An Algorithm for Verifying Java Bytecode Subroutines

Stéphane Doyon and Mourad Debbabi

Computer Science Department,
Laval University,
Quebec, Canada
{doyon,debabi}@ift.ulaval.ca

Abstract. Java is an ideal platform for implementing mobile code systems, not only because of its portability but also because it is designed with security in mind. Untrusted Java programs can be statically analyzed and validated. The program's behavior is then monitored to prevent potentially malicious operations. Static analysis of untrusted classes is carried out by a component of the Java virtual machine called the verifier. The most complex part of the verification process is the dataflow analysis which is performed on each method in order to ensure type-safety. This paper presents the underlying algorithms and discusses various implementation alternatives that allow the dataflow analysis to support the verification of subroutines, which is one of the most tricky aspects of the dataflow analysis. The dataflow analysis must take into account the fact that Subroutines are polymorphic over registers that they do not modify. Properly tracking the control flow is also a challenge. Mostly, a last-in/first-out discipline is enforced, but we also want to support "sharing" of a subroutine without strict nesting and "falling out" of a subroutine.

1 Introduction

Java is especially well suited for implementing mobile code systems for three reasons:

- Java source is compiled into a platform-independent intermediate form called *Java bytecode*. Java bytecode is then interpreted by the JVM (Java virtual machine). This makes Java bytecode completly portable, which means a piece of Java code in compiled form should run on any receiving machine.
- It is dynamically linked: the JVM will load classes from different network sources as they are needed and will link them into the program while it runs.
- The Java architecture is built with security in mind: its design makes it possible to enforce sufficient security to make mobile code safe and practical.

The *verifier* is a key component of the Java security architecture. Its job is to examine compiled classes as they are loaded into the JVM in order to ensure that they are well-formed and valid. It checks that the code respects the syntax of the bytecode language and that it respects the language rules.

The most complex step of the verification process performed by the verifier requires running a dataflow analysis on the body of each method. There are a few particularly tricky issues regarding the dataflow analysis. In this paper, we focus on how the dataflow analysis can handle Java subroutines. Two aspects of subroutines need to be considered:

- Subroutines are polymorphic over registers they do not modify.
- The dataflow analysis must properly track the flow of control and must determine to which subroutine or nesting subroutines each instruction in the method body belongs.

Our paper discusses verification of subroutines from an algorithmic and implementation point of view. We also touch on a few issues not discussed in [6]: the problem of the order in which `ret`s and `jsr`s are visited, interaction of subroutines with the strategy for verifying object initialization, and “falling out” of a subroutine. We assume that the reader has some knowledge of the Java bytecode language, as well as a basic understanding either of dataflow analysis in general or of the particular analysis technique used by the Java bytecode verifier.

2 Overview of the Dataflow Analysis

The most complex yet most interesting part of the verification process is the dataflow analysis. It is performed independently on each method. The dataflow analysis checks that each bytecode instruction gets arguments of the proper type (from the stack or from the registers), detects and prevent overflows and underflows of the expression evaluation stack and ensures that newly allocated objects are not used before being properly initialized. The dataflow analysis also must check that subroutines are used consistently. This paper will attempt to clarify the properties actually enforced over subroutines and the way in which a verifier implementation can enforce them.

In order to perform the dataflow analysis, it is necessary to keep track of the type of each value on the stack and in the registers at each program point. We will assume that each instruction of a method constitutes a program point, although it is possible to use fundamental blocks of instructions as program points. The type which is recorded by the dataflow analysis for a given location at a given program point must be consistent, irrespective of the execution path used to reach that program point. When there is a conflict because two or more paths would yield different types of values for the same location, then we record for that location a common supertype of all the types that could actually occur. If there are no common supertypes for the possible types in a certain location, then the type *unusable* is used, indicating that the value cannot be used by the following instructions. This generalization of types does imply a loss of information and precision. This is what makes the analysis conservative, in the sense that it is pessimistic.

3 Subroutines

It is important to understand that subroutines actually do more than simply provide a clean way to transfer the control flow. Subroutines have the added advantage of being polymorphic over the registers that they do not touch. Without this flexibility, subroutines are practically useless. Some special strategy must therefore be used in the dataflow analysis so that code using polymorphic subroutines can be made acceptable for the verifier.

The dataflow analysis must ensure that the value fetched by the `ret` instruction is indeed of type *returnAddress*. In theory, no further rules or restrictions should be needed to ensure the safety of the language. However, some restrictions are necessary in order to perform the dataflow analysis. To allow the dataflow analysis to successfully analyze subroutines, a bit of coherence must be imposed on the use of `jsr` and `ret`. We must be able to tell which instructions are part of a subroutine, where a subroutine starts and where it ends, which instructions belong to the same subroutine, and what the successors of a `ret` instruction are. If we do not enforce some structure, then `ret` could be used to do strange things to the control flow graph that would make the management of subroutine polymorphism infeasible. We will want to ensure that methods mostly respect a last-in / first-out discipline for their calls and returns from subroutines. However there are several ways of accomplishing all this and small variations between the possible solutions have subtle but important repercussions.

4 Handling Subroutine Polymorphism

4.1 General Strategy

Although proposed solutions do not entirely agree on the structure that should be imposed on subroutines, they roughly agree on the strategy that the dataflow analysis should use to manage polymorphism. The dataflow analysis must identify which registers the subroutine modifies (touches). The state information used by the dataflow analysis should be extended so that the state associated to each instruction includes one flag for each register, which will be used to indicate whether or not a register has been touched by the current subroutine. Formally, these register flags can be modeled as a set (the set of touched registers). An implementation may choose to represent the flags for all registers by a bit mask/vector. For the first instruction of a subroutine, the *touched* flags for all registers should be unset (except for the flags of registers containing references to uninitialized objects, as will be discussed later on). A register's *touched* flag should be set in the type state resulting from any instruction which stores a value to that register.

Successors of a `ret` are the instructions immediately following all `jsr` instructions that called the current subroutine[1]. A special procedure is used to

[1] Note that `ret` instructions can also have exception handlers as ordinary successors: for these, the normal procedure is used and no polymorphism occurs.

merge the state of a `ret` instruction into its successors. The portion of the state describing the stack is merged normally. Those registers that have their flags set are also merged as usual. To merge registers that do not have their flags set, the type they had before the subroutine was invoked is used: instead of merging the register's type from the state of the `ret`, the type the register had in the state of the `jsr` is used. As this is done on a per successor basis (each successor corresponds to a different `jsr`), polmorphism is achieved.

When merging two typing states, the register *touched* flags should be *OR*ed: when merging two flags, the resulting flag is set if any of the two flags being merged is set. The effect is that a register is considered *touched* if it is modified at some point on at least one path from the entry point of the subroutine to the `ret` instruction.

A small difficulty arises when running the dataflow analysis algorithm. When a `ret` instruction is visited, it may well happen that some `jsr` instructions calling the current subroutine have not been visited yet. In that case, their state is not available to complete the merges into the `ret`'s successors. Some trick must be devised which allows the dataflow analysis to continue but which remembers that the `ret`'s merge is not complete. Ways of doing this will be presented later on.

4.2 Order For Visiting `jsr` and `ret`

As explained above, a complication arises when trying to merge the state resulting from a `ret` instruction into its successors. To do this, the state of the `jsr` that precedes each successor must be available. However these `jsr` instructions might not all have been visited yet. Some strategy must be used to allow the dataflow analysis to continue while remembering that the merge was not completed. These are possible solutions to the problem.

- Sun's implementation requires that only one `ret` instruction correspond to each `jsr` instruction: this marginally simplifies the problem. When a `ret` instruction is processed, all `jsr`s that called the current subroutine are stamped with the instruction number of the current `ret`. When a `jsr` instruction is processed, if its corresponding `ret` instruction is known (if that `jsr` has been stamped) then the `ret` instruction's *changed* bit is set.
- A more crude but perhaps simpler approach would be to wait until the algorithm finishes and no more instructions with their *changed* bit remain, and then to mark all `ret` instructions (except possibly those that were unreachable) as *changed*. All `ret` instructions would then be merged again into their successors. If that sets any other instruction's *changed* bit, then the algorithm should be continued: again when it finishes, all `ret` instructions should be marked *changed*. The process really stops when the final merging of `ret` instruction results cause no *changed* bit to be set. A variant would be to mark incompletely merged `ret` instructions with a second, lower priority *changed* bit.

– A simplified combination of these two techniques is to set the *changed* bit of all visited `ret` instructions whenever a `jsr` instruction is being processed.
– Yet another simple alternative would be that whenever a new `jsr` is encountered, the dataflow analysis could be forced to go through that subroutine again. A trick would be to record within the *returnAddress* type the list of all `jsr` instructions calling a subroutine. As each `jsr` is processed, it could append its instruction number to the list of numbers within the *returnAddress* type found on top of the stack in the state associated to the first instruction of the subroutine. The *changed* bit of the first instruction of the subroutine would be set. Merging of *returnAddress* types could be defined so that the instruction number of the new `jsr` would propagate through all subroutine instructions (which have that *returnAddress* in their state) until the `ret` instructions are reached and processed again.
– A different approach would be to immediately merge the *touched* registers in all successors of `ret` instructions. We need to be particularly careful in order to support multiple subroutine exit points: a register could be polymorphic over a path through the subroutine leading to one `ret` instruction, yet non-polymorphic on a path leading to another `ret` instruction. In the typing state of all successors of `ret` instructions, we will need to keep track of the subset of *never-polymorphic* registers: those that are modified on at least one path leading to each `ret` of the current subroutine.

4.3 Interaction With Object Initialization

As noted in [3], subroutine polymorphism can cause the strategy used for verifying object initialization to fail, allowing two uninitialized objects produced by the same `new` instruction to exist simultaneously. The solution proposed in [3] is to change all uninitialized types on the stack or in registers to *unusable* when processing `jsr` and `ret` instructions.

We believe that a simpler and less restrictive approach would be to disable polymorphism on registers that contain references to uninitialized objects. When processing a `jsr` instruction, all registers containing an uninitialized type should be located. In the state resulting from the `jsr` (the state of the first instruction of the subroutine) the *touched* flags for these registers should be set. This will force any changes of such registers due to type merges to be propagated out of the subroutine.

5 Imposing Structure

5.1 Characteristics

We present our own structure rules, which should take advantage of the strong points of both the solution implemented by Sun and that proposed in [6].

The structure we impose on subroutines has the following restrictions:

– *Subroutine recursion is not allowed.* Writing recursive subroutines in the bytecode language is tricky because a stack of return addresses must somehow be maintained. Subroutine recursion is not needed to compile Java. From the start, it has been decided that the dataflow analysis did not need to be complicated to support subroutine recursion: it is disallowed by the JVM specification.
– *Single entry point.* All proposed solutions identify a subroutine by the number or offset of its first instruction. If the targets of two `jsr` instructions are different, then it is assumed that these are calls to different subroutines.
– *`ret` must use the right return address.* `ret` must use the instance of *returnAddress* that was created in the current invocation of the current subroutine. `ret` must not return to a return address that belongs to a different subroutine (except when falling out, see later on). A subroutine must not be able to "reuse" a return address: it must not be permitted to save a return address during a first call to the subroutine, and then use that return address during a second call.

On the other hand, our structure supports the following flexibilities:

– *Multiple exit points.* [5] requires subroutines to have only one exit point: subroutines may not split into two or more paths with each their own `ret` instruction.
– *Sharing of subroutines by other subroutines.* The reference implementation enforces strict nesting of subroutines. We believe that refusing subroutine sharing is counter-intuitive and although this flexibility is not currently required to compile Java, it may be helpful if subroutines are to be used for optimizing code or in compiling other languages to bytecode. [6] also supports this flexibility.
– *Fall out of a subroutine.* The control flow is allowed to leave the subroutine without using `ret` (although no polymorphism occurs in that case). If the subroutine is nested, then the control flow is allowed to pass from the subroutine back to one of the subroutines into which it is nested without using a `ret`, as long as it is certain that the subroutine to which control returns is part of the chain of callers. This flexibility is necessary to compile nested `finally` clauses protected by exception handlers. Unfortunately, this flexibility is not supported by the model presented in [6].

The aspects that require particular attention are therefore the issue of using the right return address, sharing and "falling out" of a subroutine, as well as subroutine recursion.

5.2 Our Solution

We now describe a strategy to implement these rules and features. The state information associated to each instruction is extended to include a set of subroutine labels, called *returnable*, indicating from which subroutines we may return. This corresponds to those subroutines which we are certain are in the current subroutine call chain (and to which we may "fall out"). In an implementation, a subroutine label is the number or offset of the first instruction of that subroutine.

These additions to the typing state, as well as the *touched* flags for each register, need not be kept for instructions that are not part of a subroutine.

When merging two typing states, the *returnable* set in the result of the merge should be the intersection of the *returnable* sets of the states being merged.

Remember that when merging two type states, register *touched* flags should be *OR*ed.

The *returnAddress* type created by `jsr` is also extended so that it contains the label of the subroutine called. Two *returnAddress* types merge only if their associated label is the same, otherwise the merge results in *unusable*.

When the dataflow analysis processes a `jsr` instruction, the label of the subroutine being called is added to the *returnable* set in the resulting state of the `jsr` (which is to be merged into the first instruction of the subroutine). Note that this same state should also initialize all register *touched* flags to unset, except possibly for those registers containing references to uninitialized objects.

When the dataflow analysis processes a `ret` instruction:

- The label of the subroutine we are returning from can be obtained from the extra information associated to the *reutrnAddress* type in the `ret`'s target register.
- If the label thus obtained is not in the *returnable* set of the state associated to the `ret`, then verification fails.
- Knowing the label of the subroutine we are returning from, it is possible to find the list of the `ret`'s successors: the instructions that follow all `jsr` instructions that call to that label.
- The stack and registers are merged in each successor as explained above (section 4).
- The register *touched* flags are also merged (if the subroutine is nested).
- But the *returnable* set is not merged. Instead, the set from the state of the `jsr` is used, just as for polymorphic registers.

Any register touched by a nested subroutine will also be marked as touched by the encompassing subroutine, whether the nested subroutine ends with `ret` or "falls out" to the encompassing subroutine. On the other hand, the encompassing subroutine may touch a register while the nested subroutine is polymorphic over that same register.

Ensuring that a `ret` instruction uses an appropriate return address does not require suplementary effort.

Strategies based on a dataflow analysis such as ours will not allow worthwhile uses of recursive subroutines. Our strategy would accept only degenerate uses of recursive subroutine, such as cases that create infinite loops.[2]

[2] Infinite loops are not a threat to the security of the system. In some special cases, infinite loops might even be the intended behavior. The verifier cannot hope to catch all cases of infinite loops: this is the halting problem, and it certainly remains infeasible even with the constraints that were added to perform the static analysis.

However the JVM specification specifically disallows subroutine recursion. To detect and prevent recursion, another set, named *called*, should be added to the typing state associated to subroutine instructions. This set is managed much in the same way as *returnable* with one subtle exception: when merging states from a `jsr` into the first instruction of a subroutine, the *called* set in the result of the merge should be the union (rather than the intersection) of the *called* sets being merged. For the *called* sets to be valid, we must wait until the dataflow analysis completes: this is necessary to identifiy paths that "fall out" of subroutines. When the dataflow analysis has completed, we can check for subroutine recursion. All `jsr` instructions should be checked: if the associated state contains a *called* set and if that set contains the label of the subroutine that the `jsr` is about to call, then recursion is detected.

6 Related Works

The Java Virtual Machine Specification [5] is the only relevant official documentation. Section 4.9 briefly presents Sun's verifier implementation. This description is reproduced in [7]. Section 4.9.6 deals with verifying `fi9nally` clauses: it introduces the notion of polymorphism and hints at other issues involved in verifying subroutines. Some of the constraints listed in section 4.8.2 restrict the uses of subroutines. It is not however explained why these restrictions should be necessary and how they fit in the verification process. [6] proposes that a type system can be used to formally describe the verification process. It presents a formalization of a small subset of the Java bytecode language and uses a type system to verify Java subroutines. That paper therefore helps to clarify many aspects of subroutines. [3] extends the framework proposed in [6] to support the verification of object initialization. This work has uncovered a bug caused by the interaction of the strategy used to verify subroutines and object initialization.

For the basic principles of dataflow analysis see [2]. [1, 4] are examples of uses of typing to accomplish security.

7 Conclusion

In this paper, we have explained the difficulties in verifying subroutines. In particular, we have dealt with the issue of register polymorphism and imposing structure on the control flow graph. We have provided algorithms and discussed various alternatives that allow the dataflow analysis to support the verification of subroutines which can be shared without strict nesting, as well as subroutines whose control flow "falls out" to merge back with the caller's code. We have also provided discussions that clarify why our proposed strategies are necessary and how they work.

Although Java subroutines are rather complex, the information provided in [6] and in this paper sums up to a thorough, in-depth discussion of the subject. Given that documentation on the verifier is relatively sparse and considering that the verifier is at the basis of the Java security architecture, we believe that our work of exploring and detailing tricky aspects of the verifier will prove invaluable to implementors and security analysts.

References

[1] Martín Abadi. Secrecy by typing in security protocols. In *Proceeedings of the International Symposium on Theoretical Aspects of Computer Software (TACS'97)*, volume 1281, pages 611–638, Tohoku University, Sendai, Japan, September 24 - 26 1997. Lecture Notes in Computer Science, LNCS.

[2] Alfred V. Aho, Ravi Sethi, and Jeffrey D Ullman. *Compilers: Principles, Techniques, and Tools.* Addison Wesley, 1986. ISBN 0-201-10088-6.

[3] Stephen N. Freund and John C. Mitchell. A type system for object initialization in the Java bytecode language. Technical Note STAN-CS-TN-98-62, Stanford, Computer Science, 1998. http://Theory.Stanford.EDU/ freunds/jvm-full.ps.

[4] Nevin Heintze and Jon Riecke. The slam calculus: Programming with security and integrity. In *Proceeedings of the DARPA Workshop on Foundations for Secure Mobile Code*, Monterey, California, USA, March 26 - 28 1997. http://www.cs.nps.navy.mil/research/languages/statements/nevjon.ps.

[5] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification.* Java Series. Addison Wesley, 1996. ISBN 0-201-63452-X. http://java.sun.com/docs/books/vmspec/index.html.

[6] Raymie Stata and Martín Abadi. A type system for Java bytecode subroutines. In *Proceeedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, San Diego, California, USA, January 19-21 1998. ACM Press. http://www.research.digital.com/SRC/personal/Martin_Abadi/Papers/jvm-short-preprint.ps.

[7] Frank Yellin. Low level security in Java. In *Fourth International World Wide Web Conference*, MIT, December 1995. http://www.w3.org/pub/Conferences/WWW4/Papers/197/40.html.