A Study of Call Graph Construction for JVM-Hosted Languages

Karim Ali, Xiaoni Lai, Zhaoyi Luo, Ondřej Lhoták, Julian Dolby, and Frank Tip

Abstract—Call graphs have many applications in software engineering, including bug-finding, security analysis, and code navigation in IDEs. However, the construction of call graphs requires significant investment in program analysis infrastructure. An increasing number of programming languages compile to the Java Virtual Machine (JVM), and program analysis frameworks such as WALA and SOOT support a broad range of program analysis algorithms by analyzing JVM bytecode. This approach has been shown to work well when applied to bytecode produced from Java code. In this paper, we show that it also works well for diverse other JVM-hosted languages: dynamically-typed functional Scheme, statically-typed object-oriented Scala, and polymorphic functional OCaml. Effectively, we get call graph construction for these languages for free, using existing analysis infrastructure for Java, with only minor challenges to soundness. This, in turn, suggests that bytecode-based analysis could serve as an implementation vehicle for bug-finding, security analysis, and IDE features for these languages. We present qualitative and quantitative analyses of the soundness and precision of call graphs constructed from JVM bytecodes for these languages, and also for Groovy, Clojure, Python, and Ruby. However, we also show that implementation details matter greatly. In particular, the JVM-hosted implementations of Groovy, Clojure, Python, and Ruby produce very unsound call graphs, due to the pervasive use of reflection, `invokedynamic` instructions, and run-time code generation. Interestingly, the dynamic translation schemes employed by these languages, which result in unsound static call graphs, tend to be correlated with poor performance at run time.

Index Terms—Call graphs, static analysis, JVM, compilation, Scheme, Scala, OCaml, Groovy, Clojure, Python, Ruby.

1 Introduction

The Java Virtual Machine (JVM) was designed for portable and efficient implementation of Java. By defining a relatively small set of bytecode instructions with clear semantics, the task of creating an interpreter or just-in-time compiler for Java is simplified significantly. In recent years, the JVM has been used to implement programming languages such as Clojure [1], Groovy [2], OCaml [3], Python [4], Ruby [5], Scala [6], and Scheme [7]. By compiling these languages to JVM bytecode, language implementors significantly reduce the amount of work needed to implement their languages, and achieve portability across many platforms.

Several frameworks, such as Chord [8], Doop [9], Soot [10], Wala [11], and OPAL [12], have been developed to support static analysis of JVM bytecode. These frameworks support a broad range of algorithms for static pointer analysis, call graph construction, data-flow analysis, and others. A JVM-based approach works well for Java because JVM bytecode is fairly close to Java but avoids a lot of its syntactic sugar. As a result, bytecode-based analysis frameworks are widely used in academia and industry.

Together, these developments raise the tantalizing possibility that many languages could get program analysis infrastructure for free by relying on analyzing generated JVM bytecode. For this to work, however, JVM-bytecode-based analysis has to produce good results, as it has been shown to do for Java. Hence, this paper investigates how well this JVM-bytecode-based approach works when applied to bytecode produced from other languages. We examine bytecodes generated from Scheme, Scala, OCaml, Groovy, Clojure, Python, and Ruby programs. We show that, while Scheme, Scala, and OCaml are a diverse set of languages, the compilers for these languages produce bytecode that by-and-large is amenable to analysis. However, implementation details matter greatly, and the other languages have complex, reflection-heavy implementations that inhibit good analysis.

Specifically, we focus on call graph construction because call graphs enable many applications in software engineering, such as bug-finding (see e.g. [13]), detecting security vulnerabilities (see e.g. [14]), IDE features such as code navigation (see e.g., [15]), and application extraction and optimization (see e.g. [16], [17]).

Our focus in this paper is on studying the following three issues: (i) the soundness of static call graphs computed from JVM bytecode (i.e., whether they contain all methods and call edges that can arise during any execution), (ii) the precision of the static call graphs (i.e., how many nodes and edges they contain that cannot arise in any execution), and (iii) the relationship between the quality of constructed call graphs and the runtime performance of applications.

To evaluate soundness and precision, we conduct qualitative and quantitative experiments. In the qualitative experiments, we inspect call graphs constructed by compiling

- *K. Ali is with the Department of Computing Science, University of Alberta. E-mail: karim.ali@ualberta.ca.*
- *X. Lai is with Google. E-mail: xlai@google.com.*
- *Z. Luo is with Microsoft. E-mail: zhaoyi.luo@microsoft.com.*
- *O. Lhoták is with the David R. Cheriton School of Computer Science, University of Waterloo. E-mail: olhotak@uwaterloo.ca.*
- *J. Dolby is with IBM Research. E-mail: dolby@us.ibm.com.*
- *F. Tip is with the Khoury College of Computer Sciences, Northeastern University. E-mail: f.tip@northeastern.edu.*

Manuscript received XXX XX, 2016; revised XXX XX, 2017.

a small example to bytecode, and study the translation of function and method calls. We look for uses of reflection, dynamic code generation, and `invokedynamic` instructions that challenge static analysis. In the quantitative experiments, we use 10 programs from the Computer Language Benchmark Game (CLBG) suite [18] with versions available for each language. This enables us to study the different programming languages in a uniform and consistent way. After compiling these programs to JVM bytecode, a standard 0-CFA analysis [19] provided by WALA [11] is used to construct static call graphs. Dynamic call graphs are constructed using WALA's instrumentation-based dynamic call graph builder. Using ProBe, a call graph comparison tool [20], we measure unsoundness by identifying nodes and edges that occur in the dynamic call graph but not the static one. Similarly, potential imprecision is reflected by nodes and edges in the static call graphs but not the dynamic ones. In such cases, we manually determine if the static analysis is imprecise, or if the discrepancy is due to low code coverage in the dynamic graph.

Since the CLBG programs are fairly small, we conducted, for each of the languages under consideration[1], additional experiments on two larger subject programs. For these programs, we performed the same quantitative experiments as for the CLBG programs. Moreover, in a detailed qualitative assessment, we determined whether additional issues arose in these larger programs that comprise soundness or precision.

We observe that call graphs constructed for Groovy, Clojure, Python, and Ruby using bytecode-based static analysis are unsound, due to pervasive use of reflection, dynamic code generation, and `invokedynamic` instructions. Even if these challenges were overcome, the constructed call graphs would remain highly imprecise due to the ways in which function calls are translated. On the other hand, sound call graphs are constructed for Kawa's implementation of Scheme, showing that dynamically-typed languages do not necessitate reflective, hard-to-analyze implementations. For statically-typed Scala and polymorphic OCaml, the use of reflection and proxies can cause unsoundness, just as in Java, but this occurs rarely in practice and similar solutions would apply [21]. Unsoundness of this kind is hard to avoid in practice [22], and reflects the state of the art. Bytecode-based analyses of these languages are as sound as for Java.

We observe that the call graphs constructed for Kawa programs are generally precise for direct calls, but some precision is lost with heavy use of lambda expressions. In Scala, precision suffers because type information is lost when compiling features such as closures. This is no different from the issues that lambda expressions face in Java 8, and can be addressed by standard forms of context sensitivity. The OCaml compiler implements closures using the JVM's `MethodHandles`, which WALA analyzes soundly and precisely.

For the performance experiments, we compiled the same 10 CLBG programs in each language and compared running times and memory consumption on a standard JVM. Interestingly, other than for Java, the lowest running times and memory consumption are measured for Scheme, Scala, and OCaml (i.e., the languages for which the constructed call graphs are the most sound). This suggests that the same translation schemes that hamper static analysis also cause performance bottlenecks.

Overall, we conclude that JVM-bytecode-based analysis is practical for a wide range of languages—with static, polymorphic and dynamic types, and with object-oriented and functional styles—but it requires careful implementation that avoids reflective features of Java as a substitute for compilation. This suggests that bytecode-based analysis could serve as a useful implementation vehicle for solving many problems in software engineering for languages for which alternative program analysis infrastructure is not readily available.

In summary, the contributions of this paper are as follows:

- We study soundness and precision of call graphs constructed from JVM bytecode produced from Scheme, Scala, OCaml, Groovy, Clojure, Python, and Ruby programs. To our knowledge, this is the first comparative study of static analysis for JVM-hosted languages.
- We show that for Kawa, Scala and OCaml, constructed call graphs are as sound as for Java, compromised only by reflection and proxies in rare cases. Precision also faces only issues similar to Java.
- We found that, for the languages Groovy, Clojure, Python, and Ruby, the constructed call graphs are highly unsound and imprecise. This is due to implementations that use reflection pervasively. Use of the new `invokedynamic` bytecode in some of these implementations leads to similarly poor results.
- Our performance experiments show that dynamic translation schemes that cause loss of soundness in static analysis are correlated with poor performance at run time.

The remainder of this paper is organized as follows. Section 2 provides some detail about MethodHandles and the `invokedynamic` JVM instruction. Section 3 reviews our experimental setup. Next, Sections 4–10 are concerned with an analysis of the soundness and precision of call graphs computed for each of the languages under consideration (Scheme, Scala, OCaml, Groovy, Clojure, Python, and Ruby, respectively). Section 11 reports on a study in which the performance of these language implementations is correlated with the observed soundness and precision results. In Section 12, some observations made during our studies are discussed, along with threats to validity. Related work is discussed in Section 13. Finally, conclusions are presented in Section 14.

2 Background

We briefly review MethodHandles and `invokedynamic` instructions, two features that were added to the JVM in Java 7 in order to facilitate the implementation of dynamic languages and that are already being used by several of the

1. Except in the case of Ruby, because the JRuby ahead-of-time build system is unable to handle the larger programs.

language implementations studied in this paper.[2] Since the static analysis community has not paid significant attention to these features until now, we briefly review the challenges they pose for static analysis.

2.1 MethodHandles

A method handle is a constant value that uniquely identifies a method and how it should be invoked (e.g., as a static call, or a virtual call). Furthermore, method handles can apply transformations to the sequence of arguments passed to the encapsulated method (e.g., unpacking an array into a sequence containing its values). Method handles can be embedded in a class file's constant pool as constants to be loaded using `ldc` instructions. A new type of constant pool entry, `CONSTANT_MethodHandle`, refers directly to an associated `CONSTANT_Methodref`, `CONSTANT_InterfaceMethodref`, or `CONSTANT_Fieldref` entry. Alternatively, method handles can be created at run time by calling one of the factory methods in class `java.lang.invoke.MethodHandles.Lookup` (e.g., `MethodHandles.Lookup.findVirtual`), with arguments specifying the encapsulated method's parameter types and return type. The method encapsulated by a method handle can be invoked by calling the `MethodHandle.invoke()` or `MethodHandle.invokeExact()` method, with arguments that should be bound to the method's receiver (in the case of virtual methods) and formal parameters. In effect, the functionality provided by method handles is similar to that of the Java reflection API, but access checking is performed only once, upon creation of the handle, whereas `java.lang.reflect.Method.invoke()` performs an access check for each reflective call. From a static analysis perspective, by denoting a method explicitly, method handles enable more precise analysis than what was possible using the reflective idioms required before Java 7.

2.2 The `invokedynamic` Instruction

The `invokedynamic` instruction provides a mechanism for deferring the association between call sites and the methods they invoke until runtime. It works as follows:

- When an `invokedynamic` instruction executes for the first time, its associated *bootstrap method* is executed. The association between `invokedynamic` instructions and their associated bootstrap methods is recorded in the *bootstrap table*, a new component of JVM `.class` files.
- A bootstrap method returns a `java.lang.invoke.CallSite` object that encapsulates a `MethodHandle` that identifies the method to be invoked. This method can be retrieved using the `CallSite.getTarget()` method, which is automatically invoked by the JVM at run time.
- The `CallSite` object returned by a bootstrap method is cached, so that for subsequent executions of an `invokedynamic` instruction, the JVM only needs to retrieve the method handle by executing `CallSite.getTarget()`.

This call resolution mechanism is considerably more flexible than other JVM invoke instructions. In particular, `invokevirtual` and `invokeinterface` specify a target method, and calls made through them dispatch to methods transitively overriding this target method. In such cases, the name and parameter types of the method are known at compile time so that a static analysis can analyze the inheritance hierarchy to conservatively approximate the set of methods that may be invoked. In the case of `invokedynamic`, there is no obvious way for a static analysis to approximate the set of possible call targets. Bootstrap methods can be arbitrarily complex, and there are no compile-time constraints on the name and parameter types of the subsequently invoked method. Further complicating matters, `CallSite` objects returned by bootstrap methods may be *mutable* (i.e., encapsulated method handles may be updated at runtime). However, if the `CallSite` object is immutable and the bootstrap method is the standard Java `java.lang.invoke.LambdaMetaFactory`, it is possible for an analysis to pre-process `invokedynamic` instructions by rewriting them using standard `invokestatic` instructions that simulate the semantics of the original method calls that use `invokedynamic`. Alternatively, the analysis would have to contain hard-coded models fo the bootstrap methods to reason about `invokedynamic`. Such approach is currently used in practice by some analysis frameworks such as OPAL [23], WALA [11], and Soot [24] to overcome the challenges around precise and sound analysis of `invokedynamic` instructions.

3 Experimental Setup

3.1 Analysis Details

For all the languages we study in this article, we use WALA's implementation of the 0-CFA algorithm [19] for constructing static call graphs, and WALA's instrumentation-based dynamic call graph builder (Shrike) for constructing dynamic call graphs. We used the same configuration for the static and dynamic call graph analyses across all languages.

We conducted all of our experiments using Oracle's Java 8u25 running on a machine with eight dual-core AMD Opteron 1.4 GHz CPUs (running in 64-bit mode) and capped the available RAM at 16 GB.

3.2 Comparing Call Graphs

We use ProBe [20], a call graph comparison tool, to identify nodes/edges that occur in the static call graphs but not the dynamic ones and vice versa, to find unsoundness and loss of precision. Comparing static and dynamic call graphs is complicated by the fact that static call graph builders, such as WALA's, analyze the Java runtime libraries to compute sound and precise call graphs with nodes and edges corresponding to library code. Instrumentation-based dynamic call graph builders, such as WALA's, typically do not track calls inside the runtime, in order to avoid disrupting runtime mechanisms invisible at the source level. Therefore, in the call graphs that WALA constructs, code in the Java standard library is represented by a single node. In order to enable a fair comparison, we collapse the parts of the static call graphs that correspond to code in the Java standard library into a single node as well. Some additional effort was involved in handling static initializers and finalizers

2. Note that, starting with Java 8, the bytecodes generated from Java programs also make use of `invokedynamic` when lambda expressions (closures) are being compiled. Thus, the analysis challenges noted here are broadly applicable to statically-typed and dynamically-typed languages.

consistently in static and dynamic call graphs, by modeling these as if they are invoked from an anonymous "root" node.

3.3 Selection of Experimental Subject Programs

In our study, we consider the Scheme, Scala, OCaml, Groovy, Clojure, Python, and Ruby programming languages and investigate whether the JVM bytecodes generated by compilers these languages are amenable to static analysis. This investigation comprises the following three steps:

1) For each programming language under consideration, we first consider a simple "hello, world" program to illustrate the challenges that are likely to arise when trying to analyze *any* program in that language. As we shall see, for several of the programming languages, the study of such a simple example is already sufficient to conclude that sound and precise static analysis is infeasible (e.g., due to pervasive use of reflection or dynamically generated code).
2) Next, we conduct a systematic evaluation in which we take 10 programs from the Computer Language Benchmark Game (CLBG) suite [18] for which versions are available for each language under consideration. The use of the same benchmark suite for all programs enables us to study the programming languages in a uniform and consistent way, and it enables us to investigate whether correlations exist between the performance characteristics of the generated JVM bytecodes and the suitability of those bytecodes for static analysis. Section 3.4 provides further details on the CLBG programs under study.
3) Lastly, for each programming language under consideration (with the exception of JRuby, because we were unable to find two large Ruby programs that we could successfully compile with JRuby's ahead-of-time build system), we selected two larger programs in an attempt to determine whether any additional issues arise due to the use of features not present in the CLBG programs.

3.4 CLBG Benchmark Suite

For the second part of our study, we consider the following 10 programs[3] from the Computer Language Benchmark Game (CLBG) suite for which versions are available in all languages under consideration.

- BINARYTREES (BT) prints the time required to allocate and collect balanced binary trees of various sizes, before any tree nodes are garbage-collected.
- FANNKUCHREDUX (FK) simulates indexed-accesses to tiny integer sequences.
- FASTA (FA) generates and writes random DNA sequences.
- KNUCLEOTIDE (KN) uses the built-in hash table implementation to accumulate count values for k-nucleotide strings, lookup the count for a given string, and update the count in the hash table.
- MANDELBROT (MB) generates a Mandelbrot set portable bitmap file.

3. We did not use the other CLBG benchmarks as they are not all implemented in the JVM-hosted languages under study.

```
1 (define (bar x y)
2   (display x)
3   (display y) (newline))
4 (define (foo x y)
5   (x y))
6 (foo (lambda (y) (bar "Hello, " y)) "World!")
```

Fig. 1. A simple Scheme program.

- NBODY (NB) models the orbits of Jovian planets using double-precision N-body simulation.
- PIDIGITS (PD) streams arbitrary-precision arithmetic for the decimal value of π.
- REGEXDNA (RD) matches DNA 8-mers and substitutes magic patterns.
- REVCOMP (RC) reads DNA sequences, and writes their reverse complement.
- SPECTRALNORM (SN) is a program that calculates Eigenvalues using the power method.

4 SCHEME

Scheme [7] is a dialect of the functional programming language Lisp. Scheme is dynamically typed, and has a simple syntax based on lists and prefix operators. Its distinguishing features include lexical scoping and higher-order functions. Kawa [25] implements an extension of Scheme that runs on the JVM and interoperates well with Java libraries. In our experiments, we used Kawa version 3.0.

4.1 Translation to JVM bytecode

Figure 1 shows a Scheme "Hello, World!" program that defines functions `foo` and `bar`. On line 6, `foo` is invoked with two arguments: a function and a string. Inside function `foo`, this function is invoked with the string as its argument. The body of the anonymous function on line 6 invokes function `bar` on the string constant "Hello, " and its second argument, so the call to `bar` will result in printing "Hello, World!".

Figure 2 shows the relevant fragments of the JVM bytecodes produced by the Kawa compiler for the program of Figure 1. In this diagram and subsequent diagrams, solid arrows represent a single call graph edge (i.e., a situation where a method call dispatches to a specific method definition), and dashed arrows represent sequences of call edges. The Kawa compiler produces a single class `hello` containing static methods `foo()` and `bar()` corresponding directly to the two functions in Figure 1. The class also contains a static initializer that initializes the environment, and a `run()` method corresponding to the top-level code. A `main()` method starts execution by invoking `runAsMain()` (indicated by the arrow labeled ① in the figure), which invokes `run()` via a library callback (see the arrow labeled ②).

For simple function calls, a one-to-one mapping exists between function calls in the Scheme source code and `invokestatic` calls in the JVM bytecode produced by the Kawa compiler. For example, the call to `foo` on line 6 in Figure 1 is reflected by the static method call labeled ③ in Figure 2.

The translation of higher-order functions is more involved. The lambda expression on line 6 is represented

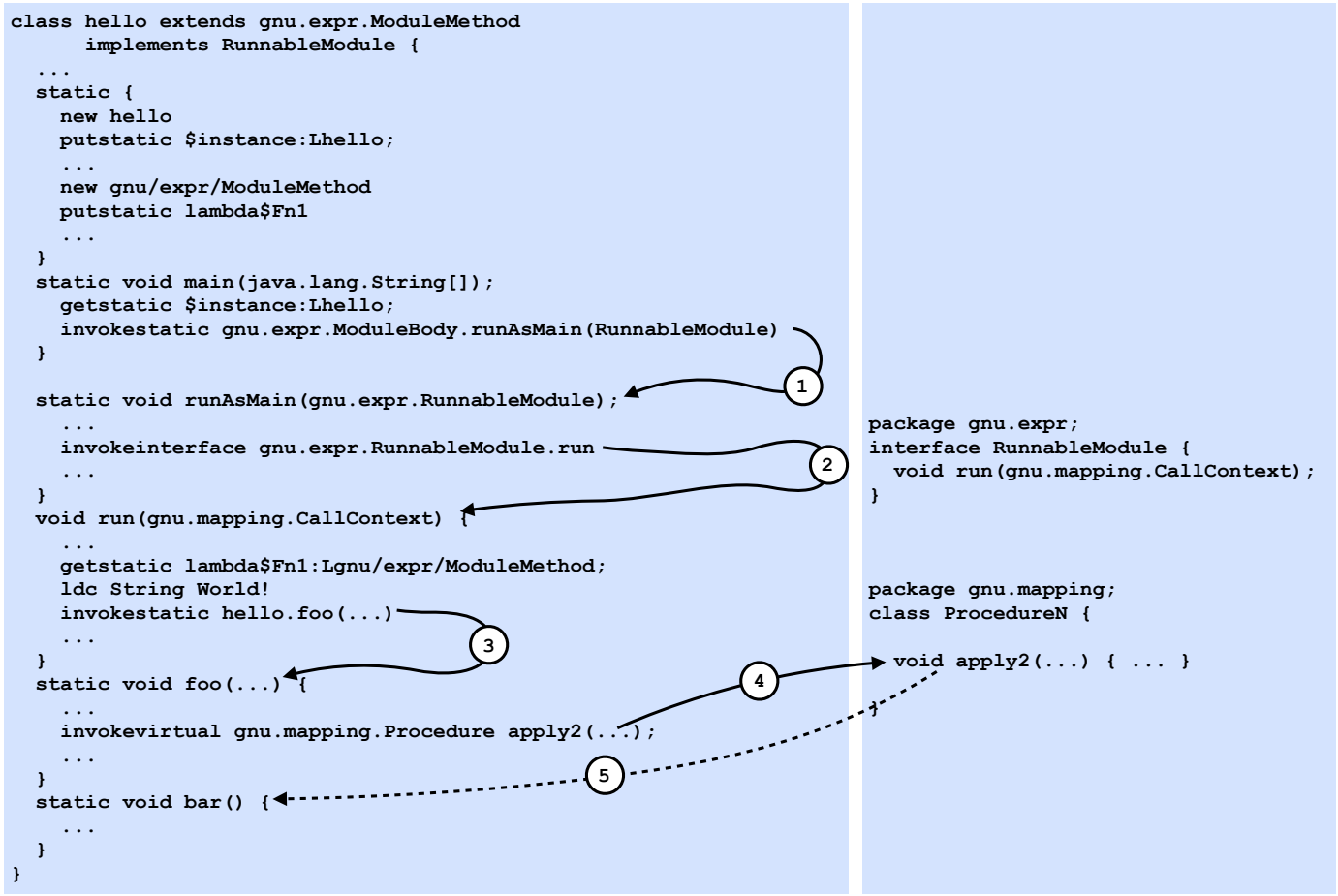


Fig. 2. Depiction of the bytecodes produced by the Kawa compiler for the program of Figure 1.

using an object of type `gnu.expr.ModuleMethod`. The static initializer of class `hello` creates this object and stores it into a field that is later read by the `run()` method and passed as an argument to `foo()`, which calls a library method `gnu.mapping.ProcedureN.apply2()` to call the function bound to the argument (see the edge labeled ④). This library method calls several other library methods, which eventually call `hello.bar()` (see the arrow labeled ⑤). In other words, the translation of higher-order functions involves lengthy sequences of calls through common library functions, which implies there is potential for significant loss of precision unless many levels of context-sensitivity are employed. However, we did not observe any use of reflection or `invokedynamic` in the entire sequence of calls that would compromise the soundness of a bytecode-based static analysis.

4.2 Qualitative Analysis

Since the Kawa compiler avoids the use of reflection and `invokedynamic`, the call graph constructed by WALA for the example program of Figure 1 is sound.

4.3 Quantitative Analysis

Table 1 shows, for each program in our benchmark suite (excluding library code), the number of lines of source code, as well as the size of the generated bytecodes, the number of classes, methods, and call sites in the generated bytecodes.

TABLE 1
Various characteristics of the Scheme CLBG programs.

	LOC	Bytecode Size (KB)	# classes	# methods	# call sites
BT	36	3.05	2	7	42
FK	106	2.71	1	4	28
FA	90	4.25	1	8	107
KN	217	4.51	1	10	97
MB	58	2.33	1	4	20
NB	116	4.05	2	13	37
PD	88	4.29	3	17	69
RD	51	3.18	1	4	65
RC	66	2.30	1	4	14
SN	39	2.29	1	8	24

Table 2 shows the number of nodes and edges in the static (columns `S`) and dynamic (columns `D`) call graphs for the Scheme programs in our benchmark suite. Also shown are the number of nodes/edges in the dynamic call graphs but not in the static call graphs (columns `D\S`), and those in the static call graphs but not in the dynamic call graphs (columns `S\D`).[4] As can be seen from the `D\S` columns of the

4. The tables in Sections 5–10 follow the same structure.

TABLE 2
Count of nodes and edges in the static and dynamic call graphs of the Scheme CLBG programs.

	Nodes				Edges			
	S	D	D\S	S\D	S	D	D\S	S\D
BT	9,066	501	0	8,565	49,497	686	0	48,811
FK	9,063	503	0	8,560	49,492	680	0	48,812
FA	9,067	511	0	8,556	49,525	704	0	48,821
KN	9,070	707	0	8,363	49,547	1,063	0	48,484
MB	9,063	460	0	8,603	49,487	617	0	48,870
NB	9,072	564	0	8,508	49,506	790	0	48,716
PD	9,076	509	0	8,567	49,518	690	0	48,828
RD	9,063	528	0	8,535	49,483	734	0	48,749
RC	9,063	286	0	8,777	49,480	378	0	49,102
SN	9,067	552	0	8,515	49,493	766	0	48,727

TABLE 3
Various characteristics of the CHESS and JEMACS Scheme programs.

	LOC	Bytecode Size (MB)	# classes	# methods	# call sites
CHESS	582	0.08	9	385	2,428
JEMACS	4,883	1.63	825	8,492	49,441

table, all nodes/edges in the dynamic call graphs also occur in the static call graphs, suggesting that the call graphs are sound.

At first glance the computed static call graphs appear to be quite imprecise because many nodes/edges in the static call graphs do not occur in the dynamic call graphs (see columns `S\D`). However, upon closer inspection we found nearly all of this imprecision to occur in parts of the static call graph corresponding to code in the Kawa libraries. The parts of the call graph corresponding to the application itself are generally precise for direct calls, though some loss of precision occurs if lambda expressions are used as in the example program discussed previously. In summary, for the programs under consideration, Kawa generates bytecode that is easily amenable to static analysis. This is in stark contrast, as we will show later, to the highly unsound static call graphs for the other dynamically-typed languages (Groovy, Clojure, Python, and Ruby).

4.4 Additional Case Studies

In addition to the Scheme CLBG programs, we have studied CHESS and JEMACS[5], two larger Scheme applications that

5. Sources are available from https://github.com/ttu-fpclub/kawa-chess and http://jemacs.sourceforge.net.

TABLE 4
Count of nodes and edges in the static and dynamic call graphs of the CHESS and JEMACS Scheme programs.

	Nodes				Edges			
	S	D	D\S	S\D	S	D	D\S	S\D
CHESS	12,297	687	0	11,610	77,981	1,293	17	76,705
JEMACS	8,601	1,313	8	7,296	64,148	2,495	28	61,681

```
7  (define-syntax do-board
8    (syntax-rules ()
9      ((_ (var  final ) body  ...)
10      (do ((i  0 (+ i  1)))
11          ((>= i  8)  final )
12        (do ((j  0 (+ j  1)))
13            ((>= j  8))
14          (let  ((var  (position i  j)))
15            body  ...) )))))
16
17 (define  ( initialize-board )
18   (do-board (pos '())
19             (let  ((col  (apply starting-color pos))
20                    (pie  (apply starting-piece pos)))
21               (array-set! *board* (yvalue pos) (xvalue pos)
22                           (apply starting-status pos)))))
```

Fig. 3. A Scheme program necessitating reflection.

are publicly available on GitHub and SourceForge, respectively. CHESS is a Swing-based program for playing chess, in which two people play against each other. JEMACS is a Java/Scheme-based Emacs text editor. Both CHESS and JEMACS are based on Kawa 3.0, which implements Scheme R7RS. Table 3 shows, for both programs (excluding library code), the number of lines of source code, as well as the size of the generated bytecodes, the number of classes, methods, and call sites in the generated bytecodes.

Table 4 shows that only 8 nodes are missing from the static call graph for JEMACS compared to its dynamic call graph. Further investigation of the call graphs shows that all 8 nodes represent methods that are called from `gnu.mapping.Future.run()`, which handles threads in the Kawa runtime library. While WALA supports Java threads, it currently does not support Kawa-specific threads. Those nodes are the root cause of missing 28 edges in the static call graph for JEMACS compared to its dynamic call graph. For CHESS, the static call graph is sound with respect to nodes, but misses 27 edges compared to the dynamic call graph. However, all missing edges are within the Kawa runtime library, which still renders the produced static call graphs useful for various applications such as code navigation in development environments.

The precision of the larger Scheme programs is similar to the pattern in the CLBG programs, where precision is compromised largely by lack of sufficient context-sensitivity in handling indirections through the standard library. However, there is one issue that we did not encounter in CLBG. The Kawa implementation uses reflection to handle first-class function calls that use `apply` in Scheme. Fig. 3 shows an example of that idiom from CHESS, in which `initialize-board` calls several functions using `apply`: `starting-color`, `starting-piece`, and `starting-status`. The `initialize-board` function is itself defined using the `do-board` macro. Kawa handles `syntax-rules` in `do-board` and generates analyzable code for it. However, `apply` uses functions from a table filled in using calls to `MethodHandle.Lookup.findStatic()`. The arguments to `findStatic()` are a `java.lang.Class` denoting the method's class, a `java.lang.String` denoting the method name, and a `java.lang.invoke.MethodType` denoting the argument and return types. The class objects are specified directly in the bytecode, and WALA can track

```
23 object hello {
24   def main(args: Array[String]) = {
25     (new T with A).bar
26     val identity : Object=>Object = { x => x }
27     println ( identity ("Hello, World!"))
28   }
29 }
30 trait A
31 trait T {
32   def bar = println ("T.bar")
33 }
```

Fig. 4. A simple Scala program.

them precisely in this case. The method name string is also specified directly, and WALA statically tracks that too to the call site. Although the method type is ignored, WALA has sufficient analysis of reflection to find all methods with the right name. Since these are Scheme functions, there is exactly one such method. Therefore, WALA finds precisely the methods to be called, and the call graph remains sound despite this use of reflection. Precision is lost in this case, though, because all such methods are stored in tables in the heap.

5 SCALA

Scala [6] is a statically-typed, object-oriented, functional programming language. Its functional programming idioms include pattern matching, lazy evaluation, and closures. In Scala, method and field definitions can be grouped into *traits* that can be mixed into classes. The Scala compiler compiles to JVM bytecode. Our experiments used Scala 2.10.2.

5.1 Translation to JVM bytecode

Figure 4 shows a Scala program that defines traits `A` and `T`. The `main` method calls `bar` on the trait composition `(T with A)`. It then defines a closure `identity` and calls it, causing "Hello, World" to be printed.

Figure 5 shows the revelant bytecode instructions produced by the Scala compiler for the program of Figure 4. A Scala trait, such as `T` in our example, is translated into two JVM class files: `T` and `T$class`. Interface `T` contains the declaration for the method `bar` of the trait `T`. The abstract class `T$class` defines a static method containing the bytecode translation of the body of `bar`. Finally, a trait composition such as `(T with A)` is translated into an anonymous class `hello$$anon$1` that implements all its traits. The call to `bar` on line 25 corresponds to two method calls in the generated bytecode: an `invokevirtual` to `hello$$anon$1.bar` (reflected by the arrow labeled ① in Figure 5), which contains an `invokestatic` to the actual implementation in `T$class.bar` (see the arrow labeled ②).

A Scala closure such as `identity` is translated into an anonymous class `hello$$anonfun$1` that extends `scala.runtime.AbstractFunction1` that in turn implements `scala.Function1` in which an abstract method named `apply` is defined. The anonymous class `hello$$anonfun$1` implements the body of the closure in a concrete implementation of this `apply` method. The call to the `identity` closure on line 27 corresponds to an `invokeinterface` call to `Function1.apply`, which is resolved at run time to the `apply` method of the `hello$$anonfun$1` class (see the arrow labeled ③ in Figure 5).

For certain Scala features (e.g., mutable fields in anonymous classes), the Scala compiler generates JVM bytecodes containing reflective method calls, which challenges sound static analysis. A sound static analysis would have to make conservative approximations that cause the static call graph to become extremely large and imprecise.

TABLE 5
Various characteristics of the Scala CLBG programs.

	LOC	Bytecode Size (KB)	# classes	# methods	# call sites
BT	27	3.57	4	9	43
FK	45	10.85	9	52	151
FA	95	7.69	6	35	91
KN	51	38.00	27	245	606
MB	35	10.44	10	51	112
NB	62	10.22	9	43	173
PD	87	16.18	12	64	203
RD	30	7.57	7	23	86
RC	22	7.59	5	37	79
SN	30	6.97	8	28	57

TABLE 6
Count of nodes and edges in the static and dynamic call graphs of the Scala CLBG programs.

	Nodes				Edges			
	S	D	D\S	S\D	S	D	D\S	S\D
BT	788	515	0	273	1,337	647	0	690
FK	1,315	799	0	516	2,533	1,106	0	1,427
FA	1,011	598	0	413	1,937	808	0	1,129
KN	2,605	1,501	0	1,104	5,985	2,421	0	3,564
MB	993	578	0	415	1,888	771	0	1,117
NB	966	639	0	327	1,846	875	0	971
PD	1,245	690	0	555	2,743	1,008	0	1,735
RD	983	590	0	393	1,711	771	0	940
RC	849	530	0	319	1,488	673	0	815
SN	1,064	616	0	448	1,989	809	0	1,180

5.2 Qualitative Analysis

For the example of Figure 4, no significant challenges for static analysis are evident and a sound call graph is constructed. Nevertheless, as reported by Ali et al. [26], analyzing JVM bytecodes generated by the Scala compiler can result in less precise call graphs than those constructed from Scala source code. This loss of precision occurs because significant type information is lost in the process of translating certain Scala features (e.g., closures) to JVM bytecode.

The Scala compiler translates each call to a closure to an `invokeinterface` to the `apply` method of `scala.FunctionN`, where `N` is the arity of the closure. Therefore, a bytecode-based static call graph analysis will create edges to the `apply` methods of all subclasses of `scala.FunctionN` *from each of the call sites* to `scala.FunctionN.apply()`, thus rendering

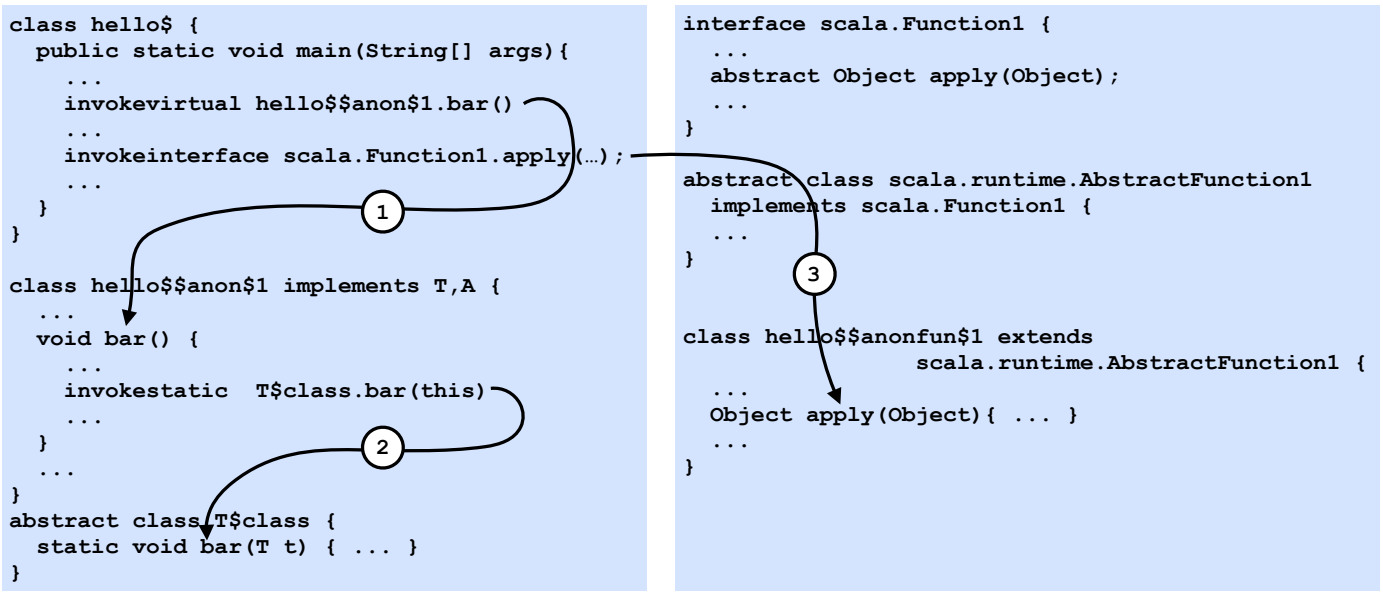


Fig. 5. Depiction of the bytecodes produced by the Scala compiler for the program of Figure 4.

the produced static call graphs extremely imprecise[6]. Ali et al. [26] present a family of algorithms for constructing call graphs of Scala programs from source code that avoids this loss of precision by taking advantage of the type parameters of `scala.FunctionN`. Those types correspond to the parameter and return types of the closure, but are erased when bytecode is generated and are therefore unavailable to a bytecode-based analysis.

5.3 Quantitative Analysis

Table 5 shows, for each program in our benchmark suite (excluding library code), the number of lines of source code, as well as the size of the generated bytecodes, the number of classes, methods, and call sites in the generated bytecodes.

For each of the benchmark programs in Table 6, the nodes and edges in the dynamic call graphs are subsets of those in the corresponding static call graphs, so they are sound for this execution. However, for FASTAREDUX, one of the CLBG programs that we did not include in our quantitative experiments[7], we noticed 2 methods and 2 call edges that are missing in the static call graph compared to the dynamic call graphs. Further investigation revealed that this unsoundness arises from the use of reflection in the bytecodes generated by the Scala compiler for converting collections into arrays, similar to what the Java method `java.util.ArrayList.toArray(T[])` does. When we examined the precision of the static call graphs, we found that on average, about 15% of the edges that are in the static call graphs but not in the dynamic call graphs involve calls to/from `apply()` methods.

TABLE 7
Various characteristics of the FACTORIE and KIAMA Scala programs.

	LOC	Bytecode Size (MB)	# classes	# methods	# call sites
FACTORIE	35,428	8.95	6,401	64,222	115,807
KIAMA	17,914	6.67	5,143	44,847	79,071

TABLE 8
Count of nodes and edges in the static and dynamic call graphs of the FACTORIE and KIAMA Scala programs.

	Nodes				Edges			
	S	D	D\S	S\D	S	D	D\S	S\D
FACTORIE	5,796	1,202	0	4,594	17,548	2,244	226	15,530
KIAMA	4,922	1,193	0	3,729	18,952	2,186	408	17,174

5.4 Additional Case Studies

In addition to the Scala CLBG programs, we have studied FACTORIE and KIAMA[8], the largest two Scala applications from the DaCapo Scala Benchmarking project [27]. FACTORIE is a toolkit for probabilistic modeling. It provides its users with a language for creating relational factor graphs, estimating parameters and performing inference. KIAMA is a library for language processing used to compile and execute several small languages. Table 7 shows, for both programs (excluding library code), the number of lines of source code, as well as the size of the generated bytecodes, the number of classes, methods, and call sites in the generated bytecodes.

For FACTORIE and KIAMA in Table 8, the static call graphs are not missing any nodes compared to the dynamic call graphs, but are missing some

6. Unless the analysis maintains multiple levels of (call-string) context-sensitivity, which would be prohibitively expensive.

7. We did not include FASTAREDUX in our study because versions of this program are not available for Scheme, OCaml, Groovy, and Python.

8. Sources are available from https://github.com/factorie/factorie and https://bitbucket.org/inkytonik/kiama.

```
34 let bar x y =
35         print_string x;
36         print_string y;
37         print_string "done\n";;
38 let foo x y =
39         print_string "starting\n";
40         x y;;
41 foo (bar "Hello, ") "World!\n";;
```

Fig. 6. A simple OCaml program.

edges (FACTORIE: 226 edges and KIAMA: 408 edges). The missing edges mainly involve reflective Scala constructs such as `scala.reflect.ManifestFactory` and `scala.reflect.ClassTag`. With respect to precision, more than 20% of the edges (FACTORIE: 22.56% and KIAMA: 23.76%) that are in the static call graphs but not in the dynamic call graphs involve calls to/from `apply()` methods. This result shows that the effect of losing type information on the static call graph precision is worse for analyzing real-world applications, such as FACTORIE and KIAMA, compared to the Scala CLBG benchmark programs.

5.5 Recent Developments

There were no significant changes to code generation in the 2.10 and 2.11 series of Scala releases. Scala 2.12 and 2.13 use a new code generation back-end that targets features introduced in version 8 of the Java Virtual Machine. In particular, these versions of Scala compile closures in the same way as Java 8, using the `invokedynamic` instruction, as discussed in Section 2. This eliminates the need for the compiler to generate a separate class (like `hello$$anonfun$1` in Figure 4) for every closure in the program. In addition, these versions of Scala compile traits to Java interfaces and implement methods using the default interface methods that were introduced in Java 8. This eliminates the need for a class to implement the methods of each trait (like `T$class` in Figure 4). Dotty, the compiler that will be used for Scala 3 when it is released, also uses these new code generation techniques.

6 OCaml

OCaml is a general-purpose programming language supporting functional, imperative and object-oriented styles. Types are strong, static, and inferred by the compiler. OCaml-Java [28] compiles OCaml code to JVM bytecode. We used OCaml-Java 2.0-alpha2, based on OCaml 4.01.0.

6.1 Translation to JVM bytecode

Figure 6 shows an OCaml program that declares functions `foo` and `bar`. This program illustrates currying, in the partial call to `bar` with one argument "`Hello, `". This closure is passed to `foo` along with the argument "`World\n`". Function `foo` calls its argument `x` (which is bound to `bar`) with `y` (bound to "`World\n`") as a parameter. Function `bar` prints both its arguments, resulting in the expected "Hello, World!" output.

Figure 7 visualizes the bytecodes produced by the OCaml compiler for the program of Figure 6. The

TABLE 9
Various characteristics of the OCaml CLBG benchmarks.

	LOC	Bytecode Size (KB)	# classes	# methods	# call sites
BT	45	10.87	4	101	438
FK	83	10.71	4	101	390
FA	124	13.08	4	101	594
KN	43	13.17	4	101	626
MB	30	9.83	4	101	271
NB	120	11.61	4	101	599
PD	63	12.95	4	101	723
RD	29	11.68	4	101	558
RC	53	10.22	4	101	353
SN	40	10.28	4	101	358

TABLE 10
Count of nodes and edges in the static and dynamic call graphs of the OCaml CLBG programs.

	Nodes				Edges			
	S	D	D\S	S\D	S	D	D\S	S\D
BT	5,963	532	0	5,431	75,056	1,464	64	73,656
FK	5,973	544	0	5,429	74,883	1,487	70	73,466
FA	5,966	349	0	5,617	75,363	1,165	8	74,206
KN	5,981	586	0	5,395	77,949	1,793	89	76,245
MB	5,958	527	0	5,431	74,832	1,417	61	73,476
NB	6,977	526	0	6,451	88,592	1,468	61	87,185
PD	5,986	424	0	5,562	76,230	1,413	25	74,842
RD	5,962	696	0	5,266	75,056	2,074	102	73,084
RC	5,960	345	0	5,615	75,133	1,166	8	73,975
SN	5,963	532	0	5,431	75,028	1,455	61	73,634

OCaml Java runtime compiles the `Hello` class to extend `AbstractNativeRunner`, which provides the machinery to invoke `moduleMain` in a threading harness (see the dashed arrow labeled ①). The top-level code from Figure 6 is translated to method `entry()`, which is called from `moduleMain()` (see the call edge labeled ②).

OCaml-Java translates functions into methods, and direct function calls into `invokestatic` calls. For instance, the call to `foo` at line 41 is represented by the edge labeled ③. Currying is translated by constructing a closure object using `org.ocamljava.runtime.values.Value.createClosure()`, in combination with an extra function object. Crucially, first-class functions are named explicitly using `MethodHandles` and stored as bytecode constants. In the example of Figure 6, the partial call to `bar` at line 41 is translated by calling `createClosure()`, calling `setClosure()` on it with a `MethodHandle` representing `bar()`, and then recording the closure parameter "`World!\n`" by calling `set2()` on the closure. Then, the closure itself is invoked (see the edge labeled ④), which invokes `bar` using `MethodHandle.invokeExact()` (see edge ⑤), avoiding string-based reflection.

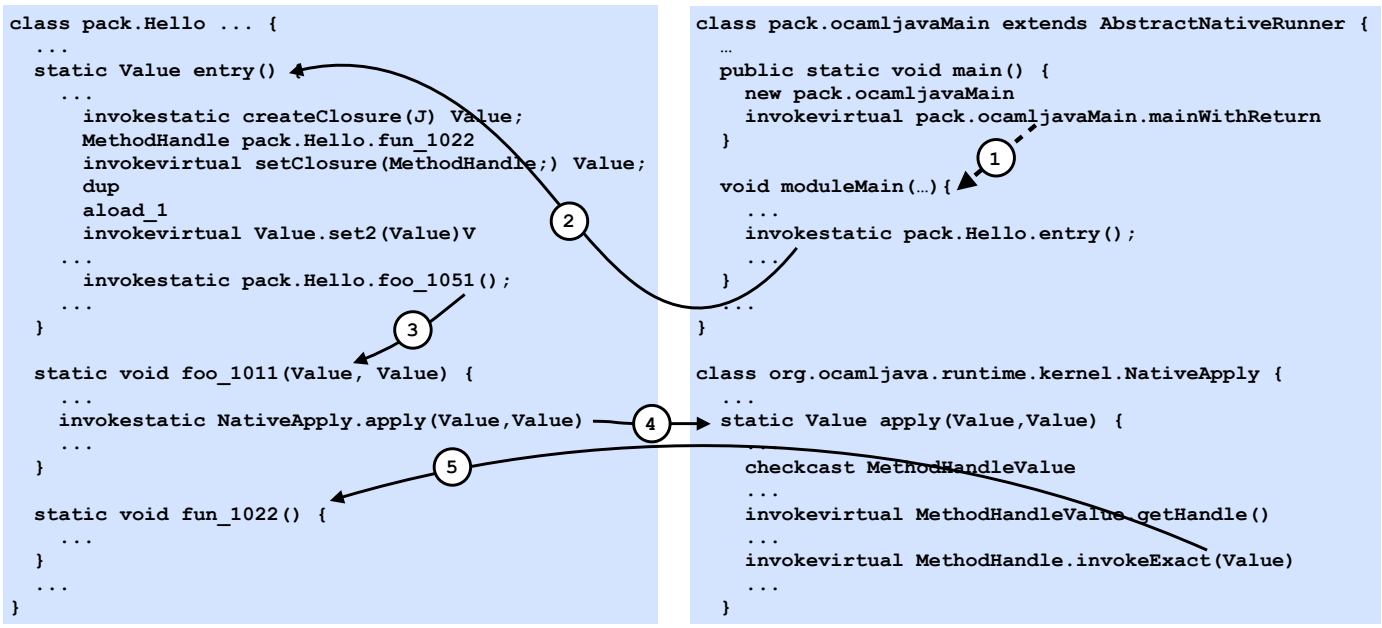


Fig. 7. Depiction of the bytecodes produced by the OCaml compiler for the program of Figure 6.

6.2 Qualitative Analysis

OCaml-Java exploits `MethodHandles` to great effect, making heavy use of constant `MethodHandles` embedded in the bytecode, and avoiding string-based reflection. Thus, first-class functions manifest as explicit method constants; WALA models these constants and invocations on them. Hence, functions such as `bar` appear in the call graph. This does not, in itself, make an analysis precise. Functions passed as arguments may cause imprecision in a context-insensitive analysis, just as dynamic dispatch on parameters can in object-oriented languages. However, this is the same well-studied problem of context-sensitivity that has inspired so many techniques for object-oriented languages [9], [29].

6.3 Quantitative Analysis

Table 9 shows, for each program in our benchmark suite (excluding library code), the number of lines of source code, as well as the size of the generated bytecodes, the number of classes, methods, and call sites in the generated bytecodes.

Table 10 presents quantitative results for the OCaml CLBG programs. The impact of using `MethodHandles` in a way that is amenable to static analysis is apparent from the absence of unsoundness in the methods in all of the static call graphs. There is some unsoundness in the edges—always less than 5%—mostly due to idioms in the OCaml runtime involving the use of `java.lang.reflect.Proxy` for method calls. WALA does not understand this reflective idiom, so edges are missing from the static graph. This sometimes causes further missing edges as needed code is deemed unreachable by the static analysis. Also, proxies result in runtime-generated code appearing on the stack, so the dynamic call graphs contain edges that do not correspond to any source code and hence will be missing from the static call graph. However, manual investigation of these missing edges show that they all occur within the OCaml runtime library. Therefore, the produced static call graphs could still be useful for various applications (e.g., code navigation support in integrated development environments such as Eclipse).

TABLE 11
Various characteristics of the OCAMLLEX and OCAMLDOC OCaml programs.

	LOC	Bytecode Size (MB)	# classes	# methods	# call sites
OCAMLLEX	2,895	1.61	756	13,079	82,225
OCAMLDOC	21,823	4.73	2,514	28,401	376,194

TABLE 12
Count of nodes and edges in the static and dynamic call graphs of the OCAMLLEX and OCAMLDOC OCaml programs.

	Nodes				Edges			
	S	D	D\S	S\D	S	D	D\S	S\D
OCAMLLEX	6,065	1,046	4	5,023	94,811	3,669	340	91,482
OCAMLDOC	26,834	3,756	4	23,082	2,352,782	19,959	2,260	2,335,083

Precision is low across all programs—many nodes and edges in the static call graphs are not present in the dynamic ones. There are several causes. First, values are sometimes stored in a boxed form (`org.ocamljava.runtime.Value`) and indirections used to access and convert them make our context-insensitive analysis imprecise. Second, `MethodHandle` objects are passed to runtime primitives to handle calls, and context-insensitive analysis of these primitives causes significant imprecision. These issues cause more of the standard library to be reachable, which adds further imprecision as edges and nodes from those functions get added. We expect that the imprecision in our OCaml analysis can be addressed to a great extent by existing techniques (e.g., using the Cartesian Product Algorithm [30]).

6.4 Additional Case Studies

In addition to the OCaml CLBG programs, we have studied OCAMLLEX and OCAMLDOC[9], two larger OCaml applications that are publicly available on GitHub. OCAMLLEX is the lexer generator part of the OCaml-Java compiler. OCAMLDOC is the documentation generator tool that ships with the OCaml-Java compiler. Both OCAMLLEX and OCAMLDOC are based on OCaml 4.01.0. Table 11 shows, for both programs (excluding library code), the number of lines of source code, as well as the size of the generated bytecodes, the number of classes, methods, and call sites in the generated bytecodes.

Table 12 shows that only 4 nodes are missing from the static call graphs for OCAMLLEX and OCAMLDOC compared to their respective dynamic call graphs. Further investigation shows that all 4 nodes are involved in reflective method calls to/from methods in the class `org.ocamljava.runtime.annotations.parameters.Parameters`. The code that allocates some of these objects uses `java.lang.reflect.Proxy.newProxyInstance`, which is not currently modelled soundly by WALA. With respect to call graph edges, the static call graphs for OCAMLLEX and OCAMLDOC are missing 9% and 11%, respectively, of the total number of edges present in their dynamic call graphs. Our manual investigation shows that those edges are missing primarily due to the missing nodes that are caused by the unsound handling of reflection in WALA, as well as some runtime-generated code that creates objects on the stack. Similar to our observation with the CLBG OCaml programs, most of those edges occur within the OCaml runtime library, which still renders the generated static call graphs useful for some applications such as IDE support.

The majority of methods (more than 82%) and edges (more than 96%) in the static call graphs for OCAMLLEX and OCAMLDOC are not present in the dynamic call graphs, mainly due to the indirections that arise from storing values in a boxed form, and the context-insensitive handling of calls that involve `MethodHandle` objects. Those are the same reasons for imprecision as in the smaller CLBG OCaml programs. We did not observe any additional issues that would further compromise precision.

7 GROOVY

Groovy [2] is a dynamically-typed object-oriented scripting language that seamlessly integrates with Java. The Groovy compiler provides the option of generating code that makes use of the `invokedynamic` instruction. We conducted two sets of experiments, with and without this option, to understand the impact of this feature. For our experiments, we used Groovy version 2.4.3.

7.1 Translation to JVM bytecode

For calls between Groovy methods, every class contains several static methods that construct an array of `CallSite` objects, implemented in the standard library. This array is indexed by numbers that are assigned to each call site in the class. Each `CallSite` object is initialized with the name of the method to be called. At a call site, the generated bytecode retrieves the corresponding `CallSite` object from the array and invokes a method named `call` on it, passing any parameters. The `call` method invokes many other methods in multiple classes within the Groovy standard library, and ultimately looks up an object of type `GroovyObject` and calls `invokeMethod()` on it. Using a dynamic representation of the class hierarchy, `invokeMethod()` looks up the name of the target method and calls it through reflection. The translation of closures involves the creation of an additional object, and invoking its `doCall()` method using a similar sequence of reflective method calls that starts with an invocation of a method `callCurrent`.

```
42 def foo(x) {x()}
43 def bar = { println 'Hello, World!'}
44 foo(bar)
```

Fig. 8. A simple Groovy program.

Figure 9 shows the relevant bytecodes produced by the Groovy compiler for the program of Figure 8. In the translated code, the generated `main()` method invokes `CallSite.call()` to reflectively invoke the `run()` method containing the top-level code (see the edges labeled ① and ②). The `run()` method calls `CallSite.callCurrent()` to invoke method `foo()` (edges ③ and ④), which in turn invokes `CallSite.call()` again to invoke the closure assigned to variable `bar` (edges ⑤ and ⑥).

If the use of `invokedynamic` is enabled, the bootstrap method used by Groovy returns a `MutableCallSite` that initially points to the same general lookup code that is used in the case without `invokedynamic`. The first time the call site is executed and the desired target method is looked up, the `MutableCallSite` is updated with the `MethodHandle` of the target method. Subsequent calls invoke this `MethodHandle` directly. However, from the point of view of a static analysis, the initial procedure used to determine the target of a call is equally complicated.

7.2 Qualitative Analysis

Each of the dashed lines in Figure 9 represents a sequence of calls containing at least one reflective method call. In general, the many levels of call indirection, object creation, dynamic data structure lookup, and reflection are too complicated for a static analysis to model. In particular, no call edges are created for the calls on lines 42 and 44 in the Groovy code example in Figure 8.

7.3 Quantitative Analysis

Table 13 shows, for each program in our benchmark suite (excluding library code), the number of lines of source code, as well as the size of the generated bytecodes, the number of classes, methods, and call sites in the generated bytecodes.

Quantitative results for the Groovy benchmarks[10] are shown in Table 14. Only the main method appears in the

9. Sources are available from https://github.com/xclerc/ocamljava.

10. CLBG does not provide Groovy implementations of KNUCLEOTIDE and FANNKUCHREDUX. Therefore, we ported existing CLBG implementations to Groovy. We verified correctness by comparing their output against expected output detailed in CLBG.

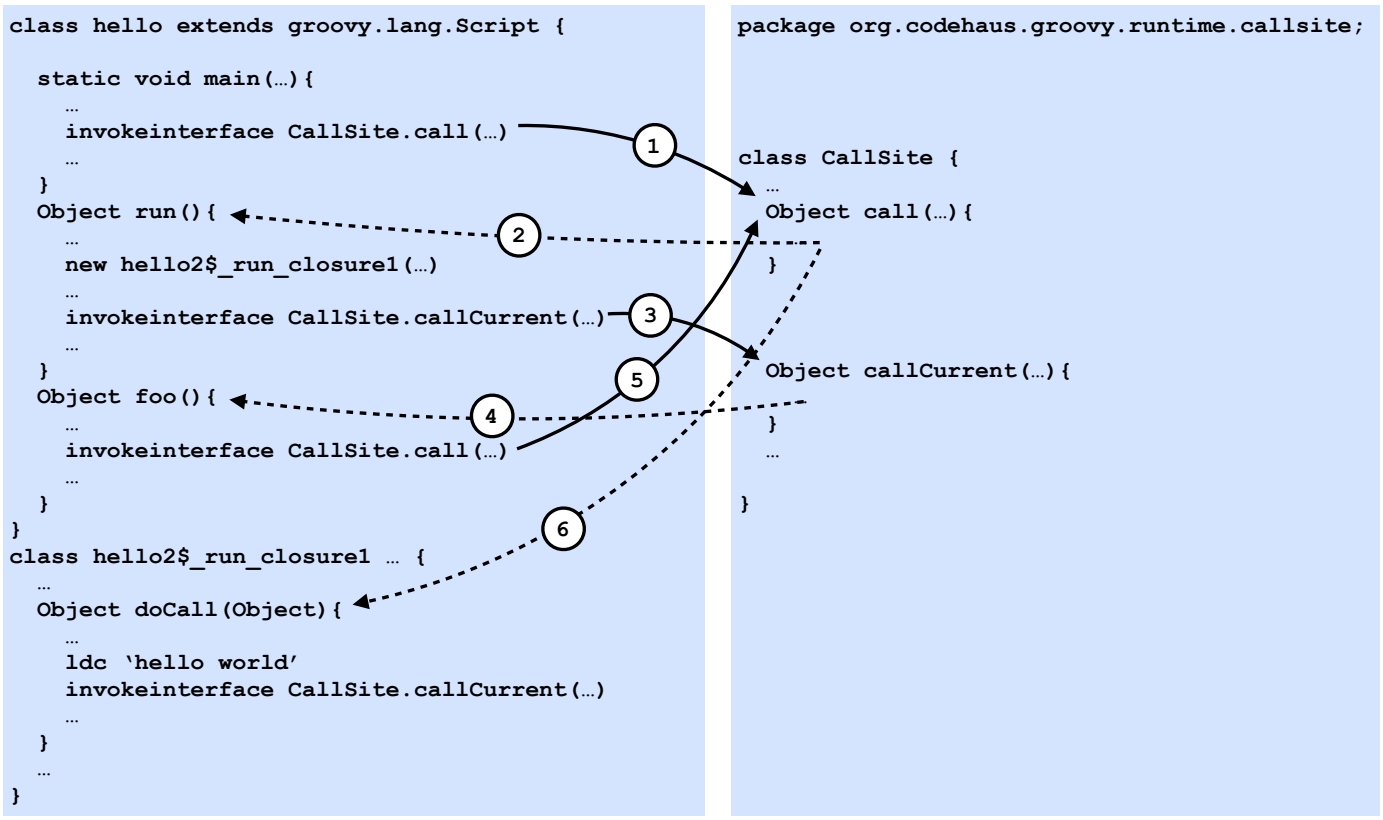


Fig. 9. Depiction of the bytecodes produced by the Groovy compiler for the program of Figure 8.

TABLE 13
Various characteristics of the Groovy CLBG programs (top: without `invokedynamic`, bottom: with `invokedynamic`).

	LOC	Bytecode Size (KB)	# classes	# methods	# call sites
BT	45	7.22	2	69	235
FK	83	5.19	1	40	206
FA	124	7.02	1	62	357
KN	43	9.57	4	68	281
MB	30	4.92	1	39	209
NB	120	12.19	3	98	436
PD	63	10.26	3	107	314
RD	29	7.05	3	59	151
RC	53	8.68	2	77	507
SN	40	4.84	1	44	196

	LOC	Bytecode Size (KB)	# classes	# methods	# call sites
BT	52	6.37	2	63	158
FK	108	4.60	1	37	159
FA	154	6.00	1	59	209
KN	232	8.49	4	58	202
MB	104	4.63	1	36	141
NB	190	10.89	3	89	384
PD	29	9.20	3	98	251
RD	53	6.52	3	50	118
RC	89	7.14	2	71	287
SN	41	4.31	1	41	154

static call graphs, because WALA is unable to compute any call edges for any call sites in application code. The computed static call graphs have very similar sizes because they call approximately the same methods in the Groovy library from boilerplate code in the generated main class.

In the case without `invokedynamic`, the static call graphs are much larger than the dynamic ones because the static analysis infers that most of the Groovy library could be called, but the benchmark programs use only a small fraction at run time. Furthermore, there is significant unsoundness due to the use of reflection. If `invokedynamic` is enabled, the static call graphs are much smaller because much less boilerplate initialization code is reachable. However, the analysis results become more unsound because analysis of the complex, reflection-heavy code reachable via `invokedynamic` calls is beyond the state of the art.

7.4 Additional Case Studies

In addition to the Groovy CLBG programs, we have studied CODENARC and GRULES[11], two large Groovy applications that are publicly available on GitHub. CODENARC is a static analysis tool for Groovy source files that checks for a predefined set of coding standards and best practices. GRULES is a rule engine for data preprocessing. Table 15 shows, for both programs (excluding library code), the number of lines of source code, as well as the size of the generated bytecodes, the number of classes, methods, and call sites in the generated bytecodes.

11. Sources are available from https://github.com/CodeNarc/CodeNarc and https://github.com/zhaber/grules.

TABLE 14
Count of nodes and edges in the static and dynamic call graphs of the Groovy CLBG programs (top: without `invokedynamic`, bottom: with `invokedynamic`).

	Nodes				Edges			
	S	D	D\S	S\D	S	D	D\S	S\D
BT	6,089	744	37	5,382	30,242	1,632	281	28,891
FK	6,089	726	19	5,382	30,242	1,573	204	28,873
FA	6,248	812	34	5,470	30,987	1,712	274	29,549
KN	6,089	832	57	5,314	30,243	1,835	390	28,798
MB	6,089	759	45	5,375	30,243	1,552	237	28,928
NB	6,089	842	57	5,304	30,243	1,850	378	28,771
PD	6,089	736	31	5,384	30,242	1,675	308	28,875
RD	6,089	761	43	5,371	30,242	1,576	226	28,892
RC	6,089	745	51	5,395	30,242	1,504	221	28,959
SN	6,089	737	20	5,372	30,242	1,667	284	28,859

	Nodes				Edges			
	S	D	D\S	S\D	S	D	D\S	S\D
BT	613	645	216	184	1,513	1,197	410	726
FK	613	623	196	186	1,513	1,131	347	729
FA	618	655	224	187	1,521	1,250	468	739
KN	613	742	296	167	1,514	1,380	567	701
MB	613	651	224	186	1,514	1,222	445	737
NB	613	727	285	171	1,514	1,383	572	703
PD	613	670	242	185	1,513	1,257	473	729
RD	613	629	199	183	1,513	1,163	392	742
RC	613	606	183	190	1,513	1,087	321	747
SN	613	632	202	183	1,513	1,224	439	728

TABLE 15
Various characteristics of the CODENARC and GRULES Groovy programs.

	LOC	Bytecode Size (MB)	# classes	# methods	# call sites
CODENARC	56,269	2.18	1,185	15,259	43,470
GRULES	6,793	0.29	163	1,752	6,743

TABLE 16
Count of nodes and edges in the static and dynamic call graphs of the CODENARC and GRULES Groovy programs (top: without `invokedynamic`, bottom: with `invokedynamic`).

	Nodes				Edges			
	S	D	D\S	S\D	S	D	D\S	S\D
CODENARC	7,188	1,320	1,302	7,170	40,420	2,513	2,492	40,399
GRULES	7,142	430	416	7,128	40,121	837	821	40,105

	Nodes				Edges			
	S	D	D\S	S\D	S	D	D\S	S\D
CODENARC	6,718	873	864	6,709	38,632	1,663	1,655	38,624
GRULES	6,735	251	248	6,732	38,659	464	463	38,658

```
45 (ns hello.core
46   (:gen-class))
47 (defn bar [& args]
48   (println "Hello, World!!"))
49 (defn foo [& args]
50   (bar args))
51 (defn -main [& args]
52   (foo args))
```

Fig. 10. A simple Clojure program.

For CODENARC and GRULES in Table 16, a significantly low percentage of the methods and call edges that appear in the static call graphs are for call sites in the application code (CODENARC: 0.25% methods and 0.33% edges; GRULES: 0.17% methods and 0.18% edges). Similar to our quantitative analysis of the Groovy CLBG programs, the static analysis is simply unable to compute any call edges for most of the call sites in the application code. Moreover, compiling the applications with support for `invokedynamic` does not improve the results (CODENARC: 0.13% methods and 0.09% edges; and GRULES: 0.03% methods and 0.02% edges).

8 CLOJURE

Clojure [1], [31] is a dialect of Lisp; key language features include higher-order functions, a powerful macro system, and concurrency control based on Software Transactional Memory. In our experiments, we have used Clojure version 1.5.1.

8.1 Translation to JVM bytecode

Figure 10 shows a simple Clojure program in which `--main` calls `foo`, `foo` calls `bar`, and `bar` prints "Hello, World!". The Clojure compiler translates each Clojure function into a class (for convenience, we will refer to such classes as "function classes" in the discussion below). In the case of our example, function classes `hello.core$foo` and `hello.core$bar` are generated. Each such class defines a method `doInvoke()` that contains code corresponding to the original function in the Clojure source code, and a method `getRequiredArity()` that returns its number of required arguments. We will use Figure 11, which shows some of the bytecodes produced by the Clojure compiler for the program of Figure 10, to illustrate how a typical function call such as the one from `foo` to `bar` is translated:

1) `hello.core$foo.doInvoke()` calls `IFn.invoke()`. This call (labeled ① in the figure) dynamically dispatches to `RestFn.invoke()` (the interface `IFn` and the class `RestFn` are both part of the Clojure runtime library).
2) `RestFn.invoke()` performs some bookkeeping, including a call to `getRequiredArity()` (labeled ②) on the object representing the target function.
3) Lastly, `RestFn.invoke()` calls `doInvoke()` (labeled ③) on the object representing the target function, which represents the actual method body of the callee `bar`.

In the static initializer of class `hello.core`, which contains the `main()` method for the compiled program, code is dynamically loaded by calling `RT.var(``clojure.core'', ``load'').invoke(``hello.core'')`. The "`hello.core`" argument is ultimately used as a classname by the Clojure

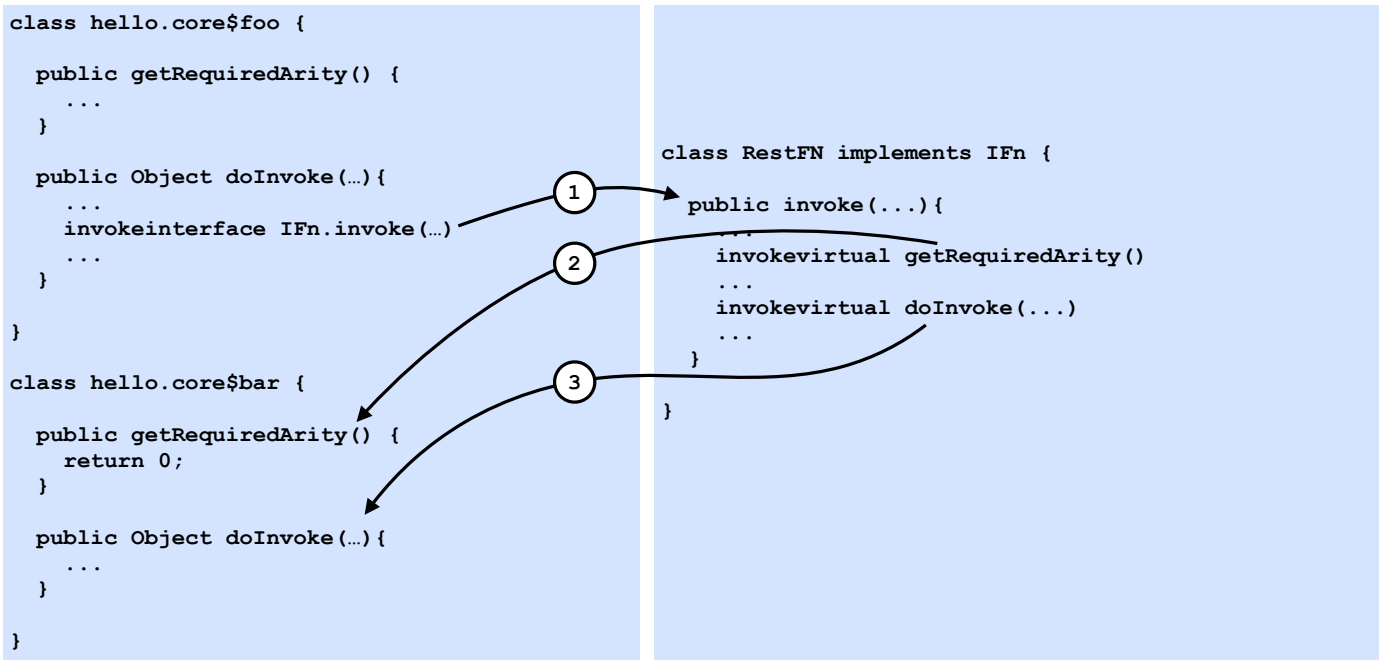


Fig. 11. Depiction of the bytecodes produced by the Clojure compiler for the program of Figure 10.

runtime in a call to the Java Reflection API. Then, in `hello.core.main()`, a call `((IFn)main__var.get()).applyTo()` is executed to launch the actual program, which ultimately calls `hello.core$_main.doInvoke()` using the calling mechanism illustrated above.

8.2 Qualitative Analysis

The use of reflection in compiled code will cause most static analyses to miss some code entirely. Specifically, in our example, since class `hello.core__init` (where function-classes such as `hello.core$foo` and `hello.core$bar` are instantiated) is loaded by reflection, any static analysis that resolves method calls by keeping track of sets of instantiated classes would omit methods such as `hello.core$foo.doInvoke()` from the static call graph. Also, in `main()`, the call to `applyTo()` should resolve to `RestFn.applyTo(ISeq)`, which is inherited by `hello.core$foo` and will ultimately call `hello.core$foo.doInvoke()`. However, since all function classes are deemed not instantiated, no implementor of `RestFn` is deemed instantiated. As a result, `((IFn)man__var.get()).applyTo()` is resolved to call just a few trivial classes rather than the actual bodies of the user-defined functions. Similarly, we found that the translation of module imports by the Clojure compiler also involves the generation of reflective code.

8.3 Quantitative Analysis

Table 17 shows, for each program in our benchmark suite (excluding library code), the number of lines of source code, as well as the size of the generated bytecodes, the number of classes, methods, and call sites in the generated bytecodes.

Table 18 shows the number of nodes and edges in static and dynamic call graphs that we constructed for Clojure versions of the programs from the CLBG suite. All static call graphs have the same number of nodes and edges because

TABLE 17
Various characteristics of the Clojure CLBG programs.

	LOC	Bytecode Size (KB)	# classes	# methods	# call sites
BT	22	21.31	16	44	493
FK	43	26.59	15	35	1,124
FA	77	38.15	26	62	1,300
KN	38	76.39	60	146	2,067
MB	32	27.40	18	45	702
NB	89	31.60	20	76	764
PD	30	14.06	9	23	333
RD	30	15.59	9	24	361
RC	20	19.90	12	47	613
SN	15	19.33	13	31	550

TABLE 18
Count of nodes and edges in the static and dynamic call graphs of the Clojure CLBG programs.

	Nodes				Edges			
	S	**D**	**D\S**	**S\D**	**S**	**D**	**D\S**	**S\D**
BT	1,687	2,996	2,541	1,232	10,791	5,664	4,941	10,068
FK	1,687	3,022	2,561	1,226	10,791	5,753	5,019	10,057
FA	1,687	2,989	2,530	1,228	10,791	5,633	4,913	10,071
KN	1,687	3,252	2,770	1,205	10,791	6,312	5,546	10,025
MB	1,687	2,996	2,554	1,245	10,791	5,638	4,942	10,095
NB	1,687	3,066	2,565	1,186	10,791	5,808	5,005	9,988
PD	1,687	3,856	3,368	1,199	10,791	7,907	7,123	10,007
RD	1,687	2,998	2,540	1,229	10,791	5,674	4,945	10,062
RC	1,687	2,929	2,492	1,250	10,791	5,485	4,795	10,101
SN	1,687	2,941	2,497	1,243	10,791	5,504	4,805	10,092

TABLE 19
Various characteristics of the CHESHIRE and INSTAPARSE Clojure programs.

	LOC	Bytecode Size (MB)	# classes	# methods	# call sites
CHESHIRE	385	4.25	3,458	11,868	128,685
INSTAPARSE	9,003	4.14	3,487	11,225	126,678

TABLE 20
Count of nodes and edges in the static and dynamic call graphs of the CHESHIRE and INSTAPARSE Clojure programs.

	Nodes				Edges			
	S	D	D\S	S\D	S	D	D\S	S\D
CHESHIRE	1,687	4,369	3,820	1,138	10,791	9,914	8,998	9,875
INSTAPARSE	1,687	4,234	3,690	1,143	10,791	10,413	9,425	9,803

they consist of only a `main()` method and parts of the Clojure runtime libraries. The application logic is completely missing in each case because analyzing complex reflective code is beyond the state of the art as discussed above. This makes the analysis extremely unsound. Manual inspection of the parts of the call graphs corresponding to the runtime libraries also reveals significant imprecision, which is due to the complex call-chains introduced during the translation of function calls that cannot be analyzed precisely by a context-insensitive analysis.

8.4 Additional Case Studies

In addition to the Clojure CLBG programs, we have studied CHESHIRE and INSTAPARSE[12], two large Clojure applications that are publicly available on GitHub. CHESHIRE provides a suite of tools for fast encoding and decoding of JSON and JSON SMILE (binary JSON format), with added support for more types and the ability to use custom encoders. INSTAPARSE is an engine that builds parsers for context-free grammars. Table 19 shows, for both programs (excluding library code), the number of lines of source code, as well as the size of the generated bytecodes, the number of classes, methods, and call sites in the generated bytecodes.

Similar to our quantitative analysis of the Clojure CLBG benchmarks, Table 20 shows that the static call graphs for CHESHIRE and INSTAPARSE have the same number of nodes and edges. Further investigation shows that those nodes and edges are the same set of nodes and edges that the static call graph analysis computes for the Clojure CLBG programs, except for the nodes and edges involving the `main()` method. This result shows that regardless of the size or complexity of the analyzed Clojure program, the JVM bytecodes generated for it are not amenable to static call graph analysis. This is mainly due to heavy use of reflection and using complex call-chains for function calls.

9 PYTHON

Python [4] is a popular dynamically-typed object-oriented programming language. In addition to classes and objects, it supports lists, sets, and dictionaries as built-in data structures. Other key Python features include lambda expressions, comprehensions, and generators. Jython [32] is a JVM-based implementation of Python. In our experiments, we used Jython 2.7-b3, which is compatible with Python 2.7.

12. Sources are available from https://github.com/dakrone/cheshire and https://github.com/Engelberg/instaparse.

```
53 def foo():
54     bar()
55 def bar():
56     print "Hello, World!"
57 foo()
```

Fig. 12. A simple Python program.

9.1 Translation to JVM bytecode

Figure 12 shows a small Python program that we will use to illustrate how Jython translates Python source code to JVM `.class` files. The program declares two functions, `foo()` and `bar()`. The program calls `foo()` on line 57, `foo()` calls `bar()` on line 54, which in turns prints "`hello world`" on line 56. For this program, the Jython compiler generates a class `hello$py` containing the main application logic. In general, each function call in the Python source code is mapped to a sequence of method calls in the generated bytecode. For example, for the call on line 54, the following sequence is generated:

1) `hello$py.foo$1()` calls `PyObject.__call__()`, a method in the Jython runtime libraries.
2) `PyObject.__call__()` invokes another library method, `PyCode.call()`, which is dynamically dispatched to `PyBaseCode.call()`.
3) `PyBaseCode.call()` invokes another `call()` method in the same class that dispatches to an overriding definition in `PyTableCode`.
4) `PyTableCode.call()` invokes `hello$py.call_function()` in the class containing the translated application functions.
5) `hello$py.call_function()` contains a switch statement in which each branch calls one of the application functions depending on the value of its first parameter, `fid`. The value of `fid` originates from an instance field.

Figure 13 shows the relevant bytecodes produced by the Jython compiler for the program of Figure 12. In this figure, the previous five calls are visualized by the correspondingly-annotated arrows.

9.2 Qualitative Analysis

Suppose we want to construct a call graph for the program of Figure 12 by analyzing the bytecodes generated for it by the Jython compiler. As mentioned, `hello$py.call_function()` calls each of `f$0()`, `foo$1()`, and `bar$2()`, which correspond to the top-level code and the functions `foo()` and `bar()` in the Python source code. For any other method call in the program (e.g., the call to `foo()` from top-level code), a similar chain of call edges exists that includes `hello$py.call_function()`. Consequently, *every call site* in a Python source file is translated into a chain of method calls that involves `hello$py.call_function()`,

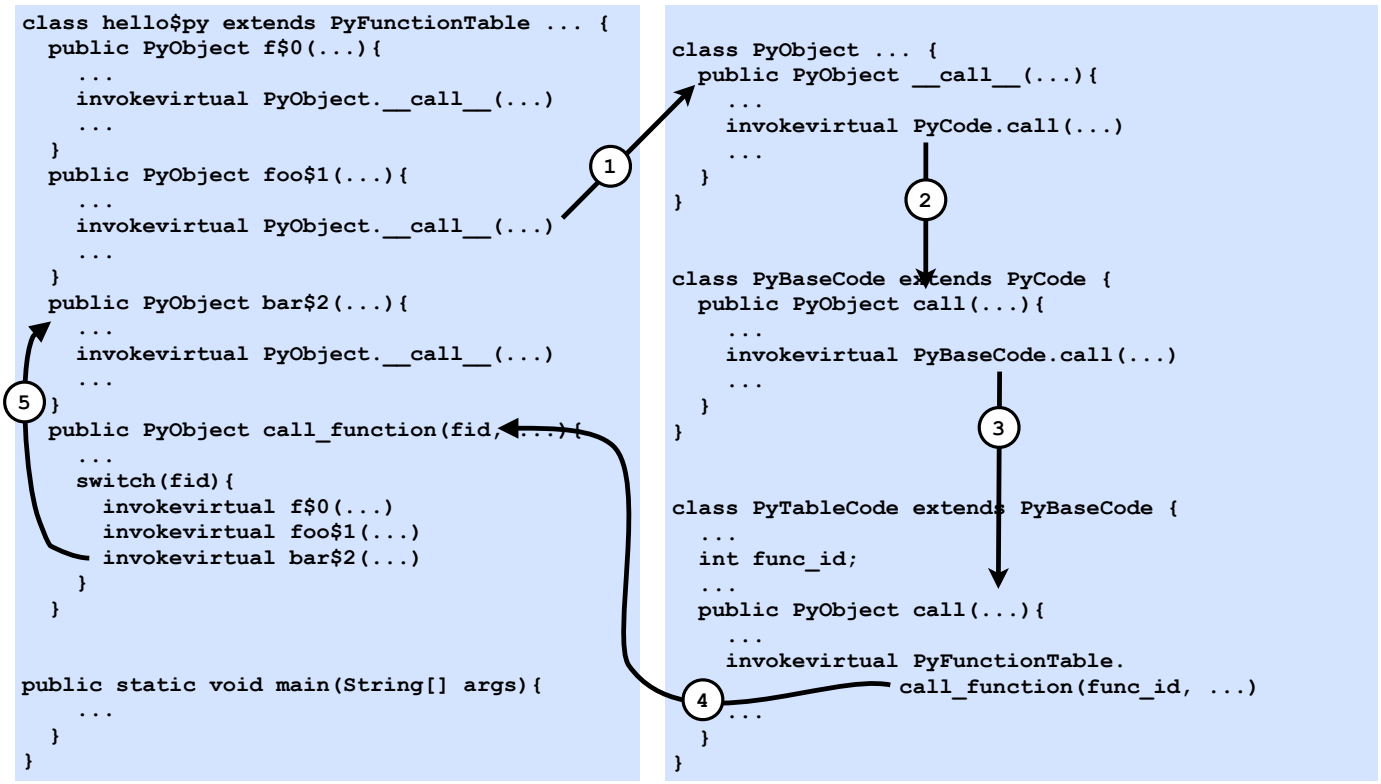


Fig. 13. Depiction of the bytecodes produced by the Jython compiler for the program of Figure 12.

TABLE 21
Various characteristics of the Python CLBG programs.

	LOC	Bytecode Size (KB)	# classes	# methods	# call sites
BT	37	2.92	1	8	183
FK	47	3.01	1	7	227
FA	68	4.44	1	12	401
KN	29	4.31	1	12	277
MB	106	2.85	1	7	157
NB	110	4.93	1	11	541
PD	33	2.51	1	6	157
RD	53	3.24	1	8	145
RC	19	3.16	1	9	130
SN	111	3.05	1	11	219

TABLE 22
Count of nodes and edges in the static and dynamic call graphs of the Python CLBG programs.

	Nodes				Edges			
	S	D	D\S	S\D	S	D	D\S	S\D
BT	10,449	1,761	71	8,759	110,549	3,141	229	107,637
FK	10,403	1,753	61	8,711	100,230	3,095	203	97,338
FA	10,481	3,267	132	7,346	110,739	6,675	406	104,470
KN	10,465	4,790	702	6,377	106,949	10,702	1,614	97,861
MB	10,444	1,770	75	8,749	107,851	3,148	233	104,936
NB	10,465	1,792	70	8,743	109,564	3,234	224	106,554
PD	10,290	1,736	62	8,616	104,167	3,104	213	101,276
RD	10,335	4,301	525	6,559	92,436	9,899	1,513	84,050
RC	10,377	4,264	304	6,417	95,170	9,819	1,035	86,386
SN	10,418	1,771	91	8,738	102,946	3,194	278	100,030

which calls *every method* corresponding to a function in the same Python file.[13]

Based on this observation, it is difficult to see how a bytecode-based analysis of the JVM bytecodes produced by Jython could compute a useful call graph. A precise static analysis would need to employ many levels of call-string context-sensitivity. It would also need to reason about heap-allocated objects and values, which is beyond the current state-of-the-art. Therefore, we conclude that generating precise call graphs from JVM bytecodes produced by Jython is infeasible. Soundness is compromised for similar reasons: all methods in imported modules are missing because Jython generates code that relies on reflection to implement module import.

13. Jython generates a separate class for each Python source file, each with its own `call_function()` method.

9.3 Quantitative Analysis

Table 21 shows, for each program in our benchmark suite (excluding library code), the number of lines of source code, as well as the size of the generated bytecodes, the number of classes, methods, and call sites in the generated bytecodes.

As Table 22 shows, on average, about 76% of the methods and about 95% of the edges in the static call graphs do not occur in the dynamic call graphs. This significant imprecision mainly arises from the call chains involving `call_function()` discussed above. Moreover, features such

TABLE 23
Various characteristics of the GRAKO and PYBARCODE Python programs.

	LOC	Bytecode Size (MB)	# classes	# methods	# call sites
GRAKO	6,682	0.35	53	1,435	32,951
PYBARCODE	1,165	0.24	11	171	4,145

TABLE 24
Count of nodes and edges in the static and dynamic call graphs of the GRAKO and PYBARCODE Python programs.

	Nodes				Edges			
	S	D	D\S	S\D	S	D	D\S	S\D
GRAKO	23,606	555	549	23,600	3,690,843	1,057	1,054	3,690,840
PYBARCODE	23,736	93	85	23,728	3,763,699	160	155	3,763,694

as module imports that are implemented using reflection cause about 6% of the methods and 9% of the edges in the dynamic call graphs to be absent from the static call graphs.

9.4 Additional Case Studies

In addition to the Python CLBG programs, we have studied GRAKO and PYBARCODE[14], two larger Python applications that are publicly available on GitHub. GRAKO is a parser generator for input grammars that are written in a variation of the extended Backus-Naur form (EBNF). PYBARCODE is a Python package that provides a convenient way to create barcodes given their numerical value as input using only built-in Python libraries. Table 23 shows, for both programs (excluding library code), the number of lines of source code, as well as the size of the generated bytecodes, the number of classes, methods, and call sites in the generated bytecodes.

Table 24 shows that the overwhelming majority (more than 99%) of methods and edges in the static call graphs for both GRAKO and PYBARCODE do not occur in the dynamic call graphs. These results are in line with our quantitative analysis of the Python CLBG benchmarks, showing that the static call graphs for GRAKO and PYBARCODE are also significantly imprecise. Similarly, the majority of methods and edges (more than 91%) in the dynamic call graphs for GRAKO and PYBARCODE are absent from the static call graphs, mainly due to the use of reflection and module imports. We did not observe any additional issues in the larger applications that would further compromise unsoundness or precision.

10 RUBY

Ruby is a popular object-oriented programming language for server-side scripting. JRuby [33] is a JVM-based implementation of Ruby. In our experiments, we used JRuby version 1.7.13. Similar to Groovy, the JRuby compiler (`jrubyc`) has a flag for enabling the generation of bytecode containing `invokedynamic` instructions.

14. Sources are available from https://bitbucket.org/neogeny/grako/ and https://bitbucket.org/whitie/python-barcode/.

```
58 def bar
59   print "Hello, World!\n"
60 end
61 def foo
62   bar
63 end
64 foo
```

Fig. 14. A simple Ruby program.

10.1 Translation to JVM bytecode

Figure 14 shows a simple Ruby program that defines functions `foo` and `bar`, and top-level code that invokes `foo`. The function `foo` calls `bar`, and `bar` prints "Hello, World!".

`Jrubyc` translates each Ruby source file into a separate class that defines methods `main()`, `load()`, and `__file__()`. The generated classes contain an additional method for each function in the Ruby source code.[15] For the program of Figure 14, a class `hello` with methods `method__0$RUBY$bar()` and `method__1$RUBY$foo()` is generated. The compiler names the class `hello` because the Ruby source is in a file called `hello.rb`. Each function call in the Ruby source code is translated to a sequence of method calls in the generated bytecode. For the call from `foo` to `bar`, the following sequence is generated:

1) `method__1$RUBY$foo()` invokes `org.jruby.runtime.CallSite.call()`, which dynamically dispatches to `org.jruby.runtime.callsite.CachingCallSite.call()`.
2) then, `CachingCallSite.call()` invokes `CachingCallSite.cacheAndCall()`, a method in the same class.
3) `CachingCallSite.cacheAndCall()` retrieves a `DynamicMethod` object from a cache that is maintained at run time and invokes `org.jruby.internal.runtime.methods.DynamicMethod.call()` on that object. This triggers a sequence of calls to methods in the JRuby runtime and Java standard libraries that ultimately invokes a method `call()` in a class `hello$method__0$RUBY$bar` that is *generated at run time*.
4) Finally, method `hello$method__0$RUBY$bar.call()` invokes `hello.method__0$RUBY$bar()`.

Figure 15 shows the relevant bytecodes produced by the JRuby compiler for the program of Figure 14. In the figure, each of the steps discussed above is visualized using a correspondingly-labeled arrow.

Enabling the use of `invokedynamic` follows a similar approach to that used in Groovy. The code that actually determines the target method is the same whether that feature is enabled or not. A static analysis would then suffer from the same issues we previously discussed in Section 7.

10.2 Qualitative Analysis

The code generated by `jrubyc` poses serious challenges to static analysis. First, classes such as `hello$method__0$RUBY$bar` are generated at run time. A static analysis is unable

15. In fact, JRuby generates 2 overloaded methods for each function in the source code, where one performs additional checks before invoking the other. In our example, only the method that does not perform argument-checking is used.

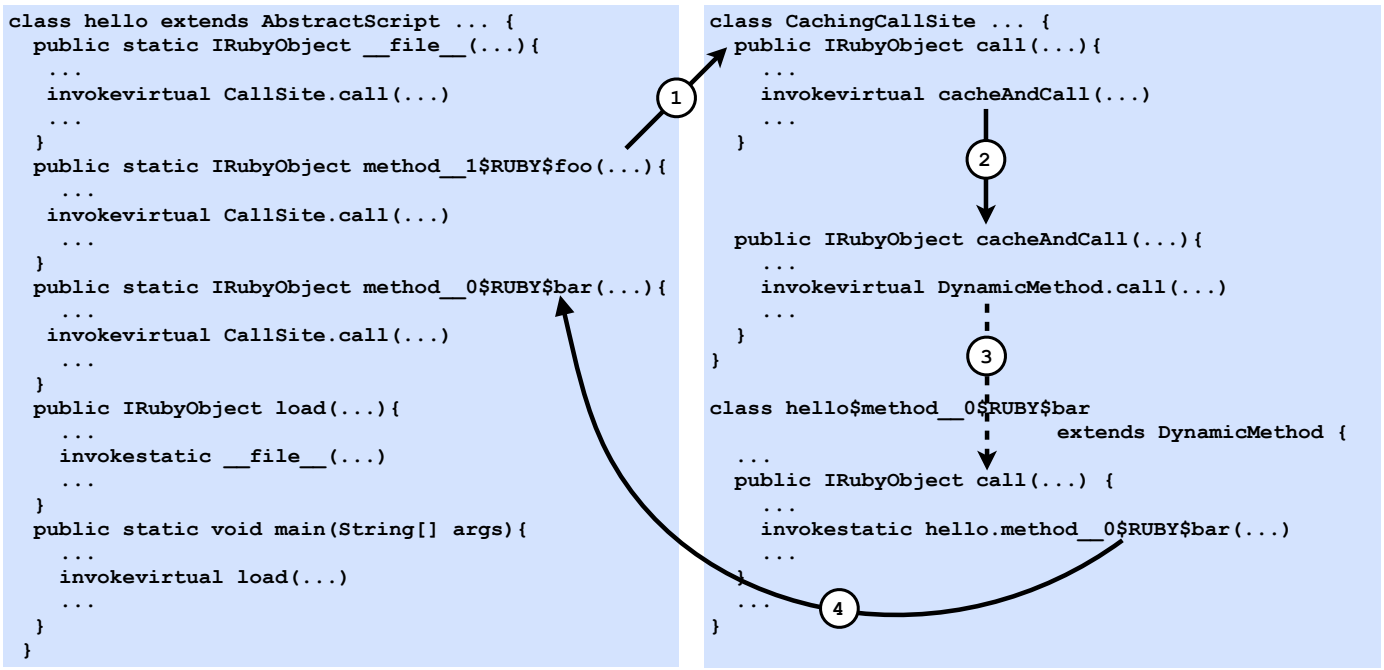


Fig. 15. Depiction of the bytecodes produced by the JRuby compiler for the program of Figure 14.

TABLE 25
Various characteristics of the Ruby CLBG programs (top: without `invokedynamic`, bottom: with `invokedynamic`).

	LOC	Bytecode Size (KB)	# classes	# methods	# call sites
BT	37	4.61	1	12	4
FK	47	4.10	1	8	4
FA	68	6.63	1	15	4
KN	29	5.31	1	18	4
MB	106	8.43	1	43	4
NB	110	7.61	1	18	4
PD	33	3.18	1	6	4
RD	53	5.47	1	14	4
RC	19	3.58	1	10	4
SN	111	8.41	1	46	4

	LOC	Bytecode Size (KB)	# classes	# methods	# call sites
BT	37	5.04	1	12	4
FK	47	4.67	1	8	4
FA	68	6.88	1	15	4
KN	29	5.65	1	18	4
MB	106	9.21	1	43	4
NB	110	8.03	1	18	4
PD	33	3.52	1	6	4
RD	53	5.57	1	14	4
RC	19	3.77	1	10	4
SN	111	9.27	1	46	4

to reason about the behavior of such classes. Therefore, the analysis will miss calls to methods such as `hello.method__0$RUBY$bar()`, which renders the computed call graph unsound.

Furthermore, `CachingCallSite.call()` and `Dynamic`

TABLE 26
Count of nodes and edges in the static and dynamic call graphs of Ruby CLBG programs (top: without `invokedynamic`, bottom: with `invokedynamic`).

	Nodes				Edges			
	S	**D**	**D\S**	**S\D**	**S**	**D**	**D\S**	**S\D**
BT	14,208	7,015	2,679	9,872	73,685	15,302	8,077	66,460
FK	14,139	6,956	2,667	9,850	73,373	15,165	8,037	66,245
FA	14,187	7,115	2,734	9,806	73,471	15,834	8,506	66,143
KN	14,212	7,224	2,773	9,761	73,620	15,877	8,440	66,183
MB	14,126	7,382	2,943	9,687	73,400	16,757	9,209	65,852
NB	14,210	7,054	2,679	9,835	73,613	15,638	8,312	66,287
PD	14,091	6,846	2,672	9,917	73,172	14,860	8,001	66,313
RD	14,206	7,118	2,749	9,837	73,477	15,589	8,323	66,211
RC	14,162	7,047	2,722	9,837	73,472	15,393	8,198	66,277
SN	14,135	7,079	2,738	9,794	73,430	16,061	8,753	66,122

	Nodes				Edges			
	S	**D**	**D\S**	**S\D**	**S**	**D**	**D\S**	**S\D**
BT	14,128	7,087	2,744	9,785	73,308	15,629	8,450	66,129
FK	14,116	7,071	2,724	9,769	73,265	15,517	8,295	66,043
FA	14,129	7,123	2,739	9,745	73,280	16,157	8,791	65,914
KN	14,129	7,179	2,760	9,710	73,303	16,212	8,805	65,896
MB	14,095	7,408	2,942	9,629	73,265	17,175	9,562	65,652
NB	14,117	7,124	2,755	9,748	73,286	15,982	8,692	65,996
PD	14,056	6,954	2,750	9,852	73,046	15,105	8,224	66,165
RD	14,090	7,137	2,795	9,748	73,112	15,807	8,638	65,943
RC	14,099	7,126	2,771	9,744	73,235	15,720	8,510	66,025
SN	14,093	7,143	2,770	9,720	73,257	16,475	9,115	65,897

TABLE 27
Various characteristics of the CHUNKYPNG and PSD.RB Ruby programs.

	LOC	Bytecode Size (MB)	# classes	# methods	# call sites
CHUNKYPNG	4,516	0.32	48	1,770	192
PSD.RB	5,638	0.61	140	2,750	560

`Method.call()` are invoked for every call in the source code. These methods, in turn, invoke all methods corresponding to source code functions such as `hello.method__0$RUBY$bar()` and `hello.method__1$RUBY$foo()`. As with Jython, precise static analysis would need many levels of context sensitivity and it would need to understand heap-allocated cached objects; this is beyond the current state of the art. Therefore, we conclude that generating precise call graphs from JVM bytecodes produced by JRuby is infeasible.

In Ruby, the `require` construct is used to import code from another file. JRuby implements this idiom by dynamically loading a script from a file using a `ClassLoader`, and then relying on the mechanisms described above to interpret these scripts. In general, static analysis is incapable of precisely accounting for code interpreted at run time in this way, resulting in additional unsoundness.

10.3 Quantitative Analysis

Table 25 shows, for each program in our benchmark suite (excluding library code), the number of lines of source code, as well as the size of the generated bytecodes, the number of classes, methods, and call sites in the generated bytecodes.

Table 26 shows the sizes of dynamic and static call graphs for the JRuby benchmarks. The numbers are very similar across all benchmarks because the large JRuby library contains overwhelmingly more methods than the benchmark programs themselves. As discussed above, the static call graphs miss large numbers of nodes and edges from the dynamic call graphs, primarily because methods in classes generated at run time are not known to the static analysis.

The computed call graphs also exhibit significant imprecision: in each benchmark, approximately 70% of the methods and 90% of the call edges in the static call graph are absent from the dynamic call graph. This imprecision occurs overwhelmingly in the JRuby standard library. Due to the library's complex structure, any current analysis would find almost all of the standard library to be reachable, although only a relatively small fraction is used at run time by our subject programs. As expected, the results are very similar regardless whether or not the use of `invokedynamic` is enabled.

10.4 Additional Case Studies

In addition to the Ruby CLBG programs, we have attempted to study CHUNKYPNG and PSD.RB[16], two larger Ruby applications that are publicly available on GitHub. CHUNKYPNG is a tool that provides read and write functionalities to PNG images in pure Ruby. PSD.RB is a Ruby-based general purpose Adobe Photoshop[17] file parser. Table 27 shows, for both programs (excluding library code), the number of lines of source code, as well as the size of the generated bytecodes, the number of classes, methods, and call sites in the generated bytecodes.

Unfortunately, the ahead-of-time build system for JRuby seems not to support building such large codebases. We were not able to get the JRuby runtime to execute without depending on dynamically interpreting a large number of ruby scripts. Such setup makes it almost impossible to perform any reasonable comparison between static and dynamic call graphs.

11 PERFORMANCE

Reflection- and indirection-heavy implementations hamper static analysis, and plausibly might harm performance as well. To observe whether any correlations exist between compilation strategies and run-time performance, we investigate the performance of the bytecode generated by the compilers for the various languages on the CLBG benchmarks. As a baseline for comparison, we evaluated the Java versions as well. We evaluate the performance in terms of *running time* and *memory usage rate*.

11.1 Setup

We used the Java Microbenchmarking Harness (JMH) tool [34] that ships with the JDK to build, run, and analyze the CLBG benchmarks written in Java and the other languages under study that target the JVM. Our experimental setup uses JMH 1.12 with 30 warmup iterations and 30 measurement iterations. However, we had to exclude the programs **KN**, **MB**, **RD**, and **RC** from the Clojure benchmark, because they use program constructs that eventually call `System.exit()` in the Java runtime library. This call causes JMH to prematurely shutdown the JVM which causes the execution to stop before reporting any measurements. Similarly, we had to exclude the programs **NB**, **PD**, and **RD** from the benchmark for JRuby with `invokedynamic`. The code for these programs ends up re-initializing constant values in the generated bytecode when the flag for `invokedynamic` in the JRuby compiler is enabled, causing the JVM to throw a runtime exception, which in turn prevents JMH from reporting its measurements. Including these CLBG programs would require editing their code to avoid those runtime errors, which might invalidate the results of this experiment. Therefore, we opted for excluding them from the performance evaluation.

Using JMH, we measure the running time using the average running time that JMH reports for each program. For the memory consumption, we use the `gc.alloc.rate.norm` output from JMH, which measures the normalized number of allocated bytes per operation for each program.

11.2 Results

Figures 16–17 present the running times and memory usage rates, respectively, in logarithmic scale, normalized to that of Java, for each of the 10 benchmark programs per language.

16. Sources are available from https://github.com/wvanbergen/chunky_png and https://github.com/layervault/psd.rb.

17. http://www.adobe.com/ca/products/photoshop.html

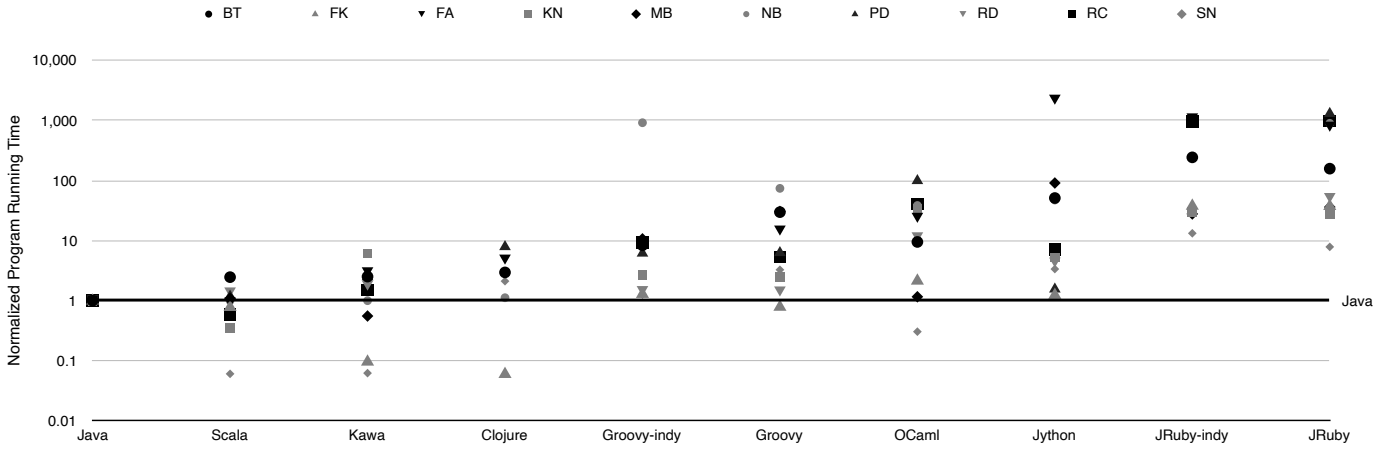


Fig. 16. The program running time, in logarithmic scale, normalized to that of Java, of the 10 CLBG benchmark programs.

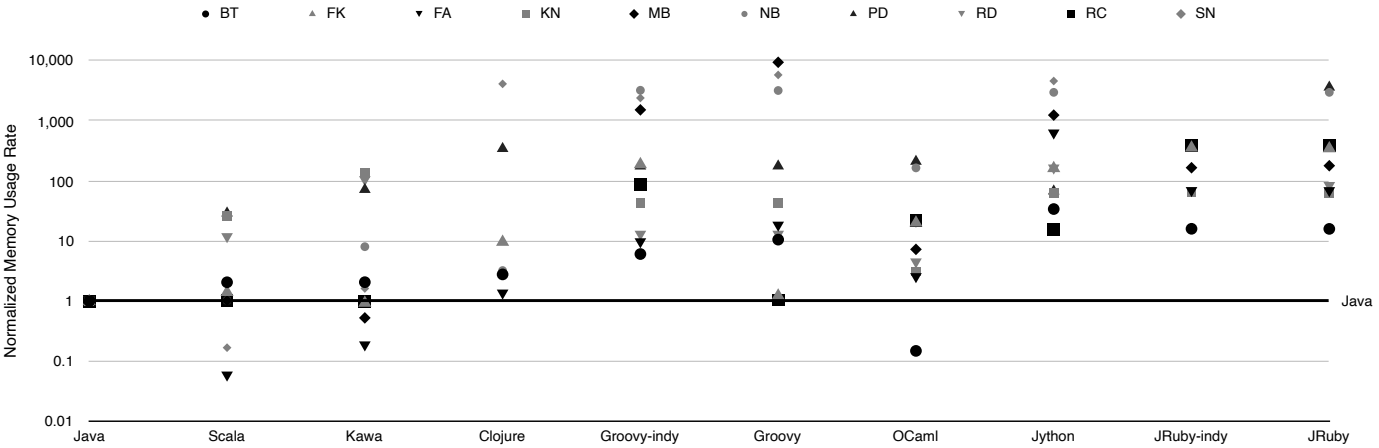


Fig. 17. The memory usage rate, in logarithmic scale, normalized to that of Java, of the 10 CLBG benchmark programs.

Across all the languages that we have studied, Scala generates JVM bytecodes with the best performance in terms of running time and memory usage. In fact, the Scala bytecodes run 1.4× faster than their Java counterparts, while using 1.7× more memory. This discrepancy in running time is primarily due to the benchmarks **SN** and **KN**. Our manual investigation shows that **SN** is approximately 15× slower in Java compared to Scala, which contributes the most to this discrepancy. This is because the Java version is multi-threaded, while the Scala version is single-threaded. Given the 32-core machine that we ran our experiments on, the synchronization barrier for the Java version is very expensive, and is roughly 15× the cost of the real computation that the benchmark actually does. **KN** exhibits the same behaviour, but to a smaller extent where the synchronization barrier causes the Java version to be approximately 3× slower than its Scala counterpart.

Surprisingly, the JVM bytecodes generated by dynamically-typed Kawa have the same performance as Java with respect to running time, while using an average of 4.3× more memory. This performance is better than that of the JVM bytecodes generated by OCaml, the only other statically-typed language in our study. When compared to Java, OCaml generates JVM bytecodes that run 10× slower, and use 10× more memory.

Another surprising observation is that dynamically-typed Clojure generates JVM bytecodes that run faster than their OCaml counterparts. This may be attributed to the Clojure JVM bytecodes being more amenable to the optimizations that the Java Just-in-Time (JIT) compiler performs at runtime. On the other hand, the Clojure JVM bytecodes use 23× more memory than Java, which is more than double the memory than the OCaml JVM bytecodes use.

The JVM bytecodes generated by all the other dynamically-typed languages have worse performance with respect to running time and memory usage. In particular, JRuby JVM bytecodes run 136× slower and use 232× more memory than Java. Compiling Ruby scripts with the `invokedynamic` flag turned on improves the performance for the JRuby JVM bytecodes as they run 101× slower and use 127× more memory than Java. On the other hand, using `invokedynamic` doubles the memory usage of the Groovy JVM bytecodes without any significant win with respect to the running time.

The results of this experiment show that there is a strong correlation between runtime performance and the extent to which the generated JVM bytecode is amenable to static analysis. This suggests that efforts by language implementors that aim to improve the run-time performance of generated code may benefit other applications of bytecode-

based program analysis as well.

12 Discussion and Threats to Validity

12.1 Discussion

For soundness of analysis, the systems that we studied mostly fall into two classes. The Kawa, Scala and OCaml-Java systems produced sound call graphs in our experiments, which we believe is due to relatively straightforward translations that make judicious use of some features of recent Java versions such as MethodHandles. MethodHandles encode reflective accesses in a way that is directly readable from the bytecode. This allows analysis to soundly and precisely model which methods are denoted, which, in turn, enables sound construction of call graphs that use these features. On the other hand, the other languages make heavy use of string-based reflection and frequently store and retrieve these strings to and from the heap. This makes good approximations problematic, and results in unsound call graphs. We experimented with versions of some languages that employ `invokedynamic`; these implementations employed similar string-based techniques and yielded similarly unsound results.

12.2 Language Features

A critical reader might argue that the programs studied in this paper are small, that they do not cover the full range of each language's features, and that they are perhaps not representative of real-world programs. We do not consider the above considerations to be serious reasons for concern, because the primary conclusions of our study (i.e., whether soundness or imprecision occurs for each language under consideration) are evident from the manual analysis of small example programs, and supported by our quantitative experiments with the CLBG programs.

To further address the concern about the CLBG programs being fairly small, we conducted, for each of the languages under consideration (except Ruby), additional experiments on two larger subject programs. For these programs, we performed the same quantitative experiments as for the CLBG programs. Moreover, in a detailed qualitative assessment, we determined whether additional issues arose in these larger programs that comprise soundness or precision. From this assessment, we conclude that the larger programs exhibit the same issues as the smaller programs and do not fundamentally alter our conclusions.

12.3 Code Coverage

In general, computing precise static call graphs is undecidable, and in this paper, we have used dynamic call graphs to estimate the precision of static call graphs. However, a dynamic call graph provides an under-approximation of a precise static call graph, and the reader may wonder if code coverage is reasonable. To address this concern, we measured basic block coverage. On average, across all the languages under consideration, the program input used to run the CLBG benchmark suite provides a basic block coverage of 87%, which is quite high.

12.4 Generality of Results

The choice of WALA and its 0-CFA analysis does not overly bias our results as the analysis challenges we found are more fundamental. For example, the qualitative studies we have conducted show that dynamic translation schemes such as run-time code generation and reflection potentially cripple any bytecode-based static call graph analysis, not just WALA's.

12.5 Applicability of Results

We chose to study call graph construction because call graphs have many applications in software engineering, including bug-finding (see e.g., [13]), detecting security vulnerabilities (see e.g., [14]), IDE features such as code navigation (see e.g., [15], and application extraction and optimization (see e.g., [16], [17]). Furthermore, call graphs are deeply connected to pointer analysis, another fundamental analysis technology [35]. Hence, our results indicate that bytecode-based program analysis could be used for a range of applications on a range of languages.

12.6 Ephemeral Results

We have used multiple versions of the language implementations over the course of the study, with no major impact. Even the use of the `invokedynamic` instruction appears to have little or no impact on analysis results. In fact, Figure 17 shows that using `invokedynamic` leads to worse memory performance for the language implementation for Groovy. We therefore expect that the results of our study will still hold for future versions of these systems.

13 Related Work

In this section, we discuss several categories of related work.

13.1 Empirical Studies

Reif et al. [36] study how unsoundness arises in call graph construction algorithms for Java due to a number of features and mechanisms, including: the use of reflection, unsafe native APIs, serialization, lambdas and method references, and default methods. In their study, they craft example programs that exercise each of these features and determine if loss of soundness is observed when analyzing these programs using existing implementations of call graph construction algorithms in SOOT [10] and WALA [11]. The study concludes that WALA provides better support than SOOT for trivial uses of reflection (e.g., using string literals) and Java 8 features such as lambdas, but that both WALA and SOOT lose soundness when serialization or the unsafe native APIs are used.

Sui et al. [37] present a similar study that investigates how the soundness of call graph construction algorithms is compromised by dynamic language features such as reflection and dynamic loading, proxies, serialization, the `invokedynamic` instruction, and the unsafe native API. They define a micro-benchmark consisting of a set of programs that exercise each of the dynamic features. Each program specifies its expected execution behavior using Java's annotation mechanism. Specifically, annotations specify whether

methods are always executed, never executed, or sometimes executed (which may happen if the execution of methods is platform-dependent). Sui et al. conduct a study where they apply call graph construction algorithms from the SOOT [10], WALA [11], and DOOP [9] frameworks to their benchmark, and report that, while none of the frameworks handles all the dynamic features soundly in all cases, each framework appears to have some unique strengths. In particular, WALA and DOOP [38], [39] support `invokedynamic` and proxies, and SOOT relies on Tamiflex to analyze reflective code. In contrast to these studies, which consider the challenges posed by dynamic language features in the context of synthetic benchmarks, our work has focused on identifying and analyzing the challenges posed to static analysis that arise in the JVM bytecodes produced by compilers for 7 programming languages on programs taken from an existing benchmark suite and from open-source repositories.

Most studies of non-Java JVM-hosted languages focus on the dynamic behavior of bytecode from the point of view of a JVM that executes it. Li et al. [40] studied JVM bytecode generated by Scala, Clojure, Jython, and JRuby for the CLBG programs. They measured the diversity of bytecode instruction sequences executed, the sizes of methods, the depths of the run time stack, the hotness distribution of methods and basic blocks, the sizes and lifetimes of objects, and the amount of boxing of primitive types. Sarimbekov et al. [41] studied Clojure, Jython, and JRuby versions of the CLBG programs. They measured polymorphic calls, immutability of fields, objects, and classes, lifetimes of objects, amount of memory zeroing, and the number of evaluations of identity hash codes. Sewe et al. [27] introduced a benchmark suite for Scala similar to the DaCapo suite [42] for Java and compare the dynamic behavior of these programs to that of the DaCapo Java programs. The use of dynamic features has also been studied for languages that are not normally compiled to JVM bytecode. Richards et al. [43], [44] studied the use of dynamic features in JavaScript, especially the `eval` construct. Hills et al. [45] studied the use of various features in PHP programs, including `eval`. In contrast to these studies, our work examines JVM bytecode from a static analysis perspective.

13.2 Multilingual Virtual Machines

The translation of various programming languages to bytecode-based platforms has received considerable attention. Several works consider the compilation of Scala to JVM bytecode [46], [47], [48], [49]. Other languages, in addition to those that were already discussed, include Star [50], Pizza [51], [52], and even machine language code [53]. The Microsoft Common Language Runtime (CLR) was designed from the outset to support multiple source languages, including C#, C++, and Visual Basic, and has since been used as the target of many others. Gordon et al. [54] presented a type system for the CLR Intermediate Language (CIL). Bebenita et al. [55] used CIL as the bytecode language for a tracing just-in-time (JIT) compiler specifically designed for dynamic scripting languages like JavaScript. Recent work has adapted the virtual machine more deeply to support new languages. Castanos et al. [56] modified an existing JIT compiler to exploit dynamic characteristics of Python for improved performance. Würthinger et al. [57] built a virtual machine that allows custom source front-ends for a variety of languages. This work laid the foundation for the later development of the GraalVM [58], a universal virtual machine that supports running JavaScript, Python, Ruby, R, in addition to JVM-hosted languages. The custom front-ends in GraalVM, which are typically implemented on top of the Truffle language implementation framework, interpret, profile, and optionally transform source programs. The system later partially evaluates these interpreters to generate machine code. Savrun-Yeniceri et al. [59] consider forms of threaded code generation to speed up JVM-hosted interpreters, by reducing indirect jumps to improve branch prediction. While these approaches help improve the runtime performance of executing dynamic languages, they still do not influence how such languages are amenable to static analysis.

13.3 Custom Data Structures

Xu and Rountev evaluated a regression test selection analysis for AspectJ [60]. They found the analysis to be extremely imprecise when based on call graphs constructed from bytecode generated from AspectJ code. To improve precision, they introduced the *interaction graph*, a structure similar to a call graph that explicitly models AspectJ features, and evaluated an analysis for constructing such graphs from AspectJ source code [61].

14 Conclusions and Future Work

Getting free program analysis infrastructure for a wide range of languages is an attractive prospect, and we have investigated whether JVM bytecode based analysis for Java can be used on other languages that compile to bytecode. We show that this is indeed possible for a wide range of statically-typed and dynamically-typed languages, based on our results for functional Scheme, object-oriented Scala, and polymorphic OCaml. Call graphs for these languages are as sound as for Java, and present similar challenges for obtaining precision. This suggests that bytecode-based analysis could serve as a useful implementation vehicle for applications such as bug-finding, security analysis, and code navigation in IDEs for languages where program analysis infrastructure is not readily available otherwise.

However, we demonstrate that the implementation of the bytecode generation is crucial, and complex, reflection-heavy implementations prevent good analysis for Groovy, Clojure, Python, and Ruby. We also show performance results indicating that these implementations tend to result in poor performance as well.

Overall, the results are encouraging in that high-quality JVM-based implementations can benefit not only from the JVM's mature implementation, but from its associated mature program analysis infrastructure as well. While the experiments in this paper focus on call graph construction, we consider our conclusions to be broadly applicable to bytecode-based interprocedural static analyses, because call graphs are a prerequisite for most static analyses.

When designing new programming languages and implementing compilers for them, designers consider and balance many different objectives for each particular language. The ability to analyze compiled code using existing JVM analysis tools can be one such objective. The results of our study show how easily the various possible compilation strategies can be analyzed by existing JVM analysis tools. Thus, they provide the data that language implementers need to balance the goal of analyzability against other objectives in the implementation of each new programming language. The data can be used to initiate discussion and future research about the relative benefits and drawbacks of various compilation strategies in the context of specific new programming languages.

ACKNOWLEDGMENTS

We are grateful to Laurie Hendren for helpful suggestions. This research was supported by the Natural Sciences and Engineering Research Council of Canada and the Ontario Ministry of Research and Innovation. This research was also supported in part by National Science Foundation grant CCF-1715153 and by Office of Naval Research (ONR) grant N00014-17-1-2945.

REFERENCES

[1] Stuart Halloway and Aaron Bedra. *Programming Clojure (2. ed.)*. Pragmatic Bookshelf, 2012.

[2] Kenneth Barclay and John Savage. *Groovy programming: an introduction for Java developers (1. ed.)*. Morgan Kaufmann, 2006.

[3] Yaron Minsky, Anil Madhavapeddy, and Jason Hickey. *Real-World OCaml: Functional Programming for the Masses (1. ed.)*. O'Reilly, 2013.

[4] Mark Lutz. *Learning Python (5. ed.)*. O'Reilly, 2013.

[5] Dave Thomas, Andy Hunt, and Chad Fowler. *Programming Ruby 1.9 & 2.0: The Pragmatic Programmer's Guide (4. ed.)*. Pragmatic Bookshelf, 2013.

[6] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala (2. ed.)*. Artima Press, 2011.

[7] R. Kent Dybvig. *The Scheme Programming Language (4. ed.)*. MIT Press, 2009.

[8] Chord: A program analysis platform for Java. Available from http://www.cc.gatech.edu/~naik/chord.html.

[9] Martin Bravenboer and Yannis Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In *OOPSLA*, pages 243–262. ACM, 2009.

[10] Raja Vallée-Rai, Etienne Gagnon, Laurie J. Hendren, Patrick Lam, Patrice Pominville, and Vijay Sundaresan. Optimizing Java Bytecode Using the Soot Framework: Is It Feasible? In *CC*, pages 18–34, 2000.

[11] T.J. Watson Libraries for Analysis WALA. Available from http://wala.sourceforge.net/.

[12] Michael Eichberg and Ben Hermann. A software product line for static analyses: the OPAL framework. In *SOAP@PLDI*, pages 2:1–2:6. ACM, 2014.

[13] Julian Dolby, Mandana Vaziri, and Frank Tip. Finding bugs efficiently with a SAT solver. In Ivica Crnkovic and Antonia Bertolino, editors, *Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2007, Dubrovnik, Croatia, September 3-7, 2007*, pages 195–204. ACM, 2007.

[14] Omer Tripp, Marco Pistoia, Stephen J. Fink, Manu Sridharan, and Omri Weisman. TAJ: effective taint analysis of web applications. In Michael Hind and Amer Diwan, editors, *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*, pages 87–97. ACM, 2009.

[15] Asger Feldthaus, Max Schäfer, Manu Sridharan, Julian Dolby, and Frank Tip. Efficient construction of approximate call graphs for JavaScript IDE services. In *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, pages 752–761, 2013.

[16] Peter F. Sweeney and Frank Tip. Extracting library-based object-oriented applications. In *ACM SIGSOFT Symposium on Foundations of Software Engineering, an Diego, California, USA, November 6-10, 2000, Proceedings*, pages 98–107, 2000.

[17] Julian Dolby and Andrew A. Chien. An automatic object inlining optimization and its evaluation. In Monica S. Lam, editor, *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Vancouver, Britith Columbia, Canada, June 18-21, 2000*, pages 345–357. ACM, 2000.

[18] The Computer Language Benchmarks Game. Available from http://benchmarksgame.alioth.debian.org.

[19] Olin Shivers. Control-flow analysis in scheme. In *PLDI*, pages 164–174, 1988.

[20] Ondrej Lhoták. Comparing call graphs. In *PASTE*, pages 37–42, 2007.

[21] 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 *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*, pages 241–250, 2011.

[22] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondrej Lhoták, José Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Mvller, and Dimitrios Vardoulakis. In defense of soundiness: a manifesto. *Communications of the ACM*, 58(2):44–46, 2015.

[23] The OPAL Project. Available from http://www.opal-project.de/DeveloperTools.html.

[24] Soot - A Java optimization framework. Available from https://github.com/sable/soot.

[25] The Kawa Scheme language. Available from http://www.gnu.org/software/kawa/.

[26] Karim Ali, Marianna Rapoport, Ondrej Lhoták, Julian Dolby, and Frank Tip. Constructing call graphs of Scala programs. In *ECOOP*, pages 54–79, 2014.

[27] Andreas Sewe, Mira Mezini, Aibek Sarimbekov, and Walter Binder. Da capo con Scala: design and analysis of a Scala benchmark suite for the Java Virtual Machine. In *OOPSLA*, pages 657–676, 2011.

[28] Xavier Clerc. OCaml-Java: an ML implementation for the Java ecosystem. In *PPPJ*, pages 45–56, 2013.

[29] Ondřej Lhoták and Laurie J. Hendren. Context-Sensitive Points-to Analysis: Is It Worth It? In *CC*, pages 47–64, 2006.

[30] Ole Agesen. The Cartesian Product Algorithm: Simple and precise type inference of parametric polymorphism. In *ECOOP*, pages 2–26, 1995.

[31] The Clojure programming language. http://clojure.org/.

[32] Josh Juneau. Polyglot Programmer: Jython 101 – A Refreshing Look at a Mature Alternative. *Oracle Java Magazine*, 2013. Available from http://www.oraclejavamagazine-digital.com.

[33] JRuby: The Ruby programming language on the JVM. http://jruby.org/.

[34] Code Tools: jmh. Available from http://openjdk.java.net/projects/code-tools/jmh/.

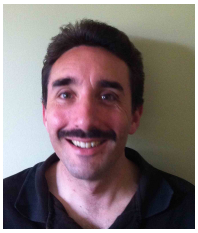
[35] David Grove and Craig Chambers. A framework for call graph construction algorithms. *ACM Transactions on Programming Languages and Systems*, 23(6):685–746, 2001.
[36] Michael Reif, Florian Kübler, Michael Eichberg, and Mira Mezini. Systematic evaluation of the unsoundness of call graph construction algorithms for Java. In *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops, ISSTA 2018, Amsterdam, Netherlands, July 16-21, 2018*, pages 107–112, 2018.
[37] Li Sui, Jens Dietrich, Michael Emery, Shawn Rasheed, and Amjed Tahir. On the soundness of call graph construction in the presence of dynamic language features - A benchmark and tool evaluation. In *Programming Languages and Systems - 16th Asian Symposium, APLAS 2018, Wellington, New Zealand, December 2-6, 2018, Proceedings*, pages 69–88, 2018.
[38] George Fourtounis, George Kastrinis, and Yannis Smaragdakis. Static analysis of Java dynamic proxies. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018*, pages 209–220, 2018.
[39] George Fourtounis and Yannis Smaragdakis. Deep static modeling of invokedynamic. In *33rd European Conference on Object-Oriented Programming, ECOOP 2019, July 15-19, 2019, London, United Kingdom.*, pages 15:1–15:28, 2019.
[40] Wing Hang Li, David Robert White, and Jeremy Singer. JVM-hosted languages: they talk the talk, but do they walk the walk? In *PPPJ*, pages 101–112, 2013.
[41] Aibek Sarimbekov, Andrej Podzimek, Lubomir Bulej, Yudi Zheng, Nathan Ricci, and Walter Binder. Characteristics of dynamic JVM languages. In *VMIL*, pages 11–20, 2013.
[42] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khan, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony L. Hosking, Maria Jump, Han Bok Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanovic, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA*, pages 169–190, 2006.
[43] Gregor Richards, Sylvain Lebresne, Brian Burg, and Jan Vitek. An analysis of the dynamic behavior of JavaScript programs. In *PLDI*, pages 1–12, 2010.
[44] Gregor Richards, Christian Hammer, Brian Burg, and Jan Vitek. The eval that men do - a large-scale study of the use of eval in JavaScript applications. In *ECOOP*, pages 52–78, 2011.
[45] Mark Hills, Paul Klint, and Jurgen J. Vinju. An empirical study of PHP feature usage: a static analysis perspective. In *ISSTA*, pages 325–335, 2013.
[46] Michel Schinz. *Compiling Scala for the Java Virtual Machine*. PhD thesis, EPFL, 2005.
[47] Iulian Dragos. *Compiling Scala for Performance*. PhD thesis, IC, Lausanne, 2010.
[48] Gilles Dubochet and Martin Odersky. Compiling structural types on the JVM: a comparison of reflective and generative techniques from Scala's perspective. In Ian Rogers, editor, *Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, pages 34–41, 2009.
[49] Iulian Dragos and Martin Odersky. Compiling generics through user-directed type specialization. In *Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, pages 42–47. ACM, 2009.
[50] Frank McCabe and Michael Sperber. Feel different on the Java platform: the star programming language. In *PPPJ*, pages 89–100, 2013.
[51] Martin Odersky and Philip Wadler. Pizza into Java: Translating theory into practice. In *POPL*, pages 146–159, 1997.
[52] Martin Odersky, Enno Runne, and Philip Wadler. Two ways to bake your Pizza - translating parameterised types into Java. In Mehdi Jazayeri, Rüdiger Loos, and David R. Musser, editors, *Generic Programming*, volume 1766 of *Lecture Notes in Computer Science*, pages 114–132. Springer, 1998.
[53] Alexander Yermolovich, Andreas Gal, and Michael Franz. Portable execution of legacy binaries on the Java Virtual Machine. In *PPPJ*, pages 63–72, 2008.
[54] Andrew D. Gordon and Don Syme. Typing a multi-language intermediate code. In *POPL*, pages 248–260, 2001.
[55] Michael Bebenita, Florian Brandner, Manuel Fähndrich, Francesco Logozzo, Wolfram Schulte, Nikolai Tillmann, and Herman Venter. Spur: a trace-based JIT compiler for CIL. In *OOPSLA*, pages 708–725, 2010.
[56] José G. Castaños, David Edelsohn, Kazuaki Ishizaki, Priya Nagpurkar, Toshio Nakatani, Takeshi Ogasawara, and Peng Wu. On the benefits and pitfalls of extending a statically typed language JIT compiler for dynamic scripting languages. In *OOPSLA*, pages 195–212, 2012.
[57] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. One VM to rule them all. In *OOPSLA Onward!*, pages 187–204, 2013.
[58] GraalVM - Run Programs Faster Anywhere. Available from https://www.graalvm.org.
[59] Gülfem Savrun-Yeniçeri, Wei Zhang, Huahan Zhang, Eric Seckler, Chen Li, Stefan Brunthaler, Per Larsen, and Michael Franz. Efficient hosted interpreters on the JVM. *TACO*, 11(1):9, 2014.
[60] Guoqing (Harry) Xu and Atanas Rountev. Regression test selection for AspectJ software. In *ICSE*, pages 65–74, 2007.
[61] Guoqing (Harry) Xu and Atanas Rountev. AJANA: a general framework for source-code-level interprocedural dataflow analysis of AspectJ software. In *AOSD*, pages 36–47, 2008.

Karim Ali obtained his PhD degree from the University of Waterloo in 2014. He is an Assistant Professor in the Department of Computing Science at the University of Alberta. His research interests are in programming languages and software engineering, particularly in scalability, precision, and usability of program analysis tools. His work ranges from developing new theories for scalable and precise program analyses to applications of program analysis in security and Just-in-Time compilers.

Xiaoni Lai obtained an MMath degree from the University of Waterloo in 2015. She is currently a Software Engineer at Google, working on Blink, the web rendering engine of the Chrome browser.

Zhaoyi Luo obtained an MMath degree from the University of Waterloo in 2015. He is currently a Software Engineer at Microsoft, working on the Azure SQL Database.

Ondřej Lhoták obtained a PhD from McGill University in 2006. He is an Associate Professor in the D. R. Cheriton School of Computer Science at the University of Waterloo. His research interests are in static program analysis, particularly of object-oriented languages.



Julian Dolby received his PhD degree from the University of Illinois at Urbana-Champaign in 2000. He has been a Research Staff Member at IBM's Thomas J. Watson Research Center ever since. His research interest include static program analysis, software testing and the semantic web. He has also worked on the Jikes Research Virtual Machine (Jikes RVM).

Frank Tip received the PhD degree from the University of Amsterdam in 1995. He is currently a Professor at the College of Computer and Information Science at Northeastern University. His research interests include program analysis, refactoring, test generation, fault localization, and automated program repair.