Testing Java Virtual Machines

An Experience Report on Automatically Testing Java Virtual Machines

Emin Gün Sirer
Dept. of Computer Science and Engineering
University of Washington, Box 352350
Seattle, WA, 98195-2350
egs@cs.washington.edu
http://kimera.cs.washington.edu

Abstract

The Java Virtual Machine (JVM) has emerged as a ubiquitous platform for network computing. JVMs have been incorporated into browsers, web servers, database engines, and personal digital assistants, and there are plans under way to use them on embedded devices and smartcards. The adoption of JVMs into such diverse and security critical domains requires that their safety and security be tested rigorously. Unfortunately, Java virtual machines are large, complex, and have many subtleties, and consequently testing them is a difficult task. Commercial virtual machines deployed so far have had to rely on manual and ad hoc techniques for testing, and have subsequently exhibited substantial weaknesses that could lead to information theft or destruction.

In this paper, we describe our experience with automatically testing Java virtual machines. We outline two effective automatic testing techniques for JVMs. The first technique is comparative evaluation with mutations, where randomly perturbed test inputs are used to identify discrepancies between different versions of JVMs. We show that this fast and effective technique achieves broad code coverage, and discuss its shortcomings. Second, we present a well-structured technique for generating complex test cases from cogent grammar descriptions. We describe *lava*, a special purpose language we developed to specify production grammars for testing, and show that grammar-based test generation can produce very complex test cases from compact specifications. The testing process is easy to steer, and can generate targeted test cases. Most importantly, grammars enable the tester to reason about the expected system behavior on the test inputs. We show how grammars can be used in conjunction with inductive proofs to solve the oracle problem, and describe various application areas for grammar based testing. Finally, we report the results of applying these techniques to commercial Java virtual machine implementations.

1. Introduction

The Java virtual machine [Lindholm&Yellin 96] has emerged as a ubiquitous platform for network computing. The adoption of the Java virtual machine across the Internet, where 91% of all browsers now in use support a JVM[1] and where over 1.5 million web pages embody Java applets[2], shows that modern virtual machines fulfill an important role in heterogeneous networked environments. Indeed, Java virtual machines can now be found not only in browsers, but also in web servers, database engines, office applications, and personal digital assistants. In addition to the popularity of JVMs in such desktop and server environments, there are numerous plans underway to place JVMs in embedded

---

[1] Data is based on access logs from our web server over the April–December 1997 period.
[2] Measurement is based on a March 1998 Alta-Vista query.

devices, mobile telephones and even smartcards [Sun]. The adoption of JVMs in such security-critical domains, where flaws in JVM implementations could mean service disruption and even potential financial loss, requires that JVM functionality be tested rigorously. Unfortunately, the large size, complex functionality and subtle features of the Java virtual machine make testing it a difficult task. Consequently, commercial Java virtual machines deployed so far have had to rely on manual techniques for testing, and have suffered as a result. Manual testing techniques, such as ad hoc test construction and code auditing, are often slow, expensive, hard to manage and only effective against a limited class of flaws. Subsequently, commercial virtual machines released to date have exhibited substantial weaknesses that could lead to information theft or destruction.

Checking the correctness of a JVM is not a simple task. The most promising technique in checking program correctness is formal program verification, as it can provide theoretically strong guarantees that the code is true to a specification. However, the current state of the art in formal program verification does not sufficiently address the needs of virtual machines. Essentially, verifiers, compilers and runtimes are too complex, and their implementation often too ad hoc, to be served well by the current state of the art in program verification. While the theory community has made substantial progress on provably correct bytecode verifiers [Stata&Abadi 98,Freund&Mitchell 98], current formal verification techniques exhibit four serious shortcomings. Namely, they operate on abstract models of the code instead of the implementation, cover only a small subset of the constructs found in a JVM, apply only to verifiers and not to compilers, interpreters or runtimes, and require extensive human involvement to map the abstract proof onto the actual code. Subsequently, virtual machine developers have had to rely on manual verification and testing techniques to gain assurance in their implementations. Techniques such as independent auditing [Dean et al. 97] and directed test case generation have been used extensively by various commercial virtual machine vendors. However, manual testing techniques are expensive and slow due to the amount of human effort involved. Further, independent auditing requires access to source code and lacks a good coverage metric, and manual test case generation often requires massive amounts of human effort. Consequently, commercial virtual machine implementations have suffered from a steady stream of security flaws at a rate of one major flaw every two months [Kimera], which points to a need for more effective testing techniques.

In this paper, we describe our experience with testing Java virtual machines using automatic techniques. We faced the testing problem when we developed our own Java virtual machine. Lacking the resources to perform manual tests, we examined ways to increase our assurance in our implementation cheaply and effectively. This paper presents the two automatic testing techniques for JVMs that came out of this effort; namely, comparative evaluation, and grammar-based test generation. The rest of the paper defines these techniques and qualitatively illustrates the types of errors they uncovered in commercial Java systems. Our findings can be summarized as follows:

- *Comparative evaluation*, where multiple JVMs are tested on the same randomly mutated inputs, is both fast and effective at locating flaws. It achieves broad code coverage, even with a simple mutation scheme such as one-byte random perturbations. The drawbacks of comparative evaluation are that it requires another, preferably strong implementation of the same functionality for comparison, and that the testing effort is limited in its coverage by the amount of complexity in the original input. Our second technique addresses both of these problems.

- *Grammar-based test generation*, where a production grammar is used to generate test cases from cogent grammar descriptions, can effectively and easily produce complex test cases. The testing process is easy to steer, well structured, and portable. Most importantly, grammars enable the tester to formally reason about the expected behavior of the system on the test inputs, thereby addressing the oracle problem.

In the next section, we provide some background on the Java virtual machine, and enumerate the desired properties of automatic testing techniques. Section 3 describes comparison testing with mutations, illustrates the types of errors it uncovered, and examines why it was so effective. Section 4 provides the basics of grammar-based test generation, describes the *lava* language which enables probabilistic grammar specifications, and shows various different application areas for grammar-generated test cases. Section 4 concludes our experience with automatic testing techniques for JVMs. A performance analysis and detailed description of the *lava* language are beyond he scope of this paper. Interested readers are encouraged to look at [] for further deails.

2. Background

The Java virtual machine consists of three fundamental components; namely, the verifier, the compiler and/or interpreter, and the runtime. Since these components together form part of the trusted computing base, comprise the functionality to be tested and define the domain for our work, it is worth examining them in detail.

The verifier forms the main line of defense against malicious applications. It statically checks untrusted applications against a set of system safety axioms to ensure that the applications will not misbehave and cause faults at run time. The safety axioms range from simple structural rules about the class file format, e.g "references to constants shall occur through valid indices into a table of constants," to structural rules about the code, e.g. "code shall not jump outside of code bounds," to typesafety, e.g. "all pointer operations shall be type-safe," to link checks, e.g. "an imported field signature must match the expected type." A typical Java virtual machine will embody several hundred fine-grain, subtle and diverse security axioms for protection. While the safety of these axioms have not yet been formally proven, we believe that they are sufficient to ensure typesafety and to protect the assumptions made in the virtual machine from being violated at runtime [Drossopoulou et al. 97,Syme 97]. A failure of the system verifier to enforce these safety axioms would enable applications to violate system integrity, and may result in data theft or corruption.

An interpreter or just in time (JIT) compiler is responsible for the correct execution of JVM applications. The compiler or the interpreter has to correctly translate Java bytecodes into a sequence of native instructions that match the expected semantics of the application. Both a JIT compiler and an interpreter may, in turn, perform optimizations and code transformations to speed up execution, such as replacing sequences of generic and slow instructions with specialized, fast equivalents. These optimizations also need to preserve the intended semantics of the application.

Finally, the system libraries are responsible for providing run time services to applications during their execution. In this paper, we will concern ourselves primarily with the verifier and compiler/interpreter components of the JVM. The runtime forms a smaller part of the JVM in comparison, and since most of it is written in Java in commercial systems, it benefits directly from increased assurance in the other two components.

The task of the system tester, then, is to ensure that the bytecode verifier correctly accepts safe code and rejects unsafe code, and that the compiler/interpreter properly executes JVM applications. We faced this task when we implemented our own JVM, and turned to automatic testing techniques to help simplify the assurance process. We wanted our testing techniques to possess the following properties:

- *Automatic*: Verification should proceed without human involvement, and therefore be relatively cheap. The technique should be easy to incorporate into nightly regression testing.
- *Complete*: Verification should cover as much of the functionality of a JVM as possible. It should also admit a metric of progress that correlates with the amount of assurance in the virtual machine being tested.
- *Well structured*: Examining, directing, checkpointing and resuming testing efforts should be simple.
- *Efficient*: Testing should result in high assurance in a JVM within a reasonable amount of time.

We started our testing effort with a very simple automated technique that fulfills most, but not all of these properties, which we adopted from the hardware testing community. We describe this baseline technique in the next section, and address its weaknesses in Section 4.

3. Comparative Evaluation with Mutations

Comparison evaluation with mutations is a commonly used testing technique in hardware manufacturing to make sure that two chip implementations are identical, and to ensure that implementations are robust even in the face of malformed inputs. The core idea, illustrated specifically with respect to bytecode verification in Figure 1, is to direct the same mechanically produced test cases to two or more versions of a bytecode verifier, and to compare their outputs. A discrepancy indicates that at least one of the verifiers differs from the others, and typically requires human involvement to determine the cause of the discrepancy and its severity.

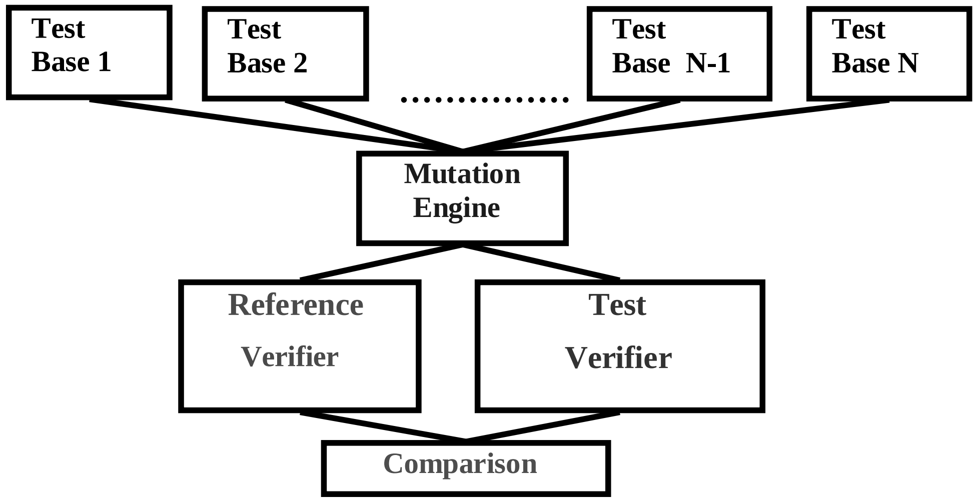

Figure 1. Comparative Evaluation with Mutations. A mutation engine injects errors into a set of test bases, which are fed to two different bytecode verifiers. A discrepancy indicates an error, a diversion from the specification, or an ambiguity.

In our case, we compared our bytecode verifier implementation to that of Sun JDK 1.0.2, Netscape 4.0 and the Microsoft Java virtual machine as found in Internet Explorer 4.0. We formed the test inputs by creating five valid test bases and introducing one-byte pseudo-random mutations. The bases were generated by hand using a bytecode assembler to exercise as many of the different features of Java bytecode as possible. We experimented with introducing multiple mutations into a single test base, but found that multiple mutations made post-mortem investigation of discrepancies more difficult, as benign mutations in strings made the program difficult to read. Therefore, each test case was generated by introducing only a single one-byte value change at a random offset in a base class file.

In most cases, all consulted bytecode verifiers agreed that the mutated test inputs were safe or unsafe, yielding no information about their relative integrity. In some cases, however, one of the bytecode verifiers would disagree from the rest. Cases where the class file was accepted by our bytecode verifier but rejected by a commercial one usually pointed to an error in our implementation. We would fix the bug and continue. Conversely, cases where the class file was rejected by our bytecode verifier but accepted by a commercial bytecode verifier usually indicated an error in the commercial implementation. Occasionally, the differences were attributable to ambiguities in the specification or to benign diversions from the specification.

Random mutations are quite effective at achieving broad coverage. At a throughput of 250K bytes per second on a 300 MHz Alpha workstation with 128K of memory, comparative evaluation with mutations exercised 75% of the checks in the bytecode verifier within an hour and 81% within a day. Since our test bases consisted of single classes, we did not expect to trigger any link-time checks, which accounted for 10% of the missing checks in the bytecode verifier. Further inspection revealed that another 7% were due to redundant checks in the bytecode verifier implementation.

We attribute the high rate of check coverage achieved by comparative testing to two factors. First, Java bytecode representation is very compact, such that small changes often drastically alter the semantics of a program. In particular, constant values, types, field, method and class names are specified through a single index into a constant pool. Consequently, a small change in the index can alter the meaning of the program in interesting ways, for instance, by substituting different types or values into an expression. For example, one of the first major security flaws we discovered in commercial JVM implementations stemmed from lack of type-checking on constant field initializers. A field of type String could be initialized to contain a **long** value, which was then taken to be the address of the string object. Consequently, a malicious program could read any part of the memory address space. A single mutation in an index field, within the first 10000 iterations (15 minutes) of testing, uncovered this loophole by which integers could be turned into object references.

A second reason that one-byte mutations achieve broad coverage is that the mutation granularity matches the granularity of fields in the class file format. Consequently, a single mutation affects only a single data structure at a time, and thereby avoids violating the internal consistency of a class file. For example, one mutation uncovered a case in commercial bytecode verifiers where the number of actual arguments in a method invocation could exceed the declared maximum. Had this field been packed, the chances of a mutation creating an interesting case would have been reduced. Similarly, a mutation in a jump offset uncovered a case where the MS JVM would incorrectly sign extend jump destinations of certain instructions, and thereby allow programs to jump to arbitrary memory locations. Packing of the offset would have made it difficult for one-byte mutations to uncover this flaw.

Although comparative evaluation with mutations achieves extensive breadth of coverage, it has some serious drawbacks. While the test cases cover a broad range of security axioms, they do not cover any deeply. For instance, while a chance mutation may affect control flow, it will not introduce excessively complicated control flow that may stretch the limits of the verifier implementation. The total complexity of the test cases is limited, therefore, by the original complexity of the test bases and the simplistic mutation strategy chosen to simplify debugging. In the next section we address this shortcoming by automatically generating complex test cases via a production grammar.

4. Grammar-based Test Production

In order to automatically generate test bases of high complexity that are targeted at interesting bytecode constructs, we rely on a production grammar. A production grammar is just like a regular parsing grammar (i.e. yacc), except it works the other way around. Starting from a non-terminal, it produces a valid program (all terminals). We use a generic code-generator-generator to parse a Java bytecode grammar and emit a specialized code-generator. The code generator in turn takes a seed as input and applies the grammar to it. Running the code-generator on a seed produces test cases in Java bytecode that can then be assembled and used for testing (see Figure 2).

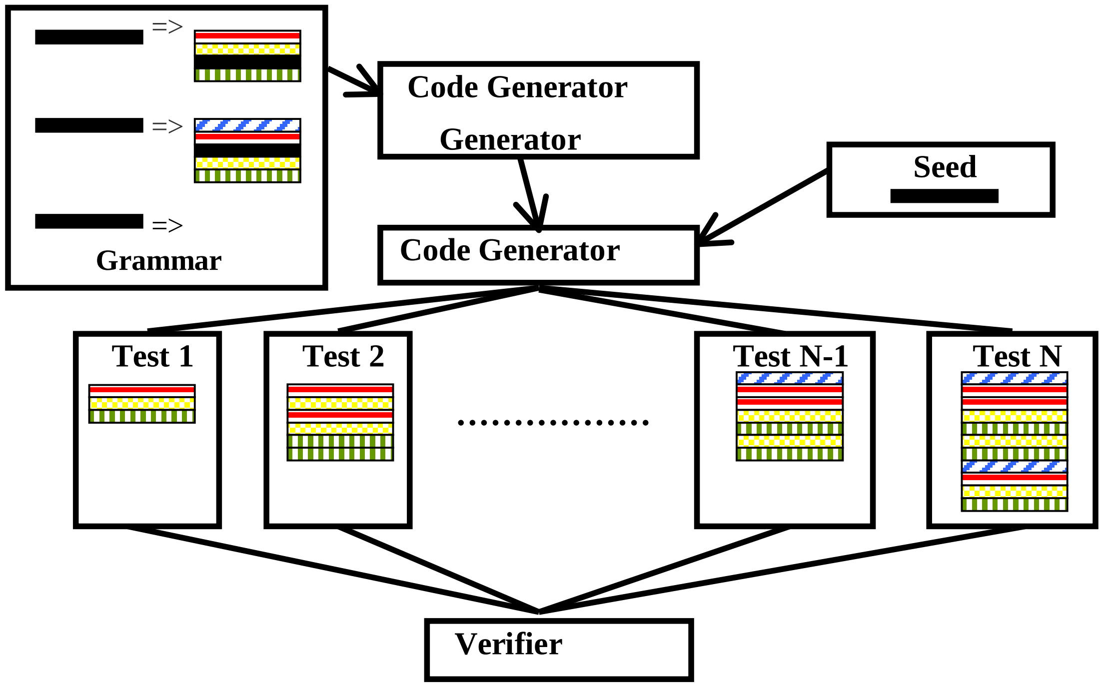

Figure 2. Grammar testing takes a production grammar and applies it to a given seed to automatically generate test cases of high complexity.

Our code-generator-generator reads in a conventional grammar description that resembles that of YACC. This grammar description is specified in a special-purpose language, called *lava*, designed specifically for test generation [Sirer&Bershad 99]. This context-free grammar consists of productions with a left-hand side containing a single non-terminal (LHS) that is matched against the seed input, and a right-hand side (RHS) that replaces the LHS if a match is found. An optional code fragment associated with the production is executed when the production is selected, thereby enabling the generation of context-sensitive programs. This feature is frequently used, for example, to generate unique labels, and also to keep track of auxiliary data structures related to the test case under production. Finally, *lava* also enables the test administrator specify the maximum number of times that a production can be used. This feature is useful for ensuring that the produced code does not violate restrictions in the Java class file format. For example, Java restricts method code bodies to 64K instructions. Specifying this limit in the grammar forces the test production to terminate when 64K instructions have been generated.

The code-generator-generator parses the *lava* grammar specification, and generates a code-generator. The code-generator in turn takes a seed input and a set of weights, and applies the grammar to it. The production process is probabilistic, and the weights help guide the relative use of different productions. That is, when it is possible to use more than one production specified in the grammar, the code generator picks a production randomly based on relative weights. This enables *lava* to generate many different instruction mixes using the same grammar without having to rewrite or rearrange the grammar. Each different random number seed will generate a different test case.

We have applied the grammar-based test cases generated by *lava* to testing JVMs in three separate ways. First, we used the complex test cases to look for gross errors and violations of the JVM specification. Since Java is meant to be a type-safe system, it should not, under any conditions, perform an illegal memory access or result in a system crash (even non type-safe systems should avoid illegal memory accesses, but typically the consequences are not as dire for closed, non-extensible systems. Since Java systems can run untrusted applications, they cannot make any assumptions that illegal memory accesses will not be exploited, for example, to probe memory and leak information). The very first test case that we generated using a production grammar exposed that Sun JDK1.0.2 on the Alpha platform will perform all kinds of illegal accesses when loading a class file with deep levels of subroutine nesting. A class file, generated by only three productions, but consisting of a few thousand Java instructions, uncovered this flaw within seconds.

The second application of complex, grammar generated test cases was in performance measurement. Since the grammar is parameterizable, it is quite straightforward to produce several different test cases, each containing roughly the same mix of instructions, and of similar complexity, only differing in length. These test cases uncovered that the performance of our verifier implementation was $O(N^2 \log N)$ in the number of basic blocks. This motivated us to find and fix the unnecessary use of ordered lists where unordered ones would have sufficed.

The third and final application of grammar generated test cases was in the testing of code transformation tools. Components such as Java compilers and binary rewriting engines [Sirer et al. 98] must retain program semantics through bytecode to native code or bytecode to bytecode transformations. Such components can be tested by submitting complex test cases to the transformations, and comparing the behavior of the transformed, e.g. compiled, program to the original. In our case, we wanted to ensure that our binary rewriter preserved the original program semantics whilst it reordered basic blocks. To test the rewriter, we created 17 test cases which averaged 1900 instructions in length and exercised every control flow instruction in Java. The grammar was constructed to create test cases with both forward and backward branches for each type of jump instruction, and instrumented to print a unique number in each basic block. We then simply executed the original test cases, captured their output and compared it to the output of the test cases after they had been transformed through the binary rewriter. This testing technique caught a sign extension error that incorrectly calculated backward jumps in **tableswitch** and **lookupswitch** statements, and took a total of two days from start to finish.

We found the overall process of generating test cases using a production grammar to be simple, easy to steer and easy to manage. Unlike traditional general-purpose scripting languages, which provide little more than string manipulation operations, *lava* provides detailed support for test generation and organization. The structure provided by productions resulted in modular specifications, which enabled easy mixing and matching of different test grammars. The compact representation also simplified code management.

The one drawback of the complex test cases generated by *lava* was that it was not possible to reason about the execution time properties of the generated test cases. Quite simply, though lava could generate very complex test cases, it was not clear, without tedious reverse engineering, what the system output ought to be on the automatically generated input.

```
insts 5000 INSTS => INST INSTS
emptyinst  INSTS => /* empty */
ifeqstmt
INST => iconst_0; INSTS; ifeq L$1;
        jmp L$2; L$1: INSTS; L$2:
jsrstmt
INST => jsr L$1; jmp L$2;
        L$1: astore $reg; INSTS;
        ret $reg; L$2:
action{ ++reg; }
```

```
Weight jsrstmt 10
Weight ifeqstmt 10
Weight insts 1
Weight emptyinst 0

.class public testapplet$NUMBER

.method test()V
  INSTS; return
.end method
```

Figure 3. A simplified *lava* grammar and a corresponding seed for its use. Non-terminals are in upper case and dollar signs indicate that the value of the variable should be substituted at that location. Overall, the grammar specification is concise and well-structured.

Luckily, the uniform structure of grammars enable some deductive reasoning about the test cases that are generated by a given grammar. The insight behind this work is that the test cases reflect the fundamental properties of the grammar they were generated from. Subsequently, it is possible to prove inductively that all tests generated by any given grammar will have the properties that are recursively preserved at every production. For instance, it is easy to see, and inductively prove, that the sample grammar given in Figure 3 will execute all labeled blocks it generates. Using a third production, we can instrument every labeled basic block produced by that grammar, and check to ensure that a virtual machine enters all of the blocks during execution. We simply print the block label when the block is entered, and, at the end of the test run, check to make sure that all blocks appear exactly once in the trace. Using this technique, for instance, we found that some of the control flow instructions in an early version of our interpreter had the sense of their comparisons reversed.

5. Related Work

There has been substantial recent work on using type-systems as a foundation for bytecode verifier implementations. Stata and Abadi have formally analyzed Java subroutines for type-safety and postulated a type-system that can be used as the basis of a Java verifier [Stata&Abadi 98]. Freund and Mitchell have further extended this model to cover object allocation, initialization and use [Freund&Mitchell 98]. While these approaches are promising, only two dozen of the two hundred operations permitted by the Java virtual machine have been covered by this type system-based model to date. Further, these approaches operate on abstract models of the code instead of the implementation, and require extensive human involvement to map the model onto a complicated body of code, such as a bytecode verifier. The strong guarantees of formal reasoning are enervated by the manual mapping of a model onto the actual implementation.

An approach to verification that is common in industry is to perform source code audits, where a group of programmers examine the code and manually check for weaknesses. The Secure Internet Programming group at Princeton has found a number of security flaws in Java implementations using this technique [Dean et al. 97]. While auditing can uncover flaws, it has a number of drawbacks. First, it is very labor intensive, and therefore expensive. Second, it requires access to source code, and is therefore not always applicable. Finally, it provides only a limited coverage metric, namely, number of lines that have been examined. Consequently, it is not clear whether scarcity of flaws being uncovered using this technique is attributable to the strength of the implementation or to the weaknesses of the auditor.

Production grammars have been used in conjunction with comparative evaluation to check compiler implementations for compatibility [McKeeman 98]. This approach resembles our grammar-based technique without the inductive proofs, and suffers from the same problem of requiring a second, preferably stronger implementation of the same functionality. Production grammars have also been used to grow realistic looking artifacts in computer graphics [Prusinkiewicz et al. 88]. Grammars are a special case of rewrite systems, described in [Dershowitz+ 90,Dershowitz 93].

6. Conclusion

In this paper, we have presented three techniques for automatically testing the correctness of virtual machines and discussed their application to commercial Java VM implementations. We have shown that a simple technique, comparative evaluation with mutations, can be quite effective at locating flaws. More sophisticated techniques such as production grammars can generate complex test cases from cogent, well-structured, easy to maintain specifications. Further, production grammars enable the test administrator to not only generate interesting test cases but also reason about their correctness and behavior. Overall, these techniques enable the verification of Java virtual machines cheaply and effectively.

Systems get progressively safer over time only if there is a systematic, concerted and well-guided testing effort to make sure that flaws are not introduced, and that existing flaws are located and fixed. We hope that the automated testing presented here will find themselves a nice in testing large, complex, and fragile virtual machine systems.

Acknowledgements

We would like to thank Jim Roskind of Netscape, Charles Fitzgerald and Kory Srock of Microsoft, and Tim Lindholm, Sheng Liang, Marianne Mueller and Li Gong of Javasoft for working with us to address the Java security issues raised by this work. We would also like to thank Arthur J. Gregory and Sean McDirmid for implementing parts of our Java virtual machine.

References

[Dean et al. 97] Dean, D., Felten, E. W., Wallach, D. S. and Belfanz, D. "Java Security: Web Browers and Beyond." In Internet Beseiged: Countering Cyberspace Scofflaws, D. E. Denning and P. J. Denning, eds. ACM Press, October 1997.

[Dershowitz+ 90] Dershowitz, N. and Jouannaud, J. Rewrite Systems. In Handbook of Theoretical Computer Science.Volume B: Formal Methods and Semantics. Van Leeuwen, ed. Amsterdam 90.

[Dershowitz 93] Dershowitz, N. A Taste of Rewrite Systems. In Lecture Notes in Computer Science 693, 199-228 (1993).

[Drossopoulou et al. 97] Drossopoulou, S., Eisenbach, S. and Khurshid, S. "Is the Java Typesystem Sound ?" Submitted for publication, October 1997.

[Freund&Mitchell 98] Freund, S. N. and Mitchell, J. C. "A type system for object initialization in the Java Bytecode Language." In *ACM Conference on Object-Oriented Programming: Systems, Languages and Applications*, 1998.

[Kimera] Sirer, E. G., Gregory, A. J., McDirmid, S. and Bershad, B. The Kimera Project. http://kimera.cs.washington .edu/

[Lindholm&Yellin 96] Lindholm, T. and Yellin, F. The Java Virtual Machine Specification. Addison-Wesley, 1996.

[McKeeman 98] McKeeman, B. "Differential Testing for Software" Digital Technical Journal Vol 10, No 1, December 1998.

[Prusinkiewicz et al. 88] Prusinkiewicz, P., Lindenmayer, A., Hanan, J. "Developmental Models of Herbaceous Plants for Computer Imagery Purposes." In Computer Graphics, volume 22, number 4, August 1988.

[Sirer&Bershad 99] Sirer, E.G., and Bershad, B.N. "Using Production Grammars in Software Testing." To appear in the Symposium on Domain Specific Languages, Austin, Texas, October 1999.

[Sirer et al. 98] Sirer, E. G., Grimm, R., Bershad, B. N., Gregory, A. J. and McDirmid, S. "Distributed Virtual Machines: A System Architecture for Network Computing." European SIGOPS, September 1998.

[Stata&Abadi 98] Stata, R. and Abadi, M. "A type system for Java bytecode subroutines." In Proceedings of the 25th ACM POPL, January 1998, p. 149--160.

[Sun] Sun Microsystems, Inc. Palo Alto, California. http://java.sun.com/products/OV_embeddedProduct.html

[Syme 97] Syme, D. "Proving Java type-soundness." University of Cambrdige Computer Laboratory Technical Report 427, June 1997.