UC Irvine
ICS Technical Reports

Title
SSA-based Java bytecode verification

Permalink
https://escholarship.org/uc/item/48q8g7bt

Authors
Andreas
Probst, Christian W.
Franz, Michael

Publication Date
2004

Peer reviewed

eScholarship.org Powered by the California Digital Library
University of California

ICS

TECHNICAL REPORT

SSA-Based Java Bytecode Verification

Andreas Gal, Christian Probst, Michael Franz

Nov. 2004

TR# 04-22

Information and Computer Science
University of California, Irvine

SSA-Based Java Bytecode Verification

Andreas Gal, Christian W. Probst, and Michael Franz

Department of Computer Science
University of California, Irvine
Irvine, CA, 92697, USA
{gal,probst,franz}@uci.edu

Abstract. Java bytecode is commonly verified prior to execution. The standard verifier is designed as a black-box component that either accepts or rejects its input. Internally, it uses an iterative data-flow analysis to trace definitions of values to their uses to ensure type safety. The results of the data-flow analysis are discarded once that verification has completed. In many JVMs, this leads to a duplication of work, since definition-use chains will be computed all over again during just-in-time compilation. We introduce a novel bytecode verification algorithm that verifies bytecode via Static Single Assignment (SSA) form construction. The resulting SSA representation can immediately be used for optimization and code generation. Our prototype implementation takes less time to transform bytecode into SSA form and verify it than it takes Sun's verifier to merely confirm the validity of Java bytecode, with the added benefit that SSA is available "for free" to later compilation stages.

1 Introduction

Mobile programs can be malicious. A host that receives such mobile programs from an untrusted party or via an untrusted network connection will want a guarantee that the mobile code is not about to cause any damage. To this end, the Java Virtual Machine (JVM) pioneered the concept of *code verification*, by which a receiving host examines each arriving mobile program to rule out potentially malicious behavior even before starting execution.

The verifiers in practically all JVMs implement essentially the same worklist-based data-flow analysis algorithm [20, 28]. This analysis is necessary since the locations of temporary variables in the JVM are not statically typed. If verification is successful, then the *original* bytecode is forwarded to the JVM's execution component, which may be an interpreter or a just-in-time compiler. Specifically, beyond the result denoting whether or not verification was successful, all other information computed by the verifier is discarded and is not passed onwards. In many cases, this results in a duplication of work when a just-in-time compiler subsequently performs a very similar data-flow analysis all over again.

In this paper, we present an alternative verification mechanism that avoids such duplication of work. Instead of verifying Java Virtual Machine Language (JVML) bytecode directly, we first annotate it in such a way that the flow of values between instructions becomes explicit rather than going through the operand stack. We call this

first intermediate step *annotated JVML ($JVML_{\mathfrak{A}}$)*. In a second step, we then transform further into a *Static Single Assignment variant of JVML ($JVML_{SS\mathfrak{A}}$)*. Verifying programs in $JVML_{SS\mathfrak{A}}$ can then be performed in near linear-time and without requiring an iterative data-flow analysis. Moreover, since Dominator Tree construction and SSA generation are already performed during the verification phase, and can be re-used "for free" during subsequent code generation and optimization, our method can speed up just-in-time (JIT) compilation significantly.

Our benchmarks indicate that the aggregate time required for first transforming JVML via $JVML_{\mathfrak{A}}$ into $JVML_{SS\mathfrak{A}}$ and then verifying the $JVML_{SS\mathfrak{A}}$ representation is still less than the time needed for performing the standard verification algorithm directly on JVML. Our approach imposes no overhead for methods that will be interpreted without JIT compilation, because SSA-based verification is still overall faster than the traditional verifier. Moreover, by providing an SSA representation "for free", the initial cost of JIT compilation is lowered and larger parts of the program can be JIT compiled to native code for faster execution.

While our actual implementation covers the complete JVM language, length restrictions prevent us from discussing in detail the SSA-based verification algorithm for the entire JVML. Instead, we use a representative subset of JVML that we subsequently call $JVML_{S}$. The full implementation is available from the authors upon request and the performance benchmarks towards the end of the paper refer to the full JVM language, not just the subset used to discuss the algorithms.

The remainder of this paper is organized as follows: Section 2 gives a brief overview of the traditional Java bytecode verifier. Section 3 introduces $JVML_{S}$, the representative subset of the Java bytecode language. In Section 4 we show the transformation from $JVML_{S}$ via $JVML_{\mathfrak{A}}$ into $JVML_{SS\mathfrak{A}}$ and discuss how bytecode can be verified once the code is in $JVML_{SS\mathfrak{A}}$ form. In Section 5 we compare the performance of our method to that of Sun's standard verifier. Section 6 discusses related work and Section 7 contains our conclusion and points to future work.

2 JVML Bytecode Verification

JVML instructions ("bytecodes") can read and store intermediate values in two locations: the operand stack and local variables. These locations are ad-hoc polymorphic in that the same stack location or local variable can hold values of different types during program execution. Verification ensures that these locations are used consistently and intermediate values are always read back with the same types that they were originally written as.

Verification also ensures control-flow safety, but this is a comparatively trivial task. Conversely, verifying that the data flow is *well-typed* is rather complex. The JVM bytecode verifier [19, 20, 38] uses iterative data-flow analysis and an abstract interpreter for JVML instructions for this. Unlike JVM, the stacks and local variables of the abstract interpreter used for verification store *types*, rather than *values*. From the perspective of the verifier, JVM instructions are operations that execute on types and not on values.

$$
\begin{aligned}
\textit{instruction} &::= \textit{core} \mid \textit{dataflow} \\
\textit{core} &::= \texttt{iconst_}n \mid \texttt{lconst_}l \mid \texttt{iadd} \mid \texttt{ladd} \mid \texttt{ifeq}\ L \mid \texttt{return} \mid \texttt{return}(x) \\
\textit{dataflow} &::= \texttt{dup} \mid \texttt{dup_2} \mid \texttt{istore_}x \mid \texttt{iload_}x \mid \texttt{lstore_}x \mid \texttt{lload_}x
\end{aligned}
$$

Fig. 1. Instructions in $\text{JVML}_\mathcal{S}$. The arguments n, l, x, and L must fulfill the conditions $-1 \leq n \leq 5, l \in \{0, 1\}, x, L \in \mathbb{N}$.

JVML verification works on method level. With a co-inductional argument it follows that if every method is verifiable, the whole program is verifiable, too. In the rest of this paper, we use program and method interchangeably.

3 $\text{JVML}_\mathcal{S}$

For the remainder of this paper, we will use $\text{JVML}_\mathcal{S}$, a subset of the Java bytecode language defined following [35]. While very compact, $\text{JVML}_\mathcal{S}$ is complete enough to reason about properties of the Java bytecode language and to explain a number of difficulties that occur during the verification of JVML. Later on, we will slightly extend $\text{JVML}_\mathcal{S}$ to deal with exceptions and arrays. $\text{JVML}_\mathcal{S}$ has no construct that corresponds to JVML's subroutine construct. Subroutines are a significant complication when dealing with Java bytecode [9, 12, 25, 35] and have been shown to be not a very effective way of reducing code size [7]. The compilers of future versions of Java will probably no longer generate code containing the subroutine construct and we will disregard subroutines in the remainder of this paper. Our prototype implementation for the full JVML resolves the rare occurrence of a subroutine by inlining it into the body of the calling method.[1]

A $\text{JVML}_\mathcal{S}$ program is a sequence of instructions (Figure 1):

$$\textit{program} ::= \textit{instruction}^*$$

$\text{JVML}_\mathcal{S}$ supports object types and two scalar types, integers (`INT`) and long integers. Similar to JVML, long integers occupy two consecutive stack locations and variables. In contrast, objects and `INT` values take up only a single stack location or variable. To reflect this special property of long integers, we divide them into two halves: the bottom half is of type `LONG`, while the top half is of type `LONG'`. Accordingly, long integer values are divided into the two sets *long* and *long'*. The static semantics of $\text{JVML}_\mathcal{S}$ guarantees that both halves always occupy consecutive stack locations or variables, and that operations on both halves are always atomic.

Instructions operate on an operand stack. Additionally, values can be stored in variables. Variables are non-negative integers that correspond to local variables in JVML.

[1] We have studied numerous bytecode applications including the Eclipse framework, different Java APIs, and the SPEC benchmarks. Of approximately 5.4 million instructions we only found 0.24% to be in subroutines. The average size of a subroutine was 7 instructions and it was only called 2 times.

Code	Abstraction
`lconst_0`	$\langle$LONG, LONG'$\rangle$
`lconst_1`	$\langle$LONG, LONG', LONG, LONG'$\rangle$
`iconst_1`	$\langle$LONG, LONG', LONG, LONG', INT$\rangle$
`ifeq` L	$\langle$LONG, LONG', LONG, LONG'$\rangle$
`dup_2`	$\langle$LONG, LONG', LONG, LONG', LONG, LONG'$\rangle$
`ladd`	$\langle$LONG, LONG', LONG, LONG'$\rangle$
L: `ladd`	$\langle$LONG, LONG'$\rangle$
`lstore_0`	$\langle\rangle$ *local variable 0* = LONG, *local variable 1* = LONG'

Fig. 2. Example code and the abstractions computed for stack states and variables. Each stack abstraction contains the bottom of stack on the left and the top of stack on the right.

The instruction set of JVML_ς consists of two kinds of instructions: *core* instructions and *data-flow* instructions. Core instructions operate on values stored on the operand stack, while data-flow instructions such as `dup`, `dup_2`, `iload_`x, and `istore_`x only facilitate the flow of values between core instructions by manipulating the state of the operand stack and exchanging values between operand stack and variables.

Values are produced by core instructions and can be consumed by other core instructions. During the lifetime of a value it can reside on the operand stack or in variables. Values can reside in multiple locations at the same time. Data-flow instructions neither produce nor consume values[2], but merely transport values between stack locations and variables.

Figure 1 gives the grammar for JVML_ς instructions. Informally, the instructions behave as follows:

- `iconst_`n and `lconst_`l push a numeric constant onto the operand stack (as INT or as pair (LONG, LONG'), respectively),
- `iadd` and `ladd` pop two INT values or two (LONG, LONG') value pairs, respectively, from the operand stack, add them, and leave the result on the stack,
- `ifeq` L pops an INT value from the stack and transfers control to label L if the value is unequal zero,
- `dup` duplicates the value on top of the operand stack,
- `dup_2` duplicates the two topmost values on the operand stack. It can be used to duplicate two INT values or one (LONG, LONG') pair,
- `istore_`x and `iload_`x transfer an INT value between the operand stack and variable x,
- `lstore_`x and `lload_`x transfer a (LONG, LONG') pair between the operand stack and variable x. The bottom half (LONG) of the long variable is stored in or read from variable x, the top half (LONG') is stored in or read from variable $x + 1$, and

[2] The *pop* instruction in JVML is a data-flow instruction. While it might seem to consume a value at the first glimpse, it actually merely manipulates the stack in a way that this particular alias of the value cannot be consumed by any core instruction.

- `return` and `return`(x) stop the execution of the program.

For a presentation of the semantics of JVML$_{\mathcal{S}}$ (as well as the other languages used in this paper) please refer to [11]. Due to length constraints we only list some of the main properties ensured by the semantics. `lconst_`l and `ladd` use the type pair $(\text{LONG}, \text{LONG}')$ atomically. For `dup_2`, two reference types or integers are allowed as topmost stack cells, or a properly formed long integer pair $(\text{LONG}, \text{LONG}')$. Similarly, long integers also occupy two consecutive variables when pushed on the stack or stored in a variable (`lstore_`x and `lload_`x).

Figure 2 shows a simple JVML$_{\mathcal{S}}$ example. The code pushes two long constants on the stack, based on an integer may add the second constant to itself, and finally adds the remaining two long integers and stores them in variable 0. The right part of the figure gives the abstractions computed for the states of stacks and variables.

4 Verification in Static Single Assignment Form

The central responsibility of the Java bytecode verifier is to ensure that stack locations and local variables are used in a type-safe manner. This is the case if the definitions and uses of values have compatible types. To ensure this, the verifier algorithm has to determine the types of all stack locations and variables for each instruction in P. It does so by using an iterative data-flow analysis to trace definitions of values to their uses.

In JVML, there is no obvious link between the definition of a value and its uses. However, even if definition-use chains [1] were available for each value in a JVML program, it would still be impossible to verify a Java program in a single pass by comparing the type of each definition with its uses.

The reason for this becomes more obvious if we consider how instructions are categorized in JVML$_{\mathcal{S}}$. In JVML$_{\mathcal{S}}$, only *core* instructions define and use values. *Data-flow* instructions merely facilitate the flow of values between core instructions. Core instructions are always *self-typed*, i.e. the expected types of any consumed operands and the types of any produced values are known statically.[3]

In contrast, data-flow instructions are not self-typed, i.e. they are polymorphic. In general, it is not possible to determine the type of the value produced by a data-flow instruction without knowing the type of its operands. The result type of a `dup` instruction, for example, depends on the type of the value on top of the stack. While local variable access instructions such as `iload_`x suggest stronger static typing, this works for scalar types only. In the JVM, object references are written and read from local variables using `astore_`x and `aload_`x, and data-flow analysis is still necessary to determine the precise type of the variables accessed.

The goal of our approach is to avoid an up-front iterative data-flow analysis to verify JVML. Instead, the JVML code is annotated so that the flow of values between core instructions becomes explicit instead of relying on an operand stack. This enables

[3] The JVML array load instruction `aaload` is a core instruction, but it is not self-typed as its return value has the element type of the array, which obviously depends on the type of the array operand. We will show that while not self-typed in JVML, `aaload` is in fact self-typed in JVML$_{\mathcal{SSA}}$.

us to eliminate all non-self-typed instructions (data-flow instructions) from the code after SSA construction. These instructions are no longer needed because they only facilitate data flow, but do not actually compute anything. Once the code consists of self-typed instructions (core instructions) only and is in JVML_{SSA} form, it is possible to perform type-safety checks by directly relating the type of each definition with the corresponding uses (*definition-use verification*). As in other SSA based analyses, a data-flow analysis becomes obsolete.

In the remainder of this section, we develop an algorithm that performs definition-use verification in SSA form. The algorithm consists of the following steps:

Step 1. Starting from the entry point, follow all branches and annotate all reachable instructions with the stack depth at that location.

Step 2. Calculate the Iterative Dominance Frontier (IDF) for all stack cells and local variables and place ϕ-nodes in the control-flow graph (CFG) accordingly.

```
ANNOTATE(i, s)
 1  if visited[i]
 2    then if s ≠ StackDepth[i]
 3           then FAIL("Stack depth mismatch")
 4         return
 5  visited[i] := true;
 6  while true
 7  do StackDepth[i] := s;
 8     switch (i)
 9       case iconst_n : ADD(DEF[s], i);  s := s + 1;
10       case lconst_l : ADD(DEF[s], i);  ADD(DEF[s + 1], i);  s := s + 2;
11       case iadd : ADD(DEF[s − 2], i);  s := s − 1;
12       case ladd : ADD(DEF[s − 4], i);  ADD(DEF[s − 3], i);  s := s − 2;
13       case ifeq L : s := s − 1;
14       case dup : ADD(DEF[s], i);  s := s + 1;
15       case dup_2 : ADD(DEF[s], i);  ADD(DEF[s + 1], i);  s := s + 2;
16       case iload_x : ADD(DEF[s], i);  s := s + 1;
17       case lload_x : ADD(DEF[s], i);  ADD(DEF[s + 1], i);  s := s + 2;
18       case istore_x : ADD(DEF[MaxStack + x], i);  s := s − 1;
19       case lstore_x :
20            ADD(DEF[MaxStack + x], i);  ADD(DEF[MaxStack + x + 1], i);  s := s − 2;
21     if (s < 0) ∨ (s >= MaxStack)
22       then FAIL("Stack underflow/overflow")
23     if P[i] = ifeq L
24       then ANNOTATE(TARGET(i), s);
25     if P[i] = return
26       then return
27     i := NEXTINSTRUCTION(i);
```

Fig. 3. Step 1: Annotate instructions with the corresponding stack depth at that location. The array *visited* is used to ensure that all instructions are visited at most once. When encountering an already visited instruction, the previous stack depth is compared to the new stack depth.

Step 3. Traverse the CFG in Dominator Tree order, assigning a unique SSA-name to all stack and local variable definitions and recording the type for each definition. Data-flow instructions are eliminated through copy propagation.

Step 4. After merging the types in all ϕ-instructions, iterate over all instructions and match the expected operand types to the actual definition types. Whenever a type error occurs, verification fails.

The objective of Step 1 is to transform the program so that the operand stack is no longer necessary. Each instruction is annotated with the current stack depth so instructions no longer depend on the stack to connect operands to their definitions. We use a simple recursive algorithm to perform the annotation (Figure 3).[4] ANNOTATE takes two arguments, the instruction i to visit next and the current stack depth s. Starting with the first instruction in the program and an empty stack ($s = 0$), for each instruction the stack depth is recorded in an array *StackDepth*. A second array *visited* is used to ensure that every instruction is visited at most once. The runtime of the algorithm is thus linear in the program length. The *visited* array is initially set to *false* for all instructions. When ANNOTATE reaches an instruction for the first time, the stack depth is recorded in *StackDepth*. When ANNOTATE revisits an instruction, it checks that the current stack depth and the previously recorded depth match (line 2). Otherwise, the verification fails.

Additionally to populating *StackDepth*, ANNOTATE also tracks the instructions that define values—either on the operand stack or in local variables—and records these in an array of sets *DEF*. Values on the stack are labeled relative to their distance to the bottom of the stack (BOS). The value produced by an `iconst` instruction executed on a previously empty stack, for example, would be labeled with 0, because it is currently at the bottom of the stack. Executing another *iconst* on this stack would produce a value that is labeled with 1, because its distance from the bottom of the stack is one stack cell. This labeling scheme allows to assign a numeric representation to values produced on the operand stack. Additionally, the label for a value will remain constant until it is removed from the stack. *DEF* is also used to track definitions of values that are stored in JVM local variables. To avoid collisions with stack cells, local variables are numbered from *MaxStack*, which is the maximum stack depth used by the program.

As explained above, following the JVML machine model we split long integers into two halves in $\text{JVML}_{\mathcal{S}}$. Thus, instructions operating on long integers push and pop pairs of values onto and from the operand stack (c.f. `lconst`, `ladd`, `lload`, and `lstore` in Figure 3). Correspondingly, for each definition of a long integer two entries are created in *DEF*, one for the bottom half (type LONG), and one for the top half (type LONG'). For the remainder of the verification process, we will consider these two halves as separate and individual values.

Figure 4 shows the result of the annotation for the example program from Figure 2. The first `lconst_0` instruction pushes the bottom half (LONG) of the long integer onto the stack, followed by the top half (LONG'). Correspondingly, the bottom half is labeled with 0 and the top half with 1. For both values the defining instruction ($PC = 1$) is

[4] Our verifier does not reject type-unsafe code that is unreachable (dead code). Instead, it is essentially stripped from the program and does not appear in the $\text{JVML}_{\mathcal{SSA}}$ representation. Such programs are also not rejected by the traditional Java verifier.

PC	Instruction	*StackDepth*	*DEF*
1	`lconst_0`	0	$DEF[0] = \{1\}, DEF[1] = \{1\}$
2	`lconst_1`	2	$DEF[2] = \{2\}, DEF[3] = \{2\}$
3	`iconst_1`	4	$DEF[4] = \{3\}$
4	`ifeq` L	5	
5	`dup_2`	4	$DEF[4] = \{3, 5\}, DEF[5] = \{5\}$
6	`ladd`	6	$DEF[2] = \{2, 6\}, DEF[3] = \{2, 6\}$
7	L: `ladd`	4	$DEF[0] = \{1, 7\}, DEF[1] = \{1, 7\}$
8	`lstore_0`	2	$DEF[6] = \{8\}, DEF[7] = \{8\}$

Fig. 4. Example code after annotation with ANNOTATE. Each instruction is labeled with the stack depth prior to the execution of that particular instruction. Additionally, each definition of a value is recorded in *DEF*. For long integers, each half is registered in *DEF* separately. Since *MaxStack* for the program is 6, the `lstore_0` instruction creates values in $DEF[6]$ and $DEF[7]$.

```
DEFINE(t)
1  n := label++;
2  type[n] := t;
3  return n;

ISPHI(x)
1  return (x < PhiCount)
```

```
PLACEPHI()
1  for each b in BasicBlocks
2  do PhiBase[b] := label;
3     for each y in IDF(b)
4     do DEFINE(⊥);
5  PhiCount := label;
```

Fig. 5. Step 2: Placing ϕ-nodes. DEFINE is called every time a new value is defined, using *label* to assign a unique SSA-name to each definition. The result types of ϕ-nodes are initialized to $\perp$. The actual type will be updated in Step 4 (RESOLVEPHI). The total number of ϕ-nodes inserted is recorded in *PhiCount*. As ϕ-nodes are numbered consecutively from 0 to $PhiCount - 1$, this property can be used to check whether a particular definition is a regular instruction or a ϕ-node (ISPHI).

recorded in *DEF*. As discussed above, definitions of local variables are labeled starting from *MaxStack*. Since *MaxStack* for the example program is 6, the final `lstore_0` instruction records the definition of the the two local variable halves in *DEF* as $6 + 0$ and $6 + 1$.

Step 2 of our verification algorithm first computes the Iterative Dominance Frontier for all definitions listed in *DEF*. It has been shown that this can be done in near-linear time (c.f. [30, 31]), and we will not discuss the IDF calculation in detail.

Once the IDF is known for all stack cells and local variables, ϕ-nodes are inserted accordingly. The corresponding algorithm PLACEPHI is shown in Figure 5. PLACEPHI does not actually modify the program to place the ϕ-nodes, but instead records information about their placement in *PhiBase*, *PhiCount*, and *PhiOperands*. DEFINE is used to introduce a new SSA-name for each encountered definition. For this, DEFINE maintains a global counter *label*, which is initially set to 0. For each basic block b, the SSA-name of the first ϕ-node in that block is recorded in $PhiBase[b]$. The total count of ϕ-nodes

```
RENAME(i, V)
 1  while true
 2  do IN[i] := ∅;
 3     s := StackDepth[i];
 4     switch (i)
 5       case iconst_n : OUT[i] := {V[s] := DEFINE(INT)};
 6       case lconst_l :
 7            OUT[i] := {V[s] := DEFINE(LONG), V[s + 1] := DEFINE(LONG')};
 8       case iadd :
 9            IN[i] := {V[s − 2], V[s − 1]};
10            OUT[i] := {V[s − 2] := DEFINE(INT)};
11       case ladd :
12            IN[i] := {V[s − 4], V[s − 3], V[s − 2], V[s − 1]};
13            OUT[i] := {V[s − 4] := DEFINE(LONG), V[s − 3] := DEFINE(LONG')};
14       case ifeq L :
15            IN[i] := {V[s − 1]};
16       case dup : V[s] := V[s − 1]; KILL(i);
17       case dup_2 : V' := V; V[s] = V'[s − 2]; V[s + 1] := V'[s − 1]; KILL(i);
18       case iload_x : V[s] := V[x]; KILL(i);
19       case lload_x :
20            V[s] := V[x]; V[s + 1] := V[x + 1]; KILL(i);
21       case istore_x :
22            V[MaxStack + x] := V[s − 1]; KILL(i);
23       case lstore_x :
24            V[MaxStack + x] := V[s − 2]; V[MaxStack + x + 1] := V[s − 1]; KILL(i);
25     for each x in IN[i]
26     do if type[x] = ⊤
27          then FAIL("undefined variable");
28     if ENDOFCURRENTBASICBLOCK(i)
29       then for each ii in SUCCESSORBLOCKS(i)
30            do q := PhiBase[ii];
31               for r := 0 to (MaxStack + NumLocals − 1)
32               do if ii ∈ IDF[r]
33                    then ADD(PhiOperands[q++], V[r]);
34            for each ii in DOM[i]
35            do V' := V;
36               RENAME(ii, V');
37            return
38     i := NEXTINSTRUCTION(i);
```

Fig. 6. Step 3: Rename definition and uses of stack cells and variables. The current SSA-name of variables in stored in an array *V*. The CFG is visited in dominator tree order. At the end of each basic block, the ϕ-nodes in all successor blocks in the CFG are updated with the current SSA-name of the ϕ-operands. Data-flow instructions are eliminated through copy propagation. For each core instruction, the SSA-name of its operands is tracked in an array of lists *IN*, the SSA-name of values it defines in an array of lists *OUT*.

```
RESOLVEPHI(w, visited)
1  t := ⊥;
2  for each x in PhiOperands[w]
3  do if ISPHI(w)
4        then t := MERGETYPES(t, type[x]);
5  visited[w] := true;
6  for each x in PhiOperands[w]
7  do if ISPHI(w) ∧ ¬visited[x]
8        then t := MERGETYPES(t, RESOLVEPHI(w));
9  return t;
```

```
CHECK(P)
 1  for w := 0 to (PhiCount − 1)
 2  do for each x in PhiOperands[w]
 3      do if ¬ISPHI(x)
 4            then type[w] := MERGETYPES(type[w], type[x]);
 5  for w := 0 to (PhiCount − 1)
 6  do type[w] := RESOLVEPHI(w);
 7  for each i in P
 8  do for each j in GETOPERANDS(i)
 9      do t := type[V[j]];
10         if MERGETYPES(t, STATICOPERANDTYPE(i,j)) ∈ {⊥, ⊤}
11           then FAIL("Operand type mismatch.");
12         if (t = LONG) ∧ (IN[i, j] ≠ IN[i, j + 1])
13           then FAIL("Invalid LONG integer.");
```

Fig. 7. Step 4: Algorithm for definition-use verification. First, the type of all ϕ-nodes is determined through a depth-first search. After that, the remaining core instructions are visited and the definition type of their uses is match to the expected operand type.

inserted is recorded in *PhiCount*. As ϕ-nodes are placed first before any other definitions are processed, their numbering ranges from 0 to $PhiCount - 1$. This property is used by ISPHI to determine whether a certain definition given by its SSA-name is a ϕ-node or a regular definition. The type of each definition is recorded by DEFINE in the array *type*. As the types of ϕ-nodes are not yet known, they are initially recorded as type $\perp$.

After placing all ϕ-nodes, Step 3 visits each reachable instruction in the program in dominator-tree (DT) order and eliminates all data-flow instructions. We will not further elaborate on the calculation of dominator-trees [18], which also runs in near-linear time, but assume that one has been calculated and that an array of sets *DOM* contains for each basic block the list of its dominated blocks.

Using the algorithm *Rename* shown in Figure 6, all references of core instructions to stack cells and local variables are resolved to SSA-names (line 5-14). The array *IN* contains for each instruction the SSA-names of all its uses. Array *OUT* stores the SSA-names of all definitions of an instruction. While RENAME iterates over the dominator tree, the current SSA-names of all stack cells and local variables are stored in a renaming table (array V).

Instruction	*StackDepth*	IN	OUT	V	*PhiOperands*
`lconst_0`	0		2, 3	$V[0] = 2, V[1] = 3$	
`lconst_1`	2		4, 5	$V[2] = 4, V[3] = 5$	
`iconst_1`	4		6	$V[4] = 6$	
`ifeq L`	5	6			
`dup_2 (*)`	4			$V[4] = 4, V[5] = 5$	
`ladd`	6	4, 5, 4, 5	7, 8	$V[2] = 7, V[3] = 8$	
L: ϕ	4		0	$V[2] = 0$	4, 7
ϕ	4		1	$V[3] = 1$	5, 8
`ladd`	4	2, 3, 0, 1	9, 10	$V[0] = 9, V[1] = 10$	
`lstore_0 (*)`	2			$V[7] = 9, V[8] = 10$	

Fig. 8. IN, OUT, V, and *PhiOperands* for the example code from Figure 4. ϕ-nodes are shown here for completeness only and are not actually inserted into the program. After renaming is complete, the data-flow instructions `dup_2` and `lstore_0` (*) are removed from the program (KILL)

Data-flow instructions do neither produce nor consume any values and thus are not recorded in *IN* and *OUT*. They only affect the renaming table V. The `dup` instruction (line 15), for example, pushes a copy of the TOS value onto the stack. As our operand stack has pre-increment semantics, the TOS value is labeled $s - 1$, with s being the current stack depth (*StackDepth*$[i]$). Accordingly, the new value `dup` pushes onto the stack will have the label s. To ensure that downstream accesses to either of these two stack cells will point to the same definition (which has just been duplicated by `dup`), RENAME copies the original SSA-name from $V[s - 1]$ into the new TOS $V[s]$. Once the renaming table has been adjusted, the data-flow instruction itself becomes obsolete and is removed using the helper function KILL. Successive accesses will be guided by the renaming table directly to the corresponding definition.

Whenever the last instruction of a basic block is encountered (line 28), the current SSA-name for each stack cell and local variable is propagated into all successor basic blocks in the control-flow graph, if a corresponding ϕ-node exists there. The operands of each ϕ-node are stored in an array of lists *PhiOperands*. The index of the first ϕ-node in each basic block b has already been recorded by PLACEPHI (line 2) in *PhiBase*$[b]$. Because PLACEPHI consecutively numbers ϕ-nodes in each basic block, we can simply set q to the index of the first ϕ-node (Figure 6, line 30) and then increment it for each successor basic block in *IDF*$[r]$ (line 33).

Once all ϕ-nodes in the successor blocks have been updated, RENAME continues to descend the dominator tree (line 36). For each child node in the dominator tree, a copy of the renaming table is created (line 35) to make sure there is no cross-interference between the effects of dominated child nodes.

After Step 3, the program is in JVML_{SSA} form and we can now perform the actual type checking (Step 4). First, the type of each ϕ-node is computed using RESOLVEPHI

(Figure 7). Similar to type inference performed by the traditional verifier, the type of ϕ-nodes is the common supertype of each definition the ϕ-node refers to (ϕ operands). Each of these definitions is either produced by a core instruction or another ϕ-node, as all data-flow instructions have already been eliminated. For core instructions the type of values they define can be looked up in the array *type*, where it was stored by DEFINE (Figure 6). For ϕ-nodes, however, it might not yet have been calculated.

To calculate the type of ϕ-nodes, CHECK first calculates the consensus type of all non-ϕ (core instruction) references of all ϕ-nodes (Figure 7, line 1-4). The consensus type of a ϕ-node is the supertype of all its operands for which the precise type is already known because its defined by a core instruction. The consensus type for each ϕ-nodes is stored in *type*, which previously only contained $\bot$ for all ϕ-nodes.

To merge types and to determine the nearest common supertype, a helper function MERGETYPES is used, which merges two types t_1 and t_2 according to the rules of the Java type system. Merging incompatible types results in $\top$, and merging any type t with $\bot$ returns t.

After calculating the consensus types, RESOLVEPHI is invoked on each ϕ-node and performs a recursive depth-first merge of all ϕ-operands of ϕ-nodes, leaving the definitive type of all ϕ-nodes in the array *type*.

With *type* now containing a complete list of all definitions and their types for core instructions and ϕ-nodes, all we have to do to complete type-checking is to match for each core instruction the type of its uses with the corresponding definitions (Figure 7, CHECK, line 10). For this, CHECK relies on the helper function STATICOPERAND-TYPE, which returns the expected type of an operand according to the static semantics of a core instruction.

Additionally, we have to verify that long integers are still used in proper pairs. The numbering scheme used by DEFINE is crucial for verifying this property. For long integers, DEFINE is invoked twice consecutively, resulting in the two long integer halves being labeled with two consecutive labels. This is verified by CHECK (line 12) to make sure that the two halves used together actually belong together.

If all checks in Step 4 are passed, the program is considered type-safe and passes our verifier.

Considering only the dynamic semantics, the data flow we just verified is obviously equivalent to the data flow that would have resulted by interpreting the original $\text{JVML}_\mathcal{S}$ program. However, because data-flow instructions have been eliminated, some of the restrictions enforced by their static semantics do no longer apply. E.g., the following $\text{JVML}_\mathcal{S}$ program will be rejected by the Java verifier, but is valid in $\text{JVML}_{\mathcal{SSA}}$:

```
1: lconst_0
2: istore_1
3: iload_1
4: lstore_2
```

In this example, in Line 1 a long integer is pushed onto the stack as a pair of halves (LONG, LONG′). Partially storing the long integer in an integer register as shown in Line 2 is rejected by the traditional verifier. In contrast, since our verifier does not consider the typing rules of data-flow instructions, it accepts this code fragment,

because the (LONG, LONG′) pair defined in Line 1 is restored on the stack before it is used in Line 4. It is important to note that this program, while rejected by the JVM, is perfectly safe when executed.

4.1 Exceptions

The $JVML_S$ subset we have presented in Section 3 does not model exceptions. However, extending our verifier algorithm to support exceptions is straightforward. In JVML, exception handlers are regular code fragments. A list of exception handlers specifies what code areas a particular handler guards. If an exception of matching type is thrown, control is transfered to the handler. Otherwise, it is rethrown in the outer scope. While the stack is cut if an exception occurs, exception handlers can access local variables defined during method execution. Thus, to ensure that the Iterative Dominance Frontier is calculated properly along regular control flow edges as well as exception edges, instructions that can throw exceptions have to be connected to all potential exception handlers within the same method. This means that during the stack depth annotation (Step 1) and for the calculation of the Dominator Tree, all instructions that can throw exceptions have to be treated as implicit branch instructions and they terminate basic blocks. As the JVML specification does not guaranteed that code regions guarded by an exception handler must contain instructions that can actually raise that kind of exception, the resulting CFG is not always fully connected. This has to be considered during calculation of the Dominator Tree.

4.2 Arrays

The complete JVM language uses the `aaload` instruction to load values from an array. Multi-dimensional arrays are modeled as arrays of array objects. To read a number from a two-dimensional integer array (*int*[][]), an `aaload` instruction followed by an `iaload` instruction are used. First the `aaload` instruction returns a reference to an array of integers (*int*[]), and then the `iaload` returns the actual numeric value of type *int*. While `aaload` is obviously a core instruction, it does not fit the definition of core instructions we gave for $JVML_S$, because it is not self-typed. With a small extension to our verifier algorithm, however, it is possible to support the proper array semantics of JVML.

After completing the operand cells and variable renaming in Step 3, the program is in SSA-form. This allows us to determine the precise type of each `aaload` instruction using a simple depth-first search (DFS). Starting at any `aaload` instruction we track its array operand to its definition, which is trivial due to the SSA-form and the copy propagation we have performed in Step 3. If the definition points to another `aaload` instruction, its return type has to be resolved first. Otherwise, the return type of the original `aaload` instruction can be derived from the type of the definition that it refers to. After visiting all `aaload` instructions, they are annotated with their precise return type and become self-typed. It is always possible to resolve such chains of `aaload` instructions, as typing rules forbid any circular flow of array operands between `aaload` instructions.

	# of methods	method size [ø / max]	stack depth [ø / max, cells]	local variables [ø / max, bytes]
java/*	6490	41.36 / 4065	2.74 / 14	2.47 / 37
java/io	1213	38.12 / 1295	2.39 / 8	2.35 / 15
java/lang	1336	38.41 / 4065	2.32 / 10	2.17 / 37
java/math	405	72.67 / 3041	3.16 / 8	3.73 / 29
java/nio	2096	26.80 / 417	3.05 / 11	2.31 / 15
java/util	2359	49.21 / 2916	2.64 / 14	2.62 / 25

Fig. 9. Characteristics of the test set we used to compare the runtime of our SSA-based verifier with the runtime of the traditional verifier. The method count of the individual subsets does not add up to 6490, because certain JVM-internal classes such as *String* and *Class* are preloaded and always verified by *preverify*.

4.3 Object Initialization

Ensuring proper object initialization in Java has been subject of intensive research [10, 27, 34]. The following code is legal in Java:

```
1: new A
2: dup
3: invokespecial A.<init>
4: invokevirtual A.m
```

In this example, an object of type A is instantiated in Line 1, and the reference to that object is duplicated in Line 2 using the dup instruction. Line 3 contains the constructor call that initializes the object. It must appear before calling any other methods (Line 4). To verify proper object initialization, an alias analysis has to be performed to make sure that at least one alias of each newly allocated object is initialized along all possible paths before any other action is performed on the object. The traditional approach is to perform this alias analysis as part of the iterative data-flow analysis.

In our verifier, we use a different approach. We extend the static semantics of new in such a way that it produces two return values: the regular return value, which is a reference to the newly allocated object, and an initialization guard. Guard variables are void in the dynamic semantics and do not incur any runtime overhead. In the renaming array V, this guard variable initially maps to $\perp$, indicating that it has not been defined yet. If a constructor call is encountered, a new definition is entered in V for the associated guard variable. All instructions operating on object references contain a rule in their static semantics that the guard associated with their array operand must not be undefined in V.

To accommodate partial initializations, for example only along one case of an *if/else* branch, during renaming the definition of guard variables is implicitly reverted to $\perp$ in the Iterative Dominance Frontier of the constructor invocation instruction.

5 Benchmarks

To evaluate the performance of our SSA-based verifier, we have implemented a prototype verifier based on the algorithm presented in this paper. Our prototype inlines sub-

	verifier (DFA) [s]	verifier (SSA) total [s]	DOM [s]	IDF [s]
java/*	14.09	12.56	1.19	3.3
java/io	2.84	2.45	0.24	0.59
java/lang	2.86	2.62	0.25	0.69
java/math	1.69	1.24	0.09	0.34
java/nio	3.28	2.89	0.30	0.73
java/util	6.66	5.42	0.48	1.28

Fig. 10. Benchmark results for 100 verifications of each method in each test set.

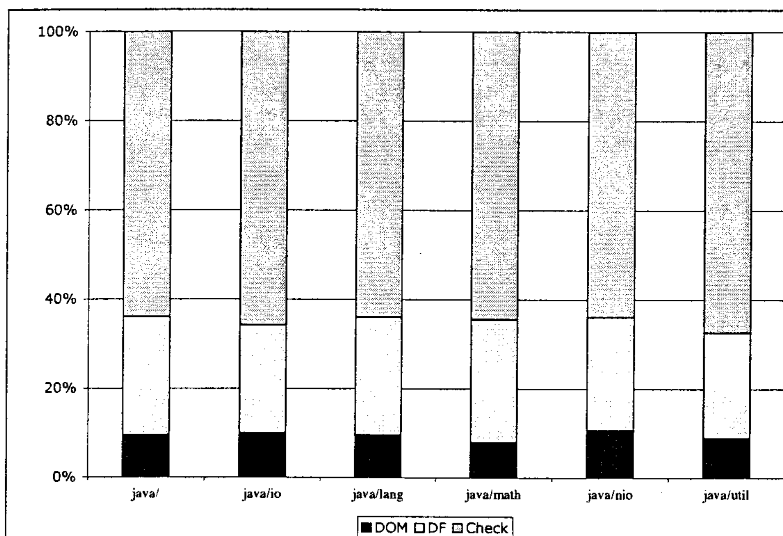

Fig. 11. Contribution of Dominator Tree construction (DOM) and Iterative Dominance Frontier calculation (DF) to the total SSA-based verification time.

routines before verification. In order to arrive at a fair comparison with Java's standard verifier, we use the same modified Java code with inlined subroutines also for the JVML verification benchmarks. Our rationale behind this is that the subroutine construct in Java is obsolete and will probably be removed in future versions of the Java virtual machine. Furthermore, our current algorithm depends on the fact that the control-flow graph can be recovered quickly from JVML code. In the presence of subroutines, this is not always the case as returning edges from subroutines are not explicit.

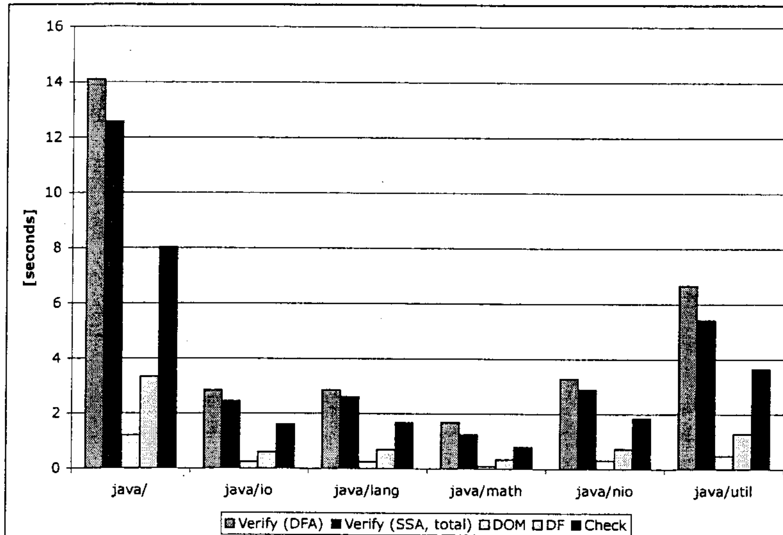

Fig. 12. Comparison of the total runtime of the traditional DFA-based verifier with our SSA-based verifier.

As a comparative benchmark, we compare the total runtime of our SSA-based verifier to the runtime of Sun's DFA-based verifier. In both cases, we use the preverify tool shipped as part of Sun's Kilobyte Virtual Machine [36] to inline all subroutine calls before measuring the actual verification times. Both verifiers are implemented in C and use the same underlying framework to read and represent Java class files as both are embedded in Sun's preverify tool. Our implementation uses the DJ-graph algorithm [30, 31] to obtain the *IDF*. The Dominator Tree is calculated according to Lengauer et al. [18].

To eliminate any cache effects and to compensate for timing errors, both verifiers are run one hundred times on each method from the test set. Unfortunately, there is currently no established set of benchmarks to test the performance of verifiers. Benchmark suites such as SPECjvm [32] are designed to evaluate the performance of code execution, *not code verification*. Thus, we have decided to use various parts of the Java Runtime Libraries (JDK 1.4.2) as test set (Figure 9). All measurements were conducted on a Pentium4 2.53GHz CPU with 512MB of RAM, running under RedHat Linux 9. An overview of the benchmark results is shown in Figure 10.

Figure 11 shows the contribution of Dominator Tree construction (DOM) and Iterative Dominance Frontier calculation (DF) to the overall run time of our algorithm. Both

of these analyses can be re-used by the just-in-time compiler. The actual time solely dedicated to verification is approximately 65% (Check).

Figure 12 compares the total runtime of the traditional DFA-based verifier with our SSA-based verifier. Verification in SSA-form is approximately 15% faster than the traditional algorithm when comparing the total runtime. Not considering the time spent to calculate DOM and DF, SSA-based verification is approximately 45% faster.

6 Related Work

Java bytecode verification has been explored extensively in the past [19, 33]. In addition to the informal description of the JVM [20], a number of formal specifications of the JVML and its verifier have been proposed [8, 10, 35], and proven to be sound [26, 13–15, 34, 4]. Resulting from the formalization of the verification process, improvements over the original specification have been proposed [5, 29]. In this context, subroutines are of particular interest and several type systems have been proposed for them [35, 25, 27, 16]. All these approaches have in common that they rely on some form of iterative data-flow analysis [19, 28] to decide type-safety.

Proof-carrying code (PCC) [23, 22] addresses this problem by relieving the code consumer of the burden to verify the code. Instead, the code producer computes a verification condition based on a public safety policy and proves it to be true for the program. This proof is shipped to the code consumer along with the code. Upon receipt, the code consumer recomputes the verification condition and can then check whether the attached proof indeed establishes the verification condition as claimed by the code producer. However, this systematic advantage for PCC does not come entirely for free as additional information has to be shipped to the code consumer, which inflates the size of mobile code components. SSA-based verification, in contrast, has the advantage that it can operate on the standard Java class file format [20] and does not rely on any additional annotations. While more recent improvements over the original PCC idea have significantly reduced the sizes of the proofs and the verifier [24, 3], we still believe that our approach is a meaningful alternative to PCC in certain domains, in particular if backward compatibility to existing, un-annotated, Java code is desired.

The split verifier approach [37] is very similar to PCC. It annotates the JVML with the fixed-point of the data-flow analysis otherwise performed by the JVM during class loading. For annotated class files the verification is reduced to confirming that the annotation is indeed a valid fixed-point, which can be completed in near-linear time. Just as in the case of PCC, the annotations enlarge the overall size of class files, while our approach does not rely on any additional annotation.

Inherently safe mobile code representation formats such as SafeTSA [2] eliminate the need for verification as mobile code is stored in a self-consistent format that cannot represent anything but well-formed and well-typed programs. Just like PCC, such formats have a systematic advantage over SSA-based verification, but require abandoning the existing Java class file format, which is not always acceptable. Our approach and SafeTSA have in common that they both make the code available to the JIT in SSA-form, which can be used to speed up code generation.

SSA-based representations have been used in several approaches to compilation of bytecode. Marmot [6] is a research platform for studying the implementation of high-level programming languages. The main difference to our work is that Marmot only accepts verifiable programs. This property of the input program allows to make certain assumptions on properties of the code, e.g. about the types of local variables and stack entries. Similar to our work, Marmot inlines subroutines to avoid complex encoding as normal control flow similar to Freund [7].

Even closer related to our approach is the work of League et al. [17]. λJVM, a functional representation of Java bytecode, makes data flow explicit, just like our work. They also split verification up in two phases, one during the construction of λJVM code, and a simple type checking later. However, they initially perform a regular data-flow analysis to infer types for the stack and local variables at each program point. This is in contrast to our approach, were the reason for splitting the verification in two phases is exactly to avoid the initial data-flow analysis.

7 Conclusions and Future Work

Existing JVML verifiers perform substantial data-flow analysis but do not preserve the results of this analysis for subsequent code generation and optimization phases. We have presented an alternative verifier that not only is faster than the standard Java verifier, but that additionally computes the Dominator Tree and brings the program into Static Single Assignment form. As a result, the respective computations need not be repeated in subsequent stages of the dynamic compilation pipeline. Since our algorithm has an overall lower cost than traditional Java bytecode verification, this essentially makes an SSA representation available "for free" to the virtual machine, reducing the cost for JIT compilation.

In the larger context of verifiable mobile code, our results indicate that verification should not be practiced in isolation "up front", but integrated with the rest of the client-side mobile code pipeline. Hence, we expect our approach to be applicable to other mobile-code systems besides the JVM, such as Microsoft's .NET platform [21].

Our work is also relevant for all existing JVM implementations which already use SSA internally for code optimization. If a VM already has means to translate code into SSA, having an "up front" data flow based verifier is simply redundant. We have shown that it is possible to delay type checking and to first transform the program into SSA. In fact, our algorithm is the first documented approach to safely translate Java code into SSA without any prior data-flow analysis and verification.

In the future, we plan to examine how subroutines could be supported in our framework. While subroutines are rapidly disappearing from JVML, they are still interesting from an academic perspective. They reinforce the question whether and how an SSA-based representation can be obtained for polymorphic code in which not all control-flow edges are explicit.

We are also interested in exploring *structural* SSA-annotation of JVML code. For this, JVML code is rearranged in such a way that a specific structure-aware SSA-based verifier can infer the final SSA-form of the code without actually calculating the Dominator Tree and Iterative Dominance Frontiers. As the code is still expressed

in pure JVML, it is fully backward compatible with existing VMs and does not require any additional annotations. While the rearranged code is likely to be less compact than its original form, this scheme will further reduce the required verification effort.

Acknowledgment

This research effort was partially funded by the National Science Foundation under grants TC-0209163 and ITR-0205712 and by the Office of Naval Research (ONR) under agreement N00014-01-1-0854. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the National Science foundation (NSF), the Office of Naval Research (ONR), or any other agency of the U.S. Government.

References

1. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, MA, 1986.
2. W. Amme, N. Dalton, J. von Ronne, and M. Franz. SafeTSA: A Type Safe and Referentially Secure Mobile-Code Representation Based on Static Single Assignment Form. In *Proceedings of the ACM SIGPLAN '01 Conference on Programming Language Design and Implementation*, pages 137–147, June 20–22, 2001. *SIGPLAN Notices,* 36(5), May 2001.
3. A. Appel. Foundational Proof-Carrying Code. In *Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science (LICS)*, pages 247–256. IEEE Computer Society Press, 2001.
4. D. Basin, S. Friedrich, J. Posegga, and H. Vogt. Java Byte Code Verification by Model Checking. In *11th International Conference on Computer-Aided Verification (CAV'99)*, number 1633 in Lecture Notes in Computer Science, pages 491–494, Trento, Italy, July 1999. Springer-Verlag.
5. A. Coglio. Improving the Official Specification of Java Bytecode Verification. In *Proceedings of 3rd ECOOP Workshop on Formal Techniques for Java Programs*, June 2001.
6. R. Fitzgerald, T. B. Knoblock, E. Ruf, B. Steensgaard, and D. Tarditi. Marmot: an optimizing compiler for Java. *Software—Practice and Experience*, 30(3):199–232, Mar. 2000.
7. S. N. Freund. The costs and benefits of java bytecode subroutines. In *Proceedings of the Formal Underpinnings of Java Workshop at OOPSLA*, oct 1998.
8. S. N. Freund and J. C. Mitchell. A Formal Specfication of the Java Bytecode Language and Bytecode Verifier. In *Proceeings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA'99)*, volume 34.10 of *ACM Sigplan Notices*, pages 147–166, N. Y., 1–5 1999. ACM Press.
9. S. N. Freund and J. C. Mitchell. Specification and verification of java bytecode subroutines and exceptions. Technical Report CS-TN-99-91, Standford University, 1999.
10. S. N. Freund and J. C. Mitchell. The Type System for Object Initialization in the Java Bytecode Language. *ACM Transactions on Programming Languages and Systems*, 21(6):1196–1250, 1999.

11. A. Gal, C. W. Probst, and M. Franz. Proofing: Efficient SSA-based Java Verification. Technical Report 04-10, University of California, Irvine, School of Information and Computer Science, April 2004.
12. M. Hagiya and A. Tozawa. On a new method for dataflow analysis of Java Virtual Machine Subroutines. In G. Levi, editor, *Static Analysis, 5th International Symposium, SAS'98*, pages 17–32. Springer-Verlag, Berlin Germany, 1998.
13. G. Klein. *Verified Java Bytecode Verification*. PhD thesis, Institut für Informatik, Technische Universität München, 2003.
14. G. Klein and T. Nipkow. Verified bytecode verifiers. *Theoretical Computer Science*, 298(3):583–626, April 2003.
15. G. Klein and M. Strecker. Verified Bytecode Verification and type-certifying Compilation. *Journal of Logic and Algebraic Programming*, 2003. To appear.
16. G. Klein and M. Wildmoser. Verified Bytecode Subroutines. *Journal of Automated Reasoning*, 30(3-4):363–398, 2003.
17. C. League, V. Trifonov, and Z. Shao. Functional Java Bytecode. In *Proc. 5th World Conf. on Systemics, Cybernetics, and Informatics*, July 2001. Workshop on Intermediate Representation Engineering for the Java Virtual Machine.
18. T. Lengauer and R. E. Tarjan. A fast algorithm for finding dominators in a flowgraph. *Transactions of Programming Languages and Systems (TOPLAS)*, 1:121–141, July 1979.
19. X. Leroy. Java Bytecode Verification: Algorithms and Formalizations. *Journal of Automated Reasoning*, 30(3/4):235–269, 2003.
20. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.
21. Microsoft Cooperation. Microsoft .NET. `http://www.microsoft.com/net/`, 2003.
22. G. C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, Paris, France, January 1997.
23. G. C. Necula and P. Lee. Safe Kernel Extensions Without Run-Time Checking. In USENIX, editor, *2nd Symposium on Operating Systems Design and Implementation (OSDI '96)*, pages 229–243, Berkeley, CA, USA, 1996. USENIX.
24. G. C. Necula and S. P. Rahul. Oracle-based Checking of Untrusted Software. *ACM SIGPLAN Notices*, 36(3):142–154, 2001.
25. R. O'Callahan. A Simple, Comprehensive Type System for Java Bytecode Subroutines. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pages 70–78, San Antonio, Texas, 1999.
26. C. Pusch. Proving the Soundness of a Java Bytecode Verifier Specification in Isabelle/HOL. *Lecture Notes in Computer Science*, 1579:89–103, 1999.
27. Z. Qian. A Formal Specification of Java Virtual Machine Instructions for Objects, Methods and Subrountines. In *Formal Syntax and Semantics of Java*, pages 271–312, 1999.
28. Z. Qian. Standard Fixpoint Iteration for Java Bytecode Verification. *ACM Transactions on Programming Languages and Systems*, 22(4):638–672, 2000.
29. R. Stärk and J. Schmid. Java Bytecode Verification is not possible (Extended Abstract). In R. Moreno-Díaz and A. Quesada-Arencibia, editors, *Formal Methods and Tools for Computer Science (Proceedings of Eurocast 2001)*, pages 232–234, Canary Islands, Spain, February 2001. Universidad de Las Palmas de Gran Canaria.
30. V. Sreedhar, G. Gao, and Y. Lee. DJ-Graphs and their Applications to Flowgraph Analyses. Technical Report ACAPS Memo 70, McGill University, May 1994.
31. V. C. Sreedhar and G. R. Gao. A Linear Time Algorithm for Placing ϕ-nodes. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 62–73, San Francisco, California, 1995.
32. Standard Performance Evaluation Corporation. SPEC JVM98 Benchmarks. `http://www.specbench.org/jvm98`, 2004.

33. R. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine: Definition, Verification, Validation*. Springer-Verlag, 2001.
34. R. F. Stärk and J. Schmid. Completeness of a Bytecode Verifier and a Certifying Java-to-JVM Compiler. *Journal of Automated Reasoning*, 30(3–4):323–361, 2003.
35. R. Stata and M. Abadi. A Type System for Java Bytecode Subroutines. *ACM Transactions on Programming Languages and Systems*, 21(1):90–137, 1999.
36. Sun Microsystems. *KVM - Kilobyte Virtual Machine White Paper*. `http://java.sun.com/products/kvm/wp/`. Palo Alto, CA, USA, 1999.
37. Sun Microsystems. JSR-000139 Connected Limited Device Configuration 1.1. `http://www.jcp.org/aboutJava/communityprocess/final/jsr139/`, 2004.
38. F. Yellin. Low level security in Java. In O'Reilly and Associates and Web Consortium (W3C), editors, *World Wide Web Journal: The Fourth International WWW Conference Proceedings*, pages 369–380. O'Reilly & Associates, Inc., 1995.