Proof Linking: Distributed Verification of Java Classfiles in the Presence of Multiple Classloaders

Philip W. L. Fong
Simon Fraser University, B.C., Canada
pwfong@cs.sfu.ca
Robert D. Cameron
Simon Fraser University, B.C., Canada
cameron@cs.sfu.ca

Abstract

To offload the computational burden of bytecode verification within Java Virtual Machines (JVM), distributed verification systems may be created using any one of a number of verification protocols, based on such techniques as proof-carrying code and signed verification by trusted authorities. This paper advocates the adoption of a previously-proposed mobile code verification architecture, proof linking, as a standard infrastructure for performing distributed verification in the JVM. Proof linking not only supports both CLDC-style and signature-based distributed verification protocols, but it also provides interoperability between the two. To ground our work in the real-world requirements of Java bytecode verification, we also extend previous work on proof linking to handle multiple classloaders.

1 Introduction

Security is the cornerstone of trustworthy mobile code systems such as that of Java. In accepting arbitrary mobile code from unknown and potentially untrustworthy sources, a Java Virtual Machine (JVM) enforces type safety—the first line of defence in mobile code security—through a a link-time bytecode verification process. The bytecode verifier performs dataflow analysis and various structural analyses to guarantee that untrusted classfiles can be linked into the JVM without producing type confusion. We 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 and 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 [12, p. 110]. Moreover, the *architectural complexity* of the JVM is greatly increased by the coupling of complex verification logic with lazy dynamic linking [7]. Compounding these concerns, the bytecode verifier also adds significantly to the JVM's *memory footprint* [16, 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 [17] 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 [16]—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 CDLC specification has hence rejected the proof-on-demand approach.

Future systems will also 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 integrity. The complex program analyzers necessary to verify these additional properties may well become impractical even for a standard JVM, let alone a CLDC-compliant device.

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 labour 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.

This paper is a contribution to the ongoing development of a standard distributed verification architecture for mobile code systems in general, and for the JVM in particular. The focus of this paper is twofold:

1. We refine our previously-proposed *proof linking* architecture [5, 7] as an architectural framework for supporting distributed verification of Java bytecode. In particular, our framework provides for highly efficient signature-based verification protocols, as well as interoperability between different verification protocols.
2. We extend our previous work to account for a unique feature of the JVM, namely, multiple classloaders. We articulate how this extension preserves the modularity of our architecture as well as the correctness conditions of the proof linking strategy. We further describe how this extension may be implemented, not only in a CLDC-compliant environment, but also in a standard JVM.

2 Distributed Verification: Related work and major issues

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.

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* [14, 13]. 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 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 [15, 9, 1], and has further been incorporated in the stack map method of the CLDC specification [16, Sec. 5.3].

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 we call *proof delegation* [2, 3]. 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 [8]. 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 [4]. 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 atttached 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. However, in order to support signature-based verification protocols such as proof delegation, 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 run-time notion, and a classfile is safe only if we check it against the loaded classes on the consumer side. For example, during verification of a classfile, the dataflow analyzer might need to show that class A is a subclass of class B. This fact can only be shown by examining the classes that are already loaded into the consumer's JVM. As a result, a conditional certification semantics for signature is needed. That is, signatures certify that a classfile is safe if specified external dependencies are further validated on the consumer side at link time.

2. **Protocol Interoperability.** A Java developer may use some off-the-shelf components, and write "glue" code to orchestrate their interaction. A common 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 protocols, but also need to *combine two different kinds of certificate (signatures and stack maps) when accessing the safety of the whole program*. What is needed, then, is a mechanism to hide the details of a module's certificate, and examine only its *certification interface*, which offers us a safe mechanism for combining certificates.

In support of both self-certifying protocols and signature-based methods, this paper proposes a *proof linking* architecture with the following features.

1. Each module is *verified separately*, using its own verification protocol.

2. Certificates are annotated by a well-defined verification interface, namely, a set of *proof obligations and commitments*, which delineate the external dependencies of the certified unit.

3. A lightweight, incremental verification process, namely, *proof linking*, is integrated into JVM's linking algorithm, so that proof obligations can be discharged by commitments.

This architecture closely follows that previously developed to deal with modularity concerns in Sun's

JVM [5, 7]. Here, however, we refine the architecture to addess the needs for conditional certification and protocol interoperability in distributed verification systems.

3 The Proof Linking Architecture

3.1 Assumption, Notations and Modelling

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 [10]. For the purpose of introducing the proof linking architecture and its application to distributed verification, however, this section makes the simplifying assumption that only a single classloader exists. This assumption will be relaxed in subsequent sections.

As a result of the single-classloader simplification, symbolic class references and loaded classes are each denoted simply by classname. A member named M of a class X with type signature S is identified by the symbolic reference X::$M(S)$. We do not differentiate a symbolic member reference and an actual member of a loaded class.

We model the linking activities of the simplified JVM by a set of *linking primitives*, including, for example, "**load** X", "**verify** X", "**resolve** Y **in** X", "**resolve** Y::$M(S)$ **in** X", and so on. The JVM executes linking primitives in a concurrent, nondeterministic fashion, in accordance with a well-defined partial ordering enforced by the implementation. We call this partial order the *linking strategy* of the runtime environment. Further details are presented in Section 4.

3.2 A Motivating Example

Signature-based verification protocols involve the remote verification of classfiles. Safety is in general a whole-program notion, and thus cannot be established by merely examining a single module. External dependencies (e.g., checking for assignment compatibility, computing the meet of two flow values, and so on) validated remotely may no longer hold when a classfile is linked into the runtime environment. The following example is illustrative.

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::$M(S)$. A bytecode verifier is supposed to check that class C is a subclass of class A. If the verification of class C is performed at link time, the bytecode verifier will recursively invoke the classloader to bring in the classfiles for both B and C to validate the required subclassing relationship. However, if the verification of class C is carried out remotely, this strategy will fail for two reasons. First, the remote verifier might not have access to the classfiles for A and B. Second, even if it does, the versions of A and B considered by the remote verifier may not be compatible with the ones actually loaded into the JVM.

This configuration management issue motivates a two-stage verification architecture based on conditional certification and proof linking. In the first stage, remote verification is performed in a modular fashion to generate conditional certificates. These certificates specify the assumptions (dependencies) that must be checked to ascertain the safety of the code unit in question. In the second stage, the JVM performs proof linking to check the asserted conditions in the context of the classes actually loaded.

3.3 First Stage: Modular Verification

Untrusted classfiles are certified by a trusted verifier at a remote site. Instead of endorsing external dependencies that might be invalidated at runtime, the remote verifier computes, for each classfile, a conservative *safety precondition* summarizing the external dependencies that must hold at runtime in order for the classfile to be safe. The safety precondition is here *represented* as a conjunctive set of database queries. In the running example, during the remote verification of classfile C, a trusted verifier will generate the query `subclass(`C`,`A`)` as the `invokespecial` instruction is scanned. The remote verifier may end up generating many such queries. The conjunctive set of all the queries formulated by the remote verification session becomes the safety precondition for endorsing the classfile being considered.

In addition, the remote verifier also schedules each of the queries for evaluation. Each query describes a safety precondition for a particular 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 resolution of A::$M(S)$ within class C. Such a query is said to be the *proof obligation* for the associated primitive, representing a condition that must be met if the runtime system is to safely execute the corresponding linking primitive. We also say that a proof obligation is *attached* to its associated primitive.

In order for the runtime system to discharge proof obligations, the remote 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 remote verification of the classfile C, the fact `extends(`C`,`B`)` is generated as one of the commitments. As we shall see below, this fact will be used to satisfy the subclassing query in our example.

Prior to shipping a classfile, the remote verifier will package together the classfile itself, the obligations annotated with the associating primitives, and the commitments. The entire package is signed by the verifier using its private key. The signed package is then distributed to consumers. The obligations, commitments, and the signature can all be packaged inside a classfile using classfile attributes (see the JVM specification for details [11]).

3.4 Second Stage: Proof Linking

When a remotely certified classfile X arrives at the code consumer's site, it is loaded into the JVM. The local "**verify** X" primitive, instead of performing bytecode verification on X, will unpackage the classfile, authenticate the signature, and then process the proof obligations and commitments following the procedure outlined below.

We assume that the runtime environment is equipped with two data structures: (1) an *obligation table* mapping linking primitives to their attached proof obligations, and (2) a *commitment database* holding commitments already known by the JVM. After unpackaging an untrusted classfile, the local "**verify** X" primitive records the newly obtained proof obligations in the corresponding entry of the obligation table, and asserts the commitments into the commitment database.

Subsequently, when a linking primitive is to be executed, the JVM will (1) look up the obligations that are already attached to the primitive, (2) attempt to satisfy each of the obligations by consulting the commitment database (raising an appropriate linking exception if the attempt fails), and (3) perform the action prescribed by the primitive.

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

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

After the local verifier has processed the classfiles for B and C, the commitment database may contain the following commitments:

```
extends(C,B).
extends(B,A).
```

When the 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 JVM then attempts to satisfy this query by consulting the facts and rules within the commitment database. The query succeeds and the primitive is executed (assuming that any other obligations are satisfied as well).

3.5 What We Have Gained

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

Proof linking also provides interoperability between distributed verification protocols such as proof delegation, proof-carrying code, and proof-on-demand. Because linking primitives communicate solely by attaching obligations and asserting commitments, they are highly modular. A local verifier may process classfiles annotated with CLDC-style stack maps to generate the appropriate proof obligations and commitments just as a remote verifier would do through bytecode analysis. In the event of a classfile that is neither signed by a remote verifier nor annotated with stack map, the local verifier may itself perform a complete bytecode verification to generate the necessary proof obligations and commitments. The proof linking mechanism checks and discharges obligations from each of these sources in the same way, without any need to know the verification protocol used for a particular classfile. As a result, a Java application could consist of a number of classfiles that are signed remotely, others that are "pre-verified", and still others that are completely uncertified. As long as each verification protocol uses the same *verification interface* for asserted proof obligations and commitments, interoperability is assured.

3.6 Correctness Considerations

That the incremental proof linking procedure is as safe as proof-on-demand is not obvious. Could an obligation be generated after its attachment target is already executed? Could it be possible that checked obligations are falsified by subsequently generated commitments? Is it possible that an obligation fails only because the commitments necessary for satisfying it are generated too late? To address these concenrs we formulate the following three correctness conditions.

1. **Safety.** Every primitive that could attach obligations to another primitive p must be completed before the execution of p begins.
2. **Monotonicity.** Obligations may not be contradicted by subsequently asserted commitments.
3. **Completion.** Let o be an obligation that could be attached to a primitive p. All primitives that could generate commitments contributing to the satisfaction of o must be completed prior to the execution of p.

By reasoning about the partial ordering of primitives, and analyzing the structure of the commitments, obligations and the initial theory, we have formally established the three correctness conditions above for our Java proof linking formulation, and have check the proof with the aid of a mechanical theorem prover [7].

3.7 Implementation

Throughout the presentation above, we have used a simple logic programming notation to represent the proof obligations, commitments, and initial theory. However, this abstract model is presented as such only to clarify ideas and to facilitate the articulation of correctness (see Section 5). By no means are we suggesting that an actual implementation employ a logic programming system in proof linking. Rather, commitments can be optimized as flags and pointers stored in the Class and Method objects inside the JVM. A recursive query such as subclassing could well be realized by a standard tree traversal algorithm. The obligation table need not even exist physically; obligations attached to a **resolve** primitive could be stored in the corresponding constant pool entry. Such optimizations were used in the prototype implementations reported previously [5, 7].

4 Java Proof Linking for Multiple Classloaders

The discussion above assumes a simplified JVM with only one classloader. We relax the assumption in this section, and analyze the interaction between proof linking and dynamic linking in the setting of multiple classloaders. It turns out that a systematic, straightforward set of extensions to the previously proposed model is sufficient to make proof linking work with multiple classloaders. This demonstrates that the proof linking technique is applicable to realistic mobile code environments and is orthogonal to Java's delegation-style classloading.

4.1 Enter Multiple Classloaders

When a Java application attempts to load a class C with a given name X with a classloader L_i, *the ini-*

load $\langle X, J \rangle$	Define a class with name X and defining classloader J.
verify $\langle X, J \rangle$	Using the appropriate verification protocol, 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 1: The Extended Set of Linking Primitives for Java

tiating classloader of C, L_i may delegate the classloading task to another classloader, which, in turn, might delegate the task to yet another classloader. 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 L_i initiates the loading of $\langle X, L_d \rangle$. When a symbolic reference Y is resolved in a class $\langle X, L \rangle$, the classloader L will be used as the initiating classloader for class Y. We identify the member $M(S)$ of a loaded class $\langle X, L \rangle$ by the notation $\langle X{::}M(S), L \rangle$. Details of Java's classloading mechanism are described elsewhere [11, Chapter 5][10].

4.2 Overview of the Solution Approach

Since a loaded class is identified not only by its classname, but also by its defining classloader, a remote verifier has no way of naming the classes in the commitments and obligations it generates. To deal with this, we extend our scheme by the following set of reformulations:

1. The remote verifier expresses commitments and obligations not in terms of loaded classes, but in terms of symbolic class references.
2. The local **verify** primitives tag the symbolic class references by their initiating classloaders.
3. Name binding events are explicitly modeled as linking primitives generating binding commitments.
4. Translation rules are introduced into the initial theory for explicit resolution of tagged symbolic references into loaded classes using binding commitments.

4.3 Linking Primitives

We begin the discussion of our extended proof linking model by looking at its linking primitives. We define a linking primitive as a self-contained action which never activates other primitives as a subroutine, nor recursively invokes itself. The JVM executes a linking primitive p using the following protocol:

1. Look up the proof obligations that have been attached to primitive p.
2. Attempt to satisfy each of the obligations using the commitments that have been asserted into the commitment database. Raise an appropriate exception if any obligation is unsatisfiable.
3. Perform the action prescribed by the linking primitive.
4. Collect the new proof obligations generated by the primitive, and record the obligations in the corresponding entry of the obligation table.
5. Collect the new commitments generated by the primitive, and assert them into the commitment database.

Our multiple-classloader linking model contains the extended set of linking primitives in Figure 1. We inherit the two **endorse** primitives from our original work. They are auxiliary targets of obligation attachment. Since they do not contribute much to our present discussion, readers may safely identify them with class preparation. Only the **verify** primitives generate obligations, and only the **verify** and **bind** primitives generate commitments.

Two changes to the original primitive set have been made [7, Sec. 4.1]:

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. These 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 JVM orders the nondeterministic, concurrent execution of linking primitives according to the constraints prescribed by the linking strategy found in Section 4.7.

4.4 Static Type Rules

The static type rules for Java under the single classloader assumption have been presented previously [7, Figure 6]. A straightforward mechanical translation to replace classnames with loaded class notations adapts these rules for the multiple classloader case. For example, consider the subclassing rule mentioned above.

$$\begin{array}{l}\texttt{subclass(}X\texttt{,}X\texttt{).}\\ \texttt{subclass(}X\texttt{,}Y\texttt{) :-}\\ \quad\texttt{extends(}X\texttt{,}Z\texttt{),}\\ \quad\texttt{subclass(}Z\texttt{,}Y\texttt{).}\end{array}$$

This rule is transformed as follows.

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

A list of all the reformulated rules is available in a technical report [6, Figure 2].

4.5 Commitment Assertion

Suppose that a classfile with classname X is being verified remotely, and that X extends a class with name Y. The verifier must assert a commitment specifying this subclassing relationship. However, at remote-certification time, the defining classloaders J and K for the classes X and Y respectively are unknown, so the commitment cannot be phrased in terms of $\langle X, J\rangle$ and $\langle Y, K\rangle$. Three reformulations address this problem. First, the remote verification procedure instead formulates the commitment:

$$\texttt{extends(this,}\,Y\texttt{)}$$

The relative reference `this` represents the class being verified. When the commitments are actually asserted into the commitment database by the local "**verify** $\langle X, J\rangle$" primitive, the defining classloader J for class X is known. Therefore, whenever "**verify** $\langle X, J\rangle$" asserts a commitment p, it *systematically* tags the commitment as $p\,\texttt{@}\,\langle X, J\rangle$. For example, the commitment above will be asserted as:

$$\texttt{extends(this,}\,Y\texttt{)}\,\texttt{@}\,\langle X,J\rangle$$

A list of all the reformulated commitments that a remote bytecode verifier should generate can be found in the technical report [6, Figure 4].

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.

Third, the initial theory is augmented with translation rules that express how subgoals expressed in terms of loaded class notations may be satisified using corresponding goals using tagged commitments. For example, to evaluate queries of the form $\texttt{subclass(}\langle X,J\rangle\texttt{,}\langle Y,K\rangle\texttt{)}$, we will eventually need to evaluate subgoals of the form $\texttt{extends(}\langle X,J\rangle\texttt{,}\langle Y,K\rangle\texttt{)}$ using the tagged commitments above. To do so, the following translation rule is used.

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

The rule basically retrieves the corresponding tagged commitment, and then validates binding information by consulting the 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 presented in the technical report [6, Figure 6].

4.6 Obligation Attachment

As with commitments, a remote verifier cannot identify the defining classloader for the classes appearing in obligations. We then follow the same strategy and formulate obligations in terms of static classnames, and then rely on the local **verify** primitive to tag the obligations with the context in which they are to be evaluated. For example, the remote verifier may formulate an obligation of the following form:

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

When the local **verify** primitive processes this obligation, it tags the query with an evaluation context before attachment:

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

Similar tagging is systematically applied to each obligation [6, Figure 7].

Translation rules 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 `subclass/2` queries:

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

The translation rule basically resolves all the symbols in the tagged context, and evaluates a corresponding query in terms of loaded classes. The complete set of these translation rules is presented in the technical report [6, Figure 8].

Recall that, in the running example, the above subclassing obligation should be attached to the primitive "**resolve** Z::$M(S)$ **in** $\langle X, J\rangle$", which is identified by the loaded class reference $\langle X, J\rangle$. 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 [6, Figure 7]. The remote verifier may thus formulate the target of attachment in terms of place holders:

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.

4.7 Linking Strategy

A linking strategy schedules the execution of linking primitives, and coordinates incremental proof linking. The following notations are used to express ordering constraints. For linking primitives p and q, the constraint "$p < q$" requires that any execution of primitive q should be preceded by the completion of primitive p. The constraint "$p < q$ if g" requires that execution of q must not begin if g holds and p has not yet completed.

Java proof linking requires the ordering constraints shown in Figure 2. Except for the newly introduced *Proper Resolution Property [PR]*, these constraints for the multiple-classloader case are extensions to the corresponding constraints for the single classloader case [7, Sec. 4.1].

4.8 Putting It All Together

To illustrate how the scheme above works, consider a refinement of the running example. 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 $\texttt{subclass(}C\texttt{, }A\texttt{) @ }\langle C, L_3\rangle$ will be attached to the primitive "**resolve** A::$M(S)$ **in** $\langle C, L_3\rangle$". When the obligation is checked, the subgoals in Figure 3 will be generated. The original obligation

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

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

[PR] **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:

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

[IC] **Import-Checked Property:** Resolving a symbolic reference requires that the target object is well-defined:

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

[SD] **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`)` `@` $\langle X, J\rangle$), We require that all superclasses and superinterfaces of $\langle X, J\rangle$ to be loaded, verified and bound before $\langle X, J\rangle$ is used. To address this need, we require that

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

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

`subtypedependent(`X^J`)` `@` $\langle X, J\rangle$.
`subtypedependent(`Y^L`)` `@` $\langle X, J\rangle$:-
  `subtypedependent(`Z^K`)` `@` $\langle X, J\rangle$,
  $Z^K \mapsto \langle Z, L\rangle$,
  `extends(this,` Y`)` `@` $\langle Z, L\rangle$.

`subtypedependent(`Y^L`)` `@` $\langle X, J\rangle$:-
  `subtypedependent(`Z^K`)` `@` $\langle X, J\rangle$,
  $Z^K \mapsto \langle Z, L\rangle$,
  `implements(this,` I`)` `@` $\langle Z, L\rangle$,
  `member(`Y`,` I`)`.

[RD] **Referential Dependency Property:** Sometimes, verification of a class Y is needed before we can safely endorse a method $\langle X{::}M(S), J\rangle$. 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, we say that Y is *relevant* to the endorsing of $\langle X{::}M(S), J\rangle$. We assume that, remote verification of the bytecode for method $\langle X{::}M(S), J\rangle$ generates commitments `relevant(`Y`, this::`$M(S)$`)` `@` $\langle X, J\rangle$ for all relevant class symbols Y, and we require that:

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

That is, we want to collect the commitments for all relevant classes (plus their superclasses and superinterfaces) before we check the obligations attached to "**endorse** $\langle X{::}M(S), J\rangle$".

Figure 2: The Extended Java Linking Strategy

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

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. We have also annotated all the innermost subgoals 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 *[NP]*, the primitive "**bind** C^{L_3} **to** $\langle C, L_3\rangle$" has already been executed. Also, subgoal 1.2 is satisfiable because

$$\begin{array}{lll} & \textbf{bind } A^{L_3} \textbf{ to } \langle A, L_1\rangle & \\ < & \textbf{resolve } A \textbf{ in } \langle C, L_3\rangle & \ldots\textit{[PR]} \\ < & \textbf{resolve } A{::}M(S) \textbf{ in } \langle C, L_3\rangle & \ldots\textit{[NP]} \end{array}$$

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 *[SD]* and other ordering constraints, we deduce

$$\begin{array}{lll} & \textbf{verify } \langle B, L_2\rangle & \\ < & \textbf{bind } B^{L_2} \textbf{ to } \langle B, L_2\rangle & \ldots\textit{[NP]} \\ < & \textbf{bind } B^{L_3} \textbf{ to } \langle B, L_2\rangle & \ldots\textit{[PR]} \\ < & \textbf{endorse } \langle C, L_3\rangle & \ldots\textit{[SD]} \\ < & \textbf{resolve } B{::}M(S) \textbf{ in } \langle C, L_3\rangle & \ldots\textit{[NP]} \end{array}$$

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. These criteria are considered in detail in the next section.

5 Correctness

Given a well-defined linking strategy, proof linking is correct if we can establish the three correctness conditions: Safety, Monotonicity and Completion [7, Sec. 3.3].

5.1 Safety and Monotonicity

Establishment of the Safety and Monotonicity properties follows the corresponding arguments for the single-classloader case [7, Sec. 4.3].

1. **Safety:** Notice that, when the local "**verify** $\langle C, L_3\rangle$" primitive generates the obligation `subclass(`C`,` A`)` `@` $\langle C, L_3\rangle$, the obligation is attached to "**resolve** A::$M(S)$ **in** $\langle C, L_3\rangle$". By the Natural Progression Property *[NP]*, the obligation is always attached on time. Similar reasoning can be applied to all the obligations [6, Figure 5].

(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 4: 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 forms a monotonic, horn clause logic (see [6] for details).

5.2 Completion

Completion has to be established on an obligation-by-obligation basis. Continuing with our running example in section 4.8, we 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** $\langle X_0, L_0\rangle$". Our 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 found in Figure 3, the proof tree of the obligation contains the innermost subgoals in Figure 4.

We number the subgoals as (α-i), (β-i) and (γ). We want to show 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 4.8, the Proper Resolution Property *[PR]* guarantees that supporting commitment (γ) is already in place. We use induction 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,

$$\begin{array}{lll} & \textbf{verify } \langle X_0, L_0\rangle & \\ < & \textbf{bind } X_0^{L_0} \textbf{ to } \langle X_0, L_0\rangle & \ldots\textit{[NP]} \\ < & \textbf{resolve } X_k\text{::}M(S) \textbf{ in } \langle X_0, L_0\rangle & \ldots\textit{[NP]} \end{array}$$

Induction Step: Assume that commitments (α-i) and (β-i) are already in place, for $0 \le i < k$, 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. We then have

$$\begin{array}{lll} & \textbf{verify } \langle X_k, L_k\rangle & \\ < & \textbf{bind } X_k^{L_k} \textbf{ to } \langle X_k, L_k\rangle & \ldots\textit{[NP]} \\ < & \textbf{bind } X_k^{L_{k-1}} \textbf{ to } \langle X_k, L_k\rangle & \ldots\textit{[PR]} \\ < & \textbf{endorse } \langle X_0, L_0\rangle & \ldots\textit{[SD]} \\ < & \textbf{resolve } X_n\text{::}M(S) \textbf{ in } \langle X_0, L_0\rangle & \ldots\textit{[NP]} \end{array}$$

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.

6 Discussion

6.1 Implementation Guidelines

The linking strategy above suggests a natural implementation of proof linking in a JVM with multiple classloaders. In particular, a call to the `defineClass` method of class loader J, with argument X as the expected classname, will execute "**load** $\langle X, J\rangle$" and "**verify** $\langle X, J\rangle$". A call to the `loadClass` method of class loader K, requesting the loading of class X, corresponds to the primitive "**bind** X^K **to** $\langle X, J\rangle$". The primitive "**endorse** $\langle X, J\rangle$" could be executed when the class "$\langle X, J\rangle$" is prepared [11, Sec. 5.4.2]. The primitive "**endorse** $\langle X{::}M(S), J\rangle$" could be executed right before the method is first resolved. The resolution primitives coincide with regular symbol resolution.

The translation rules, binding commitments, and tagging can be optimized readily. For example, a classloader usually has a hash table storing the classes whose loading it has initiated. Such a table can be reused to represent binding commitments. Also, tagging is just an abstract way to say that the tagged commitments/obligations are stored in the Class objects themselves. Notice that this suggests a very convenient way to retract commitments/obligations when a class is finalized.

6.2 Comparison with Sun's Linking Strategy

As opposed to Sun's JVM implementation, which postpones bytecode verification until a class is linked, the implementation strategy above performs eager verification, that is, the local "**verify** $\langle X, J\rangle$" primitive is executed immediately after $\langle X, L\rangle$ is defined. This is necessary to ensure that commitments are gathered as soon as possible. Sun's JVM performs one pass of verification at class definition time, postponing the second and third passes until link time.

6.3 Implementation Status

We are in the process of implementing a proof linker in a CLDC-compliant KVM extended with multiple classloaders. The exercise has clarified a lot of our thoughts, and has confirmed the compatibility of proof linking and Java's delegation style classloading mechanism.

7 Conclusion

We advocate the adoption of the proof linking architecture as a standard framework for conducting distributed verification for the JVM. The architecture supports the notion of conditional certification essential for signature-based verification protocols, and offers interoperability among various distributed verification protocols. We have also extended our original proof linking model to account for the presence of multiple classloaders in the standard JVM, thereby showing that proof linking is applicable to complex mobile code environments such as J2SE.

Acknowledgement

The research was funded in part by a scholarship and an operating grant from the Natural Sciences and Engineering Research Council of Canada.

References

[1] 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`.

[2] Prem Devanbu and Stuart Stubblebine. Automated software verification with trusted hardware. In *Proceedings of the Twelfth International Conference on Automated Software Engineering*, 1997.

[3] 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`.

[4] 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`.

[5] 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, USA, November 1998. Also available at `http://www.cs.sfu.ca/~pwfong/personal/Pub/fse98.ps`.

[6] Philip W. L. Fong and Robert D. Cameron. Java proof linking with multiple classloaders. Technical Report SFU CMPT TR 2000-04, Simon Fraser University, 2000. Available at `ftp:// fas.sfu.ca/pub/cs/TR/2000/`.

[7] 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*, 2000. To appear. Also available as `http://www.cs.sfu.ca/~pwfong/personal/Pub/tosem2000.ps`.

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

[9] Gerwin Klein and Toblias 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`.

[10] 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. Also available at `http://java.sun.com/people/gbracha/classloader.ps`.

[11] 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`.

[12] Gary McGraw and Edward W. Felten. *Java Security: Hostile Applets, Holes, and Antidotes*. Wiley, 1997.

[13] 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`.

[14] 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`.

[15] 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`.

[16] Sun Microsystems. *Connected, Limited Device Configuration: Java 2 Platform Micro Edition*. Sun Microsystems, version 1.0 edition, May 2000. Available at `http://java.sun.com/products/cldc/`.

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