InvokeDynamic support in Soot *

Eric Bodden

Secure Software Engineering Group
European Center for Security and Privacy by Design (EC SPRIDE)
Technische Universität Darmstadt
Darmstadt, Germany
eric.bodden@ec-spride.de

Abstract

Java Specification Request (JSR) 292, which was realized with Java 7, defines a new java bytecode called invokedynamic, which can be used to call methods by name, without determining statically where the implementation of the called method is to be found. This mechanism eases the implementation of highly dynamic languages for the Java Virtual Machine.

In this work we explain how we extended the Soot framework for static analysis and transformation of Java programs to properly handle invokedynamic bytecodes. Our implementation required changes on all levels of Soot, as all intermediate representations needed to be adapted appropriately. We comment on the design decisions taken and how users can use our implementation to statically process or generate invokedynamic instructions.

Our support has been integrated into Soot release 2.5.0 and is thus already available for everyone to use.

Categories and Subject Descriptors F.3.2 [*Semantics of Programming Languages*]: Program Analysis

General Terms Design, Documentation

Keywords Static analysis, dynamic analysis, invokedynamic, JSR292

1. Introduction

Prior to Java 7, the Java Virtual Machine specification [7] allowed for different method invocation bytecodes, tailored towards the invocation of virtual and static methods (`invokevirtual`/`invokestatic`), constructors and private methods (`invokespecial`) as well as invocations of methods on interface types (`invokeinterface`). Despite their variety, all those bytecodes have in common that, given any fixed call site, there is a fixed algorithm for looking up an appropriate unique callee method, depending on the particular type of bytecode chosen. This fact makes it very hard to generate Java compliant bytecode for highly dynamic languages such as Ruby or Groovy [4, 5].

Java Specification Request (JSR) 292, titled *Supporting Dynamically Typed Languages on the Java^{TM} Platform* [2] set out to change this fact by introducing a new bytecode called `invokedynamic`. The new bytecode allows programmers to call a method just by name. Method resolution is implemented through custom, user defined lookup methods.

Soot [8] is one one of the most widely used frameworks for the static analysis and transformation of Java programs [6]. In this work we explain how we extended Soot to include support for reading, representing and writing `invokedynamic` bytecodes. This is a major change to Soot that modifies all its internal representations.

The remainder of this paper is structured as follows. Section 2 explains JSR 292 and the `invokedynamic` bytecode in more detail. Sections 3–6 then explain how we represent `invokedynamic` instructions in all the different intermediate representations that Soot supports. In Section 7 we explain the current limitations of our approach and some pitfalls that we came across. Section 9 explains related work and Section 10 concludes.

2. Invokedynamic as described in JSR292

During the past few years, people noticed that the built-in Java bytecodes do not provide sufficient flexibility when trying to implement highly dynamic and often untyped languages on top of the Java Virtual Machine, languages such as Ruby or Groovy. Such languages frequently allow dynamic redefinition of methods. For instance, consider the following pseudo-code example:

* This work was supported by the German Federal Ministry of Education and Research (BMBF) within EC SPRIDE and by the Hessian LOEWE excellence initiative within CASED.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SOAP'12 June 14, Beijing, China.
Copyright © 2012 ACM ISBN 978-1-4503-1490-9/12/06…$10.00

```
o.m = { return 1 ; }
print(o.m());    //prints 1
o.m = { return 2 ; }
print(o.m());    //prints 2
```

In this case, the same invoke expression `o.m` can bind to two completely different methods, defined as unnamed closures. Generating JVM compliant bytecode for such code snippets is everything but straightforward. Just for the above simple example, one would need to do all of the following. First one would need to extract both closures into typed, named methods, say `m1` and `m2`. Then one would need to generate a method `m()` that dispatches to `m1` or `m2` respectively, depending on the particular call site. But where should `m` itself be stored? The optimal location would be the runtime type of `o`, but this type is not known at compile time. As can be seen by this very simple example, the compilation of dynamic languages for the Java virtual machine raises many important design questions and is usually everything but straightforward.

JSR 292 [2] describes a new bytecode, invokedynamic, and a series of new constant-pool entries that, together, allow the execution of invokedynamic instructions. Unlike all other Java bytecode invoke instructions, the target method of an invokedynamic instruction is not defined by the language semantics, but rather by a user-defined method, called the bootstrap method.

The purpose of a bootstrap method is to associate an invokedynamic statement with a `CallSite` object. When an invokedynamic instruction executes, the virtual machine calls the call-site object's `getTarget()` method, which, as defined through user-written code, returns a `MethodHandle` object, i.e., a handle to the method to be invoked. Call sites may be constant or mutable, i.e., may or may not return the same method handle object on each call. The virtual machine can exploit knowledge about immutability for internal optimizations. Further, invokedynamic instruction can be associated with its own bootstrap method, or multiple instructions can share the same method.

A bootstrap method computes a method handle based on a method name associated with the invokedynamic call site. As opposed to normal Java method names, invokedynamic method names can be any Utf8 string. The following code shows an example bootstrap method for the closure-example from above:

```
CallSite bootstrap(Lookup caller, String
    name, MethodType type, Object... args) {
  if(name.equals("c1"))
    return caller.findStatic(
      AllMethods.class,"const1",type);
  if(name.equals("c2"))
    return caller.findStatic(
      AllMethods.class,"const2",type);
  ...
}
```

In this example, we assume that the first invokedynamic instruction uses `"c1"` as method name, and the second one uses `"c2"`. The bootstrap method retrieves appropriate method definitions by using a lookup object `caller`. This object is passed implicitly by the Java Virtual Machine. It represents the calling context with its lookup rules. In the example, all actual method definitions reside in a single class AllMethods. This is a common implementation strategy, for instance for JRuby. The virtual machine also implicitly passes a method-type argument. As shown, this method type can be used during method lookup.

In addition, the invokedynamic call site can provide the bootstrap method with a free number of additional static arguments, here shown as `args`. The arguments are called static because the bootstrap method is executed at load time. For the same reason, static arguments are restricted to the argument types that can be encoded in the constant pool; the static arguments must be constants.

Finally, a number of regular (dynamic) invocation arguments, which are used to call the method pointed to by the method handle that the call-site object's `getTarget()` method returns.

To summarize, an invokedynamic instruction includes:

- An unnamed type signature. (Used to allow modular static type checking, bytecode verification and implicit type conversions.)
- A number of regular (dynamic) method arguments. Those arguments will be passed to the method call after resolution.
- A reference to a bootstrap method which initializes the dynamic call site, effectively binding the site to a concrete method handle.
- An uninterpreted Utf8 string, which can be seen as the method name. This string is passed as an argument to the bootstrap method.
- An arbitrary number of additional static arguments. Those arguments are passed to the bootstrap method as well; they must be constants.

JSR 292 extensions to the Java bytecode format

The information is stored in the bytecode's constant pool in a rather complex way, using various constant pool entries and a new attribute, `BootstrapMethods`. Our extension to Soot's bytecode parser Coffi parses these entries to create Jimple expressions from invokedynamic bytecodes. The `BootstrapMethods` attribute is stripped during this process; it will not occur as a tag on the corresponding SootClass. Instead, the attribute is re-generated when Jimple is converted back to bytecode. (see Section 6)

Further information on the bytecode format can be found in the official documentation for the package `java.lang.invoke`.[1]

3. Jimple representation

Jimple is Soot's primary intermediate representation. Instructions are represented in a simple three-address code format, in which control flow only occurs through conditional and unconditional branches (as well as exceptions). In our extension to the Jimple IR, a invokedynamic bytecode is represented as a `JDynamicInvokeExpr`, whose constructor expects four arguments:

SootMethodRef bootstrapMethodRef A reference to a (static) bootstrap method. This method can reside in the same class as the invokedynamic instruction or in any other class. The bootstrap method receives three implicit arguments from the virtual machine, in addition to the explicit static arguments provided. A standard bootstrap method would start with the following argument signatures:

- **java.lang.invoke.MethodHandles$Lookup** A lookup object representing the current class context.
- **java.lang.String** The uninterpreted Utf8 string stated at the invokedynamic site.
- **java.lang.invoke.MethodType** A method-type object representing the resolved method type for this invokedynamic site.

List bootstrapArgs A number of static bootstrap arguments. Those arguments are passed to the bootstrap method in addition to the three implicit arguments mentioned above. The bootstrap method's signature must be defined accordingly. Those arguments must all be Jimple Constant objects.

SootMethodRef methodRef A method reference stating the name and type signature of this invokedynamic call site. The method name may be any valid Utf8 string. The sole purpose of this name is to be passed, by the virtual machine, as an argument to the provided bootstrap method. The type signature is used for type checking the invokedynamic call site in its invocation context. This is necessary to guarantee the integrity of the bytecode. In this signature, the name of the declaring class must be `soot.dummy.InvokeDynamic`. Soot will handle this special class as a phantom class (even if phantom classes are disabled otherwise.) The dummy class name is only used internally by Jimple as a placeholder; it is stripped when the `JDynamicInvokeExpression` is converted to bytecode, because invokedynamic instructions do not have a declaring class.

List methodArgs A number of regular, dynamic, method arguments in the form of Jimple Immediate objects.

Textual representation

Soot supports Jimple as a full-fledged source-code language, i.e., Soot can not only pretty-print the Jimple IR but can also parse `.jimple` files that contain such pretty printed code (or modifications thereof). This has special implications on how we chose to pretty-print invokedynamic instructions.

Figure 1 shows a pretty-print of a typical invokedynamic instruction in Jimple format (line breaks added to enhance readability). Here the string "+" following the keyword `dynamicinvoke` is the uninterpreted Utf8 string denoting the call site's method name. This string is printed as an actual quoted string constant so that it can be parsed again by the Jimple parser. (Again, note that any Utf8 string is allowed!) On the next line, the expression contains an unnamed method signature, stripped of any declaring class name and method name (because these were given before). In the above example, `r1` is the dynamic argument to this call; according to the signature, it may hold any kind of `Object`. The last three lines contain a regular staticinvoke expression, representing the bootstrap method and the static constant arguments to invoke it at load time.

The Jimple parser fully supports this syntax, which means that Soot users may directly write Jimple files in the above syntax to create invokedynamic bytecodes.

4. Grimp, Grimple and Shimple IRs

In addition to Jimple, Soot also features the Grimp and Grimple IRs, which contain aggregated Jimple expressions, and the Shimple IR for SSA form. For all those IRs, the representation of invokedynamic expressions is the same as in Jimple.

5. Baf IR

Before Jimple is converted to bytecode, it is first converted into the stack-based Baf IR. During this conversion, Soot generates Baf code to push the instruction's dynamic arguments onto the stack, followed by a `BDynamicInvokeInst` that holds the remainder of the information, i.e., the Utf8 method name, the reference to the bootstrap method and its static arguments. In analogy to the textual Jimple format, the textual Baf format consists of the two instructions shown in Figure 2.

Note that Baf is currently an output-only IR, i.e., there is no parser that could parse Baf instructions back into Soot.

6. Jasmin

The last step before the final conversion to bytecode is to convert Baf to Jasmin, a textual assembler for Java bytecode. Soot uses a fork of the Jasmin project; Jasmin is no longer maintained. Jasmin differs from Baf in that it is very close to the bytecode, e.g., uses the same format for class references.

[1] See `http://docs.oracle.com/javase/7/docs/api/java/lang/invoke/package-summary.html`

```
dynamicinvoke
"+"
<java.lang.Object (java.lang.Object)>(r1)
<jsr292.cookbook.binop.RT: java.lang.invoke.CallSite
   bootstrapOpLeft(java.lang.invoke.MethodHandles$Lookup,
                   java.lang.String,java.lang.invoke.MethodType,int)>(1)
```

Figure 1: Example of Jimple syntax of invokedynamic bytecodes

```
load.r r0;
dynamicinvoke
"+"
<java.lang.Object (java.lang.Object)>
<jsr292.cookbook.binop.RT: java.lang.invoke.CallSite
    bootstrapOpLeft(java.lang.invoke.MethodHandles$Lookup,
                    java.lang.String,java.lang.invoke.MethodType,int)>(1);
```

Figure 2: Previous example in Baf syntax

```
invokedynamic
"+"
(Ljava/lang/Object;)Ljava/lang/Object;
jsr292/cookbook/binop/RT/bootstrapOpLeft(Ljava/lang/invoke/MethodHandles$Lookup;
                                         Ljava/lang/String;Ljava/lang/invoke/MethodType;I)
                                            Ljava/lang/invoke/CallSite;
                                         ((I)1)
```

Figure 3: Previous example in the syntax of Soot's version of the Jasmin IR

We extended Jasmin to accept invokedynamic instructions in the syntax shown in Figure 3.

This is format quite similar to Baf. The only noteworthy difference is that the static arguments to the bootstrap method are all preceded by a bracketed type identifier, in the example `(I)`. This is necessary to allow Jasmin to determine which kind of class constant to generate as a bytecode representation of that constant value. In the above example, `I` would indicate an int.

7. Current limitations and pitfalls

The current support for JSR292 in our extension to Soot is fairly complete. However, a few items are missing. First, JSR292 allows constant arguments to bootstrap methods in the form of method handle or method type constants. Our extension to Soot does not currently support those constants; such support could be added. Adding the support would mostly complicate the Jimple and Jasmin parsers, which would need to cope with some form of textual representation of method handles in argument positions.

We do not currently support bytecode constants of type `CONSTANT_MethodType_info`. As of now, such support did not seem necessary. We plan to add appropriate support if the need arises.

Current pitfalls include the odd argument passing to bootstrap methods. Remember that a bootstrap method always receives three implicit arguments from the virtual machine. Those arguments are of type `Lookup`, `String`, and `MethodType`. This does not mean, however, that one can rely on the bootstrap method's signature to start with those argument types. Instead, a bootstrap method could consume any number of those arguments through a varargs argument, as in:

```
CallSite bootstrap(Object caller,
  Object... nameAndTypeWithArgs)
```

For this reason, Soot currently does not check the correctness of the bootstrap method's arguments against the provided arguments. Users hence have to be careful to provide a correct signature. Note that we *do* check, however, that the correct return type, `java.lang.invoke.CallSite`, is provided.

8. Using the invokedynamic support

We envision multiple use cases for our invokedynamic support in Soot. First, researchers can use Soot to conveniently generate invokedynamic instructions. Generating such instructions based on our Jimple representation should be much simpler than generating them using low-level bytecode manipulation libraries such as ASM [1].

Secondly, in the future we plan to add support for taking into account invokedynamic instructions during call-graph construction and points-to analysis. Such support could be added by at least two different means: through static analysis or through runtime monitoring. With a static analysis, one could try to simulate the effect of the bootstrap methods, hence attempting to at least resolve method handles already at compile time (or at least a coarse-grain approximation of those). Another approach would be to actually monitor dynamically which method handles a call site resolves to as the program executes. The monitoring code could write appropriate information into a log file and make that log file available to static analyses. In previous work, we have implemented a similar approach for Java's reflection API, in a tool called TamiFlex [3].

Both the static and dynamic approach would need to cater for special cases, though, in which a call site does not actually resolve to a method handle representing an actual Java method. The package `java.lang.invoke` supports the generation of special method handles for imaginative methods that do not actually exist in code.[2] Examples include methods returning constants, or methods inserting a dynamic guard. A static analysis would need to simulate the effects of such methods; a dynamic analysis would need to log such method handles appropriately.

9. Related Work

In previous work, we developed the TamiFlex system for handling reflective method calls during static analysis [3]. TamiFlex records dynamic information about such calls, to then make them available to static analyses implemented in Soot. We are currently extending TamiFlex to include support for invokedynamic instructions as well. In general, reflective method calls have much in common with invokedynamic instructions in the way in which static analyses can (or cannot) handle them. Invokedynamic instructions are more flexible, however, allowing for custom, yet type-safe, lookup.

ASM [1] has support for reading and writing invokedynamic bytecodes, but generally does not have any support for advanced data-flow analysis.

10. Conclusion

We have presented an extension to Soot that allows Soot to parse, represent and synthesize invokedynamic bytecodes. The extension modifies all of Soot's internal representation. Our extension will enable users of Soot to partially analyze or instrument invokedynamic instructions in the future. Also, our extension makes Soot the first tool that we know of that supports invokedynamic as a source-level construct: researchers can write invokedynamic in Jimple source code to them have Soot convert them to Java bytecode. Our extension is available with the main Soot distribution since version 2.5.0.

Extending Soot with invokedynamic support was not a trivial task. Problems arose with the Coffi frontend, which converts bytecode into Jimple, and with Jasmin, the assembler that converts Baf back into bytecode. Those two components of Soot deal with byte-level operations, are almost undocumented and not structured to be easily extensible. Further problems arose due to a lack of documentation of the bytecode layout chosen for invokedynamic bytecodes. Oracle's documentation on this issue is quite incomplete. The Jimple, Shimple and Baf IRs were quite straightforward to extend.

Acknowledgements Thanks to Matthias Perner for providing an initial implementation of invokedynamic based on the JDK 7 beta. Thanks to Andreas Sewe and Rémi Forax for answering some questions on MethodType attributes. Patrick Lam provided useful feedback on an initial draft of this paper. Thanks!

References

[1] ASM bytecode modification library. `http://asm.ow2.org/`.

[2] JSR 292: Supporting Dynamically Typed Languages on the Java^{TM} Platform. `http://jcp.org/en/jsr/detail?id=292`.

[3] Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati, and Mira Mezini. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *ICSE '11: International Conference on Software Engineering*, pages 241–250. ACM, May 2011.

[4] D. Flanagan and Y. Matsumoto. *The ruby programming language*. O'Reilly Media, 2008.

[5] D. Koenig, A. Glover, P. King, G. Laforge, and J. Skeet. *Groovy in action*. Manning Publications Co., 2007.

[6] Patrick Lam, Eric Bodden, Ondřej Lhoták, and Laurie Hendren. The Soot framework for Java program analysis: a retrospective. In *Cetus Users and Compiler Infastructure Workshop (CETUS 2011)*, October 2011.

[7] Tim Lindholm and Frank Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 1999.

[8] Vijay Sundaresan Patrick Lam Etienne Gagnon Raja Vallée-Rai, Laurie Hendren and Phong Co. Soot - a Java optimization framework. In *Proceedings of CASCON 1999*, pages 125–135, 1999.

[2] See `http://docs.oracle.com/javase/7/docs/api/java/lang/invoke/MethodHandles.html`