
Journal of Automated Reasoning **30:** 235–269, 2003.

Java Bytecode Verification: Algorithms and Formalizations

XAVIER LEROY
INRIA Rocquencourt and Trusted Logic S.A. e-mail: 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 reviews the various bytecode verification algorithms that have been proposed, recasts them in a common framework of dataflow analysis, and surveys the use of proof assistants to specify bytecode verification and prove its correctness.

Key words: bytecode verification, Java Virtual Machine, dataflow analysis, abstract interpretation, subroutines.

1. Introduction

Web applets have popularized the idea of downloading and executing untrusted compiled code on the personal computer running the Web browser, without the 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 a home-banking software [5]), divulging personal information over the network, or modifying other programs (Trojan attacks).

Beyond Web services, the applet model is now being transferred to high-security embedded devices such as smart cards: The Java Card architecture [6] allows for postissuance downloading of applets on smart cards. Smart cards are used as security tokens in sensitive application areas such as payment, mobile telephony, and authentication. Hence, the security issues with applets are even more acute.

The solution put forward by the Java programming environment is to execute the applets in a so-called sandbox, an insulation layer preventing direct access to the hardware resources and implementing a suitable access control policy [18, 54, 34]. 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 [19, 58, 33].

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 this work on bytecode verification. We explain what bytecode verification is, describe the various algorithms that have been proposed, outline the main problems they face, and give references to machine-assisted proofs of correctness.

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 polyvariant verification algorithms that address the subroutine issue. Section 7 discusses some issues specific to low-resources embedded systems, and Section 8 presents conclusions and perspectives.

2. Overview of the JVM and of Bytecode Verification

The Java Virtual Machine (JVM) [33] 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

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
```

Figure 1. An example of JVM bytecode.

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 intramethod branch instructions: unconditional branch ("goto"), conditional branches ("branch if top of stack is 0"), and multiway 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 Figure 1 gives the general flavor 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 contain at least two elements and that these two elements be 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

contain a reference to an instance of class C or one of its subclasses (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 at least 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 words, 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.

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 [11]. 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 faster, omitting the dynamic checks for the conditions above.

We emphasize that bytecode verification by itself does not guarantee secure execution of the code: Many crucial properties of the code still need to be checked dynamically, for instance via array bounds checks and null pointer checks in the virtual machine and access control checks in the API. The purpose of bytecode verification is to shift the verifications listed above from run time to loading time.

3. Basic Verification by Dataflow Analysis

The first JVM bytecode verification algorithm is due to Gosling and Yellin at Sun [19, 58, 33]. 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

$\texttt{iconst}\ n\ :\ (S,\ R) \to (\texttt{int}.S,\ R)$ if $|S| < M_{stack}$

$\texttt{ineg}\ :\ (\texttt{int}.S,\ R) \to (\texttt{int}.S,\ R)$

$\texttt{iadd}\ :\ (\texttt{int}.\texttt{int}.S,\ R) \to (\texttt{int}.S,\ R)$

$\texttt{iload}\ n\ :\ (S,\ R) \to (\texttt{int}.S,\ R)$
if $0 \leq n < M_{reg}$ and $R(n) = \texttt{int}$ and $|S| < M_{stack}$

$\texttt{istore}\ n\ :\ (\texttt{int}.S,\ R) \to (S,\ R\{n \leftarrow \texttt{int}\})$ if $0 \leq n < M_{reg}$

$\texttt{aconst\_null}\ :\ (S,\ R) \to (\texttt{null}.S,\ R)$ if $|S| < M_{stack}$

$\texttt{aload}\ n\ :\ (S,\ R) \to (R(n).S,\ R)$
if $0 \leq n < M_{reg}$ and $R(n) <: \texttt{Object}$ and $|S| < M_{stack}$

$\texttt{astore}\ n\ :\ (\tau.S,\ R) \to (S,\ R\{n \leftarrow \tau\})$
if $0 \leq n < M_{reg}$ and $\tau <: \texttt{Object}$

$\texttt{getfield}\ C.f.\tau\ :\ (\tau'.S,\ R) \to (\tau.S,\ R)$ if $\tau' <: C$

$\texttt{putfield}\ C.f.\tau\ :\ (\tau_1.\tau_2.S,\ R) \to (S,\ R)$ if $\tau_1 <: \tau$ and $\tau_2 <: C$

$\texttt{invokestatic}\ C.m.\sigma\ :\ (\tau'_n \ldots \tau'_1.S,\ R) \to (\tau.S,\ R)$
if $\sigma = \tau(\tau_1, \ldots, \tau_n)$, $\tau'_i <: \tau_i$ for $i = 1 \ldots n$, and $|\tau.S| \leq M_{stack}$

$\texttt{invokevirtual}\ C.m.\sigma\ :\ (\tau'_n \ldots \tau'_1.\tau'.S,\ R) \to (\tau.S,\ R)$
if $\sigma = \tau(\tau_1, \ldots, \tau_n)$, $\tau' <: C$, $\tau'_i <: \tau_i$ for $i = 1 \ldots n$, $|\tau.S| \leq M_{stack}$

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

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 (a sequence of types) and a register type (a tuple of types associating a type to each register number). 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.

Figure 2 defines more formally the abstract interpreter on a number of representative JVM instructions. The abstract interpreter is presented as a transition relation $i : (S, R) \to (S', R')$, 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

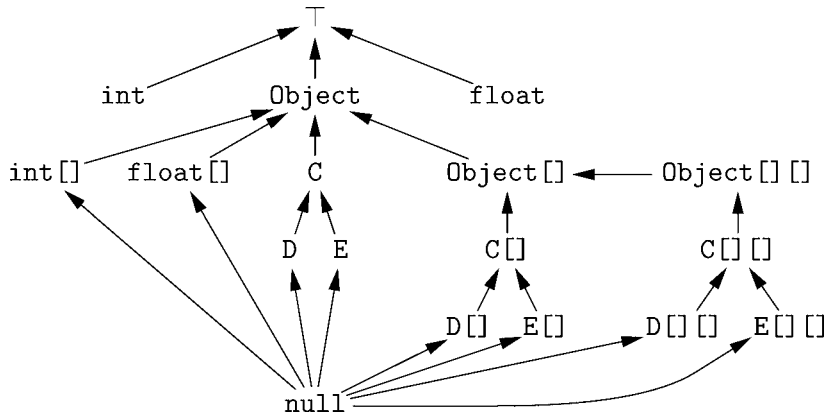


Figure 3. Some type expressions used by the verifier, with their subtyping relation. `C`, `D`, `E` are user-defined classes, with `D` and `E` extending `C`. Not all types are shown.

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` and `invokevirtual` instructions in Figure 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`), array types, and object reference types represented by the fully qualified names of the corresponding classes. The `boolean`, `byte`, `short`, and `char` types of Java are identified with `int`. Three extra types are introduced: `null` to represent the type of the null reference, $\bot$ to represent the absence of any value (at unreachable instructions), 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.)

These types are equipped with a subtyping relation, written $<:$, which is essentially identical to the subtyping relation of the Java language (the "assignment compatibility" predicate). Figure 3 illustrates the subtyping relation. Precise definitions of $<:$ can be found in [46, 38, 28, 26] but are omitted here. For the purposes of this article, all we require from the subtyping relation is the following property.

PROPERTY 1 (Well-founded semilattice). *The set of types ordered by the* $<:$ *relation is a semilattice: Any pair of types has a least upper bound. Moreover, the* $<:$

ordering is well founded: There do not exist infinite strictly increasing sequences of types.

The $<:$ relation between types is extended pointwise to register types and stack types. Two stack types are in the $<:$ relation only if they have the same size. It is easy to see that the $<:$ relation on register types and on stack types is well founded.

The type-level abstract interpreter must satisfy several formal properties. First and foremost is correctness with respect to the dynamic semantics of the defensive JVM: If the abstract interpreter can do a transition $i : (S, R) \to (S', R')$, then for all concrete states (s, r) matching (S, R), the defensive JVM, started in state (s, r), will not stop on a run-time type violation: It will either loop or transition to a state (s', r') that matches (S', R'). This correctness property was formalized and proved by several authors, including Pusch [43] (using the Isabelle/HOL prover), Coglio et al. [10] (using SpecWare), Goldberg [17], and Qian [44] (using standard mathematics).

Two other properties of the abstract interpreter are essential to ensure correctness and termination of the bytecode verification algorithms reviewed in this paper: determinacy of transitions, and monotonicity with respect to the subtyping relation.

PROPERTY 2 (Determinacy). *The transitions of the abstract interpreter define a partial function: If* $i : (S, R) \to (S_1, R_1)$ *and* $i : (S, R) \to (S_2, R_2)$*, then* $S_1 = S_2$ *and* $R_1 = R_2$.

PROPERTY 3 (Monotonicity). *If* $i : (S, R) \to (S', R')$*, then for all stack types* $S_1 <: S$ *and register types* $R_1 <: R$ *there exist a stack type* S_1' *and a register type* R_1' *such that* $i : (S_1, R_1) \to (S_1', R_1')$*, and moreover* $S_1' <: S'$ *and* $R_1' <: R$.

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," that is, 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, nondefensive JVM.

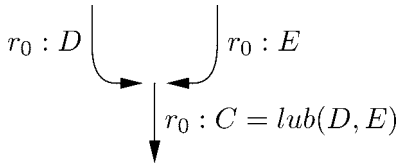


Figure 4. Handling joins in the control flow.

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 D and E extend C, and we analyze a conditional construct that stores a value of type D in register 0 in one arm, and a value of type E in the other arm. (See Figure 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 D and E.

More precisely, write $instr(p)$ for the instruction at program point p, $in(p)$ for the state "before" the instruction at p, and $out(p)$ for the state "after" the instruction at p. The verification algorithm sets up the following forward dataflow equations:

$$instr(p) : in(p) \rightarrow out(p)$$
$$in(p) = lub\{out(q) \mid q \text{ predecessor of } p\}$$

for every valid program point p, plus

$$in(0) = (\epsilon, (P_0, \ldots, P_{n-1}, \top, \ldots, \top))$$

for the entry point (the P_k are the types of the method parameters). These equations are then solved by standard fixpoint iteration using Kildall's worklist algorithm [35, Section 8.4]: A program point p is taken from the worklist, and its state "after" $out(p)$ is determined from its state "before" $in(p)$ using the abstract interpreter; then, we replace $in(q)$ by $lub(in(q), out(p))$ for each successor q of p and enter those successors q for which $in(q)$ 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 if 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 words, it suffices to keep in working memory the states $in(p)$ where p is the beginning 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 runtime stack can have different heights depending on the path by which the point is reached; such code cannot be proved correct in the framework described in this section, and must be rejected. (See Section 6.2 for alternative verification algorithm that can handle this situation.)

The least upper bound of two types for a register (or stack slot) 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.

Several formalizations and proofs (on paper or on machine) of the bytecode verification algorithm described above have been published. Nipkow and Klein's development in Isabell/HOL [38, 28, 26] is the closest to the dataflow presentation that we gave above. Other formalizations and correctness proofs of the dataflow approach include those of Qian [44], Coglio et al. [10], and Stärk et al. [49]. Bytecode verification can also be specified and proved sound by using type systems [51, 14, 13]; in this framework, the forward dataflow analysis algorithm that we described can be viewed as a type inference algorithm for these type systems. Hartel and Moreau [21] survey other approaches.

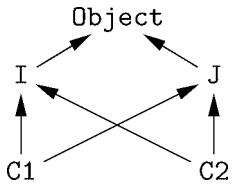
3.3. INTERFACES AND LEAST UPPER BOUNDS

The dataflow framework presented above requires that the type algebra, ordered by the subtyping relation constitutes a semilattice. 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  { ... }
```

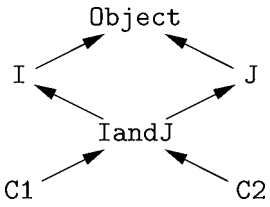
The subtyping relation induced by these declarations is



This is obviously not a semilattice because the two types `C1` and `C2` have two common supertypes `I` and `J` that are not comparable (neither is subtype of the other).

One can address this issue in several ways. 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; that is, the set {`I`, `J`}, like the conjunctive type `I` ∧ `J`, represents values that have both types `I` and `J` and therefore is a suitable least upper bound for the types {`C1`} and {`C2`} in the example above. This is the approach followed in [17, 44, 43, 49].

Another approach is to complete the class and interface hierarchy of the program into a lattice before performing verification [30]. This is an instance of a general mathematical construction known as the Dedekind–MacNeille completion of a poset. In the example above, the completion would add a point `IandJ` to the lattice of types, which is a subtype of both `I` and `J`, and a supertype of `C1` and `C2`. We then obtain the following semilattice:



The additional type `IandJ` plays the same role as the type set {`I`, `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.) Specifically, 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 semilattice: 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.

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 initialization methods for class C (methods named C.<init> resulting from the compilation of the constructors of class C) must be invoked on the newly created object. Initialization methods, just like their source-level counterpart (constructors), are typically used to initialize instance fields to nondefault 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 initialization methods for its class is invoked on the new object and returns normally is the new object considered fully initialized and usable like any other object. (Additional restrictions that we will not discuss here are imposed on initialization methods themselves; see [33, 15, 13].)

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. Thus, the verifier must prevent situations where several distinct objects created by the same instruction `new` C at p can be "in flight": These would be given the same type $\overline{C}_p$, and initializing one would cause the verifier to assume incorrectly that the others are also initialized. This situation could happen if a `new` instruction is executed repeatedly as part of a loop; another example involving subroutines is given in [15].

To avoid this problem, Sun's verifier requires that no uninitialized object type appear in the machine state when a backward branch is taken. Freund and Mitchell [15] formalize a simpler, equally effective restriction. They propose that when verifying a `new` C instruction at location p, there must be no occurrence of the type $\overline{C}_p$ in the stack type and register type at p. Bertot [3] proves the correctness of this approach using the Coq theorem prover and extracts a verification algorithm from

the proof. (Bertot's algorithm is not a standard dataflow analysis, since it features an additional pass of constraint solving.)

Both Sun's restriction and Freund and Mitchell's restriction are not monotone, in the sense that raising the type of a register or stack location from $\overline{C}_p$ to $\top$ can transform an abstract state from which no transitions are possible into an abstract state from which a transition is possible. In other words, Property 3 does not hold, and this fact causes difficulties with dataflow analysis. To address this issue, a modified formulation of Freund and Mitchell's restriction is introduced in [49] and used in [28, 26, 13]: A `new` C instruction at location p does not fail if there are occurrences of the type $\overline{C}_p$ in the stack and register types before p; instead, it sets these entries to $\top$ in the resulting state. This suffices to enforce nonaliasing (the unitialized objects whose types were set to $\top$ cannot be initialized nor used later on), while respecting the monotonicity property.

Subroutines, as described in Section 5, complicate the verification of object initialization. As discovered by Freund and Mitchell [15], a `new` instruction inside a subroutine can result in distinct uninitialized objects having the same static type $\overline{C}_p$, thus fooling Sun's verifier into believing that all of them become initialized after invoking an initialization method on one of them. The solution is to prohibit or set to $\top$ all registers and stack locations that have type $\overline{C}_p$ across a subroutine call.

Coglio [9] observes that Sun's restriction on backward branches as well as Freund and Mitchell's restriction on `new` are unnecessary for a bytecode verifier based on monovariant dataflow analysis. More precisely, [9, Section 5.8.2] shows that, in the absence of subroutines, a register or stack location cannot have the type $\overline{C}_p$ just before a program point p containing a `new` C instruction. Thus, the only program points where uninitialized object types in stack types or register types must be prohibited (or turned into $\top$) are subroutine calls.

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; and `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 as its caller and therefore can access and modify the registers of the caller.

```
                  // register 0 uninitialized here
  0: jsr 100      // call subroutine at 100
  3: ...
 50: iconst_0
 51: istore_0     // register 0 has type "nt" 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 calle
```

Figure 5. An example subroutine.

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 Figure 5 shows.

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 counterintuitive. 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. In the

remainder of this section and in Section 6, we will present 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 [33, Section 4.9.6], and formalized (with various degrees of completeness and faithfulness to Sun's implementation) in [51, 44, 49, 13, 50]. 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 let 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:

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

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 are subroutines 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" [33, 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 and formalizations of this part of Sun's verifier have been published. The presentations closest to Sun's implementation are due to Qian [44] and Stärk et al. [49]. A characteristic feature of Sun's implementation, correctly captured by these presentations, is that the subroutine structure and the $Used(\ell)$ sets are not determined prior to setting up and solving the dataflow equations, as we suggested above; instead, the types and the $Used(\ell)$ sets are inferred simultaneously during the dataflow analysis. This simultaneous computation of types and $Used(\ell)$ sets complicates the analysis. As shown by Qian [45], the transfer function of the dataflow analysis is no longer monotonous, and special iteration strategies are required to reach the fixpoint.

5.3. OTHER APPROACHES TO THE VERIFICATION OF SUBROUTINES

Many other approaches to the verification of subroutine have been proposed in the literature, often in the context of small subsets of the JVM, leading to algorithms that are simpler and more elegant but do not scale to the whole JVM.

The first formal work on subroutine verification is due to Abadi and Stata [51]: it relies on a separate analysis, performed before type verification proper, that labels bytecode instructions with the names of the subroutines they belong to, thus reconstructing the subroutine structure. The $Used(\ell)$ sets can then be computed for each subroutine ℓ, and injected in the dataflow equations as described in Section 5.2.

Later work by Hagiya and Tozawa [20] also relies on prior determination of the subroutine structure but expresses the flow of types through subroutines in a different way, using special types `last`(n) to refer to the type of register n in the caller of the subroutine. The `last`(n) types behave very much like type variables in a type system featuring parametric polymorphism. In other words, the subroutine is given a type that is polymorphic over the types of the local variables that it does not use.

While these works shed considerable light on the issue of subroutines, they are carried in the context of a small subset of the JVM that excludes exceptions and object initialization in particular. The delicate interactions between subroutines and object initialization were discussed in Section 4. As for exceptions, exception handling complicates significantly the determination of the subroutine structure. Examination of bytecode produced by a Java compiler shows 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 nonsubroutine 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 nonsubroutine instruction, and this branch is not a `ret` instruction; this situation is not allowed in Abadi and Stata's subroutine labeling system.

Thus, it is desirable to develop subroutine verification strategies that do not rely on prior determination of the subroutine structure but instead "discover" this structure during verification. The polyvariant verification algorithms that we discuss next in Section 6 satisfy this requirement. O'Callahan [39] proposes a different approach, based on continuation types: The type assigned to a return address contains the complete stack and register type expected at the program point following the `jsr` instruction. O'Callahan gives only type checking rules, however, but no effective type inference algorithm.

6. Polyvariant Bytecode Verification

In this paper so far, we have presented bytecode verification as a *monovariant* flow analysis: At each program point, only one abstract state (register type and stack type) is considered. *Polyvariant* flow analyses, also called context-sensitive analyses [37, Section 3.6] lift this restriction: Several states are allowed per program point. As we show in this section, polyvariant bytecode verification provides an alternative solution to the subroutine problem of Section 5: Polyvariant analysis allows instructions inside subroutine bodies to be analyzed several times, for example, once per call site for the subroutine, without merging the corresponding states as the monovariant analysis of Section 3.2 does.

6.1. POLYVARIANT VERIFICATION BASED ON CONTOURS

The first polyvariant bytecode verification analysis that we describe is based on *contours* and is used in the Java Card off-card verifier [53]. In contour-based polyvariant flow analysis, the distinct states maintained at each program point 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 sequence of `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. When a `jsr` ℓ instruction is encountered in the current contour c, however, 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 Figure 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 $3.\epsilon$, assigns type $\top$ to register 0, and contains `retaddr(3)`

at the top of the stack;* the other state has contour $55.\epsilon$, assigns type `int` to register 0, and contains `retaddr`(55) at the top of the stack. Then, the instructions at 101...110 are analyzed twice, in the two contours $3.\epsilon$ and $55.\epsilon$. In the contour $3.\epsilon$, the `ret 1` at 110 is treated as a branch to 3, where register 0 still has type $\top$. In the contour $55.\epsilon$, 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 we 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

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

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

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

where the type of register r in the state $out(i, c)$ is `retaddr`(j) and the context c' is obtained from c by popping return addresses until j is found, that is, $c = c''.j.c'$. Finally, for instructions i other than `jsr` and `ret`, the equation is simply

$$i : in(i, c) \rightarrow out(i, c);$$

that is, the instruction triggers no change of contour.

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 5.3, 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 an early return from nested subroutines.

Another advantage of polyvariant verification is that it handles code that is reachable both from subroutine bodies and from the main program, such as the exception handlers mentioned in Section 5.3: 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 type `retaddr`(i) represents a return address to the instruction at i. It is subtype of $\top$ and itself only.

```
while (true) {          0: invokestatic m
  try {                 3: jsr 15
    m();                6: goto 0
  } finally {           9: astore 0      // exception handler
    continue;          10: jsr 15
  }                    13: aload 0
}                      14: athrow
                       15: astore 1      // subroutine
                       16: goto 0
                       Exception table: [0, 2] → 9
```

Figure 6. The problem with contour-based polyvariant verification (left: Java source code; right: corresponding JVM bytecode).

One 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. Typical Java code has low nesting of subroutines, however. 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.

A more serious drawback of this approach is that contour-based polyvariant verification can fail to accept valid JVM code because the subroutine structure that it infers through the construction of contours can be infinite. Consider the Java method shown in Figure 6, along with the corresponding JVM code. The subroutine at 15 does not terminate by a `ret` instruction but by a `goto 0` that branches back to the nonsubroutine part of the method code. (This branch corresponds to the `continue` statement in the source code.)

Polyvariant verification starts by analyzing instructions at 0, 3, and 6 in the empty contour ϵ. The `jsr 15` at 3 causes instructions at 15 and 16 to be analyzed in the contour $15.\epsilon$. The `goto 0` at 16 does not affect the contour (unlike a `ret` instruction), thus causing instructions at 0, 3, and 6 to be reanalyzed in the contour $15.\epsilon$. Iterating this process, we end up analyzing instructions 0, 3, and 6 in an infinite number of contexts: ϵ, $15.\epsilon$, $15.15.\epsilon$, ... Thus, verification does not terminate.

Alternatively, termination can be ensured by requiring that contours never contain twice the same subroutine label: a `jsr` ℓ in a contour containing ℓ is rejected, in accordance with the JVM specification that states that subroutines cannot be recursive. (This is what the Java Card verifier [53] does.) But in this case we end up rejecting a valid piece of JVM bytecode, generated by compilation of a valid (albeit artificial) Java program.

As demonstrated by this example, using data-independent contours to direct polyvariant verification can cause too many – or even infinitely many – different

states to be kept for a given program point, even if these states are exactly identical. (In the example, the states at point 0 in contours $15.\epsilon$, $15.15.\epsilon$, etc., are all identical.) We now describe another approach to polyvariant verification, not using contours, that avoids these issues.

6.2. MODEL CHECKING OF ABSTRACT INTERPRETATIONS

It is folklore that dataflow analyses can be viewed as model checking of abstract interpretations [48]. 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 [40] 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). The article by Basin et al. in this volume [2] explores the model-checking approach to bytecode verification in great details.

Brisset [4] and independently Coglio [8] extract the essence of the model-checking approach: the idea of exploring all reachable states of the abstract interpreter. They consider the transition relation obtained by combining the transition relation of the type-level abstract interpreter (Figure 2) with the "successor" relation between instructions. This relation is of the form $(p, S, R) \rightarrow (p', S', R')$, 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'. Additional transitions $(p, S, R) \rightarrow \texttt{err}$ are introduce to reflect states (p, S, R) in which the abstract interpreter is "stuck" (cannot make a transition because some check failed).

$$
\begin{array}{ll}
(p, S, R) \rightarrow (p', S', R') & \text{if } \mathit{instr}(p) : (S, R) \rightarrow (S', R') \\
 & \text{and } p' \text{ is a successor of } p \\
(p, S, R) \rightarrow \texttt{err} & \text{if } \mathit{instr}(p) : (S, R) \not\rightarrow
\end{array}
$$

The BC (Brisset–Coglio) verification algorithm simply explores all states reachable by repeated applications of the extended transition relation starting with the initial state $\sigma_0 = (0, \epsilon, (P_0, \ldots, P_{n-1}, \top, \ldots, \top))$ corresponding to the method entry. In other words, writing $C(\Sigma) = \Sigma \cup \{\sigma' \mid \exists \sigma \in \Sigma.\ \sigma \rightarrow \sigma'\}$ for the one-step closure of a set of states Σ, the BC algorithm computes by fixpoint iteration the smallest set Σ_c containing σ_0 and closed under C. If $\texttt{err} \in \Sigma_c$, the error state is reachable and the bytecode is rejected. Otherwise, the bytecode passes verification. In the latter case (`err` is not reachable), the correctness of the abstract interpretation (as proved in [43]) guarantees that the concrete, defensive JVM interpreter

will never fail a safety check 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.

The problem with subroutines described in Section 5.1 completely disappears in this approach. It suffices to use the following transitions for the `jsr` and `ret` instructions:

$$
\begin{aligned}
(p, S, R) &\rightarrow (\ell, \texttt{retaddr}(p+3).S, R) \quad \text{if } \mathit{instr}(p) = \texttt{jsr } \ell \\
(p, S, R) &\rightarrow (q, S, R) \quad \text{if } \mathit{instr}(p) = \texttt{ret } r \text{ and } R(r) = \texttt{retaddr}(q) \\
(p, S, R) &\rightarrow \texttt{err} \quad \text{if } \mathit{instr}(p) = \texttt{ret } r \text{ and } R(r) \neq \texttt{retaddr}(q)
\end{aligned}
$$

The fact that the BC algorithm never merges the types inferred along two execution paths leading to the same instruction guarantees that subroutine bodies are analyzed as many times as necessary to propagate type information correctly across subroutine calls. However, we will never consider twice the same stack and register types at a given point, thus guaranteeing termination and avoiding the problem with contour-based polyvariance. For instance, the example of Figure 6 that cannot be accepted by contour-based polyvariant verification is now correctly accepted: instructions at 0, 3, and 6 are verified exactly twice, once under the assumption $r_0 : \top$, the other under the assumption $r_0 : \texttt{retaddr}(3)$.

Another interest of this approach is that it allows us to reconsider some of the design decisions explained in Sections 3.2, 3.3, and 4. In particular, the BC algorithm never computes least upper bounds of types, but simply checks subtyping relations between types. Thus, it can be applied to any well-founded subtyping relation, not just relations that form a semilattice. Indeed, it can keep track of interface types and verify `invokeinterface` instructions accurately, without having to deal with sets of types or lattice completion. Similarly, it is possible to verify code where the stack size is not the same on all paths leading to an instruction.

Brisset [4] 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. Klein and Wildmoser [29] also prove the correctness of this approach using Isabelle/HOL. Their proof builds on a generic dataflow analysis framework and thus seems reusable for the variants of the BC algorithm discussed in Section 6.3. Coglio [8] argues that the BC verification algorithm is the most precise of all "reasonable" verification algorithms, in the sense that it accepts all bytecode that does not crash a defensive virtual machine that would follow all execution paths across conditional jumps, regardless of the value of the condition. Thus, the only correct bytecode that could be rejected by the BC algorithm is bytecode whose correctness depends on additional knowledge on the values (and not just the types) of Booleans, integers, and object references manipulated by the code.

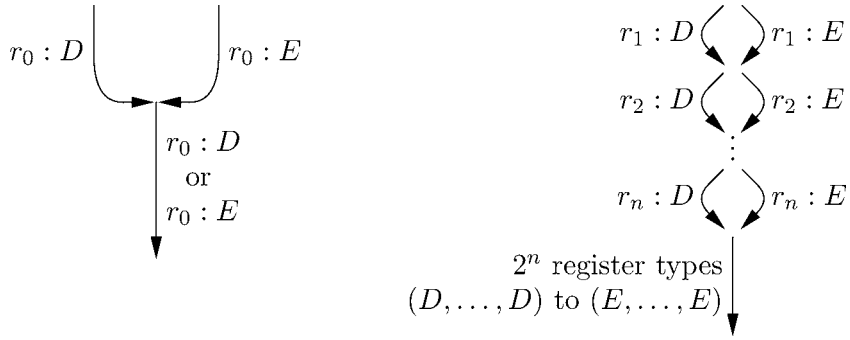


Figure 7. Control-flow joins in the model-checking approach.

6.3. WIDENING IN THE MODEL-CHECKING APPROACH

The BC verification algorithm based on model checking, described in Section 6.2, can be impractical: it runs in time exponential in the number of conditional or N-way branches in the method. Consider the control-flow joint depicted in Figure 7, left part. While the dataflow-based algorithms verify the instructions following the join point only once under the assumption $r : C$ where $C = lub(D, E)$, the BC algorithm verifies them twice, once under the assumption $r : D$, once under the assumption $r : E$. Consider now the control-flow graph shown in the right part of Figure 7. It comprises N such conditional constructs in sequence, each assigning a different type to registers $r_1 \ldots r_N$. This causes the instructions following the last conditional to be verified 2^N times under 2^N different register types.

One way to improve the efficiency of the BC algorithm is to reduce the number of states that need be explored via judicious application of *widening steps*: some transitions $(pc, S, R) \rightarrow (pc', S', R')$ can be replaced by $(pc, S, R) \rightarrow (pc', S'', R'')$ where $R' <: R''$ and $S' <: S''$. If the error state `err` is still not reachable, the bytecode remains safe. A widening step such as the above reduces the total number of states to be explored if we judiciously choose R'' and S'', for example, if the widened state (pc', S'', R'') is already reachable.

More formally, define a subtyping relation between states by $\sigma <:$ `err` for all states σ, and $(p, S, R) <: (p', S', R')$ if $p = p'$, $S <: S'$ and $R <: R'$. A widening scheme W is a function from sets of states to sets of states, such that for all state $\sigma \in \Sigma$, there exists a state $\sigma' \in W(\Sigma)$ such that $\sigma <: \sigma'$. Note that this property implies that if `err` $\in \Sigma$, then `err` $\in W(\Sigma)$.

Bytecode verification by "widened model checking," then, computes by fixpoint iteration of $W \circ C$ the smallest state set Σ_w containing the initial state σ_0 and such that $W(C(\Sigma_w)) = \Sigma_w$. The bytecode passes verification if and only if `err` $\notin \Sigma_w$.

It is easy to show that all bytecode accepted by a widened model-checking algorithm is also accepted by the BC model-checking algorithm and is therefore type-safe at run time. The proof is given in Appendix A.

As proposed in [23], we can construct interesting examples of widening functions from an equivalence relation $\approx$ determining which states should be merged in a single state, and a merging function M from sets of equivalent states to states such that $\sigma <: M(\Sigma)$ for all $\sigma \in \Sigma$. The widening function W is then defined by

$$W(\Sigma) = \{M(\Sigma_1), \ldots, M(\Sigma_n)\},$$

where $\Sigma_1, \ldots, \Sigma_n$ is a partition of Σ in equivalence classes for the $\approx$ relation. The W function thus defined is a widening function, since for every $\sigma \in \Sigma$, writing Σ_i for the equivalence class of σ in Σ, we have $\sigma <: M(\Sigma_i) \in W(\Sigma)$.

The monovariant dataflow analysis of Section 3.2 is a trivial instance of widened model checking where the widening function W merges all stack and register types associated with the same program point by taking their least upper bound. More precisely, the widening function is constructed as described above from the following equivalence relations and merging functions. The equivalence relation $\approx$ is defined by $\texttt{err} \approx \texttt{err}$ and $(p_1, S_1, R_1) \approx (p_2, S_2, R_2)$ if and only if $p_1 = p_2$. The merging function is defined:

$$\begin{aligned}
&M(\{\texttt{err}\}) = \texttt{err}\\
&M(\{(p, S_1, R_1), \ldots, (p, S_n, R_n)\}) = (p, lub(S_1, \ldots, S_n), lub(R_1, \ldots, R_n))\\
&\qquad\qquad\qquad\qquad\qquad\qquad \text{if } |S_1| = \cdots = |S_n|\\
&M(\{(p, S_1, R_1), \ldots, (p, S_n, R_n)\}) = \texttt{err}, \quad \text{otherwise.}
\end{aligned}$$

A more interesting widening function is described by Henrio and Serpette [23]. Similar ideas were implemented by Frey [16] and by Coglio [8]. They propose to merge stack and register types at a given program point if and only if these types agree on return addresses, that is, contain the same $\texttt{retaddr}(p)$ types in the same registers or stack slots, but may differ on non-`retaddr` types.

More precisely, we say that two types agree on return addresses if they are equal, or none of them is a `retaddr` type. Two register types agree on return addresses if the types they assign to every register agree. Two stack types agree on return addresses if they have the same size, and the types they contain agree pointwise. Then, define the equivalence relation $\approx$ by $\texttt{err} \approx \texttt{err}$ and $(p_1, S_1, R_1) \approx (p_2, S_2, R_2)$ if and only if $p_1 = p_2$ and S_1, S_2 agree on return addresses, and R_1, R_2 agree on return addresses. The merging function is defined by

$$\begin{aligned}
&M(\{\texttt{err}\}) = \texttt{err}\\
&M(\{(p, S_1, R_1), \ldots, (p, S_n, R_n)\} = (p, lub(S_1, \ldots, S_n), lub(R_1, \ldots, R_n)).
\end{aligned}$$

(By construction of $\approx$, M is never presented with stack types of different heights.) The CFHS (Coglio–Frey–Henrio–Serpette) verification algorithm, then, corresponds to widened model-checking using the widening function induced by the $\approx$ relation and M merging function.

In bytecode that contains no subroutines, stack and register types never contain return address types and thus always agree on return addresses; in this case, the

CFHS algorithm essentially reduces to standard, efficient monovariant dataflow analysis.* In the presence of subroutines, however, sufficient polyvariance is ensured because return address types prevent type merging inside subroutines. In the example of Figure 5, analysis of the two `jsr` at 0 and 52 causes two states with $pc = 100$ to be explored: one has `retaddr(3)` at the top of the stack type, and $\top$ as the type of register 0; the other has `retaddr(55)` at the top of the stack type, and `int` as the type of register 0. These two states disagree on return addresses (namely, on the top element of the stack type) and thus are not merged. This causes the `astore_1` at 100 to be analyzed twice, resulting in two states at 101 that are not mergeable either, because one has `retaddr(3)` for register 1 and the other has `retaddr(55)` for register 1. Thus, the instructions in the subroutine body (from 100 to 110) are given two independent sets of stack and register types, as needed to verify the two subroutine calls with sufficient precision.

In Appendix B, we prove that the CFHS algorithm accepts exactly the same bytecode as the BC algorithm. In other terms, it turns out that the additional merging of type information performed in the CFHS algorithm does not degrade the precision of the analysis. This is a consequence of the following "continuity" property of the type-level abstract interpreter with respect to least upper bounds.

PROPERTY 4 (Continuity). *Assume that the stack types S_1 and S_2 agree on return addresses, and the register types R_1 and R_2 agree on return addresses. Further assume that $i : (S_1, R_1) \rightarrow (S'_1, R'_1)$ and $i : (S_2, R_2) \rightarrow (S'_2, R'_2)$. Then, $i : (lub(S_1, S_2), lub(R_1, R_2)) \rightarrow (lub(S'_1, S'_2), lub(R'_1, R'_2))$.*

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 postissuance code downloading to these devices requires that bytecode verification be performed on the embedded system itself. However, bytecode verification is an expensive process that can exceed the resources (processing power and working memory space) of small embedded systems. For instance, a typical Java Card (Java-enabled smart card) has 1 or 2 kilobytes of working memory, 16 to 32 kilobytes of rewritable persistent memory,** and an 8-bit microprocessor that is approximately 1,000 times slower than a personal computer.

* One minor difference with monovariant dataflow analysis is that the latter fails if the stack height differs on several paths that reach a given program point, while the CFHS algorithm does not fail immediately in this case but analyzes the join point and its successors in a polyvariant manner, once per different stack height. At any rate, subroutine-free bytecode that passes monovariant verification also passes the CFHS algorithm, with only one state being considered for each program point.

** Rewritable persistent memory (EEPROM or Flash) is distinct from working memory (RAM) in that writes to the former take 1,000–10,000 times longer than writes to the latter. Moreover, rewritable persistent memory allows a limited number of writes to the same memory location.

On such small computing devices, a conventional bytecode verification algorithm (such as Sun's or any of the polyvariant verifiers described in this paper) cannot run in working memory: The amount of RAM available is too small to store the stack and register types inferred during bytecode verification. One avenue is to use persistent rewritable memory instead of RAM to hold the data structures of the verifier. It was long believed that this solution was impractical, because these data structures change rapidly during bytecode verification, resulting in excessive writing times and "wear" on the persistent memory. Recent work by Deville and Grimaud [12] develops specially designed encodings of types and memory representations of the verification data that alleviate this issue, and suggests that this approach is indeed feasible.

Another avenue for fitting a bytecode verifier into a small computing device is to design new verification algorithms that can run in tiny amounts of working memory. We now discuss two such algorithms.

7.1. LIGHTWEIGHT BYTECODE VERIFICATION USING CERTIFICATES

Inspired by Necula and Lee's proof-carrying code [36], Rose and Rose [47] 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 (in a single, linear pass over the bytecode) 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 verification to some extent.* Second, certificates are only read, but never modified during verification. Hence, they can be stored in persistent rewritable memory without risking to "wear" this memory space by repeated rewriting of the data during verification.

A practical limitation of this approach is that certificates are relatively large: about 50% of the size of the code they annotate [31]. Even if certificates are stored in persistent memory, they can still exceed the available memory space.

Later work by Rose [46] addresses the issue of the certificate size by noticing that type information for certain branch targets can be omitted from the certificate, as long as this type information is correctly computed by the verification algorithm during its linear scan of the bytecode. This hybrid verification strategy thus reduces the size of the certificate, at the cost of increased working memory

* The speedup is not so important as one might expect, since experiments show that the fixpoint is usually reached after examining every instruction at most twice [31].

requirements to store the inferred type information for branch targets not described in the certificate.

Lightweight bytecode verification is used in the KVM, one of Sun's embedded variants of the JVM [52]. It was formalized and proved sound and complete by Klein and Nipkow [27, 26], using the Isabelle/HOL prover. These presentations differ in their treatment of subroutines. Rose's presentation [47, 46] deals only with monovariant verification and does not handle subroutines. The KVM implementation follows this approach and relies on subroutine expansion before verification, as part of the certificate generation phase. Klein and Nipkow's formalization, however, is applicable not only to monovariant verification but to any verification algorithm that can be expressed using dataflow equations, including the polyvariant algorithms of Section 6.

7.2. ON-CARD VERIFICATION WITH OFF-CARD CODE TRANSFORMATION

The Java Card bytecode verifier described in [31] attacks the memory problem from another angle. Like the standard bytecode verifiers, 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 words, 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 expression 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 fit comfortably 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 runtime initialization of registers to a safe value (typically, the value of `null`) on method entry. 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, this verifier relies 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 are described in [31].

These transformations can increase the size of the code, as well as the number of registers used by a method. However, experience shows that these increases are minimal: On typical Java Card code, the increase in code size is less than 2%, and the increase in registers is negligible [31].

This off-card code transformation phase plays a role similar to that of adding certificates in Rose and Rose's approach: Both are off-card processing that adds enough information to the applet to facilitate its on-card verification. However, the

off-card code transformer embeds directly its information inside the code, via code transformations, instead of storing it into separate certificates. Moreover, the size of the extra information is much smaller (2% vs. 50%).

8. Conclusions and Perspectives

Java bytecode verification is now a well-researched technique. The considerable body of formal work reviewed in this paper led not only to a precise understanding of what bytecode verification is and what it guarantees, but also to a number of new verification algorithms (besides Sun's original implementation) that cover a large part of the precision/cost spectrum.

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, using stack inspection [18]. Various static analyses and program transformations have been proposed to perform some of these checks statically [25, 57, 42]. In the Java Card world, interapplet communications via shared objects raises delicate security issues; [7] applies model-checking technology to this problem. 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 [56, 55, 22, 1, 41].

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 bugs in the API in a type-safe way, rather than breaking type safety through bugs in the verifier. Verification of the API is a promising area of application for formal methods [32, 24].

Acknowledgments

We thank Alessandro Coglio, Ludovic Henrio, and the anonymous referees for their helpful comments and suggestions for improvements.

Appendix A. Correctness of Widened Model Checking

In this appendix, we formally prove that all bytecode accepted by a widened model-checking algorithm (Section 6.3) is also accepted by the BC model-checking algorithm (Section 6.2) and is therefore type-safe at run time. In the following, W stands for an arbitrary widening function.

LEMMA 1. *Let $\sigma_1, \sigma_2, \sigma_1'$ be three states such that $\sigma_1 <: \sigma_2$ and $\sigma_1 \rightarrow \sigma_1'$. Further assume that $\sigma_2 \neq \texttt{err}$ and $\sigma_2 \not\rightarrow \texttt{err}$. Then, there exists σ_2' such that $\sigma_2 \rightarrow \sigma_2'$ and $\sigma_1' <: \sigma_2'$.*

Proof. We first show that $\sigma_1' \neq \texttt{err}$. Assume, by way of contradiction, that $\sigma_1 \rightarrow \texttt{err}$. This means that $\sigma_1 = (pc, S_1, R_1)$ and the type-level abstract interpreter cannot make a transition from (S_1, R_1) on instruction $instr(pc)$. By hypotheses $\sigma_1 <: \sigma_2$ and $\sigma_2 \neq \texttt{err}$, we have $\sigma_2 = (pc, S_2, R_2)$ with $S_1 <: S_2$ and $R_1 <: R_2$. The monotonicity property of the abstract interpreter guarantees that it cannot make a transition from (S_2, R_2) on $instr(pc)$. Thus, $\sigma_2 \rightarrow \texttt{err}$, which is a contradiction.

Therefore, $\sigma_1' \neq \texttt{err}$, and by definition of the $\rightarrow$ and $<:$ relations, this implies $\sigma_1 \neq \texttt{err}$ and $\sigma_2 \neq \texttt{err}$. We can therefore write

$$\sigma_1 = (p, S_1, R_1) \qquad \sigma_1' = (p', S_1', R_1') \qquad \sigma_2 = (p, S_2, R_2)$$

and the following properties hold:

$$S_1 <: S_2 \qquad R_1 <: R_2 \qquad instr(p) : (S_1, R_1) \rightarrow (S_1', R_1')$$

By hypothesis $\sigma_2 \not\rightarrow \texttt{err}$, the abstract interpreter can make a transition on $instr(p)$ from state (S_2, R_2). Let (S_2', R_2') be the result of this transition. The monotonicity and determinacy properties of the type-level abstract interpreter guarantee that $S_1' <: S_2'$ and $R_1' <: R_2'$. Define $\sigma_2' = (p', S_2', R_2')$. We thus have $\sigma_1' <: \sigma_2'$.

To conclude the proof, we need to show that $\sigma_2 \rightarrow \sigma_2'$. Since $instr(p) : (S_2, R_2) \rightarrow (S_2', R_2')$, it suffices to show that p' is a valid successor of the instruction at p in state (S_2, R_2). By hypothesis $\sigma_1 \rightarrow \sigma_1'$, we know that p' is a valid successor of the instruction at p in state (S_1, R_1). If the instruction at p is not a `ret` instruction, its successors are independent of the stack and register types "before" the instruction; thus, $\sigma_2 \rightarrow \sigma_2'$. If the instruction at p is `ret` r, we have $R_1(r) = \texttt{retaddr}(p')$. Since $\sigma_2 \not\rightarrow \texttt{err}$, it must be the case that $R_2(r)$ is a `retaddr` type. Moreover, $R_1 <: R_2$. This entails $R_2(r) = R_1(r) = \texttt{retaddr}(p')$. Hence, p' is the successor of the instruction at p in state (S_2, R_2). The expected result $\sigma_2 \rightarrow \sigma_2'$ follows. □

In the following, we write $\Sigma \sqsubseteq \Sigma'$ to mean that for all $\sigma \in \Sigma$, there exists $\sigma' \in \Sigma'$ such that $\sigma <: \sigma'$.

LEMMA 2. *Let Σ and Σ' be two sets of states such that $\Sigma \sqsubseteq \Sigma'$ and $\texttt{err} \notin C(\Sigma')$. Then, $C(\Sigma) \sqsubseteq W(C(\Sigma'))$.*

Proof. The $\sqsubseteq$ relation is transitive, and by hypothesis on the widening function W, we have $\Sigma \sqsubseteq W(\Sigma)$ for all Σ. It therefore suffices to show that $C(\Sigma) \sqsubseteq C(\Sigma')$.

Choose σ in $C(\Sigma)$. By definition of C, one of the following two cases holds:

- $\sigma \in \Sigma$. Since $\Sigma \sqsubseteq \Sigma'$, there exists $\sigma' \in \Sigma'$ such that $\sigma <: \sigma'$. Since $\Sigma' \subseteq C(\Sigma')$, it follows that σ is a subtype of an element of $C(\Sigma')$.
- $\rho \to \sigma$ for some $\rho \in \Sigma$. Let ρ' be an element of Σ' such that $\rho <: \rho'$. By hypothesis $\texttt{err} \notin C(\Sigma')$, we have $\rho' \neq \texttt{err}$ and $\rho' \not\to \texttt{err}$. Applying Lemma 1, we obtain σ' such that $\rho' \to \sigma'$ and $\sigma <: \sigma'$. Moreover, $\sigma' \in C(\Sigma')$ since σ' is obtained by taking one transition from an element of Σ'.

Since the result above holds for all $\sigma \in C(\Sigma)$, the expected result follows. □

THEOREM 1. *Let Σ_c be the smallest set closed under C containing σ_0, and Σ_w be the smallest set closed under $W \circ C$ containing σ_0. Then,* $\texttt{err} \notin \Sigma_w$ *implies* $\texttt{err} \notin \Sigma_c$. *In other words, if the bytecode passes a widened model-checking algorithm, it also passes the BC model-checking algorithm.*

Proof. Write $\Sigma_c = \bigcup_{n\in\mathbb{N}} \Sigma_c^n$ and $\Sigma_w = \bigcup_{n\in\mathbb{N}} \Sigma_w^n$, where $\Sigma_c^0 = \Sigma_w^0 = \{\sigma_0\}$ and $\Sigma_c^{n+1} = C(\Sigma_c^n)$ and $\Sigma_c^{n+1} = W(C(\Sigma_c^n))$.

We now show by induction on n that $\Sigma_c^n \sqsubseteq \Sigma_w^n$. The base case $n = 0$ is trivial. The inductive case follows from Lemma 2, noticing that $\texttt{err} \notin \Sigma_w$ implies $\texttt{err} \notin C(\Sigma_w^n)$ for all n.

It follows that $\Sigma_c \sqsubseteq \Sigma_w$. Assume, by way of contradiction, that $\texttt{err} \in \Sigma_c$. Thus, there exists $\sigma \in \Sigma_w$ such that $\texttt{err} <: \sigma$. By definition of the $<:$ relation between states, this implies $\sigma = \texttt{err}$ and contradicts the hypothesis $\texttt{err} \notin \Sigma_w$. Hence $\texttt{err} \notin \Sigma_c$ as claimed. □

Appendix B. Relative Completeness of the CFHS Algorithm

In this appendix, we prove that the CFHS verification algorithm (defined in Section 6.3 as an instance of widened model checking) is complete with respect to the BC model-checking algorithm: All programs accepted by the latter algorithm are also accepted by the former. Combined with Theorem 1, this result shows that the CFHS algorithm accepts exactly the same programs as the BC algorithm.

We recall the definitions that describe the CFHS algorithm: the equivalence relation $\approx$ is defined by $\texttt{err} \approx \texttt{err}$ and $(p_1, S_1, R_1) \approx (p_2, S_2, R_2)$ iff $p_1 = p_2$ and S_1, S_2 agree on return addresses and R_1, R_2 agree on return addresses. The merging function M is defined by $M(\{\texttt{err}\}) = \texttt{err}$ and $M(\{(p, S_1, R_1), \ldots, (p, S_n, R_n)\}) = (p, \mathit{lub}(S_1, \ldots, S_n), \mathit{lub}(R_1, \ldots, R_n))$. Finally, the widening function W is defined by $W(\Sigma) = \{M(\Sigma_1), \ldots, M(\Sigma_n)\}$, where $\Sigma_1, \ldots, \Sigma_n$ is a partition of Σ into equivalence classes for $\approx$.

The following key lemma shows that the type-level abstract machine performs parallel transitions from a set of equivalent states and from the single state obtained by merging the equivalent states. In particular, if `err` is not reachable from the equivalent states, it is not reachable from the merged state either. We write $T(\Sigma)$ for the states reachable by one transition from a state in Σ, that is, $T(\Sigma) = \{\sigma' \mid \exists \sigma \in \Sigma.\, \sigma \to \sigma'\}$.

LEMMA 3. *Let Σ be a nonempty set of equivalent states such that* $\texttt{err} \notin \Sigma$ *and* $\texttt{err} \notin T(\Sigma)$. *We then have $W(T(\Sigma)) = T(W(\Sigma))$.*

Proof. Write $\Sigma = \{(p, S_1, R_1), \ldots, (p, S_n, R_n)\}$, where the S_i agree on return addresses, and the R_i also agree on return addresses. Since $\texttt{err} \notin T(\Sigma)$, for every i, there exists S'_i and R'_i such that $\mathit{instr}(p) : (S_i, R_i) \to (S'_i, R'_i)$. According to the Continuity Property 4, we thus have $\mathit{instr}(p) : (\mathit{lub}(S_1 \ldots S_n), \mathit{lub}(R_1 \ldots R_n)) \to (\mathit{lub}(S'_1 \ldots S'_n), \mathit{lub}(R'_1 \ldots R'_n))$. Moreover, by the Determinacy Property 2, there are no other transitions from these initial states.

Consider the successors of the instruction at p. If $\texttt{instr}(p)$ is not a `ret` instruction, these successors $p_1, \ldots, p_k$ depend only on $\texttt{instr}(p)$ and are independent of the stack and register types. If $\texttt{instr}(p)$ is a $\texttt{ret}\ r$ instruction, the successor for the instruction is determined by the `retaddr` type found in $R(r)$, where R is the register type before the instruction. However, the register types "before" considered in this proof, namely, $R_1, \ldots, R_n$ and $\mathit{lub}(R_1, \ldots, R_n)$, all agree on return addresses. That is, they all assign the same `retaddr` type to register r. Thus, the successor for the $\texttt{ret}\ r$ instruction at p is the same in all these register types, just as in the case of a non-`ret` instruction at p.

It follows from this discussion of successors that the set $T(\Sigma)$ of states reachable by one transition from Σ is exactly the Cartesian product of the set of possible successors $p_1, \ldots, p_k$ with the set of stack and register types "after" $(S'_1, R'_1), \ldots, (S'_n, R'_n)$, and similarly for $T(W(\Sigma))$:

$$T(\Sigma) = \{(p_i, S'_j, R'_j) \mid i \in \{1, \ldots, k\},\ j \in \{1, \ldots, n\}\}$$
$$T(W(\Sigma)) = \{(p_i, \mathit{lub}(S'_1 \ldots S'_n), \mathit{lub}(R'_1 \ldots R'_n)) \mid i \in \{1, \ldots, k\}\}$$

Finally, we note that the stack and register types "after" $S'_1, \ldots, S'_n$ and $R'_1, \ldots, R'_n$ agree on return addresses: if $\mathit{instr}(p)$ is not a `jsr` instruction, any `retaddr` type in the state after comes from a matching `retaddr` type in the state before; if $\mathit{instr}(p)$ is a `jsr` instruction, the state after contains an additional $\texttt{retaddr}(p + 3)$ type at the top of the stack, but this type is the same in all stack types after. Thus,

$$W(T(\Sigma)) = \{(p_i, \mathit{lub}(S'_1 \ldots S'_n), \mathit{lub}(R'_1 \ldots R'_n)) \mid i \in \{1, \ldots, k\}\}.$$

It follows that $W(T(\Sigma)) = T(W(\Sigma))$ as expected. □

We now prove a number of algebraic properties of the M and W functions.

LEMMA 4. *Let Σ be a nonempty set of equivalent states. Then, $M(\Sigma)$ is equivalent to every $\sigma \in \Sigma$.*

Proof. We first show that if two types τ_1 and τ_2 agree on return addresses, then $lub(\tau_1, \tau_2)$ agrees on return addresses with both τ_1 and τ_2. Indeed, either $\tau_1 = \tau_2$, in which case $lub(\tau_1, \tau_2) = \tau_1 = \tau_2$ and the result is obvious; or neither τ_1 nor τ_2 are `retaddr` types, in which case $lub(\tau_1, \tau_2)$ is not a `retaddr` type either, thus agrees with τ_1 and with τ_2.

The previous result extends trivially to the least upper bound of one or several stack types or one or several register types.

Let Σ be a nonempty set of equivalent states. Either $\Sigma = \{\texttt{err}\}$, or $\Sigma = \{(p, S_1, R_1), \ldots, (p, S_n, R_n)\}$. In the former case, $M(\Sigma) = \texttt{err}$ and the expected result obviously holds. In the latter case, $M(\Sigma) = (p, lub(S_1, \ldots, S_n), lub(R_1, \ldots, R_n))$. For every i, $lub(S_1, \ldots, S_n)$ and S_i agree on return addresses, and $lub(R_1, \ldots, R_n)$ and R_i agree on return addresses. Thus, $M(\Sigma) \approx \sigma$ for any $\sigma \in \Sigma$. □

LEMMA 5. *Let $\Sigma_1, \ldots, \Sigma_n$ be nonempty sets of states such that all states in $\Sigma_1 \cup \cdots \cup \Sigma_n$ are equivalent. Then, $M(\Sigma_1 \cup \cdots \cup \Sigma_n) = M(\{M(\Sigma_1), \ldots, M(\Sigma_n)\})$.*

Proof. There are two cases to consider. In the first case, the Σ_i are all $\{\texttt{err}\}$. In this case, the expected equality trivially holds. In the second case, `err` does not belong to any Σ_i. We write

$$\Sigma_i = \{(p, S_{i,1}, R_{i,1}), \ldots, (p, S_{i,n_i}, R_{i,n_i})\}.$$

We thus have

$$M(\Sigma_i) = (p, lub_j(S_{i,j}), lub_j(R_{i,j}))$$
$$M(\Sigma_1 \cup \cdots \cup \Sigma_n) = (p, lub_{i,j}(S_{i,j}), lub_{i,j}(R_{i,j})).$$

By hypothesis, all elements of Σ_i are equivalent to all elements of Σ_j. By Lemma 4, $M(\Sigma_i)$ is equivalent to all elements of Σ_i, and $M(\Sigma_j)$ is equivalent to all elements of Σ_j. By transitivity, it follows that $M(\Sigma_i) \approx M(\Sigma_j)$. This holds for all i, j. We thus have

$$M(\{M(\Sigma_1), \ldots, M(\Sigma_n)\}) = (p, lub_i(lub_j(S_{i,j})), lub_i(lub_j(R_{i,j}))).$$

The expected result follows from the associativity and commutativity of the *lub* operation. □

LEMMA 6. *$W(\Sigma_1 \cup \cdots \cup \Sigma_n) = W(W(\Sigma_1) \cup \cdots \cup W(\Sigma_n))$ for all sets of states $\Sigma_1, \ldots, \Sigma_n$.*

Proof. Let $A_1, \ldots, A_k$ be a partition of $\Sigma_1 \cup \cdots \cup \Sigma_n$ into equivalence classes. Thus, for every i, $\{A_j \cap \Sigma_i \mid j \in \{1 \ldots k\}, A_j \cap \Sigma_i \neq \emptyset\}$ is a partition of Σ_i into equivalence classes. We thus have

$$W(\Sigma_1) \cup \cdots \cup W(\Sigma_n)$$
$$= \{M(A_j \cap \Sigma_i) \mid j \in \{1 \ldots k\}, i \in \{1 \ldots n\}, A_j \cap \Sigma_i \neq \emptyset\}.$$

Consider two elements of this set, $M(A_j \cap \Sigma_i)$ and $M(A_{j'} \cap \Sigma_{i'})$. By Lemma 4, the former is equivalent to every element of $A_j \cap \Sigma_i$, and the latter to every element of $A_{j'} \cap \Sigma_{i'}$. Moreover, the $A_1, \ldots, A_k$ are disjoint equivalence classes. Hence, $M(A_j \cap \Sigma_i) \approx M(A_{j'} \cap \Sigma_{i'})$ if and only if $j = j'$. Therefore, the equivalence classes of $W(\Sigma_1) \cup \cdots \cup W(\Sigma_n)$ for the $\approx$ relation are the sets

$$B_j = \{M(A_j \cap \Sigma_i) \mid i \in \{1 \ldots n\}, A_j \cap \Sigma_i \neq \emptyset\}$$

for $j = 1, \ldots, k$. By Lemma 5,

$$M(B_j) = M\Big(\bigcup\{A_j \cap \Sigma_i \mid i \in \{1 \ldots n\}, A_j \cap \Sigma_i \neq \emptyset\}\Big) = M(A_j).$$

It follows that

$$\begin{aligned} W(W(\Sigma_1) \cup \cdots \cup W(\Sigma_n)) &= \{M(B_1), \ldots, M(B_k)\} \\ &= \{M(A_1), \ldots, M(A_k)\} \\ &= W(\Sigma_1 \cup \cdots \cup \Sigma_n). \end{aligned}$$

This is the expected result. □

LEMMA 7. *Let Σ be a set of states such that* `err` $\notin C(\Sigma)$. *Then,* $W(C(\Sigma)) = W(C(W(\Sigma)))$.

Proof. Let $\Sigma_1, \ldots, \Sigma_n$ be a partition of Σ into equivalence classes. By definition of C, we have

$$C(\Sigma) = \Sigma_1 \cup \cdots \cup \Sigma_n \cup T(\Sigma_1) \cup \cdots \cup T(\Sigma_n).$$

Applying Lemma 6, we obtain

$$W(C(\Sigma)) = W(W(\Sigma_1) \cup \cdots \cup W(\Sigma_n) \cup W(T(\Sigma_1)) \cup \cdots \cup W(T(\Sigma_n))).$$

It follows from the hypothesis `err` $\notin C(\Sigma)$ that `err` $\notin \Sigma_i$ and `err` $\notin T(\Sigma_i)$ for all $i = 1, \ldots, n$. By Lemma 3, we thus have $W(T(\Sigma_i)) = T(W(\Sigma_i))$. It follows that

$$\begin{aligned} W(C(\Sigma)) &= W(W(\Sigma_1) \cup \cdots \cup W(\Sigma_n) \cup T(W(\Sigma_1)) \cup \cdots \cup T(W(\Sigma_n))) \\ &= W(C(W(\Sigma_1) \cup \cdots W(\Sigma_n))). \end{aligned}$$

Since $\Sigma_1, \ldots, \Sigma_n$ is a partition of Σ into equivalence classes, we have $W(\Sigma_i) = \{M(\Sigma_i)\}$ and $W(\Sigma) = \{M(\Sigma_1), \ldots, M(\Sigma_n)\} = W(\Sigma_1) \cup \cdots \cup W(\Sigma_n)$. The expected result follows. □

THEOREM 2. *Let Σ_c be the smallest set closed under C containing σ_0, and let Σ_w be the smallest set closed under $W \circ C$ containing σ_0. Then,* `err` $\notin \Sigma_c$ *implies* `err` $\notin \Sigma_w$. *In other words, if the bytecode passes the BC algorithm, it also passes the CFHS algorithm.*

Proof. Write $\Sigma_c = \bigcup_{n\in\mathbb{N}} \Sigma_c^n$ and $\Sigma_w = \bigcup_{n\in\mathbb{N}} \Sigma_w^n$, where $\Sigma_c^0 = \Sigma_w^0 = \{\sigma_0\}$ and $\Sigma_c^{n+1} = C(\Sigma_c^n)$ and $\Sigma_c^{n+1} = W(C(\Sigma_c^n))$. Assume that $\texttt{err} \notin \Sigma_c$. This implies $\texttt{err} \notin \Sigma_c^n$ for any n.

We now show by induction on n that $\Sigma_w^n = W(\Sigma_c^n)$. The base case $n = 0$ is trivial, since W is the identity function on singleton sets. For the inductive case, assume $\Sigma_w^n = W(\Sigma_c^n)$. Then, $\Sigma_w^{n+1} = W(C(\Sigma_w^n)) = W(C(W(\Sigma_c^n)))$. Note that $\texttt{err} \notin C(\Sigma_c^n) = \Sigma_c^{n+1}$. Applying Lemma 7, we have that $\Sigma_w^{n+1} = W(C(\Sigma_c^n)) = W(\Sigma_c^{n+1})$, as desired.

By construction of the widening function W, for any set of states Σ we have $\texttt{err} \in W(\Sigma)$ if and only if $\texttt{err} \in \Sigma$. Thus, for any n, we have $\texttt{err} \notin \Sigma_w^n$, since $\Sigma_w^n = W(\Sigma_c^n)$ and $\texttt{err} \notin \Sigma_c^n$. It follows that $\texttt{err} \notin \Sigma_w$. □

References

1. Abadi, M., Banerjee, A., Heintze, N. and Riecke, J. G.: A core calculus of dependency, in *26th Symposium on Principles of Programming Languages*, 1999, pp. 147–160.
2. Basin, D., Friedrich, S. and Gawkowski, M.: Bytecode verification by model checking, *J. Automated Reasoning*. Special issue on bytecode verification (this issue).
3. Bertot, Y.: Formalizing a JVML verifier for initialization in a theorem prover, in *Proc. Computer Aided Verification (CAV'01)*, Lecture Notes in Comput. Sci. 2102, 2001, pp. 14–24.
4. Brisset, P.: Vers un vérifieur de bytecode Java certifié, Seminar given at Ecole Normale Supérieure, Paris, October 2, 1998.
5. Brunnstein, K.: Hostile ActiveX Control demonstrated, *RISKS Forum* **18**(82) (1998).
6. Chen, Z.: *Java Card Technology for Smart Cards: Architecture and Programmer's Guide*, The Java Series, Addison-Wesley, 2000.
7. Chugunov, G., Åke Fredlund, L. and Gurov, D.: Model checking multi-applet Java Card applications, in *Smart Card Research and Advanced Applications Conference (CARDIS'02)*, 2002.
8. Coglio, A.: Simple verification technique for complex Java bytecode subroutines, in *4th ECOOP Workshop on Formal Techniques for Java-like Programs*, 2002. Extended version available as Kestrel Institute technical report.
9. Coglio, A.: Improving the official specification of Java bytecode verification, *Concurrency and Computation: Practice and Experience* **15**(2) (2003), 155–179.
10. Coglio, A., Goldberg, A. and Qian, Z.: Towards a provably-correct implementation of the JVM bytecode verifier, in *OOPSLA Workshop on Formal Underpinnings of Java*, 1998.
11. Cohen, R.: The defensive Java virtual machine specification, Technical Report, Computational Logic Inc., 1997.
12. Deville, D. and Grimaud, G.: Building an "impossible" verifier on a Java Card, in *USENIX Workshop on Industrial Experiences with Systems Software (WIESS'02)*, 2002.
13. Freund, S. N. and Mitchell, J. C.: A type system for the Java bytecode language and verifier, *J. Automated Reasoning*. Special issue on bytecode verification (this issue).
14. Freund, S. N. and Mitchell, J. C.: A formal framework for the Java bytecode language and verifier, in *Object-Oriented Programming Systems, Languages and Applications 1999*, pp. 147–166.
15. Freund, S. N. and Mitchell, J. C.: A type system for object initialization in the Java bytecode language, *ACM Transactions on Programming Languages and Systems* **21**(6) (1999), 1196–1250.

16. Frey, A.: On-terminal verifier for JEFF files, Personal communication, 2001.
17. Goldberg, A.: A specification of Java loading and bytecode verification, in *ACM Conference on Computer and Communications Security*, 1998, pp. 49–58.
18. Gong, L.: *Inside Java 2 Platform Security: Architecture, API Design, and Implementation*, The Java Series, Addison-Wesley, 1998.
19. Gosling, J. A.: Java intermediate bytecodes, in *Proc. ACM SIGPLAN Workshop on Intermediate Representations*, 1995, pp. 111–118.
20. Hagiya, M. and Tozawa, A.: On a new method for dataflow analysis of Java virtual machine subroutines, in G. Levi (ed.), *SAS'98*, Lecture Notes in Comput. Sci. 1503, 1998, pp. 17–32.
21. Hartel, P. H. and Moreau, L. A. V.: Formalizing the safety of Java, the Java virtual machine and Java Card, *ACM Computing Surveys* **33**(4) (2001), 517–558.
22. Heintze, N. and Riecke, J. G.: The SLam calculus: Programming with secrecy and integrity, in *25th Symposium Principles of Programming Languages*, 1998, pp. 365–377.
23. Henrio, L. and Serpette, B.: A framework for bytecode verifiers: Application to intra-procedural continuations, Research Report, INRIA, 2001.
24. Huisman, M., Jacobs, B. and van den Berg, J.: A case study in class library verification: Java's Vector class, *Software Tools for Technology Transfer* **3**(3) (2001), 332–352.
25. Jensen, T., Le Métayer, D. and Thorn, T.: Verification of control flow based security properties, in *IEEE Symposium on Security and Privacy*, 1999.
26. Klein, G.: Verified Java bytecode verification, Ph.D. thesis, Technische Universität München, 2003.
27. Klein, G. and Nipkow, T.: Verified lightweight bytecode verification, *Concurrency and Computation: Practice and Experience* **13** (2001), 1133–1151.
28. Klein, G. and Nipkow, T.: Verified bytecode verifiers, *Theoret. Comput. Sci.* (2002). To appear.
29. Klein, G. and Wildmoser, M.: Verified bytecode subroutines, *J. Automated Reasoning*. Special issue on bytecode verification (this issue).
30. Knoblock, T. and Rehof, J.: Type elaboration and subtype completion for Java bytecode, in *27th Symposium Principles of Programming Languages*, 2000, pp. 228–242.
31. Leroy, X.: Bytecode verification for Java smart card, *Software Practice & Experience* **32** (2002), 319–340.
32. Leroy, X. and Rouaix, F.: Security properties of typed applets, in J. Vitek and C. Jensen (eds.), *Secure Internet Programming – Security Issues for Mobile and Distributed Objects*, Lecture Notes in Comput. Sci. 1603, Springer-Verlag, 1999, pp. 147–182.
33. Lindholm, T. and Yellin, F.: *The Java Virtual Machine Specification*, 2nd edn, The Java Series, Addison-Wesley, 1999.
34. McGraw, G. and Felten, E.: *Securing Java*, Wiley, 1999.
35. Muchnick, S. S.: *Advanced Compiler Design and Implementation*, Morgan Kaufmann, 1997.
36. Necula, G. C.: Proof-carrying code, in *24th Symposium Principles of Programming Languages*, 1997, pp. 106–119.
37. Nielson, F., Nielson, H. R. and Hankin, C.: *Principles of Program Analysis*, Springer-Verlag, 1999.
38. Nipkow, T.: Verified bytecode verifiers, in *Foundations of Software Science and Computation Structures (FOSSACS'01)*, Lecture Notes in Comput. Sci. 2030, 2001, pp. 347–363.
39. O'Callahan, R.: A simple, comprehensive type system for Java bytecode subroutines, in *26th Symposium Principles of Programming Languages*, 1999, pp. 70–78.
40. Posegga, J. and Vogt, H.: Java bytecode verification using model checking, in *Workshop Fundamental Underpinnings of Java*, 1998.
41. Pottier, F. and Simonet, V.: Information flow inference for ML, in *29th Symposium Principles of Programming Languages*, 2002, pp. 319–330.

42. Pottier, F., Skalka, C. and Smith, S.: A systematic approach to static access control, in D. Sands (ed.), *Proceedings of the 10th European Symposium on Programming (ESOP'01)*, Lecture Notes in Comput. Sci. 2028, 2001, pp. 30–45.
43. Pusch, C.: Proving the soundness of a Java bytecode verifier specification in Isabelle/HOL, in W. R. Cleaveland (ed.), *TACAS'99*, Lecture Notes in Comput. Sci. 1579, 1999, pp. 89–103.
44. Qian, Z.: A formal specification of Java virtual machine instructions for objects, methods and subroutines, in J. Alves-Foss (ed.), *Formal Syntax and Semantics of Java*, Lecture Notes in Comput. Sci. 1523, Springer-Verlag, 1998.
45. Qian, Z.: Standard fixpoint iteration for Java bytecode verification, *ACM Transactions on Programming Languages and Systems* **22**(4) (2000), 638–672.
46. Rose, E.: Vérification de code d'octet de la machine virtuelle Java: formalisation et implantation, Ph.D. thesis, University Paris 7, 2002.
47. Rose, E. and Rose, K.: Lightweight bytecode verification, in *OOPSLA Workshop on Formal Underpinnings of Java*, 1998.
48. Schmidt, D. A.: Data flow analysis is model checking of abstract interpretations, in *25th Symposium Principles of Programming Languages*, 1998, pp. 38–48.
49. Stärk, R., Schmid, J. and Börger, E.: *Java and the Java Virtual Machine*, Springer-Verlag, 2001.
50. Stärk, R. F. and Schmid, J.: Completeness of a bytecode verifier and a certifying Java-to-JVM compiler, *J. Automated Reasoning*. Special issue on bytecode verification (this issue).
51. Stata, R. and Abadi, M.: A type system for Java bytecode subroutines, *ACM Transactions on Programming Languages and Systems* **21**(1) (1999), 90–137.
52. Sun Microsystems: Java 2 platform micro edition technology for creating mobile devices, White paper, `http://java.sun.com/products/cldc/wp/KVMwp.pdf`, 2000.
53. Trusted Logic: Off-card bytecode verifier for Java card, 2001. Distributed as part of Sun's Java Card Development Kit.
54. Vigna, G. (ed.): *Mobile Agents and Security*, Lecture Notes in Comput. Sci. 1419, Springer-Verlag, 1998.
55. Volpano, D. and Smith, G.: A type-based approach to program security, in *Proceedings of TAPSOFT'97, Colloquium on Formal Approaches in Software Engineering*, Lecture Notes in Comput. Sci. 1214, 1997, pp. 607–621.
56. Volpano, D., Smith, G. and Irvine, C.: A sound type system for secure flow analysis, *J. Computer Security* **4**(3) (1996), 1–21.
57. Walker, D.: A type system for expressive security policies, in *27th Symposium Principles of Programming Languages*, 2000, pp. 254–267.
58. Yellin, F.: Low level security in Java, in *Proceedings of the Fourth International World Wide Web Conference*, 1995, pp. 369–379.