Deep Static Modeling of `invokedynamic`

George Fourtounis
University of Athens, Department of Informatics and Telecommunications, Greece
gfour@di.uoa.gr

Yannis Smaragdakis
University of Athens, Department of Informatics and Telecommunications, Greece
smaragd@di.uoa.gr

Abstract

Java 7 introduced programmable dynamic linking in the form of the `invokedynamic` framework. Static analysis of code containing programmable dynamic linking has often been cited as a significant source of unsoundness in the analysis of Java programs. For example, Java lambdas, introduced in Java 8, are a very popular feature, which is, however, resistant to static analysis, since it mixes `invokedynamic` with dynamic code generation. These techniques invalidate static analysis assumptions: programmable linking breaks reasoning about method resolution while dynamically generated code is, by definition, not available statically. In this paper, we show that a static analysis can predictively model uses of `invokedynamic` while also cooperating with extra rules to handle the runtime code generation of lambdas. Our approach plugs into an existing static analysis and helps eliminate all unsoundness in the handling of lambdas (including associated features such as method references) and generic `invokedynamic` uses. We evaluate our technique on a benchmark suite of our own and on third-party benchmarks, uncovering all code previously unreachable due to unsoundness, highly efficiently.

2012 ACM Subject Classification Software and its engineering → Compilers; Theory of computation → Program analysis; Software and its engineering → General programming languages

Keywords and phrases static analysis, invokedynamic

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2019.27

Funding We gratefully acknowledge funding by the European Research Council, grants 307334 (SPADE) and 790340 (PARSE), a Facebook Research and Academic Relations award, and an Oracle Labs collaborative research grant.

1 Introduction

Object-oriented and functional programming have combined in recent years to produce hybrid programming languages. Some of these, such as Scala [45], are new languages, designed from the ground up to incorporate features of both programming paradigms. Others, for instance Java [23] and C# [25], have adapted to the demand for functional features by carefully adding them in an existing language design; examples of this evolution are lambdas [49] and the streams API [74] in the Java platform and the Language Integrated Query (LINQ) facility in the .NET ecosystem [36].

On another axis, programming languages occupy different places in the spectrum between static and dynamic typing. At the extremes, programming languages either have to supply static ("type") information for every entity in the program, or do away with all such types, in a completely dynamic coding style. In practice, most programming languages are closer to the middle, having a fundamental static or dynamic design, while mixing elements from the opposite approach. For example, the Java Virtual Machine (JVM), the best-established language runtime system, supports dynamic facilities, such as reflection and dynamic class loading, that offer significant flexibility, outside the control of the static type system.

A recent dynamic facility added to the JVM, in order to combine flexibility with highly optimized performance, is that of programmable method resolution and dynamic linking, in the form of the `invokedynamic` instruction [62]. The `invokedynamic` instruction and its accompanying `java.lang.invoke` framework permit the expression of fully dynamic behavior, in much the same way as traditional Java reflection. However, whereas reflection can be thought of as dynamically *interpreting* dispatch logic, programmable linking can be thought of as dynamically *compiling* dispatch logic, transforming call sites at load time with decisions possibly cached and subsequently executed at full speed. This facility enables the JVM to support dynamic language patterns with great efficiency. As a result, the framework has also been used to implement Java lambdas—the newly-added functional feature of the language.[1]

Dynamic features are welcome by many programmers as they offer a needed flexibility. However, they come at a cost: static reasoning is greatly hindered. For instance, static analysis tools for Java are largely ineffective when faced with `invokedynamic` code, although static analysis has long dealt with (statically-typed) dynamic dispatch (a.k.a. *virtual dispatch*) facilities. Virtual method resolution in statically-typed bytecode is much easier to analyze, compared to purely dynamic code that lacks explicit method signatures. (Virtual dispatch in standard object-oriented languages performs a dynamic lookup of the function, based on its name, signature, and the type hierarchy. This is still significantly friendlier to static reasoning than completely dynamic calls, of functions with possibly statically-unknown names or types.)

These problems of static reasoning for the dynamic features of the JVM (and, by extension, its functional lambdas) have been well identified. In recent work, *Reif et al.* [60] and *Sui et al.* [68] describe the unsoundness in the construction of call graphs for Java, caused by features such as lambdas and `invokedynamic`. These features are not going away: in a recent study, Mazinanian *et al.* [35] "found an increasing trend in the adoption rate of lambdas." Also, Holzinger *et al.* found method handles, a core part of the `invokedynamic` framework, to pose "a risk to the secure implementation of the Java platform" [26]. This is a design problem: to control performance overhead, method handles are less secure by design, compared to the core reflection API [65].

In this paper, we propose a static analysis that can successfully analyze both the `invokedynamic` framework and its particular combination with generated code in Java lambdas. Our analysis cooperates with an existing points-to analysis and an existing reflection analysis (when needed), in mutually recursive fashion. The analysis also simulates parts of the Java API that either do dynamic code generation or call native code, to maintain soundness. Finally, we supply a special static analysis extension that can analyze lambdas and method references, without any reflection support. This last feature permits the static analysis of large Java code bases without paying the performance overhead of reflection reasoning.

In more detail, our work makes the following contributions:

- We offer the first static analysis that handles general-purpose `invokedynamic`—the basis of modern dynamic features of Java. The static analysis operates at a deep level: it

[1] We emphasize again that the concepts of lambdas (a functional language feature) and programmable linking (a dynamic language implementation technique) are orthogonal. Lambdas could be implemented via front-end class generation, dynamic code generation plus traditional virtual dispatch, or other similar techniques. They are implemented using programmable linking in Oracle's JDK only as a matter of choice, since the mechanism is flexible, powerful, and efficient.

includes full modeling of the underlying `java.lang.invoke` framework: a DSL-like facility for capturing and manipulating methods as values.
- We present a static modeling of Java lambdas—the main functional feature of Java. Although lambdas and `invokedynamic` are conceptually orthogonal, in practice lambdas are implemented using `invokedynamic`, making the analyses of the two features closely interrelated. Still, the analysis of lambdas is not a mere client of the general-purpose `invokedynamic` analysis, since it both needs extra modeling (for generated code) and admits more efficient implementation, due to its specialized use of `invokedynamic`, eschewing the need for complex reflection reasoning.
- The analysis is accompanied by a micro-benchmarking suite covering many patterns found in realistic uses of lambdas and `invokedynamic`. The suite is independently usable for validation of static support of these features.
- The analysis is evaluated on the third-party suite of *Sui et al.* [68], which was designed for showcasing the unsoundness of call-graph construction under dynamic and functional Java features. Our analysis models all general-purpose uses of `invokedynamic` and fully models uses of lambdas.

This paper is structured as follows: we first present a set of examples that explain how the dynamic and functional features of Java work (Section 2) and proceed to give a more technical background of these features (Section 3). We then present our technique for the static analysis of these features, in a declarative analysis framework (Section 4). We evaluate our model (Section 5), connect with related work (Section 6), and conclude (Section 7).

2 Motivation and Illustration

This section introduces `invokedynamic` and Java lambdas with the help of examples.

2.1 Motivating Example 1: Late Linking

A common use of dynamic linking is for breaking dependencies between pieces of code so that they do not have to be compiled together. An example of Java code using `invokedynamic` to break a compile-time dependency is shown in Figure 1. Since Java does not permit `invokedynamic`-equivalent expressions at the source level,[2] we use in the example an INVOKEDYNAMIC pseudo-intrinsic that contains the following information:
- a dynamic name (`print`),
- a method type (`(A)V`),
- a list of arguments (just `this.obj` here),
- a bootstrap method signature (here: `<A: CallSite bootstrap(MethodHandles.Lookup, String, MethodType)>`), and
- a list of bootstrap arguments (empty in this example).

While the code without `invokedynamic` has to explicitly state which version to call (and thus store an immutable signature in an `invokestatic` in the bytecode), the code using `invokedynamic` looks up the method programmatically, via a "bootstrap" method, which initializes the call site. (This lookup could be arbitrarily complex, although in this example the outcome is always the same.) Here we note that the programmer could also use classic Java reflection to do a similar lookup-and-invoke (retrieving a `Method` metaobject

[2] A proposal is underway to allow such expressions via intrinsics [21].

and calling an `invoke` method on it), but that would be inefficient, since standard Java reflection contains an interpretive layer of introspection. In contrast, `invokedynamic` can be compiled away: the bootstrap method is executed at *load time*, not run time (i.e., not when method `run` is invoked, but when it is loaded). The bootstrap method effectively acts as a load-time macro, accepting as arguments load-time constants (e.g., string constants) or fragments of uninterpreted expression syntax. This bootstrap method returns a "constant call site", which the JVM can inline in place of the `invokedynamic` call as needed, similar to having the `invokestatic` call that is missing from the bytecode.

2.2 Motivating Example 2: Lambdas

For a simple program that creates and uses a lambda, we can take the following example (adapted from the dynamic benchmark of *Sui et al.* [68]):

```
import java.util.function.Consumer;
public class LambdaConsumer {

  public void source() {
    Consumer<String> c = (input) -> target(input);
    c.accept("input");
  }

  public void target(String input) { }

}
```

Here, method `source()` creates a lambda that consumes a string value. The lambda takes an `input` parameter and calls method `target()` in its body, passing the parameter to the callee.

The arrow syntax declares a lambda function, which is rather a mismatch for object orientation: it looks like a bare method, without an instance or declaring type. However, that syntax behind the scenes constructs an object of type `Consumer`, as shown by the static type of variable `c`. This type is one of the "functional interfaces" [16] provided by Java, which are interface types that have a functional flavor, i.e., declare a single method. Generic typing helps with annotating uses of such instances (as with the type parameter of `Consumer` here).

Indeed, the Consumer type declares a single `accept` method that takes a `String`. Calling that method on a lambda should then evaluate the body of the lambda with the appropriate parameter passed to it. If we were to inline the code in the body of `source()` to eliminate the lambda, it would read:

```
public void source() {
  target("input");
}
```

However, such inlining cannot happen in the general case: lambdas are often passed to code or returned by it, to be applied in a location remote to their origin. Reasoning about the code above is thus based on non-local (possibly whole-program) reasoning about the "functional object" that was created and assigned to variable `c`.

In Figure 2, we see the bytecode generated for the two statements in the body of `source` in our example. For presentation purposes, instead of stack-based bytecode, we use the friendlier 3-address Jimple intermediate language [75]. (In the Jimple syntax, the `invokedynamic` JVM instruction is denoted `dynamicinvoke`.) We observe that the call that generates the lambda does the following:

Code without invokedynamic

```
class C implements Runnable {
  A obj;

  C(A obj) {
    this.obj = obj;
  }

  void run() {
    A.print(this.obj);        // Direct call
  }
}

class A {
  public static void print(A a) { }
}

(new C(new A())).run();
```

Code using invokedynamic

```
class C implements Runnable {
  A obj;

  C(A obj) {
    this.obj = obj;
  }

  void run() {
    INVOKEDYNAMIC "print" "(A)V" [this.obj]
      <A: CallSite bootstrap(MethodHandles.Lookup,String,MethodType)>
      []
  }
}

class A {
  public static void print(A a) { }
  public static CallSite bootstrap(MethodHandles.Lookup caller,
                                   String name, MethodType type) {
    MethodType mt = MethodType.methodType(Void.TYPE, A.class);
    MethodHandles.Lookup lookup = MethodHandles.lookup();
    MethodHandle handle = lookup.findStatic(A.class, name, mt);
    return new ConstantCallSite(handle);
  }
}

(new C(new A())).run();
```

Figure 1 Example: using `invokedynamic` to postpone linking of a method call.

```
l0 := @this: LambdaConsumer;

l1 = dynamicinvoke "accept" <java.util.function.Consumer (LambdaConsumer)>(l0)
       <java.lang.invoke.LambdaMetafactory: java.lang.invoke.CallSite metafactory(
         java.lang.invoke.MethodHandles$Lookup,java.lang.String,java.lang.invoke.MethodType,
         java.lang.invoke.MethodType,java.lang.invoke.MethodHandle,java.lang.invoke.MethodType)>
       (class "(Ljava/lang/Object;)V",
        handle: <LambdaConsumer: void lambda$source$0(java.lang.String)>,
        class "(Ljava/lang/String;)V");

interfaceinvoke l1.<java.util.function.Consumer: void accept(java.lang.Object)>("input");
```

Figure 2 The `invokedynamic` behind a lambda creation (Jimple syntax for example in Section 2.2).

- It invokes as a bootstrap method (i.e., the method to execute at load-time over the site of `invokedynamic`) a special "lambda metafactory" method. Again, this is a method executing at load time (i.e., akin to a macro). It processes the call site directly and returns a `CallSite` value, not the `Consumer` value of the user code.
- It passes to the lambda metafactory enough information to specify what kind of lambda needs to be generated: one with an "accept" method, implementing interface `Consumer` and capturing from its environment parameter `l0`. The `l0` capture means that the current value of `this` escapes to the new code that will construct the lambda. This is to be expected, as the lambda body needs a receiver to resolve the call to `target`.
- A method handle pointing to a compiler-generated method `lambda$source$0` is also passed as an argument. This method encodes the body of the lambda expression.

Note that `invokedynamic` is used at the site of lambda generation, not lambda invocation. The latter (in the final line of Figure 2) is a regular interface call.

From the perspective of a static analysis, the only method call that can be resolved in the `invokedynamic` instruction is the call to the metafactory but analysis of that cannot complete: the metafactory does load-time code generation. The compiler-metafactory synergy (of generating methods at compile time, yet leaving other code generation and call-site transformation to load time) is a design that cannot be penetrated by a conventional static analysis. When, in the next instruction, the static analysis tries to analyze the interface call on the object returned in the `invokedynamic` instruction, it cannot resolve the target method and analysis of this call fails.

2.3 Motivating Example 3: Method References

Java 8 introduced lambdas due to popular demand for the feature but also because they were needed for scaling stream processing over multicore hardware [17]. Streams were another new functional feature added to Java, that supported combinator functions over series of data ("streams"), enabling function composition and higher-order programming idioms. An example of streams and lambdas is the following snippet from Urma's streams tutorial [74]:

```
List<Integer> transactionsIds = transactions
  .stream()
  .filter(t -> t.getType() == Transaction.GROCERY)
  .sorted(comparing(Transaction::getValue).reversed())
  .map(Transaction::getId)
  .collect(toList());
```

Here, we see that function `filter` takes a lambda using the arrow syntax. We also see another higher-order feature added in Java 8: method references, such as `Transaction::getValue` and `Transaction::getId`. These pass regular methods as function parameters to combinator functions `comparing` and `map`.

While the syntax of method references is different compared to lambdas, these expressions are also implemented by the lambda metafactory in a similar way. Method references may be a simplified version of lambdas but they still have semantic complexities as they can capture a value from the environment for their receiver.

2.4 Motivating Example 4: SAM Conversion

The use of lambdas (and, by extension, `invokedynamic`) in Java is not limited to pure functional programming patterns. Lambdas are backwards compatible with pre-Java-8 code. In the following example, we see two `Runnable` objects being constructed, both with the same functionality:

```
public class Main {
  public static void main(String[] args) {

    // Use anonymous class.
    Runnable a = new Runnable() {
                   public void run() {
                     System.out.println("Hello.");
                   }
                 };
    a.run();

    // Use a lambda.
    Runnable b = (() -> System.out.println("Hello."));
    b.run();

  }
}
```

The `Runnable` interface is a standard type of the Java platform that happens to have a single method. It is, thus, a "single abstract method" ("SAM") type[3] and the lambda syntax can be used to generate an instance of it, which can be passed to code compiled with older Java versions. This approach makes pre-Java-8 code "forward-compatible to lambdas" [17] by viewing all existing single-method interfaces as lambdas ("SAM conversion" [14, 49]). In practice, this ease of constructing many types as lambdas means that, even in a simple "hello world" Java program, several `invokedynamic` calls to the lambda metafactory take place.

This has caused a regression in the power of static analysis tools on bytecode: *unless it supports lambdas, an analysis may find fewer facts for the same program under Java 8, compared to Java 7.* Java has become more dynamic and functional under the hood.

3 Technical Background

This section gives a basic background on the technology behind method handles, the `invokedynamic` framework, lambdas, and method references. We show as much as needed

[3] Or a "functional interface" [19].

for the needs of the model of the static analysis that will follow.

3.1 Method Handles and Method Types

Two important kinds of values that are used in the rest of this section are method handles and method types.

Method handles are the equivalent of type-safe function pointers [64] and a lightweight alternative to standard reflective method objects [41]. They represent targets for invocation that can point to methods, constructors, fields, or other parts of an object [64]. There are three basic kinds of method handles: direct method handles are very similar to pointers; bound method handles are partial applications of methods [46, 63], and adapter method handles perform various adjustments of method parameters (e.g., from a flat list of arguments to a single argument array) [63].

Method types are type descriptors that help method handle invocations guarantee run-time type safety. A method type describes the parameter types and the return type that a method handle can accept. Method types can be modified to produce new method types: for example, their return type can be changed and types can be dropped, changed, or appended [54].

A method handle can be invoked via two methods called on it:

- `invokeExact()` calls the method handle directly, matching its types against the handle method type.
- `invoke()` is more permissive: it permits conversions of arguments and return type during the method handle invocation. Such conversions must be compatible with appropriate conversions of its method type [52].

The general `java.lang.invoke` API [47], offers ways to compose method handles, convert, fill in, or rearrange their arguments, perform conditional logic on them, or manipulate them in other ways. In practice, the method handles API is an embedded domain-specific language (DSL), which has the flavor of a combinatorial language over functional types. This DSL does deep embedding [69], i.e. the API creates an intermediate representation that reflects the semantics of the intended method handle.

The method handles are translated to an intermediate representation called *lambda forms* [48, 27]. (Not to be confused with the synonymous "lambda" high-level functional language feature of the language that we discuss extensively in this paper.) The lambda form representations can be cached and reused, interpreted, or compiled (using Just-in-Time technology). *This aspect of method handles argues for a static analysis to model them as primitive concepts: since they eventually do dynamic code generation, their semantics are impenetrable to a conventional static analysis.*

The compilation of lambda forms creates dynamically-generated bytecode of a special form, called *anonymous classes* [61]. This is bytecode that is not even visible to the runtime system class dictionary and is used for fast lightweight code generation [41]. Not only are these classes hidden; they also violate the read-only invariant of loaded classes in the VM, as they can patch other classes on the fly.

This design introduces a complete embedded mini-language on top of bytecode, together with a small implementation (intermediate representation, interpreter, and compilation back-end). For static analysis tools to reason about custom dynamic behavior, they must, thus, reason about this small language, from its front-end API embedding, through the implementation, to the generated bytecode.

3.2 The `invokedynamic` Instruction

The JVM was initially used to implement only the Java language. As the virtual machine became a state-of-the art optimizing Just-in-Time (JIT) compiler and the underlying platform grew, other statically-typed object-oriented languages (such as Scala [45] and Fortress [1]) chose to reuse it by having a compiler front-end from their syntax to bytecode. At the same time, the rise of dynamic languages, combined with the desire of their implementers to reuse the Java platform, led to a proliferation of dynamic languages implemented on top of the JVM, both existing ones such as Ruby (JRuby [44]), Python (Jython [56]), and JavaScript (Rhino/Nashorn [5]), and new ones such as Groovy and BeanShell. In the meantime, functional features entered the mainstream, influencing the object-oriented programming paradigm; functional languages gained enough traction to warrant implementations on top of the JVM. Examples of functional languages on the JVM are Clojure [24], the Haskell-inspired Eta [73] and Frege [76], and the Erjang version of Erlang [72]. Finally, Java itself had to evolve and incorporate functional features (we describe them in detail in Section 3.3).

To become multi-lingual in an efficient way, the JVM design had to gain two new powers: the capability to implement diverse dynamic behaviors; and native support for the basic building block of functional programming, lambdas. In this subsection, we give an overview of `invokedynamic`, while on the next subsection, we will see how the functional features are supported under the hood as an instance of dynamic behavior (Section 3.3).

A classic characteristic of dynamically-typed languages is their reliance on runtime optimization for performance, since there are no statically-available types to use for optimization. Naive implementations of dynamically-typed languages are slow, since they are usually interpreters that constantly query metadata to discover the runtime types of objects in order to perform safe operations on them. Runtime optimization systems come to the rescue: modern high-performance dynamic languages profile the running program and optimize it, often generating good code at runtime, when more information is known about the behavior of the program (the "Just-in-Time" or "JIT" approach).

JIT optimization has a long history, for instance one of its techniques to speed up method calls, "inline caching", appears in the classic implementation of the Smalltalk object-oriented dynamic language [8]. Today, the JIT approach forms the basic technology behind successful implementations as diverse as the cutting-edge Java Virtual Machine [33] or the browser runtimes of JavaScript that enabled the Web 2.0 wave of applications.

As dynamic languages on the JVM were pushing for more performance on the JVM, Java 7 introduced a new bytecode opcode, `invokedynamic` [62], together with an API around it, that could offer the programmer the capability to completely customize dynamic program behavior. The program could now implement its own method dispatch semantics, for example perform linking, unlinking, and relinking of code on the fly, add or remove fields and methods in objects, or implement inline caching using plain Java code. The crucial advantages of this approach, compared to writing adapter code by hand, are not only in saving engineering effort through a friendly API, but also in informing the JIT optimizer so that better optimizations (such as inlining) can happen across dynamic dispatch borders.

Oracle offers this as motivation: *"The invokedynamic instruction simplifies and potentially improves implementations of compilers and runtime systems for dynamic languages on the JVM. The invokedynamic instruction does this by allowing the language implementer to define custom linkage behavior. This contrasts with other JVM instructions such as invokevirtual,*

in which linkage behavior specific to Java classes and interfaces is hard-wired by the JVM."[4]

Dynamic languages on the JVM were naturally the first users of this new functionality (JRuby [40, 77], Jython [3], the Nashorn JavaScript engine [31], Groovy [71], Redline Smalltalk [43], and a significant subset of PHP [12]), as they could improve their performance [55]. The `invokedynamic` instruction even inspired the creation of at least one new JVM-based language [58]. Moreover, this new capability was used for other applications, such as live code modification [59], aspect-oriented programming [39], context-oriented programming [2, 34], multiple dispatch/multi-methods (a generalization of object-oriented dynamic dispatch to take more than one method arguments into consideration when choosing the target method of an invocation) [42], lazy computations [42, 15], generics specialization [20], implementation of actors [38], and dynamically adaptable binary compatibility via cross-component dynamic linking [28]. This new low-level functionality also became available for programmable high-level dynamic linking and metaobject protocol implementation via the Dynalink library [70].

Informally, `invokedynamic` can be seen as configurable initialization (and possible reconfiguration) of invocations in Java bytecode. When the JVM loads a class, it resolves every `invokedynamic` instruction in it. For every `invokedynamic` instruction:

1. A special *bootstrap method* is called. The method reads information either embedded in the instruction or coming from the constant pool of the class.
2. The bootstrap method returns a *call site* object. That object belongs to the instruction location in the bytecode and contains a method handle.
3. Since the call site contains a method handle, the invocation is resolved now and the call site has been linked. The method handle can be thus invoked (see Section 3.1).
4. The call site is a Java object, so the program can access it and can later mutate its method handle so that the invocation is effectively re-linked to resolve to another method. This is essential for modeling fully dynamic behavior (e.g., making an object support an extra method during run time).

The model above means that the program can now control the linking of method calls. Moreover, this framework makes dynamically-linked invocations efficient. Since the JVM internally supports `invokedynamic`, it can optimize such invocations. For example, if the call site is a constant call site,[5] the invocation can be inlined. The efficiency of `invokedynamic` invocations has been confirmed by Kaewkasi [29] and Ortin *et al.* [55].

3.3 Method References and Lambdas

As seen in the examples of Section 2, method references and lambdas are functional programming features added to Java for more expressive power. Eventually, Java 8 implemented these two features with `invokedynamic` [15]. A crucial motivation for this implementation choice has been compatibility, i.e., to avoiding a commitment to a single bytecode-visible implementation of lambdas (e.g., as classes). Describing the implementation of lambdas in terms of `invokedynamic` gives the Java compiler developers the freedom to later change the underlying implementation, without breaking binary compatibility [15, 17]. The only trace of the translation of lambdas inside the bytecode is an `invokedynamic` call to a specific lambda metafactory, but the code emitted by that may later change.

[4] `https://docs.oracle.com/javase/7/docs/technotes/guides/vm/multiple-language-support.html#invokedynamic`

[5] `https://docs.oracle.com/javase/8/docs/api/java/lang/invoke/ConstantCallSite.html`

Both lambdas and method references use the same implementation technique: `invokedynamic` sites that use special bootstrap methods, the lambda metafactories [50]. A lambda metafactory initializes a call site so that it contains a lambda factory, i.e., it can generate functional objects. The Java 8 lambda metafactory generates an inner class that implements the functional interface.[6] The functional objects created can either be stateless or access values from their enclosing environment [18]. The implementation of lambdas is a thin layer of code that only uses small pieces of dynamically-generated code as glue.

In practice, the Java compiler creates appropriate methods for the bodies of lambdas (*implementing methods*) and registers method handles of them in the constant pool. These method handles are then used in `invokedynamic` invocations to the lambda metafactories, together with any values captured from the environment. The lambda metafactories can then create new anonymous classes that can be used to instantiate the functional objects and forward method calls to the implementing methods.

4 Static Analysis

We next present our model for handling the `java.lang.invoke` API (i.e., method handles), the `invokedynamic` instruction (in general), as well as Java lambdas. We offer a declarative set of inference rules that appeal to relations defined and used by an underlying value-flow/points-to static analysis. Our implementation is on the declarative Doop framework [6], so it is to a great extent isomorphic to the analysis model presented.

The essence of our analysis approach is threefold:

- Our baseline model gives semantics to method handles. (This is also the main novelty of our approach: the deep modeling of the `java.lang.invoke` API at its most fundamental level.) This requires appealing to an existing value flow analysis, since method handles have no hard-coded signatures in the bytecode: they offer `invoke` operations that are "signature-polymorphic". Therefore, any resolution of method handles requires a static model of all possible signature arguments to `invoke` instructions. Modeling the semantics of method handles is necessary since their implementation is un-analyzable, relying on run-time code generation (via the aforementioned "lambda forms"). Furthermore, this model requires static analysis of Java reflection, since method handles can also be looked up via reflection operations (e.g., by method types generated via reflective class values, or by "unreflecting" method objects into method handles).
- Based on the modeling of method handles, we straightforwardly model `invokedynamic` as an invocation of a method handle computed by a bootstrap method.
- Reasoning about lambdas appeals to a part of the `invokedynamic` reasoning. However, modeling lambdas both requires extra reasoning (because of dynamic code generation) and can avoid the need for expensive reflection analysis, since the method handles computed for lambdas do not employ reflection.

4.1 Model Basics

We assume the following domains and (meta)variables, also listing some simple convenience predicates along the way:

- $s \in S$ are strings.

[6] `https://bugs.openjdk.java.net/browse/JDK-8000806`

- $n, k \in \mathcal{N}$ are numbers.
- The symbol $*$ denotes arguments that can be ignored.
- $v \in V$ are variables, $val \in Val$ are values, $\lambda \in Val$ are functional objects.
- $t \in T$ are types while $t^i \in T^I \subset T$ are interface types. Constructor $mock_c(t, i)$ creates a mock object of type t that corresponds to an instruction i. The Class metaobject of a type t is given by $Reified_C(t)$ and is a value.
- $m \in M$ are methods. The formal of m at position n is represented as F_n^m. The special "this" variable of an instance method m is represented as $m/\texttt{this}$. The Method metaobject of a method m is given by $Reified_C(m)$ and is a value. We use the following predicates:
  - $Constr(m)$: m is a constructor method.
  - $Static(m)$: m is a static method.
  - $m \in t$: m is declared in type t.
  - The return variable v of m is represented as $RetVar(v, m)$.
- $m_t = \{t, [t_0, \ldots, t_{n-1}]\} \in M^T, n \geq 0$ are method types, which are pairs of a return type t and a (possibly empty) list of parameter types. Predicate $AsType(m_t^1, m_t^2)$ holds when method type m_t^2 has the same arity as m_t^1, and for every pair t, t' of m_t^1 and m_t^2 (at the same position), it holds that the two types are compatible: $t \leftrightarrows t'$. ($t \leftrightarrows t'$ is one of the analysis's main input predicates from Figure 3.) This type compatibility represents the `asType` rules of the specification [52]. Function $MethodMT(m)$ maps a method m to its method type.
- $i \in I$ are invocation instructions. Predicate $i \in m$ means that instruction i belongs to method m. The actual parameter that is passed at invocation i in position n is represented as A_n^i. For `invokedynamic` instructions, these are the non-bootstrap parameters of the bytecode instruction. If instruction i returns a value, $Ret(i)$ is the variable that will hold the returned value.
- $h \in MH$ are method handles. A method handle h has the form $\langle m, m_t \rangle$, which is a pair of a method m and a method type m_t. We also assume predicate $DMHLookup(t, s, m_t)$, which returns the direct method handle that corresponds to a method with name s, declared in type t, with method type m_t. Constructor $mock_h(t, h)$ creates a mock object of type t that corresponds to method handle h.
- $c \in C$ are call site identifiers. (These are different from mere instructions: because of the dynamic nature of calls, the same instruction can play the role of distinct call sites.)
- We assume lookup objects $\mathcal{L}_t$, one for each type t. These are opaque objects in the `java.lang.invoke` API that are used as intermediate values in a lookup: to retrieve, e.g., a method handle, first one retrieves a lookup object over a type, and subsequently uses it with method-identifying information.[7]

The table in Figure 3 lists the main relations that will be used in the analysis rules (i.e., all relations other than convenience predicates described earlier). We annotate each relation with IN if it is consumed by our rules and OUT if our rules inform it. Relation $v \mapsto val$ is both IN and OUT, since our analysis is mutually recursive with the existing points-to analysis. Relations annotated with INTER are intermediate relations used in the analysis, that may not be externalized.

[7] Maintaining a distinct lookup object for each type also shows that our technique can potentially track access restrictions per type, as mandated by the specification of method lookup objects: `https://docs.oracle.com/javase/8/docs/api/java/lang/invoke/MethodHandles.Lookup.html` .

Relation	Description	Use
$v \mapsto val$	Variable v points to val.	IN, OUT
$f \mapsto val$	Field f points to val.	IN
$v[*] \mapsto val$	Variable v is an array and $v[i]$ points to val for some i.	OUT
$i \xrightarrow{h} m$	Instruction i calls method m using method handle h.	OUT
$i \xrightarrow{\lambda} m$	Instruction i calls method m using functional object λ.	OUT
$t \leftrightarrows t'$	Types t and t' are either subtypes of each other or can be converted to each other via boxing or unboxing.	IN
$CSite(c, i, t)$	Instruction i creates call site c with dynamic return type t.	INTER
$CSite_C(c, h, m)$	Call site c contains method handle h pointing to method m.	INTER
$MetafactoryInvo(i, s, t^i)$	Lambda metafactory invocation at instruction i, with dynamic method name s and functional interface t^i.	INTER
$Lambda(\lambda, m, s, i)$	Functional object λ with implementing method m, dynamic method name s and `invokedynamic` source instruction i.	INTER
$Capture(i, n, val)$	Instruction i captures environment value val at position n.	INTER
$InstanceImpl(i, m, \lambda)$	Functional object λ, generated at instruction i, uses non-static method m as implementing method.	INTER

Figure 3 Analysis relations.

4.2 Model: Method Types and Method Handles

We show how the analysis can understand the APIs of method types and method handles. This includes handling the *polymorphic signatures* of Java bytecode.

A fundamental problem in the static analysis of method handles is that they contain native code, for example their "invoke" methods that must be used to do the method call are native.

The basic relation in this model is $i \xrightarrow{h} m$ which is a call-graph edge from instruction i to method m annotated with a method handle h. This relation is both created by rules (that discover method handle invocations) and consumed by rules (that handle argument passing and value returns).

The method handle invocation rules are shown in Figure 4 while Figure 5 shows the rules that simulate part of the method handles API. For clarity, we omit packages from qualified types (e.g., we write `MethodHandle` instead of `java.lang.invoke.MethodHandle`).

The rules of Figure 4 are relatively straightforward, capturing regular calling semantics for method handle invocations, once a method handle value has been determined. Interesting elements include the mutual recursion with an existing points-to analysis, as well as the construction of new (mock) objects, per the API specification, when a method handle that corresponds to a constructor is invoked.

Rule *MHMETHOD*. This rule creates a method handle h and a method type m_t for every method found in the program.

Rule *MHCGE*. This rule informs the method handles call graph relation that an invocation i calls method m using method handle h (notation: $i \xrightarrow{h} m$).

Rules *RETH* and *MHARGS*. These rules pass arguments and return parameters.

Rule *MHCONSTR*. For method handles that correspond to constructors, a mock value is constructed and both the `this` variable in it and the return value of the invocation point to this value.

$$\frac{m_t = MethodMT(m)}{\langle m, m_t \rangle}\ MHMETHOD$$

$$\frac{i = v.\texttt{<MethodHandle.invokeExact>(...)} \qquad v \mapsto h \qquad h = \langle m, * \rangle}{i \xrightarrow{h} m}\ MHCGE$$

$$\frac{i \xrightarrow{h} m \qquad A_n^i \mapsto val}{F_k^m \mapsto val}\ MHARGS \qquad \frac{i \xrightarrow{h} m \qquad RetVar(v, m) \qquad v \mapsto val \qquad Ret(i) = v'}{v' \mapsto val}\ RETH$$

$$\frac{i \xrightarrow{h} m \qquad Constr(m) \qquad val = mock_h(t, h) \qquad Ret(i) = v}{m/\texttt{this} \mapsto val \qquad v \mapsto val}\ MHCONSTR$$

Figure 4 Rules for handling method handle invocations.

The rules of Figure 5 are a bit more demanding, since they capture precisely the semantics of the `java.lang.invoke` API, including lookup objects, using reflection to retrieve method handles, and more.

Rule *ASTYPE*. This rule models the `asType()` method of the `MethodHandle` API using predicate $AsType(m_t^1, m_t^2)$.

Rule *MHLOOKUP*. This rule models the per-type lookup object needed to find method handles. The `lookup()` method modeled in this rule is caller-sensitive [53], thus the caller type t characterizes the returned lookup object and is available for future uses of the object.

Rule *MHLOOKUPC*. This rule models the connection between a lookup object and its type (e.g., to be used in the code for accessibility checks).

Rule *UNREFLECT*. This rule models the API methods that bridge the Reflection API with the `java.lang.invoke` API. These methods convert reified methods/constructors to method handles.

Rule *MTYPE*. This rule models the two-argument method `methodType()` of class `MethodHandle`. The other overloaded versions of `methodType()` are modeled similarly. The rule needs access to reflection support, since it takes advantage of points-to information that points to reified Class objects.

Reflection Support. A useful subset of these rules does not need reflection support in the analysis. For some programs, method types and method handles may come from the constant pool instead of being looked up by the `java.lang.invoke` API; for such code, our rules do not require reflection support.

`invoke()` vs. `invokeExact()`. As mentioned in Section 3.1, the method handle API offers two different ways to invoke a method handle. The most fundamental is `invokeExact()`, which assumes the arguments and the return value have types that exactly match the method type of the method handle. In contrast, `invoke()` permits conversions in arguments and return values, as if the method handle could successfully change its method type via the `asType()` method. For presentation purposes, we only show the rules for `invokeExact` in Figure 4 and the rules for `asType()` in Figure 5. The handling of `invoke()` follows directly from these rules, accounting for autoboxing in the case of primitive conversions. The

$$\frac{i = v.\texttt{<MethodHandle.asType>}(v') \quad v \mapsto \langle m, m_t^1 \rangle \quad v' \mapsto m_t^2 \quad AsType(m_t^1, m_t^2) \quad Ret(i) = v''}{v'' \mapsto \langle m, m_t^2 \rangle} \; ASTYPE$$

$$\frac{i = \texttt{<MethodHandles.lookup>}() \quad i \in m \quad m \in t \quad Ret(i) = v}{v \mapsto \mathcal{L}_t} \; MHLOOKUP$$

$$\frac{i = v.\texttt{<MethodHandles.Lookup.lookupClass>}() \quad v \mapsto \mathcal{L}_t \quad Ret(i) = v'}{v' \mapsto Reified_C(t)} \; MHLOOKUPC$$

$$\frac{Ret(i) = v' \quad MethodMT(m) = m_t \quad i = \texttt{<MethodHandles.Lookup.}s\texttt{>}(v) \quad v \mapsto Reified_M(m) \quad s \in \{\texttt{unreflect}, \texttt{unreflectSpecial}, \texttt{unreflectConstructor}\}}{v' \mapsto \langle m, m_t \rangle} \; UNREFLECT$$

$$\frac{i = v.\texttt{<MethodHandles.Lookup.}s\texttt{>}(v_0, v_1, v_2) \quad s \in \{\texttt{findVirtual}, \texttt{findStatic}\} \quad v \mapsto \mathcal{L}_t \quad Ret(i) = v' \quad v_0 \mapsto Reified_C(t') \quad v_1 \mapsto s \quad v_2 \mapsto m_t \quad DMHLookup(t', s, m_t) = h}{v' \mapsto h} \; FIND$$

$$\frac{i = v.\texttt{<MethodType.methodType>}(v_0, v_1) \quad v_0 \mapsto Reified_C(t_0) \quad v_1 \mapsto Reified_C(t_1) \quad Ret(i) = v}{v \mapsto \{t_0, [t_1]\}} \; MTYPE$$

Figure 5 Rules for handling part of the method handles API.

handling of `invokedynamic` (shown in Section 4.3) is not affected, since that only needs the functionality of invocations via `invokeExact` [33].

Generalized method handles. Method handles are also able to represent fields; we don't model this behavior here since it is not important for the `invokedynamic` analysis (that follows in the next section) but it is a simple extension of our model.

4.3 Generic Handling of `invokedynamic`

We next discuss the static modeling of `invokedynamic` instructions. The model effects the dynamic linking that eventually computes a method handle and invokes it. The key concept employed is call sites ($c \in C$). These are the return objects of `invokedynamic` bootstrap methods (as determined by regular points-to analysis) and internally use method handles to determine the calling behavior.

Our rules model the `invokedynamic` framework in order to discover the method handles contained in each call site. When a method handle h that maps to a method m is discovered to be contained in the call site of instruction i, a new call-graph edge $i \xrightarrow{h} m$ is created and the rules of the previous section analyze the method handle invocation. The rules for handling `invokedynamic` invocations are shown in Figure 6. Evaluation-wise, these rules *precede* the earlier rules that give semantics to method handles: The purpose of the rules in Figure 6 is to express what an `invokedynamic` does in terms of method handles, so that the earlier reasoning can take over.

We extend the earlier domains and predicates with:

- $I^d \subset I$ are `invokedynamic` instructions. Predicate $i \dashrightarrow_b m$ holds when an

`invokedynamic` instruction i calls bootstrap method m.[8] We also assume the following `invokedynamic` projections:

- $Boot : I^d \to M$ returns the bootstrap method.
- $B^p : I^d \to n \to V$ returns the bootstrap parameter at position n.
- $Dyn : I^d \to (S \times M^T)$ returns the dynamic method name / method type pair.

The rules are explained below:

Rules $BARGS$, $BARGS_0$, and $BARGS_V$. The first rule passes arguments to the boot method, shifted by three positions, since the first three arguments are filled in by the JVM (and handled by rule $BARGS_0$). Boot methods such as the alt metafactory may also take varargs that require special handling by the JVM, thus we also have rule $BARGS_V$. Note the introduction of an artificial (mock) array object to maintain the vararg values.

Rule $RETB$. This is the standard rule that returns value from a method call. It is adapted here for completeness, for the case of bootstrap method invocations.

Rule $CSITE$. This rule stores information about a call site object computed at an `invokedynamic` instruction.

Rules $CSITE1$ and $CSITE2$. These two rules relate a call site object with its method handle and the method it points to.

Rule $MHCGE_{DYN}$. This rule relates the `invokedynamic` call site and its method handle to create call-graph edges with method handle semantics. From this point on, the rules in the previous section take over and complete the method handle invocation.

Reflection Support. The rules presented in the subsection do not require reflection. For example, a program which contains an `invokedynamic` instruction that passes a method handle constant (read from the class constant pool) to its bootstrap method, can be analyzed without reflection support. In practice, however, bootstrap methods often employ reflective reasoning to compute the method handle that will be returned in the call site return value, and thus reflection support should be provided.

4.4 Model: Method References and Lambdas

Both method reference expression and lambdas are implemented by the same machinery, a "lambda metafactory" [50]. At a very high level, the metafactory takes two arguments, (1) a method handle pointing to a method m and (2) a SAM type t, and returns a functional object implementing t whose (single) method calls m.

The functional object may be an instance of a new dynamically-generated class, thus a naive points-to analysis cannot penetrate the object to analyze calls on it. Our analysis understands the semantics of the functional objects created by the dynamic linking and method resolution of the metafactory, and creates a mock value in place of the functional object. That value can be propagated in the program as usual by the underlying points-to analysis. Appropriate metadata on the value help the analysis compute intended semantics such as the invocation target or the captured values of the environment.

[8] We assume that all `invokedynamic` instructions call their bootstrap methods when their containing type is loaded.

$$\frac{i \dashrightarrow_b m \quad Dyn(i) = \langle s, m_t \rangle}{F_0^m \mapsto \mathcal{L}_t \quad F_1^m \mapsto s \quad F_2^m \mapsto m_t} \; BARGS_0 \qquad \frac{i \dashrightarrow_b m \quad B^p(i,n) \mapsto val}{F_{n+3}^m \mapsto val} \; BARGS$$

$$\frac{i \dashrightarrow_b m \quad val' = mock_c(\texttt{java.lang.Object[]}, i) \quad B^p(i,n) \mapsto val \quad n > 2}{F_3^m \mapsto a \quad val'[*] \mapsto val} \; BARGS_V$$

$$\frac{i \dashrightarrow_b m \quad RetVar(v,m) \quad v \mapsto val \quad Ret(i) = v'}{v' \mapsto val} \; RETB$$

$$\frac{Dyn(i) = \langle *, m_t \rangle \quad m_t = \{t, *\} \quad Boot(i) = m \quad RetVar(v,m) \quad v \mapsto c}{CSite(c,i,t)} \; CSITE$$

$$\frac{CSite(c,*,t) \quad c.\texttt{target} \mapsto h \quad h = \langle m, \{t,*\} \rangle}{CSite_C(c,h,m)} \; CSITE1$$

$$\frac{CSite(c,*,t) \quad c.\texttt{target} \mapsto h \quad h = \langle m, * \rangle \quad Constr(m) \quad m \in t}{CSite_C(c,h,m)} \; CSITE2$$

$$\frac{CSite(c,i,t) \quad CSite_C(c,h,m) \quad h = \langle *, \{t,*\} \rangle}{i \xrightarrow{h} m} \; MHCGE_{DYN}$$

Figure 6 Rules for generic handling of `invokedynamic`.

The Three Phases of `invokedynamic` for Lambdas. When used for lambdas, functional object creation by the lambda metafactory works in three phases [50]:

1. **Linkage.** The bootstrap method is called and a call site object is returned, at the location of the `invokedynamic` instruction. The bootstrap method being the "metafactory", the call site is then a "lambda factory", which must be invoked to produce a functional object.
2. **Capture.** The method handle in the call site object is invoked, possibly with some arguments. This permits different behavior for different contexts by capturing values of the enclosing environment. The result is the functional object.
3. **Invocation.** The functional object can then be passed around in the code and the method of its functional interface can be eventually called.

The rules that enable analysis of method references are shown in Figure 7. The basic idea is to create mock values in the analysis for functional objects and simulate all three phases so that calls are correctly resolved. We assume the following domains, (meta)variables, and predicates:

- The L^λ constant stands for the lambda metafactory [50] of the OpenJDK.
- $\lambda \in Val$ ranges over functional objects.
- $\#i$ returns the arity of instruction i (the number of actual parameters passed to the functional object).

The rules are explained below:

Rule *METAFACTORY*. This rule marks an `invokedynamic` invocation as a lambda metafactory invocation.

Rule *LAMBDA*. This rule creates the mock functional object λ that will propagate in the program and behave (in the analysis) as if it was an object created by the metafactory. The object keeps related metadata in relation $Lambda(\lambda, m, s, i)$: its implementing method

m (found in a constant method handle argument of the metafactory), the name of the functional interface method it implements, and the `invokedynamic` instruction i that created the functional object.

Rule *CAPTURE*. This rule records possibly captured values from the enclosing environment. (All arguments are eagerly recorded as possible captured values and the appropriate capture arguments are recognized in later rules *CAPTARGS* and *LAMBDATHIS*.)

Rule CGE_L. This rule creates call-graph edges to the actual implementing method of the functional object. Following these edges bypasses the dynamically-generated classes and lets the static analysis discover the code of method references and lambdas.

Rule RET_L. This is the standard rule for return values from methods.

Rule *INSTIMPL*. This rule records that a functional object is implemented by a non-static method. This means that further rules should discover the receiver and pass it to the method.

Rules $SHIFT_1$, $SHIFT_2$, and $SHIFT_3$. These rules populate relation $Shift(\lambda, m, n, k)$, which records if the arguments passed to the functional object must be shifted to make room for a receiver. This is because instance methods may implicitly consume one of the actual arguments of the `invokedynamic` or of the functional interface invocation, to use as the receiver. Static methods take all `invokedynamic`-actual arguments before the ones passed to the functional object during method invocation.

Rule *LARGS*. This passes arguments to the implementing method, from the method invocation on the functional object. The shifting of parameters addresses a number of patterns that the metafactory follows to capture and pass values from the environment.

Rule *CAPTARGS*. This rule passes captured arguments to the implementing method.

Rule *LAMBDATHIS*. This rule handles the pattern of captured receiver parameters.

Rule *MREFTHIS*. This rule handles the pattern where a method reference to an instance method has not captured a receiver, but will receive it during invocation as an extra argument.

Rule *CCALL*. This handles the special case where a method reference points to a constructor (is thus a "constructor reference"). Since constructor methods are `void` and assume an already constructed (but not initialized) object, this rule creates such an object and binds it both to the 'this' variable of the constructor and the return variable of the invocation.

Additional Features. The JDK also has a second metafactory, the "alt metafactory": a generalization of the lambda metafactory that provides additional features, such as bridging, support for multiple interfaces, and serializability. We do not model such extra properties of its lambdas here, but these features are type-based so they are amenabe to handling in a similar way to the rules we already present.

Reflection Support. The method handles passed to the metafactory are statically known: either the programmer provided them as method references or the compiler generated them for lambdas. Thus our rules for handling lambdas and method references do not need reflection support; the only method handles used come from the constant pool. This means that our approach can integrate with the baseline configuration of a points-to analysis, in order to analyze programs without overhead due to reflection support.

Context sensitivity. The analysis, as presented, has a context-insensitive formulation, to avoid unnecessary complication of the rules. Careful (but conceptually standard) addition of

context elements to predicates (as shown, e.g., in reference [66]) produces a context-sensitive version. Our implementation is fully context sensitive.

5 Evaluation

We evaluate our analysis on two test suites: a microbenchmark suite of our own (Section 5.1) and the test suite of *Sui et al.* [68] (Section 5.2).

Our analysis is implemented in the declarative static analysis framework Doop [6]. All analyses are run on a 64-bit machine with an Intel Xeon CPU E5-2667 v2 3.30GHz with 256 GB of RAM. We use the Soufflé compiler (v.1.4.0), which compiles Datalog specifications into binaries via C++ and run the resulting binaries in parallel mode using four jobs. Doop uses the Java 8 platform as implemented in Oracle JDK v1.8.0_121. All running times and precision numbers are for Doop's default context-insensitive analysis. (Context sensitivity adds no precision to the high-level metrics shown.) For benchmarks of generalized `invokedynamic` features (i.e., not lambdas and method references), we enable reflection support in Doop.

5.1 Microbenchmark Suite

To evaluate our technique, we have built our own suite of microbenchmarks. These benchmarks capture a large number of idioms found in realistic uses of method references (Section 5.1.1), lambdas (Section 5.1.2), and method handles combined with `invokedynamic` (Section 5.1.3), including most of the patterns shown in the examples of Section 2. (Other patterns are captured in the Sui et al. suite, discussed later.) The suite is freely available.

Analysis times for the three component benchmarks are shown in Figure 8. As can be seen, enabling reflection analysis, for fully general handling of `invokedynamic`, incurs higher cost.

Our static analysis fully models all behavior in the microbenchmark suite. Although the suite was developed in tandem with the analysis, it still provides partial validation of analysis completeness, given the effort to encode many variations of operations, as detailed next.

5.1.1 Microbenchmark: Method References

This benchmark includes Oracle's tutorial code MethodReferencesTest [51]. We capture the behavior of all four kinds of methods references (found in the tutorial table): to static methods, to instance methods of a particular object, to instance methods of an arbitrary object of a particular type, and to constructors.

The microbenchmark also contains code that showcases the following features:

1. Construction of functional objects directly from method references.
2. Use of functional objects together with Java 8 stream API methods.
3. Auto-boxing conversions.

5.1.2 Microbenchmark: Lambdas

This benchmark shows the handling of the following features:

1. Creating lambdas with arrow notation. This includes nested lambdas.
2. Creating lambdas that can access values of the outside environment (forming closures).

LINKAGE

$$\frac{i \dashrightarrow_b m \qquad m \in L^{\lambda} \qquad Dyn(i) = \langle s, m_t \rangle \qquad m_t = \{t^i, *\}}{MetafactoryInvo(i, s, t^i)}\ METAFACTORY$$

$$\frac{MetafactoryInvo(i, s, t^i) \qquad B^p(i, 1) \mapsto \langle m, * \rangle \qquad Ret(i) = v \qquad \lambda = mock_c(t^i, i)}{v \mapsto \lambda \qquad Lambda(\lambda, m, s, i)}\ LAMBDA$$

CAPTURE

$$\frac{MetafactoryInvo(i, *, *) \qquad A^i_n \mapsto val}{Capture(i, n, val)}\ CAPTURE$$

INVOCATION

$$\frac{Lambda(\lambda, m, s, *) \qquad v \mapsto \lambda \qquad i = v.\texttt{<}s\texttt{>}(\ldots)}{i \xrightarrow{\lambda} m}\ CGE_L$$

$$\frac{i \xrightarrow{\lambda} m \qquad R^m_n = v \qquad v \mapsto val \qquad Ret(i) = v'}{v' \mapsto val}\ RET_L$$

$$\frac{* \xrightarrow{\lambda} m \qquad \neg Static(m) \qquad Lambda(\lambda, *, *, i)}{InstanceImpl(i, m, \lambda)}\ INSTIMPL$$

$$\frac{Lambda(\lambda, m, *, *) \qquad Static(m)}{Shift(\lambda, m, 0, 0)}\ SHIFT_1$$

$$\frac{InstanceImpl(i, m, \lambda) \qquad \#i = 0}{Shift(\lambda, m, 0, 1)}\ SHIFT_2 \qquad \frac{InstanceImpl(i, m, \lambda) \qquad \#i > 0}{Shift(\lambda, m, 1, 0)}\ SHIFT_3$$

$$\frac{\begin{array}{l} i \xrightarrow{\lambda} m \qquad Shift(\lambda, m, k, n) \qquad Lambda(\lambda, m, *, i) \\ A^i_{n'} = v' \qquad F^m_{n''} = v \qquad n'' = \#i - (k + n) + n' \qquad v' \mapsto val \end{array}}{v \mapsto val}\ LARGS$$

$$\frac{* \xrightarrow{\lambda} m \qquad Shift(\lambda, m, k, *) \qquad Lambda(\lambda, m, *, i) \qquad Capture(i, n, val) \qquad k + n \leq \#i}{F^m_{n-k} \mapsto val}\ CAPTARGS$$

$$\frac{Shift(\lambda, m, 1, 0) \qquad InstanceImpl(i, m, \lambda) \qquad Capture(i, 0, val)}{m/\texttt{this} \mapsto val}\ LAMBDATHIS$$

$$\frac{i \xrightarrow{\lambda} m \qquad Shift(\lambda, m, 0, 1) \qquad A^i_0 \mapsto val}{m/\texttt{this} \mapsto val}\ MREFTHIS$$

$$\frac{i \xrightarrow{\lambda} m \qquad Constr(m) \qquad m \in t \qquad Ret(i) = v \qquad val = mock_c(t, i)}{v \mapsto val \qquad m/\texttt{this} \mapsto val}\ CCALL$$

Figure 7 Rules for handling method references and lambdas.

Benchmark	Time (sec)
Method References	27
Lambdas	23
Method Handles and `invokedynamic`	378

Figure 8 Microbenchmark times.

5.1.3 Microbenchmark: Method Handles and `invokedynamic`

Java currently does not support the direct representation of `invokedynamic` in source code, although such a feature is considered for inclusion in future versions of the language [21]. For this reason, this benchmark uses the ASM bytecode manipulation library[9] to dynamically generate and load a class with `invokedynamic` invocations.

The benchmark captures the following patterns:

1. Lookup of a `MethodHandles.Lookup` object via `MethodHandles.lookup()`.
2. Construction of method type values via `MethodType.methodType()` methods.
3. Look-up of virtual and static methods via `MethodHandles.Lookup.findVirtual()` and `MethodHandles.Lookup.findStatic()`.
4. Calling method handles with `MethodHandle.invokeExact()`.
5. Passing a receiver for non-static methods (thus handling places where the signature of the target method differs from the signature of the `MethodHandle.invokeExact()` signature found in the bytecode).
6. Bootstrapping calls to another class in a manner similar to the motivating example in Section 2.1.

5.2 Sui et al. Test Suite

We also evaluate our technique using the dynamic features test suite of *Sui et al.* [68]. This is a test suite that examines the soundness of call-graph construction and is written to specifically test the static analysis of features such as lambdas and `invokedynamic`, by authors with extensive experience in systematic Java benchmarking efforts (e.g., XCorpus [9]).

The benchmark suite contains three benchmarks for lambdas, plus a benchmark for `invokedynamic` in general (Dynamo). Dynamo is a realistic software artifact [28] that has been configured in the benchmark suite to specifically evaluate the analysis of `invokedynamic`. In particular, Dynamo exercises all features of dynamic invocation sites (static vs. non-static, constructors, signature adaptation, interaction with plain Java reflection). It injects `invokedynamic` calls in unsuspecting code to address cross-component linking errors. Thus, if these `invokedynamic` sites are not analyzed, then the static analysis cannot find calls from code to a library.

For every benchmark, the following ground truth is provided: one or more methods are expected to be found reachable, while one or more different methods are expected to be found unreachable. The results of applying our analysis to these benchmarks are shown in Figure 9.

Notably:

- All lambda benchmarks are analyzed precisely: the expected methods are found reachable or unreachable.

[9] `https://asm.ow2.io/`

Benchmark	Reachable		Unreachable		Time (sec)
	expected	analysis	expected	analysis	
LambdaConsumer	1	✓	1	✓	21
LambdaFunction	1	✓	2	✓	21
LambdaSupplier	1	✓	1	✓	22
Dynamo	1	✓	1	–	242

Figure 9 Dynamic benchmark results.

- For Dynamo, our analysis over-approximates reachability. Dynamo uses `invokedynamic` as a layer between components to ensure binary compatibility with evolving code. As seen in Figure 9, our analysis over-approximates reachability: it discovers the expected method as reachable but also discovers the expected unreachable method. This problem is not fundamental to the technique that we present, but is caused by the lack of flow sensitivity in the underlying points-to analysis, provided by the Doop framework. Dynamo code creates method handles by gathering reflectively all members of classes and then selectively filtering out the ones that do not match; Doop's flow insensitivity causes it to ignore this filtering. Coupled with flow sensitivity, our technique should be able to ignore the expected unreachable method.
- The efficiency of a lambda-specialized analysis vs. a general-purpose `invokedynamic` analysis that requires reflection support is again demonstrated in the running times.

6 Related work

Static Analysis of Java Lambdas and Dynamic Calls. Some recent work has attempted to treat lambdas and their static analysis, mostly in isolation, as another high-level feature for practical tools. Cifuentes *et al.* [7] perform a pattern-based vulnerability analysis (i.e., not a full low-level analysis of value flow) and recognize code patterns containing lambdas. There has also been work on dynamic analyses that understand Java-style lambdas [11].

Reflection and programmable dynamic calls are subtle features that should be formalized in order to be addressed. However, the bibliography is lacking: we only know of the work of Landman *et al.* [30], who give a syntax of the DSL behind the standard Java Reflection API. They do not treat its semantics, as they did not need to (their work was on mining big codebases for the existence of specific patterns).

To the best of our knowledge, no formal semantic model of `invokedynamic` and its API exists. Other Java APIs that cannot be easily analyzed statically have also been candidates for static semantic modeling. Smaragdakis *et al.* model the reflection API [67] and Fourtounis *et al.* model dynamic proxies [13]. Our approach differs in two aspects: (a) we do not necessarily incur performance overheads (our handling of functional objects does not require expensive reflection support) and (b) we model the lower-level `java.lang.invoke` API, which requires handling of JVM features such as signature polymorphism, caller sensitivity, and reasoning about code running at class-loading time.

The IBM WALA static analysis framework [10] has limited support for `invokedynamic`, specifically for call-graph edges over lambdas. The WALA documentation explicitly states no general-purpose `invokedynamic` support is provided.[10]

[10] `https://groups.google.com/forum/#!topic/wala-sourceforge-net/omsGtp_ow7I`, `https:`

Transforming Away `invokedynamic`. Lambdas are not easy to work with; Soot, a popular Java manipulation and analysis framework, even considers statically transforming them away [4], since `invokedynamic` has been too difficult to analyze: "Soot does not fully support dynamic invokes ... could not find an easy workaround and instead decided that it would be best to change Schaapi such that dynamic invokes (and thus lambdas) are ignored completely."[11]

Along the same lines, but more completely, the OPAL bytecode rectifier[12] removes instances of `invokedynamic` as used in Java lambdas. This is a general alternative static treatment of lambdas, but not of other instances of `invokedynamic`. Similar removal of stylized uses of `invokedynamic`, without handling the general case, are performed by RetroLambda[13] and Google's D8.[14] These tools cannot, e.g., make the Dynamo benchmark analyzable by analyses that do not understand `invokedynamic`.

Other Platforms. Apart from the popular OpenJDK and its VM, used on servers or desktops, the other mainstream Java platform is Android. The implementation of `invokedynamic` on Android posed some complications because dynamic code generation is restricted on Android due to resource constraints [57, 64]. `invokedynamic` was prototyped for Android [64] and, eventually, became officially supported when the latest "Android N" switched to Java 8. Our work is, thus, applicable to Android as well. Android is a platform that commands special attention due to its popularity. `invokedynamic` enables new optimizations and analyses [78, 79, 80]. However, the instruction is also a security threat, since it is so powerful that it can, for example, hide method calls and make malware undetectable (as demonstrated by the DexProtector tool [32] or the survey of Gorenc and Spelman [22]) and provides less security by-design compared to classic reflection [65].

The .NET platform also has functionality similar to method handles and anonymous classes, called "dynamic methods" [37]. We, thus, expect that our approach can be ported to other runtimes and to their implementation of dynamic features.

7 Conclusion

We presented a static analysis modeling of programmable dynamic linking in Java, i.e., the `invokedynamic` instruction and accompanying framework. The approach addresses the most fundamental level of the language feature, fully modeling method handles, while at the same time it maintains high efficiency and completeness for common uses of `invokedynamic` in Java lambdas. This is the first thorough handling of the `invokedynamic` feature, which had so far resisted static analysis.

References

1 Eric Allen, David Chase, Joe Hallett, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu, Guy L Steele Jr, Sam Tobin-Hochstadt, Joao Dias, Carl Eastlund, et al. The Fortress language specification. *Sun Microsystems*, 139:140, 2005.

//github.com/wala/WALA/issues/132.

[11] https://github.com/cafejojo/schaapi/pull/295

[12] http://www.opal-project.de/DeveloperTools.html

[13] https://github.com/luontola/retrolambda

[14] https://jakewharton.com/androids-java-8-support/

2 Malte Appeltauer, Michael Haupt, and Robert Hirschfeld. Layered method dispatch with INVOKEDYNAMIC: An implementation study. In *Proceedings of the 2nd International Workshop on Context-Oriented Programming*, COP '10, pages 4:1–4:6, New York, NY, USA, 2010. ACM. URL: http://doi.acm.org/10.1145/1930021.1930025, doi:10.1145/1930021.1930025.

3 Shashank Bharadwaj. Optimizing Jython using invokedynamic and gradual typing. Master's thesis, University of Colorado at Boulder, 2012.

4 Eric Bodden. Develop transformer that gets rid of indy calls for lambda capture #226. https://github.com/Sable/soot/issues/226, 2014. Open issue.

5 Norris Boyd et al. Rhino: Javascript for Java. *Mozilla Foundation*, 2007.

6 Martin Bravenboer and Yannis Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In *Proc. of the 24th Annual ACM SIGPLAN Conf. on Object Oriented Programming, Systems, Languages, and Applications*, OOPSLA '09, New York, NY, USA, 2009. ACM.

7 Cristina Cifuentes, Andrew Gross, and Nathan Keynes. Understanding caller-sensitive method vulnerabilities: A class of access control vulnerabilities in the Java platform. In *Proceedings of the 4th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*, SOAP 2015, pages 7–12, New York, NY, USA, 2015. ACM. URL: http://doi.acm.org/10.1145/2771284.2771286, doi:10.1145/2771284.2771286.

8 L. Peter Deutsch and Allan M. Schiffman. Efficient implementation of the Smalltalk-80 system. In *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '84, pages 297–302, New York, NY, USA, 1984. ACM. URL: http://doi.acm.org/10.1145/800017.800542, doi:10.1145/800017.800542.

9 Jens Dietrich, Henrik Schole, Li Sui, and Ewan D. Tempero. XCorpus - an executable corpus of Java programs. *Journal of Object Technology*, 16(4):1:1–24, 2017. URL: https://doi.org/10.5381/jot.2017.16.4.a1, doi:10.5381/jot.2017.16.4.a1.

10 Julian Dolby, Stephen J. Fink, and Manu Sridharan. T.J. Watson libraries for analysis (WALA). http://wala.sourceforge.net.

11 Sebastian Erdweg, Vlad Vergu, Mira Mezini, and Eelco Visser. Finding bugs in program generators by dynamic analysis of syntactic language constraints. In *Proceedings of the Companion Publication of the 13th International Conference on Modularity*, MODULARITY '14, pages 17–20, New York, NY, USA, 2014. ACM. URL: http://doi.acm.org/10.1145/2584469.2584474, doi:10.1145/2584469.2584474.

12 Rémi Forax. JSR 292 / PHP.reboot. https://www.lrde.epita.fr/dload/seminar/2010-12-08/forax.pdf, 2010.

13 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, pages 209–220, New York, NY, USA, 2018. ACM. URL: http://doi.acm.org/10.1145/3213846.3213864, doi:10.1145/3213846.3213864.

14 Brian Goetz. One VM, many languages. https://gotocon.com/dl/jaoo-aarhus-2010/slides/BrianGoetz_OneVMManyLanguages.pdf, 2010.

15 Brian Goetz. From lambdas to bytecode. http://wiki.jvmlangsummit.com/images/1/1e/2011_Goetz_Lambda.pdf, 2011.

16 Brian Goetz. Implementing lambda expressions in Java. http://wiki.jvmlangsummit.com/images/7/7b/Goetz-jvmls-lambda.pdf, 2012.

17 Brian Goetz. Lambda: A peek under the hood. https://www.slideshare.net/jaxlondon2012/lambda-a-peek-under-the-hood-brian-goetz, 2012.

18 Brian Goetz. Translation of lambda expressions. http://cr.openjdk.java.net/~briangoetz/lambda/lambda-translation.html, April 2012.

19 Brian Goetz. State of the lambda. http://cr.openjdk.java.net/~briangoetz/lambda/lambda-state-final.html, September 2013.

20 Brian Goetz. Project Valhalla update, 2016.

21 Brian Goetz. JEP 303: Intrinsics for the LDC and INVOKEDYNAMIC Instructions, 2018.

22 Brian Gorenc and Jasiel Spelman. Java every-days – exploiting software running on 3 billion devices. https://media.blackhat.com/us-13/US-13-Gorenc-Java-Every-Days-Exploiting-Software-Running-on-3-Billion-Devices-WP.pdf.

23 James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. The Java Language Specification, Java SE 8 Edition (Java Series), 2014.

24 Stuart Halloway. *Programming Clojure*. Pragmatic Bookshelf, 1st edition, 2009.

25 Anders Hejlsberg, Scott Wiltamuth, and Peter Golde. *C# Language Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

26 Philipp Holzinger, Stefan Triller, Alexandre Bartel, and Eric Bodden. An in-depth study of more than ten years of Java exploitation. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 779–790, New York, NY, USA, 2016. ACM. URL: http://doi.acm.org/10.1145/2976749.2978361, doi:10.1145/2976749.2978361.

27 Vladimir Ivanov. Invokedynamic: Deep dive. http://cr.openjdk.java.net/~vlivanov/talks/2015-Indy_Deep_Dive.pdf.

28 Kamil Jezek and Jens Dietrich. Magic with Dynamo – flexible cross-component linking for Java with invokedynamic. In Shriram Krishnamurthi and Benjamin S. Lerner, editors, *30th European Conference on Object-Oriented Programming (ECOOP 2016)*, volume 56 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 12:1–12:25, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. URL: http://drops.dagstuhl.de/opus/volltexte/2016/6106, doi:10.4230/LIPIcs.ECOOP.2016.12.

29 Chanwit Kaewkasi. Towards performance measurements for the java virtual machine's invokedynamic. In *Virtual Machines and Intermediate Languages*, VMIL '10, pages 3:1–3:6, New York, NY, USA, 2010. ACM. URL: http://doi.acm.org/10.1145/1941054.1941057, doi:10.1145/1941054.1941057.

30 Davy Landman, Alexander Serebrenik, and Jurgen J. Vinju. Challenges for static analysis of Java reflection – literature review and empirical study. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*, 2017.

31 Jim Laskey. Adventures in JSR-292 or how to be a duck without really trying. http://wiki.jvmlangsummit.com/images/c/ce/Nashorn.pdf, 2011.

32 Licel. DexProtector. https://dexprotector.com/docs.

33 Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. *The Java Virtual Machine Specification, Java SE 8 Edition*. Addison-Wesley Professional, 1st edition, 2014.

34 Baptiste Maingret, Frédéric Le Mouël, Julien Ponge, Nicolas Stouls, Jian Cao, and Yannick Loiseau. Towards a decoupled context-oriented programming language for the Internet of Things. In *Proceedings of the 7th International Workshop on Context-Oriented Programming*, COP'15, pages 7:1–7:6, New York, NY, USA, 2015. ACM. URL: http://doi.acm.org/10.1145/2786545.2786552, doi:10.1145/2786545.2786552.

35 Davood Mazinanian, Ameya Ketkar, Nikolaos Tsantalis, and Danny Dig. Understanding the use of lambda expressions in Java. *Proc. ACM Program. Lang.*, 1(OOPSLA):85:1–85:31, October 2017. URL: http://doi.acm.org/10.1145/3133909, doi:10.1145/3133909.

36 Erik Meijer, Brian Beckman, and Gavin Bierman. LINQ: Reconciling object, relations and XML in the .NET framework. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD '06, pages 706–706, New York, NY, USA, 2006. ACM. URL: http://doi.acm.org/10.1145/1142473.1142552, doi:10.1145/1142473.1142552.

37 Microsoft. DynamicMethod Class. https://msdn.microsoft.com/en-us/library/system.reflection.emit.dynamicmethod(v=vs.110).aspx.

38 Behrooz Nobakht and Frank S. de Boer. *Programming with Actors in Java 8*, pages 37–53. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014. URL: http://dx.doi.org/10.1007/978-3-662-45231-8_4, doi:10.1007/978-3-662-45231-8_4.

39 S. Nopnipa and C. Kaewkasi. Aspect-aware bytecode combinators for a dynamic AOP system with invokedynamic. In *The 2013 10th International Joint Conference on Computer Science and Software Engineering (JCSSE)*, pages 246–251, May 2013. doi:10.1109/JCSSE.2013.6567353.

40 Charles Nutter. A first taste of invokedynamic. http://blog.headius.com/2008/09/first-taste-of-invokedynamic.html.

41 Charles Nutter. The power of the JVM. http://blog.headius.com/2008/05/power-of-jvm.html.

42 Charles Nutter. invokedynamic: You ain't seen nothing yet. https://www.slideshare.net/CharlesNutter/jax-2012-invoke-dynamic-keynote, 2012.

43 Charles Nutter. GOTO Night with Charles Nutter Slides. https://www.slideshare.net/AlexandraMasterson/goto-night-with-charles-nutter-slides, 2014.

44 Charles O. Nutter, Thomas Enebo, Nick Sieger, Ola Bini, and Ian Dees. *Using JRuby: Bringing Ruby to Java*. Pragmatic Bookshelf, 1st edition, 2011.

45 Martin Odersky and Tiark Rompf. Unifying functional and object-oriented programming with Scala. *Communications of the ACM*, 57(4):76–86, April 2014. URL: http://doi.acm.org/10.1145/2591013, doi:10.1145/2591013.

46 OpenJDK Compiler Team. Bound method handles - HotSpot - OpenJDK Wiki. https://wiki.openjdk.java.net/display/HotSpot/Bound+method+handles.

47 Oracle. java.lang.invoke (Java Platform SE 7). https://docs.oracle.com/javase/7/docs/api/java/lang/invoke/package-summary.html.

48 Oracle. JEP 160: Lambda-form representation for method handles. http://openjdk.java.net/jeps/160.

49 Oracle. JSR 335: Lambda Expressions for the Java™ Programming Language. https://jcp.org/en/jsr/detail?id=335.

50 Oracle. LambdaMetafactory (Java Platform SE 8). https://docs.oracle.com/javase/8/docs/api/java/lang/invoke/LambdaMetafactory.html.

51 Oracle. Method References (The Java™ Tutorials > Learning the Java Language > Classes and Objects). https://docs.oracle.com/javase/tutorial/java/javaOO/methodreferences.html, 2017.

52 Oracle. MethodHandle (Java Platform SE 8). https://docs.oracle.com/javase/8/docs/api/java/lang/invoke/MethodHandle.html, 2018.

53 Oracle. MethodHandles (Java Platform SE 8). https://docs.oracle.com/javase/8/docs/api/java/lang/invoke/MethodHandles.html, 2018.

54 Oracle. MethodType (Java Platform SE 8). https://docs.oracle.com/javase/8/docs/api/java/lang/invoke/MethodType.html, 2018.

55 F. Ortin, P. Conde, D. Fernandez-Lanvin, and R. Izquierdo. The runtime performance of invokedynamic: An evaluation with a Java library. *IEEE Software*, 31(4):82–90, July 2014. doi:10.1109/MS.2013.46.

56 Samuele Pedroni and Noel Rappin. *Jython Essentials: Rapid Scripting in Java*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1 edition, 2002.

57 Jerome Pilliet, Remi Forax, and Gilles Roussel. DualStack: Improvement of invokedynamic implementation on Android. In *Proceedings of the 13th International Workshop on Java Technologies for Real-time and Embedded Systems*, JTRES '15, pages 4:1–4:8, New York, NY, USA, 2015. ACM. URL: http://doi.acm.org/10.1145/2822304.2822310, doi:10.1145/2822304.2822310.

58 Julien Ponge, Frédéric Le Mouël, and Nicolas Stouls. Golo, a dynamic, light and efficient language for post-invokedynamic JVM. In *Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, PPPJ '13, pages 153–158, New York, NY, USA, 2013. ACM. URL: http://doi.acm.org/10.1145/2500828.2500844, doi:10.1145/2500828.2500844.

59 Julien Ponge and Frédéric Le Mouël. JooFlux: Hijacking Java 7 invokedynamic to support live code modifications. *CoRR*, abs/1210.1039, 2012. URL: http://arxiv.org/abs/1210.1039.

60 M. Reif, F. Kübler, M. Eichberg, and M. Mezini. Systematic evaluation of the unsoundness of call graph construction algorithms for Java. In *Proceedings of SOAP 2018*. ACM, 2018.

61 John R. Rose. Anonymous classes in the VM. `https://blogs.oracle.com/jrose/entry/anonymous_classes_in_the_vm`, January 2008.

62 John R. Rose. Bytecodes meet combinators: Invokedynamic on the JVM. In *Proceedings of the Third Workshop on Virtual Machines and Intermediate Languages*, VMIL '09, pages 2:1–2:11, New York, NY, USA, 2009. ACM. URL: `http://doi.acm.org/10.1145/1711506.1711508`, `doi:10.1145/1711506.1711508`.

63 John R. Rose. Method handles and beyond... some basis vectors. `http://wiki.jvmlangsummit.com/images/8/88/Rose-2011-FutureDirections.pdf`, 2011.

64 Gilles Roussel, Remi Forax, and Jerome Pilliet. Android 292: Implementing invokedynamic in Android. In *Proceedings of the 12th International Workshop on Java Technologies for Real-time and Embedded Systems*, JTRES '14, pages 76:76–76:86, New York, NY, USA, 2014. ACM. URL: `http://doi.acm.org/10.1145/2661020.2661032`, `doi:10.1145/2661020.2661032`.

65 Security Explorations. Security vulnerabilities in Java SE. `http://www.security-explorations.com/materials/se-2012-01-report.pdf`. Technical Report.

66 Yannis Smaragdakis and George Balatsouras. Pointer analysis. *Foundations and Trends in Programming Languages*, 2(1):1–69, 2015. URL: `http://dx.doi.org/10.1561/2500000014`, `doi:10.1561/2500000014`.

67 Yannis Smaragdakis, George Balatsouras, George Kastrinis, and Martin Bravenboer. More sound static handling of Java reflection. In *Proc. of the Asian Symp. on Programming Languages and Systems*, APLAS '15. Springer, 2015.

68 L. Sui, J. Dietrich, M. Emery, S. Rasheed, and A. Tahir. On the soundness of call graph construction in the presence of dynamic language features - a benchmark and tool evaluation. `https://sites.google.com/site/jensdietrich/publications/preprints/On%20the%20Soundness%20of%20Call%20Graph%20Construction%20in%20the%20Presence%20of%20Dynamic%20Language%20Features.pdf?attredirects=0&d=1`. Accepted for APLAS'18.

69 Josef Svenningsson and Emil Axelsson. Combining deep and shallow embedding of domain-specific languages. *Computer Languages, Systems and Structures*, 44(PB):143–165, December 2015. URL: `http://dx.doi.org/10.1016/j.cl.2015.07.003`, `doi:10.1016/j.cl.2015.07.003`.

70 Attila Szegedi. Dynalink - dynamic linker framework for JVM languages. `http://medianetwork.oracle.com/video/player/1113272541001`, July 2011.

71 The Apache Groovy Project. Invoke dynamic support. `http://groovy-lang.org/indy.html`.

72 Trifork. erjang. `https://github.com/trifork/erjang/wiki`.

73 TypeLead. The Eta programming language. `http://eta-lang.org/`.

74 Raoul-Gabriel Urma. Processing data with Java SE 8 Streams, part 1. *Java Magazine*, 2014.

75 Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a Java bytecode optimization framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '99, pages 13–. IBM Press, 1999. URL: `http://dl.acm.org/citation.cfm?id=781995.782008`.

76 Ingo Wechsung. The Frege programming language (draft). `http://www.frege-lang.org/doc/Language.pdf`, 2014.

77 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 *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2013, pages 187–204, New York, NY, USA, 2013. ACM. URL: `http://doi.acm.org/10.1145/2509578.2509581`, `doi:10.1145/2509578.2509581`.

78 Shijie Xu, David Bremner, and Daniel Heidinga. Mining method handle graphs for efficient dynamic JVM languages. In *Proceedings of the Principles and Practices of Programming*

on The Java Platform, PPPJ '15, pages 159–169, New York, NY, USA, 2015. ACM. URL: http://doi.acm.org/10.1145/2807426.2807440, doi:10.1145/2807426.2807440.

79 Shijie Xu, David Bremner, and Daniel Heidinga. MHDeS: Deduplicating method handle graphs for efficient dynamic JVM language implementations. In *Proceedings of the 11th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems*, ICOOOLPS '16, pages 4:1–4:10, New York, NY, USA, 2016. ACM. URL: http://doi.acm.org/10.1145/3012408.3012412, doi:10.1145/3012408.3012412.

80 Shijie Xu, David Bremner, and Daniel Heidinga. Fusing method handle graphs for efficient dynamic jvm language implementations. In *Proceedings of the 9th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages*, VMIL 2017, pages 18–27, New York, NY, USA, 2017. ACM. URL: http://doi.acm.org/10.1145/3141871.3141874, doi:10.1145/3141871.3141874.