POLClass class);
bool pol_class_is_primitive(POLClass class);
bool pol_class_is_array(POLClass class);
bool pol_class_is_array_of_reference(POLClass class);
bool pol_class_is_array_of_primitive(POLClass class);
bool pol_class_is_loaded(POLClass class);
POLClass pol_class_array_component(POLClass class);
POLPackage pol_class_package(POLClass class);
POLClassloader pol_class_classloader(POLClass class);
const char *pol_class_name(POLClass class);
POLClass pol_class_super_class(POLClass class);
uint16_t pol_class_interfaces_count(POLClass class);
POLClass pol_class_interface(POLClass class, uint16_t index);
uint16_t pol_class_fields_count(POLClass class);
POLField pol_class_field(POLClass class, uint16_t index);
uint16_t pol_class_methods_count(POLClass class);
POLMethod pol_class_method(POLClass class, uint16_t index);

/*
 * Field interface interrogation
 */

bool pol_field_is_public(POLField field);
bool pol_field_is_private(POLField field);
bool pol_field_is_protected(POLField field);
bool pol_field_is_package_private(POLField field);
```

```
bool pol_field_is_static(POLField field);
bool pol_field_is_final(POLField field);
bool pol_field_is_volatile(POLField field);
bool pol_field_is_transient(POLField field);
POLClass pol_field_class(POLField field);
uint16_t pol_field_index(POLField field);
const char *pol_field_name(POLField field);
const char *pol_field_descriptor(POLField field);

/*
 * Method interface interrogation
 */

bool pol_method_is_public(POLMethod method);
bool pol_method_is_private(POLMethod method);
bool pol_method_is_protected(POLMethod method);
bool pol_method_is_package_private(POLMethod method);
bool pol_method_is_static(POLMethod method);
bool pol_method_is_final(POLMethod method);
bool pol_method_is_synchronized(POLMethod method);
bool pol_method_is_native(POLMethod method);
bool pol_method_is_abstract(POLMethod method);
bool pol_method_is_strict(POLMethod method);
POLClass pol_method_class(POLMethod method);
uint16_t pol_method_index(POLMethod method);
const char *pol_method_name(POLMethod method);
const char *pol_method_descriptor(POLMethod method);
uint16_t pol_method_exceptions_count(POLMethod method);
const char *pol_method_exception_name(POLMethod method, uint16_t index);

/*
 * Subtyping relationship
 */
```

```
bool pol_subclass(POLClass class1, POLClass class2);
bool pol_subinterface(POLClass class1, POLClass class2);
bool pol_assignable(POLClass class1, POLClass class2);

/*
 * Contextual information
 */

POLClass pol_env_global_class(POLEnv env, uint16_t index);
const void *pol_env_global_constant(POLEnv env, uint16_t index);
const void *pol_env_commitments(POLEnv env, POLClass class);

#endif
```

Appendix C

Pluggable Verification Module API

This appendix contains the C header file that every Pluggable Verification Module must include. The file captures the PVM API. The version shown here is slightly edited to improve presentation.

```
/*
    libaegisvm - The Aegis Virtual Machine for executing Java bytecode
    Copyright (C) 2003  Philip W. L. Fong

    This library is free software; you can redistribute it and/or
    modify it under the terms of the GNU Lesser General Public
    License as published by the Free Software Foundation; either
    version 2.1 of the License, or (at your option) any later version.

    This library is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
    Lesser General Public License for more details.

    You should have received a copy of the GNU Lesser General Public
    License along with this library; if not, write to the Free Software
    Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
```

```
*/

#ifndef __AE_PVM_H__
#define __AE_PVM_H__

#include <jpr/type.h>
#include <jpr/arena.h>
#include <prelude/classfile.h>
#include <prelude/type_analysis.h>

/*
 * Name of the global variable containing the profile of a PVM.
 */

#define PVM_PROFILE_SYMBOL "pvm_profile"

/*
 * Type signatures for verifier function, initialization function,
 * and clean-up function.
 */

typedef int (*pvm_init_t)(void);

typedef int (*pvm_finish_t)(void);

typedef const PREVI *(*pvm_verify_t)(JArena *arena,
                                     const PREClassFile *classfile,
                                     const PREAnalysis **analyses);

/*
 * Type in terface for PVM profile
 */
```

```
typedef struct pvm_profile_t pvm_profile_t;

struct pvm_profile_t {
  const char *pol;
  pvm_init_t init;
  pvm_finish_t finish;
  pvm_verify_t verify;
};

#endif
```

Bibliography

[1] Martin Abadi, Michael Burrows, Butler Lampson, and Gordon D. Plotkin. A calculus for access control in distributed systems. *ACM Transactions on Programming Languages and Systems*, 15(4):706–734, September 1993.

[2] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, 2nd edition, 1996.

[3] Anurag Acharya, Mustafa Uysal, and Joel Saltz. Active disks: programming model, algorithms and evaluation. In *Proceedings of the 8th International Conference on Architectural Support for Programming Lanugages and Operating Systems*, pages 81–91, San Jose, California, October 1998.

[4] Ali-Reza Adl-Tabatabai, Geoff Langdale, Steven Lucco, and Robert Wahbe. Efficient and language-independent mobile programs. In *Proceedings of ACM SIGPLAN'96 Conference on Programming Language Design and Implementation (PLDI'96)*, pages 127–136, May 1996.

[5] Jonathan Aldrich, Craig Chambers, and David Notkin. ArchJava: Connecting software architecture to implementation. In *Proceedings of the 24th International Conference on Software Engineering*, pages 187–197, Orlando, FL, May 2002.

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

[7] Jonathan Aldrich, Vibha Sazawal, Craig Chambers, and David Notkin. Language support for connector abstractions. In *Proceedings of the 17th European Conference on Object-Oriented Programming*, Darmstadt, Germany, July 2003.

[8] Paulo Sérgio Almeida. Balloon types: Controlling sharing of state in data types. In *Proceedings of the 11th European Conference for Object-Oriented Programming (ECOOP'97)*, Jyvaskyla, Finland, June 1997.

[9] Bowen Alpern and Fred B. Schneider. Recognizing safety and liveness. *Distributed Computing*, 2(3):117–126, 1987.

[10] Allen L. Ambler, Donald I. Good, James C. Browne, Wilhelm F. Burger, Richard M. Cohen, Charles G. Hoch, and Robert E. Wells. GYPSY: A language for specification and implementation of verifiable programs. In *Proceedings of an ACM Conference on Language Design for Reliable Software*, pages 1–10, Raleigh, North Carolina, March 1977.

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

[12] 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.

[13] Patricia Anthony and Nicholas R. Jennings. Developing a bidding agent for multiple heterogeneous auctions. *ACM Transactions on Internet Technology*, 3(3):185–217, August 2003.

[14] Ken Arnold, James Gosling, and David Holmes. *The Java Programming Language*. Addison Wesley, 3rd edition, 2000.

[15] John Aycock. A brief history of just-in-time. *ACM Computing Surveys*, 35(2):97–113, June 2003.

[16] 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.

[17] Henry G. Baker. 'Use-Once' variables and linear objects — storage management, reflection and multi-threading. *ACM SIGPLAN Notices*, 30(1):45–52, January 1995.

[18] Dirk Balfanz and Ed Felten. A Java filter. Technical Report 567-97, Department of Computer Science, Princeton Univerisyt, October 1997.

[19] Lujo Bauer, Jarred Ligatti, and David Walker. More enforceable security policies. In *Proceedings of the Workshop on Foundations of Computer Security (FCS'02)*, Copenhagen, Denmark, July 2002.

[20] D. Elliot Bell and Leonard J. LaPadula. Secure computer systems: Mathematical foundations. Technical Report 2547 (Volume I), MITRE, March 1973.

[21] Brian Bershad, Stefan Savage, Przemyslaw Pardyak, David Becker, Marc Fiuczynski, and Emin Gun Sirer. Protection is a software issue. In *Proceedings of the Fifth Workshop on Hot Topics in Operating Systems (HotOS-V)*, pages 62–65, Orcas Island, Washington, 1995.

[22] 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.

[23] Matt Bishop. *Computer Security: Art and Science*. Addison-Wesley, 2003.

[24] Nathaniel S. Borenstein. EMail with a mind of its own: The Safe-Tcl language for enabled mail. In *IFIP WG 6.5 Conference*, Barcelona, North Holland, Amsterdam, May 1994.

[25] Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *Proceedings of the 17th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'02)*, Seattle, Washington, November 2002.

[26] Chandrasekhar Boyapati and Martin Rinard. A parameterized type system for race-free Java programs. In *Proceedings of the 16th Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'01)*, Tampa Bay, Florida, October 2001.

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

[28] John Boyland, James Noble, and William Retert. Capabilities for sharing: A generalization of uniqueness and read-only. In *Proceedings of the 15th European Conference for Object-Oriented Programming*, pages 2–27, Budapest, Hungary, June 2001.

[29] 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, October 1998.

[30] Luca Cardelli. Program fragments, linking, and modularization. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 266–277, Paris, France, January 1997.

[31] 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, May 1997.

[32] CERT* Coordination Center. Gostscript vulnerability. CERT* Advisory CA-95.10, August 1995.

[33] Jeffrey S. Chase, Henry M. Levy, Michael J. Feeley, and Edward D. Lazowska. Sharing and protection in a single-address-space operating system. *ACM Transactions on Computer Systems*, 12(4):271–307, November 1994.

[34] Thomas M. Chen and Alden W. Jackson, editors. *Special Issue on Active and Programmable Networks*. IEEE Network Magazine. IEEE, July 1998.

[35] David Chess, Benjamin Grosof, Colin Harrison, David Levine, Colin Parris, and Gene Tsudik. Itinerant agents for mobile computing. Technical Report RC 20010, IBM Research Division, 1995.

[36] David Chess, Colin Harrison, and Aaron Kershenbaum. Mobile agents: Are they a good idea? In Jan Vitek and Christian Tschudin, editors, *Mobile Object Systems: Towards the Programmable Internet*, volume 1222 of *Lecture Notes in Computer Science*, pages 25–45. Springer-Verlag, 1997.

[37] David M. Chess. Security issues in mobile code systems. In *Mobile Agents and Security*, volume 1419 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.

[38] Tzi cker Chiueh, Ganesh Venkitachalam, and Prashant Pradhan. Integrating segmentation and paging protection for safe, efficient and transparent software extensions. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, pages 140–153, Charleston, South Carolina, December 1999.

[39] Joris Claessens, Bart Preneel, and Joos Vandewalle. (How) can mobile agents do secure electronic transactions on untrusted hosts? a survey of the security issues and the current solutions. *ACM Transactions on Internet Technology*, 3(1):28–48, 2003.

[40] David Clarke, John Potter, and James Noble. Ownership types for flexible alias protection. In *Proceedings of the ACM Conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA'98)*, pages 48–64, Vancouver, BC, October 1998.

[41] 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.

[42] Alessandro Coglio and Allen Goldberg. Type safety in the JVM: Some problems in Java 2 SDK 1.2 and proposed solutions. *Concurrency and Computation: Practice and Experience*, 13(13):1153–1171, November 2001.

[43] 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.

[44] Christopher Colby, Peter Lee, George C. Necula, Fred Blau, Mark Plesko, and Kenneth Cline. A certifying compiler for Java. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation (PLDI'2000)*, pages 95–107, Vancouver, British Columbia, June 2000.

[45] Patrick Cousot. Methods and logics for proving programs. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 15, pages 841–993. MIT Press, 1990.

[46] Gianpaolo Cugola, Carlo Ghezzi, Gian Pietro Picco, and Giovanni Vigna. Analyzing mobile code languages. In Jan Vitek and Christian Tschudin, editors, *Mobile Object Systems: Towards the Programmable Internet*, volume 1222 of *Lecture Notes in Computer Science*, pages 93–110. Springer-Verlag, 1997.

[47] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, 2nd 2002.

[48] Drew Dean. The security of static typing with dynamic linking. In *Proceedings of the 4th ACM Conference on Computer and Communications Security*, pages 18–27, Zurich, Switzerland, April 1997.

[49] Drew Dean, Edward W. Felten, and Dan S. Wallach. Java security: From hotjava to netscape and beyond. In *Proceedings of the 1996 Symposium on Security and Privacy*, Oakland, California, May 1996.

[50] Dorothy E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, May 1976.

[51] Dorothy E. Denning. *Cryptography and Data Security*. Addison-Wesley, 1982.

[52] Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, July 1977.

[53] P. Deutsch and C. A. Grant. A flexible measurement tool for software systems. *Information Processing*, 71:320–326, 1972.

[54] Prem Devanbu and Stuart Stubblebine. Automated software verification with trusted hardware. In *Proceedings of the Twelfth International Conference on Automated Software Engineering*, 1997.

[55] Premkumar T. Devanbu, Philip W. L. Fong, and Stuart G. Stubblebine. Techniques for trusted software engineering. In *Proceedings of the 20th International Conference on Software Engineering (ICSE'98)*, Kyoto, Japan, April 1998.

[56] Sohpia Drossopoulou. An abstract model of java dynamic linking and loading. In *Proceedings of the 3rd International Workshop on Types in Compilation*, Montreal, Canada, September 2000.

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

[58] 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.

[59] David Evans and Andrew Twyman. Flexible policy-directed code safety. In *Proceedings of the 1999 IEEE Symposium on Security and Privacy*, pages 32–47, Oakland, California, May 1999.

[60] William M. Farmer, Joshua D. Guttman, and Vipin Swarup. Security for mobile agents: Authentication and state appraisal. In *Proceedings of the Fourth*

European Symposium on Research in Computer Security (ESORICS'96), volume 1146 of *Lecture Notes in Computer Science*, pages 118–130, Rome, Italy, September 1996. Springer-Verlag.

[61] William M. Farmer, Joshua D. Guttman, and Vipin Swarup. Security for mobile agents: Issues and requirements. In *Proceedings of the 19th National Information Systems Security Conference*, pages 591–597, 1996.

[62] Roger Faulkner and Ron Gomes. The process file system and process model in UNIX System V. In *Proceedings of the USENIX Winter 1991 Conference*, pages 243–252, Dallas, Texas, January 1991.

[63] Federal Information Processing Standards Publication. Security requirements for cryptographic modules. Technical Report FIPS PUB 140-1, U.S. Department of Commerce/National Institute of Standards and Technology, January 1994. Available at `http://csrc.nist.gov/fips/fips1401.pdf`.

[64] Edward W. Felten, Dirk Balfanz, Drew Dean, and Dan S. Wallach. Web spoofing: An internet con game. In *Proceedings of the 20th National Information Systems Security Conference*, Baltimore, Maryland, October 1997.

[65] 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.

[66] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[67] M. F. Florio, R. Gorrieri, and G. Marchetti. Coping with denial of service due to malicious Java applets. *Computer Communications*, 23(17):1645–1654, November 2000.

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

[69] Philip W. L. Fong. Access control by tracking shallow execution history. Technical Report CS-2003-9, Department of Computer Science, University of Regina, Regina, Saskatchewan, Canada S4S 0A2, November 2003.

[70] Philip W. L. Fong. Pluggable verification modules: An extensible protection mechanism for the JVM. Technical Report CS 2003-11, Department of Computer Science, University of Regina, Regina, Saskatchewan, Canada S4S 0A2, November 2003.

[71] Philip W. L. Fong and Robert D. Cameron. Proof linking: An architecture for modular verification of dynamically-linked mobile code. In *Proceedings of the Sixth ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE'98)*, Orlando, Florida, November 1998.

[72] Philip W. L. Fong and Robert D. Cameron. Java proof linking with multiple classloaders. Technical Report SFU CMPT TR 2000-04, School of Computing Science, Simon Fraser University, Burnaby, BC, Canada V5A 1S6, August 2000.

[73] 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.

[74] 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)*, Monterey, California, April 2001.

[75] 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.

[76] 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.

[77] Michael Franz. *Code-Generation On-the-Fly: A Key to Portable Software*. Doctoral dissertation no. 10497, ETH Zürich, 1994.

[78] Timothy Fraser. LOMAC: Low water-mark integrity protection for COTS environments. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy*, Oakland, California, 2000.

[79] Timothy Fraser, Lee Badger, and Mark Feldman. Hardening COTS software with generic software wrappers. In *Proceedings of the 1999 IEEE Symposium on Security and Privacy*, pages 2–16, Oakland, California, May 1999.

[80] 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.

[81] J. Goguen and J. Meseguer. Security policies and security models. In *Proceedings of the 1982 IEEE Symposium on Security and Privacy*, pages 11–20, 1982.

[82] 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.

[83] Ian Goldberg, David Wagner, Randi Thomas, and Eric A. Brewer. A secure environment for untrusted helper applications. In *Proceedings of the 6th USENIX Security Symposium*, San Jose, California, July 1996.

[84] Li Gong. Java security: Present and near future. *IEEE Micro*, 17(3):14–19, May 1997.

[85] Li Gong. New security architectural directions for Java. In *Proceedings of 42nd IEEE International Computer Conference (COMPCON'97)*, pages 97–102, San Jose, California, February 1997.

[86] Li Gong, Gary Ellison, and Mary Dageforde. *Inside Java 2 Platform Security: Architecture, API Design, and Implementation.* Addison-Wesley, 2nd edition, 2003.

[87] Li Gong, Marianne Mueller, Hemma Prafullchandra, and Roland Schemers. Going beyond the sandbox: An overview of the new security architecture in the Java Development Kit 1.2. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, pages 103–112, Monterey, California, December 1997.

[88] Li Gong and Roland Schemers. Implementing protection domains in the Java Development Kit 1.2. In *Proceedings of the 1998 Internet Society Symposium on Network and Distributed System Security (NDSS'98)*, San Diego, California, March 1998.

[89] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification.* Addison-Wesley, 2nd edition, 2000.

[90] Robert S. Gray. Agent Tcl: A flexible and secure mobile agent system. In *Proceedings of the 4th Annual USENIX Tcl/Tk Workshop*, pages 9–23, Monterey, California, July 1996.

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

[92] David R. Hanson. *C Interfaces and Implementations: Techniques for Creating Reusable Software*. Addison-Wesley, 1996.

[93] Pieter H. Hartel and Luc Moreau. Formalizing the safety of Java, the Java virtual machine, and Java card. *ACM Computing Surveys*, 33(4):517–558, 2001.

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

[95] Minghua He and Nicholas R. Jennings. SouthamptonTAC: An adaptive autonomous trading agent. *ACM Transactions on Internet Technology*, 3(3):218–235, August 2003.

[96] Gernot Heiser, Kevin Elphinstone, Jerry Vochteloo, Stephen Russell, and Jochen Liedtke. The Mungi single-address-space operating system. *Software—Practice and Experience*, 28(9):901–928, July 1998.

[97] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 3rd edition, 2003.

[98] Michael Hicks, Pankaj Kakkar, Jonathan T. Moore, Carl A. Gunter, and Scott Nettles. PLAN: A programming language for active networks. In *Proceedings of the 3rd ACM SIGPLAN International Conference on Functional Programming Languages (ICFP'98)*, pages 86–93, Baltimore, Maryland, September 1998.

[99] Michael Hicks, Jonathan T. Moore, and Scott Nettles. Compiling PLAN to SNAP. In *IFIP-TC6 3rd International Working Conference On Active Networks (IWAN'01)*, volume 2207 of *Lecture Notes in Computer Science*, Philadelphia, Pennsylvania, October 2001. Springer-Verlag.

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

[101] IBM. IBM PCI cryptographic coprocessor. Available at `http://www-3.ibm.com/security/cryptocards`.

[102] Samuel P. Harbison III and Guy L. Steele Jr. *C: A Reference Manual*. Prentice Hall, 5th edition, 2002.

[103] Sotiris Ioannidis, Steven M. Bellovin, and Jonathan M. Smith. Sub-operating systems: A new approach to application security. In *Proceedings of the 10th ACM SIGOPS European Workshop*, Saint-Emilion, France, September 2002.

[104] Trent Jaeger, Atul Prakash, Jochen Liedtke, and Nayeem Islam. Flexible control of downloaded executable content. *ACM Transactions on Information and System Security*, 2(2):177–228, May 1999.

[105] W. A. Jansen. Countermeasures for mobile agent security. *Computer Communications*, 23(17):1667–1676, November 2000.

[106] Wayne Jansen and Tom Karygiannis. Mobile agent security. Special Publication 800-19, National Institute of Standards and Technology, August 1999.

[107] JavaSoft. Chronology of security-related bugs and issues. `http://java.sun.com/sfaq/chronology.html`.

[108] JavaSoft. More details on the university of washington kimera research project. `http://java.sun.com/security/UWdetails.html`.

[109] T. Jensen, D. Le Métayer, and T. Thorn. Security and dynamic class loading in java: A formalisation. In *Proceedings of the 1998 IEEE International Conference on Computer Languages*, pages 4–15, Chicago, May 1998.

[110] M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger, Héctor M. Brice no, Russell Hunt, David Mazières, Thomas Pinckney, Robert Grimm, John Jannotti, and Kenneth Mackenzie. Application performance and flexibility on Exokernel systems. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 52–65, Saint Malo, France, October 1997.

[111] John B. Kam and Jeffrey D. Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7:305–317, 1977.

[112] Kazuhiko Kato. Safe and secure execution mechanisms for mobile objects. In Jan Vitek and Christian Tschudin, editors, *Mobile Object Systems: Towards the Programmable Internet*, volume 1222 of *Lecture Notes in Computer Science*, pages 201–211. Springer-Verlag, 1997.

[113] Charlie Kaufman, Radia perlman, and Mike Speciner. *Network Security: Private Communication in a Public World*. Pearson Education, 2nd edition, 2002.

[114] Kimberly Keeton, David A. Patterson, and Joseph M. Hellerstein. A case for intelligent disks (IDISKs). *ACM SIGMOD Record*, 27(3):42–52, September 1998.

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

[116] 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.

[117] 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 of *LNCS*, pages 327–353, Budapest, Hungary, June 2001.

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

[119] Gerwin Klein and Toblias Nipkow. Verified lightweight bytecode verification. In *ECOOP 2000 Workshop on Formal Techniques for Java Programs*, 2000.

[120] Frederick C. Knabe. *Language Support for Mobile Agents*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213-3891, USA, December 1995.

[121] Günter Kniesel. Encapsulation = visibility + accessibility. Technical Report IAI-TR-96-12, Computer Science Department III, University of Bonn, November 1996.

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

[123] Mark D. LaDue. A collection of increasingly hostile applets. `http://www.cigital.com/hostile-applets`.

[124] Leonard J. LaPadula and D. Elliot Bell. Secure computer systems: A mathematical model. Technical Report 2547 (Volume II), MITRE, May 1973.

[125] Doug Lea. *Concurrent Programming in Java: Design Principles and Patterns*. Addison-Wesley, 1997.

[126] Stanley Letovsky and Elliot Soloway. Delocalized Plans and Program Comprehension. *IEEE Software*, pages 41–49, May 1986.

[127] 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.

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

[129] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison Wesley, 1st edition, 1997.

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

[131] John Wylie Lloyd. *Foundations of Logic Programming*. Symbolic computation: Artificial intelligence. Springer-Verlag, 2nd, extended edition edition, 1987.

[132] R. L. London, J. V. Guttag, J. J. Horning, B. W. Lampson, J. G. Mitchell, and G. J. Popek. Proof rules for the programming language Euclid. *Acta Informatica*, 10:1–26, 1978.

[133] Steve Lucco, Oliver Sharp, and Robert Wahbe. Omniware: A universal substrate for web programming. In *Proceedings of the 4th International World Wide Web Conference*, Boston, Massachusetts, December 1995.

[134] Pattie Maes, Robert H. Guttman, and Alexandros G. Moukas. Agents that buy and sell. *Communications of the ACM*, 42(3):81–91, March 1999.

[135] Gary McGraw and Edward W. Felten. *Java Security: Hostile Applets, Holes, and Antidotes*. John Wiley & Sons, Inc, 1997.

[136] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996.

[137] Microsoft. ActiveX controls. `http://www.microsoft.com/com/tech/ActiveX.asp`.

[138] Naftaly Minsky. Towards alias-free pointers. In *Proceedings of the 10th European Conference on Objected Oriented Programming (ECOOP'96)*, volume 1098 of *Lecture Notes in Computer Science*, pages 189–209, Linz, Austria, July 1996. Springer.

[139] Jonathan Moore, Michael Hicks, and Scott Nettles. Practical programmable packets. In *Proceedings of the 20th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM'2001)*, pages 41–50, Anchorage, Alaska, April 2001.

[140] Greg Morrisett, Karl Crary, Neal Glew, and David Walker. Stack-based typed assembly language. In *Workshop on Types in Compilation*, pages 95–118, Kyoto, Japan, March 1998.

[141] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From system F to typed assembly language. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'98)*, pages 85–97, San Diego, CA., January 1998.

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

[143] 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.

[144] 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.

[145] George C. Necula and Peter Lee. The design and implementation of a certifying compiler. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation (PLDI'98)*, pages 333–344, Montreal, Quebec, November 1998.

[146] George C. Necula and Peter Lee. Safe, untrusted agents using proof-carrying code. In *Mobile Agent Security*, volume 1419 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.

[147] Netscape Communications. Netscape devedge: Javascript central. `http://devedge.netscape.com/central/javascript`.

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

[149] Tobias Nipkow. Special issue on Java bytecode verification. *Journal of Automated Reasoning*, 30(3–4), 2003.

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

[151] 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.

[152] John K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, 1994.

[153] John K. Ousterhout, Jacob Y. Levy, and Brent B. Welch. The Safe-Tcl security model. In *Mobile Agents and Security*, volume 1419 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.

[154] Flavio De Paoli, Andre L. Dos Santos, and Richard A. Kemmerer. Web browsers and security. In *Mobile Agents and Security*, volume 1419 of *Lecture Notes in Computer Science*. Springer-Verlag, January 1998.

[155] Andy Podgurski and Lori A. Clarke. A formal model of program dependences and its implications for software testing, debugging, and maintenance. *IEEE Transactions on Software Engineering*, 16(9):965–979, September 1990.

[156] G. J. Popek, J. J. Horning, B. W. Lampson, J. G. Mitchell, and R. L. London. Notes on the design of Euclid. In *Proceedings of an ACM Conference on Language Design for Reliable Software*, pages 11–18, Raleigh, North Carolina, March 1977.

[157] Princeton Secure Internet Programming Group. Secure internet programming: News. `http://www.cs.princeton.edu/sip/history/`.

[158] Princeton Secure Internet Programming Team. Security tradeoffs: Java vs. ActiveX. `http://www.cs.princeton.edu/sip/faq/java-vs-activex.html`.

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

[160] 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.

[161] Christian Queinnec and David De Roure. Sharing code through first-class environments. In *Proceedings of the 1st ACM SIGPLAN International Conference on Functional Programming*, pages 251–261, Philadelphia, Pennsylvania, May 1996.

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

[163] Erik Riedel, Christos Faloutsos, Garth A. Gibson, and David Nagle. Active disks for large-scale data processing. *IEEE Computer*, 34(6):68–74, 2001.

[164] Stuart Ritchie. Systems programming in java. *IEEE Micro*, pages 30–35, May 1997.

[165] Eva Rose and Kristoffer Hogsbro Rose. Lightweight bytecode verification. In *The OOPSLA'98 Workshop on Formal Underpinnings of Java*, Vancouver, BC, Canada, November 1998.

[166] François Rouaix. A web navigator with applets in Caml. In *Proceedings of the 5th International World Wide Web Conference*, pages 1365–1371, Paris, France, May 1996.

[167] Algis Rudys and Dan S. Wallach. Termination in language-based systems. *ACM Transactions on Information and System Security*, 5(2):138–168, May 2002.

[168] Algis Rudys and Dan S. Wallach. Transactional rollback for language-based systems. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN'02)*, pages 439–448, Washington, D.C., June 2002.

[169] Spencer Rugaber, Kurt Stirewalt, and Linda M. Wills. Understanding interleaved code. *Automated Software Engineering*, 3(1–2):47–76, June 1996.

[170] John Rushby and Brian Randell. A distributed secure system. *IEEE Computer*, pages 55–67, July 1983.

[171] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(5), January 2003.

[172] Pierangela Samarati and Sabrina de Capitani di Vimercati. Access control: Policies, models, and mechanisms. In *Foundations of Security Analysis and Design: Tutorial Lectures*, volume 2171 of *Lecture Notes in Computer Science*. Springer-Verlag, January 2001.

[173] Tuomas Sandholm and Qianbo Huai. Nomad: Mobile agent system for an internet based auction house. *IEEE Internet Computing*, 4(2):80–86, 2000.

[174] Vijay Saraswat. Java is not type-safe. `http://matrix.research.att.com/vj/bug.html`, 1997.

[175] Yoshiki Sato, Shigeru Chiba, and Michiaki Tatsubori. A selective, just-in-time aspect weaver. In *Proceedings of the 2nd International Conference on Generative Programming and Component Engineering (GPCE'03)*, volume 2830 of *Lecture Notes in Computer Science*, pages 189–208, Erfurt, Germany, September 2003. Springer-Verlag.

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

[177] Bruce Schneier. *Applied Cryptography: Protocols, Algorithms, and Source Code in C*. John Wiley, 1996.

[178] Margo I. Seltzer, Yasuhiro Endo, Christopher Small, and Keith A. Smith. Dealing with disaster: surviving misbehaved kernel extensions. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation*, Seattle, Washington, October 1996.

[179] 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.

[180] Abraham Silberschatz and Peter B. Galvin. *Operating System Concepts*. Addison-Wesley, 4th edition, 1994.

[181] Emin Gun Sirer, Marc Fiuczynski, Przemyslaw Pardyak, and Brian Bershad. Safe dynamic linking in an extensible operating system. In *Proceedings of the 1st Annual Workshop on Compiler Support for System Software*, February 1996.

[182] Mats Skoglund and Tobias Wrigstad. A mode system for read-only references in Java. In *3rd Workshop on Formal Techniques for Java Programs*, Budapest, Hungary, June 2001.

[183] Christopher Small and Margo Seltzer. A comparison of OS extension technologies. In *Proceedings of the 1996 USENIX Conference*, 1996.

[184] Geoffrey Smith. A new type system for secure information flow. In *Proceedings of the 14th IEEE Computer Security Foundations*, pages 115–125, Cape Breton, Nova Scotia, June 2001.

[185] Geoffrey Smith. Weak probabilistic bisimulation for secure information flow. In *Proceedings of the Workshop on Issues in the Theory of Security (WITS'02)*, Portland, Oregon, January 2002.

[186] Geoffrey Smith and Dennis Volpano. Secure information flow in a multi-threaded imperative language. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'98)*, pages 355–364, San Diego, California, January 1998.

[187] Geoffrey Smith and Dennis Volpano. Confinement properties for multi-threaded programs. In Stephen Brookes, Achim Jung, Michael Mislove, and Andre Scedrov, editors, *Proceedings of the Fifteenth Conference on the Mathematical Foundations of Programming Semantics*, volume 20 of *Electronic Notes in Theoretical Computer Science*, New Orleans, LA, April 1999. Elsevier Science Publishers.

[188] SRI Computer Science Laboratory. The PVS specification and verification system. Available at `http://pvs.csl.sri.com/`.

[189] James W. Stamos and David K. Gifford. Remote evaluation. *ACM Transactions on Programming Languages and Systems*, 12(4):537–565, October 1990.

[190] R. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine — Definition, Verification, Validation*. Springer, 2001.

[191] 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.

[192] Sun Microsystems. *JAR File Specification*. `http://java.sun.com/j2se/1.4.1/docs/guide/jar/jar.html`.

[193] Sun Microsystems. *Connected, Limited Device Configuration: Java 2 Platform Micro Edition*. Sun Microsystems, May 2000. Available at `http://java.sun.com/products/cldc/`.

[194] Sun Microsystems. *Connected Device Configuration: Java 2 Platform Micro Edition*. Sun Microsystems, March 2001. Available at `http://java.sun.com/products/cdc/`.

[195] Michael M. Swift, Steven Martin, Henry M. Levy, and Susan J. Eggers. Nooks: An architecture for reliable device drivers. In *Proceedings of the 10th ACM SIGOPS European Workshop*, Saint-Emilion, France, September 2002.

[196] Ed Taft and Jeff Walden. *PostScript Language Reference Manual*. Addison-Wesley, 2nd edition, 1990.

[197] Joseph Tardo and Luis Valente. Mobile agent security and Telescript. In *Proceedings of the 41st IEEE International Computer Conference (COMPCON'96)*, pages 58–63, San Jose, California, February 1996.

[198] David L. Tennenhouse, Jonathan M. Smith, W. David Sincoskie, David J. Wetherall, and Gary Minden. A survey of active network research. *IEEE Communications Magazine*, 35(1):80–86, January 1997.

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

[200] Akihiko Tozawa and Masami Hagiya. Formalization and analysis of class loading in Java. *Higher-Order and Symbolic Computation*, 15:7–55, 2002.

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

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

[203] 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.

[204] Giovanni Vigna, editor. *Mobile Agents and Security*, volume 1419 of *Lecture Notes in Computer Science*. Springer-Verlag, January 1998.

[205] Mahesh Viswanathan. *Foundations for the Run-time Analysis of Software systems*. PhD thesis, University of Pennsylvania, December 2000.

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

[207] Dennis Volpano. Secure introduction of one-way functions. In *Proceedings of the 13th IEEE Computer Security Foundations Workshop*, pages 246–254, Cambridge, UK, July 2000.

[208] Dennis Volpano and Geoffrey Smith. Eliminating covert flows with minimum typings. In *Proceedings of the Tenth IEEE Computer Security Foundations Workshop*, pages 156–168, Rockport, Massachusetts, June 1997.

[209] Dennis Volpano and Geoffrey Smith. A type-based approach to program security. In *Proceedings of the Seventh International Joint Conference on the Theory and Practice of Software Development*, volume 1214 of *Lecture Notes in Computer Science*, pages 607–621, Lille, France, April 1997. Springer.

[210] Dennis Volpano and Geoffrey Smith. Language issues in mobile program security. In *Mobile Agents and Security*, volume 1419 of *Lecture Notes in Computer Science*, pages 25–43. Springer-Verlag, 1998.

[211] Dennis Volpano and Geoffrey Smith. Probablistic noninterference in a concurrent language. *Journal of Computer Security*, 7(2,3):231–253, November 1999.

[212] Dennis Volpano and Geoffrey Smith. Verifying secrets and relative secrecy. In *Proceedings of the 27th ACM Symposium on Principles of Programming Languages (POPL'2000)*, pages 268–276, Boston, Massachusetts, January 2000.

[213] Dennis Volpano, Geoffrey Smith, and Cynthia Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–187, December 1996.

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

[215] Dan S. Wallach, Andrew W. Appel, and Edward W. Felten. SAFKASI: A security mechanism for language-based systems. *ACM Transactions on Software Engineering and Methodology*, 9(4):341–378, October 2000.

[216] Dan S. Wallach, Dirk Balfanz, Drew Dean, and Edward W. Felten. Extensible security architectures for Java. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 116–128, Saint Malo, France, October 1997.

[217] Dan S. Wallach and Edward W. Felten. Understanding Java stack inspection. In *Proceedings of 1998 IEEE Symposium on Security and Privacy*, Oakland, California, May 1998.

[218] WAP Forum. *WAP-193-WMLScript Language Specification*. WAP Forum, version 1.2 edition, June 2000. Available at `http://www.wapforum.org/what/technical.htm`.

[219] Ian Welch and Robert J. Stroud. Kava - using byte code rewriting to add behavioural reflection to Java. In *Proceedings of the 6th USENIX Conference on Object-Oriented Technologies and Systems*, San Antonio, Texas, January 2001.

[220] Ian Welch and Robert J. Stroud. Using reflection as a mechanism for enforcing security policies on compiled code. *Journal of Computer Security*, 10(4):399–432, 2002.

[221] James E. White. Mobile agents. In Jeffrey Bradshaw, editor, *Software Agents*, chapter 19, pages 437–472. AAAI Press/MIT Press, 1996.

[222] Steve Wilson and Jeff Kesselman. *Java Platform Performance: Strategies and Tactics*. Addison-Wesley, 2000.

[223] David B. Wortman. On legality assertions in Euclid. *IEEE Transactions on Software Engineering*, 5(4):359–367, 1979.

[224] David B. Wortman and James R. Cordy. Early experiences with Euclid. In *Proceedings of the 5th International Conference on Software Engineering*, pages 27–32, San Diego, Carlifornia, March 1981.

[225] Frank Yellin. Low level security in Java. In *Proceedings of the 4th International World Wide Web Conference*, Boston, Massachusetts, December 1995.

[226] Xun Yi and Chee Kheong Siew. Secure agent-mediated online auction framework. *International Journal of Information Technology*, 7(1), April 2001.