Lightweight Bytecode Verification

Eva Rose (eva@rose.name)
UFR d'informatique, University of Paris 7; 2,place Jussieu, 75251 Paris, France. *

Abstract. In this paper, we provide a theoretical foundation and improvements to the existing bytecode verification technology, a critical component of the Java security model, for mobile code used with the Java "micro edition" (J2ME) which is intended for embedded computing devices.

In Java, remotely loaded "bytecode" class files are required to be *bytecode verified* before execution, *i.e.*, to undergo a static type analysis that protects the platform's Java run-time system from so-called "type confusion" attacks such as pointer manipulation. The data flow analysis which performs the verification, however, is beyond the capacity of most embedded devices because of the memory requirements which the typical algorithm will need.

We propose to take a Proof-Carrying Code (PCC) approach to data flow analysis in defining an alternative technique called "Lightweight Analysis" which uses the notion of a "certificate" to *reanalyze* a previously analyzed data flow problem, even on poorly resourced platforms. We formally prove that the technique provides the same guarantees as standard bytecode safety verification analysis, in particular that it is "tamper proof" in the sense that the guarantees provided by the analysis, cannot be broken by crafting a "false" certificate or by altering the analyzed code.

We show how the Java bytecode verifier fits into this framework for an important subset of the Java Virtual Machine; we also show how the resulting "lightweight bytecode verification" technique generalizes and simulates the J2ME verifier (to be expected as Sun's J2ME "K-Virtual machine" verifier was directly based on an early version of this work), as well as Leroy's "on-card bytecode verifier" which is specifically targeted for Java Cards.

1. Introduction

In this section we will present the context of lightweight bytecode verification. Specifically, we will advocate for our approach to bytecode type safety verification. Finally, we will give an overview of the paper.

1.1. Mobile code on small devices

Over the last years, there has been an increasing demand for generalizing the programming capability of small, independent, network-connected devices such as smart cards, point-of-sale terminals, personal digital assistants, set-top boxes and other types of pervasive devices which feature an on-device microprocessor.

* Funded by GIE Dyade, a joint INRIA/Bull alliance.

In this paper, we specifically consider how to ensure type safety for code which is downloaded over an untrusted network onto such a small device. The Java platform seems ideally suited for this task because Java, and its predecessor Oak, originally were developed with this type of deployment in mind (Sun, 1997; O'Connell, 1995). The original Oak prototype did, in fact, already include a *bytecode verifier* which locally ensured the well-typedness of transmitted code. For various reasons, the prototype failed in performance, but the idea survived in terms of the Java Virtual Machine (JVM). At the virtual machine level it is possible to address both safety and security issues independently of both the underlying execution platform and an untrusted code provider. Indeed the initial success of Java was associated with the ability to send mobile code as "applets" over an untrusted network to be executed in a web browser in a secure manner. The technique which allows this is called "sandboxing" (Sun, 2002). A cornerstone of the sandboxing security model is *type safety* which basically guarantees that all object references truly refer to objects of a compatible type. This is crucial because so-called "type confusion" attacks may seriously alter a system's security measures, *e.g.*, by updating an object at an address manufactured in an integer. (for a survey of Java security attacks see McGraw and Felten, 1997).

In Figure 1 we show how bytecode verification normally has been sceduled for a Java class file transfer over an untrusted network.

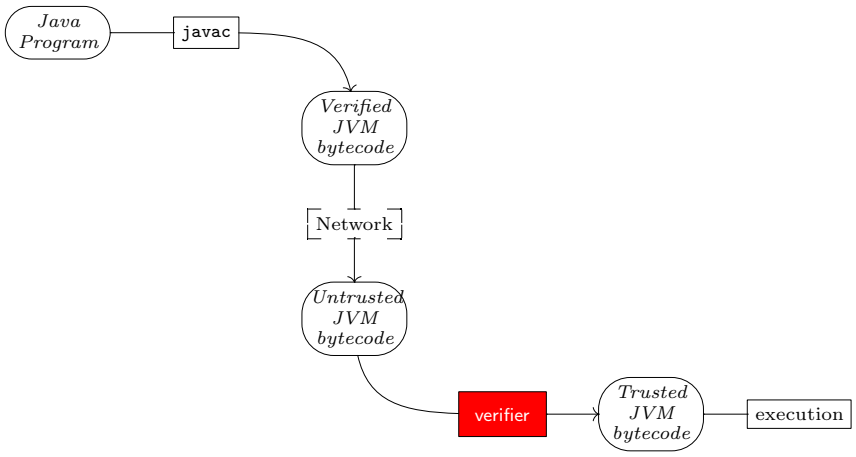

Figure 1. Standard bytecode verification.

Today, the Java 2 standard is organized into a variety of different platforms. One of these, the *Java 2 Micro Edition* (J2ME) is specifically targeted for small devices featuring an on-device microprocessor (Sun, 1999a). The smallest of such devices are described by the *Connected Limited Device Configuration* (CLDC) specification (Taivalsaari, 2000). To give an idea of the available memory on such a device, let us consider a *Java Card* (Sun, 1999b) platform that implements the CLDC (Chen, 2000):

Scratch memory: General name for the part of the RAM which has the highest flexibility and permits full speed read and write operations of individual bytes. The contents, however, is lost when the power is removed. Quite expensive. This kind of memory is typically available in the range of a few thousand bytes on a smart card.

Flash memory: Persistent memory which can be read bytewise, but only written to in blocks of continuous bytes (typically 64) by a process which is rather slow (typically ten thousand times slower than for scratch memory). Relatively inexpensive. This type of memory typically comes in the range of thirty to a hundred thousand bytes on a smart card.

ROM: The Read-Only Memory part becomes fixed when the circuit is manufactured. Earlier smart card editions typically contained sixteen thousand bytes of ROM for completely constant system programs and data. ROM is becoming increasingly obsolete because of decreasing costs on Flash memory (so even system data can now be updated).

When the present study began it was not obvious how to provide for bytecode type safety on such sparsely resourced execution platforms.

Sun's original bytecode verifier was specified as a data flow algorithm with a *space requirement proportional to the number of backward jumps in the bytecode.* (To be specific, the data flow algorithm could be implemented with a space requirement which is given by $O(B*(S+L))$, where B it the number of backward jump targets, S is the maximal stack size, and L is the number of local variables (Lindholm and Yellin, 1996, §4.9)). For many small devices, however, this is beyond their capacity. The traditional way to overcome this has been to "sign" the bytecode with a digital signature and then check the authenticity of this at the device (Sun, 2002).

When we began this study, yet a different approach to prevent safety violations for code transmitted over an untrusted network had just been

published: emphProof-Carrying Code (Necula, 1997; Necula and Lee, 1996). However, it was for a long time unclear how the approach specifically could be applied to Java, and why there should be a need, when bytecode could be secured from attacks by means of digital signatures.

1.2. Chryptographic signatures

Code signing can be done using symmetric, shared-key chryptography or by using public-key signatures. (Smart card standards such as "Global Platform" support both types of key signatures on Java Card applets.) All techniques which implements this concept, however, works by storing a "key" *from a trusted party* by which the authencity of the code signature eventually is checked at the execution platform.

We have listed some issues concerning cryptographic signatures.

- A well-estabiliched technology already exists for many smart card standards (Sun, 2002),
- guarantees that the code is *literally identical* to the code which was sent off by the trusted party,
- the code consumer, however, is obliged trust an *external party* (the key provider),
- the number of keys to store has a tendency to grow, which may be problematic on very sparsely resourced devices (Anderson, 1994),
- the trust-relation between code consumer and code provider creates a *single point of failure* system. Whenever a device receives code which is widely distributed, it is an unfortunate situation when the device producer and the code provider do not necessarily trust the same sources (or each other). In banking, *e.g.*, this is the frequent case when credit cards are issued by one bank, whereas code is produced and encrypted by a competing bank (Anderson, 1994).

For systems or devices which are storing high-sensitive data (credit cards, banking cards, personal identification devices, *etc*) the last three of the listed arguments are even more problematic to overcome than for other constrained systems because of the devastating perspective of a safety or security leak.

1.3. Proof-carrying code (PCC)

The concept was launched by Necula and Lee (1996). It allows untrusted code to be statically verified as safe to execute, in the case where safe code behavior can be logically specified as type properties and automatically verified.[1] PCC works by the definition of a "security policy" expressed as a logical system of decidable program safety properties. Thus, for a correct program, an additional proof with respect to the defined logical system can be constructed upon code transfer time, such that it can be mechanically decided whether the program adheres to the adopted policy or not at the code receiving platform. (We refer to section 1.4 for further discussion on the definition of PCC.)

We have listed some issues concerning PCC.

- The code consumer only has to trust it's *own, internal security policy management.*
- The checker component implements a decidable proof check at the code receiving platform.
- Application of PCC (security policy, proof generator, and mechanical proof checker) for different kinds of compiler schemas to real assembler code have already been successfully implemented (Colby et al., 2000b; Necula and Lee, 1996).
- The number of transfered bytes will increase with the size of the certificate.
- PCC methods can only ensure that the code doesn't *alter* the code consumer's security policy. It cannot ensure that the received code is the same as the emitted code.
- It is crucial that the proof producer and the code consumer have integrated the *same* safety policy in order for a correct program to be accepted. Thus, if the code emitter sends an accompanying proof based upon a different (non-compatible) safety policy, the code will not get accepted, even in those cases where the program is safe to execute.

For systems/devices which are storing high-sensitive data (credit cards, banking cards, personal identification devices, *etc*) in particular, the first of the listed arguments which credits PCC is highly attractive.

[1] By the term safe program behavior, we understand behavior which does not allow access to private files/data, to overwrite important files/data, to access unauthorized resources, *etc.*

The fact that a code receiver only has to trust it's *own, internal safety policy* makes the system very robust for attacks. The main disadvantage to overcome is how to deal with the increased size of transmitted bytes as well as the proof (type) checker's size as well as the memory consumption on the code receiving platform.

Finally it is important to consider if it is critical that the received code, even when proven safe to execute, may have been altered along the network in some (safe) manner.

1.4. Application of PCC for Java

Earlier work, dating to year 2000, have successfully addressed this issue for a compiler of Java source code to X86 (Intel) assembly code (Colby et al., 2000b; Colby et al., 2000a). The certifying Java compiler in this case produces the native target code, annotations, and a proof with respect to a set of axioms and rules which specifies the Java type safety requirements for the intel architecture.

In this study, however, we have been concerned with a description of a *platform independent* application of PCC to object oriented languages in general, and Java in particular, in order for our solution to scale well over a large network.

When this work began in 1997 and 1998 (Rose, 1997; Rose and Rose, 1998), the notion of PCC was defined in a more restricted manner than it is today (Necula, 1997). At the time, PCC had been specifically defined for proof systems with respect to a well-founded logic with a Curry-Howard isomorphism[2] to the transmitted language's formal type system.

The definition at the time made a direct application of PCC for Java impossible, because no well-founded formal logic has yet been specified for object systems with subtyping.

With the altered definition of PCC in 2000, however, our original approach to lightweight bytecode verification (Rose and Rose, 1998), where Java bytecode safety was decided by an axiomatic system for the Java virtual machine's static, operational semantics, now comes under the definition of PCC.

We notice, that the listing of issues for PCC in section 1.3 is valid for both the original and later specification of PCC.

[2] A "Curry-Howard" correspondence guarantees the existence of an isomorphism between a formal type system and a formal logic.

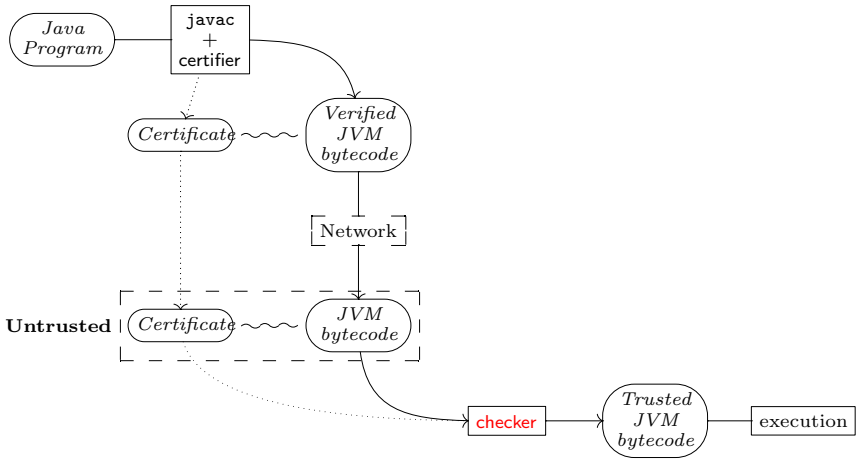

Figure 2. Lightweight bytecode verification.

1.5. Our approach

By taking a proof carrying code approach to bytecode verification over an untrusted network, we assume that the type safety process is performed in several steps: at the code transmitter platform, which we assume is large enough to host a certifying bytecode verifier, and at the code receiver platform, which we assume is very constrained in memory space. As a result of this approach, type safety verification becomes divided into several stages as illustrated in Figure 2.

We intend to formalize the lightweight bytecode verification in two steps: first we specify lightweight bytecode verification as a *general data flow problem*, then we specifically apply the problem to the Java virtual machine. In both cases, we shall formalize the technique as an inference system.

1.6. Overview

In Section 2 we formally specify lightweight bytecode verification in a data flow framework. Specifically, we adress when a (constraint based) data flow problem can be converted into a *lightweight* data flow problem, and we show that the technique is "tamper-proof". In Section 3 we apply lightweight verification to Java (bytecode) for an important Java virtual machine subset. In Section 4 we specifically relate lightweight

verification for Java to Sun's J2ME bytecode verifier, and Leroy's Java on-card verifier. Finally, in Section 5, we conclude over the work.

2. The Lightweight Data Flow Concept

In this section, we shall present the theory of "lightweight verification" as a way to solve a data flow problem. We will do so in accoedance with the way data flow analysis is presented by Nielson et al. (1999).

2.1. Data flow analysis

We shall take a constraint-based approach to data flow analysis and represent programs as directed graphs where nodes are given in terms of labelled "elementary blocks" connected by control flow edges. (Any traditional flow graph organized in *basic blocks* is an example of such a graph; the simplest way to obtain a labelling of basic blocks is to use the entry address of the blocks as the flow graph's label set.)

```
class B {int dummy() {return 0;}}
class A extends B {}

class C {
  void m() {
    B x = new A();
    while (x.dummy() != 0)
      x = new B();
  }
}
```

Figure 3. Sample Java source program.

Example 1 (Java source). Consider a Java source program for a method `m()` as shown in Figure 3. Flow analysis of `m()` will determine that `x` has either the type assignment $\texttt{x} : t = \texttt{A}$ or $\texttt{x} : t = \texttt{B}$. (This is consistent with the *declared* type `B` but the declared type of local variables is not maintained until runtime so this has no significance for verification.) Subsequently the method call `x.dummy()` is safe as it just requires $t \leq: \texttt{B}$, *i.e.*, that t is a subtype of (or the same as) `B`, which is true for both type assignments.

Definition 2 (Data flow problems and solutions). Let *Lab* be the *non-empty* set of node labels of a flow graph, let *State* be the associated domain of program analysis states, and let $\sqsubseteq$ be a partial order defined on *State*.

A *data flow problem*, represented by a flow graph *exits*, is a map from blocks to program state constraints in terms of edges between elementary blocks. A *data flow solution*, represented by *entry*, is a map from the node labels to the program analysis states, such that the program's constraint set is satisfied. Formally this correspond to the following specifications.

$$\begin{aligned} \mathit{exits} &: \ \mathit{Exits} = (\mathit{Lab} \to \mathit{Edge}^*) \\ \mathit{entry} &: \ \mathit{Entries} = (\mathit{Lab} \to \mathit{State}) \end{aligned}$$

where a flow edge is specified by

$$\langle \tau, \ell \rangle \in \mathit{Edge} = (\mathit{State} \to \mathit{State}) \times \mathit{Lab}$$

We say that *entry solves* (or is a *solution* to) the data flow problem given by *exits iff*

$$\forall \langle \tau', \ell' \rangle \in \mathit{exits}(\ell) : \ \tau'(\mathit{entry}(\ell)) \sqsupseteq \mathit{entry}(\ell') \tag{2a}$$

Similarly, we say that *exits* is *solvable* with respect to $\langle \mathit{State}, \sqsubseteq \rangle$ if there exists an *entry* which solves the data flow problem given by *exits*.

We have that both *entry* and *exits* are naturally defined as *total functions* on *Lab*. In particular we have that if elementary block ℓ has no exit edges then $\mathit{exits}(\ell) = \emptyset$. Notice, however, that the state transformer map τ do *not* have to be a total map. When it is not defined we have that (2a) cannot be solved, and the data flow problem has no solution (as expected).

The data flow constraints which are given for a particular elementary block is defined by those flow edges which are directed to the block. In the rest of this paper, we may therefore simply refer to *exits* as the program's or flow graph's *constraint set*.

Example 3 (Java bytecode). In Figure 4 we have listed the virtual machine bytecode as a sequence of basic blocks (separated by blanks in the figure) which are annotated with their relative bytecode address. Except for a few, clarifying simplifications, the code has been generated by a standard Java compiler when applied to the program in Example 1. The *Lab* set consists of the relative *entry addresses* for the three basic blocks, *i.e.*, $\{0, 11, 19\}$.

The purpose of Java bytecode verification is to analyse the type of data which a method can process. A "state" description becomes a (static) type description of a Java method frame, *i.e.*, a (static) type description of the operand stack and a local variable table for the method being analyzed.

```
 0: new A
 7: astore 1
 8: goto 19

11: new B
18: astore 1

19: aload 1
20: invokevirtual mref(B,msig(dummy,[]),I)
23: ifne 11
26: return
```

Figure 4. Sample Java bytecode.

Our *State* set for this method consequently consists of type descriptions of a stack with at most one element, and a local variable table with at most two elements (in the JVM the first element always contains the `this` reference for instance methods.) The state represented by type triples $\langle t_{\text{stack}}, t_{\texttt{this}}, t_{\texttt{x}} \rangle$. The type of the stack, t_{stack} is a stack of the types on the stack; we'll need ϵ for the empty stack and stacks with a single type in this example. For the local variables, $t_{\texttt{this}}$ and $t_{\texttt{x}}$, $\perp$ denotes that the value might not have been initialized.

The example has an initial parameter constraint and constraints from the three edges, one to 11 and two directed to the basic block at address 19, written

$$\begin{aligned} \mathit{entry}(0) &\sqsubseteq \langle \epsilon, \texttt{C}, \perp \rangle \\ \mathit{entry}(11) &\sqsubseteq \mathit{entry}(19) \\ \mathit{entry}(19) &\sqsubseteq (\mathit{entry}(0))[t_{\texttt{x}} \mapsto \texttt{A}] \\ \mathit{entry}(19) &\sqsubseteq (\mathit{entry}(11))[t_{\texttt{x}} \mapsto \texttt{B}] \end{aligned}$$

where the "$(\mathit{entry}(\ell))[t_{\texttt{x}} \mapsto t']$" denotes the state obtained from $\mathit{entry}(\ell)$ by overriding the type of the local variable for `x` with t'.

Once we have defined the $\sqsubseteq$ approximation relation on the frame type states we can solve these constraints. For Java it turns out (details in the following section) that $\sqsubseteq$ is derived from the Java *assignment compatibility* type rules by pointwise extension to the frame type constituents (stack elements and variable types). The constraint system can then be solved by setting

$$\begin{aligned} \mathit{entry}(0) &= \langle \epsilon, \texttt{C}, \perp \rangle \\ \mathit{entry}(11) &= \langle \epsilon, \texttt{C}, \texttt{B} \rangle \\ \mathit{entry}(19) &= \langle \epsilon, \texttt{C}, \texttt{B} \rangle \end{aligned}$$

In general we have that the determination of an *entry* solution requires a generic data flow analysis (Kildall, 1973). With *lightweight analysis* we suggest that a flow graph can be *re-analyzed in a single traversal* of its nodes in ascending order based on an appropriate label annotations, a "certificate", collected during a prior proper analysis. By assuming an order on the nodes, it is possible to separate those constraints which are forward directed, and therefore can be incorporated directly in the "reconstruction" of a solution, and the "backward" directed constraints, which cannot be reconstructed, but still need to be checked against the candidate. In order to formalize this idea, we will make the assumption that the graph nodes are totally ordered and that the states are comparable.

Definition 4 (Lightweight hypothesis). We assume the following properties for a program data flow analysis.

$$(Lab, \leq) \text{ is a finite, total order} \tag{4a}$$

$$(State, \sqsubseteq) \text{ is a complete lattice} \tag{4b}$$

As usual we denote the meet and join operators of $(State, \sqsubseteq)$ by $\sqcap$ and $\sqcup$, respectively, and note that for a complete lattice they are total, commutative, and associative.

Remark 5. As pointed out by Klein and Nipkow (2002), lightweight bytecode verification for Java can actually be proven with a weaker hypothesis than (4b), namely for any Java type domains that construct a *well-founded semilattice*. This is consistent with general data flow analysis (Kildall, 1973) so lightweight verification can be specified in the more general case of a well-founded semilattice. In order to simplify our formalization and proofs, however, we shall insist on the completeness of the lattice, as this is a sufficient condition for our application to the J2ME.

The idea of lightweight bytecode verification is to perform a single pass of the flow graph in ascending node order, and for each node check that the constraints imposed by flow edges in both directions are satisfied with respect to a candidate solution. With respect to the forward edges in the graph, we need to maintain a structure of "pending" constraint checks (to be checked when we reach the target of the edge) and, similarly for the backward edges, we need to maintain a structure of the solution candidate states, which are "saved" to be checked later.

There are two problems with such an approach:

– During the linear pass we cannot know whether a label is the target of a backwards jump from some block to be processed later. We suggest that these "backward labels" are provided externally, to notify the algorithm of when to save a program state.

– We have to determine a solution candidate value when a data flow analysis iteration, resulting in another pass of the flow graph, would otherwise be inevitable. We suggest that the correct *entry* program state information it provided externally for these "iteration nodes" (*i.e.*, program states which satisfy the program's constraint set).

The external provision is formalized in terms of a certificate:

Definition 6 (A lightweight certificate). Let $exits\colon Lab \to State$ define a data flow problem. A certificate is given by a pair of a label set and a state map.

$$ce \in Cert = \{\langle sc, Lc\rangle \mid sc\colon Lab \to State, Lc \subseteq Lab\}$$

In order to facilitate the lightweight formalization, we define sc as a *total map* with the convention that $sc(\ell) = \top$ when no backwards constraints to ℓ are imposed by the flow graph.

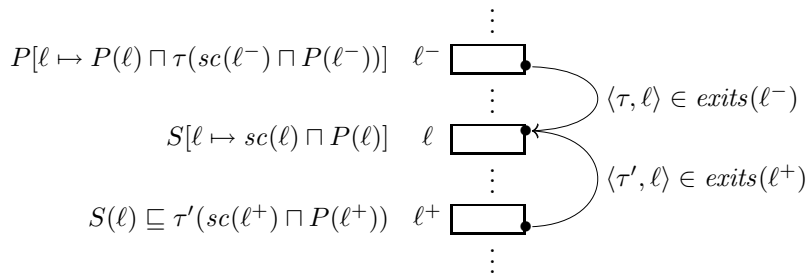

Figure 5. Lightweight analysis data structures.

In Figure 5, we have illustrated, in a semi-formal way, the changes to the "pending" and "saved" data structures for the three possible flow situations a node can be involved in. (We tacitly assume the mentioned convention that $P(\ell) = \top$ if there are no pending constraints registered for ℓ, and $S(\ell) = \bot$ if there are no saved constraints registered for ℓ.)

The flow edges, given as arrows, are annotated with their formal name. The nodes, labeled by $\ell^- \leq \ell \leq \ell^+$, are annotated with the kind

of action that lightweight analysis requires during the ascending node traversal. We write "$M[x \to v]$" to denote the map obtained from M by overriding the map of x to yield the value v.

We explain the three situations the lightweight algorithm must handle, in accordance with Figure 5.

- At ℓ^-: a forward directed edge to ℓ is discovered from the node labeled with ℓ^-. The contribution from this edge to the "pending" constraint P at ℓ is formally specified as $\tau^-(sc(\ell^-) \sqcap P(\ell^-))$. At this point, P becomes updated at ℓ with all the previously accumulated, pending constraints, *i.e.*, $P(\ell)$, and the specified contribution from the edge comming from the node at ℓ^-.

- At ℓ: at this point, *the current solution at* ℓ consists of all previously accumulated, pending constraints, specified by $P(\ell)$, as well as of the constraint imposed by the certificate, *i.e.*, $sc(\ell)$. We notice, that the current solution satisfies all forward directed flow constraints. The node at ℓ is also a target for a backward directed flow constraint. At this point, the current state is saved in S for a later constraint check.

- At ℓ^+: a backward directed edge to ℓ is discovered. At this point, the saved constraint $S(\ell)$ and constraint imposed from the node at ℓ^+, *i.e.*, $\tau^+(P(\ell^+) \sqcap sc(\ell^+))$, are checked for type safety.

Saved and pending constraints are formalized as simple maps.

Definition 7 (Saved and pending). The saved and pending total maps are defined by

$$\begin{aligned} S&: \ \mathit{Save} = (\mathit{Lab} \to \mathit{State}) \\ P&: \ \mathit{Pend} = (\mathit{Lab} \to \mathit{State}) \end{aligned}$$

Example 8 (Java lightweight bytecode verification). Reconsider the Java bytecode in Figure 4 from Example 3. The code is organized in three basic blocks with two forward jumps (from the block at address 0 to 19, and from the block at address 11 to 19) and one backward jump (from the block at address 19 to 11). We will illustrate how the lightweight verifier updates S and P according to the description in Figure 5 with a lightweight certificate given as follows:

$$ce = \langle 0 \mapsto \top, \{11 \mapsto \langle \epsilon, \mathtt{C}, \bot \rangle\}, \{11\}, 19 \mapsto \top \rangle$$

The `exit` maps are defined by the labeled frame type transformers τ^ℓ. These are defined over each basic block according to the Java instruction semantics (Lindholm and Yellin, 1999). The arguments are the

entry states for the blocks. To indicate that there are no requirements on a type we write "_".

$$\tau^0(\langle \_, \_, \_ \rangle) = \langle \_, \_, \mathtt{A} \rangle$$
$$\tau^{11}(\langle \_, \_, \_ \rangle) = \langle \_, \_, \mathtt{B} \rangle$$
$$\tau^{19}(\langle \_, \_, \_ \rangle) = \langle \_, \_, \_ \rangle$$

Below, we have listed the sequence of basic block labels in the order the blocks are being visited.

0: Initial constraint and forward jump.
We have $P(19) = (\top \sqcap \tau^0(\top \sqcap \langle \epsilon, \mathtt{C}, \bot \rangle\})) = \langle \epsilon, \mathtt{C}, \mathtt{A} \rangle$.

11: Backward jump target and forward jump.
We have $S(11) = \langle \epsilon, \mathtt{C}, \bot \rangle \sqcap \top = \langle \epsilon, \mathtt{C}, \bot \rangle$,
and $P(19) = (\langle \epsilon, \mathtt{C}, \mathtt{A} \rangle \sqcap \tau^{11}(\langle \epsilon, \mathtt{C}, \bot \rangle \sqcap \top) = \langle \epsilon, \mathtt{C}, \mathtt{A} \sqcap \mathtt{B} \rangle$.

19: Forward jump target.
Typecheck (successfully) enforced: $S(11) \sqsubseteq \tau^{19}(\top \sqcap \langle \epsilon, \mathtt{C}, \mathtt{A} \sqcap \mathtt{B} \rangle)$.

Where the initial constraint enforces $P(0)$ to be updated by $\langle \epsilon, \mathtt{C}, \bot \rangle$.

We proceed with a formalization of the general lightweight certification as an inference system, based on the notations introduced informally above. The initial analysis rule initializes the "current" label of the considered graph and the auxiliary "pending" and "saved" maps. We have tacitly assumed that *graphs have to contain at least one node* for lightweight analysis to make sense.

Definition 9 (lightweight bytecode verification). The lightweight bytecode verification judgment of a (non-empty) flow graph has the signature:

$$\vdash \mathit{Exits}, \mathit{Cert}$$

and is defined by the inference rule

$$\frac{\mathit{exits}, \mathit{ce} \vdash \ell_0, P_0, S_0}{\vdash \mathit{exits}, \mathit{ce}} \tag{9a}$$

$$\begin{aligned} \text{where} \quad & \ell_0 = \min\{\, \mathrm{Dom}(\mathit{exits}) \,\} \\ & P_0 = \{\ell \mapsto \top \mid \ell \in \mathrm{Dom}(\mathit{exits})\} \\ & S_0 = \{\ell \mapsto \bot \mid \ell \in \mathrm{Dom}(\mathit{exits})\} \end{aligned}$$

$\vdash \mathit{exits}, \mathit{ce}$ reads as follows: the lightweight certificate *ce lightweight verifies* the data flow problem *exits*. If $\exists \mathit{ce} \in \mathit{Cert}\colon \ \vdash \mathit{exits}, \mathit{ce}$ holds then we say that *exits* is *lightweight verifiable*.

The lightweight analysis rules for a subgraph is just a (tail) recursive rule set.

Definition 10. The lightweight analysis judgment for a subgraph has the signature:

$$\mathit{Exits}, \mathit{Cert} \vdash \mathit{Lab}, \mathit{Pend}, \mathit{Save}$$

where $\mathit{exits}, \mathit{ce} \vdash \ell, P, S$ reads as follows: the subgraph with node labels greater than or equal to ℓ lightweight analyzes in a flow graph context of constraints *exits*, and of the certificate *ce*, with the forward constraint set P and the backward constraint set S.

The judgement is specified by the inference rules below, one for the last node and one for the preceding nodes.

$$\frac{\mathit{exits}, \mathit{ce} \vdash \ell, P, S \to P', S'}{\mathit{exits}, \mathit{ce} \vdash \ell, P, S} \quad \text{where} \quad \ell = \max(\mathrm{Dom}(\mathit{exits})) \tag{10a}$$

$$\frac{\mathit{exits}, \mathit{ce} \vdash \ell, P, S \to P', S' \quad \mathit{exits}, \mathit{ce} \vdash \ell', P', S'}{\mathit{exits}, \mathit{ce} \vdash \ell, P, S} \quad \text{where} \quad \ell < \max(\mathrm{Dom}(\mathit{exits})) \wedge \ell' = \min\{\, \ell' \in \mathrm{Dom}(\mathit{exits} \mid \ell' > \ell \,\} \tag{10b}$$

using the node judgment of the following definition.

The lightweight analysis of a node reflects the scenarios for node ℓ in Figure 5, *i.e.*, whether a node is involved in forward or backward directed constraints.

Definition 11. The lightweight bytecode verification judgment for a node has the signature:

$$\mathit{Exits}, \mathit{Cert} \vdash \mathit{Lab}, \mathit{Pend}, \mathit{Save} \to \mathit{Pend}, \mathit{Save}$$

where $\mathit{exits}, \mathit{ce} \vdash \ell, P, S \to P', S'$ reads as follows: the node ℓ lightweight analyzes in a flow graph context of constraints *exits* and a certificate *ce*, modifying the accumulated forward constraints P to P' and backward constraints S to S'. It is defined by a single axiom.

$$\frac{}{\mathit{exits}, \langle \mathit{sc}, \mathit{Lc} \rangle \vdash \ell, P, S \to P', S'} \tag{11a}$$

where

$$
\begin{aligned}
s &= sc(\ell) \sqcap P(\ell) \\
\forall \langle \tau, \ell' \rangle \in exits(\ell), \ell' < \ell &: S(\ell) \sqsubseteq \tau(s) \\
\forall \langle \tau, \ell' \rangle \in exits(\ell), \ell' = \ell &: s \sqsubseteq \tau(s)
\end{aligned}
$$

$$
\forall \ell' \in Lab : S'(\ell') = \begin{cases} s & \text{if } \ell' = \ell \text{ and } \ell \in Lc \\ S(\ell') & \text{otherwise} \end{cases}
$$

$$
\forall \ell' \in Lab : P'(\ell') = \begin{cases} \top & \text{if } \ell' \leq \ell \\ P(\ell') \sqcap \left(\bigsqcap \{\, \tau(s) \mid \langle \tau, \ell' \rangle \in exits(\ell) \,\}\right) & \text{if } \ell' > \ell \end{cases}
$$

Notice how the two cases of the modified P' ensure that no old pending values are kept and that the pending entry for each subsequent label encodes *all* the constraint edges to that label.

In order to prove the equivalence between lightweight analysis and standard analysis, we shall begin by stating an important invariant property for lightweight analysis. The invariant ensures that for a given lightweight verified node *all constraints which have previously been imposed on that node have been met* (with respect to the given traversal order).

Lemma 12 (Lightweight invariant). Let *exits* denote a graph, and $Lc \subseteq \mathrm{Dom}(exits)$ a label subset, and $sc : \mathrm{Dom}(exits) \to State$ a map from labels to states. We have the equivalence:

$$\forall \ell \in \mathrm{Dom}(exits), L = \{\, \ell' \in \mathrm{Dom}(exits) \mid \ell' \leq \ell \,\} : (12a) \Leftrightarrow (12b)$$

where

$$\vdash exits|_L, \langle sc|_L, L \cap Lc \rangle \tag{12a}$$

$$\forall \langle \tau, \ell \rangle \in exits(\ell'), \ell' < \ell : (sc(\ell) \sqcap P(\ell)) \sqsubseteq \tau(sc(\ell') \sqcap P(\ell')) \tag{12b}$$

The state $sc(\ell) \sqcap P(\ell)$ is called the *current solution* in ℓ.

Proof. The invariant is proven by induction with respect to the formal lightweight specifications in definition 9 through definition 11.

Case $\Rightarrow$: Assume (12a). The proof is obtained by induction over the height of the proof tree.

Case $\Leftarrow$: Assume (12b). The proof is obtained by induction in the label ℓ.

□

Based on the invariant property we can formulate our main result as follows.

Theorem 13 (lightweight bytecode verification equivalence). *Given a data flow problem, exits*: $Lab \rightarrow Edge^*$*, the following statements are equivalent:*

1. *exits is solvable (cf. definition 2).*
2. *exits is lightweight verifiable (cf. definition 9).*

Proof. The theorem is proven by way of lemma 12.

Case $1 \Rightarrow 2$**:** is given as a proof scetch.

Assume the problem specified by *exits* is solvable. We show that there exists a certificate *ce* such that $\vdash$ *exits*, *ce* can be proven by lightweight verification inference system in definition 9 through definition 11.

The proof is conducted as an induction proof over the size of the graph (Lc).

We notice that the base case, give by definition 11, is instantates a certificate *ce* to the form $\langle Sc, Lc\rangle$. We also notice that the condition 2a ensures that the side conditions of definition 11 holds for the choice of Sc. By induction over Lc then follows that $\langle Sc, Lc\rangle$ is a valid certificate.

Case $2 \Rightarrow 1$**:** Assume a graph denoted by *exits* and a certificate $ce = \langle sc, Lc\rangle$ such that $\vdash$ *entry*, *ce* is provable. The lightweight invariance condition (12b) implies that for all $\ell \in \mathrm{Dom}(exits)$ we can associate a state $sc(\ell) \sqcap P(\ell)$ which satisfies the analysis condition (2a). In other words, a solution to the analysis is given by *entry*$'$ when defined by $entry'(\ell) = sc(\ell) \sqcap P(\ell)$.

□

Finally, we investigate the space complexity of the type check algorithm suggested by the proof (and explained in Figure 5), based on the current certification strategy.

2.3. Memory analysis

A program and its lightweight certificate can be placed in any sort of "slow write" memory on the target platform (flash, EPROM, or EEPROM), since they only need to be written once. From here on simply denoted "Flash".

The maximal *scratch space* we require to perform a lightweight bytecode verification of a (connected) graph in one traversal of its nodes, is decided by the maximal byte-size of a program state and the the

number of backward or forward edges in the graph. The number of backward labels indicates how many states will be stored in the "saved" contraint structure, whereas "pending" specifies the maximal number of simultaneously active (*i.e.* overlapping) forward edges in the flow graph. The reason why we specifically consider "overlapping" forward edges is that nodes for which there is a forward edge, clearly will not be scanned again if the node and its target has already been scanned. This means that the "pending" constraint structure well may be garbage collected during lightweight bytecode verification.

We summarize the discussion by the following equation.

$$\#\text{state descriptors} = (\max \mathit{state}) \times (\mathit{Lc} + \#P) \tag{14}$$

where $\#\mathit{state}$ is the (maksimal) size of a state, Lc is the number of backward labels, and $\#P$ is the greatest number of simultaneously (or overlapping) forward edges during the flow graph traversal.

In Table I, we have listed some interesting memory trade-offs for a flow graph with a certificate which is given by $\langle \mathit{sc}, \mathit{Lc} \rangle$.

Table I. Memory tradeoffs.

	Scratch	Flash	Comment
ce		✓	only defined when $\mathit{sc}(\ell) \sqsubset P(\ell)$)
S	$\mathit{Lc} \times (\#\mathit{state})$	×	can reuse *ce*
P	$\#P \times (\#\mathit{state})$	×	can reuse *ce*

We notice that all graphs are connected, $\#P \geq 1$. Depending on the number of jumps and the amount of certificate reuse, we have:

Corollary 15 (Scratch memory requirements). *The number of states needed to lightweight verify a program is* at least $O(1)$*, and* at most $O(\#P + \mathit{Lc})$*.*

2.4. The certificate and backward edge structures

In order to perform lightweight bytecode verification of a data flow graph by *one single* traversal of each node, we have to include the flow graph's backward directed label set Lc in the certificate. Furthermore, we cannot predict when a backward directed label will not be further targeted during the analysis; consequently we cannot "garbage collect" the constraint structure S until verification has ended. Two kinds of changes to the certificate would overcome this problem:

– the certificate could be constructed with a reference counter for each label,

– we could allow *one additional pass* of the flow graph before lightweigh verification, in order to record the label set Lc and the number of references to each of these.

The first suggestion would in most cases only increase the certificate insignificantly (depending on the number of backward edges to the same nodes.)

The second suggestion, however, forces the labels and references to be stored in flash ram. For the same reason as before, this would in most cases mean an insignificant increase in space.

With the introduction of reference counts, we could replace Lc in (14) with $\#S$.

2.5. Optimization of the certificate

In the previous paragraph, we discussed the effect of changing or adding other information to the certificate. The original lightweight certificate design, as it was originally presented for Java bytecode by the author (Rose, 1997), proposes a certificate which *by itself* embeds the program state component sc, which must contain a *solution*, *i.e.*, an *entry* function, to the program's data flow constraint set *exits*.

The certification approach totally eliminates the need for S and P to build up in the scratch memory, and only requires one single program state allocation. Even though a certificate can be placed in "slow" memory (such as flash or EEPROM), this design significantly increases the size of the certificate to proportional to the number of backward edges in the flow graph. In Section 4.1 we shall comment further on this in the case of Java bytecode.

3. Java Lightweight Bytecode Verification

The purpose of the section is to specify how the lightweight verification concept applies to Java bytecode for an important JVM subset.

3.1. The bytecode states

Java bytecode verification is concerned with type based data flow analysis. In the present research we support a type set which is closely related to the JavaCard set of Java value types (Sun, 1999b), *e.g.*,

Primitive types: `int`.

Reference types: class reference types `TypeName`, *or* one-dimensional arrays of class reference types `TypeName[]`, *or* `int[]`.

In Java, type safety (and thus type confusion) is closely related to the correspondance between the Java subtype concept and the class inheritance principle, sometimes referred to as the *inheritance-is-subtyping* property. The correspondance is formalized by the subtype relation $\leq:$ (Abadi and Cardelli, 1996) with the following meaning:

$$a \leq: b \quad \text{if and only if} \quad a = b \text{ or } a <: b \text{ , and}$$
$$a <: b \quad \text{if and only if} \quad a \text{ is a (direct or indirect) subclass of } b$$

The resulting *class hierarchy* is the outcome of a subtle interaction between class resolution, class loading, and bytecode verification (Lindholm and Yellin, 1999, §5.3–5.4). It is the *class resolver* which actually constructs the class hierarchy by allocating the necessary space on the object heap during method execution, whereas the *class loader* takes care of fetching the referenced class files. Bytecode verification performs in between, since the Java runtime system requires that a method is verified before it can be run. Class resolution and class loading happen in-between the method bytecode verification and are not encompassed by the verifier (Jensen et al., 1998). We can therefore assume that the subclass type context does not change while a method is being verified.

In order for the lightweight bytecode analysis to work, we have added four abstract type values: $\bot$ and $\top$, `Null`, and $\bot$`[]` to the set of Java value types.

Definition 16 (Abstract Java types). The abstract Java type sort *Type* is inductively specified.

$$\begin{aligned} t \in \mathit{Type} &::= t_{ob} \mid \texttt{int} \mid \bot \mid \top \\ t_{ob} \in \mathit{ObjectType} &::= t_{ob} \mid \texttt{int} \mid \bot \mid \top \\ t_{ob} \in \mathit{ObjectType} &::= \mathit{cid} \mid \texttt{Null} \mid \bot\texttt{[]} \mid \texttt{int[]} \mid \mathit{cid}\texttt{[]} \\ \mathit{cid} \in \mathit{ClassIdent} &\quad \textit{the set of defined Java classes} \end{aligned}$$

The order on the abstract type set is specified by the *assignment compatibility relation* between Java types (Lindholm and Yellin, 1999, §2.6.7). The relation describes when it is "safe" to assign a value of a given type to a variable of a given type. In other words, a (type) guarantee for when a method call will be prevented from failing at execution time. The assignment compatibility rules defines an ordering on the abstract type set.

Definition 17 (Assignment compatibility ordering). The *type compatibility ordering* $\sqsubseteq$ is defined over the type set *Type* by the following statements.

– For any type $t \in \mathit{Type}$: $t \sqsubseteq \top$ and $\bot \sqsubseteq t$.

– For any two (non-array) types $t_1, t_2 \in \mathit{Type}$: $t_1 \sqsubseteq t_2$ implies $t_1\texttt{[]} \sqsubseteq t_2\texttt{[]}$.

– For any two classes $\mathit{cid}_1, \mathit{cid}_2 \in \mathit{ClassIdent}$: $\mathit{cid}_1 \leq: \mathit{cid}_2$ implies $\mathit{cid}_2 \sqsubseteq \mathit{cid}_1$ (or $\sqsubseteq$ contains the reverse pointwise extension of $\leq:$).

– For any class $\mathit{cid} \in \mathit{ClassIdent}$: $\mathit{cid} \sqsubseteq \texttt{Null}$.

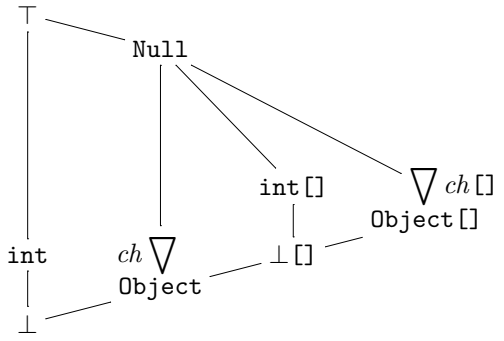

Figure 6. The Java type compatibility ordering.

In Figure 6, we have illustrated the assignment compatibility ordering for which "$\nabla\, ch$" represents a given "reversed class hierarchy". Specifically, we have defined the meaning of $\bot$ to signify "no value", and $\top$ to signify "all values". We notice, that this interpretation is somewhat opposite to mainstream conventions. The chosen meanings are based on the observation that subtypes correspond to subclasses, and that a subclass is *richer* in structure (more methods and fields may be defined) than its superclass (and vice versa).

We recall that the content and ordering of a class hierarchy is fixed during verification. Definition 16 is in reality specifying a *family* of type orderings $(\mathit{Type}, \sqsubseteq)_{ch}$ and indexed by (fixed) class hierarchies ch. To simplify the presentation, however, we will omit the index whenever it is clear from the context. For every given class hierarchy we obtain the following property for the corresponding type set Type.

Observation 18. The ordered set $\langle \mathit{Type}, \sqsubseteq \rangle$ is a finite lattice.

Before we proceed, we shall briefly comment on some of the omitted types.

Remark 19 (Omitted types).

– Other primitive types such as `float` types, `long float` types, or `long integer`s, can be added to the compatibility lattice in the same manner as `int`.

– Multi-dimensional arrays can be added to the compatibility lattice in the same manner as one-dimensional arrays without violating the finiteness of the lattice, because the array dimension in Java is limited at the JVM level (by two bytes).

To ensure that no method which may not be defined can be invoked at execution time, we statically have to verify the states of associated Java frames. Because only the operand stack and local variable table may contain Java program values, we shall omit a type description of other frame elements than the operand stack and the local variable table. The type description of this pair constitute our set of lightweight bytecode verification states, the *FrameType* set.

Definition 20 (Frame types). A *Frame type* is specified by a type description of an operand stack (of a given maximal lenght ms) and a local variable table (of a given maximal lenght ml).

$$
\begin{aligned}
ft \in \mathit{FrameType} &= (\mathit{StackType} \times \mathit{LocalType}) \cup \{\top\} \cup \{\bot\} \\
st \in \mathit{StackType} &= \bigcup_{n=0}^{ms} \mathit{Type}^n \\
lt \in \mathit{LocalType} &= \mathit{Type}^{ml} \\
ms \in \mathit{MaxStack} &= 0..65535 \\
ml \in \mathit{MaxLocals} &= 1..255
\end{aligned}
$$

We define $(\mathit{FrameType}, \sqsubseteq)$ as the pointwise extension of $(\mathit{Type}, \sqsubseteq)$ onto $\mathit{FrameType}$. We have added $\top$ with the meaning "all frame type values" and $\bot$ with the meaning "no frame type value".

We shall briefly comment on the formalization choises we have made for *FrameType*.

– Frame types are overly general in that they can describe states that are not possible, *e.g.*, uninitialized values on the stack. These abstract (frame) types have been included for the sake of completentess.

– Notice that we do not permit two frame types to be comparable if they have different stack lengths. The decision follows the official Java specification guidelines (Lindholm and Yellin, 1999, p.144). As argued in (Rose, 2002, p.61), the decision may at first glance seem unnecessary, but makes sense from the perspective that it might introduce a potential violation of ms, unless we keep track of the stack lengths for all compared frame types.

Three things are fixed during bytecode verification: the verified method, the class file where it is specified, the class (or subtype) context given by ch. We have that the (maximal) stack size ms and local variable table size ml for the method are fixed. As a consequence we have that a frame type ordering is a *family* of type descriptions $(\mathit{FrameType}, \sqsubseteq)_{ms,ml,ch}$ which are indexed the (maximal) frame dimensions and the (fixed) class hierarchy. To simplify the presentation, however, we will omit the index whenever it is clear from the context.

We observe that the frame type set *FrameType* is defined as a finite composition of (abstract) types in *Type*. Because *Type* is finite for any (fixed) class hierarchy, *FrameType* is a finite set.

These observations leads to the following statement.

Proposition 21. The ordered set $\langle \mathit{FrameType}, \sqsubseteq \rangle$ is a finite lattice.

3.2. The bytecode Label set and elementary blocks

In our formalization we shall let an elementary block be a virtual machine instruction (Lindholm and Yellin, 1999, §6). The label set *Lab*, thereby becomes represented by the bytecode program points.

Definition 22 (Bytecode label set and blocks).

$$\mathit{code} \in \mathit{Code} = \{\mathit{ProgramPoint} \to \mathit{Ins}\}$$
$$\mathit{pp} \in \mathit{ProgramPoint} = -1 \ldots 65535$$

Remark 23. The set of program points are naturally limited by the maximal number which can be stored in a two byte field in the class file. The number -1, however, is added to formalize the initial frame type constraint.

We consider an instruction set which is sufficiently detailed to facilitate the compilation of a non-trivial subset of Java. These non-trivial features include the ability to

- create and manipulate objects,
- perform object instance method calls,
- allow access to object instance fields,

Furthermore, we want to be able to program with loops, control flow branching and recursion, as well as exceptions. The subset is in fact close to the Java Card subset (Sun, 1999b), with one significant omission: jump subroutines.

Jump subroutines can be mechanically "unfolded" in a separate code transformation pass prior to code transmittance. Even though code unfolding may seem harder than to verify the subroutines directly on the original code, it should be compared with the strain of introducing polymorphic types into the formal frame work, as described in (Stata and Abadi, 1998). Specifically when we take into consideration that subroutines have been shown to bearly have any practical use in reality (Freund, 1998). (`jsr` is primarily used to compile `final` statements of exception handlers.) For these reasons, unfolding of subroutines is in fact already common, commercial practice for Sun's pre-verifier as described in Sun (2000).

The instruction set is specified as follows.

Definition 24 (Instructions).

$$\begin{aligned} \mathit{Ins} = {} & \mathsf{dup} \mid \mathsf{pop} \mid \mathsf{iconst}\ n \mid \mathsf{aconst\_null} \mid \mathsf{i}op \\ & \mid \mathsf{istore}\ n \mid \mathsf{astore}\ n \mid \mathsf{iload}\ n \mid \mathsf{aload}\ n \\ & \mid \mathsf{iastore}\ n \mid \mathsf{aastore}\ n \mid \mathsf{iaload}\ n \mid \mathsf{aaload}\ n \\ & \mid \mathsf{newarray\_int} \mid \mathsf{anewarray}\ cid \mid \mathsf{arraylength} \\ & \mid \mathsf{checkcast}\ t_{ob} \mid \mathsf{new}\ cid \\ & \mid \mathsf{putfield}(cid, id, t) \mid \mathsf{getfield}(cid, id, t) \\ & \mid \mathsf{invokevirtual}(cid, \langle id, t^* \rangle, rt) \\ & \mid \mathsf{goto}\ pp' \mid \mathsf{if}cmp\ pp' \mid \mathsf{ifnull}\ pp' \mid \mathsf{athrow}\ pp \\ & \mid \mathsf{return}\ pp' \mid \mathsf{ireturn}\ pp' \mid \mathsf{areturn}\ pp' \end{aligned}$$

with the auxiliary sorts

$$\begin{aligned} rt \in \mathit{ReturnType} &::= \mathtt{void} \mid t \\ op \in \mathit{Operation} &::= \mathtt{add} \mid \mathtt{sub} \\ cmp \in \mathit{Comparison} &::= \mathtt{eq} \mid \mathtt{ne} \mid \mathtt{lt} \mid \mathtt{le} \mid \mathtt{gt} \mid \mathtt{ge} \\ id \in \mathit{Identifier}&\ \textit{variable, field, and method names} \end{aligned}$$

3.3. The bytecode *exits* function

In Table II we have specified the *exit* function for each of the instructions in the instruction set. Each function is specified as a frame type transformer τ, which for each instruction is defined according to the Java instruction semantics (Lindholm and Yellin, 1999, §6). The columns are specified as follows (listed from left to right):

"*code*(*pp*)": specifies the instruction at program point *pp* in the method given by *code*,

Table II. Exit constraints.

$code(pp)$	ft	where	$\tau_{code(pp)}(ft)$	$exits_{code}(pp)$
dup	$\langle st \cdot t, lt \rangle$	$\|st\| + 2 \leq ms$	$\langle st \cdot t \cdot t, lt \rangle$	$\{\langle \tau_{code(pp)}, pp+1 \rangle\} \cup E(pp)$
pop	$\langle st \cdot t, lt \rangle$		$\langle st, lt \rangle$	$\{\langle \tau_{code(pp)}, pp+1 \rangle\} \cup E(pp)$
iop	$\langle st \cdot \texttt{int} \cdot \texttt{int}, lt \rangle$		$\langle st \cdot \texttt{int}, lt \rangle$	$\{\langle \tau_{code(pp)}, pp+1 \rangle\} \cup E(pp)$
iconst n	$\langle st, lt \rangle$	$\|st\| + 1 \leq ms$	$\langle st \cdot \texttt{int}, lt \rangle$	$\{\langle \tau_{code(pp)}, pp+1 \rangle\} \cup E(pp)$
aconst_null	$\langle st, lt \rangle$	$\|st\| + 1 \leq ms$	$\langle st \cdot \texttt{Null}, lt \rangle$	$\{\langle \tau_{code(pp)}, pp+1 \rangle\} \cup E(pp)$
istore n	$\langle st \cdot \texttt{int}, lt \rangle$		$\langle st, lt[n \mapsto \texttt{int}] \rangle$	$\{\langle \tau_{code(pp)}, pp+2 \rangle\} \cup E(pp)$
astore n	$\langle st \cdot t_{ob}, lt \rangle$		$\langle st, lt[n \mapsto t_{ob}] \rangle$	$\{\langle \tau_{code(pp)}, pp+2 \rangle\} \cup E(pp)$
iload n	$\langle st, lt \rangle$	$\|st\| + 1 \leq ms \wedge lt(n) = \texttt{int}$	$\langle st \cdot \texttt{int}, lt \rangle$	$\{\langle \tau_{code(pp)}, pp+2 \rangle\} \cup E(pp)$
aload n	$\langle st, lt \rangle$	$\|st\| + \leq ms \wedge lt(n) = t_{ob}$	$\langle st \cdot t_{ob}, lt \rangle$	$\{\langle \tau_{code(pp)}, pp+2 \rangle\} \cup E(pp)$
iastore n	$\langle st \cdot \texttt{int}[] \cdot \texttt{int} \cdot \texttt{int}, lt \rangle$		$\langle st, lt \rangle$	$\{\langle \tau_{code(pp)}, pp+1 \rangle\} \cup E(pp)$
aastore n	$\langle st \cdot cid[] \cdot \texttt{int} \cdot cid', lt \rangle$		$\langle st, lt \rangle$	$\{\langle \tau_{code(pp)}, pp+1 \rangle\} \cup E(pp)$
iaload n	$\langle st \cdot \texttt{int}[] \cdot \texttt{int}, lt \rangle$		$\langle st \cdot \texttt{int}, lt \rangle$	$\{\langle \tau_{code(pp)}, pp+1 \rangle\} \cup E(pp)$
aaload n	$\langle st \cdot cid[] \cdot \texttt{int}, lt \rangle$		$\langle st \cdot \texttt{int}, lt \rangle$	$\{\langle \tau_{code(pp)}, pp+1 \rangle\} \cup E(pp)$
newarray_int	$\langle st \cdot \texttt{int}, lt \rangle$		$\langle st \cdot \texttt{int}[], lt \rangle$	$\{\langle \tau_{code(pp)}, pp+2 \rangle\} \cup E(pp)$
anewarray cid	$\langle st \cdot \texttt{int}, lt \rangle$		$\langle st \cdot cid[], lt \rangle$	$\{\langle \tau_{code(pp)}, pp+3 \rangle\} \cup E(pp)$
arraylength	$\langle st \cdot t[], lt \rangle$		$\langle st \cdot \texttt{int}, lt \rangle$	$\{\langle \tau_{code(pp)}, pp+1 \rangle\} \cup E(pp)$
checkcast t_{ob}	$\langle st \cdot t_{ob}, lt \rangle$		$\langle st \cdot t, lt \rangle$	$\{\langle \tau_{code(pp)}, pp+3 \rangle\} \cup E(pp)$
new cid	$\langle st, lt \rangle$	$\|st\| + 1 \leq ms$	$\langle st \cdot cid, lt \rangle$	$\{\langle \tau_{code(pp)}, pp+3 \rangle\} \cup E(pp)$
putfield(cid, id, t)	$\langle st \cdot cid' \cdot t', lt \rangle$	$t \leq: t' \wedge cid' \leq: cid$	$\langle st, lt \rangle$	$\{\langle \tau_{code(pp)}, pp+3 \rangle\} \cup E(pp)$
getfield(cid, id, t)	$\langle st \cdot cid', lt \rangle$	$cid' \leq: cid$	$\langle st \cdot t, lt \rangle$	$\{\langle \tau_{code(pp)}, pp+3 \rangle\} \cup E(pp)$
invokevirtual$(cid, \langle id, t_1 \ldots t_n \rangle, \texttt{void})$	$\langle st \cdot cid' \cdot t'_1 \ldots t'_n, lt \rangle$	$\bigwedge_{i=1}^{n}(vt_i \leq: t_i') \wedge cid' \leq: cid$	$\langle st, lt \rangle$	$\{\langle \tau_{code(pp)}, pp+3 \rangle\} \cup E(pp)$
invokevirtual$(cid, \langle id, t_1 \ldots t_n \rangle, t)$	$\langle st \cdot cid' \cdot t'_1 \ldots t'_n, lt \rangle$	$\bigwedge_{i=1}^{n}(vt_i \leq: t_i') \wedge cid' \leq: cid$	$\langle st \cdot t, lt \rangle$	$\{\langle \tau_{code(pp)}, pp+3 \rangle\} \cup E(pp)$
goto pp'	ft		ft	$\{\langle \tau_{code(pp)}, pp' \rangle\} \cup E(pp)$
ifcmp pp'	$\langle st \cdot \texttt{int}, lt \rangle$		$\langle st, lt \rangle$	$\{\langle \tau_{code(pp)}, pp+3 \rangle, \langle \tau_{code(pp)}, pp' \rangle\} \cup E(pp)$
if`null` pp'	$\langle st \cdot t, lt \rangle$		$\langle st, lt \rangle$	$\{\langle \tau_{code(pp)}, pp+3 \rangle, \langle \tau_{code(pp)}, pp' \rangle\} \cup E(pp)$
athrow pp	$\langle st \cdot cid, lt \rangle$	$\texttt{Throwable} \sqsubseteq cid$	$\langle cid, lt \rangle$	$E(pp)$
return pp'	$\langle st, lt \rangle$	$rt = \texttt{void}$	$\top$	$E(pp)$
ireturn pp'	$\langle st \cdot \texttt{int}, lt \rangle$	$rt = \texttt{int}$	$\top$	$E(pp)$
areturn pp'	$\langle st \cdot t_{ob}, lt \rangle$	$rt \sqsubseteq t_{ob}$	$\top$	$E(pp)$

"*ft*": specifies the frame type as expected by the instruction semantics. In other words, an input frame type must match *ft* by equality up to instantiation of the free variables of *ft*.

"where": lists additional verification requirements (if any),

$\tau_{\mathit{code}(\mathit{pp})}(\mathit{ft})$**:** specifies the resulting frame type after application of $\tau_{code(pp)}$ on *ft* ($\tau_{code(pp)}(ft)$ is only defined for *ft* of the expexted form),

"*exits*(*pp*)": specifies the set of data flow edges from the instruction at *pp*, where $E(pp)$ specifies the flow edges created by the exception set which can be thrown at *pp*.

We specify the initial frame type constraint on program point 0 as a virtual edge, given by $exits(-1)$. Recall that the initial constraint is specified by an empty stack, the self-reference type at the local variable location 0, and a list of the actual method parameter types at local variable location 1 and on.

Definition 25 (Initial flow constraint). Let *cid* be the type of the method's self reference, and *ml* the constraint on the local variable table size. Finally, let $t_1 \ldots t_k$ denote the static types of the method parameters.

$$\begin{aligned} &exits(-1) = \{\langle \tau_{-1}, 0\rangle\}, \text{ where} \\ &\quad \tau_{-1}(pp) = \langle \epsilon, (cid, t_1, \ldots, t_k, \bot_{k+1}, \ldots, \bot_{ml-1})\rangle \, , 0 \leq k < ml \end{aligned}$$

We have that τ_{-1} is a constant function over *ProgramPoint*.

Notice that we could have formalized the last instruction state as a virtual edge, *e.g.*, for the `return` instruction (which has no semantically well-defined resulting state). Instead, we have chosen to restrict the definition of the state transformer τ for `return` and similar (non-fall through) instructions.

3.4. Exception verification

We have formalized exception verification in the following manner: if an exception can be thrown from a program point *pp* which is within an exception handler's try-range, then a flow edge is created to that handler.

Definition 26 (Exception handling).

$$\begin{aligned} E(pp) = \{\, \langle \tau_E, pp' \rangle \mid &pp_1 \leq pp < pp_2, \\ &\langle pp_1, pp_2, pp' \rangle \in ExcHandler \,\} \\ ExcHandler = ProgramPoint \times &ProgramPoint \times ProgramPoint \end{aligned}$$

The formalized approach to exception verification is rather conservative, because it also includes exceptions which, by a more detailed analysis, could be excluded from being processed by the associated handler. In Rose (2002) we have formalized a more precise analysis of how type safety can be verified for exceptions.

Finally, we can specify the constraint set function *exits* for any method given by *code*.

Definition 27 (*exits* function).

$$exits(pp) = \begin{cases} \{\langle \tau_{-1}, 0 \rangle\} & \text{if } pp = -1 \\ exits_{code}(pp) & \text{if } pp \in \mathrm{Dom}(code) \end{cases}$$

where τ_{-1} is specified in definition 25 and *exits* in Table II.

Our main result states that standard bytecode verification and lightweight bytecode verification are equivalent in that they provide the *same type safety guarantees*.

Theorem 28 (Lightweight bytecode verification is equivalent to standard bytecode verification). *For a Java method specified by code within the static type context FrameType and flow constraints specified by exits, the following statements are equivalent:*

1. *There exists a solution fta* : $\mathrm{Dom}(code) \cup \{-1\} \rightarrow$ *FrameType (a frame type approximation) which solves the flow constraints exits.*
2. *There exists a certificate ce so that FrameType* $\vdash$ *exits, ce is provable.*

Proof. From Proposition 21 and definition 22 we have that the *lightweight hypothesis* in definition 4 is satisfied for any method *code* with a label set $\mathrm{Dom}(code)$ and a type context *FrameType*.

The proof now follows from Theorem 13 with the following changes:

- *entry* is replaced by *fta*,
- $\langle sc, Lc \rangle$ is replaced by $\langle ftc, ls \rangle$, and
- ℓ is replaced by *pp*.

□

We notice that if a "false" certificate is transmitted (*e.g.*, altered during an untrusted network transfer) or if the code is has been changed so that it doesn't match the the certificate any longer, then the code will be rejected by the lightweigh bytecode verifier, even if the code is type safe.

Corollary 29 (Tamper-proof). *The lightweight bytecode verification technique is "tamper-proof".*

Finally, we shall comment on the way the memory is used by the algorithm. In Section 2.3 we described the memory requirements in the general lightweight bytecode verification case. In the specific case of Java bytecode, all of the previous considerations hold for the following changes:

- the "Lc" becomes ls
- the number of "state descriptors" becomes the number of Java "type descriptors",
- the "max state" dimension is given by ms and ml.

The number of stored Java types is given as follows.

$$\#\text{type descriptors} = (ms + ml) \times (ls + \#P) \tag{30}$$

where ls is the number of backward labels, and $\#P$ is the maximal number of simultaneously active forward edges along the code traversal. (For a more detailed explanation we refer to Section 2.3.)

4. Related Work

In this section we report on work which is directly related to lightweigth verification. In Section 4.1 we explain how Sun's lightweight bytecode verification variant, as implemented in the KVM, can be simulated by the present framework. In Section 4.2 we show how lightweight bytecode verification is a generalization of Leroy's proposed "On-card" bytecode verifier for Java Cards. Finally, in Section 4.3, we relate to the automated series of proofs of the lightweight bytecode verification technique which has been implemented by Klein and Nipkow.

In general there is a rich literature on general formalizations of the JVM and the bytecode verifier (Barthe et al., 2000; Drossopoulou et al., 1999; Freund and Mitchell, 1999; Hartel and Moreau, 2001; Stärk et al., 2001).

4.1. The K-virtual machine

Sun's principal implementation of the J2ME virtual machine is the "K-virtual machine", or KVM (Sun, 2000). The KVM verifier implements our original lightweight approach to type safety (Liang, 1999; Rose and

Rose, 1998) as initially proposed by the author (Rose, 1997). It uses what is called the "StackMap attribute" to implement a "naive" certificate which contains a frame type component for *every* jump target except the trivial "following instruction" ones. This violates the restriction that the certificate is only specified when $ftc(pp) \sqsubset (S(pp) \sqcap P(pp))$ which, in turn,

- makes S (and thus ls) obsolete because every "saved" type descriptor is already available in the certificate, and
- reduces P to contain only edges from an instruction to the immediately following instruction of which at most one is active at any time.

So the KVM is optimized to use only a single frame type descriptor variable at run-time at the cost of the certificate containing the full frame type descriptor for every jump target. KVM's certificates are consequently rather large: Leroy (2001) reports that the certificate of a method is up to half the size of the method code it certifies.

4.2. The "on-card" bytecode verifier

Leroy's "on-card" bytecode verifier (Leroy, 2002), which is targeted for Java Cards, works by imposing a series of pre-conditions on the code, obtained by *code transformation*, to introduce special *type invariants* prior to the type verification analysis:

- all local variables are initialized (by the modified virtual machine) at method entry,
- the operand stack is empty at all jump targets, and
- the type of each local variable is constant.

Consequently, all comparisons of frame types become equality tests with the constant "jump target frame type" $\langle \epsilon, t_0 \cdots t_n \rangle$:

- S becomes constant where defined so all that we need to test is that *if* $pp \in ls$ *then* the frame type at pp must be equal to the jump target frame type.
- P becomes constant so all we have to check is that *if* $P(pp) \neq \top$ then the frame type at pp is the jump target frame type.

For code without subroutines (`jsr`) we can thus reproduce Leroy's result in the following way:

1. Do Leroy's code transformation.
2. Add code at the start setting all non-parameter local variables to `null` (unless, as in Leroy's case, the JVM is known to do this).
3. Use the certificate $\langle \epsilon, T \rangle$.

The presence of the certificate avoids Leroy's fixed point iteration to compute the types of the local variables, however, executing the lightweight algorithm on the above uses more space that Leroy's because the pending and saved structures are still maintained. However, with the following modifications this can be avoided, resulting in equivalent space use of the two algorithms:

1. When the algorithm would extend P to $P[pp \mapsto ft]$ this is equivalent to testing that $pp \in T$ and $ft = \langle \epsilon, t_0 \cdots t_n \rangle$.
2. For $pp \in T$ the algorithm would
   - extend S to $S[pp \mapsto ft]$ if pp is a backwards jump, and
   - test $P(pp) \sqsubseteq ft$ pending if this was a forward jump,

   which in both cases is equivalent to testing that $ft = \langle \epsilon, t_0 \cdots t_n \rangle$

The lack of a certificate in the on-card verifier approach imposes the running time penalty of the fixed point iteration to obtain the local variable types, however, the on-card verifier could safely use a certificate like ours to avoid that.

An interesting question is how the code and frame size cost of the on-card algorithm of not allowing reuse of variable locations, *e.g.*, for nested local variables, compares to the code size cost incurred by lightweight bytecode verification's code size growth due to the unfolding of jsr instructions as done by Sun's preverifier.

4.3. Klein and Nipkow's mechanically verified proofs

Klein and Nipkow have formalized lightweight bytecode verification in the Isabelle/HOL theorem prover (Klein and Nipkow, 2001). However, the certificate they operate on is also based on a complete *entry* solution with respect to basic blocks, in accordance with the initial lightweight bytecode verification proposal (Rose, 1997) as also implemented in the KVM, *cf.*, Section 4.1. Also, they do not address exceptions.

When the certificate specifies frame types of all jump targets then the space bound collapses to a constant, as for the KVM, which is indeed what the authors find from analyzing the formlization as an algorithm (in a high-level functional language).

However, the main goal of Klein and Nipkow is to mechanically prove type safety and their result, verified with Isabelle/HOL, is similar to Theorem 28. Furthermore, since the Isabelle/HOL formalization is executable, they have achieved an executable lightweight bytecode verifier.

In (Klein, 2003) it is documented how both exceptions and object initialization can be mechanically proven type safe.

5. Conclusions

In this paper, we have formally defined lightweight bytecode verification by taking a PCC approach to constraint based data flow analysis. An approach which we have exploited to stage Java bytecode verification to perform safely over an untrusted network, on a small platform, in only one, straight code pass, for an important Java subset.

lightweight bytecode verification has resulted in important industrial applications: the **off-device pre-verifier**, the **in-device verifier**, and the certificate **StackMap** in the CLDC specification, implemented by the Sun (KVM) and by the company GEMPLUS (for smart cards).

The technique has here been formally proven to be "tamper-proof" in the sense that no clever conversion of a certificate can make a program's unsolvable constraint set suddently appear as solvable. For Java bytecode, the notion of tamper-proof is translated into a formal proof which states that conversion of a lightweight bytecode certificate cannot result in type unsafe bytecode to pass the lightweight verifier.

Finally, we have obtained a unified description of KVM, Leroy's "Java on-card verifier", and general lightweight bytecode verification as described in this paper. In particular we have found that with a naive certificate (frame types for every jump target), the type safety verification in all three cases requires exactly one frame type of scratch RAM.

Future Directions As "run-once" mobile code becomes a more and more elaborate on all the Java platforms the cost of bytecode verification is becoming noticeable even on standard systems, so Bracha (2000) proposed to develop the KVM verifier for the standard Java platform. We plan to work with the Java community to make this happen.

One of the most interesting future projects in relation to bytecode safety is to investigate how to replace existing runtime checks by static checks through an enrichment of the Java type system. One first attempt has been an investigation of how to represent field protection

information by the type system (Rose and Rose, 2001), however, more options should be investigated.

Acknowledgement First of all I would like to thank my former Ph.D. thesis supervisor Jean Goubault-Larrecq, as well as Xavier Leroy, for their interest and comments on earlier stages of Java lightweight bytecode verification, which was also my thesis subject. Then I would like to thank Kristoffer Rose for an earlier collaboration as accounted for in Rose and Rose (1998), but in particular his valuable comments on the final draft of this paper. I also appreciate the valuable input on the content and style of the paper from the anonymous referees. Finally, I thank Gie Dyade and INRIA-Rocquencourt, France, for having initiated and hosted the project.

References

Abadi, M. and L. Cardelli: 1996, *A Theory of Objects*, Monographs in Computer Science. Springer-Verlag.

Anderson, R.: 1994, 'Why Cryptosystems Fail'. *Comm. of the ACM* **37**(11), 32–40.

Barthe, G., G. Dufay, L. Jakubiec, B. Serpette, S. Sousa, and S. Yu: 2000, 'Formalization in Coq of the Java Card Virtual Machine'. In: S. Drossopoulou, S. Eisenbach, B. Jacobs, G. T. Leavens, P. Müller, and A. Poetzsch-Heffter (eds.): *Formal Techniques for Java Programs (ECOOP 2000 workshop)*. Sophia-Antipolis, France.

Bracha, G.: 2000, 'Java Class File Specification Update'. `http://jcp.org/en/jsr/detail?id=202`.

Chen, Z.: 2000, *Java Card Technology for Smart Cards*, The Java Series. Addison Wesley.

Colby, C., P. Lee, G. C. Necula, F. Blau, M. Plesko, and K. Cline: 2000a, 'A certifying compiler for Java'. *ACM SIGPLAN Notices* **35**(5), 95–107. Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '00).

Colby, C., G. C. Necula, and P. Lee: 2000b, 'A Proof-Carrying Code Architecture for Java'. In: *In Proceedings of the 12th International Conference on Computer Aided Verification (CAV00)*. Chicago, USA.

Drossopoulou, S., S. Eisenbach, and S. Khurshid: 1999, 'Is the Java Type System Sound?'. *Theory and Practice of Object Systems* **5**(1), 3–24.

Freund, S.: 1998, 'The Costs and Benefits of Java Bytecode Subroutines'. In: S. Eisenbach (ed.): *Formal Underpinnings of Java (an OOPSLA workshop)*. Vancouver, BC, Canada.

Freund, S. and J. Mitchell: 1999, 'A Formal Framework for the Java Bytecode Language and Verifier'. In: *ACM Conference on Object-Oriented Programming: Systems, Languages and Applications*.

Hartel, P. and L. Moreau: 2001, 'Formalizing the Safety of Java, the Java Virtual Machine and Java Card'. *ACM Computing Surveys* **33**(4), 517–558.

Jensen, T., D. Le Métayer, and T. Thorn: 1998, 'A Formalisation of Visibility and Dynamic Loading in Java'. In: *ICCL '98*. Also published as a IRISA Technical Report no 1137, October 1997.

Kildall, G. A.: 1973, 'A unified approach to global program optimization'. In: *Conference Record of the ACM Symposium on Principles of Programming Languages*. Boston, Massachusetts, pp. 194–206.

Klein, G.: 2003, 'Verified Java Bytecode Verification'. Ph.D. thesis, Institut für Informatik, Technische Universität München.

Klein, G. and T. Nipkow: 2001, 'Verified Lightweight Bytecode Verification'. *Concurrency and Computation: Practice and Experience* **13**(13), 1133–1151. Invited contribution to special issue of papers from Formal Techniques for Java Programs (ECOOP 2000 workshop).

Klein, G. and T. Nipkow: 2002, 'Verified Bytecode Verifiers'. *Theoretical Computer Science*. To appear.

Leroy, X.: 2001, 'Java bytecode verification: an overview'. In: *Computer Aided Verification, CAV 2001*, Vol. 2102 of *Lecture Notes in Computer Science*. pp. 265–285, Springer-Verlag.

Leroy, X.: 2002, 'Bytecode Verification for Java Smart Card'. *Software Practice & Experience* **32**, 319–340.

Liang, S.: 1999, 'Sun's New Verifier'. Personal Communication (e-mail). Explains how the KVM's verifier implements lightweigt verification.

Lindholm, T. and F. Yellin: 1996, *The Java Virtual Machine Specification*, The Java Series. Addison-Wesley.

Lindholm, T. and F. Yellin: 1999, *The Java Virtual Machine Specification*, The Java Series. Addison-Wesley, second edition.

McGraw, G. and E. W. Felten: 1997, *Java Security: Hostile Applets, Holes, and Antidotes*. John Wiley and Sons.

Necula, G. C.: 1997, 'Proof-Carrying Code'. In: *POPL '97—24th Annual ACM Symposium on Principles of Programming Languages*. SIGPLAN Notices.

Necula, G. C. and P. Lee: 1996, 'Safe Kernel Extensions Without Run-Time Checking'. In: *OSDI '96–Second Symposium on Operating Systems Design and Implementation*. Seattle, Washington.

Nielson, F., H. R. Nielson, and C. Hankin: 1999, *Principles of Program Analysis*. Springer-Verlag.

O'Connell, M.: 1995, 'Java: The inside story'. *SunWorld*. http://sunsite.uakom.sk/sunworldonline/swol-07-1995/swol-07-java.html.

Rose, E.: 1997, 'Towards Bytecode Verification on a Java Card'. In: M. Abadi (ed.): *Workshop on security and languages*. Palo Alto, California.

Rose, E.: 2002, 'Vérification de Code d'Octet de la Machine Virtuelle Java. Formalisation et Implantation'. Ph.D. thesis, SE. RoseUniversité Paris VII, 2, Place de Jussieu, 75251 Paris Cedex 05, France. Available from `http://www.evarose.net/thesis-submitted.pdf`.

Rose, E. and K. H. Rose: 1998, 'Lightweight Bytecode Verification'. In: S. Eisenbach (ed.): *Formal Underpinnings of Java (an OOPSLA workshop)*. Vancouver, BC, Canada.

Rose, E. and K. H. Rose: 2001, 'Java Access Protection through Typing'. *Concurrency and Computation: Practice and Experience* **13**(13), 1125–1132. First presented at the ECOOP 2000 workshop on Formal Techniques for Java Programs.

Stärk, R., J. Schmid, and E. Börger: 2001, *Java and The Java Virtual Machine – Definition, Verification, Validation*. Springer-Verlag.

Stata, R. and M. Abadi: 1998, 'A Type System for Java Bytecode Subroutines'. In: L. Cardelli (ed.): *Proceedings of the Twenty-Fifth Annual ACM Symposium on Principles of Programming Languages*. San Diego, California, ACM.

Sun: 1997, 'Java Frequently Asked Question 1.1: Where did Java come from?'. http://www.ibiblio.org/javafaq/javafaq.html.

Sun: 1999a, 'Java 2 Platform, Micro Edition'. `http://java.sun.com/j2me`.

Sun: 1999b, 'Java Card 2.1 Platform'. `http://java.sun.com/products/javacard/javacard21.html`.

Sun: 2000, 'Java 2 Platform Micro Edition (J2ME) Technology for Creating Mobile Devices'. `http://java.sun.com/products/cldc/wp/KVMwp.pdf`.

Sun: 2002, 'Secure Computing with Java: Now and the Future'. http://java.sun.com/marketing/collateral/security.html. White paper.

Taivalsaari, A.: 2000, 'J2ME Connected, Limited Device Configuration'. `http://jcp.org/en/jsr/detail?id=30`.