Proof Linking: An Architecture for Modular Verification of Dynamically-Linked Mobile Code

Philip W. L. Fong and Robert D. Cameron
{pwfong,cameron}@cs.sfu.ca
School of Computing Science
Simon Fraser University, B.C., Canada

Abstract

Security flaws are routinely discovered in commercial implementations of mobile code systems such as the Java Virtual Machine (JVM). Typical architectures for such systems exhibit complex interdependencies between the loader, the verifier, and the linker, making them difficult to craft, validate, and maintain. This reveals a software engineering challenge that is common to all mobile code systems in which a static verification phase is introduced before dynamic linking. In such systems, one has to articulate how loading, verification, and linking interact with each other, and how the three processes should be organized to address various security issues.

We propose a standard architecture for crafting mobile code verifiers, based on the concept of proof linking. This architecture modularizes the verification process and isolates the dependencies among the loader, verifier, and linker. We also formalize the process of proof linking and establish properties to which correct implementations must conform. As an example, we instantiate our architecture for the problem of Java bytecode verification and assess the correctness of this instantiation. Finally, we briefly discuss alternative mobile code verification architectures enabled by our modularization.

1 Introduction

Recent years have witnessed a significant growth of interest in mobile code in the form of active contents (e.g. applets) or code-on-demand [2]. Java [7] has emerged as an industrial standard for crafting mobile code of the mentioned forms. One of the crucial language features behind this success is Java's claim of security. As a mobile code hosting environment (e.g. browser) brings in programs of untrusted origin, it is imperative to guarantee that the untrusted program does not exploit the hosting machine in any undesirable way. As some authors have pointed out, Java security is type safety [13]. As long as the strict type rules are not violated, a piece of mobile code is guaranteed to behave in a well-mannered way. To enforce type safety, a bytecode verifier [10] is invoked prior to the execution of untrusted code.

To appear in the Sixth ACM SIGSOFT International Symposium on the Foundations of Software Engineering, Nov. 3–5, 1998, Orlando, Florida.

The verifier makes sure that all type rules are honored in the code. The verification algorithm is a very complex process including type checking and data-flow analysis. Due to the complexity of the verification algorithm, it is a nontrivial undertaking to produce an implementation that is correct. Because of this, various security breaches were discovered in several major implementations [8, 17]. Given that security is a critical concern for parties hosting mobile code, it becomes imperative to have a way of validating the bytecode verifier itself.

The current implementation from Sun and other vendors make the validation of the bytecode verifier very difficult.

1. **Interleaving logic.** The Sun bytecode verifier interleaves verification, loading, and linking. In the middle of verifying a class, new classes might have to be brought in to provide enough information for the verification to proceed. It is therefore difficult to single out a "verifier" module and validate it individually.

2. **Delocalized implementation.** The Sun bytecode verifier has a four-pass architecture. Pass one is performed at load-time; pass two and three are performed when a class is actually linked into the run-time environment; pass four is performed at run-time. Consequently, checks are spread all over the run-time system, and it is difficult to guarantee the completeness of the verification process.

It has been well-known in program understanding literature that interleaving and delocalized program plans lead to programs that are difficult to comprehend [15, 9]. This so-called "scattershot security" [13] renders validation of the verifier extremely difficult.

Not only this, interleaving and delocalization result in an overly tight coupling between the verifier and the rest of the Java run-time environment. Whenever a flaw is found in the verifier, the entire run-time environment (e.g. a browser) has to be replaced. Avoiding instability, many system administrators simply do not acquire the latest versions of browsers, leaving the users open to well-published security attacks.

Despite the above shortcomings, the mentioned four-pass architecture is designed for the purpose of making dynamic linking possible. A verification pass is applied to a class only when necessary. In this way, classes can be loaded in and linked together in an incremental fashion. Dynamic linking has two advantages:

1. **Flexibility in configuration management:** Because a class is linked in only when it is needed by

the run-time environment, one can make sure that the latest version of the class is used. This eliminates the need for relinking the software when a class is changed.

2. **Loading efficiency:** Applets/servlets start running as soon as the necessary code is loaded and linked. There is no need to load or verify a class if it is not needed in a particular execution session. This performance advantage is particularly obvious for classes with high static coupling but low dynamic coupling.

Although this ability to load and link classes dynamically has much to be desired, its current implementation introduces complex interactions among the class loader, the verifier, and the linker.

The above analysis reveals a software engineering challenge that is common to all dynamically-linked languages with high security concern. In particular, for mobile code systems in which each dynamically-loaded compilation unit is subject to a static verification phase before its execution, one has to determine how loading, verification, and linking interact with each other, and how to organize the three processes to address various security issues. In a traditional run-time environment, the loading and linking of program code is relatively modularized. Dynamic-linking in such an environment is common. However, as we start to introduce a verification phase into a dynamic-linking system, coupling among the loader, verifier, and linker increases dramatically. We argue that a well-designed mobile code architecture should localize all the security-related code into a stand-alone verifier module. In particular, it should allow one to (1) *craft* and *validate* the mobile code verifier as an individual engineering component independent of the run-time environment (e.g. browser, server, appletviewer), and (2) make the verifier a replaceable module that can be upgraded without changing the run-time environment.

In this paper, we propose a language-independent infrastructure for building dynamically-linked mobile code systems. Our design results in a run-time environment in which static verification is encapsulated in a stand-alone, replaceable module, while at the same time preserving dynamic-linking. The interaction among loading, verification, and linking is precisely isolated so that the correctness of their interaction can be established formally. In our proposal, given a compilation unit, the verifier computes a *conservative precondition* that will make the compilation unit pass the verification. It then submits the precondition to a *proof linker*. The proof linker maintains a database of all known facts of the loaded compilation units, plus a set of preconditions asserted by the verifier. Its role is to make sure that dynamically-loaded compilation units do not introduce into the database facts that violate preconditions already asserted by the verifier. This design results in a mobile code loading architecture in which the verifier module can be validated separately, and can be replaced easily when needed. Moreover, we formally state the sufficient conditions for the correctness of proof linking. The correctness conditions are expressed in terms of a partial ordering of linking events, and as properties of the database's data model. We demonstrate how one can formulate Java type checking in our framework, and we also establish the correctness of this formulation.

The architecture for modular verification is described in section 2. Section 3 develops a theoretical framework in which we can articulate the correctness of modular verification in the presence of dynamic linking. Section 4 applies the modularization to Java bytecode verification, and demonstrates how the correctness of modular verification can be established. In section 5, we briefly discuss alternative mobile code verification architectures enabled by our modularization. The paper concludes with a discussion of related work and potential extension to our work (section 6).

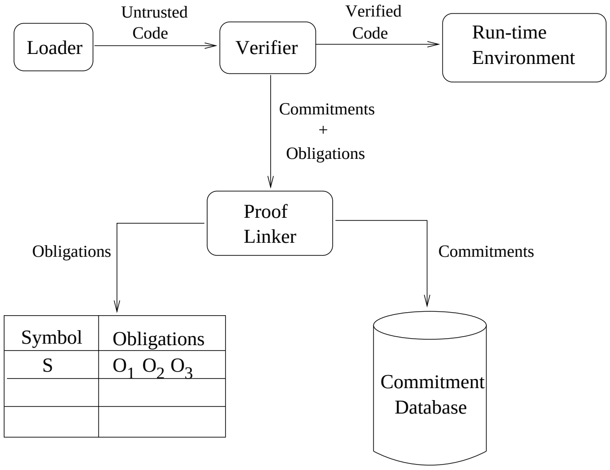

Figure 1: Modular Verification

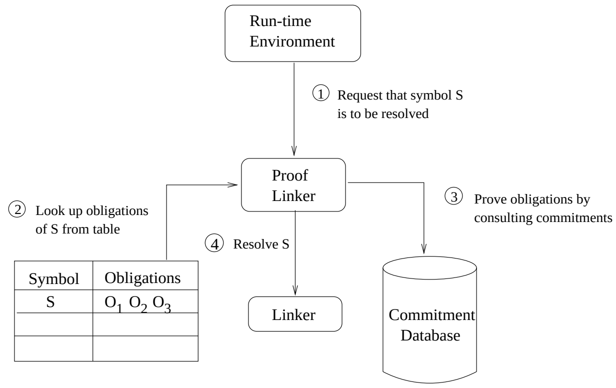

Figure 2: Proof Linking

2 A Dynamic-Linking Architecture

We assume that a program is composed of one or more compilation units. Each compilation unit is uniquely identified by a symbolic name. A compilation unit may contain references to other compilation units through the use of their symbolic names. When a program is executed, its compilation units are loaded, verified, and the symbolic references that refer to other compilation units are incrementally replaced by actual machine pointers.

In the dynamic-linking architecture we are about to propose, 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 poses the following challenge: *Verification requires knowledge of other compilation units which might not be loaded yet. How do we remove such dependencies while maintaining the integrity of the verification process?*

Modular verification. Figure 1 depicts the setup for modular verification. Untrusted compilation units are subjected to static verification after loading. The verifier might need the knowledge of another compilation unit in order to decide if the current compilation unit should be endorsed. Instead of recursively loading (or even verifying) the other compilation unit, the verifier computes a conservative *safety precondition* that will guarantee the safety of the compilation unit. The safety precondition is represented as a conjunctive set of database queries. For example, during the verification of a Java classfile, we might find out that an exception of class `ArithmeticException` is raised by the code in the classfile. Since the classfile is safe only if `ArithmeticException` is a subclass of the Java class `Throwable`, the verifier formulates the query[1] `?subclass(ArithmeticException, Throwable)`. The Java 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. Moreover, each of the queries describes the property of a specific symbolic reference. For example, the precondition `?subclass(ArithmeticException, Throwable)` is a property of class `ArithmeticException`. So, when a query is generated, it is attached to a symbolic reference not yet resolved. The query is said to be the *proof obligation* for resolving the symbolic reference. A global *obligation table* is used to store the obligations that are attached to symbolic references. Obligations represent conditions that must be met if the run-time system attempts to resolve the symbolic reference in the future.

In order for the run-time system to discharge the proof obligations, the verifier also computes, for each compilation unit, a set of clauses called *commitments*. The commitments are facts that describe the structural properties of the compilation unit. For example, during the verification of the Java classfile `ArithmeticException`, the verifier generates a commitment `!extends(ArithmeticException, Exception)` to indicate that `Exception` is the immediate superclass of `ArithmeticException`. The generated commitments are asserted into a global *commitment database*. When the proof obligations are to be discharged, the commitment database provides the set of facts against which the query can be evaluated.

Proof linking. The process by which the run-time system cross-validates the results of verifying different compilation units is called *proof linking*. Figure 2 describes the procedure for proof linking. When the run-time system needs to resolve a symbolic reference to a machine pointer, it sends the request to a *proof linker*. The proof linker looks up the obligations that have been attached to the symbol, and then posts them to the commitment database as deductive queries. If the queries are satisfied, then the linker will be activated to proceed with the resolution. Otherwise, a linking exception will be raised to signal this failure to endorse the interoperability of the compilation units.

To make proof linking more expressive, arbitrary logic programs can be provided as an initial theory in the commitment database. For example, recursive queries of the following form can be written to capture the transitive closure of subclassing relationship:

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

If the verifier asserts commitments

```
!extends(ArithmeticException, Exception)
```

and

```
!extends(Exception, Throwable)
```

then the obligation

```
?subclass(ArithmeticException, Throwable)
```

can be deduced.

Type-safe linking. Although it is ultimately our intention to apply the proof linking architecture to other types of mobile code verification problem, we restrict our attention in this paper primarily to the issue of type safety, with the specific example of Java byte code verification. In this context, we can characterize our architecture in the framework of Cardelli's theoretical treatment of type-safe linking [1]. In short, module verification in our system is comparable to Cardelli's intra-checking, while proof linking may be viewed as an incremental form of inter-checking. Our approach differs substantially in the treatment of typing environments. In particular, we replace the notion of an import environment as an input to intra-checking by our notion of obligations produced as output. In essence, obligations represent a logical specification of all allowable typing environments for which a module intra-checks. This technique is key to our implementation of lazy dynamic linking. The second distinction in the treatment of typing environments is that we replace the notion of export environments by the set of commitments produced during module verification. In this case, however, the replacement is a more-or-less a direct encoding of the typing environment in logical form.

Although the components of our proof-linking architecture may be related to Cardelli's theoretical framework for type-safe linking in this way, that framework provides no justification for our use of the the use of these components to achieve incremental dynamic linking. In essence, we may have confidence that our architecture is correct, assuming that all modules are loaded and linked before execution begins, but how can we be sure that it is safe to permit execution to proceed while some modules remain unlinked? This is a critical problem in validation of mobile code verification systems and is the topic of the next section.

3 Correctness of Incremental Proof Linking

In this section, we develop an argument for the correctness of incremental proof linking, while assuming that the overall logic of our verification method is correct *in the nonincremental sense*. Our argument is based on the following three conditions for correctness.

1. **Safety:** All obligations relevant to the safe execution of a code fragment are checked before that fragment is executed.
2. **Monotonicity:** Once an obligation is checked, no subsequently asserted commitment will contradict it.
3. **Completeness:** All commitments that may support an obligation will be generated before the obligation is checked. That is, safety of a program will not be prematurely ruled out.

Our goal is to establish a set of properties to which an implementation of proof linking must conform in order to ensure that these correctness conditions hold.

[1] To differentiate the various roles played by a predicate symbol, we prefix a query by a question mark ("?") and an assertion by an exclamation mark ("!").

3.1 A Model of Dynamic Linking

In the proof linking architecture, loading, verification, and linking are all atomic in the sense that, although they may occur concurrently, none of them include another recursively as a sub-operation, nor do they communicate directly with each other. Therefore, we assume that the run-time environment defines a set of *linking primitives*, each of which can be executed at most once during the life-time of the run-time environment. Each linking primitive performs a basic operation on a compilation unit or a symbolic reference thereof. Minimally, they should include the following for each compilation unit X:

load X**:** acquire compilation unit X.

verify X**:** verify compilation unit X.

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

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

One could see the linking primitives as events that happen asynchronously in a mobile code system. We assume that such events are then queued up in some synchronized event queue, waiting to be processed by the run-time system. The sequence of linking primitives 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.

Given a set P of linking primitives, a linking strategy $\sigma = (P, \leq)$ is a partial ordering of the linking events in P. Every implementation of a mobile code run-time system defines a linking strategy. The strategy expresses the order in which linking events are fired by the run-time system. More precisely, an execution trace τ is σ-conforming if the following hold: (1) all linking primitives in τ are from P, and (2) if y is in τ, then all $x \in P$ such that $x <_\sigma y$ must occur in τ before y. To say that a run-time system implements a linking strategy σ is to say that the run-time system guarantees that all possible execution traces are σ-conforming.

A linking strategy $\pi = (P, \leq_\pi)$ is a *substrategy* of another linking strategy $\sigma = (P, \leq_\sigma)$ iff $x \leq_\sigma y$ implies $x \leq_\pi y$ for every $x, y \in P$.

A strategy is *admissible* if the following properties hold: Given compilation units X and Y, we have

1. **Natural Progression Property:**

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

2. **Import-Checked Property:**

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

From now on we only consider admissible strategies.

3.2 Proof Linking

We assume that the execution of every linking primitive generates two sets. The first is a set of first-order clauses called *commitments*. Commitments describe the structure of the compilation unit processed by the linking primitive. The second is a set of *guards*. A guard is an ordered pair of a linking primitive and a set of first-order queries. The queries are said to be the *obligations* of the associated linking primitives[2].

ProofLinker(Γ_0):

01: $\Gamma \leftarrow \Gamma_0$; $\Omega[] \leftarrow \emptyset$;
02: Ready $\leftarrow \emptyset$; Checked $\leftarrow \emptyset$;
03: **while** ($\neg$ terminated()) **do**
04:   $p \leftarrow$ get-next-primitive();
05:   **for all** $o \in \Omega[p]$ **do**
06:     **if** ($\Gamma \vdash o$) **then**
07:       Checked $\leftarrow$ Checked $\cup$ { o };
08:     **else**
09:       raise exception in the thread that generated event p;
10:     **end if**
11:   **end for**
12:   Ready $\leftarrow$ Ready $\cup$ { p };
13:   $\langle$*Commitments*, *Guards*$\rangle \leftarrow$ execute(p);
14:   $\Gamma \leftarrow \Gamma \cup$ *Commitments*;
15:   **for all** $\langle$*Obligation*, *Primitive*$\rangle \in$ *Guards* **do**
16:     Ω[*Primitive*] $\leftarrow \Omega$[*Primitive*] $\cup$ {*Obligation*};
17:   **end for**
18: **end while**

Figure 3: The Proof-Linker Model Algorithm

Figure 3 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, a commitment database (Γ) and an obligation table (Ω). The commitment database is a 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 (Γ_0), and the obligation table is empty (line 1). The proof linker removes and executes linking primitives in the order specified by the linking strategy (line 4). When a linking primitive is removed from the event queue, its associated obligations are looked up from the obligation table (line 5). These queries are then verified by consulting the theory in the commitment database (line 6). If the obligations cannot be deduced from the theory, then a linking exception will be raised (line 9). Otherwise, the linking primitive is executed (line 13). The result of execution is a set of new commitments and a set of new guards. The commitments are asserted into the commitment database (line 14). The guards associate new obligations to linking primitives. These new associations are incorporated into the obligation table (line 15-17). The proof linker is then ready to accept a new request from the event queue. The proof linker terminates when the run-time environment terminates (line 3).

3.3 Formalization of Correctness Conditions

To formalize the correctness conditions of the proof linker, we have introduced two auxiliary variables into the listing in figure 3. "Checked" (in line 7) is the set of obligations that are already checked by line 6. "Ready" (in line 12) is the set of primitives that are ready for execution.

Given a fixed, admissible linking strategy σ, the proof linker is correct if the following conditions hold:

[2]Notice that we have generalized the role of obligations so that they are not only attached to the **resolve** primitive (as discussed in section 2), but can also be attached to arbitrary linking primitive. Such generalization simplifies the formulation of correctness conditions.

1. **Safety:** All obligations are checked before a primitive is executed. For any linking primitives x and y, if x may introduce the guard $\langle o, y\rangle$, then we require that $x <_\sigma y$. In summary, the following invariant should be true at all times:

$$\forall p \in \text{Ready} \ . \ \forall o \in \Omega[p] \ . \ o \in \text{Checked}$$

2. **Monotonicity:** Obligations may not be contradited by subsequently asserted commitments. In our system, we confine our database to definite Horn clause programs and definite queries. This avoids the problems that could arise if negation by failure were used. In summary, the following invariant should hold at all time:

$$\forall o \in \text{Checked} \ . \ \Gamma \vdash o$$

3. **Completeness:** A commitment c is said to be a support for obligation o if, possibly with other commitments asserted by some linking primitives, it forms a proof of o. If linking primitive x may assert a support for o, and if o may be attached to linking primitive y, then we require that $x <_\sigma y$. Thus, if the proof linker does not raise an exception for some σ-conforming execution trace τ, then it does not raise an exception for any σ-conforming execution trace τ' which is a permutation of τ.

In summary, the correctness of proof linking depends on (1) the linking strategy σ, (2) the kind of logic we are using, and, (3) the commitments and obligations returned by each linking primitive. Notice that the framework does not impose a strict policy on the linking strategy. Both eager linking (linking every compilation unit in an one-fell-swoop-manner) or lazy linking (linking a compilation unit only when its code is being executed), and anything in between, can be tailored to satisfy the sufficient conditions. In general, if two linking strategies satisfy the sufficient conditions, with one being the substrategy of the other, we usually prefer the latter.

4 An Example: Java Bytecode Verification

This section describes an instantiation of our modular verification framework. Specifically, we use Java bytecode verification as an example to illustrate various concepts we discussed in previous sections. We also give a proof (sketch) of the correctness of proof linking for this instantiation of Java bytecode verification.

The Java Linking Model. In Java, a class is a compilation unit. Besides class names, Java symbols may refer to members of a class. A member (a variable or a method) of a class is uniquely identified by the member name and its descriptor (type signature). The descriptor of a method specifies the type of the parameters and the return value. Class symbols and member symbols are resolved separately. We denote the linking primitive that resolves, in class X, the method M of class Y with descriptor S as "**resolve** Y::$M(S)$ **in** X".

We also introduce auxiliary primitives "**endow** Y" and "**endow** Y::$M(S)$". These primitives are introduced to impose desirable ordering among other primitives, and they do not correspond to any actual linking activities. In particular, complex obligations are attached to them, and supports for such obligations are forced to be asserted before the auxiliary primitives are fired[3].

We articulate an admissible strategy for Java linking. Besides modifying the Natural Progression Property and the Import-Checked Property to accommodate the introduction of new primitives, we also need to capture the linking dependencies peculiar to Java:

1. **Natural Progression Property:**

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

2. **Import-Checked Property:**

$$\textbf{endow } Y < \textbf{resolve } Y \textbf{ in } X$$

and also

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

3. **Subtype Dependency Property:** If Y is a superclass or a superinterface of X then

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

4. **Referential Dependency Property:** Sometimes, the knowledge of other classes are needed before we can correctly endow a class. If "**verify** X" asserts the commitment "`!relevant(`Y`, `X`::`$M(S)$`)`" then

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

In English, we need to verify all relevant classes before we endow a method symbol for resolution.

Commitments and Obligations for Java. In Java, only the "**verify** X" primitive generates commitments and obligations. Figure 4 describes the commitments generated by "**verify** X". Figure 5 describes the obligations generated by "**verify** X", together with the primitives to which the generated obligations are attached[4].

The Initial Theory. Figure 6 shows the clauses in the initial theory. For clarity, we have deliberately omitted the rules for handling array classes. Specification of such rules is straightforward.

Correctness of Java Proof Linking (Proof Sketch). The above arrangement satisfies the sufficient conditions for correct proof linking:

1. **Safety:** Only "**verify** X" generates obligations. According to figure 5, obligations are only attached to "**endow** X", "**endow** X::$M(S)$", "**resolve** Y **in** X", and "**resolve** Y::$M(S)$ **in** X. In any case, the Natural Progression Property and the Import-Checked Property guarantee that the contributors of obligations are always fired before the primitives to which the obligations are attached.

[3] The actual Java loading model further involves two additional primitives "**prepare** X" and "**initialize** X". In principle, one can always extend the current framework to include them if such refinement starts to interact with verification; but since they do not, we ignore them to facilitate our analysis.

[4] In principle, we could have formulated the commitments and obligations related to the checking of *resolution errors* [10, chapter 5]. But the detection of resolution errors is not strictly part of the verification process, so we ignore it here.

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

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

`!non_final(`X`)`
: X is not declared to be final.

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

`!implements(`X`, `Y`)`
: Y is a direct superinterface of X.

`!member(`X`, `M`, `S`)`
: M with descriptor S is a member of X.

`!public_member(`X`, `M`, `S`)`
: The member M with descriptor S is public in X.

`!protected_member(`X`, `M`, `S`)`
: The member M with descriptor S is protected in X.

`!private_member(`X`, `M`, `S`)`
: The member M with descriptor S is private in X.

`!default_member(`X`, `M`, `S`)`
: The member M with descriptor S has default access in X.

Figure 4: Commitments that may be asserted by **verify** X

2. **Monotonicity:** The initial theory and the obligations are all definite Horn clauses.

3. **Completeness:** Consider the obligation `?subclassible(`Y`)` attached to "**endow** X". Supports of the obligation are asserted by "**verify** Z" where Z is either Y or one of its superclass. Since Y is declared to be the superclass of X, the Subtype Dependency Property guarantees that Y and all its superclasses are verified before X is endowed. Therefore, the obligation is consistently established.

   Consider now the obligation `?assignment_compatible(`Y`, `Z`)` attached to "**endow** X::$M(S)$". Since both Y and Z are relevant to X::$M(S)$ if the the obligation is to be asserted, it follows from the Referential Dependency Property that the superclasses and superinterfaces of Y and Z are already verified when the obligation is tested. Thus, all the supports are already present, and the obligation can be consistently established.

   Using similar argument, one can prove the consistency of every obligation attached to every primitive.

Generating commitments and obligations. We have implemented a stand-alone Java bytecode verifier that generates the commitments and obligations in figure 4 and 5. In Sun's implementation of the verifier, merging of two classes Z_1 and Z_2 yields their most specific common superclass [10]. Computation of this superclass involves the recursive loading of all superclasses of Z_1 and Z_2. Our implementation avoids recursive loading by representing the most specific common superclass algebraically as $Z_1 \sqcap Z_2$. All the obligations will then be formulated in terms of these "aggregate" classes[5]. It is straightforward to introduce additional rules into the

[5]Such algebraic representation does not affect the termination of the data-flow analysis since only finitely many class symbols may appear in a method, and thus there are only a fixed number of meet expressions.

`?subclassible(`Y`)`
: **Target: endow** X
: **Intention:** Superclass Y of X can be subclassed.

`?class(`Y`)`
: **Target: endow** X
: **Intention:** Superclass Y of X should be a non-interface class.

`?interface(`Y`)`
: **Target: endow** X
: **Intention:** Superinterface Y of X should be an interface class.

`?class(`Y`)`
: **Target: resolve** Y **in** X
: **Intention:** Y should be a non-interface class.

`?interface(`Y`)`
: **Target: resolve** Y **in** X
: **Intention:** Y should be an interface class.

`?throwable(`Y`)`
: **Target: endow** X::$M(S)$
: **Intention:** Relevant to X::$M(S)$, Y is throwable.

`?subclass(`Y`, `Z`)`
: **Target: endow** X::$M(S)$
: **Intention:** Both relevant to X::$M(S)$, Y is a subclass of Z.

`?assignment_compatible(`Y`, `Z`)`
: **Target: endow** X::$M(S)$
: **Intention:** Both relevant to X::$M(S)$, Y is assignment compatible with Z.

`?invocation_compatible(`Y`, `Z`)`
: **Target: endow** X::$M(S)$
: **Intention:** Both relevant to X::$M(S)$, Y is invocation compatible with descriptor Z.

`?accessible_instance_member(`Y`, `M`, `S`, `X`, `Z`)`
: **Target: resolve** Y::$M(S)$ **in** X
: **Intention:** Relevant to X, Z can be used to reference the member Y::$M(S)$.

`?accessible_special_member(`Y`, `M`, `S`, `X`, `Z`)`
: **Target: resolve** Y::$M(S)$ **in** X
: **Intention:** Relevant to X, Z can be used to reference the special member Y::$M(S)$.

Figure 5: Obligations that may be asserted by **verify** X

initial theory to account for such change. For example, the introduction of the following clause will make the `subclass` rule work perfectly as before:

$$\texttt{subclass}(Y,\ Z_1 \sqcap Z_2)\ \texttt{:-}\ \texttt{subclass}(Y,\ Z_1),\ \texttt{subclass}(Y,\ Z_2).$$

The above implementation is an infrastructure for the work in [6]. We are now investigating how to incorporate a proof linker into a Java-enabled web browser.

5 Rethinking Verification

The modular verification model we propose here invites several natural extensions to mobile code verification. Firstly, since the verifier can be completely detached from the loader and the linker, it is then possible for the mobile code hosting technology (e.g. browser) and the verification technology to evolve independently of each other. In the context of Java applet verification, one may conceive that the verifier is made into some kind of replaceable module for any browser equipped with a proof linker. Third party vendors can specialize in producing highly secure verifier modules, while browser vendors can concentrate their efforts on producing more usable browsers. For example, a user may use Netscape Navigator with a Java verifier produced by Symantec. We believe this business model may yield higher quality mobile code hosting environments.

Secondly, modularization makes it feasible for verification to be performed remotely. The example in section 4 only requires 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 verifier can analyze a classfile, generate the corresponding commitments and obligations, and digitally sign the entire package. Upon receiving the package, a browser can perform a **verify** primitive that (1) validates the signature of the package, and (2) processes the commitments and obligations as if they are generated locally. Had we not modularized verification, remote verification would not be possible because the verification of one classfile will require the knowledge of other classfiles, which may not be accessible at the remote verifier's site. Combining modular verification with key-management technologies, and by employing a physically secure coprocessor to perform verification, Devanbu, Fong, and Stubblebine [6] produce a distributed mobile code verification architecture that has various security-related and configuration-management-related benefits. In this way, our work renders verification a service that can be offered by any third-party verification service provider.

Thirdly, modularization of verification provides interoperability among various *verification protocols*. A verification protocol specifies where and how verification is performed. There are at least three known verification protocols in the mobile code literature:

1. **Proof-on-demand**: The existing implementations of Java bytecode verification exemplify this protocol. Verification is performed dynamically whenever a classfile is linked into the run-time environment. The protocol introduces link-time overhead, but it allows dynamically-generated code to be verified properly.

2. **Proof-carrying code [14]**: Verification is performed remotely. A correctness proof is attached to a code module when it is shipped. Upon arrival at the execution site, the correctness proof is checked before execution is granted. Since proof checking is usually easier than proof generation, this protocol introduces less link-time overhead than proof-on-demand. Also, since proof generation is now performed once and for all by the developer of the code, one can afford to consider difficult-to-prove safety properties, including those that have to be verified with human assistance.

3. **Proof delegation [6]**: Code is passed to a trusted program analyzer, which certifies the correctness of the code, and then digitally signs it. Upon arrival at the execution site, verification is replaced by signature checking. This protocol is potentially the most efficient when the safety properties can be mechanically established.

Since each of these protocols has merit, it may be worthwhile to avoid premature commitment to a single one. For example, a mobile program may consist of some untrusted modules, some proof-carrying modules, and some remotely-certified modules, each from a different source. To make

```
subclassible('java/lang/Object').
subclassible(C) :-
    non_final(C), extends(C, D), subclassible(D).

subclass(C, D) :- extends(C, D).
subclass(C, E) :- extends(C, D), subclass(D, E).

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

superinterface(D, C) :- implements(C, D).
superinterface(E, C) :-
    implements(C, D), superinterface(E, D).
superinterface(E, C) :-
    extends(C, D), superinterface(E, D).

accessible_instance_member(C, M, S, _, _) :-
    public_member(C, M, S).
accessible_instance_member(C, M, S, D, I) :-
    protected_member(C, M, S), subclass(I, D).

accessible_special_member(C, '<init>', D, S, C) :-
    accessible_instance_member(C, '<init>', D, S, C).
accessible_special_member(C, M, D, C, _) :-
    private_member(C, M, D).
accessible_special_member(C, M, D, S, R) :-
    subclass(S, C),
    accessible_instance_member(C, M, D, S, R).

assignment_compatible(X, X).
assignment_compatible(S, T) :- subclass(S, T).
assignment_compatible(S, T) :- superinterface(T, S).

invocation_compatible(A, B) :-
    assignment_compatible(A, B).
```

Figure 6: The Initial Theory

this possible, we need to be able to combine the results of verification produced by multiple protocols. Proof linking provides an infrastructure for such interoperability.

6 Discussion and Related Work

Neither dynamic linking nor modular verification is new: dynamic linking is common in operating systems like UNIX and Windows; the notion of local certifiability is well known in software engineering [19]; composability of security features was studied as early as in the 1980's [11, 12]. Our main contribution lies in the proposal of an infrastructure that permits mudular verification in the presence of dynamic linking. Eliminating interleaving and delocalization, the resulting verifier can be understood and validated independent of the rest of the run-time system. This reduces the complexity of maintaining the verifier itself. We formalize the sufficient conditions under which loading, verification, and linking will interact correctly in this architecture. We apply the above to articulate the correctness of dynamic linking in an instantiation of Java bytecode verification. Moreover, our modularization leads naturally to several potential extensions of mobile code verification.

Dean pioneered the study of type-safe dynamic linking [4]. In order to make sure that the reflection facilities of Java do not produce type confusion, he formalized an approximation to the linking behavior of Sun's implementation of the Java run-time environment. He then formulates a sufficient condition, called consistent extension (of type environment), for type-safe linking. He established the correctness of his approximation using the PVS theorem prover [16]. While he focusses mainly on the articulartion of linking correctness in the presence of reflection facilities, we attempt to provide a modular architecture upon which one can dynamically link together results from static analysis of different compilation units. In our work, consistency of type extension is achieved by the monotonicity condition.

Cardelli [1] proposes a formal model for type-safe (static) linking in a simply-typed lambda calculus. In his model, program fragments to be linked to gether is called a linkset. Linking is modelled as a series of substitutions that preserve type safety invariants. Our **verify** primitive is equivalent to his *intra-checking*, while our **resolve** primitive could be seen as an incremental version of his *inter-checking*. The correctness of his model is also dependent (though implicitly) on some specific ordering of substitution steps [1, Lemma 3-3 & Section 6]. Extension of this framework to incremental dynamic linking would be an interesting area for further work.

One may argue that, by making the verifier module replaceable, we are simply transferring the point of attack to the proof linker. We believe that this does not create a problem because (1) the correctness of proof linking can be established fairly mechanically, (2) the architecture of the proof linker is extremely simple (in our Java example, even a pure Prolog interpreter will do the job), and (3) the underlying technology, deductive query evaluation, is extremely well-studied in fields like proof theory, logic programming, and database theory; many well-tested implementations are around.

Although the first application of our framework, Java bytecode verification, is mainly type checking, we are exploring the application of our framework to the verification of other security properties like confidentiality [5, 18] and integrity [3]. In the long run, we would like to formally characterize the kind of program properties that allow modular verification in the manner prescribed by this paper.

Acknowledgement

The research was funded in part by a scholarship and an operating grant from the Natural Sciences and Engineering Research Council of Canada. We thank Prem Devanbu for introducing us to the wonderful world of Java bytecode verification, and the anonymous referees of FSE-6 for their comments.

References

[1] Luca Cardelli. Program fragments, linking, and modularization. In *Proceedings of the 24th Annual ACM Symposium on Principles of Programming Languages (POPL'97)*, Paris, France, January 1997. Also available at `http://research.microsoft.com/research/cambridge/luca/Papers/Linking.ps`.

[2] 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.

[3] D. Clark and D. Wilson. A comparison of commercial and military computer security policies. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 184–194, 1987.

[4] Drew Dean. The security of static typing with dynamic linking. In *Proceedings of the Fourth ACM Conference on Computer and Communication Security*, Zurich, Switzerland, April 1997. Also available at `http://www.CS.Princeton.EDU/sip/pub/ccs4.html`.

[5] Dorothy E. Denning and Peter J. Denning. Certification of Programs for Secure Information Flow. *Communications of the ACM*, 20(7):504–513, July 1977.

[6] 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. Available at `http://seclab.cs.ucdavis.edu/~devanbu/icse98.ps`.

[7] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[8] The Secure Internet Programming Group. History of the group, 1997. Availablat at `http://www.cs.princeton.edu/sip/History.html`.

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

[10] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison Wesley, 1997.

[11] Daryl McCullough. Specification for Multi-Level Security and a Hook-Up Property. In *Proceedings for the IEEE Symposium on Security and Privacy*, pages 161–166, 1987.

[12] Daryl McCullough. Noninterference and the Composability of Security Properties. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 177–186, 1988.

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

[14] 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.cs.cmu.edu/~necula/popl97.ps.gz`.

[15] Spencer Rugaber, Kurt Stirewalt, and Linda W. Wills. The Interleaving Problem in Program Understanding. In *2nd Working Conference on Reverse Engineering*, pages 166–175, Toronto, Ontario, Canada, July 1995.

[16] SRI International Computer Science Laboratory. The PVS verification system. Available at `http://www.csl.sri.com/pvs.html`.

[17] The Kimera Team. Security Flaws in Java Implementations, 1997. Available at `http://kimera.cs.washington.edu/flaws/`.

[18] 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, April 1997. Also available at `http://www.cs.nps.navy.mil/research/languages/papers/atsc/tapsoft97.ps.Z`.

[19] B. W. Weide and J. E. Hollingsworth. Scalability of Reuse Technology to Large Systems Requires Local Certifiability. In *Proceedings of the 5th Annual Workshop on Software Reuse*, 1992.