Available online at www.sciencedirect.com
SCIENCE DIRECT®
Electronic Notes in Theoretical Computer Science
ELSEVIER Electronic Notes in Theoretical Computer Science 131 (2005) 27–38
www.elsevier.com/locate/entcs

Integrated Java Bytecode Verification

Andreas Gal, Christian W. Probst, Michael Franz[1]

Donald Bren School of Information and Computer Science
University of California
Irvine, CA, 92697, USA

Abstract

Existing Java verifiers perform an iterative data-flow analysis to discover the unambiguous type of values stored on the stack or in registers. Our novel verification algorithm uses abstract interpretation to obtain definition/use information for each register and stack location in the program, which in turn is used to transform the program into Static Single Assignment form. In SSA, verification is reduced to simple type compatibility checking between the definition type of each SSA variable and the type of each of its uses. Inter-adjacent transitions of a value through stack and registers are no longer verified explicitly. This integrated approach is more efficient than traditional bytecode verification but still as safe as strict verification, as overall program correctness can be induced once the data flow from each definition to all associated uses is known to be type-safe.

Keywords: abstract interpretation, verification, optimization

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

[1] Email: {gal,probst,franz}@uci.edu
This research effort was partially funded by the National Science Foundation (NSF) 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, the Office of Naval Research, or any other agency of the U.S. Government.

doi:10.1016/j.entcs.2005.01.020

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. 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 give a brief overview of an alternative verification mechanism that avoids such duplication of work. Instead of verifying Java Virtual Machine Language (JVML) bytecode directly, we annotate it in such a way that the flow of values between instructions becomes explicit rather than going through the operand stack and then transform the annotated bytecode into Static Single Assignment (SSA) form [3].

Verifying programs in SSA significantly reduces the number of points in the program that have to be type-checked, because only producers and consumers of values are verified. Inter-adjacent transitions of a value through stack and registers are no longer verified explicitly. This integrated approach is more efficient than traditional bytecode verification but still as safe as strict verification, as overall program correctness can be induced once the data flow from each definition to all associated uses is known to be type-safe.

Our benchmarks indicate that the aggregate time required for transforming JVML into SSA and verifying the program in this 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.

The remainder of this paper is organized as follows: Section 2 gives a brief overview of the traditional Java bytecode verifier and introduces SSA-based verification. Section 3 compares the performance of our method to that of Sun's standard verifier. Section 4 discusses related work and Section 5 contains our conclusion and points to future work.

$$
\begin{aligned}
\mathit{instruction} &::= \mathit{core} \mid \mathit{dataflow} \\
\mathit{core} &::= \texttt{iconst\_}n \mid \texttt{lconst\_}l \mid \texttt{iadd} \mid \texttt{ladd} \mid \texttt{ifeq}\ L \mid \texttt{return} \\
\mathit{dataflow} &::= \texttt{pop} \mid \texttt{dup} \mid \texttt{dup\_2} \mid \texttt{istore\_}x \mid \texttt{iload\_}x \mid \texttt{lstore\_}x \mid \texttt{lload\_}x
\end{aligned}
$$

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

2 Verification in Static Single Assignment Form

This section introduces a subset of JVML, briefly describes traditional Java bytecode verification, and discusses the abstraction used in our approach as well as our novel verification method.

2.1 JVML$_S$

Figure 1 shows the grammar for JVML$_S$, a subset of JVML which we use here for illustration purposes. We split the instruction set in *core* instructions and *data-flow* instructions. Core instructions operate on values stored on the operand stack, while data-flow instructions 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 and in multiple locations at the same time. Data-flow instructions neither produce nor consume values, they merely transport values between stack locations and variables. [2]

2.2 Java Bytecode Verification

JVML instructions 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 [12,25] uses iterative data-flow analysis and an abstract interpreter for JVML instructions. Unlike JVM, the stack cells and local variables of the abstract interpreter store *types*, rather than

[2] Even though it consumes a value, the *pop* instruction is a data-flow instruction, since it merely manipulates the stack such that the topmost value can no longer be used.

values. From the perspective of the verifier, JVM instructions are operations that execute on types.

JVML verification works at the method level. With a co-inductive 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.

The central responsibility of the Java bytecode verifier is to check 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.

2.3 Abstractions

In JVML, there is no obvious link between the definition of a value and its uses. However, even if definition-use chains 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 we categorized the instructions of JVML$_S$. Only *core* instructions define and use values. *Data-flow* instructions merely facilitate the flow of values between core instructions. For Core instructions the expected types of any consumed operands and the types of any produced values are always known statically. In contrast, data-flow instructions 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 rationale of our approach is to replace the stack and local variables by a register file, and to redefine the dynamic semantics of instructions to actually work on these registers. This replacement allows us to transform the stack based code into SSA and to perform type checking only between the definitions of values and their actual uses. We abstract each instruction in a program to a tuple consisting of the depth of the stack before that instruction is executed, a mapping from stack cells and local variables to the instructions that define them, the set of stack cells and local variables the instruction reads and writes, as well as a map from stack cells to the values that reside in them.

The main contribution of these components is to allow the dynamic semantics to work on a register file and to enable the transformation of the code into SSA *before* verification.

2.4 Algorithm

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 us to eliminate all 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 core instructions only and is in 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*).

For a small example program, the result of the annotation step is shown in Figure 2. Each instruction is annotated with the current stack depth before the instruction is executed. Using these annotations and the dynamic semantics of JVML_S, instructions no longer depend on the stack to connect operands to their definitions. Values on the stack are labeled relative to their distance to the bottom of the stack. 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. This labeling permits to resolve stack references without actually maintaining a stack data structure. An `iconst` instruction annotated with $sd = 0$, for example, always writes its result to stack cell 0. In unannotated JVML the stack cells receiving the produced value would depend on the state of the dynamic stack at that point in the program.

Following the JVML machine model we split long integers into two halves. Thus, instructions operating on long integers push and pop pairs of values

PC	**Instruction**	**StackDepth**	**Stack**						**Vars**	
			0	1	2	3	4	5	0	1
1	`lconst_0`	0	L	L'						
2	`lconst_1`	2	L	L'	L	L'				
3	`iconst_1`	4	L	L'	L	L'	I			
4	`ifeq L`	5	L	L'	L	L'				
5	`dup_2`	4	L	L'	L	L'	L	L'		
6	`ladd`	6	L	L'	L	L'				
7	`L: ladd`	4	L	L'						
8	`lstore_0`	2							L	L'

Fig. 2. An example program, and the abstraction for stack and variable states. Each instruction is labeled with the stack depth prior to the execution of that particular instruction. `L` stands for `LONG`, `L'` for `LONG'`, and `I` for `INT`.

onto and from the operand stack. Correspondingly, for each definition of a long integer two values are defined, one for the bottom half (type `LONG`), and one for the top half (type `LONG'`).

After the annotation phase, our verification algorithm first computes the Iterative Dominance Frontier (IDF) [20] for all definitions of values, that is values written into stack cells or local variables. Each reachable instruction in the program is visited in dominator-tree order and all references of core instructions to stack cells and local variables are resolved to SSA-names. Data-flow instructions do neither produce nor consume any values and are eliminated through copy propagation.

After transformation into SSA and copy-propagation, we can perform the actual type-checking. 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), while regular core instructions always define a value with a distinct type. These can be matched to their respective uses in a single sweep over the program in linear time.

Type-checking is performed lazily in the sense that only the minimal number of instructions is checked to ensure overall type-safety while for dataflow instructions only the proper data flow is guaranteed. Considering only the dynamic semantics, the data flow verified is obviously equivalent to the data flow that would have resulted by interpreting the original JVML program. However, since data-flow instructions have been eliminated, some of the restrictions enforced by their static semantics do no longer apply. The following JVML program, for example, will be rejected by the Java verifier, but is valid in our SSA-based dialect:

```
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 (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 pushed 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.

Due to space limitations, we are unable to elaborate on how to verify exceptions, arrays, and object initialization and refer to our technical report [7] instead.

3 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 subroutines 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.

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 KVM [24] 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.

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. There currently is no established set of benchmarks to test the performance of verifiers. Benchmark suites such as SPECjvm [19] 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 a test set. Figure 3 list some characteristics of the used classes. All measurements were conducted on a Pentium4 2.53GHz CPU with 512MB of RAM, running under RedHat Linux 9.

Figure 4 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 the dominator relation and the dominance frontier, SSA-based verification is approximately 45% faster. The total

	# of	method size		stack depth		local variables	
	methods	ø	max	ø	max	ø	max
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. 3. Characteristics of the test set we used to compare the runtime of our SSA-based verifier with the runtime of the traditional verifier.

number of instructions that has to be type-checked in the case of SSA-based verification is roughly 38% less than for the traditional verifier. The only noteworthy exception is *java/math*, which actually requires slightly more instructions to be type-checked in SSA form. This is caused by our treatment of the `LONG` and `DOUBLE` types, which we split in two halves while the traditional verifier can treat them in a single step.

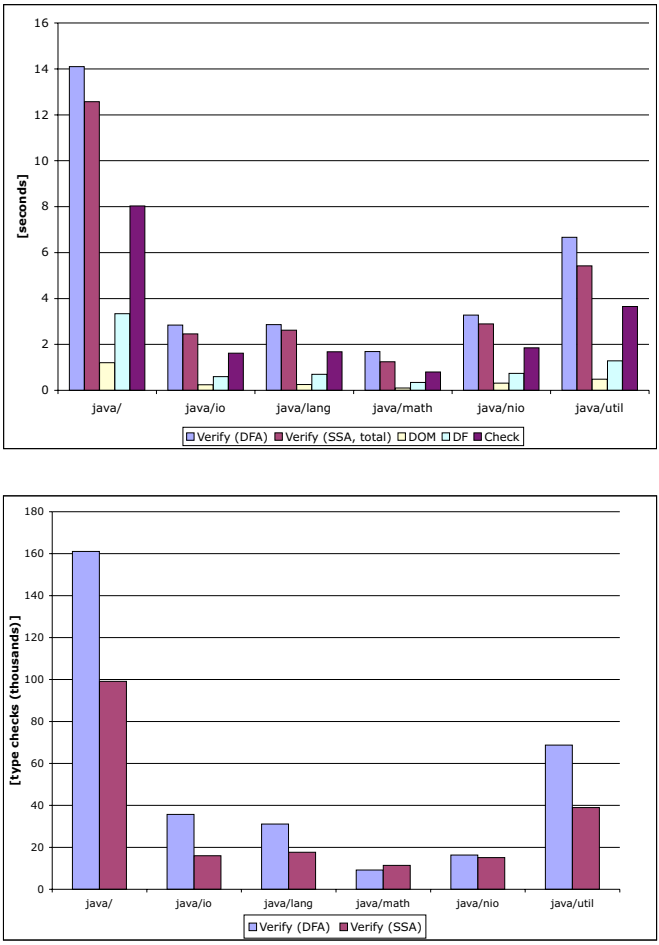

Fig. 4. Comparison of the total runtime and the number of instructions that have to be type-checked for the traditional DFA-based verifier and our SSA-based verifier.

4 Related Work

In addition to the informal description of the JVM [12], a number of formal specifications of the JVML and its verifier have been proposed [6,11,21]. In this context, subroutines are of particular interest and several type systems have been proposed for them [15,16,22]. All these approaches have in common that they rely on some form of iterative data-flow analysis [11,17] to decide type-safety.

Proof-carrying code (PCC) [14] addresses this problem by relieving the code consumer of the burden to verify the code. Instead, the code producer computes and proves a verification condition. The code consumer recomputes the verification condition and checks whether the attached proof is valid. PCC can even be used to prove safety properties of machine code. SSA-based verification, in contrast, is limited to mobile code formats such as Java, but has the advantage that it only requires the actual code as input, and no additional information such as proofs.

The split verifier approach [23], based on the idea of Lightweight Bytecode Verification [18], applies the PCC idea to Java bytecode. A *preverifier* 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. Just as in the case of Necula's PCC, the annotations enlarge the overall size of class files, while our approach does not rely on any additional annotation.

Similar to the split verifier, the verifier for Java smart cards [10] reduces the burden on the verifier through offline bytecode transformation. A preprocessor tool ensures that the Java stack is empty after every branch instruction and that all registers are mono-typed. In contrast to our approach, the Java smart card verifier fails for Java class files which have not been processed this way.

Inherently safe mobile code representation formats such as SafeTSA [1] 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 [4] 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 [5].

As Kelsey and Appel have observed [2,8], there is a close relation between SSA form and functional programming. Therefore, the work of League et al. [9] is directly related to our work. 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.

5 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 [13].

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.

References

[1] Amme, W., 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*, 2001, pp. 137–147, *SIGPLAN Notices,* 36(5), May 2001.

[2] Appel, A. W., *SSA is functional programming*, ACM SIGPLAN Notices **33** (1998), pp. 17–20.

[3] Cytron, R., J. Ferrante, B. K. Rosen, M. N. Wegman and F. K. Zadeck, *Efficiently Computing Static Single Assignment Form and the Control Dependence Graph*, ACM Transactions on Programming Languages and Systems **13** (1991), pp. 451–490.

[4] Fitzgerald, R., T. B. Knoblock, E. Ruf, B. Steensgaard and D. Tarditi, *Marmot: an optimizing compiler for Java*, Software—Practice and Experience **30** (2000), pp. 199–232.

[5] Freund, S. N., *The costs and benefits of java bytecode subroutines*, in: *Proceedings of the Formal Underpinnings of Java Workshop at OOPSLA*, 1998.

[6] Freund, S. N. 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)*, ACM Sigplan Notices **34.10** (1999), pp. 147–166.

[7] Gal, A., 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 (2004).

[8] Kelsey, R. A., *A correspondence between continuation passing style and static single assignment form*, ACM SIGPLAN Notices **30** (1995), pp. 13–22.

[9] League, C., V. Trifonov and Z. Shao, *Functional Java Bytecode*, in: *Proc. 5th World Conf. on Systemics, Cybernetics, and Informatics*, 2001, workshop on Intermediate Representation Engineering for the Java Virtual Machine.

[10] Leroy, X., *Bytecode verification on java smart cards*, Software Practice and Experience **32** (2002), pp. 319–340.

[11] Leroy, X., *Java Bytecode Verification: Algorithms and Formalizations*, Journal of Automated Reasoning **30** (2003), pp. 235–269.

[12] Lindholm, T. and F. Yellin, "The Java Virtual Machine Specification," Addison-Wesley, 1996.

[13] Microsoft Cooperation, *Microsoft .NET.* http://www.microsoft.com/net/.

[14] Necula, G. C., *Proof-Carrying Code*, in: *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, Paris, France, 1997, pp. 106–119.

[15] O'Callahan, R., *A Simple, Comprehensive Type System for Java Bytecode Subroutines*, in: *Proceedings of the 26th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, San Antonio, Texas, 1999, pp. 70–78.

[16] Qian, Z., *A Formal Specification of Java Virtual Machine Instructions for Objects, Methods and Subrountines*, in: *Formal Syntax and Semantics of Java*, 1999, pp. 271–312.

[17] Qian, Z., *Standard Fixpoint Iteration for Java Bytecode Verification*, ACM Transactions on Programming Languages and Systems **22** (2000), pp. 638–672.

[18] Rose, E. and K. H. Rose, *Lightweight Bytecode Verification*, in: *OOPSLA-Workshop on the Formal Underpinnings of the Java Paradigm*, 1998.

[19] SPEC, *JVM98 Benchmarks.* http://www.spec.org/jvm98 (2001).

[20] Sreedhar, V. C. 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)*, San Francisco, California, 1995, pp. 62–73.

[21] Stärk, R., J. Schmid and E. Börger, "Java and the Java Virtual Machine: Definition, Verification, Validation," Springer-Verlag, 2001.

[22] Stata, R. and M. Abadi, *A Type System for Java Bytecode Subroutines*, ACM Transactions on Programming Languages and Systems **21** (1999), pp. 90–137.

[23] Sun Microsystems, *JSR-000139 Connected Limited Device Configuration 1.1.* http://www.jcp.org/en/jsr/detail?id=139.

[24] Sun Microsystems, "KVM - Kilobyte Virtual Machine White Paper." Palo Alto, CA, USA, 1999.

[25] Yellin, F., *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* (1995), pp. 369–380.