ICS

TECHNICAL REPORT

Proofing: An Efficient and Safe Alternative to Mobile-Code Verification

Andreas Gal *Christian W. Probst* *Michael Franz*

Technical Report 03-24
School of Information and Computer Science
University of California, Irvine, CA 92697-3425

November 17, 2003

Abstract

The safety of the Java Virtual Machine is founded on bytecode verification. Although verification complexity appears to roughly correlate with program size in the average case, its worst-case behavior is quadratic. This can be exploited for denial-of-service attacks using relatively short programs (applets or agents) specifically crafted to keep the receiving virtual machine's verifier busy for an inordinate amount of time. Instead of the existing, quadratic-complexity verification algorithm, which needs to decide the validity of any given bytecode program, we present a linear-complexity alternative that merely ensures that no unsafe program is ever passed on to the virtual machine. Hence, in certain cases, our algorithm will modify an unsafe bytecode program to make it safe, a process that we call 'proofing". Proofing does not change the semantics of programs that would have passed the original bytecode verifier. For programs that would have failed verification, our algorithm will, in linear time, either reject them, or transform them into programs (of unspecified semantics) that are guaranteed to be safe. Our method also solves a long-standing problem, in which for certain perfectly legal Java source programs the bytecodes produced by Java compilers are erroneously rejected by existing verifiers.

Information and Computer Science
University of California, Irvine

Proofing: An Efficient and Safe Alternative to Mobile-Code Verification

Andreas Gal, Christian W. Probst, and Michael Franz
Department of Computer Science
University of California, Irvine
Irvine, CA, 92697
{gal,cprobst,franz}@uci.edu

ABSTRACT

The safety of the Java Virtual Machine is founded on bytecode verification. Although verification complexity appears to roughly correlate with program size in the average case, its worst-case behavior is quadratic. This can be exploited for denial-of-service attacks using relatively short programs (applets or agents) specifically crafted to keep the receiving virtual machine's verifier busy for an inordinate amount of time.

Instead of the existing, quadratic-complexity verification algorithm, which needs to decide the validity of any given bytecode program, we present a linear-complexity alternative that merely ensures that no unsafe program is ever passed on to the virtual machine. Hence, in certain cases, our algorithm will modify an unsafe bytecode program to make it safe, a process that we call "proofing".

Proofing does not change the semantics of programs that would have passed the original bytecode verifier. For programs that would have failed verification, our algorithm will, in linear time, either reject them, or transform them into programs (of unspecified semantics) that are guaranteed to be safe.

Our method also solves a long-standing problem, in which for certain perfectly legal Java source programs the bytecodes produced by Java compilers are erroneously rejected by existing verifiers.

1. INTRODUCTION

Mobile programs can be malicious. To protect itself, 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. 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.

Unfortunately, the verification algorithm employed by the JVM has a worst-case execution complexity that increases quadratically with method length. This can be exploited in a denial of service attack on the computer hosting the JVM. It is possible to systematically craft bytecode programs (applets or agents) that exhibit worst-case verification complexity. A JVM that receives such a program is effectively shut down while verification is underway.

We have demonstrated this attack in previous work [11]. Using the standard JVM verification algorithm, relatively short programs that were specifically designed for difficulty of verification but only a few thousand bytes in length, led to verification efforts in the order of hours on workstation-class machines. Therefore, it would appear advisable to incorporate some time-out mechanism into bytecode verifiers, although no current JVM verifier to our knowledge implements such a scheme.

We present an alternative to Java's verification mechanism, which we call *bytecode proofing*, that has linear, rather than quadratic complexity. The unique feature of bytecode proofing is that it does not attempt to decide whether an arbitrary bytecode program is actually safe. Instead, it guarantees that (a) all code forwarded to the virtual machine for execution is safe, and (b) the semantics of this code are identical with those of the original program in all cases in which the traditional verifier would have returned a verdict of "safe". In cases in which the standard verifier would have rejected the original bytecode, but ours does not, the semantics of the resulting program after transformation by our algorithm are unspecified, but *safe* with respect to the original verification criteria.

Our approach also solves an old problem with existing bytecode verifiers that was first reported by Stärk et al. [28]: there exists a class of legal Java source code programs that, when compiled into JVML, cannot be verified using traditional verifiers and are erroneously rejected. Our bytecode proofing algorithm does not run into these problems and will accept these programs, as it should.

The rest of this paper is organized as follows: Section 2 briefly introduces the traditional Java bytecode verification algorithm. Section 3 explains how certain properties of this algorithm can be exploited, resulting in a denial of service attack. Section 4 introduces our approach of *proofing* mobile code, and Section 5 shows how this enables efficient verification. Section 6 describes our preliminary implementation of a bytecode proofer and compares its performance with the traditional Java verifier, while Section 7 discusses related work. In Section 8 we draw conclusions, and outline future work in Section 9.

$\texttt{Types} = \{\texttt{I}, \texttt{A}, \ldots, \top\} \cup \texttt{Classes}$
$\mathit{LocVar} : \mathbb{N} \rightarrow \texttt{Types}, \mathit{Stack} = (\texttt{Types} \cup \{\texttt{Err}\})^{n \geq 0}$
$\mathit{push} : \texttt{Types} \times \mathit{Stack} \rightarrow \mathit{Stack}, \mathit{pop} : \mathit{Stack} \rightarrow \mathit{Stack}$
$\mathit{State} = \mathit{Stack} \times \mathit{LocVar}, \phi : \mathit{State} \times \texttt{Instr} \rightarrow \mathit{State}$

$\phi((S, L), \texttt{iconst}\ n) = (\mathit{push}(\texttt{I}, S), L)$
$\phi((S, L), \texttt{aconst\_null}) = (\mathit{push}(\texttt{A}, S), L)$
$\phi(((\texttt{I}, \texttt{I}, S), L), \texttt{iadd}) = (\mathit{push}(\texttt{I}, S), L)$
$\phi(((\texttt{I}, S), L), \texttt{istore}\ n) = (S, L[n \leftarrow \texttt{I}])$
$\phi((S, L), \texttt{iload}\ n) = (\mathit{push}(L(n), S), L)$, if $L(n) = \texttt{I}$
$\phi(((\tau, S), L), \texttt{astore}\ n) = (S, L[n \leftarrow \tau])$,
if $L(n) \in \texttt{Classes}$
$\phi((S, L), \texttt{aload}\ n) = (\mathit{push}(L(n), S), L)$,
if $L(n) \in \texttt{Classes}$
$\phi(((\alpha_n, \ldots, \alpha_1, S), L), \texttt{invokestatic}\ C.m.sig) =$
$(\mathit{push}(\beta, S), L)$
if $sig = \beta(\beta_1, \ldots, \beta_n)$ and α_i is subtype of β_n

Figure 1: Internal state and selected rules for the type-level abstract interpreter.

2. JAVA VERIFICATION

Java bytecode verification has to ensure that (a) input programs are *well-formed* in that the control flow is properly contained within methods and (b) that the data-flow is *well-typed*. While well-formedness can be decided without much effort, deciding whether a program is type-safe is complicated by the JVM architecture.

JVM instructions can read and store intermediate values in two locations, the stack and virtual registers. These locations are dynamically typed in that the same stack location or virtual register can hold values of different types at different times. Verification aims to ensure that these locations are used consistently and that intermediate values are always read back with the same types that they were originally written as.

The basic ingredient of every bytecode verifier is an abstract interpreter for Java Virtual Machine Language (JVML) instructions. Unlike JVM, its stacks and virtual registers store *types*, rather than *values*. Thus, the interpreter translates instructions into operations that execute on types.

Figure 1 shows the definitions for the internal state as well as selected rules of such an interpreter. *push* and *pop* have the usual definition on stacks; a stack overflow or underflow generates the `Err` state. Note that exceptions do not add to the behavior of the abstract interpreter and are hence ignored in this step.

The rules describe the preconditions for the stack and the register component of the internal state. If there is no applicable definition for ϕ, an error occurs. It is noteworthy that the interpretation of method calls such as `invokestatic` does not actually call the method. Instead, it assumes that the method's effect is to push an object of type β on the stack as described by the method's signature.

The abstract interpreter described above is implemented by the Java verifier [18, 19, 35] through a data flow analysis. The verifier iterates for each method over all reachable instructions. Before the first instruction of a method is analyzed, the internal state is set to $(S, L) \in \texttt{Types} \times \mathit{LocVar}$. S is initialized to be empty, and the local variable mapping function L is set to the types of the parameters as described by the method's signature. Using the terminology in Figure 1, this results in $L(i) = \beta_i$ for a method signature $sig = \beta(\beta_1, \ldots, \beta_n)$. All other local variables start undefined, that is, they must not be used. Additionally, a *changed* bit is associated with every instruction, but initially is set only for the first instruction. This bit signifies that an instruction's input states have changed and that the instruction must be verified again on the algorithm's next iteration.

Then, the verifier loops over all instructions i in syntactical order. If $\mathit{changed}(i)$ is `true`it verifies i using the rules in Figure 1. This ensures, that the stack always has enough and correctly typed operands, that the stack will not overflow or underflow, and that local variables are initialized before their use and correctly typed for instruction i. Verification fails at this step if any of these requirements is not met.

After verifying an instruction i, the verifier identifies all instructions j that succeed i, either *directly* or as branch targets or exception handlers. For every every possible successor j, the state Out_i of the stack and the local variables after execution of i are merged into the state In_j before j. For exception handlers, an object of the type of the exception is pushed onto the stack component S.

If j has not been visited before, its incoming state In_j is set to be Out_i and $\mathit{changed}(j)$ is set. If j has been visited before, the verifier computes $\epsilon = \mathit{merge}(\mathit{Out}_i, \mathit{In}_j)$ and, if $\epsilon \neq \mathit{In}_j$, sets $\mathit{changed}(j)$ and $\mathit{In}_j = \epsilon$. Two stack states can be merged if the stack depth is equal for both of them. Additionally, the types of corresponding stack cells must either be equal, or are abstracted by the common supertype of the two types. In contrast, two *local* variable states can always be merged: if the types of corresponding variable abstractions differ, then the abstraction is either the common supertype in the case of references, or the variable is marked as unusable, that is, may no longer be read.

After having finished one iteration, the verifier checks whether there exists an instruction i for which $\mathit{changed}(i)$ is `true`. If so, it starts another iteration. Since instructions are visited in syntactical order, in the case of backward jumps the algorithm will never visit the target instruction again in the same iteration, but only in the next one.

Interestingly enough, the original verification algorithm for Sun's first Java Virtual Machine [19, 35] and its implementation have been adopted with very few modifications by many current JVMs. Even though the algorithm has a problematic worst-case behavior, it apparently performs quite well for the normal applet- or agent-type programs that it was designed for. The accepted wisdom seems to be that verification (a) contributes only marginally to the startup time of JVM bytecode programs and that it (b) will scale in some acceptable fashion for larger programs.

As we will show below, this accepted wisdom is wrong. Rather than the *average-case* behavior, it is the *worst-case* behavior that needs to be studied in any security-relevant context. We have successfully exploited the worst-case behavior of the JVM bytecode verification algorithm in what amounts to a denial-of-service attack on the machine hosting the JVM [11].

3. AN EXPLOIT OF THE JVM VERIFICATION ALGORITHM

In this section we will explore the shortcomings of the traditional JVM verifier in more detail and discuss an example program that

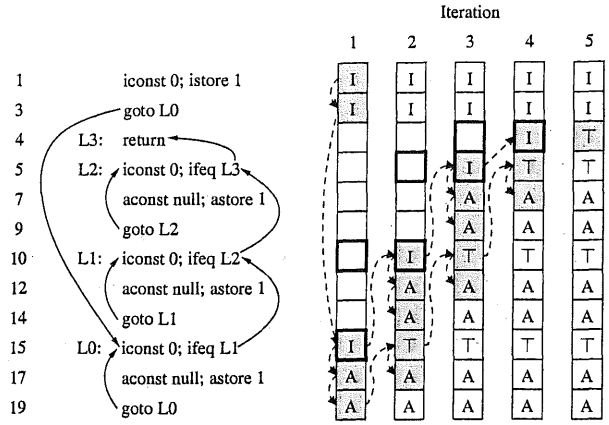

Figure 2: Java bytecode program that takes n iterations to be verified using the standard DFA verifier approach. The entry state for each basic block depends on the successor basic block. To the right of the figure, the local variable states are displayed for each iteration of the DFA. The shaded boxes indicate values that have changed, the framed boxes indicate instructions that will be visited in the next iteration.

was designed to expose the quadratic worst-case runtime behavior of the JVM verifier. While this program is perfectly well-typed and will eventually verify, it takes the JVM verifier an excessive amount of time to do so.

The right hand part gives the computed abstraction for virtual register 1 ($V1$) for several stages during verification. At the entry point of the method, an integer constant is loaded into $V1$ and the control is transfered to instruction 15. The verifier will actually not follow the branch instruction to the target, but continue to check instructions in sequence. However, it sets the bit *changed*(15). Therefore, once reaching instruction 15, it is interpreted. Since the first two instructions after $L0$ constitute a conditional branch, $V1$ is unchanged and the state after instruction 3 is propagated. While the branch is actually statically predictable in this example, the verifier does not consider the `ifeq` instruction as conditional. Instead, *changed*(10) is set, and the branch target will thus be interpreted in the next iteration. The `aconst_null` instruction 17 loads a `null` value onto the stack, which is then stored in $V1$. Thus, before instruction 19, $V1$ holds a value of type address (A). The unconditional `goto` at the end of this block transfers control back to $L0$, resulting in *changed*(15) being set.

During the next iteration, the verifier will inspect instructions 10 and 15, since their input states have changed. As the verifier hits instruction 15 again, it invalidates the previous assumption $L(1) =$ I, because it now knows that $L0$ has two predecessing states. The one at instruction 3 has $L(1) =$ I, the one at instruction 19 has $L(1) =$ A. Thus, the verifier updates $L(1)$ to $\top$, indicating that the value is not accessible at this point because its type depends on which path was taken to get to $L0$.

This discovery also affects the $L1$ basic block, which was previously assumed to be entered with $L(1) =$ I only. Now it is known to be entered with $L(1) = \top$. The verifier has thus to iterate over the code again to correct the wrong assumptions previously made for instruction 10. This process is repeated until all basic blocks are verified and a fixed point has been reached.

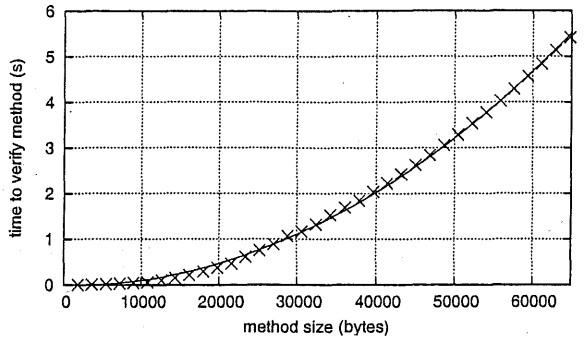

Figure 3: Verification time for a single method constructed to exhibit worst-case data-flow behavior. The x-axis indicates the the size of the method in bytes, which is proportional to the number of basic blocks n used to construct the method.

The number of basic blocks arranged in this fashion determines how often the verifier has to iterate over the code. For n basic blocks the verifier will have to iterate at least n times over the code, because n is the length of the longest path along which information has to flow backwards.

Figure 3 shows our measurement for the verification time of a single method containing bytecode with increasing maximum data-flow path length n. This time includes only the time it takes the verifier to prove safety.[1] As can be seen, the verification time increases quadratically with the code size.

To achieve the runtime necessary for a denial-of-service attack, several identical methods can be combined into a single classfile, multiplying the time required for verification. For example, a classfile with 100 methods (each 64kb bytecode) modeled after Figure 2 takes well over 600 seconds to verify on a 2.54GHz Pentium 4 with JVM 1.4.2. While the resulting classfile is fairly large with 6.5MB, we have shown in [11] that it can be compressed very efficiently using the standard Java archive format. At the bytecode level, each method consists of repetitions of an identical bytecode pattern. A compressed archive of this classfile is only 16kb in size.

4. PROOFING

The rationale of embedding a verifier into the JVM was *to ensure that no unsafe program is ever executed by the virtual machine.* The verifier performs a data-flow analysis to *decide* whether the input program is well-typed or not. As we have explained in the previous section, the data-flow algorithm involved in this decision has quadratic worst-case performance. On closer inspection, the JVM verifier approach is actually over-fulfilling its charter.

To guard the integrity of the virtual machine, the knowledge whether the input program was well-typed or not is actually irrelevant. All that matters is that no unsafe (not well-typed or not well-formed) program code is ever allowed to execute on the virtual machine. As we will show below, this can be guaranteed without actually deciding these properties of the input program and without paying the price for this decision in terms of quadratic worst-case runtime behavior.

[1]The Benchmark was performed on a 2.54GHz Pentium 4 machine with Sun Microsystem's JVM 1.4.2 for Linux. Similar results can be observed with other JVM implementations.

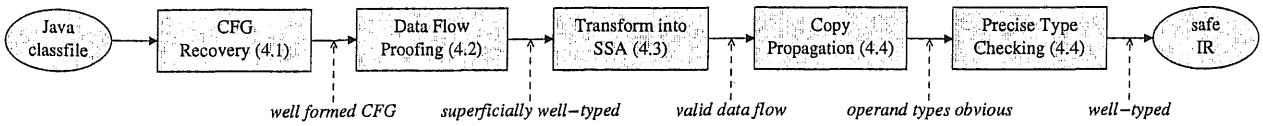

Figure 4: Proofing a Java program by first recovering the control flow graph, then proofing the data-flow based on preliminary type-checking using a superficial type system, and then transforming into SSA. After copy propagation to eliminate any dispensable register move instructions, the CFG is type checked using the full Java type system. While each intermediate step fails for a small number of easily detectable errors in Java programs, the majority of data-flow problems that traditional have to be *decided*, are instead *corrected*. The virtual machine is shielded from programs that aren't well-typed, but might execute a proofed program for which the original version would have been rejected by a verifier.

We propose a novel approach that we call bytecode *proofing*. It shields the virtual machine from not well-formed or well-typed program code without actually proving these properties for the input program. Instead, through a series of transformation steps, the input program P is transformed into a program P' that is guaranteed to be both well-formed and well-typed. Figure 4 shows the individual steps of this transformation. First, the control flow graph is recovered from the flat bytecode representation and subroutines are inlined into their call sites. The second phase uses a superficial type system to perform a preliminary type-checking and stack merging. This transforms the stack-based bytecode into a register-based intermediate representation (IR). Then, the IR is brought into Static Single Assignment (SSA) form to simplify successive analyses. Finally, a precise type-checking is performed on the IR using the full Java type system. The resulting control flow graph is by construction well-formed and well-typed and can be transformed either into a format suitable for interpretation or used to generate native machine code (JIT compilation). For the latter, the IR is conveniently already in SSA form, which greatly simplifies many optimization techniques. In the following sections, we will describe each step of the bytecode proofing process in more detail.

4.1 Control Flow Graph Recovery

The first step towards bytecode proofing is to recover the control flow graph (CFG) from the flat Java bytecode sequence. In Java, this process is significantly complicated by the existence of *subroutines* in the Java Virtual Machine Language (JVML). Subroutines can be best described as small methods within a method, but they completely execute within the scope of the actual method and do not take any arguments. When a subroutine is invoked with `jsr`, the value of the program counter of the virtual machine is pushed onto the stack. The subroutine is responsible for popping this return address from the stack and placing it in a virtual register. To return to the caller, is uses the `ret` instruction with the index of the virtual register holding the return address. However, it is also legal for a subroutine to directly return to a specific location in the method using conditional or unconditional branches. Subroutines are intended to prevent code duplication when compiling `finally`-clauses in Java, because code inside a `finally`-clause has to be executed before each branch leaving the guarded block associated with that clause. To reduce overall code size, the code within the `finally`-clause is represented at the bytecode level as a subroutine and before each exit point from the guarded block a `jsr` instruction is used to call the subroutine.

The algorithm we use for the recovery of the CFG consists of three steps: (1) discovering basic block boundaries, (2) compiling the method body, including exception handlers, but excluding any subroutines, and (3) processing all subroutines and their invocations by inlining the subroutine code into the method body. Our algorithm

FIND-BRANCH-TARGETS(M, BT)
1 **for each** $i \in M$
2 **do** $BT[i] \leftarrow false$
3 $BT[\text{ENTRY}(M)] \leftarrow true$
4 **for each** $i \in M$
5 **do for each** $t \in$ BRANCH-TARGETS(i)
6 **do** $BT[t] \leftarrow true$
7 **for each** $e \in$ EXCEPTION-HANDLERS(M)
8 **do** $BT[e] \leftarrow true$

Figure 5: Determine basic block boundaries for method M by identifying all potential branch targets, including exception handler entry points, and recording them in *BT*. The method entry point ENTRY(M) has to be marked separately, as it is not necessarily target of any instruction of the method, but still represents the beginning of a basic block.

uses the following data structures during construction of the CFG:

***BT*: array of booleans** For each instruction, this array indicates whether it is the target of a branch instruction.
***MAP*: array of pointers** This array maps branch targets to their corresponding basic blocks. Not yet compiled basic blocks are indicated with a *nil* value.
***SR*: stack of pointers** Subroutine invocations sites (`jsr` instructions) are collected using this stack.
***RB*: pointer** This variable points to the basic block immediately following the `jsr` instruction that is associated with the subroutine currently being processed, or *nil* otherwise.
***RI*: integer** Index of the virtual register containing the return address of the subroutine currently being processed, or 0 otherwise.

To detect basic block boundaries we use the algorithm shown in Figure 5. It performs a simple scan of all instructions i of a method M and records in BT any instruction that is the target of a branch instruction and thus represents the start of a new basic block. This includes instructions at the entry point of exception handlers. Because the entry point of a method M is not necessarily explicitly targeted by any instruction in M, but still represents the beginning of a basic block, it is flagged automatically. Since every instruction i is visited at most once, this algorithm has a worst-case runtime complexity of $O(n)$, with n being the number of instructions in M.

Knowing the basic block boundaries, we can start to reconstruct the CFG. As shown in Figure 6, the entry basic block of M is built using BUILD-BASIC-BLOCK, which in turn will recursively build all basic blocks of M that are reachable from the entry point.

```
BUILD-CONTROL-FLOW-GRAPH(M)
1  RB ← NIL
2  for each i ∈ M
3  do MAP[i] ← NIL
4  BUILD-BASIC-BLOCK(ENTRY(M))
5  PROCESS-SUBROUTINES(SR)
```

Figure 6: Recover the control flow graph from the bytecode of a method M. First, BUILD-BASIC-BLOCK is invoked to recursively build the CFG, including exception handlers. As a second step, all subroutines are inlined into their call sites by PROCESS-SUBROUTINES.

The actual work of translating the bytecode stream into the basic block format is performed in BUILD-BASIC-BLOCK. Successfully processed basic blocks are recorded in *MAP* to ensure that every basic block is translated at most once and thus every instruction is also visited at most once. This yields a complexity of $O(n)$ for constructing the CFG of a method body with n instructions.

During this phase of the CFG construction, any `jsr` instruction encountered by BUILD-BASIC-BLOCK is only recorded in *SR*, and its branch target (subroutine code) is not immediately processed. Instead, we temporarily treat the `jsr` to be a simple branch instruction to the immediately following instruction. Array *RB* is used to indicate the target block for `ret` statements if a subroutine is compiled. Until *RB* is populated in a later phase with a valid subroutine return vector, encountering `ret` instructions causes the construction to fail (Figure 7, Line 22). Besides all code reachable from the entry point of M, all local exception handlers are processed as soon as an instruction is encountered that might raise them. Like regular basic blocks, each basic block of an exception handler is recorded in *MAP* and is thus translated at most once.

Having visited all basic blocks of the method body, PROCESS-SUBROUTINES (Figure 6, Line 5) is invoked to process the encountered `jsr` instructions collected in the worklist *SR*. The code of PROCESS-SUBROUTINES is shown in Figure 8. For simplicity, we expect subroutines to begin with an `astore` instruction that pops the return address from the stack and stores it in a virtual register, as suggested by the Java Virtual Machine Specification. In Line 29 of BUILD-BASIC-BLOCK we ensure that while compiling the code of subroutines, the value of this virtual register is not overwritten. If the first instruction is not `astore`, it is assumed that the subroutine does not make use of `ret` to return to the subroutine invocation site. Currently, our algorithm cannot deal with code that tries to shuffle the return address back and forth on the stack or between virtual registers. While theoretically this is permissible, we do not see any benefit in allowing to do so and consider this more a bug in the JVM specification than a limitation of our CFG construction algorithm. So far we have not encountered any compiler generated Java bytecode that did not fulfill this requirement.

Each subroutine invocation recorded in *SR* is processed separately. First, the basic block immediately following this `jsr` instruction is recorded in *RB*. The index at which the `astore` instruction at the beginning of the subroutine stores the return address is recorded in *RI* before BUILD-BASIC-BLOCK is invoked to process the subroutine code starting with the instruction following the initial `astore`.

Subroutines can be terminated either by a `ret` instruction or by transferring the control flow back to the method body by branching to an already processed basic block as indicated by *MAP*. To

```
BUILD-BASIC-BLOCK(i)
 1  if MAP[i] ≠ NIL
 2    then return MAP[i]
 3  NEW(b)
 4  MAP[i] ← b
 5  while i ≠ NIL
 6  do if BT[i] = true and |b| ≠ 0
 7        then n ← BUILD-BASIC-BLOCK(i)
 8             ADD-BRANCH-TARGET(b, n)
 9             return b
10     switch
11       case i = UnconditionalBranch :
12            t ← BUILD-BASIC-BLOCK(TARGET(i))
13            ADD-BRANCH-TARGET(b, t)
14            return b
15       case i = ConditionalBranch :
16            ...
17            t ← BUILD-BASIC-BLOCK(TARGET(i))
18            ADD-BRANCH-TARGET(b, t)
19       case i = jsr :
20            ...
21            PUSH(SR, i)
22       case i = ret :
23            if RB = NIL
24              then error "retoutside a subroutine"
25            if RI ≠ REGISTER-INDEX(i)
26              then error "Wrong register used with ret"
27            ADD-BRANCH-TARGET(b, RB)
28            return b
29       case i = RegisterStore :
30            if RB ≠ nil and RI = REGISTER-INDEX(i)
31              then error "Attempt to modify return address"
32            ...
33       case default :
34            ...
35    i ← i.next
```

Figure 7: Build a basic block by translating instructions until we reach a branch instruction. Using the information recorded previously by FIND-BRANCH-TARGETS, a new basic block is started for every instruction that is a branch target. Subroutine invocations are not immediately processed, but instead recorded in *SR*.

```
PROCESS-SUBROUTINES(SR)
1  MAP' ← MAP
2  while (i ← POP(SR)) ≠ NIL
3  do MAP ← MAP'
4     i ← TARGET(i)
5     if i = ReferenceRegisterStore
6       then RI ← REGISTER-INDEX(i)
7            RB ← MAP[i.next]
8     b ← BUILD-BASIC-BLOCK(i)
9     REDIRECT-JSR(i, b)
```

Figure 8: Process one by one every subroutine collected in the worklist *SR*. If the subroutine records its return address in a local variable, this information is retained in *RI* and *RB*, where *RI* contains the index of the virtual register containing the return address and *RB* points to the basic block immediately following the subroutine invocation. This information is used by BUILD-BASIC-BLOCK to identify the target of any `ret` instruction it encounters within the subroutine code.

ensure that each subroutine starts out with a list of basic blocks belonging to the method body only, *MAP* is copied to *MAP'* after the method body was processed and *MAP* is initialized from *MAP'* for every subroutine processed. If a `ret` instruction is encountered, its virtual register index must match *RI*. If this is not fulfilled, the CFG construction fails (Figure 7, Line 24). For correct Java code, this property is guaranteed as the JVM Specification does not allow subroutine code to "overlap". As we process each subroutine one by one, each `ret` instruction we encounter must relate to the subroutine we are currently processing. If such a `ret` instruction is located, it is transformed into a branch back to the basic block *RB* immediately following the `jsr` that called the subroutine currently being process.

It is important to remember that even while processing a subroutine, any additional `jsr` we encounter is only recorded in the work-list *SR* where it will be picked up later by PROCESS-SUBROUTINES.

The `jsr` instruction, after being processed, is replaced with a branch instruction to the subroutine entry. In contrast to instructions in the method body, subroutine code is not guaranteed to be visited only once. Instead, subroutine code is inlined into the CFG for each subroutine invocation with `jsr`. Thus, the worst-case runtime of PROCESS-SUBROUTINES is $O(j * m)$, with j being the number of subroutine invocations and m being the largest number of instruction in any subroutine. It is obvious that for certain (admittedly not very meaningful) programs, $O(j * m)$ could exceed $O(n)$, for example if a method consists mostly of subroutine invocations to a single subroutine with $m > 1$.

While theoretically this runtime is no longer $O(n)$, we suggest to view the subroutine construct of JVML as a form of structural compression. The subroutine construct was introduced into the Java bytecode format to prevent code duplication that would have otherwise resulted from compiling the Java *finally* clause. If we consider n to be the number of instructions of a method before applying this form of compression, our algorithm is indeed $O(n)$. The inlining of subroutine code into the CFG as performed by our algorithm is effectively "uncompressing" the bytecode to its original form.[2]

[2]For similar reasons, Sun Microsystems is considering to drop the subroutine construct from future versions of the JVM.

4.2 Proofing the Data Flow

Having obtained the control flow graph, we move on to translating the the stack-based Java bytecode representation to a typed, register-based intermediate representation (IR) that we will use for further analysis. Unfortunately, this process is anything but trivial for Java bytecode and traditionally requires a data-flow analysis to infer the correct type of stack values at join points in the control flow. The left side of Figure 9 shows such a join point. Basic block (b) leaves a value of type B on the stack before branching to (a). Basic block (c) branches to the same basic block, but leaves instead a value of type C on the stack. At the entry of (a), the stack can thus contain a value of either type B or C, in which case we have to infer the nearest common supertype of B and C, which is A. Since often the structure of the control flow graph is not obvious, basic blocks are translated with premature assumptions regarding the type of certain stack slots and are later iteratively re-evaluated using additional information discovered in the analysis. The same kind of iterative type inference has to be performed for virtual machine registers, with the additional complication that registers are allowed to contain values with conflicting or not statically determinable types as long as the code does not attempt to read such a register. Traditionally, such type inference problems are solved with a data-flow analysis.

Since we aim to provide an efficient alternative to data-flow analysis for bytecode verification, we cannot rely on a DFA based algorithm as part of our proofing approach. Instead, we construct a preliminary intermediate representation based on a superficial type system that distinguishes only between scalar types and a generic reference type ζ that represents all object types defined in Java. The mapping function Γ translates Java types into this superficial type system:

$$\Gamma(T) = \begin{cases} T : T \in \{byte, short, char, int, long, float, double\} \\ \zeta : \text{else} \end{cases}$$

As shown on the right hand of Figure 9, this reduced type system eliminates the need to perform type inference at join points in the control flow to merge stack states: Both A and B are mapped to ζ. This greatly simplifies the merging of stack states. Either the superficial type of each stack location is identical for all stack states being merged, or the stacks are incompatible and the proofing process is aborted. For each stack location s of type T_s we can calculate $\Gamma(T_s)$ in a single scan over the bytecode, while determining T_s itself would require to perform a DFA. However, while ensuring simple and efficient construction, using the superficial type system does not prevent incompatible reference types from being merged. This problem will be addressed in a later phase of our algorithm and we will revisit this issue shortly.

Like the types for stack locations, the types of virtual registers must be tracked throughout the code. In Java, virtual registers are addressed by an index only. Every time a value is written to a virtual register, the virtual register assumes the type of that value. It is the verifier's responsibility to ensure that values are read back from registers with the proper type. For every join point in the control flow, type maps for virtual registers have to be reconciled. If the same virtual register holds values of incompatible type along incoming control flow edges, its type is set to $\top$ to indicate that this register must not be read back until it is overwritten with a value of known type (left side of Figure 10).

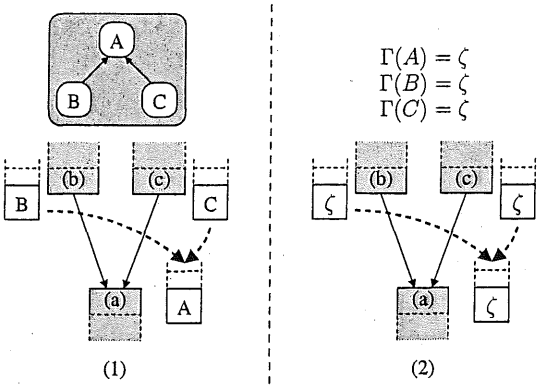

Figure 9: (1) A join point in the control flow is shown that requires type inference to determine the correct type of the value on top of the stack when using the full Java type system. (2) The same operation is performed using a superficial type system obtained by applying the mapping function Γ to each type. The need for iterative type inference is eliminated, as all Java object types are replaced by a single type ζ.

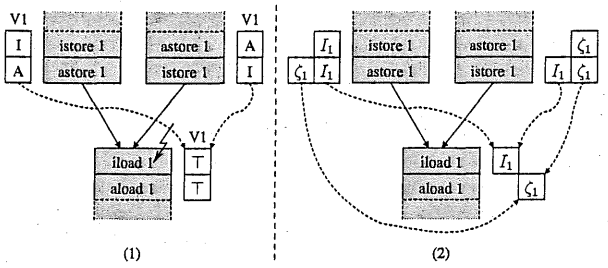

Figure 10: Proofing overlapping live ranges of values with different types in the same virtual register $V1$. The right side of the figure shows how we use separate register planes per superficial type to eliminate any potential live range collisions.

Verifying proper handling of virtual registers requires the understanding of the live ranges of values stored in them. This information can easily be calculated during the DFA traditionally performed for verification. However, this option is not available to us for obvious reasons. Instead, we opt to guarantee safety by *proofing* any potentially incorrect handling of virtual registers. Incorrect handling of registers can occur in two different ways: (1) code can attempt to read a value from a register with an incorrect/incompatible type and (2) code can try to read a register that is not yet initialized. (1) can be prevented easily by reinterpreting how register read/write instructions address registers. Instead of identifying registers by an index number, we introduce separate register planes for each type in the superficial type system. Thus, an `iload` n instruction will never be able to read anything but an integer from a register because its index n refers to the nth integer register. For correct code, this reinterpretation of the instruction set is semantically equivalent as it affects only code that (illegally) tries to overlap the live ranges of values with different types in the same register. As shown on the right side of Figure 10, we can correct such overlapping live ranges. The `istore 1` and `astore 1` instructions store their values in completely independent registers on different register planes even though they both use the same register index 1. Both load instructions in the join node are thus valid.

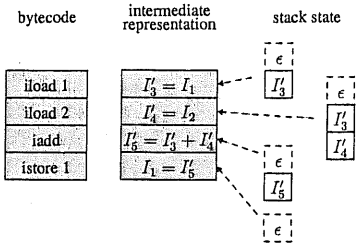

Figure 11: Transforming stack-based bytecode into a register-based intermediate representation. I_1 and I_2 correspond to virtual registers while I'_3, I'_4, and I'_5 are additional temporary registers newly allocated to hold values previously stored on the stack.

4.3 Transforming the CFG into SSA

Ensuring that no register is read before a value has been assigned to it requires the calculation of intraprocedural DEF-USE chains [1] for the CFG. We do this by transforming the CFG into the Static Single Assignment (SSA) form [29, 2, 8]. As the construction of a SSA representation is significantly more complicated for a stack-based representation, we eliminate the stack and transform the code to a register-based IR. Starting at the entry point and with an empty stack, we visit each instruction and allocate a new virtual register every time an instruction calculates a new intermediate result. We also maintain an abstract stack to track which virtual register has to be substituted when instructions expect to get their argument from the stack (Figure 11).

Each basic block is visited exactly once, thus this step again can be completed in linear time. Join nodes that are reached with a non-empty stack expect the value to reside in a particular register. `move` instructions are added to predecessor basic blocks to ensure that the value is copied to the appropriate register before branching to the join node. The resulting register-based IR is transformed into SSA using the DJ-Graph algorithm [30, 31]. In [31], the authors have shown that it is possible to transform into SSA [6] and to place ϕ-instruction in linear time (per variable). Transforming into SSA also implicitly checks that all registers are defined before their use and any program containing undefined uses of registers is rejected.

4.4 Precise Type Checking

Up to this point, we have only considered the superficial types of values. This allowed us to shortcut the iterative DFA usually required for this step, and we were able to transform the bytecode into a register-based CFG in SSA form. However, to ensure that the code is indeed well-typed, we have to re-evaluate the code based on the original Java type system. For this, we first perform a copy propagation sweep over the CFG, which is trivial in SSA: For all registers defined by a `move` instruction, we record the source register in a table. In a second iteration over the code, all uses of such registers are replaced with their definition and all `move` instructions are deleted.

As shown in Figure 12, pruning all `move` instructions from the CFG directly connects all uses of registers with their definition site. This automatically removes all ambiguities introduced by the dynamically typed virtual registers. While the traditional Java verification algorithm has to calculate the precise type of a value that is stored in a virtual register and has to make sure it is read back with the proper type, we have effectively skipped this step by propagating the real definition to the use site. This dramatically simplifies type checking. We simply have to ensure that each use

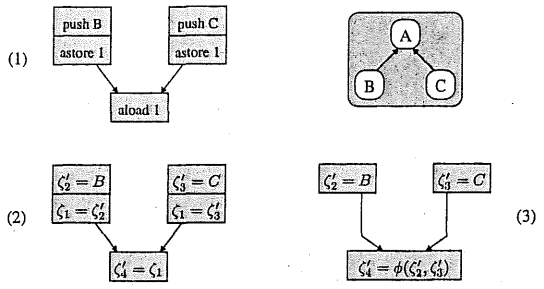

Figure 12: Precise type checking: The stack-based bytecode (1) is transformed to a register-based representation (2). Both `push` instructions store their return value in a temporary register (ζ'_2 and ζ'_3). The temporary registers are then assigned to ζ_1, which corresponds to the JVM register 1 addressed by the `astore` instructions. (3) The IR is transformed into SSA and copy propagation is performed. The temporary register ζ'_4 defined by the ϕ-instruction has the type A, which is the common ancestor of the operand types of the ϕ-node.

refers to a definition with a compatible type. As registers are always defined exactly once in SSA, this is again simple and can be performed in a single linear scan over the code.

Join points in the control flow that cause so much work for the Java verifier are represented by ϕ-nodes in our CFG. For each ϕ-node, we only have to check that all operands are compatible. Type inference can be done in a single step as all incoming types are known at once. A register defined by a ϕ-node has exactly the same type as all the operands of the ϕ-node—or a common ancestor if the ϕ-node has operands with different (but compatible) types. If a ϕ-node contains any operands for which no common ancestor can be found (incompatible), the program is not well-typed and is rejected.

5. VERIFICATION BY PROOFING

After all steps described in the previous section, the intermediate representation is guaranteed to contain a (1) well-formed and (2) well-typed program. (1) results from the internal representation as a control flow graph, and (2) results from the SSA form and the additional type checking we performed for all uses of register values, and in particular ϕ-node operands. As we have mentioned earlier, being able to construct a well-formed, well-typed IR for a Java program using the algorithm described in this paper does not necessarily mean that the input program was well-typed according to the Java specification. As we have seen in the previous section, only a small number of obvious syntactical and semantical errors are rejected; in particular, those that can be decided in linear time. More complex data-flow related potential type errors are instead corrected by reinterpreting the Java bytecode representation and using typed register planes.

Moreover, our approach has another subtle advantage over traditional DFA-based bytecode verification. Resulting from certain features in the Java source code language [12] and some restrictions in the specification of the Java bytecode verifier, a number of legal Java source code programs exist that cannot be verified using traditional verifiers. Two examples for such programs were published by Stärk et al. in [28] and are shown in Figure 13.

The first example (`Test1`) contains a conditional return statement encapsulated in a `try` block. If b is `true`, the method terminates in line 6 with the return value 1. However, before returning to the caller, the `finally`-clause still has to be executed. The `finally`-clause sets i to 3 if b is `true`, which is actually moot in this case as the return value was determined to be 1. On the other hand, if b is `false`, i will be set to 2 in line 7, the `try` block ends, which means that the `finally`-clause has to be invoked. As b is `false` now, i remains unchanged (line 9) and 2 is returned in line 11.

```
 1: class Test1 {
 2:   int m1(boolean b) {
 3:     int i;
 4:     try  {
 5:       if (b)
 6:         return 1;
 7:       i=2;
 8:     } finally {
 9:       if (b) i=3;
10:     }
11:     return i;
12:   }
13: }
```

```
 1: class Test2 {
 2:   int m2(boolean b) {
 3:     int i;
 4:   L: { try {
 5:          if (b) return 1;
 6:          i=2;
 7:          if (b) break L;
 8:        } finally {
 9:          if (b) i=3; }
10:       i=4;
11:     }
12:     return i;
13:   }
14: }
```

Figure 13: Two Java source code programs that can be compiled by a Java compiler, but are not verifiable using the standard DFA-based verifier. Bytecode proofing, in contrast, allows the programs to execute.

At the bytecode level, a `finally`-clause is represented by a subroutine, which is called every time the `finally`-clause has to be invoked. Unfortunately, from the verifier's perspective, this particular subroutine is not verifiable, because the Java verifier specification [19] requires each subroutine to have a unique stack and virtual register state for each of its instructions. This is, however, not the case in this example. If the subroutine representing the `finally`-clause is triggered in line 6 by the `return` statement, the register holding the local variable i is not initialized. On the other hand, if the subroutine is called after completing the guarded block, i (and the register holding it) is initialized and contains a valid value. As the local variable holding i is not defined along all paths, it is assumed to be not usable when the `return` statement is reached in line 11, which in turn does not verify because i is not defined—or at least the verifier cannot prove that it is defined.

In contrast, our bytecode proofing algorithm is not fooled by the ambiguity of the data-flow through the subroutine. Since we inline the subroutine code into each call site, the data-flow through the subroutine is reconsidered separately for each call site.[3] Thus, the algorithm is able to see the definition of i in line 7 and the `return` statement in line 11 is recognized to be valid.

The second example (`Test2`, Figure 13) is again not verifiable using traditional means, because the verifier is not able to see the definition of i at line 6 as the data flow is obfuscated by the subroutine invocation. As in the previous example, this definition of i becomes obvious if for each subroutine the data-flow is considered separately.

To ensure that all valid Java source code programs can pass the verifier in bytecode form, Stärk et al. proposed in [28] a number of additional restrictions for the Java source code language. Using our bytecode proofing approach, however, these additional restrictions

[3]Considering the data-flow separately for each subroutine invocation is the reason why our CFG recovery algorithm has only linear complexity w.r.t. the "uncompressed" size of the bytecode with all subroutines inlined into their respective invocation sites.

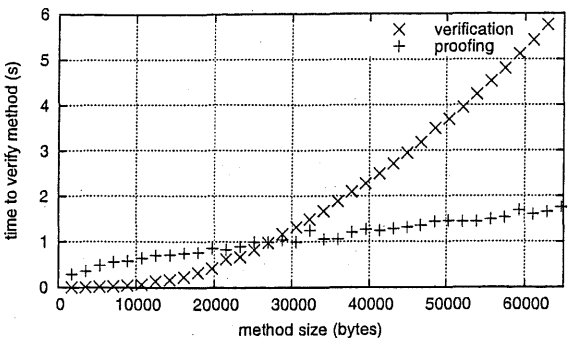

Figure 14: Comparing the performance of the traditional verifier with the performance of bytecode proofing for a difficult-to-verify method.

are no longer necessary.

6. IMPLEMENTATION

We have implemented the bytecode proofing algorithm described in this paper as part of our network-centric mobile code framework [24]. The choice of Java itself as our implementation language was sub-optimal in retrospect, as far as performance is concerned, in particular when trying to evaluate linear time algorithms. For a more thorough quantitative analysis and comparison to existing verifiers we will have to port our implementation to C or C++, which are more commonly used in this domain.

Even though our implementation is an early prototype, we expect it to be free of any major vulnerabilities, because the code handed to the virtual machine is in a format that guarantees, by construction, that the code is well-formed and well-typed. It is, however, much more difficult and time consuming to ensure that the implementation is correct in terms of producing a semantically equivalent CFG for the input program and thus would produce the expected result when interpreted or compiled to native code. Since no established metric exists for verifier and compilation correctness, we will limit ourselves here to a performance comparison of the traditional verifier algorithm and bytecode proofing. Figure 14 shows how the two algorithms perform in case of the worst-case data-flow scenario described in Section 3. The time shown for bytecode proofing does not include the time to transform the CFG into SSA, because we are currently using the standard SSA transformation component available in our mobile code framework, which is based on [8] and does not have linear time complexity. We plan to replace the SSA transformation with the more efficient DJ-graph algorithm in the near future.

7. RELATED WORK

Java bytecode verification has been explored quite extensively in the past [18, 32]. In addition to the informal description of the JVM [19], a number of formal specifications of the JVML and its verifier have been proposed [9, 10], and proven to be sound [25, 14, 15, 16, 33, 5]. Resulting from the formalization of the verification process, improvements for the original specification have been proposed [7, 28]. In this context, subroutines are of particular interest and several type systems have been proposed for them [34, 23, 26, 17]. All these approaches have in common that they rely on some form of iterative data-flow analysis [18, 27] to decide type-safety and thus have a quadratic worst-case runtime complexity.

Proof-carrying code (PCC) [21, 20] 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 only has to recompute the verification condition and can then check whether the attached proof indeed establishes the verification condition as claimed by the code producer.

The bytecode *proofing* approach presented in this paper was in many ways inspired by the work on PCC. Just as with PCC, we try to avoid having to prove type safety on the code consumer side by not deciding through analysis whether mobile code is safe or not, but by merely guaranteeing that the virtual machine is shielded from code that is not well-typed. While both approaches, PCC and bytecode proofing, require only a linear effort on the code consumer side, in practice a proof checker for PCC is much less complex and easier to implement than the proofing process described in this paper. 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. In this regard, *bytecode proofing* has the advantage that it can operate on the standard Java classfile format [19] 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 [22, 4], 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.

Inherently safe mobile code representation formats such as Safe-TSA [3] 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 bytecode proofing, but require abandoning the existing Java classfile format, which is not always acceptable.

8. CONCLUSIONS

Guarding the virtual machine from malicious code has traditionally been accomplished by *deciding* the type safety of the input program. This decision can require quadratic runtime behavior in certain cases, allowing malicious programs to stall the verifier (and thus the entire virtual machine) for a prolonged time in what can amount to a denial-of-service attack. Bytecode verification through iterative data-flow analysis is not only expensive, but also not strictly required if the only goal is to protect the virtual machine from malicious code.

We have presented *bytecode proofing* as an alternative approach to verification. It guarantees through a series of transformations that malicious programs are either rejected or transformed into a well-formed and well-typed program that no longer threatens the integrity of the virtual machine. By avoiding the unnecessary overhead of proving safety, and merely proofing the input program to guarantee that the code received by the virtual machine is safe, we are able to complete this transformation in linear time, which is a significant improvement over the quadratic worst case runtime complexity of bytecode verification.

The key contribution of this work is the introduction of a *superficial type system* to efficiently construct a preliminary register-based

intermediate representation from Java bytecode. This preliminary IR allows us to reason about the data-flow before type-safety is established through precise type-checking using the full Java type system. Thus, we can transform the intermediate representation into SSA form and perform copy propagation before knowing that the program is actually type-safe. This allows us to perform the actual precise-type checking using SSA form and in linear time.

As we have shown in Section 5, our bytecode proofing approach has not only a better worst-case performance than traditional bytecode verification, but also allows us to execute a certain class of legal Java programs that are not verifiable by existing JVMs. As an additional benefit, our bytecode proofing algorithm provides a well-formed and type-checked control flow graph in SSA to the virtual machine, eliminating the need to separately transform the bytecode into SSA for just in time compilation.

9. FUTURE WORK

Evaluating a new verifier algorithm requires an extensive set of test cases. While our prototype implementations accepts a large test set of publicly available Java programs and libraries, and rejects (or proofs) a number of hand constructed not well-typed programs, this can hardly be seen as a general proof of correctness. It would be desirable to test our implementation more rigorously using an established test set. An extensive set of test cases is available from Sun Microsystems as part of the Java Compatibility Kit (JCK). However, the JCK is available to commercial JVM implementors only. A free alternative to JCK is the Mauve Project [13], but it offers only very limited coverage of many critical areas of the JVM. We are currently working on adding missing parts to the test cases provided by the Mauve Project, to have a more complete testing environment for our prototype implementation.

Another area for improvement is the handling of subroutines. The algorithm described in this paper does not currently allow a subroutine to use `ret` with anything but the virtual register that the initial `astore` stored the return address in. In particular, return addresses cannot be moved to other virtual registers and nested subroutine calls cannot return to the calling site of outer subroutines. While this kind of code is not actually generated by any compiler we know of, from a more academic perspective this is still a restriction we would prefer to eliminate. We are exploring how to properly track return addresses in registers without giving up the linear time complexity of the algorithm.

10. ACKNOWLEDGMENT

This research effort was sponsored 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 ONR or any other agency of the U.S. Government.

We are indebted to Christian H. Stork and Vasanth Venkatachalam for valuable comments on earlier versions of this paper.

11. REFERENCES

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, 1986.

[2] B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting Equality of Values in Programs. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pages 1–11, San Diego, California, January 1988.

[3] 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.

[4] 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.

[5] 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.

[6] G. Bilardi and K. Pingali. Algorithms for Computing the Static Single Assignment Form. *Journal of the ACM (JACM)*, 50:375–425, may 2003.

[7] A. Coglio. Improving the Official Specification of Java Bytecode Verification. In *Proceedings of 3rd ECOOP Workshop on Formal Techniques for Java Programs*, June 2001.

[8] R. Cytron, 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(4):451–490, October 1991.

[9] 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.

[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. A Denial of Service Attack on the Java Bytecode Verifier. Technical Report 03-23, University of California, Irvine, School of Information and Computer Science, 2003.

[12] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[13] A. Green. The Mauve Project Home Page, November 2003. `http://sources.redhat.com/mauve/`.

[14] G. Klein. *Verified Java Bytecode Verification*. PhD thesis, Institut für Informatik, Technische Universität München, 2003.

[15] G. Klein and T. Nipkow. Verified bytecode verifiers. *Theoretical Computer Science*, 298(3):583–626, April 2003.

[16] G. Klein and M. Strecker. Verified Bytecode Verification and type-certifying Compilation. *Journal of Logic and Algebraic Programming*, 2003. To appear.

[17] G. Klein and M. Wildmoser. Verified Bytecode Subroutines. *Journal of Automated Reasoning*, 30(3-4):363–398, 2003.

[18] X. Leroy. Java Bytecode Verification: Algorithms and Formalizations. *Journal of Automated Reasoning*, 30(3/4):235–269, 2003.

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

[20] G. C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, Paris, France, January 1997.

[21] G. C. Necula and P. Lee. Safe Kernel Extensions Without Run-Time Checking. In USENIX, editor, 2^{nd} *Symposium on Operating Systems Design and Implementation (OSDI '96)*, pages 229–243, Berkeley, CA, USA, 1996. USENIX.

[22] G. C. Necula and S. P. Rahul. Oracle-based Checking of Untrusted Software. *ACM SIGPLAN Notices*, 36(3):142–154, 2001.

[23] 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.

[24] C. W. Probst, A. Gal, and M. Franz. Code Generating Routers: A Network-Centric Approach to Mobile Code. In *Proceedings of the IEEE Computer Communications Workshop (CCW 2003)*, Laguna Niguel, CA, 2003.

[25] C. Pusch. Proving the Soundness of a Java Bytecode Verifier Specification in Isabelle/HOL. *Lecture Notes in Computer Science*, 1579:89–103, 1999.

[26] 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.

[27] Z. Qian. Standard Fixpoint Iteration for Java Bytecode Verification. *ACM Transactions on Programming Languages and Systems*, 22(4):638–672, 2000.

[28] 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.

[29] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global Value Numbering and Redundant Computations. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pages 12–17, San Diego, California, January 1988.

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

[33] 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.

[34] R. Stata and M. Abadi. A Type System for Java Bytecode Subroutines. *ACM Transactions on Programming Languages and Systems*, 21(1):90–137, 1999.

[35] 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.