Proof Linking: Progress Report and Research Proposal\textsuperscript{1}

Philip W. L. Fong  
pwfong@cs.sfu.ca

SFU CMPT TR 2002-12  
School of Computing Science  
Simon Fraser University,  
Burnaby, B.C., Canada V5A 1S6  
ftp://fas.sfu.ca/pub/cs/TR/2002/

December, 2002

\textsuperscript{1}This is a revised edition of the author’s PhD thesis proposal, as approved by his supervisory committee on December 30, 2002.
Abstract

This report presents a critical rethinking of the Java security 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. Progress to date is reported, and proposals for additional work to evaluate the benefits and feasibility of the Proof Linking architecture are detailed.
Contents

1 Introduction .................................................. 3
1.1 Motivation .................................................. 3
    1.1.1 Stand-alone Verification Module ................... 4
1.1.2 Distributed Verification .............................. 5
1.1.3 Augmenting Type Systems .......................... 8
1.2 Thesis Statement .......................................... 9
1.3 Overview of the Report ................................. 10

2 The Proof Linking Architecture ............................. 11
2.1 Architectural Overview .................................... 11
    2.1.1 Modular Verification ............................... 12
    2.1.2 Proof Linking ....................................... 13
    2.1.3 Remarks ............................................ 14
2.2 Correctness of Incremental Proof Linking ............... 15
    2.2.1 A Model for Lazy, Dynamic Linking ............. 15
    2.2.2 A Model Proof Linking Algorithm ................ 17
    2.2.3 Formalization of Correctness Conditions ... 19
2.3 Potential Benefits of Proof Linking .................... 20
    2.3.1 Stand-alone Verification Modules ............... 20
    2.3.2 Distributed Verification .......................... 21
    2.3.3 Augmenting Type Systems ........................ 21

3 Progress To Date and Proposed Work ....................... 22
3.1 Modeling Adequacy and Soundness ....................... 22
    3.1.1 Lazy, Dynamic Linking ............................ 22
    3.1.2 Multiple Classloaders ............................. 23
    3.1.3 Summary ........................................... 23
3.2 Benefits of Proof Linking ................................ 23
    3.2.1 The Aegis VM ...................................... 24
    3.2.2 Stand-alone Verification Modules ............... 24
    3.2.3 Distributed Verification .......................... 25
    3.2.4 Augmenting Type Systems ........................ 26
    3.2.5 Summary ........................................... 28

A Timeline .................................................. 29
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 [12]. A key factor in this growth has been the development of suitable security models for the protection of host computer systems against the potential dangers 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. an httpd process) 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 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 [28] and its associated support technologies have achieved considerable success through a strong security model implemented within the Java Virtual Machine (JVM). As Java bytecode is downloaded from an untrusted origin, the JVM subjects it to a verification step [39, Chapter 5] in order to ensure that it cannot affect the host machine in an undesirable way. The bytecode verifier performs dataflow analysis and various 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 [15]. As 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 needs 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 flaw to bypass security checks. In fact, several security breaches have been discovered in major Java implementations [33, 47, 32]. These flaws 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 become manifest in two problematic architectural features of Sun’s JVM:

1. **Interleaved logic.** Sun’s implementation 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 \( X \) to proceed. For example, in order for the verifier to make sure that a method may throw an “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 subclasses. Moreover, since the loader cannot trust the bytecode of “ArithmeticException” (and its subclasses) 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 Sun’s JVM.

2. **Delocalized implementation.** Sun’s 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 link time, 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 [49, 38]. This so-called “scattershot security” [41] 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 security system based on link-time verification, 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. 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 [41, p. 110]. Compounding these concerns, the architectural complexity of link-time bytecode verification also adds significantly to the JVM’s *memory footprint* [59, 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 (see, for example, the mobile code language WMLScript [70] for the Wireless Application Protocol). With its stability and widespread acceptance, the Java platform — and specifically realizations thereof based on the Connected, Limited Device Configuration (CLDC) specification [59] and the Connected Device Configuration (CDC) specification [60] — will likely become a major infrastructure for hosting mobile programs in small devices. 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 interoperation 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 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 machine 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 protocol have been proposed in the related literature.

1. Self-Certifying Code. The first protocol family involves augmenting the untrusted code to make it self-certifying. This approach is exemplified in the work on proof-carrying code [44, 43]. 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 [48, 34, 14], and has further been incorporated in the stack map method of the CLDC specification [59, Sec. 5.3].

A related approach, as found in the work on SafeTSA [2], is to give up using the difficult-to-verify Java bytecode representation as a transport media for mobile programs. As a competitive alternative, the SSA-based SafeTSA representation explicitly captures the dataflow structure of the underlying mobile program, making verification very efficient.

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 kind of annotation), 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 [19, 20]. The protocol proceeds as follows. (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 [31]. 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 [22]. 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\(^1\).

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

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, during the verification of a classfile, the dataflow analyzer might need to show that class X is a subclass of class Y. This fact can only be shown by examining the classes that are actually loaded into the consumer's JVM. As a result, 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 defeating the very distinctiveness 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.

\(^{1}\text{When combined with a public key management infrastructure, signature-based verification protocols also enable a very flexible configuration management solution, in which software releases known to be flawed can be disabled remotely [19, 20].}\)
A JVM hosting this program will not only need to be fluent in both protocols, but also need to combine 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 class file format [39, Chapter 4] nor the Java Archive (JAR) [58] 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 Augmenting 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 safety properties are usually formulated as augmenting type systems on top of the base type system of a mobile code language. While the literature about such augmenting 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 sensitivity 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 [4, 37]. 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 [18, 16, 17] first applied this idea to the control of information flow in high level programming languages through static analysis. Subsequent developments have been constantly reported [50], among which the work of Volpano and Smith [68, 64, 63, 55, 65, 56, 66, 67, 62, 53, 54] has recently attracted considerable attention from the mobile code community. They defined an augmenting type system on a prototypical high level imperative programming language, so that programmers may decorate a variable by a discrete sensitivity level. They have proven a form of noninterference property [27], 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-effect is difficult in object-oriented systems in which writable aliases could be created in an unrestrained manner [29]. Augmenting
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 either by using unique types [3, 42, 10] or placing constraints on the connectivity of the object graph [29, 1, 13].
- **Access control:** Side effects resulted from aliases is controlled either by tagging aliases to be read-only [29, 36, 51] or imposing other forms of access control [45].

As these type systems are effectively access control constraints, they could be applied to enforce safety policies [61, 9, 8].

One critique of the researches mentioned above is that the notion of type safety is almost always formulated as a compile-time property, enforced by the code producer 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 augmenting 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 gets implemented in a realistic mobile code system, a phenomenon that partly motivates the proposal of the EVM [7]. 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

This research 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 augmenting type systems. By design of this architecture, the verification logic of the run-time environment is localized in a stand-alone module completely 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:** The Proof Linking architecture is theoretically sound and is adequate for modeling the complexity of Java’s dynamic linking semantics.

**TS2:** The Proof Linking architecture delivers the following benefits to the Java run-time environment:

1. It brings modularity to the Java run-time environment, thereby allowing the byte-code verification component to be crafted as a *stand-alone module*.
2. It enables the Java platform to support *interoperability of distributed verification protocols and conditional certification*.
3. It offers extensibility to the link-time verification procedure, thereby allowing *augmenting type systems* to be defined conveniently on top of the dynamic linking process of a JVM.

### 1.3 Overview of the Report

This report is structured as follows. Chapter 2 is an overview of the Proof Linking architecture, its formal modeling, and its potential benefits. Chapter 3 describes what I have achieved so far in establishing the theses outlined in Section 1.2, and presents further works I propose to work on. Appendix A gives a timeline of my execution plan.
Chapter 2

The Proof Linking Architecture

This chapter contains 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 augmenting type systems is also discussed.

2.1 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 loading, verification, and linking are performed by three separate modules. No module attempts to invoke any other during its processing, nor will one recursively invoke itself. This setup poses the following challenge:

\[
\text{Verification requires knowledge of other code units which might not have been loaded yet. How does one remove such dependencies while maintaining the integrity of the verification process?}
\]

For illustrative purpose, consider the example as depicted in Figure 2.1. Suppose 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 \( C \) overrides the method \( M(S) \). Say the method \( C :: M(S) \) contains an invokespecial bytecode instruction that delegates the call to method \( A :: M(S) \). When the 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. How can modularity be achieved without
overlooking such external dependencies? The problem is addressed by decomposition of verification into two subtasks: *modular verification* and *proof linking*.\(^1\)

### 2.1.1 Modular Verification

Figure 2.2 depicts the setup for modular verification in the context of Java classloading. 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 recursively verifying (or even loading) the other code unit, the verifier computes a conservative *safety precondition* that will guarantee the safety of the code unit. The safety precondition is represented as a conjunctive set of database queries. In the running example (Figure 2.1), the classfile is safe only if class \( C \) is a subclass of the class \( A \), the modular verifier therefore formulates the query\(^2\) \(?\text{subclass}(\text{`C'}, \ 'A')\). 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 classfile being considered. More specifically, each of the queries describes a safety precondition of a certain linking action. For example, the query \(?\text{subclass}(\text{`C'}, \ 'A')\) is a safety precondition for the action “resolving symbol \( A :: M(S) \) in class \( C \)”. Such queries are said to be the *proof obligations* for the associated actions, representing conditions that must be met if the run-time system attempts to safely perform the corresponding actions in the future. A proof obligation is said to be *attached* to its associated action. Obligations generated by the verifier are collected by the *proof linker*, which records in a global *obligation table* the mapping from linking actions 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

\(^1\)Capitalization is used to differentiate between “Proof Linking” as a verification architecture and “proof linking” as a verification subtask

\(^2\)To differentiate the various roles played by a predicate symbol, a query is typographically prefixed by a question mark (“?”), and an assertion by an exclamation mark (“!”).
that describe the interface properties of the code unit. For example, during the verification of the Java classfile $C$, the verifier generates a commitment $!\text{extends}('C', 'B$) to indicate that $B$ is the immediate superclass of $C$. The generated commitments are collected by the proof linker, and subsequently asserted into a global commitment database. When proof obligations are to be checked, the commitment database provides the set of facts against which the query can be evaluated.

### 2.1.2 Proof Linking

The process by which the run-time system cross-validates the results of verifying different code units is called proof linking. Figure 2.3 depicts the setup for proof linking. When the run-time system needs to resolve a symbolic reference to a machine pointer, it sends the request to the proof linker. The proof linker looks up the obligations that have been attached to the request, and then posts them to the commitment database as deductive queries. If the queries are satisfied, the requested action is performed. Otherwise, a linking exception is raised to signal failure to endorse the consistency of the code units.

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 form can be supplied to capture the transitive closure of the subclassing relationship:

$$
\text{subclass}(X, X).
\text{subclass}(X, Y) :- \text{extends}(X, Z), \text{subclass}(Z, Y).
$$

If the verifier asserts commitments

$$!\text{extends}('C', 'B')$$
and

\[ !\text{extends}('B', 'A') \]

then the obligation

\[ ?\text{subclass('C', 'A')} \]

can be satisfied.

### 2.1.3 Remarks

Although a deductive database model has been used as a means of representing 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 \textit{extends} relationship can be represented in a space-efficient manner using an appropriate graph data structure, while the logic of the \textit{subclass} relationship (transitive closure) may be implemented using a graph traversal algorithm.

There are however two reasons to model proof linking as a series of database updates and queries. First of all, the database model provides an abstract framework to describe the general notion of proof linking, without getting into the idiosyncrasies of individual mobile code systems. Secondly, and more importantly, it allows us to define a formal model of proof linking and its correctness conditions.
2.2 Correctness of Incremental Proof Linking

To assess the adequacy and soundness of the Proof Linking architecture in a complex dynamic linking environment such as the JVM, one needs a theoretical model upon which the semantics and correctness of the architecture can be articulated. Such a formal model of 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 code unit are generated and checked before that unit is executed.

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

3. **Completion**: All commitments that may be used 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 obligations and the enabling of code unit execution 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 2.2.1 models dynamic linking processes as partially-ordered sequences of primitive linking actions. Section 2.2.2 presents a simple proof linking algorithm as an operational definition of commitment and obligation processing. Section 2.2.3 then goes on to formalize the safety, monotonicity and completion conditions in terms of the terminology developed in the previous two subsections.

2.2.1 A Model for Lazy, Dynamic Linking

The fundamental simplification achieved by the Proof Linking architecture is that loading, verification and linking may be decomposed into independent primitive actions. That is, although concurrent execution of the primitive actions is allowed, each step of loading or verifying a particular code unit, or resolving a particular external reference must be a self-contained action independent of any other. Thus, actions are modeled by atomic *linking primitives*, each of which can be executed at most once during the life time of the run-time environment. Although the precise set of primitives that are used in a particular system may vary, it is assumed that the following minimal set exists 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, “\(\text{begin } p\)” and “\(\text{end } p\)”, 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, “\(\text{begin } p\)” will be generated. Similarly, the run-time system generates “\(\text{end } p\)” 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. It is further assumed that, event “\(\text{end } p\)” can occur in an execution trace only if there is a corresponding event “\(\text{begin } p\)” occurring strictly before it.

Given a set \(P\) of linking primitives, a linking strategy \(\sigma = (P, <)\) is a strict partial ordering of the linking primitives in \(P\). Every implementation of a mobile code run-time system defines a linking strategy. The strategy expresses the order in which linking events may be processed by the run-time system. More precisely, an execution trace \(\tau\) is \(\sigma\)-conforming if the following hold: (1) all linking events in \(\tau\) initiate or terminate primitives from \(P\), and (2) for all \(p, q \in P\) such that \(p <_\sigma q\), if “\(\text{begin } q\)” occurs in \(\tau\) then, “\(\text{end } p\)” occurs in \(\tau\) before “\(\text{begin } q\)”. To say that a run-time system implements a linking strategy \(\sigma\) is to say that the run-time system guarantees that all possible execution traces are \(\sigma\)-conforming. Notice that this definition of linking strategy allows primitives to be executed concurrently as long as the strategy does not explicitly order them.

A strategy is admissible if the following properties hold: given any code units \(X\) and \(Y\), and a symbol \(S\) imported by \(X\) from \(Y\), the following holds:

1. **Natural Progression Property:**

   \[
   \text{load } X < \text{verify } X < \text{resolve } S \text{ in } X
   \]

2. **Import-Checked Property:**

   \[
   \text{verify } Y < \text{resolve } S \text{ in } X < \text{use } S \text{ in } X
   \]

Only admissible strategies are considered hereafter.

For example, consider the minimal strategy imposing only the Natural Progression and Import-Checked Properties as ordering constraints. Assuming that code unit \(X\) imports symbol \(S\) from code unit \(Y\), the following execution trace conforms to the strategy:
(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 X"
(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:

```
(1) load X  (5) verify X  (6) verify X  (8) resolve S in X  (10)
```

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:

```
(1) load X  (5) verify X  (6) verify X  (8) resolve S in X  (10)
```

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**.

### 2.2.2 A Model Proof Linking Algorithm

The proof linking process is modeled operationally as an algorithm handling asynchronously generated linking events. Generalizing the notions of commitments and obligations, every linking primitive may generate both commitments and obligations. Commitments are facts describing the information collected as a result of executing a primitive. Obligations are queries that are attached as safety preconditions to subsequent primitives, called *targets*. An obligation-target pair is called an *attachment*. Notice that obligations can be attached to targets other than the *resolve* primitive.

Figure 2.4 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 an initial theory (**Initial-Theory**), and the obligation table is empty (line 1). The proof linker consumes linking events in the order specified by the
algorithm ProofLinker(Initial-Theory):

01: DB ← Initial-Theory; Obligations[ ] ← ∅;
02: Ready ← ∅; Satisfied ← ∅; Failed ← ∅;
03: while (¬ run-time-env-terminated()) do
04:   e ← get-next-event();
05:   switch e of
06:   case “begin p”:
07:      All-Obligations-Satisfied ← true;
08:      for all o ∈ Obligations[p] do
09:         if (DB ⊢ o) then
10:            Satisfied ← Satisfied ∪ { o };
11:         else
12:            Failed ← Failed ∪ { o };
13:      All-Obligations-Satisfied ← false;
14:     end if
15:   end for
16:   if (All-Obligations-Satisfied) then
17:      Ready ← Ready ∪ { p };
18:      authorize the execution of p;
19:   else
20:      deny request to execute p;
21:   endif
22:   case “end p”:
23:      DB ← DB ∪ get-commitments(p);
24:      for all ⟨o, t⟩ ∈ get-attachments(p) do
25:         Obligations[t] ← Obligations[t] ∪ {o};
26:     end for
27:   end switch
28: endwhile

Figure 2.4: The Proof-Linker Model Algorithm
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). 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).

### 2.2.3 Formalization of Correctness Conditions

To formalize the correctness conditions of the proof linker, three auxiliary variables are introduced into the listing in Figure 2.4. “**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.

Given a fixed, admissible linking strategy \( \sigma \), the proof linker is 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:

   \[
   \forall p \in \text{Ready} \land \forall o \in \text{Obligations}[p] \land o \in \text{Satisfied}
   \]

   To enforce this syntactically, one may require that, for any linking primitives \( x \) and \( t \), if \( x \) may introduce the attachment \( \langle o, t \rangle \), then \( x <_\sigma t \).

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:

   \[
   \forall o \in \text{Satisfied}. \text{DB} \vdash o
   \]

   In a deductive database model, monotonicity results naturally from the application of Horn clause logic [40]. 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 be resulted 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 time:

   \[
   \forall o \in \text{Failed}. \text{DB} \not\vdash o
   \]

   This condition can be enforced syntactically as follows. A commitment \( c \) is said to **support** an obligation \( o \) if there is a proof of \( o \) in which \( c \) is a necessary premise. If a
linking primitive \( p \) may assert a commitment that supports \( o \), and if \( o \) may be attached to linking primitive \( q \), then one may require that \( p <_\sigma q \). Thus, if an obligation \( o \) of primitive \( q \) is eventually provable, then generation of the commitments necessary for its proof must be complete when "\textbf{begin} \( q \)" is processed.

In general, the correctness of proof linking depends on (1) the linking strategy \( \sigma \), (2) the kind of logic being used, and (3) the specific commitments and obligations returned by each linking primitive. 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.

### 2.3 Potential Benefits of Proof Linking

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 augmenting type systems.

#### 2.3.1 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 applet verification, this would mean that the bytecode verifier is manufactured as a replaceable component that can be "plugged" into any virtual machine that supports pluggable verification engine. It should be possible to validate the correctness of these verification component 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 further work in developing stand-alone verification modules.
2.3.2 Distributed Verification

Conditional Certification

Modularization makes it feasible for mobile code verification to be performed remotely. The safety conditions in Section 2.2.3 only require that the verify primitive correctly generates all commitments and obligations. It does not specify how such commitments and obligations are generated. Therefore, a remote Java bytecode verifier can analyze a classfile, attach the corresponding commitments and obligations, and conditionally certify the entire package. Upon acquiring the package, a browser can perform a special verify primitive that (1) checks the certificate of the package, and (2) processes the commitments and obligations as if they were generated locally. To the proof linker, this special verify primitive looks no different than a normal verify primitive, and will proof-link the remotely-verified classfile correctly. Had verification not been modularized, remote verification would not be possible, because the verification of one classfile would require the knowledge of other classfiles, which might not be accessible at the remote verifier’s site.

Protocol Interoperability

Proof linking provides an infrastructure for the interoperability of the verification protocols proof-on-demand, proof-carrying code and proof delegation. Specifically, an intelligent verify primitive will handle code units certified by different verification protocols in different manners. In the case of proof-on-demand, the intelligent verify primitive will verify an untrusted code unit and generate obligations and commitments as usual. In the cases of proof-carrying code and proof delegation, relevant proof obligations and commitments could be attached to an untrusted code unit by a remote verifier. Upon acquiring the code unit, the intelligent verify primitive perform either proof checking or signature authentication, and assert the attached obligations and commitments as if they were generated locally. Proof linking thus proceeds normally even in the presence of multiple verification protocols.

2.3.3 Augmenting 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 augmenting 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 augmenting 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 experimenting with alternative protection mechanisms based on static type analysis.
Chapter 3

Progress To Date and Proposed Work

The previous chapter introduces a language-independent verification architecture, Proof Linking, for mobile code hosting environments with lazy-dynamic linking. It describes a formal model upon which one could evaluate the adequacy and correctness of a language-specific instantiation of the architecture. It is also claimed that Proof Linking offers benefits including support for stand-alone verification modules, distributed verification, and augmenting type systems. In this chapter, a series of studies is proposed to evaluate the Proof Linking architecture in the context of the Java platform.

3.1 Modeling Adequacy and Soundness

As an archetypical mobile code protection mechanism, Java bytecode verification is chosen to be the first application of the Proof Linking architecture. It is hypothesized that, on one hand, Proof Linking architecture is rich enough to handle the 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 2.2.3, 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.

3.1.1 Lazy, Dynamic Linking

The first complication comes from the fact that the Java 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 super-types, and, as mentioned before, verification 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 chapter. This motivates the first research problem.

Research Problem 1: Can Proof Linking be properly instantiated to handle Java bytecode verification in the presence of Java’s lazy, dynamic linking process?
To address this research problem, the Proof Linking architecture was instantiated to handle bytecode verification in the context of a Java-specific dynamic linking model [24, 25]:

- The mentioned temporal dependencies were captured in a relatively lazy linking strategy.
- A set of relevant proof obligations and commitments was designed to capture the verification interface of Java bytecode verification.
- This instantiation of Proof Linking was formally proven to satisfy the three correctness conditions: safety, monotonicity, and completion.
- The correctness proof was formally verified by the formal specification and verification system PVS [57].

### 3.1.2 Multiple Classloaders

The second modeling complication originates from the fact that the standard Java classloading semantics uses multiple classloaders to implement namespace partitioning. 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 Chapter 2 assumes that there is only one, static namespace. This complication is especially significant in the case of remote verification, in which the verifier does not have access to the run time state of the JVM. This motivates a second research problem:

**Research Problem 2:** Can Proof Linking be correctly implemented to handle Java bytecode verification in the presence of multiple classloaders?

To address this research problem, the Proof Linking architecture was extended to account for the existence of multiple classloaders in Java [26]. 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.

### 3.1.3 Summary

In summary, the modeling adequacy and theoretical soundness of the Proof Linking architecture had been satisfactorily established in the context of Java bytecode verification. This provides a strong support for thesis statement **TS1.** In the sequel, additional studies will be proposed to evaluate the benefits of Proof Linking as promised in thesis statement **TS2.**

### 3.2 Benefits of Proof Linking

To assess if the promised benefits of the Proof Linking architecture can be feasibly realized, the Java instantiation of Proof Linking as described in Section 3.1 will be implemented in a
real JVM. This section describes a series of qualitative case studies proposed for evaluating this implementation exercise. The implementation work is subdivided into three progressive stages. A prototype from one stage brings experience that informs the proceeding of the next, eventually leading to a reusable product in the third.

3.2.1 The Aegis VM

I have been developing an open source JVM, the Aegis VM [23], as a test bed for Proof Linking. The project is still at its pre-alpha stage, but 5 development releases result in a JVM that supports the following features:

- dynamic linking
- multiple classloaders
- class unloading
- loading and unloading of native libraries
- access control
- loading constraints
- reflection

This list represents all the VM features that Proof Linking should account for. Implementation efforts proposed in the subsequent sections are all to be based on the Aegis VM.

3.2.2 Stand-alone Verification Modules

By design of the Proof Linking architecture, the verifier is a stand-alone module. Theoretical results from Section 3.1 has generated an in-depth understanding of the interplay between Proof Linking and the architectural peculiarities of the Java platforms, namely, that of lazy, dynamic linking and multiple classloaders. Such preliminary studies have offered valuable support for the feasibility of Proof Linking. Nevertheless, the claim in Section 2.3.1, that it is feasible to base an implementation of a verifier plug-in mechanism on the Proof Linking architecture, still demands empirical evidence:

**Research Problem 3:** Is it feasible to base the implementation of a plug-in mechanism for Java bytecode verifiers on the Proof Linking architecture?

This research problem will be addressed by the design and implementation of a verifier plug-in mechanism for the Aegis VM. Specifically, the implementation effort will involve the following components:

1. **Verifier Plug-in Mechanism.** An extension framework is needed for security architects to supply their own Java bytecode verifiers.
2. **Proof Linking Mechanism.** An implementation of proof linking process is needed for processing the proof obligations and commitments needed by Java bytecode verification.

3. **Verifier Plug-in API.** A Java bytecode verifier plug-in API is needed for plug-in developers to implement the posting of proof obligations and commitments, and for the VM to access the verification facilities exported by a verifier plug-in. The latter includes the entry points for (1) the modular bytecode verification routine and (2) procedural implementation of the initial theory.

The implementation exercise will enable one to investigate the architectural impact of a verifier plug-in mechanism 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 proof linking process. Such architectural complication has the potential of making it difficult for programmers to reproduce the Proof Linking architecture, and also increasing the complexity of the already delicate dynamically linking process of the JVM to an unacceptable level of brittleness.

2. **Linking strategies.** The correctness conditions imposed on linking strategies affect the temporal ordering of linking activities. A provably correct linking strategy for Java bytecode verification will likely not to be the laziest one allowed by the JVM specification. The degree in which laziness is prohibited will incur a proportional degrade on the efficiency of the overall linking process.

The extent to which these concerns are valid in the resulting implementation will be investigated.

### 3.2.3 Distributed Verification

It was claimed in Section 2.3.2 that the Proof Linking architecture addresses the issues of conditional certification and protocol interoperability in the context of distributed verification. Again, Java bytecode verification is chosen to be an archetypical application.

**Research Problem 4:** Can the Proof Linking architecture be feasibly applied to Java bytecode verification so as to support conditional certification and interoperability of multiple verification protocols?

To restrict the scope of the study, focus is placed on the integration of a signature-based verification protocol into the existing security architecture of the Java platform. This allows one to study the issue of conditional certification, an issue especially pressing for signature-based verification protocols, and the interoperability of signature-based and proof-on-demand protocols.

The research problem will be addressed by the design and implementation of a Java bytecode verification framework for the Aegis VM that will support both proof-on-demand and signature-based protocols. The components involved are as follows:
1. **Verifier Plug-in.** A special Java bytecode verifier plug-in that is aware of signed classes will be built. This plug-in will need to collaborate with the built-in authentication mechanism of the Java platform in order to bypass bytecode verification when certificated classfiles arrive. The plug-in will also need to be aware of the conditional semantics of the incoming certificates, and appropriately process proof obligations and commitments associated with the certificates.

2. **Transport Encoding.** Remote verification of Java bytecode results in a set of annotations including proof obligations, commitments and references to linking primitives. An encoding scheme is needed for representing such annotations in a Java classfile. Tools will be built in order to insert the encoded annotations into remotely verified Java classfiles. The JAR transport mechanism will be reused to associate certificates with classfiles.

Given the complexity of the high level security architecture of the Java platform, it is particularly interesting to see how Proof Linking fits in:

1. **Authentication:** How does the authorization elements of a verifier plug-in collaborate with the existing authentication facilities of the Java platform?

2. **Certification:** How can Java’s existing certification infrastructure surrounding the JAR file format be reused in the context of conditional certification?

The ways in which these technological hurdles are resolved will be investigated.

### 3.2.4 Augmenting Type Systems

The realization of Proof Linking proposed in the previous sections are specific to Java bytecode verification. In section 2.3.3, it is claimed that one could generalize an implementation of Proof Linking into a general-purpose, link-time verification architecture, so that new static analyses can be conveniently incorporated into the JVM’s dynamic linking process. 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 shall 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 to compensate for its invocation frequency.

3. **Usability.** The effort required of a programmer to introduce a new type of analysis into the generalized Proof Linking framework shall be significantly less than that without the help of the framework.
The above challenges represent a tradeoff that one must balance in the design and implementation of this open Proof Linking architecture.

**Research Problem 5:** Is it feasible for Proof Linking to be implemented as a general-purpose, efficient, and usable mechanism to service user-defined, link-time analyses?

This research problem will be addressed by a case study of two implementation efforts — the implementation of one possible design that is mindful of the issues of generality, efficiency and usability, and the application of this implementation to realize a security-related type analysis.

**An Extension Mechanism**

An extension mechanism for incorporating alternative link-time verification into the dynamic linking process of the Aegis VM is planned. Specifically, the extension mechanism will include the following components:

1. **Generic Verifier Plug-in Mechanism.** A generic extension framework is needed for security architects to introduce new kinds of link-time verification as verifier plug-ins.

2. **Generic Transport Encoding.** A generic encoding scheme is needed for representing proof obligations, commitments, and linking primitives in a Java classfile. Tools will also need to be built in order to insert encoded annotations into Java classfiles.

3. **Generic Proof Linking Mechanism.** A generic proof linking infrastructure is needed for processing proof obligations and commitments on behalf of verifier plug-ins. As mentioned in Section 2.1.3, it is unrealistic to expect that the inference system employed by Proof Linking to be a general-purpose, declarative one. Domain-specific, procedural inference is needed if Proof Linking is to be practically efficient. This amounts to building a native code dispatching mechanism specialized for guarding linking primitives.

4. **Generic Verifier Plug-in API.** A verifier plug-in API is needed for verifier plug-in developers to implement the posting of proof obligations and commitments, and for the VM to dynamically discover the capabilities of the verifier plug-ins. The latter includes entry points for (1) the modular verification routine and (2) procedural implementation of query evaluation logic to be dispatched the generic proof linking mechanism.

Notice that the implementation work proposed here closely parallels those found in the previous sections, and will thus benefit from the prior implementation experience. The above extension mechanism will be evaluated qualitatively by how well it balances the tradeoff of generality, efficiency and usability.
Application Of The Extension Mechanism

To make evaluating its adequacy concrete and informative, the above extension mechanism will be used to incorporate an augmenting type system into the Aegis VM. As the goal of the current study is not to define new type systems, a known augmenting type system will be chosen for this purpose: the read-only type system proposed in [36]. In short, this type system is a transitive variant of the C/C++ const type qualifier, in which read-only access is imposed on all objects reachable from the qualified object reference. The rationales for this decision are the following:

- The selected augmented type system should be relevant to mobile code security.
- Of the two kinds of augmenting type systems identified in Section 1.1.3, alias control appears to have more hope of being implementable in a complex environment like the JVM. Information flow control, on the contrary, is still quite immature, in the sense that known incarnations for this form of analysis are extremely conservative if they are to be safe.
- Of the two approaches in alias control, read-only types are extremely well understood, and are very easy to implement.

3.2.5 Summary

Progressive implementation efforts are proposed to demonstrate the feasibility of the three benefits promised in thesis statement TS2. A projection of the complexities involved in the implementation efforts is given. An execution plan is included as Appendix A.
Appendix A

Timeline

This appendix describes a timeline for my execution plan:

| Date       | Milestone                                           |
|------------|-----------------------------------------------------|
| Dec 31, 2002 | **Implementation.** Finish with the implementation of a stand-alone Java bytecode verification module for Aegis VM. **Writing.** Finish with the writing up of dissertation chapters 1–3, roughly corresponding to the aggregation of this proposal and an updated version of my depth exam survey paper. |
| Jan 31, 2003 | **Implementation.** Finish with the implementation of support for distributed verification in Aegis VM. **Writing.** Finish with the writing up of dissertation chapters 4–5, roughly corresponding to the work reported in [24, 25, 26] concerning the modeling adequacy and theoretical soundness of the Proof Linking architecture. |
| Feb 28, 2003 | **Implementation.** Finish with the implementation of an open proof linking mechanism, and testing the mechanism by realizing read-only types in the JVM. |
| May 31, 2003 | **Writing.** Finish with the writing up of dissertation chapters 6–9, presenting the implementation work and the concluding of the study. Schedule for thesis defense. |
Bibliography

[1] 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.

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

[3] Henry G. Baker. 'use-once’ variables and linear objects — storage management, reflection and multi-threading. ACM SIGPLAN Notices, 30(1):45–52, January 1995. Also available at http://home.pipeline.com/~hbaker1/Use1Var.html.

[4] D. Elliot Bell and Leonard J. LaPadula. Secure computer systems: Mathematical foundations. Technical Report 2547 (Volume I), MITRE, March 1973. Also available at http://www.mitre.org/resources/centers/infosec/secure_computers/.

[5] K. Biba. Integrity considerations for secure computer systems. Technical Report 76–372, U. S. Air Force Electronic Systems Division, 1977.

[6] Boris Bokowski and Jan Vitek. Confined types. In Proceedings of the 14th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA’99), volume 34 of ACM SIGPLAN Notices, pages 82–96, Denver, Colorado, November 1999. Also available at http://www.cs.purdue.edu/homes/jv/pub/oopsla99.ps.gz.

[7] Chandrasekhar Boyapati. Towards an extensible virtual machine. Technical Report MIT-LCS-TR-842, MIT Laboratory for Computer Science, Cambridge, MA 02139, April 2002. Also available at http://www.pmg.lcs.mit.edu/~chandra/publications/evm.ps.

[8] 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), volume 37 of ACM SIGPLAN Notices, Seattle, Washington, November 2002. Also available at http://www.pmg.lcs.mit.edu/~chandra/publications/oopsla02.ps.
[9] 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), volume 36 of ACM SIGPLAN Notices, Tampa Bay, Florida, October 2001. Available at http://www.pmg.lcs.mit.edu/~chandra/publications/oopsla01.ps.

[10] John Boyland. Alias burying: Unique variables without destructive reads. Software — Practice and Experience, 31(6):533–553, May 2001. Also available at http://www.cs.uwm.edu/~boyland/papers/unique.html.

[11] John Tang Boyland. Alias killing: Unique variables without destructive reads. In Intercontinental Workshop on Aliasing in Object Oriented Systems (IWAOOS'99), Lisbon Portugal, June 1999. Also available at http://cui.unige.ch/~ecoopws/iwaos/papers/papers/boyland.ps.gz.

[12] 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), Boston, MA, May 1997.

[13] 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), volume 33 of ACM SIGPLAN Notices, pages 48–64, Vancouver, BC, October 1998. Also available at http://citeseer.nj.nec.com/clarke98ownership.html.

[14] Christopher Colby, Peter Lee, George C. Necula, Fred Blau, Ken Cline, and Mark Plesko. A certifying compiler for Java. In Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'00), Vancouver, BC, Canada, June 2000. Also available as http://www-nt.cs.berkeley.edu/home/necula/public_html/pldi00b.ps.gz.

[15] 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.

[16] Dorothy E. Denning. A lattice model of secure information flow. Communications of the ACM, 19(5):236–243, May 1976. Also available at http://www.cs.georgetown.edu/~denning/publications.html.

[17] Dorothy E. Denning. Cryptography and Data Security. Addison-Wesley, 1982.

[18] Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. Communications of the ACM, 20(7):504–513, July 1977.

[19] Prem Devanbu and Stuart Stubblebine. Automated software verification with trusted hardware. In Proceedings of the Twelfth International Conference on Automated Software Engineering, 1997.
[20] 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. Also available at http://seclab.cs.ucdavis.edu/~devanbu/files/icse98.pdf.

[21] Patrick Doyle and Tarek Abdelrahman. Jupiter: A modular and extensible JVM. In *Proceedings of the Third Annual Workshop on Java for High-Performance Computing*, ACM International Conference on Supercomputing, pages 37–48, Sorrento, Italy, June 2001. ACM. Also available at http://www.eecg.toronto.edu/~doyle/jupiter/jhpc01.pdf.

[22] 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 as http://csrc.nist.gov/fips/fips1401.pdf.

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

[24] 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. Also available at http://www.cs.sfu.ca/~pwfong/personal/Pub/fse98.ps.

[25] 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. Also available at http://www.cs.sfu.ca/~pwfong/personal/Pub/tosem2000.ps.

[26] 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. Also available at http://www.cs.sfu.ca/~pwfong/personal/Pub/jvm01.ps.

[27] 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.

[28] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison-Wesley, 2nd edition, 2000. Also available at http://java.sun.com/docs/books/jls/.

[29] 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)*, volume 26 of *ACM SIGPLAN Notices*, pages 271–285, Phoenix, Arizona, November 1991.
[30] John Hogg, Doug Lea, Alan Wills, Dennis de Champeaux, and Richard Holt. The Geneva convention on the treatment of object aliasing. *OOPS Messenger*, 3(2), April 1992. Also available at http://gee.cs.oswego.edu/dl/aliasing/aliasing.html.

[31] IBM. IBM PCI cryptographic coprocessor. Available at http://www-3.ibm.com/security/cryptocards.

[32] JavaSoft. Java security: Chronology of security-related bugs. Available at http://java.sun.com/sfaq/chronology.html.

[33] Kimera Team. Security flaws in Java implementations. Available at http://kimera.cs.washington.edu/flaws/.

[34] Gerwin Klein and Tobias Nipkow. Verified lightweight bytecode verification. In *ECOOP 2000 Workshop on Formal Techniques for Java Programs*, 2000. Also available at http://www4.informatik.tu-muenchen.de/~nipkow/pubs/lbv.ps.

[35] Günter Kniesel and Dirk Theisen. JAC — Java with transitive read-only access control. In *Intercontinental Workshop on Aliasing in Object-Oriented Systems (IWAOOS'99)*, Lisbon, Portugal, June 1999. Also available at http://cui.unige.ch/~ecoopws/iwaoo/papers/papers/kniesel.ps.gz.

[36] Günter Kniesel and Dirk Theisen. JAC — access right based encapsulation for Java. *Software — Practice and Experience*, 31(6):555–576, May 2001. Also available at http://javablab.cs.uni-bonn.de/data2/papers/ace/jac.spe2000.pdf.

[37] Leonard J. LaPadula and D. Elliot Bell. Secure computer systems: A mathematical model. Technical Report 2547 (Volume II), MITRE, May 1973. Also available at http://www.mitre.org/resources/centers/infosec/secure_computers/.

[38] Stanley Letovsk and Elliot Soloway. Delocalized Plans and Program Comprehension. *IEEE Software*, pages 41–49, May 1986.

[39] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison Wesley, 2nd edition, 1999. Also available at http://java.sun.com/docs/books/vmspec.

[40] John Wylie Lloyd. *Foundations of Logic Programming*. Symbolic computation: Artificial intelligence, Springer-Verlag, 2nd, extended edition edition, 1987.

[41] Gary McGraw and Edward W. Felten. *Java Security: Hostile Applets, Holes, and Antidotes*. John Wiley & Sons, Inc, 1997.

[42] 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. Also available at http://www.cs.rutgers.edu/~minsky/papers/unique-papers.ps.
[43] George C. Necula. Proof-carrying code. In Proceedings of the 24th ACM Symposium on Principles of Programming Languages (POPL'97), Paris, France, January 1997. Also available at http://www-nt.cs.berkeley.edu/home/necula/public_html/popl97.ps.gz.

[44] George C. Necula and Peter Lee. Safe kernel extensions without run-time checking. In Proceedings of the Second Symposium on Operating System Design and Implementation (OSDI'96), Seattle, WA., October 1996. Also available at http://www.cs.cmu.edu/~necula/osdi96.ps.gz.

[45] James Noble, Jan Vittek, 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. Also available at http://www.cs.purdue.edu/homes/jv/pub/ecoop98.ps.gz.

[46] Hirotaka Ogawa, Kouya Shimura, Satoshi Matsuoka, Fuyuhiko Maruyama, Yukihiko Sohda, and Fuminori Kimura. OpenJIT: An open-ended, reflective JIT compiler framework for Java. In Proceedings of the 14th European Conference on Object-Oriented Programming, Sophia Antipolis and Cannes, France, June 2000. Also available at http://www.openjit.org/publications/ecoop2000.

[47] Princeton Secure Internet Programming Group. History of the group. Available at http://www.cs.princeton.edu/sip/history.

[48] Eva Rose and Kristoffer Hogsbro Rose. Lightweight bytecode verification. In The OOPSLA’98 Workshop on Formal Underpinnings of Java, Vancouver, BC, Canada, November 1998. Available at http://www-dse.doc.ic.ac.uk/~sue/oopsla/rose.f.ps.

[49] Spencer Rugaber, Kurt Stirewalt, and Linda M. Wills. Understanding interleaved code. Automated Software Engineering, 3(1–2):47–76, June 1996.

[50] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. IEEE Journal on Selected Areas in Communications, 2002. To appear. Also available at http://www.cs.cornell.edu/~andrei/jsac.ps.

[51] 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. Also available at http://www.dsv.su.se/~matte/publ/ecoop2001.html.

[52] Mats Skoglund and Tobias Wrigstad. Alias control with read-only references. In Proceedings of the 6th International Conference on Computer Science and Informatics, Durham, NC, March 2002. Also available at http://www.dsv.su.se/~matte/publ/csi_2002.html.

[53] 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.
[54] 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.

[55] 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. Also available at http://www.cs.nps.navy.mil/research/languages/papers/atsc/popl98.ps.Z.

[56] 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. Available at http://www.elsevier.com/gej-ng/31/29/23/44/23/62/tcs20009.ps.

[57] SRI Computer Science Laboratory. The PVS specification and verification system. Available at http://pvs.csl.sri.com/.

[58] Sun Microsystems. JAR File Specification. Available at http://java.sun.com/j2se/1.4.1/docs/guide/jar/jar.html.

[59] Sun Microsystems. Connected, Limited Device Configuration: Java 2 Platform Micro Edition. Sun Microsystems, May 2000. Available at http://java.sun.com/products/cldc/.

[60] Sun Microsystems. Connected Device Configuration: Java 2 Platform Micro Edition. Sun Microsystems, March 2001. Available at http://java.sun.com/products/cdc/.

[61] Jan Vitek and Boris Bokowski. Confined types in java. Software — Practice and Experience, 31(6):507–532, May 2001. Also available at http://www.cs.purdue.edu/homes/jv/pub/spe00.ps.gz.

[62] 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. Also available at http://www.cs.nps.navy.mil/people/faculty/volpano/papers/csfw00.ps.gz.

[63] 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. Also available at http://www.cs.nps.navy.mil/research/languages/papers/atsc/csfw97.ps.Z.

[64] 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. Also available at http://www.cs.nps.navy.mil/research/languages/papers/atsc/tapsoft97.ps.Z.

[65] 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. Also available at http://www.cs.nps.navy.mil/research/languages/papers/atsc/lncs98.ps.Z.

[66] Dennis Volpano and Geoffrey Smith. Probabilistic noninterference in a concurrent language. Journal of Computer Security, 7(2,3):231–253, November 1999. Also available at http://citeseer.nj.nec.com/volpano00verifying.html.

[67] 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, 2000. Also available at http://citeseer.nj.nec.com/volpano00verifying.html.

[68] 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. Also available at http://www.cs.nps.navy.mil/research/languages/papers/atsc/jcs.ps.Z.

[69] WAP Forum. WAP-194-WMLScript Standard Libraries Specification, version 1.3 edition. Available at http://www.wapforum.org/what/technical.htm.

[70] WAP Forum. WAP-193-WMLScript Language Specification. WAP Forum, version 1.2 edition, June 2000. Available at http://www.wapforum.org/what/technical.htm.

36