Lightweight Bytecode Verification*

Eva Rose[†] and Kristoffer Høgsbro Rose
LIP, ENS-Lyon; 46, Allee d'Italie; 69364 Lyon 7; France
E-mail: `Eva.Rose@ENS-Lyon.FR` and `Kristoffer.Rose@ENS-Lyon.FR`

September 21, 1998

Abstract

Java Bytecode Verification ensures that bytecode can be trusted to avoid various dynamic runtime errors, but it requires an analysis which is currently unrealistic to implement on systems with very sparse resources such as Sun's Java Cards featuring a reduced Java virtual machine embedded on a smart-card (credit card with an integrated microprocessor). Commonly it is assumed that verification has to be performed off-card and shipped by means of cryptographic protection techniques, *e.g.*, digital signatures, to ensure that the bytecode is not tampered with after it has been verified. These techniques, however, create a *single point of failure trust* because of their dependence on a secret (private) key. This is particularly serious when dealing with embedded systems with a very wide distribution such as Java Cards.

This paper proposes an alternative solution by splitting the verification process in two parts: an off-card part, where sufficient verification information is constructed as a *verification certificate* to be sent together with the bytecode, and an on-card part, where the remaining verification is performed as a check of the code and the certificate, the *lightweight verification*, which has the advantage of running in a reasonable almost constant space (and almost linear time). We give a formal specification of the two components for a subset of Sun's *Java Card Language*, sufficiently strong to write non-trivial down-loadable applets. We prove that the technique is *tamper-proof* by showing it both *sound* and *complete* with respect to standard bytecode verification. Finally we argue that our approach is safer than usual bytecode verification because it isolates a non-complex safety-critical component, the *bytecode checker*.

---

*[*] **Extended abstract for FUJ '98.***

[†] Funded by GIE Dyade, a joint INRIA/Bull alliance.

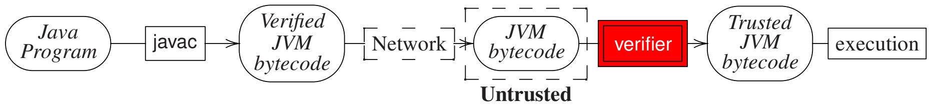

Figure 1: Standard bytecode verification.

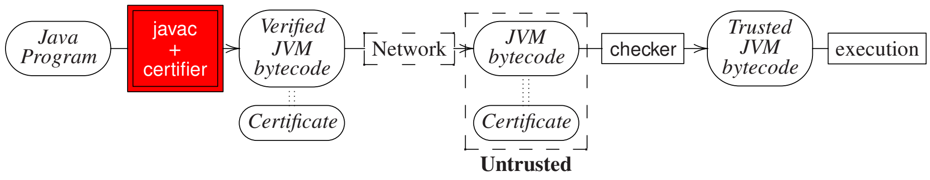

Figure 2: *Lightweight* bytecode verification.

1 Introduction

There is a growing interest for using personalized smart cards, such as Sun's Java Card (Sun 1997), because of the independence they offer their users by having an on-card microprocessor. In particular in combination with the fast and flexible information exchange, offered by the Internet and other global networks. However, the current standard (Sun 1997) does not support dynamic downloading of code onto Java Cards since they represent *too small a runtime environment*, typically 16 kB ROM and $\frac{1}{2}$ kB RAM, for hosting and running any of the Java security components which are normally applied by the code receiver to ensure that downloaded code can be trusted. The Java bytecode verifier is considered as the basic of these components since it ensures the received code against various, basic dynamic errors (for a comprehensive description of the verifier and the Java virtual machine, JVM, we refer to the official specification, Lindholm & Yellin 1996). Figure 1 illustrates where the bytecode verifier traditionally works during a bytecode transfer over a network.

Bytecode verification is inherently a data-flow analysis problem applied to individual Java methods (Lindholm & Yellin 1996). So even though some optimizations are possible, it requires that constraints are collected and resolved for all program points. Clearly, bytecode verification is not realizable on a small system in its current form. It is commonly assumed that downloading bytecode to a Java Card requires that verification is performed off-card, establishing trust in the bytecode by cryptographic protection techniques, *e.g.*, digital signatures, to ensure that the bytecode is not tampered with after it has been verified. These techniques, however, create a *single point of failure trust* because of their dependence on a secret (private) key. This is not only a problem when the embedded systems are widely distributed, it is also unfortunate when designing cards where the code producer and receiver do not necessarily trust the same sources (or each other).

In the full paper we explain "proof carrying code" (PCC, Necula 1997).

The approach chosen in this paper is inspired by the work on PCC. We propose that bytecode, which has been verified by the code provider, is sent with a *verification certificate*, which makes it possible at the receiver's side simply to perform a check of this information against the bytecode actually received, in order to ensure that it is verifiable. We call this check for *lightweight verification* because for ordinary methods the receiver can perform it with modest storage requirements. Figure 2 illustrates how the bulk of the verification is moved into the compilation system at the cost of adding the certificate to the bytecode. (We envision extending the existing bytecode verifier in the javac compiler to produce certificates.)

The bytecode verifier assigns a description of the types of each stack element and local variable at each instruction. Formally, this means that bytecode verification can be seen as *type reconstruction*. From this point of view the certificate becomes a *typing* and lightweight bytecode verifier a *type checker* (Rose 1998). We will exploit these observations to give a complete formal description of the lightweight bytecode verifier as well as a certificate. In particular we prove that lightweight bytecode verification does not introduce any security breaches, *i.e.*, is tamper-proof. This is obtained by showing that lightweight verification is

Sound: If a certified bytecode method is accepted by the checker, then the same method will pass standard bytecode verification.

Complete: If a method can be bytecode verified the ordinary way, then the certifier can build a certificate.

As specification language, we use Natural Semantics (Kahn 1987) because it is particularly suitable for specifications of modular inductive descriptions and proofs of properties.

Finally, we assume that our reader is familiar with the Java language (Gosling, Joy & Steele 1996) and has some knowledge of the virtual machine (Lindholm & Yellin 1996).

Notation. We will apply the Backus-Naur Form (BNF) style $::=$ to express structural inductive sort definitions with $|$ to separate alternatives. When inductive structure is not needed, we simply write $=$. We use a special font for constants and constructors, *e.g.*, methsig. Moreover, we will use a * mark to describe *sequences* of zero or more elements of the marked sort. In particular, sequences are constructed in equations with an infix associative "$\cdot$" (dot), or if empty, simply written "ϵ".

Overview. In Section 2 we explain the Java subset that we use. In Section 3 we formalize the elements of (standard) verification used in the (standard) verification inference rules presented in Section 4. Based on these we present *lightweight* verification in Section 5, including a formal statement of equivalence of the usual bytecode verification and the combination of the lightweight certification and verification. Finally, in Section 6, we conclude as well as discuss related work and future directions

2 The Virtual Machine Subset

We consider a subset of the Java Virtual Machine (JVM) which is strong enough to treat a non-trivial subset of the Java Card language (Sun 1997). (In Rose (1998), a compiler defining our JVM subset from the source subset is specified.) A subset which focuses on object-oriented features, notably object creation, dynamic method dispatch, and instance variables, as well as exceptions. The later deserves special mentioning since the `finally` clause, in order to compile to compact bytecode, is the only reason why subroutine call instructions, jsr and ret, are required at the machine level (Lindholm & Yellin 1996). So, since they represent a delicate (Stata & Abadi 1998) but not essential part of the Virtual Machine, and since we do not believe that the complexity of handling it outweighs the penalty of code duplication when compiling without, they have been omitted from our JVM subset.

Otherwise the most significant features which we have chosen to omit are static declarations, arrays, interfaces and packages, access modifiers (`protected` and `private`), and type structures (as provided by the Java `package` and `interface` constructions), and hence the corresponding bytecode support. For reasons of clarity we have moreover excluded most of the "short forms" of instructions used to make the bytecode shorter (and slightly faster). We remark that the source subset which has dictated our JVM subset is close to the Java subset BALI of Nipkow & von Oheimb (1998); even if they were developed independently this is hardly surprising because both sublanguages were designed with the purpose of writing *non-trivial object-oriented programs*.

An important remark is that JVM bytecode is transmitted in units called *class files* corresponding to all the data needed to execute a Java `class`. However, no class file format exists yet for the Java Card subset, so we have chosen to just model the two components that are needed to perform bytecode verification: the bytecode *instruction sequence* of the method to verify and the *constant pool* with constant values, external reference names, and type information. A formalization of these components is given below, followed by a formalization of the *dynamic load context*, needed to formalize bytecode verification in the next sections.

2.1 Bytecode Instruction Sequences

The translation of our non-trivial Java subset into JVM results in a JVM subset which consists of the following 24 JVM instructions:

$$\begin{aligned}\mathtt{i} \in \mathit{Instruction} ::= {} & \mathsf{iconst\_0} \mid \mathsf{iconst\_1} \mid \mathsf{aconst\_null} \\ & \mid \mathsf{dup} \mid \mathsf{pop} \mid \mathsf{iadd} \mid \mathsf{isub}\end{aligned}$$

$$
\begin{aligned}
&\mid \mathsf{iload}[n] \mid \mathsf{aload}[n] \mid \mathsf{istore}[n] \mid \mathsf{astore}[n] \\
&\mid \mathsf{checkcast}[n] \mid \mathsf{ldc\_w}[n] \mid \mathsf{new}[n] \\
&\mid \mathsf{putfield}[n] \mid \mathsf{getfield}[n] \\
&\mid \mathsf{invokevirtual}[n] \mid \mathsf{invokespecial}[n] \\
&\mid \mathsf{goto}[n] \mid \mathsf{ifle}[n] \mid \mathsf{ifnull}[n] \\
&\mid \mathsf{return} \mid \mathsf{ireturn} \mid \mathsf{areturn}
\end{aligned}
$$

where i in the sequel stands for a JVM *Instruction.* (For a complete description of the translation we refer to Rose 1998).

We recall that bytecode verification is officially described *per method* though applied for a given class file. Single methods are hence the smallest unit that can be verified. Abstractly, a method at the JVM level is described as an instruction sequence. With the sort notation given in the introduction:

$$c \in \mathit{MethodByteCode} ::= i^*$$

where each instruction is identified by a *program point*, which is their (byte) position or address, starting from 0, within that code sequence.

$$
\begin{aligned}
pp &\in \mathit{PPoints} = \mathbf{N} \\
PP &\in \mathit{PPointSet} = \mathbf{P}(\mathit{PPoints})
\end{aligned}
$$

where $\mathbf{P}$ denotes the power set of the parameter set.

2.2 The Constant Pool

The constant pool contains constant arguments of bytecode instructions that are too large to be contained in the instructions themselves, called *items*. For our subset this is integer constants (for example the argument of ldc_w), references to fields (the argument of putfield and getfield), and references to methods (the argument of invokevirtual and invokespecial). References are stored as the class name of the referenced class.

$$cp \in \mathit{ConstPool} ::= it^*$$

We write $cp(n)$ for the nth item in the constant pool cp from the left (counting from 0).

$$
\begin{aligned}
it &\in \mathit{Item} ::= int \mid fref \mid mref \\
int &\in \mathit{Integer} = \mathbf{Z}
\end{aligned}
$$

References to fields are tagged triples with the class name, field name, and field type:

$$fref \in \mathit{FieldRef} ::= \mathsf{fieldref}(cid, id, mt)$$

$$\mathtt{cid} \in \mathit{ClassId}$$
$$\mathtt{mt} \in \mathit{MType} ::= \mathtt{int} \mid \mathtt{cid}$$

We leave the representation of class names cid unspecified; *MType* stands for "machine type" (JVM uses a compact string representation for this information).

Finally references to methods are also tagged triples:

$$\mathtt{mref} \in \mathit{MethodRef} ::= \mathsf{methref}(\mathtt{cid}, \mathtt{msig}, \mathtt{rmt})$$
$$\mathtt{msig} \in \mathit{MethodSig} ::= \mathsf{methsig}(\mathtt{id}, \mathtt{mt}^*)$$
$$\mathtt{rmt} \in \mathit{RMType} ::= \mathtt{void} \mid \mathtt{mt}$$

(*RMType* stands for "returned machine type" for a method, which may be void for a non-returning method.)

2.3 Dynamic Load Context

A dynamic load context contains the class hierarchy and the current class name. When a class is dynamically loaded this means that the class is already known to the execution context where it is needed. In practice this entails that its position in the *class hierarchy* is already known in the dynamic load context. This information is important for the bytecode verifier because it is needed to verify whether implicit type-casts, *e.g.*, when passing parameters, are legal. For simplicity we will assume that the entire class hierarchy is known to the dynamic class load context in the form of a subclass relation.

$$\leq: \in \mathit{SubClass} = \mathbf{P}(\mathit{ClassId} \times \mathit{ClassId})$$

So an $\leq:$ is a set where each element is a pair $(\mathtt{cid}_1, \mathtt{cid}_2)$ where $\mathtt{cid}_1$ is a subclass of $\mathtt{cid}_2$. Finally, the dynamic load context should contain the name of the class being loaded.

A more exact formalization of dynamic class loading must include a formalization of the notion of *resolution*. This is beyond the scope of this paper because resolution is not yet defined for Java Cards; instead we assume that all resolution has happened, *i.e.*, that the entire class hierachy is available and that no false types are given for any method (resolution and class loading is discussed for the full Java language by Jensen, Le Métayer & Thorn 1998).

3 Verification vs Checking

Bytecode verification proceeds by assigning a description of the type of the stack and local variable table, the "current frame", at each program point. We call this a *frame type*. Verification succeeds if it can be checked that

- all instructions have the arguments they need in the frame type just before their execution, and

- all instructions guarantee that the state just after execution has a frame type that is “compatible” with the frame type assigned to the next program point.
- Entry into and exit from a method happens in a way that is compatible with the type of the method.

“Compatibility” here means that it is acceptable that the frame description just after an instruction has more information than is actually needed. We formalize this by giving the definition of frame types, the approximation relation, and the result of a verification.

3.1 Frame Types

For each instruction the verifier needs to check that all the needed arguments are available. If the argument is in the constant pool, then that is checked. Otherwise it is a component of the current frame. We can then check that it is always available at execution time by verifying that the corresponding frame component's type is right.

$$\begin{aligned} \mathsf{ft} &\in \mathit{FrameType} = \mathit{StackType} \times \mathit{LvType} \\ \mathsf{st} &\in \mathit{StackType} = \mathit{MType}_\perp{}^* \\ \mathsf{lvt} &\in \mathit{LvType} = \mathit{MType}_\perp{}^* \end{aligned}$$

(the $_\perp$ annotation means that a special $\perp$ (“bottom”) element has been added, representing “no information available”). The local variables can be indexed individually: $\mathsf{lvt}(n)$ is the nth local variable type of lvt, from the left, counting from 0.

The assignment of a frame type to each program point is done by a map

$$\mathsf{md} \in \mathit{MDescr} = \mathit{PPoints} \overset{\text{fin}}{\longrightarrow} \mathit{FrameType}$$

(the $\overset{\text{fin}}{\longrightarrow}$ arrow indicates that it is a finite function from program points to frame types).

3.2 Frame Type Approximation

If, for example, $\mathsf{md}(25) = \big((\perp, \mathtt{int}), (\mathsf{cid})\big)$, then we can imply that, before the instruction at program point 25, the stack has one element that can be any type and one `int` element (at the top), and that there is exactly one local variable which is reference to an instance of the cid class or a subclass of it. So we can execute a pop or an ifle instruction, for example, since they have what they need on the stack, but we can neither execute an ifnull instruction, since that requires the top stack element to be an object reference, nor an iadd instruction because it needs two `int` elements on the stack.

To capture this we define a “less defined” approximation relation $\sqsubseteq$ on frame types that expresses when the right-hand frame type has sufficient information to guarantee that the requirements represented by the left-hand one are satisfied.

On simple types the relation is easy: it is defined by the inference rules

$$\frac{\mathtt{cid}_2 \leq: \mathtt{cid}_1}{\mathtt{cid}_1 \sqsubseteq \mathtt{cid}_2} \qquad \frac{}{\bot \sqsubseteq \mathtt{mt}}$$

Notice how "less defined" between classes means "superclass" in object-oriented terminology, extended with a "bottom" element which is an artificial type that is less defined than any other type. In particular for any class $\mathtt{c}$ we have that $\bot \sqsubset$ `Object` $\sqsubseteq \mathtt{c}$ ($\sqsubset$ is the restriction to non-equal types). It is extended point-wise to frame types, *i.e.*,

$$\frac{\mathtt{mt}_{111} \sqsubseteq \mathtt{mt}_{211} \ \cdots \ \mathtt{mt}_{12m_1} \sqsubseteq \mathtt{mt}_{22m_2}}{\mathtt{ft}_1 \sqsubseteq \mathtt{ft}_2}$$

where

$$\mathtt{ft}_i = \langle \mathtt{mt}_{i11} \cdots \mathtt{mt}_{i1n_i}, \mathtt{mt}_{i21} \cdots \mathtt{mt}_{i2m_i} \rangle,\ i \in \{1,2\}$$
$$n_1 = n_2 \quad \wedge \quad m_1 = m_2$$

(two frame types with the same stack (type) size and the same number of local variable (types) can be compared if all the individual types can be compared positionally).

3.3 Verification

Now we can express verification. Only one thing needs to be added: verification must determine a maximal size of the stack and the number of local variables.

$$\mathtt{ms} \in \mathit{MaxStack} = \mathbf{N}$$
$$\mathtt{mlv} \in \mathit{MaxLVar} = \mathbf{N}$$

Thus standard bytecode verification can be specified as a natural semantics judgment of the form

$$\leq:, \mathtt{cid}, \mathtt{cp} \vdash \mathtt{msig}, \mathtt{c}, \mathtt{ms}, \mathtt{mlv} : \mathtt{md} \tag{BV}$$

understanding the proof of the judgment as a way to construct $\mathtt{md}$. The judgment follows the convention that the class *context* is to the left of the $\vdash$, the *local* method information is between the $\vdash$ and the : after which the *derived* type descriptor is written.

4 Bytecode Checking

If, however, the "result" $\mathtt{md}$ of a verification is already available, then (BV) can be read as a judgment for merely checking that the verification was correct. The challenge is not to change the reading – from a logical viewpoint there is no difference, in fact. But it is easier to specify this new interpretation of (BV) *constructively*, and this is what we will present in this section. (At the end we comment on how a constructive version of the generic verifier can be described.)

4.1 Method Check

Checking a method is simply checking that each of its instructions have their constraints satisfied and that the stack and local variable sizes are sufficiently large.

$$\frac{\langle \leq:, \mathsf{cp}, \mathsf{ms}, \mathsf{mlv}, \mathsf{rmt}\rangle \vdash \emptyset, 0, \mathsf{c} : \mathsf{md} \rightarrow \mathsf{PP}}{\leq:, \mathsf{cid}, \mathsf{cp} \vdash \mathsf{msig}, \mathsf{c}, \mathsf{ms}, \mathsf{mlv} : \mathsf{md}} \tag{1}$$

where

$$\begin{aligned} \mathsf{msig} &= \mathsf{methsig}(\mathsf{rmt}, \mathsf{mt}_1 \ldots \mathsf{mt}_n) \\ \mathsf{lvt} &= \{0 \mapsto \mathsf{cid}, 1 \mapsto \mathsf{mt}_1, \ldots, n \mapsto \mathsf{mt}_n, \\ &\qquad n{+}1 \mapsto \bot, \ldots, \mathsf{mlv}{-}1 \mapsto \bot\} \\ \mathsf{md}(0) &\sqsubseteq \langle \epsilon, \mathsf{lvt}\rangle \\ \mathrm{Dom}(\mathsf{md}) &= \mathsf{PP} \end{aligned}$$

The premise checks all the instructions in c (since none are already checked in $\emptyset$), starting from the instruction at program point 0 requiring the initial frame type $\mathsf{md}(0)$ to be less defined than the initial frame type established by JVM, $\langle \epsilon, \mathsf{lvt}\rangle$. For each we must verify that the information in md is *correct*. All the verified instructions are collected in PP such that we can check at the end that the information in md is complete.

4.2 Sequences

We start by naming the static information about the available methods.

$$\Xi \in \mathit{StatMethInfo} ::= \langle \leq:, \mathsf{cp}, \mathsf{ms}, \mathsf{mlv}, \mathsf{rmt}\rangle$$

In the rules below we will always assume that Ξ consists of those same components, *i.e.*, $\langle \leq:, \mathsf{cp}, \mathsf{ms}, \mathsf{mlv}, \mathsf{rmt}\rangle$.

Checking of sequence statements is expressed as follows:

$$\frac{\begin{array}{l}\Xi \vdash \mathsf{pp}, \mathsf{i} : \mathsf{md} \rightarrow \mathsf{pp}' \\ \Xi \vdash \mathsf{PP}', \mathsf{pp}', \mathsf{c} : \mathsf{md} \rightarrow \mathsf{PP}''\end{array}}{\Xi \vdash \mathsf{PP}, \mathsf{pp}, \mathsf{i} \cdot \mathsf{c} : \mathsf{md} \rightarrow \mathsf{PP}''} \quad \mathsf{PP}' = \mathsf{PP} \cup \{\mathsf{pp}\} \tag{2}$$

$$\frac{}{\Xi \vdash \mathsf{PP}, \mathsf{pp}, \epsilon : \mathsf{md} \rightarrow \mathsf{PP}} \tag{3}$$

The rules merely check that the constraints of each instruction are satisfied until none are left, each time adding the program point of the checked instruction to the accumulating set, *i.e.*, PP' always contains one more program point than PP, namely pp. Each rule below will advance the program point counter as appropriate for the instruction, *i.e.*, set pp' to be the program point of the subsequent instruction in c (thus inductively ensuring that $\mathsf{pp} \notin \mathsf{PP}$).

$\mathtt{i}$	$\mathtt{st}$	$\mathtt{st}'$
iconst_0	$\mathtt{st}$	$\mathtt{st}\cdot\mathtt{int}$
iconst_1	$\mathtt{st}$	$\mathtt{st}\cdot\mathtt{int}$
aconst_null	$\mathtt{st}$	$\mathtt{st}\cdot\mathtt{Object}$
dup	$\mathtt{st}_1\cdot\mathtt{mt}$	$\mathtt{st}_1\cdot\mathtt{mt}\cdot\mathtt{mt}$
pop	$\mathtt{st}_1\cdot\mathtt{mt}$	$\mathtt{st}_1$
iadd	$\mathtt{st}_1\cdot\mathtt{int}\cdot\mathtt{int}$	$\mathtt{st}_1\cdot\mathtt{int}$
isub	$\mathtt{st}_1\cdot\mathtt{int}\cdot\mathtt{int}$	$\mathtt{st}_1\cdot\mathtt{int}$

(a)

$\mathtt{i}$	$\mathtt{st}$	$\mathtt{st}'$	$\mathtt{it}$
putfield[$\mathtt{n}$]	$\mathtt{st}'\cdot\mathtt{cid}\cdot\mathtt{mt}$	$\mathtt{st}'$	fieldref($\mathtt{cid}'$, $\mathtt{id}$, $\mathtt{mt}$)
getfield[$\mathtt{n}$]	$\mathtt{st}_1\cdot\mathtt{cid}$	$\mathtt{st}_1\cdot\mathtt{mt}$	fieldref($\mathtt{cid}'$, $\mathtt{id}$, $\mathtt{mt}$)
invokevirtual[$\mathtt{n}$]	$\mathtt{st}'\cdot\mathtt{cid}\cdot\mathtt{mt}_1\cdots\mathtt{mt}_j$	$\mathtt{st}'$	methref($\mathtt{cid}'$, methsig($\_$, $\mathtt{mt}_1,\ldots,\mathtt{mt}_j$), $\mathtt{void}$)
invokevirtual[$\mathtt{n}$]	$\mathtt{st}_1\cdot\mathtt{cid}\cdot\mathtt{mt}_1\cdots\mathtt{mt}_j$	$\mathtt{st}_1\cdot\mathtt{mt}$	methref($\mathtt{cid}'$, methsig($\_$, $\mathtt{mt}_1,\ldots,\mathtt{mt}_j$), $\mathtt{mt}$)
invokespecial[$\mathtt{n}$]	$\mathtt{st}'\cdot\mathtt{cid}\cdot\mathtt{mt}_1\cdots\mathtt{mt}_j$	$\mathtt{st}'$	methref($\mathtt{cid}'$, methsig($\_$, $\mathtt{mt}_1,\ldots,\mathtt{mt}_j$), $\mathtt{void}$)
invokespecial[$\mathtt{n}$]	$\mathtt{st}_1\cdot\mathtt{cid}\cdot\mathtt{mt}_1\cdots\mathtt{mt}_j$	$\mathtt{st}_1\cdot\mathtt{mt}$	methref($\mathtt{cid}'$, methsig($\_$, $\mathtt{mt}_1,\ldots,\mathtt{mt}_j$), $\mathtt{mt}$)

(d)

$\mathtt{i}$	$\mathtt{st}$
ifle[$\mathtt{n}$]	$\mathtt{st}'\cdot\mathtt{int}$
ifnull[$\mathtt{n}$]	$\mathtt{st}'\cdot\mathtt{cid}$

(e)

$\mathtt{i}$	$\mathtt{st}$	Condition
return	$\mathtt{st}_1$	$\mathtt{rmt} = \mathtt{void}$
ireturn	$\mathtt{st}_1\cdot\mathtt{int}$	$\mathtt{rmt} = \mathtt{int}$
areturn	$\mathtt{st}_1\cdot\mathtt{cid}$	$\mathtt{cid} \leq: \mathtt{rmt}$

(f)

Figure 3: Instruction-specific constraints. *Complete in full paper.*

4.3 Simple Stack Instructions

These are the simplest instructions, with just constraints on the stack.

$$\frac{}{\Xi \vdash \mathtt{pp}, \mathtt{i} : \mathtt{md} \rightarrow \mathtt{pp}'} \tag{4}$$

where

$$\begin{aligned} \mathtt{pp}' &= \mathtt{pp} + 1 \\ \mathtt{md}(\mathtt{pp}) &= \langle \mathtt{st}, \mathtt{lvt} \rangle \\ \mathtt{ft}' &= \left\langle \mathtt{st}', \mathtt{lvt} \right\rangle \\ \mathtt{md}(\mathtt{pp}') &\sqsubseteq \mathtt{ft}' \\ \left| \mathtt{st}' \right| &< \mathtt{ms} \end{aligned}$$

with specific instruction constraints in figure 3(a). (One obtains the exact rule for a particular instruction by adding each table entry as an equality, *i.e.*, one rule has the three extra side conditions $\mathtt{i} = \mathsf{dup}$, $\mathtt{st} = \mathtt{st}_1 \cdot \mathtt{mt}$, and $\mathtt{st}' = \mathtt{st}_1 \cdot \mathtt{mt} \cdot \mathtt{mt}$.)

The full paper has similar rules for all the instructions but in this extended abstract we just include the four instruction groups that are most interesting from a verification viewpoint, omitting instructions accessing the local variables and constant pool, and the goto *instruction.*

4.4 Object-oriented Instructions

These are more interesting, handling access to instance fields and methods.

$$\frac{}{\Xi \vdash \mathtt{pp}, \mathtt{i} : \mathtt{md} \rightarrow \mathtt{pp}'} \tag{5}$$

where

$$\begin{aligned} \mathtt{pp}' &= \mathtt{pp} + 3 \\ \mathtt{md}(\mathtt{pp}) &= \langle \mathtt{st}, \mathtt{lvt} \rangle \\ \mathtt{ft}' &= \left\langle \mathtt{st}', \mathtt{lvt}' \right\rangle \\ \mathtt{md}(\mathtt{pp}') &\sqsubseteq \mathtt{ft}' \\ \mathtt{cp}(\mathtt{n}) &= \mathtt{it} \\ \mathtt{cid} &\leq: \mathtt{cid}' \\ \left| \mathtt{st}' \right| &< \mathtt{ms} \end{aligned}$$

with instruction specific constraints in figure 3(d). The instruction is as simple as the others: all the typing information is in the constant pool. It is the job of the *resolver* to verify that the type information in the constant pool is consistent with the real type of that class, and we do not consider resolution here as discussed above.

4.5 Branch Instructions

Conditional branch instructions have two constraints, of course.

$$\frac{}{\Xi \vdash \mathtt{pp}, \mathtt{i} : \mathtt{md} \rightarrow \mathtt{pp}'} \tag{6}$$

where

$$\begin{aligned} \mathtt{md}(\mathtt{pp}) &= \langle \mathtt{st}, \mathtt{lvt} \rangle \\ \mathtt{pp}' &= \mathtt{pp} + 3 \\ \mathtt{ft}' &= \left\langle \mathtt{st}', \mathtt{lvt} \right\rangle \\ \mathtt{md}(\mathtt{pp}') &\sqsubseteq \mathtt{ft}' \\ \mathtt{pp}'' &= \mathtt{pp} + \mathtt{n} \\ \mathtt{md}(\mathtt{pp}'') &\sqsubseteq \mathtt{ft}' \end{aligned}$$

with instruction-specific constraints in figure 3(e).

4.6 Return Instructions

Return instructions are peculiar in that no instruction of the method is executed afterwards, so the instruction introduces no constraints on the frame types.

$$\frac{}{\Xi \vdash \mathtt{pp}, \mathtt{i} : \mathtt{md} \rightarrow \mathtt{pp}'} \tag{7}$$

where

$$\begin{aligned} \mathtt{pp}' &= \mathtt{pp} + 1 \\ \mathtt{md}(\mathtt{pp}) &= \langle \mathtt{st}, \mathtt{lvt} \rangle \end{aligned}$$

with instruction constraints in figure 3(f); notice that an additional side condition is included in the table.

The full paper includes an analysis of how the inference rules can be read constructively, i.e., as a checker algorithm, and how they can be read as a constraint construction phase for a full verifier algorithm based on constraint solving.

5 Lightweight Verification

A naive "lightweight" bytecode verification would consist of defining the certificate for a method as simply the $\mathtt{md}$ value obtained by bytecode verification. However, this is too large for practical use. The challenge that concerns us here is to identify a *small* representation of the method descriptor such that we can realistically transmit it along with the method bytecode. (In practice such a certificate can be included as a *code attribute* in class files; a similar possibility should exist for the Java Card download format once it becomes available.)

5.1 Certificate

The central observation is the following: consider all the $\sqsubseteq$-constraints for any concrete verification, where $\mathtt{md}$ is fully known. Then extract all the individual (point-wise) constraints. They will be of the following kinds:

Next equal to constant or current. Requiring that a stack element or local variable type of $\mathtt{md}(\mathtt{pp}')$ is equal to a type in $\mathtt{ft}'$ or a constant type (implicit in the instruction or extracted from the constant pool).

Next super-type of constant or current. Requiring that a stack element or local variable type of $\mathtt{md}(\mathtt{pp}')$ is strictly less defined (a superclass or $\perp$) than a constant type or a type in $\mathtt{ft}'$.

Other. Requiring that a stack element or local variable type of $\mathtt{md}(\mathtt{pp}'')$ is equal to or less defined than a constant type or a type in $\mathtt{ft}'$.

The lightweight certificate is based on the idea that all the constraints of the first kind can be reconstructed by the checker without error by running sequentially through the code, and by keeping only one "current frame type", one "previous frame type", and in general as few frame types as possible, in the algorithm.

Then constraints of the first kind are not a problem: they are just a matter of comparing the contents of the current and previous frame type thus there is no need to keep them in the certificate.

Constraints of the second kind should result in a "delta" (difference) to the current frame type, since they require the following frame type to be less defined than what the immediately guessed, $\mathtt{ft}'$, would suggest. Again the required checks can be reproduced without error by the checker since the information is there.

The third kind requires a bit of care. If the other program point was *before* ours then we need to have saved the frame when we passed it such that the constraint can be checked when encountered. If the other program point is *after* the present then the checker must save the present program point *and* a demand that the constraint be checked once the other program point is reached, but it need not be registered in the certificate.

Formally, we express the certificate as

$$\begin{aligned}\mathtt{cert} \in \mathit{Certificate} &::= \langle \delta, \ell \rangle\\ \delta \in \Delta &= \mathit{PPoints} \xrightarrow{\text{fin}} \mathit{FrameType}\\ \ell \in \mathit{Labels} &= \mathbf{P}(\mathit{PPoints})\end{aligned}$$

Thus a certificate is a pair $\langle \delta, \ell \rangle$ with the first component, δ, storing differences in the frame type from what is derived (in this paper simplified to just store full frame types), and the second, ℓ, stores target labels.

5.2 Certification

The above suggests how the certificate should be produced as a side effect to bytecode checking, thus by extending (BV) to "lightweight bytecode certification"

$$\leq:, \mathsf{cid}, \mathsf{cp} \vdash \mathsf{msig}, \mathsf{c}, \mathsf{ms}, \mathsf{mlv} : \mathsf{md} \to \mathsf{cert} \tag{LBC}$$

The root rule (1) should then be augmented to

$$\frac{\Xi \vdash \emptyset, 0, \mathsf{c} : \mathsf{md} \to \mathsf{PP}, \mathsf{cert}}{\leq:, \mathsf{cid}, \mathsf{cp} \vdash \mathsf{msig}, \mathsf{c}, \mathsf{ms}, \mathsf{mlv} : \mathsf{md} \to \mathsf{cert}} \tag{8}$$

with the same side conditions.

For instruction sequences we need to make the *union* of two certificates (on the two components separately), thus:

$$\frac{\begin{array}{c}\Xi \vdash \mathsf{pp}, \mathsf{i} : \mathsf{md} \to \mathsf{pp}', \mathsf{cert} \\ \Xi \vdash \mathsf{PP}', \mathsf{pp}', \mathsf{c} : \mathsf{md} \to \mathsf{PP}'', \mathsf{cert}'\end{array}}{\Xi \vdash \mathsf{PP}, \mathsf{pp}, \mathsf{i} \cdot \mathsf{c} : \mathsf{md} \to \mathsf{PP}'', \mathsf{cert}''} \tag{9}$$

where $\mathsf{PP}' = \mathsf{PP} \cup \{\mathsf{pp}\}$ and $\mathsf{cert}'' = \mathsf{cert} \cup \mathsf{cert}'$, and the certificate starts out empty:

$$\frac{}{\Xi \vdash \mathsf{PP}, \mathsf{pp}, \epsilon : \mathsf{md} \to \mathsf{PP}, \emptyset} \tag{10}$$

For the single instruction rules all that needs to be done is replace pp' with $\mathsf{pp}', \mathsf{cert}$ in each conclusion and then add appropriate side conditions. To (4) and (5) add

$$\begin{aligned} \mathsf{cert} &= \langle \delta, \epsilon \rangle \\ \delta &= \begin{cases} \{\mathsf{pp}' \mapsto \mathsf{md}(\mathsf{pp}')\} & \text{if } \mathsf{md}(\mathsf{pp}') \sqsubset \mathsf{ft}' \\ \emptyset & \text{otherwise} \end{cases} \end{aligned}$$

to reflect that these just use the following instruction normally, thus making it possible to exploit the default frame type if possible.

The addition to (6) is similar:

$$\begin{aligned} \mathsf{cert} &= \langle \delta, \ell \rangle \\ \delta &= \begin{cases} \{\mathsf{pp}' \mapsto \mathsf{md}(\mathsf{pp}')\} & \text{if } \mathsf{md}(\mathsf{pp}') \sqsubset \mathsf{ft}' \\ \emptyset & \text{otherwise} \end{cases} \\ \ell &= \begin{cases} \{\mathsf{pp}''\} & \text{if } \mathsf{pp}'' < \mathsf{pp} \\ \emptyset & \text{otherwise} \end{cases} \end{aligned}$$

Finally we add the following to (7):

$$\mathsf{cert} = \langle \delta, \ell \rangle$$

$$\delta = \begin{cases} \{\mathrm{pp}' \mapsto \mathrm{md}(\mathrm{pp}')\} & \text{if } \mathrm{pp}' \in \mathrm{Dom}(\mathrm{md}) \\ \emptyset & \text{otherwise} \end{cases}$$

where we only create a delta if there is an instruction following the return-instruction.

Notice that one cannot "compress" or "decompress" between a full *MDescr*, md, and a *Certificate*, cert: in both cases the actual bytecode is needed since the redundant information that is removed depends on the actual bytecode instructions.

We conclude with an example of a certificate.

Example 1 (Certification). Consider the small Java class in the left part of figure 4. The full verification of the `rectoy` method is shown in the right part; the bytecode is as produced by Sun's javac compiler (except we are not using "fast" instructions), and we have inserted the frame type "before" the program point for each instruction. The constant pool is not shown: the only knowledge used is that

$$\mathrm{cp}(4) = \mathsf{methref}(\texttt{RecToy}, \mathsf{methsig}(\texttt{rectoy}, \texttt{int}), \texttt{int})$$

(for the invokevirtual instruction). Notice how it is clear that local variable 0 and 1 are initialized with `this` and the `n` parameter, and that local variable 2 and 3 are used for `m` and `r`, respectively.

The certificate only has to inform the checker that the local variable number 3 may be uninitialized in instruction 30 if it is reached via the conditional branch in instruction 5, *i.e.*, that the last entry in lvt should be $\perp$. If this is not the case then the frame type comparison for the second branch in the ifle instruction fails. The certificate thus just has a single element in the δ component:

$$\mathrm{cert} = \langle \{30 \mapsto \langle \epsilon, \texttt{RecToy} \cdot \texttt{int} \cdot \texttt{int} \cdot \perp \rangle\}, \epsilon \rangle$$

(In practice δ should be compacted into something like $\{30 \mapsto \{3 : \perp\}\}$, of course, reusing the unchanged types from the preceding instruction.) The ℓ component is empty because there were no backwards jumps in the code such as would arise, *e.g.*, from `while` statements.

The full paper also includes an example using `while` *to get a backwards jump.*

5.3 Lightweight Verification

Now the certificate is produced all we need to do is use it instead of the complete method descriptor to do bytecode checking. Operationally this means that the information in the certificate should be used to *emulate* the use of md in bytecode checker rules. Fortunately the certificate is constructed such that all information extracted from md can be available *when needed*. Thus we can achieve the effect by systematically modifying the bytecode checker to obtain a system for the "lightweight bytecode verifier" judgment

$$\leq:, \mathrm{cid}, \mathrm{cp} \vdash \mathrm{msig}, \mathrm{c}, \mathrm{ms}, \mathrm{mlv} : \mathrm{cert} \qquad \text{(LBV)}$$

```
class RecToy extends Object {

  int rectoy(int n) {
    int m;
    int r;

    m = 0;
    if (n > 0) {
      n = n - 1;
      r = this.rectoy(n);
      m = n + r;
    }

    return m;
  }
}
```

pp	st	lvt	i
0	ϵ	RecToy · int · ⊥ · ⊥	iconst_0
1	int	RecToy · int · int · ⊥	istore [2]
3	ϵ	RecToy · int · int · ⊥	iload [1]
5	int	RecToy · int · int · ⊥	ifle [25]
8	int	RecToy · int · int · ⊥	iload [1]
10	int · int	RecToy · int · int · ⊥	iconst_1
11	int	RecToy · int · int · ⊥	isub
12	ϵ	RecToy · int · int · ⊥	istore [1]
14	int	RecToy · int · int · ⊥	aload [0]
16	RecToy	RecToy · int · int · ⊥	iload [1]
18	RecToy · int	RecToy · int · int · ⊥	invokevirtual [4]
21	int	RecToy · int · int · ⊥	istore [3]
23	ϵ	RecToy · int · int · int	iload [1]
25	int	RecToy · int · int · int	iload [3]
27	int · int	RecToy · int · int · int	iadd
28	int	RecToy · int · int · int	istore [2]
30	ϵ	RecToy · int · int · ⊥	iload [2]
32	int	RecToy · int · int · ⊥	ireturn

Figure 4: `rectoy` method source and bytecode verification.

The bytecode checker judgments generally have the form $\cdots : \mathtt{md} \to \cdots$. This must be changed to a form where the certificate is used to maintain a data structure with the subset of $\mathtt{md}$ that will be needed. Formally this is achieved by replacing $\mathtt{md}$ in the judgments with the composite $\mathtt{cert}, \mathtt{ft}, \mathtt{S}, \mathtt{P}$ with new sorts

$$\begin{aligned} \mathsf{S} &\in \mathit{SavedFrameTypes} = \mathit{PPoints} \xrightarrow{\text{fin}} \mathit{FrameType} \\ \mathsf{P} &\in \mathit{PendingChecks} = \mathbf{P}(\mathit{PPoints} \times \mathit{FrameType}) \end{aligned}$$

with the following intentions

- $\mathtt{cert} = \langle \delta, \ell \rangle$, as before.
- $\mathtt{ft}$ contains the "current frame type", *i.e.*, is always equal to $\mathtt{md}(\mathtt{pp})$ in the current context.
- S is a map of the saved frame types that are needed later, corresponding to those program points that are in $\mathsf{PP} \cap \ell$ (targets of backwards jumps).
- P contains all the pending frame type comparisons created by forward jumps from instructions already passed.

We omit unneeded parts in each case, of course, as well as pass parts that are updated when appropriate.

We will go through the rules, changing them to accommodate the new structure (and thus demonstrating that the transformation is correct). As before we use Ξ without mentioning the components, and similarly we assume $\mathtt{cert} = \langle \delta, \ell \rangle$.

5.4 Method Check

Rule (1) becomes

$$\frac{\Xi \vdash \emptyset, 0, \mathtt{c} : \mathtt{cert}, \mathtt{ft}_0, \emptyset, \emptyset \to \mathtt{ft}, \mathsf{S}, \mathsf{P}, \mathsf{PP}}{\le:, \mathtt{cid}, \mathtt{cp} \vdash \mathtt{msig}, \mathtt{c}, \mathtt{ms}, \mathtt{mlv} : \mathtt{cert}} \tag{11}$$

with the side conditions

$$\begin{aligned} \mathtt{msig} &= \mathsf{methsig}(\mathtt{rmt}, \mathtt{mt}_1 \cdots \mathtt{mt}_n) \\ \mathtt{lvt} &= \{0 \mapsto \mathtt{cid}, 1 \mapsto \mathtt{mt}_1, \ldots, n \mapsto \mathtt{mt}_n, \\ &\qquad n{+}1 \mapsto \bot, \ldots, \mathtt{mlv}{-}1 \mapsto \bot\} \\ \mathtt{ft}_0 &= \langle \epsilon, \mathtt{lvt} \rangle \end{aligned}$$

which initializes the current frame type to something possibly more defined than $\mathtt{md}(0)$ (if so then $\delta(0)$ is defined).

5.5 Sequences

The sequence rules are essentially unchanged, except that they take care of applying a delta to the current frame type if this is appropriate, since this is uniform for all instructions. Thus (2) becomes

$$\frac{\begin{array}{c}\Xi \vdash \mathtt{pp}, \mathtt{i} : \mathtt{cert}, \mathtt{ft}_1, \mathtt{S}, \mathtt{P}_1 \to \mathtt{ft}', \mathtt{P}', \mathtt{pp}' \\ \Xi \vdash \mathtt{PP}', \mathtt{pp}', \mathtt{c} : \mathtt{cert}, \mathtt{ft}', \mathtt{S}', \mathtt{P}' \to \mathtt{ft}'', \mathtt{S}'', \mathtt{P}'', \mathtt{PP}''\end{array}}{\Xi \vdash \mathtt{PP}, \mathtt{pp}, \mathtt{i} \cdot \mathtt{c} : \mathtt{cert}, \mathtt{ft}, \mathtt{S}, \mathtt{P} \to \mathtt{ft}'', \mathtt{S}'', \mathtt{P}'', \mathtt{PP}''} \tag{12}$$

where

$$\begin{aligned}
\mathtt{PP}' &= \mathtt{PP} \cup \{\mathtt{pp}\} \\
\mathtt{cert} &= \langle \delta, \ell \rangle \\
\mathtt{ft}_1 &= \begin{cases} \delta(\mathtt{pp}) & \text{if } \mathtt{pp} \in \mathrm{Dom}(\delta) \wedge \delta(\mathtt{pp}) \sqsubseteq \mathtt{ft} \\ \mathtt{ft} & \text{otherwise} \end{cases} \\
\mathtt{S}' &= \begin{cases} \mathtt{S} \cup \{\mathtt{pp} \mapsto \mathtt{ft}_1\} & \text{if } \mathtt{pp} \in \ell \\ \mathtt{S} & \text{otherwise} \end{cases} \\
\forall \mathtt{ft}''', \langle \mathtt{pp}, \mathtt{ft}''' \rangle &\in \mathtt{P} : \mathtt{ft}''' \sqsubseteq \mathtt{ft}_1 \\
\mathtt{P}_1 &= \{ \langle \mathtt{pp}''', \mathtt{ft}''' \rangle \in \mathtt{P} \mid \mathtt{pp}''' \neq \mathtt{pp} \}
\end{aligned}$$

The third side condition applies the delta of $\mathtt{cert}$, if there is one, passing the current frame type (which is now known to be equal to $\mathtt{md}(\mathtt{pp})$) to the instruction handling judgment (it is intentional that $\mathtt{ft}_1$ is not defined if $\delta(\mathtt{pp}) \not\sqsubseteq \mathtt{ft}$ since then the checking should fail). The fourth saves the current frame type in the $\mathtt{S}$ used for subsequent instructions if the program point is in the ℓ component of the certificate. The fifth checks any pending constraints that have waited for $\mathtt{ft}_1$ to become available, and the sixth and last removes them from the $\mathtt{P}$ used for subsequent instructions.

The (3) rule becomes

$$\frac{}{\Xi \vdash \mathtt{PP}, \mathtt{pp}, \epsilon : \mathtt{cert}, \mathtt{ft}, \mathtt{S}, \emptyset \to \mathtt{ft}, \emptyset, \mathtt{PP}} \tag{13}$$

where the use of $\emptyset$ for $\mathtt{P}$ means that all pending constraints must have been checked for lightweight verification to succeed.

5.6 Single Instructions

The single instruction rules are changed by systematic (literal) textual replacement as follows:

- Replace "$\mathtt{md}(\mathtt{pp})$" with "$\mathtt{ft}$".
- Remove conditions "$\mathtt{md}(\mathtt{pp}') \sqsubseteq \mathtt{ft}'$". This is safe because all such constraints are checked by (12).

- Replace conditions "$\mathtt{md}(\mathtt{pp}'') \sqsubseteq \mathtt{ft}'$" with

$$\begin{aligned}&\big(\mathtt{pp}'' < \mathtt{pp}' \;\wedge\; \mathtt{S}(\mathtt{pp}'') \sqsubseteq \mathtt{ft}' \;\wedge\; \mathtt{P}' = \mathtt{P}\big)\\ &\qquad\qquad\qquad \vee\; \big(\mathtt{pp}'' \geq \mathtt{pp}' \;\wedge\; \mathtt{P}' = \mathtt{P} \cup \{\mathtt{pp}' \mapsto \mathtt{ft}'\}\big)\end{aligned} \tag{14}$$

The full paper contains all the rules.

5.7 Correctness and Safety

We can now prove that the technique is *tamper-proof* by showing it both *sound* and *complete* with respect to standard bytecode verification, as explained in the introduction. We express this using the three judgments for bytecode verification, lightweight bytecode certification, and lightweight bytecode verification (all defined above):

$$\leq:, \mathtt{cid}, \mathtt{cp} \vdash \mathtt{msig}, \mathtt{c}, \mathtt{ms}, \mathtt{mlv} : \mathtt{md} \tag{BV}$$

$$\leq:, \mathtt{cid}, \mathtt{cp} \vdash \mathtt{msig}, \mathtt{c}, \mathtt{ms}, \mathtt{mlv} : \mathtt{md} \rightarrow \mathtt{cert} \tag{LBC}$$

$$\leq:, \mathtt{cid}, \mathtt{cp} \vdash \mathtt{msig}, \mathtt{c}, \mathtt{ms}, \mathtt{mlv} : \mathtt{cert} \tag{LBV}$$

Our completeness lemma states that certification succeeds if an only if normal bytecode verification succeeds. This ensures that our technique has exactly the right expressiveness (and that it should be possible to integrate a certification algorithm into a normal bytecode verifier).

Lemma 2 (completeness). (BV) *iff* $\exists \mathtt{cert}$: (LBC).

Proof. $\Rightarrow$ follows from the fact that the construction of $\mathtt{cert}$ can never fail, thus a (BV) proof can be enriched with these to become an (LBC) proof. $\Leftarrow$ is trivial by observing that erasing all mention of $\mathtt{cert}$ in the (LBC) proof gives the (BV) proof. □

Our soundness lemma states that lightweight verification can only succeed if the certificate is valid. This ensures that the technique is *tamper-proof*, *i.e.*, that one cannot fabricate a certificate that will fool the (LBV) to accept non-verifiable bytecode.

Lemma 3 (soundness). (LBV) *iff* $\exists \mathtt{md}$: (LBC).

Proof. $\Rightarrow$ is proved by constructing an (LBC) proof from the (LBV) proof. The $\mathtt{md}$ that is needed is obtained by extracting the "$\Xi \vdash \mathtt{pp}, \mathtt{i} : \mathtt{cert}, \mathtt{ft}_1, \mathtt{S}, \mathtt{P}_1 \rightarrow \mathtt{ft}', \mathtt{P}', \mathtt{pp}'$" premise of all instances of (12) in the (LBV) proof: the instantiation of the current frame type, $\mathtt{ft}_1$, is the value needed for $\mathtt{md}(\mathtt{pp})$. It is easy to see that such a value must be present for all program points. We then only need to show that all the required constraints of the (LBC) proof are already present in the (LBV) proof. This requires a study of the values in $\mathtt{S}$, $\mathtt{P}$, and the ℓ component of $\mathtt{cert}$, treating each of the cases from Section 5.1. Notice that we do not reuse the checked certificate directly to construct the one built in the (LBC) proof: this is intentional as the actual certificate in the (LBV) proof may have additional (harmless and redundant) information.

Conversely, $\Leftarrow$ is proved by constructing an (LBV) proof from the (LBC) proof. This essentially amounts to inserting $\mathtt{md}(\mathtt{pp})$ into the current frame type, and to extract all side conditions of the form "$\mathtt{md}(\mathtt{pp}'') \sqsubseteq \mathtt{ft}'$" into either S or P depending on whether $\mathtt{pp}'' < \mathtt{pp}'$ or not.

The proof details can be found in the full paper. □

From this we can derive that lightweight bytecode verification is completely equivalent to normal bytecode verification.

Theorem 4. (BV) *iff* $\exists \mathtt{cert} :$ (LBC) $\wedge$ (LBV).

Proof. Immediate from soundness and completeness (Lemmas 3 and 2) □

One very important remark is that we can prove the equivalence of usual and lightweight verification without reference to the operational semantics of JVM execution: neither usual nor lightweight verification make sense without such an operational semantics, of course, but that is not the study of this paper, and the reported result does not depend on it.

6 Conclusions

We have demonstrated how a static semantics for bytecode checking of a Java (sublanguage) method can be split into a "lightweight" certificate generator and a "lightweight" bytecode verifier.

6.1 Assessment

The certificate generator has the complexity of the usual bytecode verifier, however, only the code producer needs to run it. The critical component is the checker for which we have shown

Completeness. A certificate always exists for bytecode that goes through the normal verifier.

Soundness. If the lightweight bytecode verifier accepts a certificate/method bytecode pair then the bytecode can be verified normally – the method is *tamperproof.*

Compact certificates. The lightweight certificate is much smaller than the full set of frame types assigned by the bytecode verification: it only has to contain

- the program point and difference to the derived frame type where there is a such, and

- the program point numbers of all *backward jump targets*;

we remarked that compactness can be improved by elimination goto and return instructions.

High speed. The lightweight bytecode verifier runs in time almost-linear in the length of the verified method plus the certificate: each rule has to check a constant number of entries in the frame type against the saved frame type and the certificate.

Low space. The lightweight bytecode verifier only needs the space taken up by the certificate, saved frame types, and the single current frame type.

In summary we have shown a way to realize safe downloading of applets to very tight environments such as Java Cards with the need for cryptographic methods.

6.2 Assurance Issues

In fact, the separation of the generation and check of the certificate means that only the checker is safety-critical: soundness assures that if a code producer cheats by trying to fake a certificate then the checker will reject it.

This should be contrasted with the current situation for Java, where even "small" Java environments, such as in web browsers, must contain a complete bytecode verifier to conform to the Java standard. Since the entire verifier is safety critical, this essentially makes it necessary for companies providing such environments to need to trust a large and complicated portion of the code, a portion for which they might not even have the source.

Our approach offers a backwards-compatible method: systems with a full bytecode verifier can ignore any lightweight certificate and do conventional verification, whereas systems with only a lightweight verifier can *either*

- reject the non-certified code as insecure, *or*
- submit the code to an external lightweight certifier, which does not have to be trusted, because the result is checked anyway.

In fact this means that it would make sense to make a standard out of lightweight certificates. Owners of source of full bytecode verifiers could then offer *lightweight certification* services, for example in the form of a web proxy that certifies all class files that passes by.

The full paper includes a list of future work, notably extending the language with exceptions and protection, and an account of the relation to work on formalizing the JVM (Bertelsen 1997, Pusch 1998) on Java type soundness (Drossopoulou, Eisenbach & Khurshid 1997, Drossopoulou & Eisenbach 1997, Oheimb & Nipkow 1998, Nipkow & von Oheimb 1998), class loading (Jensen et al. 1998), safety in connection with downloading (Leroy & Rouaix 1998), and surely other work reported in the FUJ '98 meeting.

Acknowledgements. The authors would like to thank Xavier Leroy, Pierre Lescanne, Christine Paulin-Mohring, and the anonymous FUJ '98 referees, for helpful comments and interactions.

References

Bertelsen, P. (1997), Semantics of java byte code, Student report, Technical University of Denmark.

Drossopoulou, S. & Eisenbach, S. (1997), Java is type safe – probably, *in* 'European Conference of Object Oriented Programming', LNCS, Springer-Verlag.

Drossopoulou, S., Eisenbach, S. & Khurshid, S. (1997), Is java type safe, Technical report, Imperial College.

Gosling, J., Joy, B. & Steele, G. (1996), *The Java Language Specification*, The Java Series, Addison-Wesley.

Jensen, T., Le Métayer, D. & Thorn, T. (1998), A formalisation of visibility and dynamic loading in java, *in* 'ICCL '98', IEEE. Also published as a IRISA Technical Report no 1137, October 1997.

Kahn, G. (1987), Natural semantics, Rapport de Recherche 601, INRIA, Sophia-Antipolis, France.

Leroy, X. & Rouaix, F. (1998), Security properties of typed applets, *in* 'POPL '98—25th Annual ACM Symposium on Principles of Programming Languages', SIGPLAN Notices, pp. 391–403.

Lindholm, T. & Yellin, F. (1996), *The Java Virtual Machine Specification*, The Java Series, Addison-Wesley.

Necula, G. C. (1997), Proof-carrying code, *in* 'POPL '97—24th Annual ACM Symposium on Principles of Programming Languages', SIGPLAN Notices.

Nipkow, T. & von Oheimb, D. (1998), Java$_{light}$ is type-safe – definitely, *in* 'POPL '98—25th Annual ACM Symposium on Principles of Programming Languages', SIGPLAN Notices, pp. 161–170.

Oheimb, D. v. & Nipkow, T. (1998), Machine-checking the Java specification: Proving type-safety, *in* J. Alves-Foss, ed., 'Formal Syntax and Semantics of Java', LNCS, Springer. To appear, available from http://www4.informatik.tu-muenchen.de/~isabelle/bali/doc/Springer98.html.

Pusch, C. (1998), Formalizing the java virtual machine in isabelle/hol, Technical Report TUM-I9816, Institut für Informatik, Technische Universität München.

Rose, E. (1998), Towards secure bytecode verification on a java card, Master's thesis, DIKU, University of Copenhagen.

Stata, R. & Abadi, M. (1998), A type system for java bytecode subroutines, *in* 'POPL '98—25th Annual ACM Symposium on Principles of Programming Languages', SIGPLAN Notices.

Sun (1997), *Java Card 2.0 Language Subset and Virtual Machine Specification*, revision 1.0 final edn. ftp://ftp.javasoft.com/docs/javacard/JC20-Language.ps.