Java bytecode verification: an overview

Xavier Leroy

INRIA Rocquencourt and Trusted Logic S.A.
Domaine de Voluceau, B.P. 105, 78153 Le Chesnay, France
Xavier.Leroy@inria.fr

Abstract. Bytecode verification is a crucial security component for Java applets, on the Web and on embedded devices such as smart cards. This paper describes the main bytecode verification algorithms and surveys the variety of formal methods that have been applied to bytecode verification in order to establish its correctness.

1 Introduction

Web applets have popularized the idea of downloading and executing untrusted compiled code on the personal computer running the Web browser, without user's approval or intervention. Obviously, this raises major security issues: without appropriate security measures, a malicious applet could mount a variety of attacks against the local computer, such as destroying data (e.g. reformatting the disk), modifying sensitive data (e.g. registering a bank transfer via the Quicken home-banking software [4]), divulging personal information over the network, or modifying other programs (Trojan attacks).

To make things worse, the applet model is now being transferred to high-security embedded devices such as smart cards: the Java Card architecture [5] allows for post-issuance downloading of applets on smart cards in sensitive application areas such as payment and mobile telephony. This raises the stake enormously: a security hole that allows a malicious applet to crash Windows is perhaps tolerable, but is certainly not acceptable if it allows the applet to perform non-authorized credit card transactions.

The solution put forward by the Java programming environment is to execute the applets in a so-called "sandbox", which is an insulation layer preventing direct access to the hardware resources and implementing a suitable access control policy [8, 32, 16]. The security of the sandbox model relies on the following three components:

1. Applets are not compiled down to machine executable code, but rather to bytecode for a virtual machine. The virtual machine manipulates higher-level, more secure abstractions of data than the hardware processor, such as object references instead of memory addresses.
2. Applets are not given direct access to hardware resources such as the serial port, but only to a carefully designed set of API classes and methods that perform suitable access control before performing interactions with the outside world on behalf of the applet.

3. Upon downloading, the bytecode of the applet is subject to a static analysis called bytecode verification, whose purpose is to make sure that the code of the applet is well typed and does not attempt to bypass protections 1 and 2 above by performing ill-typed operations at run-time, such as forging object references from integers, illegal casting of an object reference from one class to another, calling directly private methods of the API, jumping in the middle of an API method, or jumping to data as if it were code [9, 36, 15].

Thus, bytecode verification is a crucial security component in the Java "sandbox" model: any bug in the verifier causing an ill-typed applet to be accepted can potentially enable a security attack. At the same time, bytecode verification is a complex process involving elaborate program analyses. Consequently, considerable research efforts have been expended to specify the goals of bytecode verification, formalize bytecode verification algorithms, and prove their correctness.

The purpose of the present paper is to survey briefly this formal work on bytecode verification. We explain what bytecode verification is, survey the various algorithms that have been proposed, outline the main problems they are faced with, and give references to formal proofs of correctness. The thesis of this paper is that bytecode verification can be (and has been) attacked from many different angles, including dataflow analyses, abstract interpretation, type systems, model checking, and machine-checked proofs; thus, bytecode verification provides an interesting playground for applying and relating various techniques in computed-aided verification and formal methods in computing.

The remainder of this paper is organized as follows. Section 2 gives a quick overview of the Java virtual machine and of bytecode verification. Section 3 presents the basic bytecode verification algorithm based on dataflow analysis. Sections 4 and 5 concentrate on two delicate verification issues: checking object initialization and dealing with JVM subroutines. Section 6 presents a more abstract view of bytecode verification as model checking of an abstract interpretation. Some issues specific to low-resources embedded systems are discussed in section 7, followed by conclusions and perspectives in section 8.

2 Overview of the JVM and of bytecode verification

The Java Virtual Machine (JVM) [15] is a conventional stack-based abstract machine. Most instructions pop their arguments off the stack, and push back their results on the stack. In addition, a set of registers (also called local variables) is provided; they can be accessed via "load" and "store" instructions that push the value of a given register on the stack or store the top of the stack in the given register, respectively. While the architecture does not mandate it, most Java compilers use registers to store the values of source-level local variables and method parameters, and the stack to hold temporary results during evaluation of expressions. Both the stack and the registers are part of the activation record for a method. Thus, they are preserved across method calls. The entry point for a method specifies the number of registers and stack slots used by the method, thus allowing an activation record of the right size to be allocated on method entry.

Control is handled by a variety of intra-method branch instructions: unconditional branch ("goto"), conditional branches ("branch if top of stack is 0"), multi-way branches (corresponding to the `switch` Java construct). Exception handlers can be specified as a table of (pc_1, pc_2, C, h) quadruples, meaning that if an exception of class C or a subclass of C is raised by any instruction between locations pc_1 and pc_2, control is transferred to the instruction at h (the exception handler).

About 200 instructions are supported, including arithmetic operations, comparisons, object creation, field accesses and method invocations. The example in Fig. 1 should give the general flavor of JVM bytecode.

Source Java code:

```
static int factorial(int n)
{
  int res;
  for (res = 1; n > 0; n--) res = res * n;
  return res;
}
```

Corresponding JVM bytecode:

```
method static int factorial(int), 2 registers, 2 stack slots
    0:  iconst_1        // push the integer constant 1
    1:  istore_1        // store it in register 1 (the res variable)
    2:  iload_0         // push register 0 (the n parameter)
    3:  ifle 14         // if negative or null, go to PC 14
    6:  iload_1         // push register 1 (res)
    7:  iload_0         // push register 0 (n)
    8:  imul            // multiply the two integers at top of stack
    9:  istore_1        // pop result and store it in register 1
   10:  iinc 0, -1      // decrement register 0 (n) by 1
   11:  goto 2          // go to PC 2
   14:  iload_1         // load register 1 (res)
   15:  ireturn         // return its value to caller
```

Fig. 1. An example of JVM bytecode

An important feature of the JVM is that most instructions are typed. For instance, the `iadd` instruction (integer addition) requires that the stack initially contains at least two elements, and that these two elements are of type `int`; it then pushes back a result of type `int`. Similarly, a `getfield` $C.f.\tau$ instruction (access the instance field f of type τ declared in class C) requires that the top of the stack contains a reference to an instance of class C or one of its sub-classes (and not, for instance, an integer – this would correspond to an attempt to forge an object reference by an unsafe cast); it then pops it and pushes back a value of type τ (the value of the field f). More generally, proper operation of the JVM is not guaranteed unless the code meets the following conditions:

- Type correctness: the arguments of an instruction are always of the types expected by the instruction.
- No stack overflow or underflow: an instruction never pops an argument off an empty stack, nor pushes a result on a full stack (whose size is equal to the maximal stack size declared for the method).
- Code containment: the program counter must always point within the code for the method, to the beginning of a valid instruction encoding (no falling off the end of the method code; no branches into the middle of an instruction encoding).
- Register initialization: a load from a register must always follow at least one store in this register; in other terms, registers that do not correspond to method parameters are not initialized on method entrance, and it is an error to load from an uninitialized register.
- Object initialization: when an instance of a class C is created, one of the initialization methods for class C (corresponding to the constructors for this class) must be invoked before the class instance can be used.
- Access control: method invocations, field accesses and class references must respect the visibility modifiers (`private`, `protected`, `public`, etc) of the method, field or class.

One way to guarantee these conditions is to check them dynamically, while executing the bytecode. This is called the "defensive JVM approach" in the literature [6]. However, checking these conditions at run-time is expensive and slows down execution significantly. The purpose of bytecode verification is to check these conditions once and for all, by static analysis of the bytecode at loading-time. Bytecode that passes verification can then be executed at full speed, without extra dynamic checks.

3 Basic verification by dataflow analysis

The first JVM bytecode verification algorithm is due to Gosling and Yellin at Sun [9, 36, 15]. Almost all existing bytecode verifiers implement this algorithm. It can be summarized as a dataflow analysis applied to a type-level abstract interpretation of the virtual machine. Some advanced aspects of the algorithm that go beyond standard dataflow analysis are described in sections 4 and 5. In this section, we describe the basic ingredients of this algorithm: the type-level abstract interpreter and the dataflow framework.

3.1 The type-level abstract interpreter

At the heart of all bytecode verification algorithms described in this paper is an abstract interpreter for the JVM instruction set that executes JVM instructions like a defensive JVM (including type tests, stack underflow and overflow tests, etc), but operates over types instead of values. That is, the abstract interpreter manipulates a stack of types and a register type (an array associating types to register numbers). It simulates the execution of instructions at the level of types. For instance, for the `iadd` instruction (integer addition), it checks that the stack of types contains at least two elements, and that the top two elements are the type `int`. It then pops the top two elements and pushes back the type `int` corresponding to the result of the addition.

$\texttt{iconst}\ n\ :\ (S,\ R) \rightarrow (\texttt{int}.S,\ R)$ if $|S| < M_{stack}$
$\texttt{iadd}\ :\ (\texttt{int}.\texttt{int}.S,\ R) \rightarrow (\texttt{int}.S,\ R)$
$\texttt{iload}\ n\ :\ (S,\ R) \rightarrow (\texttt{int}.S,\ R)$
if $0 \leq n < M_{reg}$ and $R(n) = \texttt{int}$ and $|S| < M_{stack}$
$\texttt{istore}\ n\ :\ (\texttt{int}.S,\ R) \rightarrow (S,\ R\{n \leftarrow \texttt{int}\})$ if $0 \leq n < M_{reg}$
$\texttt{aconst\_null}\ :\ (S,\ R) \rightarrow (\texttt{null}.S,\ R)$ if $|S| < M_{stack}$
$\texttt{aload}\ n\ :\ (S,\ R) \rightarrow (R(n).S,\ R)$
if $0 \leq n < M_{reg}$ and $R(n) <: \texttt{Object}$ and $|S| < M_{stack}$
$\texttt{astore}\ n\ :\ (\tau.S,\ R) \rightarrow (S,\ R\{n \leftarrow \tau\})$ if $0 \leq n < M_{reg}$ and $\tau <: \texttt{Object}$
$\texttt{getfield}\ C.f.\tau\ :\ (\texttt{ref}(D).S,\ R) \rightarrow (\tau.S,\ R)$ if $D <: C$
$\texttt{invokestatic}\ C.m.\sigma\ :\ (\tau'_n \ldots \tau'_1.S,\ R) \rightarrow (\tau.S,\ R)$
if $\sigma = \tau(\tau_1, \ldots, \tau_n)$ and $\tau'_i <: \tau_i$ for $i = 1 \ldots n$

Fig. 2. Selected rules for the type-level abstract interpreter. M_{stack} is the maximal stack size and M_{reg} the maximal number of registers.

Figure 2 defines more formally the abstract interpreter on a number of representative JVM instructions. The abstract interpreter is presented as a transition relation $i : (S, R) \rightarrow (S', R')$, where i is the instruction, S and R the stack type and register type before executing the instruction, and S' and R' the stack type and register type after executing the instruction. Errors such as type mismatches on the arguments, stack underflow, or stack overflow, are denoted by the absence of a transition. For instance, there is no transition on `iadd` from an empty stack.

Notice that method invocations (such as the `invokestatic` instruction in Fig. 2) are not treated by branching to the code of the invoked method, like the concrete JVM does, but simply assume that the effect of the method invocation on the stack is as described by the method signature given in the "invoke" instruction. All bytecode verification algorithms described in this paper proceed method per method, assuming that all other methods are well-typed when verifying the code of a method. A simple coinductive argument shows that if this is the case, the program as a whole (the collection of all methods) is well typed.

The types manipulated by the abstract interpreter are similar to the source-level types of the Java language. They include primitive types (`int`, `long`, `float`, `double`), object reference types represented by the fully qualified names of the corresponding classes, and array types. The `boolean`, `byte`, `short` and `char` types of Java are identified with `int`. Two extra types are introduced: `null` to represent the type of the null reference, and $\top$ to represent the contents of uninitialized registers, that is, any value. ("Load" instructions explicitly check that the accessed register does not have type $\top$, thus detecting accesses to uninitialized registers.) A subtyping relation between these types, similar to that of the Java language (the "assignment compatibility" relation), is defined as shown in Fig. 3.

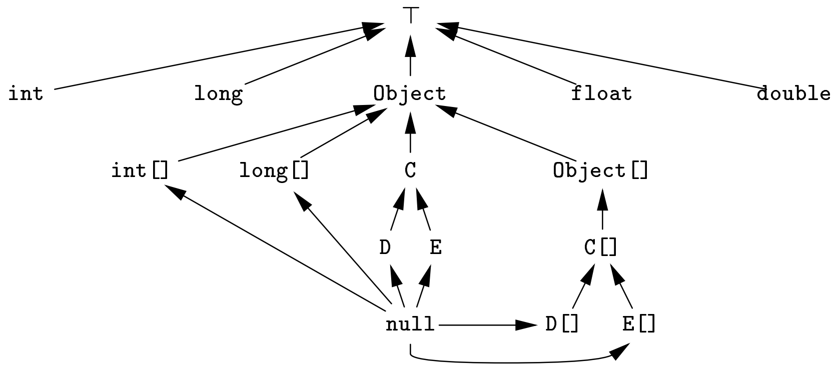


Fig. 3. Type expressions used by the verifier, with their subtyping relation. `C`, `D`, `E` are user-defined classes, with `D` and `E` extending `C`. Not all types are shown.

3.2 The dataflow analysis

Verifying a method whose body is a straight-line piece of code (no branches) is easy: we simply iterate the transition function of the abstract interpreter over the instructions, taking the stack type and register type "after" the preceding instruction as the stack type and register type "before" the next instruction. The initial stack and register types reflect the state of the JVM on method entrance: the stack type is empty; the types of the registers $0 \ldots n-1$ corresponding to the n method parameters are set to the types of the corresponding parameters in the method signature; the other registers $n \ldots M_{reg}-1$ corresponding to uninitialized local variables are given the type $\top$.

If the abstract interpreter gets "stuck", i.e. cannot make a transition from one of the intermediate states, then verification fails and the code is rejected. Otherwise, verification succeeds, and since the abstract interpreter is a correct approximation of a defensive JVM, we are certain that a defensive JVM will not get stuck either executing the code. Thus, the code is correct and can be executed safely by a regular, non-defensive JVM.

Branches and exception handlers introduce forks and joins in the control flow of the method. Thus, an instruction can have several predecessors, with different stack and register types "after" these predecessor instructions. Sun's bytecode verifier deals with this situation in the manner customary for data flow analysis: the state (stack type and register type) "before" an instruction is taken to be the least upper bound of the states "after" all predecessors of this instruction. For instance, assume classes C_1 and C_2 extend C, and we analyze a conditional construct that stores a value of type C_1 in register 0 in one arm, and a value of type C_2 in the other arm. (See Fig. 4.) When the two arms meet, register 0 is assumed to have type C, which is the least upper bound (the smallest common supertype) of C_1 and C_2.

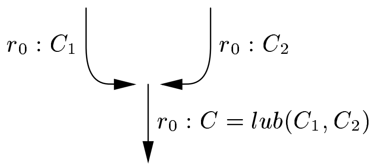


Fig. 4. Handling joins in the control flow

More precisely, writing $in(i)$ for the state "before" instruction i and $out(i)$ for the state "after" i, the algorithm sets up the following dataflow equations:

$$i : in(i) \rightarrow out(i)$$
$$in(i) = lub\{out(j) \mid j \text{ predecessor of } i\}$$

for every instruction i, plus

$$in(i_0) = (\varepsilon, (P_0, \ldots, P_{n-1}, \top, \ldots, \top))$$

for the start instruction i_0 (the P_k are the types of the method parameters). These equations are then solved by standard fixpoint iteration using Kildall's worklist algorithm [17, section 8.4]: an instruction i is taken from the worklist and its state "after" $out(i)$ is determined from its state "before" $in(i)$ using the abstract interpreter; then, we replace $in(j)$ by $lub(in(j), out(i))$ for each successor j of i, and enter those successors j for which $in(j)$ changed in the worklist. The fixpoint is reached when the worklist is empty, in which case verification succeeds. Verification fails if a state with no transition is encountered, or one of the least upper bounds is undefined.

As a trivial optimization of the algorithm above, the dataflow equations can be set up at the level of extended basic blocks rather than individual instructions. In other terms, it suffices to keep in working memory the states $in(i)$ where i is the first instruction of an extended basic block (i.e. a branch target); the other states can be recomputed on the fly as needed.

The least upper bound of two states is taken pointwise, both on the stack types and the register types. It is undefined if the stack types have different heights, which causes verification to fail. This situation corresponds to a program point where the run-time stack can have different heights depending on the path by which the point is reached; such code must be rejected because it can lead to unbounded stack height, and therefore to stack overflow. (Consider a loop that pushes one more entry on the stack at each iteration.)

The least upper bound of two register types can be $\top$, causing this register to have type $\top$ in the merged state. This corresponds to the situation where a register holds values of incompatible types in two arms of a conditional (e.g. `int` in one arm and an object reference in the other), and therefore is treated as uninitialized (no further loads from this register) after the merge point. The least upper bound of two stack slots can also be $\top$, in which case Sun's algorithm aborts verification immediately. Alternatively, it is entirely harmless to continue verification after setting the stack slot to $\top$ in the merged state, since the corresponding value cannot be used by any well-typed instruction, but simply discarded by instructions such as `pop` or `return`.

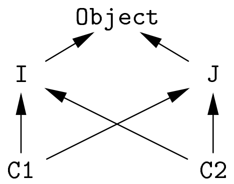
3.3 Interfaces and least upper bounds

The dataflow framework presented above requires that the type algebra, ordered by the subtyping relation, constitutes a semi-lattice. That is, every pair of types possesses a smallest common supertype (least upper bound).

Unfortunately, this property does not hold if we take the verifier type algebra to be the Java source-level type algebra (extended with $\top$ and `null`) and the subtyping relation to be the Java source-level assignment compatibility relation. The problem is that interfaces are types, just like classes, and a class can implement several interfaces. Consider the following classes:

```
interface I  { ... }
interface J  { ... }
class C1 implements I, J   { ... }
class C2 implements I, J   { ... }
```

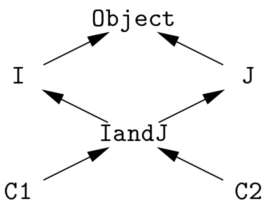
The subtyping relation induced by these declarations is:



This is obviously not a semi-lattice, since the two types `C1` and `C2` have two common super-types `I` and `J` that are not comparable (neither is subtype of the other).

There are several ways to address this issue. One approach is to manipulate sets of types during verification instead of single types as we described earlier. These sets of types are to be interpreted as conjunctive types, i.e. the set $\{\mathtt{I}, \mathtt{J}\}$, like the conjunctive type $\mathtt{I} \wedge \mathtt{J}$, represents values that have both types `I` and `J`, and therefore is a suitable least upper bound for the types $\{\mathtt{C1}\}$ and $\{\mathtt{C2}\}$ in the example above. This is the approach followed by Qian [25] and also by Pusch [24].

Another approach is to complete the class and interface hierarchy of the program into a lattice before performing verification. In the example above, the completion would add a pseudo-interface `IandJ` extending both `I` and `J`, and claim that `C1` and `C2` implement `IandJ` rather than `I` and `J`. We then obtain the following semi-lattice:



The pseudo-interface `IandJ` plays the same role as the set type $\{\texttt{I},\texttt{J}\}$ in the first approach described above. The difference is that the completion of the class/interface hierarchy is performed once and for all, and verification manipulates only simple types rather than sets of types. This keeps verification simple and fast.

The simplest solution to the interface problem is to be found in Sun's implementation of the JDK bytecode verifier. (This approach is documented nowhere, but can easily be inferred by experimentation.) Namely, bytecode verification ignores interfaces, treating all interface types as the class type `Object`. Thus, the type algebra used by the verifier contains only proper classes and no interfaces, and subtyping between proper classes is simply the inheritance relation between them. Since Java has single inheritance (a class can implement several interfaces, but inherit from one class only), the subtyping relation is tree-shaped and trivially forms a lattice: the least upper bound of two classes is simply their closest common ancestor in the inheritance tree.

The downside of Sun's approach, compared with the set-based or completion-based approach, is that the verifier cannot guarantee statically that an object reference implements a given interface. In particular, the `invokeinterface` $I.m$ instruction, which invokes method m of interface I on an object, is not guaranteed to receive at run-time an object that actually implements I: the only guarantee provided by Sun's verifier is that it receives an argument of type `Object`, that is, any object reference. The `invokeinterface` $I.m$ instruction must therefore check dynamically that the object actually implements I, and raise an exception if it does not.

3.4 Formalizations and proofs

Many formalizations and proofs of correctness of Java bytecode verification have been published, and we have reasons to believe that many more have been developed internally, both in academia and industry. With no claims to exhaustiveness, we will mention the works of Cohen [6] and Qian [25] among the first formal specifications of the JVM. Qian's specification is written in ordinary mathematics, while Cohen's uses the specification language of the ACL2 theorem prover. Pusch [24] uses the Isabelle/HOL prover to formalize the dynamic semantics of a fragment of the JVM, the corresponding type-level abstract interpreter used by the verifier, and proves the correctness of the latter with respect to the former: if the abstract interpreter can do a transition $i : (S, R) \rightarrow (S', R')$, then for all concrete states (s, r) matching (S, R), the concrete interpreter can do a transition $i : (s, r) \rightarrow (s', r')$, and the final concrete state (s', r') matches (S, R). Nipkow [20] formalizes the dataflow analysis framework in Isabelle/HOL and proves its correctness.

4 Verifying object initialization

Object creation in the Java virtual machine is a two-step process: first, the instruction `new` C creates a new object, instance of the class C, with all instance fields filled with default values (0 for numerical fields and `null` for reference fields); second, one of the initializer methods for class C (methods named C.`<init>` resulting from the compilation of the constructor methods of C) must be invoked on the newly created object. Initializer methods, just like their source-level counterpart (constructors), are typically used to initialize instance fields to non-default values, although they can also perform nearly arbitrary computations.

The JVM specification requires that this two-step object initialization protocol be respected. That is, the object instance created by the `new` instruction is considered uninitialized, and none of the regular object operations (i.e. store the object in a data structure, return it as method result, access one of its fields, invoke one of its methods) is allowed on this uninitialized object. Only when one of the initializer methods for its class is invoked on the new object and return normally is the new object considered fully initialized and usable like any other object.

Unlike the register initialization property, this object initialization property is not crucial to ensure type safety at run-time: since the `new` instruction initializes the instance fields of the new object with correct values for their types, type safety is not broken if the resulting default-initialized object is used right away without having called an initializer method. However, the object initialization property is important to ensure that some invariants between instance fields that is established by the constructor of a class actually hold for all objects of this class.

Static verification of object initialization is made more complex by the fact that initialization methods operate by side-effect: instead of taking an uninitialized object and returning an initialized object, they simply take an uninitialized object, update its fields, and return nothing. Hence, the code generated by Java compilers for the source-level statement `x = new C(arg)` is generally of the following form:

```
new C                       // create uninitialized instance of C
dup                         // duplicate the reference to this instance
code to compute arg
invokespecial C.<init>      // call the initializer
```

```
astore 3                // store initialized object in x
```

That is, two references to the uninitialized instance of C are held on the stack. The topmost reference is "consumed" by the invocation of `C.<init>`. When this initializer returns, the second reference is now at the top of the stack and now references a properly initialized object, which is then stored in the register allocated to `x`. The tricky point is that the initializer method is applied to one object reference on the stack, but it is another object reference contained in the stack (which happens to reference the same object) whose status goes from "uninitialized" to "fully initialized" in the process.

As demonstrated above, static verification of object initialization requires a form of alias analysis (more precisely a must-alias analysis) to determine which object references in the current state are guaranteed to refer to the same uninitialized object that is passed as argument to an initializer method. While any must-alias analysis can be used, Sun's verifier uses a fairly simple analysis, whereas an uninitialized object is identified by the position (program counter value) of the `new` instruction that created it. More precisely, the type algebra is enriched by the types $\overline{C}_p$ denoting an uninitialized instance of class C created by a `new` instruction at PC p. An invocation of an initializer method C.`<init>` checks that the first argument of the method is of type $\overline{C}_p$ for some p, then pops the arguments off the stack type as usual, and finally finds all other occurrences of the type $\overline{C}_p$ in the abstract interpreter state (stack type and register types) and replaces them by C. The following example shows how this works for a nested initialization corresponding to the Java expression `new C(new C(null))`:

```
 0: new C      // stack type after: C̄0
 3: dup                      //     C̄0, C̄0
 4: new C                    //     C̄0, C̄0, C̄4
 7: dup                      //     C̄0, C̄0, C̄4, C̄4
 8: aconst_null              //     C̄0, C̄0, C̄4, C̄4, null
 9: invokespecial C.<init>   //     C̄0, C̄0, C
12: invokespecial C.<init>   //     C
15: ...
```

In particular, the first `invokespecial` initializes only the instance created at PC 4, but not the one created at PC 0.

This approach is correct only if at any given time, the machine state contains at most one uninitialized object created at a given PC. Loops containing a `new` instruction can invalidate this assumption, since several distinct objects created by this `new` instruction can be "in flight", yet are given the same uninitialized object type (same class, same PC of creation). To avoid this problem, Sun's verifier requires that no uninitialized object type appear in the machine state when a backward branch is taken. Since a control-flow loop must take at least one backward branch, this guarantees that no initialized objects can be carried over from one loop iteration to the next one, thus ensuring the correctness of the "PC of creation" aliasing criterion.

Freund and Mitchell [7] formalize this approach to verifying object initialization. Bertot [2] proves the correctness of this approach using the Coq theorem prover, and extracts a verification algorithm from the proof.

5 Subroutines

Subroutines in the JVM are code fragments that can be called from several points inside the code of a method. To this end, the JVM provides two instructions: `jsr` branches to a given label in the method code and pushes a return address to the following instruction; `ret` recovers a return address (from a register) and branches to the corresponding instruction. Subroutines are used to compile certain exception handling constructs, and can also be used as a general code-sharing device. The difference between a subroutine call and a method invocation is that the body of the subroutine executes in the same activation record than its caller, and therefore can access and modify the registers of the caller.

5.1 The verification problem with subroutines

Subroutines complicate significantly bytecode verification by dataflow analysis. First, it is not obvious to determine the successors of a `ret` instruction, since the return address is a first-class value. As a first approximation, we can say that a `ret` instruction can branch to any instruction that follows a `jsr` in the method code. (This approximation is too coarse in practice; we will describe better approximations later.) Second, the subroutine entry point acts as a merge point in the control-flow graph, causing the register types at the points of call to this subroutine to be merged. This can lead to excessive loss of precision in the register types inferred, as the example in Fig. 5 shows.

```
                   // register 0 uninitialized here
  0: jsr 100       // call subroutine at 100
  3: ...

 50: iconst_0
 51: istore_0      // register 0 has type "int" here
 52: jsr 100       // call subroutine at 100
 55: iload_0       // load integer from register 0
 56: ireturn       // and return to caller
     ...
                   // subroutine at 100:
100: astore_1      // store return address in register 1
101: ...           // execute some code that does not use register 0
110: ret 1         // return to caller
```

Fig. 5. An example of subroutine

The two `jsr 100` at 0 and 52 have 100 as successor. At 0, register 0 has type $\top$; at 52, it has type `int`. Thus, at 100, register 0 has type $\top$ (the least upper bound of $\top$ and `int`). The subroutine body (between 101 and 110) does not modify register 0, hence its type at 110 is still $\top$. The `ret 1` at 110 has 3 and 55 as successors (the two instructions following the two `jsr 100`). Thus, at 55, register 0 has type $\top$ and cannot be used as an integer by instructions 55 and 56. This code is therefore rejected.

This behavior is counter-intuitive. Calling a subroutine that does not use a given register does not modify the run-time value of this register, so one could expect that it does not modify the verification-time type of this register either. Indeed, if the subroutine body was expanded inline at the two `jsr` sites, bytecode verification would succeed as expected.

The subroutine-based compilation scheme for the `try`...`finally` construct produces code very much like the above, with a register being uninitialized at one call site of the subroutine and holding a value preserved by the subroutine at another call site. Hence it is crucial that similar code passes bytecode verification. We will now see two refinements of the dataflow-based verification algorithm that achieve this goal.

5.2 Sun's solution

We first describe the approach implemented in Sun's JDK verifier. It is described informally in [15, section 4.9.6], and formalized in [29, 25]. This approach implements the intuition that a call to a subroutine should not change the types of registers that are not used in the subroutine body.

First, we need to make precise what a "subroutine body" is: since JVM bytecode is unstructured, subroutines are not syntactically delimited in the code; subroutine entry points are easily detected (as targets of `jsr` instructions), but it is not immediately apparent which instructions can be reached from a subroutine entry point. Thus, a dataflow analysis is performed, either before or in parallel with the main type analysis. The outcome of this analysis is a consistent labeling of every instruction by the entry point(s) for the subroutine(s) it logically belongs to. From this labeling, we can then determine, for each subroutine entry point ℓ, the return instruction $Ret(\ell)$ for the subroutine, and the set of registers $Used(\ell)$ that are read or written by instructions belonging to that subroutine.

The dataflow equation for subroutine calls is then as follows. Let i be an instruction `jsr` ℓ, and j be the instruction immediately following i. Let $(S_{jsr}, R_{jsr}) = out(i)$ be the state "after" the `jsr`, and $(S_{ret}, R_{ret}) = out(Ret(\ell))$ be the state "after" the `ret` that terminates the subroutine. Then:

$$in(j) = \left(S_{ret}, \{r \mapsto \begin{cases} R_{ret}(r) & \text{if } r \in Used(\ell) \\ R_{jsr}(r) & \text{if } r \notin Used(\ell) \end{cases} \} \right)$$

In other terms, the state "before" the instruction j following the `jsr` is identical to the state "after" the `ret`, except for the types of the registers that are not used by the subroutine, which are taken from the state "after" the `jsr`.

In the example above, we have $Ret(100) = 110$ and register 0 is not in $Used(100)$. Hence the type of register 0 before instruction 55 (the instruction following the `jsr`) is equal to the type after instruction 52 (the `jsr` itself), that is `int`, instead of $\top$ (the type of register 0 after the `ret 1` at 110).

While effective in practice, Sun's approach to subroutine verification raises a challenging issue: determining the subroutine structure is difficult. Not only subroutines are not syntactically delimited, but return addresses are stored in general-purpose registers rather than on a subroutine-specific stack, which makes tracking return addresses and matching `ret`/`jsr` pairs more difficult. To facilitate the determination of the subroutine structure, the JVM specification states a number of restrictions on correct JVM code, such as "two different subroutines cannot 'merge' their execution to a single `ret` instruction" [15, section 4.9.6]. These restrictions seem rather ad-hoc and specific to the particular subroutine labeling algorithm that Sun's verifier uses. Moreover, the description of subroutine labeling given in the JVM specification is very informal and incomplete.

Several rational reconstructions of this part of Sun's verifier have been published. The first, due to Abadi and Stata [29], is presented as a non-standard type system, and determines the subroutine structure before checking the types. The second is due to Qian [26] and infers simultaneously the types and the subroutine structure, in a way that is closer to Sun's implementation. The simultaneous determination of types and $Used(\ell)$ sets complicates the dataflow analysis: the transfer function of the analysis is no longer monotonous, and special iteration strategies are required to reach the fixpoint. Finally, O'Callahan [21] and Hagiya and Tozawa [10] also give non-standard type systems for subroutines based on continuation types and context-dependent types, respectively. However, these papers give only type checking rules, but no effective verification (type inference) algorithms.

While these works shed considerable light on the issue, they are carried in the context of a small subset of the JVM that excludes exceptions and object initialization in particular. Delicate interactions between subroutines and object initialization were discovered later by Freund and Mitchell [7], exposing a bug in Sun's verifier. As for exceptions, exception handling complicates significantly the determination of the subroutine structure. Examination of bytecode produced by Java compiler show two possible situations: either an exception handler covers a range of instructions entirely contained in a subroutine, in which case the code of the exception handler should be considered as part of the same subroutine (e.g. it can branch back to the `ret` instruction that terminates the subroutine); or, an exception handler covers both instructions belonging to a subroutine and non-subroutine instructions, in which case the code of the handler should be considered as outside the subroutine. The problem is that in the second case, we have a branch (via the exception handler) from a subroutine instruction to a non-subroutine instruction, and this branch is not a `ret` instruction; this situation is not allowed in Abadi and Stata's subroutine labeling system.

5.3 Polyvariant dataflow analysis

An alternate solution to the subroutine problem, used in the Java Card off-card verifier [31], relies on a polyvariant dataflow analysis: instructions inside subroutine bodies are analyzed several times, once per call site for the subroutine. The principles of polyvariant flow analyses, also called context-sensitive analyses, are well known [19, section 3.6]: whereas monovariant analyses maintain only one state per program point, a polyvariant analysis allows several states per program point. These states are indexed by *contours* that usually approximate the control-flow path that led to each state.

In the case of bytecode verification, contours are subroutine call stacks: lists of return addresses for the `jsr` instructions that led to the corresponding state. In the absence of subroutines, all the bytecode for a method is analyzed in the empty contour. Thus, only one state is associated to each instruction and the analysis degenerates into the monovariant dataflow analysis of section 3.2. However, when a `jsr` ℓ instruction is encountered in the current contour c, it is treated as a branch to the instruction at ℓ in the augmented contour $\ell.c$. Similarly, a `ret` r instruction is treated as a branch that restricts the current context c by popping one or several return addresses from c (as determined by the type of the register r).

In the example of Fig. 5, the two `jsr 100` instructions are analyzed in the empty context ε. This causes two "in" states to be associated with the instruction at 100; one has contour $3.\varepsilon$, assigns type $\top$ to register 0, and contains $\texttt{retaddr}(3)$ at the top of the stack[1]; the other state has contour $55.\varepsilon$, assigns type `int` to register 0, and contains $\texttt{retaddr}(55)$ at the top of the stack. Then, the instructions at 101…110 are analyzed twice, in the two contours $3.\varepsilon$ and $55.\varepsilon$. In the contour $3.\varepsilon$, the `ret 1` at 110 is treated as a branch to 3, where register 0 still has type $\top$. In the contour $55.\varepsilon$, the `ret 1` is treated as a branch to 55 with register 0 still having type `int`. By analyzing the subroutine body in a polyvariant way, under two different contours, we avoided merging the types $\top$ and `int` of register 0 at the subroutine entry point, and thus obtained the desired type propagation behavior for register 0: $\top$ before and after the `jsr 100` at 3, but `int` before and after the `jsr 100` at 52.

More formally, the polyvariant dataflow equation for a `jsr` ℓ instruction at i followed by an instruction at j is

$$in(\ell, j.c) = (\texttt{retaddr}(j).S, T) \text{ where } (S,T) = out(i,c)$$

For a `ret` r instruction at i, the equation is

$$in(ra, c') = out(i,c)$$

where the type of register r in the state $out(i,c)$ is $\texttt{retaddr}(ra)$ and the context c' is obtained from c by popping return addresses until ra is found, that is, $c = c''.ra.c'$.

[1] The type $\texttt{retaddr}(i)$ represents a return address to the instruction at i.

Another way to view polyvariant verification is that it is exactly equivalent to performing monovariant verification on an expanded version of the bytecode where every subroutine call has been replaced by a distinct copy of the subroutine body. Instead of actually taking N copies of the subroutine body, we analyze them N times in N different contours. Of course, duplicating subroutine bodies before the monovariant verification is not practical, because it requires prior knowledge of the subroutine structure (to determine which instructions are part of which subroutine body), and as shown in section 5.2, the subroutine structure is hard to determine exactly. The beauty of the polyvariant analysis is that it determines the subroutine structure along the way, via the computations on contours performed during the dataflow analysis. Moreover, this determination takes advantage of typing information such as the `retaddr`(ra) types to determine with certainty the point to which a `ret` instruction branches in case of early return from nested subroutines.

Another advantage of polyvariant verification is that it has no problem dealing with code that is reachable both from subroutine bodies and from the main program, such as the exception handlers mentioned at the end of section 5.2: rather than deciding whether such exception handlers are part of a subroutine or not, the polyvariant analysis simply analyzes them several times, once in the empty contour and once or several times in subroutine contours.

The downside of polyvariant verification is that it is more computationally expensive than Sun's approach. In particular, if subroutines are nested to depth N, and each subroutine is called k times, the instructions from the innermost subroutine are analyzed k^N times instead of only once in Sun's algorithm. However, typical Java code has low nesting of subroutines: most methods have $N \leq 1$, very few have $N = 2$, and $N > 2$ is unheard of. Hence, the extra cost of polyvariant verification is entirely acceptable in practice.

6 Model checking of abstract interpretations

It is folk lore that dataflow analyses can be viewed as model checking of abstract interpretations [28]. Since a large part of bytecode verification is obviously an abstract interpretation (of a defensive JVM at the type level), it is natural to look at the remaining parts from a model-checking perspective.

Posegga and Vogt [22] were the first to do so. They outline an algorithm that takes the bytecode for a method and generates a temporal logic formula that holds if and only if the bytecode is safe. They then use an off-the-shelf model checker to determine the validity of the formula. While this application uses only a small part of the power and generality of temporal logic and of the model checker, the approach sounds interesting for establishing finer properties of the bytecode that go beyond the basic safety properties of bytecode verification (see section 8).

Unpublished work by Brisset [3] extracts the essence of Posegga and Vogt's approach: the idea of exploring all reachable states of the abstract interpreter. Brisset considers the transition relation obtained by combining the transition relation of the type-level abstract interpreter (Fig. 2) with the "successor" relation between instructions. This relation is of the form $(p, S, R) \to (p', S', R')$, meaning that the abstract interpreter, started at PC p with stack type S and register type R, can abstractly execute the instruction at p and arrive at PC p' with stack type S' and register type R'.

Starting with the initial state $(0, \varepsilon, (P_0, \ldots, P_{n-1}, \top, \ldots, \top))$ corresponding to the method entry, we can then explore all states reachable by repeated applications of the transition function. If we encounter a state where the abstract interpreter is "stuck" (cannot make a transition because some check failed), verification fails and the bytecode is rejected. Otherwise, the correctness of the abstract interpretation guarantees that the concrete, defensive JVM interpreter will never get "stuck" either during the execution of the method code, hence the bytecode is safe.

This algorithm always terminates because the number of distinct states is finite (albeit large), since there is a finite number of distinct types used in the program, and the height of the stack is bounded, and the number of registers is fixed. Brisset formalized and proved the correctness of this approach in the Coq proof assistant, and extracted the ML code of a bytecode verifier from the proof.

This approach is conceptually interesting because it is the ultimate polyvariant analysis: rather than having one stack-register type per control point (as in Sun's verifier), or one such type per control point and per subroutine contour (as in section 5.3), we can have arbitrarily many stack-register types per control point, depending on the number of control-flow paths that lead to this control point. Consider for instance the control-flow joint depicted in Fig. 4. While the dataflow-based algorithms verify the instructions following the join point only once under the assumption $r : lub(C_1, C_2) = C$, Brisset's algorithm verifies them twice, once under the assumption $r : C_1$, once under the assumption $r : C_2$.

In other terms, this analysis is polyvariant not only with respect to subroutine calls, but to all conditional or N-way branches as well. This renders the analysis impractical, since it runs in time exponential in the number of such branches in the method. (Consider a control-flow graph with N conditional constructs in sequence, each assigning a different type to registers $r_1 \ldots r_N$; this causes the code following the last conditional to be verified 2^N times under 2^N different register types.)

Of course, the precision of Brisset's algorithm can be degraded by applying widening steps in order to reduce the number of states. Some transitions $(pc, S, R) \to (pc', S', R')$ can be replaced by $(pc, S, R) \to (pc', S'', R'')$ where $R' <: R''$ and $S' <: S''$. If the abstract interpreter is still not stuck on any of the reachable states, the bytecode remains safe. The monovariant dataflow analysis of section 3.2 corresponds to keeping only one state per program point by replacing multiple states by their least upper bounds. The polyvariant dataflow analysis of section 5.3 is similar, except that the merging of states into least upper bounds is relaxed for subroutines and controlled via contours.

Another interest of Brisset's approach is that it allows us to reconsider some of the design decisions explained in sections 3.3 and 4. For instance, Brisset's algorithm never computes least upper bounds of types, but simply checks subtyping relations between types. Thus, it can be applied to any subtyping relation, not just relations that form a semi-lattice. Indeed, it can keep track of interface types and verify `invokeinterface` instructions accurately, without having to deal with sets of types or lattice completion.

7 Bytecode verification on small computers

Java virtual machines run not only in personal computers and workstations, but also in a variety of embedded computers, such as personal digital assistants, mobile phones, and smart cards. Extending the Java model of safe post-issuance code downloading to these devices requires that bytecode verification be performed on the embedded system itself. However, bytecode verification is an expensive process that exceeds the resources (processing power and memory space) of small embedded systems. For instance, a typical Java card (Java-enabled smart card) has 1 or 2 *kilo*-bytes of RAM and an 8-bit microprocessor that is approximately 1000 times slower than a personal computer. Fitting a bytecode verifier into one of these devices requires new verification algorithms, which we discuss now.

7.1 Lightweight bytecode verification using certificates

Inspired by Necula and Lee's proof-carrying code [18], Rose and Rose [27] propose to split bytecode verification into two phases: the code producer computes the stack and register types at branch targets and transmit these so-called certificates along with the bytecode; the embedded system, then, simply checks that the code is well-typed with respect to the types given in the certificates, rather than inferring these types itself. In other terms, the embedded system no longer solves iteratively the dataflow equations characterizing correct bytecode, but simply checks that the types provided in the code certificates are indeed a solution of these equations.

The benefits of this approach are twofold. First, checking a solution is faster than inferring one, since we avoid the cost of the fixpoint iteration. This speeds up verification to some extent[2]. Second, certificates are only read, but never modified during verification. Hence, they can be stored in persistent rewritable memory (EEPROM or Flash). Smart card-class embedded systems offer relatively large amounts of persistent memory (e.g. 16-32 kilo-bytes). Writing data to such memory is slow (1000-10000 times slower than reading from it), hence it is not possible to store there rapidly-changing data such as the fixpoint computed by a standard verification algorithm. However, Rose and Rose's certificates are written only once, on reception of the bytecode, and only read during verification, so they can fit in the "comfortable" EEPROM memory space.

[2] The speedup is not as important as one might expect, since experiments show that the fixpoint is usually reached after examining every instruction at most twice [13].

There are two limitations to this approach. First, it is currently not known how to deal with subroutines in this framework. Indeed, Sun proposed to drop subroutines entirely in order to use Rose and Rose's bytecode verification algorithm in the KVM, one of Sun's embedded variants of the JVM [30]. Second, certificates are relatively large: without compression, about the same size as the code they annotate; with compression, about 20% of the code size. Even if certificates are stored in persistent memory, they can still exceed the available memory space.

7.2 On-card verification with off-card code transformation

The Java Card bytecode verifier described in [13] attacks the memory problem from another angle. Like the standard bytecode verifier, it solves dataflow equations using fixpoint iteration. To reduce memory requirements, however, it has only one global register type that is shared between all control points in the method. In other terms, the solution it infers is such that a given register has the same type throughout the method. For similar reasons, it also requires that the stack be empty at each branch instruction and at each branch target instruction. With these extra restrictions, bytecode verification can be done in space $O(M_{stack} + M_{reg})$, instead of $O(N_{branch} \times (M_{stack} + M_{reg}))$ for Sun's algorithm, where N_{branch} is the number of branch targets. In practice, the memory requirements are small enough that all data structures comfortably fit in RAM on a smart card.

One drawback of this approach is that register initialization can no longer be checked statically, and must be replaced by run-time initialization of registers to safe values (0 or `null`) on method entrance. Another drawback is that the extra restrictions imposed by the on-card verifier cause perfectly legal bytecode (that passes Sun's verifier) to be rejected. To address the latter issue, we rely on an off-card transformation, performed on the bytecode of the applet, that transforms any legal bytecode (that passes Sun's verifier) into equivalent bytecode that passes the on-card verifier. The off-card transformations include stack normalizations around branches and register reallocation by graph coloring, and increase the size of the code by less than 2% [13].

8 Conclusions and perspectives

Java bytecode verification is now a well researched technique, although it is still defined only by Sun's reference implementation: all the formal works reviewed in this paper have not yet resulted in a complete formal specification of what it is and what it guarantees.

A largely open question is whether bytecode verification can go beyond basic type safety and initialization properties, and statically establish more advanced properties of applets, such as resource usage (bounding the amount of memory allocated) and reactiveness (bounding the running time of an applet between two interactions with the outside world). Controlling resource usage is especially important for Java Card applets: since Java Card does not guarantee the presence of a garbage collector, applets are supposed to allocate all the objects they need at installation time, then run in constant space.

Other properties of interest include access control and information flow. Currently, the Java security manager performs all access control checks dynamically. Various static analyses and program transformations have been proposed to perform some of these checks statically [35, 23]. As for information flow (an applet does not "leak" confidential information that it can access), this property is essentially impossible to check dynamically; several type systems have been proposed to enforce it statically [34, 33, 11, 1].

Finally, the security of the sandbox model relies not only on bytecode verification, but also on the proper implementation of the API given to the applet. The majority of known applet-based attacks exploit (in a type-safe way) bugs in the API, rather than breaking type safety through bugs in the verifier. Verification of the API is a promising and largely open area of application for formal methods [14, 12].

References

1. M. Abadi, A. Banerjee, N. Heintze, and J. G. Riecke. A core calculus of dependency. In *26th symp. Principles of Progr. Lang*, pages 147–160. ACM Press, 1999.
2. Y. Bertot. A Coq formalization of a type checker for object initialization in the Java virtual machine. Research report 4047, INRIA, 2000. Also published in the proceedings of CAV'01.
3. P. Brisset. Vers un vérifieur de bytecode Java certifié. Seminar given at Ecole Normale Supérieure, Paris, October 2nd 1998.
4. K. Brunnstein. Hostile ActiveX control demonstrated. *RISKS Forum*, 18(82), Feb. 1997.
5. Z. Chen. *Java Card Technology for Smart Cards: Architecture and Programmer's Guide*. The Java Series. Addison-Wesley, 2000.
6. R. Cohen. The defensive Java virtual machine specification. Technical report, Computational Logic Inc., 1997.
7. S. N. Freund and J. C. Mitchell. A type system for object initialization in the Java bytecode language. *ACM Trans. Prog. Lang. Syst.*, 22(5), 2000.
8. L. Gong. *Inside Java 2 platform security: architecture, API design, and implementation*. The Java Series. Addison-Wesley, 1999.
9. J. A. Gosling. Java intermediate bytecodes. In *Proc. ACM SIGPLAN Workshop on Intermediate Representations*, pages 111–118. ACM, 1995.
10. M. Hagiya and A. Tozawa. On a new method for dataflow analysis of Java virtual machine subroutines. In G. Levi, editor, *SAS'98*, volume 1503 of *LNCS*, pages 17–32. Springer-Verlag, 1998.
11. N. Heintze and J. G. Riecke. The SLam calculus: programming with secrecy and integrity. In *25th symp. Principles of Progr. Lang*, pages 365–377. ACM Press, 1998.
12. M. Huisman, B. Jacobs, and J. van den Berg. A case study in class library verification: Java's Vector class. Technical Report CSI-R0007, Computing Science Institute, University of Nijmegen, 2000.

13. X. Leroy. On-card bytecode verification for Java Card. Submitted for publication, available from `http://cristal.inria.fr/~xleroy`, 2001.
14. X. Leroy and F. Rouaix. *Security properties of typed applets*, volume 1603 of *LNCS*, pages 147–182. Springer-Verlag, 1999.
15. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. The Java Series. Addison-Wesley, 1999. Second edition.
16. G. McGraw and E. Felten. *Securing Java*. John Wiley & Sons, 1999.
17. S. S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann, 1997.
18. G. C. Necula. Proof-carrying code. In *POPL'97*, pages 106–119. ACM Press, 1997.
19. F. Nielson, H. R. Nielson, and C. Hankin. *Principles of program analysis*. Springer-Verlag, 1999.
20. T. Nipkow. Verified bytecode verifiers. In *Foundations of Software Science and Computation Structures (FOSSACS'01)*. Springer-Verlag, 2001. To appear.
21. R. O'Callahan. A simple, comprehensive type system for Java bytecode subroutines. In *POPL'99*, pages 70–78. ACM Press, 1999.
22. J. Posegga and H. Vogt. Java bytecode verification using model checking. In *Workshop Fundamental Underpinnings of Java*, 1998.
23. F. Pottier, C. Skalka, and S. Smith. A systematic approach to static access control. In D. Sands, editor, *Proceedings of the 10th European Symposium on Programming (ESOP'01)*, volume 2028 of *LNCS*, pages 30–45. Springer-Verlag, 2001.
24. C. Pusch. Proving the soundness of a Java bytecode verifier specification in Isabelle/HOL. In W. R. Cleaveland, editor, *TACAS'99*, volume 1579 of *LNCS*, pages 89–103. Springer-Verlag, 1999.
25. Z. Qian. A formal specification of Java virtual machine instructions for objects, methods and subroutines. In J. Alves-Foss, editor, *Formal syntax and semantics of Java*, volume 1523 of *LNCS*. Springer-Verlag, 1998.
26. Z. Qian. Standard fixpoint iteration for Java bytecode verification. *ACM Trans. Prog. Lang. Syst.*, 22(4):638–672, 2000.
27. E. Rose and K. Rose. Lightweight bytecode verification. In *Workshop Fundamental Underpinnings of Java*, 1998.
28. D. A. Schmidt. Data flow analysis is model checking of abstract interpretations. In *POPL'98*, pages 38–48. ACM Press, 1998.
29. R. Stata and M. Abadi. A type system for Java bytecode subroutines. *ACM Trans. Prog. Lang. Syst.*, 21(1):90–137, 1999.
30. Sun Microsystems. Java 2 platform micro edition technology for creating mobile devices. White paper, `http://java.sun.com/products/cldc/wp/KVMwp.pdf`, 2000.
31. Trusted Logic. Off-card bytecode verifier for Java Card. Distributed as part of Sun's Java Card Development Kit, 2001.
32. G. Vigna, editor. *Mobile Agents and Security*, volume 1419 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.
33. D. Volpano and G. Smith. A type-based approach to program security. In *Proceedings of TAPSOFT'97, Colloquium on Formal Approaches in Software Engineering*, volume 1214 of *LNCS*, pages 607–621. Springer-Verlag, 1997.
34. D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):1–21, 1996.
35. D. Walker. A type system for expressive security policies. In *27th symp. Principles of Progr. Lang*, pages 254–267. ACM Press, 2000.
36. F. Yellin. Low level security in Java. In *Proceedings of the Fourth International World Wide Web Conference*, pages 369–379. O'Reilly, 1995.