Simple Verification Technique for Complex Java Bytecode Subroutines

Alessandro Coglio

Kestrel Institute
3260 Hillview Avenue, Palo Alto, CA 94304, USA
Ph. +1-650-493-6871 Fax +1-650-424-1807
http://www.kestrel.edu/~coglio
coglio@kestrel.edu

Abstract. Java is normally compiled to bytecode, which is verified and then executed by the Java Virtual Machine. Bytecode produced via compilation must pass verification. The main cause of complexity for bytecode verification is subroutines, used by compilers to generate more compact code. The techniques to verify subroutines proposed in the literature reject certain programs produced by mundane compilers or are otherwise difficult to realize within an implementation of the Java Virtual Machine. This paper presents a novel technique which is very simple to understand, implement, and prove sound. It is also very powerful: the set of accepted programs has a simple characterization which most likely includes all code generable by current compilers and which enables future compilers to make more extensive use of subroutines.

1 Java Bytecode

Java [2, 11] is normally compiled to a platform-independent bytecode language, which is executed by the Java Virtual Machine (JVM) [18]. This bytecode language features intra-method subroutines, used by Java compilers to generate more compact code [18, Sect. 7.13].

In an idealized version of Java bytecode, similar to those in [8, 13, 20, 25], a program P is a list of instructions. The positions in the list (starting from 0) are the addresses of P. Instructions operate on values stored in a finite collection of named variables and in a stack of bounded size. Values are integers, floats, and (some) addresses; values carry explicit type tags (e.g., 0_{I} is the integer zero, 0_{F} the float zero, and $0_{\mathrm{I}} \neq 0_{\mathrm{F}}$).

For instance, the div instruction pops two integers from the stack and pushes back their quotient, as formalized by the following rule:

$$\frac{P_i = \mathsf{div}}{\langle i, vr, sk \cdot \iota_{\mathrm{I}} \cdot \iota'_{\mathrm{I}} \rangle \rightsquigarrow \langle i+1, vr, sk \cdot div(\iota', \iota)_{\mathrm{F}} \rangle} \quad (\textsc{dv})$$

The triple $\langle i, vr, sk \cdot \iota_{\mathrm{I}} \cdot \iota'_{\mathrm{I}} \rangle$ is an execution state where: i is the program counter, i.e., the address of the instruction about to the executed; variable x contains the value $vr(x)$, i.e., vr maps variable names to values; and the stack consists of two integers ι'_{I} and ι_{I} on top of the (possibly empty) sub-stack sk. The rule says that if the current instruction is div, then the program counter is advanced to $i+1$, the two integers are popped, and the float $div(\iota', \iota)_{\mathrm{F}}$ is pushed (which, for simplicity, is defined even if the divisor $\iota = 0$).

Subroutines are realized by the jsr and ret instructions:

$$\frac{P_i = \mathsf{jsr}\ s \qquad |sk| < max}{\langle i, vr, sk\rangle \rightsquigarrow \langle s, vr, sk \cdot i_{\mathrm{C}}\rangle}\ (\textsc{js}) \qquad \frac{P_i = \mathsf{ret}\ x \qquad vr(x) = c_{\mathrm{C}}}{\langle i, vr, sk\rangle \rightsquigarrow \langle c+1, vr, sk\rangle}\ (\textsc{rt})$$

The former pushes the calling address and jumps to the subroutine address, provided there is room in the stack (whose size limit is max). The latter jumps to the successor of the calling address stored in the variable. (There are instructions to move values between variables and stack.)

There is no explicit notion of subroutine as a textually delimited piece of code: jsr and ret may be scattered here and there in P. While compilers usually produce code where the address range of each subroutine is clearly identifiable, certain Java programs result in bytecode where subroutines can be exited implicitly via branching or exceptions (see Sect. 5 for an example), making the determination of their boundaries more difficult.

The above rules include type safety checks ensuring that the values operated upon have the required types and that no stack overflow or underflow ever occurs. If these checks fail, an explicit error state is reached:

$$\frac{P_i \neq \mathsf{halt} \qquad \nexists\, i', vr', sk'.\ \langle i, vr, sk\rangle \rightsquigarrow \langle i', vr', sk'\rangle}{\langle i, vr, sk\rangle \rightsquigarrow \mathsf{Err}}\ (\textsc{er})$$

The first condition requires the current instruction to be not halt, because halt causes graceful program termination. The second condition requires that no normal state be reachable via the other rules, which happens when the type safety checks of the rule for the current instruction are violated.

Typical implementations of the JVM do not perform these type safety checks for performance reasons. The outcome of operating on values with the wrong types is undefined [18, Sect. 6.1]; Err corresponds to the undefined state that a JVM implementation would move into. So, the notion of type safety is:

$$TypeSafe(P) \Leftrightarrow \mathsf{Init} \not\rightsquigarrow^{+} \mathsf{Err},$$

where $\rightsquigarrow^{+}$ is the transitive closure of $\rightsquigarrow$ and $\mathsf{Init} = \langle 0, \lambda x.0_{\mathrm{I}}, [\,]\rangle$ is the initial state, characterized by program counter 0, 0_{I} in each variable, and empty stack.

2 Verification

For security reasons [31, 10] the JVM verifies incoming code prior to executing it. The goal of this verification is to statically establish that certain type safety checks (e.g., that there are two integers at the top of the stack when a div is reached) would always succeed if they were performed at run time; so, the execution engine can safely omit them. Since compilers check equivalent properties on Java source, code produced via compilation must pass verification.

The last requirement is not so easy to assess because there is currently no precise characterization of the output of Java compilers, which is furthermore susceptible to change as compilers evolve. As advocated in [3], a solution is to use a precise characterization of a set of bytecode programs as a "contract" between compiler developers and JVM developers: the former shall write compilers whose generable programs belong to the set, and the latter shall write bytecode verifiers that accept any program belonging to the set.

3 Complexity of Subroutines

Subroutines are the main cause of complexity for bytecode verification. Without them, the simple technique described in [18, Sect. 4.9.2] works beautifully well. But with them, in order to accept code produced by mundane compilers, a more elaborate analysis is needed.

As explained in [18, Sect. 4.9.6], a variable x may have different types, say int and flt, at different calling addresses c and c' of the same subroutine. So, inside the subroutine x has type any, the result of merging int and flt. If x is not modified inside the subroutine, x should have type int at $c+1$ and flt at $c'+1$. But a simple-minded data flow analysis would blindly propagate any to $c+1$ and $c'+1$, thus causing mundane programs to be rejected.

Sun's technique to verify subroutines, informally described in [18, Sect. 4.9.6] and implemented in [26], keeps track, during the data flow analysis, of the variables modified inside subroutines. If x is not modified, its type at $c+1$ is propagated from c and not from the ret.

A thorough study of Sun's approach, along with its problems and ways to fix some of them, is presented in [5]. The bottom line is that while it works for simpler programs, it rejects less simple programs that are nonetheless produced by mundane compilers (see Sect. 5 for an example).

4 The Technique

4.1 Definition

The new technique is a data flow analysis [19] based on two key ideas: (1) instead of merging two different types into any, keep both; and (2) use calling addresses as types for themselves.

Besides int and flt, there is a type ca_c for each calling address c of P; there is no type any. The lattice $\langle L, \sqsubseteq, \sqcap, \sqcup \rangle$ of this data flow analysis is the result of adjoining a top element fail to the lattice $\langle \mathcal{P}(\mathit{VSType}), \subseteq, \cap, \cup \rangle$, where $\mathcal{P}$ is the powerset operator and VSType is the set of all pairs $\langle vt, st \rangle$ where vt is a map assigning a type to each variable name and st is a list of types for the stack (of length at most max). So, during the analysis every address of P is labeled by a set of such pairs or by fail. The join operation $\sqcup$ is used to merge lattice elements from converging control paths in P: the result of merging two sets of pairs is simply their union, while merging fail with anything yields fail.

For each instruction (except halt), there is a transfer function $\mathit{tf} : L \rightarrow L$. For instance, the one for div is:

$$\mathit{tf}_{\mathsf{div}} = \mathit{lift}(\lambda\langle vt, st\rangle.\textbf{if } st = st' \cdot \mathsf{int} \cdot \mathsf{int} \textbf{ then } \langle vt, st' \cdot \mathsf{flt}\rangle \textbf{ else } \mathsf{fail}),$$

where the higher-order function $\mathit{lift} : (\mathit{VSType} \rightarrow \mathit{VSType} \cup \{\mathsf{fail}\}) \rightarrow (L \rightarrow L)$ is defined by:

$$\begin{aligned}\mathit{lift}(f)(l) = {} & \textbf{if } (l \neq \mathsf{fail} \wedge (\forall \langle vt, st\rangle \in l.\ f(vt, st) \neq \mathsf{fail})) \\ & \textbf{then } \{f(vt, st) \mid \langle vt, st\rangle \in l\} \textbf{ else } \mathsf{fail}.\end{aligned}$$

The argument of lift in the definition of $\mathit{tf}_{\mathsf{div}}$ is the function that maps a pair $\langle vt, st\rangle$ to the pair $\langle vt, st' \cdot \mathsf{flt}\rangle$ if $st = st' \cdot \mathsf{int} \cdot \mathsf{int}$, to fail otherwise; this corresponds to rule (DV) in Sect. 1. This function is lifted to operate element-wise on a set of pairs: if it yields fail on any pair, the result is fail; otherwise, the result is the set of resulting pairs.

Also the transfer function for jsr operates on a set of pairs element-wise:

$$\mathit{tf}^{c}_{\mathsf{jsr}} = \mathit{lift}(\lambda\langle vt, st\rangle.\textbf{if } |st| < max \textbf{ then } \langle vt, st \cdot \mathsf{ca}_c\rangle \textbf{ else } \mathsf{fail}).$$

This transfer function is parameterized by the calling address c at which the jsr appears. It pushes ca_c onto the (type) stack; see rule (JS).

The only transfer function that does not operate element-wise is the one for ret, which is parameterized by a variable x and by a calling address c:

$$\begin{aligned}\mathit{tf}^{x,c}_{\mathsf{ret}}(l) = {} & \textbf{if } (l \neq \mathsf{fail}\ \wedge\ (\forall \langle vt, st\rangle \in l.\ (\exists c'.\ vt(x) = \mathsf{ca}_{c'}))) \\ & \textbf{then } \{\langle vt, st\rangle \in l \mid vt(x) = \mathsf{ca}_c\} \textbf{ else } \mathsf{fail}.\end{aligned}$$

It filters a set of pairs with respect to c: only the pairs with ca_c assigned to x are kept, while the others are discarded. The parameter c is determined by the address $c+1$ that the result of $\mathit{tf}^{x,c}_{\mathsf{ret}}$ is propagated to. If any pair in the set does not assign a calling address type to x, the transfer function yields fail. See rule (RT).

Since L is finite and all the transfer functions are monotone, the data flow analysis always converges to the least solution σ, which assigns $\sigma_i \in L$ to address i. The notion of verification is:

$$\mathit{Verified}(P) \Leftrightarrow (\forall i.\ \sigma_i \neq \mathsf{fail}).$$

If $\sigma_i \neq \mathsf{fail}$, then at run time whenever i is reached there is a pair $\langle vt, st\rangle \in \sigma_i$ containing the types of the values in the variables and stack.

An example of least solution is shown in Fig. 1. Some instructions have not been previously described: $\mathsf{push0}$ pushes 0_{I} onto the stack; inc increments the integer at the top of the stack; $\mathsf{load}\ x$ pushes the value stored in x; and $\mathsf{store}\ x$

i	P_i	st_i	$vt_i(x)$	$vt_i(y)$
0	push0	[ ]	int	int
1	push0	[int]	int	int
2	div	[int, int]	int	int
3	store x	[flt]	int	int
4	jsr 11	[ ]	flt	int
5	push0	[ ]	flt	ca_4
6	store x	[int]	flt	ca_4
7	jsr 11	[ ]	int	ca_4
8	load x	[ ]	int	ca_7
9	inc	[int]	int	ca_7
10	halt	[int]	int	ca_7
11	store y	[ca_4 | ca_7]	flt | int	int | ca_4
12	ret y	[ ]	flt | int	ca_4 | ca_7

Fig. 1. Example of least solution.

moves the top value of the stack into x. Most addresses are labeled by one pair, whose types are shown under the appropriate columns. Addresses 11 and 12 are labeled by two pairs; for improved readability, their types have been spread across the columns: one pair consists of the types at the left of the "|" symbols, the other pair consists of the types at the right.

The technique is very simple to understand and implement. Unlike others, it does not attempt to determine subroutine boundaries or variables modified inside subroutines. Rather, its "unstructured" nature reflects the possibly unstructured occurrences of jsr and ret in programs. Its treatment of jsr and ret is as simple as their run time behavior, described by rules (JS) and (RT).

4.2 Properties

The data flow analysis explores all paths in P, simulating execution at the type level. If P were not type-safe, fail would appear in the least solution.

Theorem 1 (Soundness). $\mathit{Verified}(P) \;\Rightarrow\; \mathit{TypeSafe}(P)$

Some type-safe programs are unjustly rejected by the technique. The reason is the instruction if0 j, which pops an integer from the stack and then jumps to the target address j if the integer is 0_I (if not, execution continues at the next address). If a program contains a push0 immediately followed by an if0, the instruction following the if0 can never be reached. But the data flow analysis is insensitive to the actual value of the integer when the if0 is encountered: it is just an int. So, if the if0 is followed by instructions performing type-unsafe operations, the program is rejected.

Consider an integer-insensitive operational semantics $\rightsquigarrow_\mathrm{I}$ that extends $\rightsquigarrow$ by allowing the execution of if0 to non-deterministically transfer control to either the target or the next address, regardless of the value of the popped integer. The notion of integer-insensitive type safety, $\mathit{TypeSafe}_{\mathrm{I}}(P)$, is defined as in Sect. 1, with $\rightsquigarrow$ replaced by $\rightsquigarrow_{\mathrm{I}}$.

Theorem 2 (Characterization). $\mathit{Verified}(P) \;\Leftrightarrow\; \mathit{TypeSafe}_{\mathrm{I}}(P)$

The rightward implication is proved analogously to Theorem 1. The leftward implication is proved by constructing an assignment γ of lattice elements to addresses as follows: γ_i is the set of all pairs $\langle vt, st\rangle$ containing the types of some execution state $\langle i, vr, sk\rangle$ reachable from Init (with the integer-insensitive operational semantics). By construction, $\gamma_i \neq \mathsf{fail}$. It is then shown that γ is a solution to the data flow analysis; so, fail cannot appear in the least solution σ (because $\sigma_i \sqsubseteq \gamma_i$ for all i). As a matter of fact, it can be also shown that $\gamma = \sigma$.

Theorem 2 provides a very simple and precise characterization of the programs accepted by the technique, usable as the contract mentioned in Sect. 2. The technique is very powerful, in the sense that it accepts a very large set of programs. To date, I have not found any bytecode program generated by a compiler that does not satisfy this characterization. Since compilers are quite unlikely to expect bytecode verifiers to be integer-sensitive (for the full Java bytecode, the notion of insensitivity must be extended from integers to null references and other features [6]), there are reasons to believe that the characterization includes all code generable by current and future compilers. In addition, it may enable future compilers to make a more elaborate use of subroutines in order to generate more compact code.

While the technique can be refined to accept more type-safe programs (e.g., by refining int into a type for 0_{I} and a type for possibly non-zero integers, and refining the transfer functions for $\mathsf{push0}$, $\mathsf{if0}$, etc. accordingly), such refinements would add complexity and invalidate Theorem 2. In addition, the benefit is dubious: for example, no sensible compiler would ever generate a program with a $\mathsf{push0}$ immediately followed by an $\mathsf{if0}$. These considerations support the (informal) argument that the technique embodies an optimal trade-off between power and simplicity.

4.3 Implementation

The need to carry around sets of pairs during the data flow analysis arises in order to separate them at the ret instruction. But if two singleton sets are merged that do not both contain calling addresses in the same variable or stack element, then the two pairs will never be separated, and so they can be merged into a single pair (re-introducing type any).

In other words, a hybrid merging strategy can be used: if pairs cannot be later separated, they are merged into one pair; they are kept distinct if there are different calling addresses in corresponding positions. So, if a program has no subroutines, all sets are singletons and the analysis essentially reduces to the one in [18, Sect. 4.9.2].

Experimental measures [7, 20] suggest that current compilers generate code with very infrequent use of subroutines. So, in the presence of subroutines, the sets of pairs should be fairly small.

5 Related Work

As a point of comparison with other techniques, consider the Java code in Fig. 2 (adapted from [24, Fig. 16.8]). The variable `y`, which contains an undefined value at the beginning of the method `m`, is definitely assigned a value before it is used by the `return` [11]. Definite assignment is part of type safety and must be checked by bytecode verification in the JVM. The bytecode naturally produced (e.g., with [26]) from Fig. 2 is accepted by the new technique, as shown in Fig. 3, where the real Java bytecode instructions (not abstractions of them) are used [18, Chap. 6], the exception handler for the `try` block [18, Sect. 7.12] is omitted, the variables are denoted by names instead of numbers, and the type udf indicates that a variable contains an undefined value.

As another point of comparison, consider the Java code in Fig. 4 (adapted from [20, Fig. 6]). The `continue` inside the `finally` block, if executed, transfers control to the beginning of the `while` loop [11]. The bytecode naturally produced (e.g., with [26]) from Fig. 4 is accepted by the new technique, as shown in Fig. 5. Note that the subroutine, whose address range is 5–9, can be exited implicitly (i.e., not via a ret) from address 8, thus realizing the semantics of `continue`.

Sun's technique rejects the code in Fig. 3 because the types int and udf for y are merged into udf inside the subroutine (in Sun's technique, udf coincides with any). Since y may be modified at address 15, udf is propagated from address 16 to both 5 and 10, and thus eventually to 17, where iload y causes verification to fail (because it requires an int in y).

Most of the formal techniques proposed in the literature [8, 13, 23–25] also reject the code in Fig. 3 because they inaccurately assign udf to y inside the subroutine and propagate it to the caller's successor, similarly to Sun's technique.

The remaining formal techniques [20, 22, 30] are comparable in power with the new technique, but none of them is as straightforward to realize in a JVM implementation as a data flow analysis. While the declarative rules for type assignments presented in [20] are easy to check, it is not so easy to compute type assignments that satisfy those rules. The use of a model checker [22] or of a Haskell type checker [30], while viable for off-line verification, is problematic to realize within the JVM.

There exist several commercial and academic implementations of the JVM, which include bytecode verifiers, but no documentation is readily available about their treatment of subroutines. Anyhow, [24, Sect. 16.1.1] reports that the code in Fig. 3 is rejected by all the verifiers tried by the authors, including those in [26] and in various versions of Netscape and Internet Explorer, as well as the Kimera verifier [16]. Probably, all these verifiers employ the "official" approach to subroutines described in [18, Sect. 4.9.6].

```
static int m(boolean x) {
  int y;
  try {
    if (x) return 1;
    y = 2;
  } finally {
    if (x) y = 3;
  }
  return y;
}
```

Fig. 2. Type-safe Java code rejected by most techniques and verifiers.

i	P_i	st_i	$vt_i(x)$	$vt_i(y)$	$vt_i(z)$	$vt_i(w)$
0	iload x	[ ]	int	udf	udf	udf
1	ifeq 7	[int]	int	udf	udf	udf
2	iconst_1	[ ]	int	udf	udf	udf
3	store z	[int]	int	udf	udf	udf
4	jsr 11	[ ]	int	udf	int	udf
5	iload z	[ ]	int $\mid$ int	udf $\mid$ int	int $\mid$ int	ca$_4$ $\mid$ ca$_4$
6	ireturn	[int $\mid$ int]	int $\mid$ int	udf $\mid$ int	int $\mid$ int	ca$_4$ $\mid$ ca$_4$
7	iconst_2	[ ]	int	udf	udf	udf
8	istore y	[int]	int	udf	udf	udf
9	jsr 11	[ ]	int	int	udf	udf
10	goto 17	[ ]	int	int	udf	ca$_9$
11	astore w	[ca$_4$ $\mid$ ca$_9$]	int $\mid$ int	udf $\mid$ int	int $\mid$ udf	udf $\mid$ udf
12	iload x	[ ]	int $\mid$ int	udf $\mid$ int	int $\mid$ udf	ca$_4$ $\mid$ ca$_9$
13	ifeq 16	[int $\mid$ int]	int $\mid$ int	udf $\mid$ int	int $\mid$ udf	ca$_4$ $\mid$ ca$_9$
14	iconst_3	[ ]	int $\mid$ int	udf $\mid$ int	int $\mid$ udf	ca$_4$ $\mid$ ca$_9$
15	istore y	[int $\mid$ int]	int $\mid$ int	udf $\mid$ int	int $\mid$ udf	ca$_4$ $\mid$ ca$_9$
16	ret w	[ ]	int $\mid$ int $\mid$ int	udf $\mid$ int $\mid$ int	int $\mid$ int $\mid$ udf	ca$_4$ $\mid$ ca$_4$ $\mid$ ca$_9$
17	iload y	[ ]	int	int	udf	ca$_9$
18	ireturn	[int]	int	int	udf	ca$_9$

Fig. 3. Successful verification of the bytecode for Fig. 2.

```
static void m(boolean x) {
  while (x) {
    try {
      x = false;
    } finally {
      if (x) continue;
    }
  }
}
```

Fig. 4. Type-safe Java code rejected by some techniques and verifiers.

i	P_i	st_i	$vt_i(x)$	$vt_i(y)$
0	goto 10	[ ]	int	udf
1	iconst_0	[ ]	int | int	udf | ca$_3$
2	istore x	[int | int]	int | int	udf | ca$_3$
3	jsr 5	[ ]	int | int	udf | ca$_3$
4	goto 10	[ ]	int	ca$_3$
5	astore y	[ca$_3$ | ca$_3$]	int | int	udf | ca$_3$
6	iload x	[ ]	int	ca$_3$
7	ifeq 9	[int]	int	ca$_3$
8	goto 10	[ ]	int	ca$_3$
9	ret y	[ ]	int	ca$_3$
10	iload x	[ ]	int | int	udf | ca$_3$
11	ifne 1	[int | int]	int | int	udf | ca$_3$
12	return	[ ]	int | int	udf | ca$_3$

Fig. 5. Successful verification of the bytecode for Fig. 4.

The off-card verifier of [27], developed by Trusted Logic, uses a polyvariant data flow analysis for subroutines, i.e., it analyzes them in different contexts for different callers [17]. The contexts include subroutine call stacks, which are extended by jsr and shrunk by ret. While the code in Fig. 3 is accepted, the code in Fig. 5 is rejected. Apparently, that verifier includes checks for non-recursive calls to subroutines, as prescribed in [18, Sect. 4.8.2]: the path out of the subroutine and back to address 1 propagates the call stack with the subroutine to address 3, where a false recursive call is detected.

As a matter of fact, the technique proposed in [22] also includes subroutine call stacks and non-recursion checks, which cause the rejection of the code in Fig. 5. However, these can be easily removed from that technique.

As evidenced by the new technique, recursive subroutine calls are harmless to type safety. The prescription in [18, Sect. 4.8.2] prohibiting recursive subroutine calls is not only unnecessary, but also misleading, as manifested by the two examples above. Interestingly, Sun's verifier [26] accepts the code in Fig. 5 because it merges subroutine call stacks by computing their common sub-stacks; so, at address 10 the non-empty stack from 8 is merged with the empty stack from 0 resulting in the empty stack, which is propagated back to 1 and eventually to 3, with no false recursion being detected.

As part of the OVM project [21], Christian Grothoff independently implemented a verifier whose treatment of subroutines is the same as the new technique [12]. Trusted Logic's on-terminal verifier [29] (for the JEFF file format [14]), independently implemented by Alexandre Frey, also treats subroutines in the same way [9]; this verifier is very space- and time-efficient, thus demonstrating the practicality of the technique. Anyhow, this paper is the only publication that formalizes and proves properties about this approach to subroutines.

Alternatives to direct subroutine verification are subroutine in-lining [28] and variable splitting [1]. But given the possibly unstructured use of jsr and ret, determining subroutine boundaries (for in-lining) and variable usage (for splitting) may require a non-trivial analysis of the code. So, it is unclear whether bytecode rewriting followed by the simpler technique of [18, Sect. 4.9.2] is altogether simpler or faster than using the new technique on the original bytecode.

This paper is a short version of [4], while [5] is a comprehensive paper on the topic of subroutines. In [6] the new technique is lifted to a complete formalization of Java bytecode verification; in 2000, I used the Specware system [15] to formally derive a bytecode verifier from that formalization.

References

1. Ole Agesen, David Detlefs, and J. Eliot B. Moss. Garbage collection and local variable type-precision and liveness in Java virtual machines. In *Proc. 1998 ACM Conference on Programming Language Design and Implementation (PLDI'98)*, volume 33, number 5 of *ACM SIGPLAN Notices*, pages 269–279, June 1998.
2. Ken Arnold, James Gosling, and David Holmes. *The Java^{TM} Programming Language.* Addison-Wesley, third edition, 2000.
3. Alessandro Coglio. Improving the official specification of Java bytecode verification. In *Proc. 3rd ECOOP Workshop on Formal Techniques for Java Programs*, June 2001.
4. Alessandro Coglio. Simple verification technique for complex Java bytecode subroutines. Technical report, Kestrel Institute, December 2001. Revised May 2002. Available at http://www.kestrel.edu/java.
5. Alessandro Coglio. Java bytecode subroutines demystified. Technical report, Kestrel Institute, 2002. Forthcoming at http://www.kestrel.edu/java.
6. Alessandro Coglio. Java bytecode verification: A complete formalization. Technical report, Kestrel Institute, 2002. Forthcoming at http://www.kestrel.edu/java.
7. Stephen Freund. The costs and benefits of Java bytecode subroutines. In *Proc. OOPSLA'98 Workshop on Formal Underpinnings of Java*, October 1998.
8. Stephen Freund and John Mitchell. A type system for Java bytecode subroutines and exceptions. Technical Note STAN-CS-TN-99-91, Computer Science Department, Stanford University, August 1999.
9. Alexandre Frey. Private communication, May 2002.
10. Li Gong. *Inside Java^{TM} 2 Platform Security.* Addison-Wesley, 1999.
11. James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java^{TM} Language Specification.* Addison-Wesley, second edition, 2000.

12. Christian Grothoff. Private communication, June 2001.
13. Masami Hagiya and Akihiko Tozawa. On a new method for dataflow analysis of Java virtual machine subroutines. In *Proc. 5th Static Analysis Symposium (SAS'98)*, volume 1503 of *Lecture Notes in Computer Science*, pages 17–32. Springer, September 1998.
14. J Consortium. JEFF$^{\mathrm{TM}}$ file format, 2002. Available at http://www.j-consortium.org.
15. Kestrel Institute. Specware$^{\mathrm{TM}}$. Information at http://www.specware.org.
16. The Kimera project Web site. http://kimera.cs.washington.edu.
17. Xavier Leroy. Java bytecode verification: An overview. In *Proc. 13th Conference on Computer Aided Verification (CAV'01)*, volume 2102 of *Lecture Notes in Computer Science*, pages 265–285. Springer, July 2001.
18. Tim Lindholm and Frank Yellin. *The Java^{TM} Virtual Machine Specification.* Addison-Wesley, second edition, 1999.
19. Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis.* Springer-Verlag, Berlin, 1998.
20. Robert O'Callahan. A simple, comprehensive type system for Java bytecode subroutines. In *Proc. 26th ACM Symposium on Principles of Programming Languages (POPL'99)*, pages 70–78, January 1999.
21. The OVM project Web site. http://ovmj.org.
22. Joachim Posegga and Harald Vogt. Java bytecode verification using model checking. In *Proc. OOPSLA'98 Workshop on Formal Underpinnings of Java*, October 1998.
23. Zhenyu Qian. A formal specification of Java virtual machine instructions for objects, methods and subroutines. In Jim Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *Lecture Notes in Computer Science*, pages 271–312. Springer, 1999.
24. Robert Stärk, Joachim Schmid, and Egon Börger. *Java and the Java Virtual Machine: Definition, Verification, Validation.* Springer, 2001.
25. Raymie Stata and Martín Abadi. A type system for Java bytecode subroutines. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21(1):90–137, January 1999.
26. Sun Microsystems. Java 2 SDK Standard Edition version 1.3.1. Available at http://java.sun.com/j2se.
27. Sun Microsystems. Java Card Development Kit version 2.1.2. Available at http://java.sun.com/javacard.
28. Sun Microsystems. Connected, limited device configuration: Specification version 1.0a, May 2000. Available at http://java.sun.com/j2me.
29. Trusted Logic. TL Embedded Verifier. Information at http://www.trusted-logic.fr/solution/TL_Embedded_Verifier.html.
30. Phillip Yelland. A compositional account of the Java virtual machine. In *Proc. 26th ACM Symposium on Principles of Programming Languages (POPL'99)*, pages 57–69, January 1999.
31. Frank Yellin. Low level security in Java. In *Proc. 4th International World Wide Web Conference*, pages 369–379. O'Reilly & Associates, December 1995. Also available at http://java.sun.com/sfaq/verifier.html.