Language-Independent Information Flow Tracking Engine for Program Comprehension Tools

Mohammad Reza Azadmanesh and Matthias Hauswirth
University of Lugano
Lugano, Switzerland
Email: {azadmanm, matthias.hauswirth}@usi.ch

Michael L. Van De Vanter
Oracle Labs
Email: michael.van.de.vanter@oracle.com

***Abstract*—Program comprehension tools are often developed for a specific programming language. Developing such a tool from scratch requires significant effort. In this paper, we report on our experience developing a language-independent framework that enables the creation of program comprehension tools, specifically tools gathering insight from deep dynamic analysis, with little effort. Our framework is language independent, because it is built on top of Truffle, an open-source platform, developed in Oracle Labs, for implementing dynamic languages in the form of AST interpreters. Our framework supports the creation of a diverse variety of program comprehension techniques, such as query, program slicing, and back-in-time debugging, because it is centered around a powerful information-flow tracking engine.**

Tools developed with our framework get access to the information-flow through a program execution. While it is possible to develop similarly powerful tools without our framework, for example by tracking information-flow through bytecode instrumentation, our approach leads to information that is closer to source code constructs, thus more comprehensible by the user.

To demonstrate the effectiveness of our framework, we applied it to two of Truffle-based languages namely Simple Language and TruffleRuby, and we distill our experience into guidelines for developers of other Truffle-based languages who want to develop program comprehension tools for their language.

I. Introduction

Industry likes to reuse mature services for their products. In addition to saving time and costs, reusing contributes to more reliable products. The Truffle framework, together with the Graal compiler, developed by Oracle Labs with a focus on programming technologies, derives from this point of view. Truffle and Graal are open source projects that aim at taking advantage of the stability and performance of a mature Java Virtual Machine (JVM) for providing competitive runtime implementations of dynamic languages. They represent a programming language implementation technology that significantly reduces the effort for implementing dynamic languages, while staying competitive in terms of runtime performance, often exceeding traditional implementations of the specific dynamic languages. Additionally, Truffle provides support for low-overhead language interoperation, as well as a general instrumentation framework that supports multi-language debugging and other external developer tools. With the addition of the JVM Compiler Interface (JVMCI), developed by the Graal Project, the Graal compiler is now an option in the Java 9 VM and is under consideration as an eventual replacement for the existing C2 compiler.

As new languages emerge, there is also the need for tools that support programmers in comprehending the programs written in those languages. In fact, the accessibility of tools for a language can affect the popularity of that language among the users. However, developing tools from scratch for different languages is not trivial. Developing program comprehension tools that go beyond the syntactic level, and that depend on the semantics of the numerous features of a language, requires tool developers to cover a large number of cases. To simplify tool development it would be beneficial to have a tooling framework that takes care of and provides an abstraction over the many language-specific details. There are basically two issues in realizing such a framework: 1) language independence, and 2) tool independence.

Language Independence. Language-level virtual machines were built for a specific programming language. Specifically, the JVM was built to run Java programs. However, it is possible to "abuse" such platforms as runtimes for different languages. The JVM actually strives to support other languages, even dynamically typed languages (JSR-292). Many language implementers now develop their runtime on top of the JVM and take advantage of its mature features, such as memory management and just-in-time compilation. Many languages have been implemented on top of the JVM, including Ruby, Python, Clojure, and Javascript. A side benefit of implementing a language on top of the JVM is that software engineering tools which target the JVM and its bytecode theoretically can immediately be used for that language. However, the fact that the JVM was originally designed for Java is visible when looking at the implementation and the runtime behavior of other JVM languages, especially for dynamically typed languages [1]. Compilers of dynamically typed languages targeting the JVM must work around the strict typing of Java bytecode and the idiosyncrasies of the JVM to achieve the desired execution model. These workarounds introduce a considerable distance between the executed bytecode constructs and the source language constructs a developer is familiar with. As a consequence, while a dynamic analysis tool targeting Java bytecode can work for any JVM language, the results of its analysis are often far from the language constructs of that language, and thus are hardly usable directly for program comprehension.

Tool Independence. There is a multitude of tools that help in program comprehension. A useful framework for developing program comprehension tools needs to provide abstractions that are challenging to implement and that are useful in implementing a multitude of tools. Program comprehension tools use static as well as dynamic analyses to collect information about a program. Both kinds of information can be equally useful, but in this paper we focus specifically on dynamic information that has to be collected at runtime.

Our goal is to address the above problems by providing access to the most detailed information about a program execution in a form that is closely tied to source code. Specifically, our framework tracks and provides access to the explicit information flow, capturing how values flow throughout the whole program, and thus reflects program behavior across program dependencies. The reason we chose information flow tracking as the core of our framework is that the information on the dependencies within a program is what many program comprehension tools use for their analyses. As shown by Horwitz and Reps [2], dependence graphs play an important role as the basis for software engineering tools which address program comprehension. There is a large body of research on this topic, including approaches for program slicing [3], [4], [5], feature location [6], [7], [8], and debugging [9], [10], amongst the others.

Our framework has two characteristics:

1) It enables the creation of information-flow based program comprehension tools with reasonable effort.
2) The captured information maps intuitively to parts of the source code.

In this paper we emphasize the second characteristic, because the closeness to source code constructs of the provided information is essential for program comprehension. We build our framework for the Truffle language implementation technology. We focus on two of Truffle languages: Simple Language (SL), a demonstration language for Truffle, and TruffleRuby (TR), an implementation of Ruby on top of Truffle.

This work is part of a collaborative project with Oracle Labs. We make the following contributions:

1) A discussion of the comprehensibility of the information flow provided by two fundamentally different approaches, namely Java bytecodes vs Truffle ASTs. We use JRuby as our case study.
2) An approach for tracking information flow within a program regardless of the language in which it is implemented.
3) Developing a language-independent tooling framework usable by different program comprehension tools which require information flow. We discuss how language developers can employ this framework for their language using the concepts introduced in the approach.
4) A set of guidelines for tooling teams to take into account in their design decisions.

Section II motivates our approach using an example in Ruby. Sections III and IV discuss some background information. Section V explains our approach for having a language-independent information flow tracking framework. Section VI briefly presents the implementation details of the engine. Section VII provides the results of the evaluation for a case study, together with the messages we provide for the language developers based on our experience. Section VIII discusses the related work. We conclude the paper in section IX.

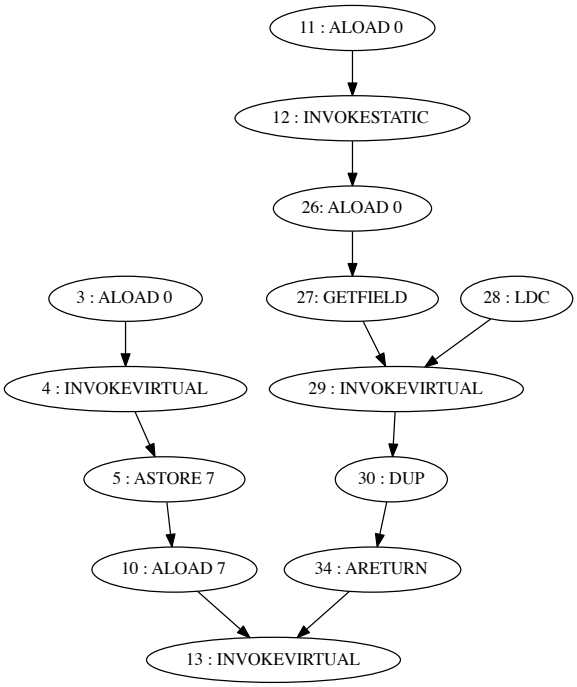
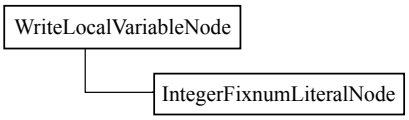
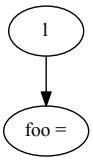
II. Motivation

Consider the code snippet `foo = 1`. This is a very simple, yet legitimate program in Ruby. One can explain/comprehend the program easily: it initializes the variable `foo` with the value `1`. As for the dependencies, we would expect the value of the variable `foo` to depend on the literal `1`. In Table I, under the column entitled `Dependency Graph`, we show a simplified variant of the dependency graphs for this very simple program extracted by 1) instrumenting Java-bytecodes (first row) and 2) instrumenting Truffle AST (second row). The codes from which the graphs were extracted are shown under the column entitled with `Code`. In the case of Truffle, we show the AST which TruffleRuby parser produces, and in the case of bytecode, we show the Java bytecodes which JRuby compiler generates for this program.

As can be seen, the AST is composed of only two nodes, where the nodes map one-to-one to the source code elements. However, the generated bytecode array is quite long and it does not map to the instructions of the original Ruby source code at all. As an example, the bytecode array has 4 invocations, whereas there is no invocation within the source code. Moreover, there is no bytecode corresponding to assignment.

In fact, the characteristics of Ruby makes the compiled code to be like that. In Ruby, every literal is represented as an object. So, every occurrence of a literal in the Ruby source code, say `1` in our example Ruby source code, leads to an invocation of a factory method (bytecode 12) in the generated class file by the JRuby compiler. The factory method (`fixnum0` in this case) provides a Ruby object corresponding to the specified literal. To provide a Ruby object, this method first tries to return an already cached object (bytecodes 22-24) and if it doesn't exist, it creates a new object of type `RubyFixnum` by calling a library method with the required arguments (bytecodes 25-30), caches it (bytecode 31) and returns the newly created object. The assignment happens through another library method call (bytecode 13). This happens because the runtime must first determine the appropriate dynamic scope for the target variable (`foo` in thise case) and store the value into that. Finding the current dynamic scope is done through another method invocation (bytecode 4) and store into that scope happens through another library call (bytecode 13). It should be noted that the graphs for the bytecode case is a simplified one, because 1) we don't trace the body of the library methods in our studies, 2) the graph in Table. I represents a slice of the full program trace. It only includes the nodes contributing to the value stored in the variable, and not those trying to find the value from cache.

TABLE I
THE COMPARISON OF JAVA-BYTECODE VS TRUFFLE AST APPROACH FOR THE RUBY PROGRAM `foo = 1`

	Code	Dependence Graph
Bytecode for program `foo = 1`	1 public static RUBY$script(Lorg/jruby/runtime/ThreadContext;Lorg/jruby/parser/StaticScope;Lorg/jruby/runtime/builtin/IRubyObject;[Lorg/jruby/runtime/builtin/IRubyObject;Lorg/jruby/runtime/Block;Lorg/jruby/RubyModule;Ljava/lang/String;)Lorg/jruby/runtime/builtin/IRubyObject; 2 L0 3 ALOAD 0 4 INVOKEVIRTUAL org/jruby/runtime/ThreadContext.getCurrentScope ()Lorg/jruby/runtime/DynamicScope; 5 ASTORE 7 6 NOP 7 NOP 8 L1 9 LINENUMBER 1 L1 10 ALOAD 7 11 ALOAD 0 12 INVOKESTATIC LocalVarStoreDemo.fixnum0 (Lorg/jruby/runtime/ThreadContext;)Lorg/jruby/RubyFixnum; 13 INVOKEVIRTUAL org/jruby/runtime/DynamicScope.setValueZeroDepthZeroVoid (Lorg/jruby/runtime/builtin/IRubyObject;)V 14 ALOAD 0 15 INVOKESTATIC LocalVarStoreDemo.fixnum0 (Lorg/jruby/runtime/ThreadContext;)Lorg/jruby/RubyFixnum; 16 ARETURN 17 L2 18 FRAME FULL [] [java/lang/Throwable] 19 ATHROW 20 21 private static synthetic fixnum0(Lorg/jruby/runtime/ThreadContext;)Lorg/jruby/RubyFixnum; 22 GETSTATIC LocalVarStoreDemo.fixnum0 : Lorg/jruby/RubyFixnum; 23 DUP 24 IFNONNULL L0 25 POP 26 ALOAD 0 27 GETFIELD org/jruby/runtime/ThreadContext.runtime : Lorg/jruby/Ruby; 28 LDC 1 29 INVOKEVIRTUAL org/jruby/Ruby.newFixnum (J)Lorg/jruby/RubyFixnum; 30 DUP 31 PUTSTATIC LocalVarStoreDemo.fixnum0 : Lorg/jruby/RubyFixnum; 32 L0 33 FRAME SAME1 org/jruby/RubyFixnum 34 ARETURN	
Truffle for program `foo = 1`		

However, the generated AST for the same program is composed of two nodes, namely IntegerFixnumLiteralNode and WriteLocalVariableNode. As a result, the dependency graph for a literal in case of instrumenting ASTs contains two nodes: One node representing the loading of literal and the other node representing the store into the local variable. So, a programmer in this case needs to go through only two nodes to understand where the value came from, where as in the case of bytecode instrumentation, she has to go through 13 nodes. Moreover, the content of the graph for bytecodes includes some information which do not map to the source code. We discuss how our approach can abstract all the complexities in the compiled code.

III. INFORMATION FLOW

The dependencies in a program cause information flow. The notion of Information Flow is mostly tied to the area of computer security where provisions are made to prevent from unauthorized information flow during a program execution [11]. Unauthorized information flow happens as a result of using a tainted value in a trusted computation. Thus, an information flow tracking system focuses on the origin of values in computations.

The same notion can be applied to other areas such as debugging, and program comprehension. In this case, a tainted value would be the one not being expected at a point of the execution. A programmer would need to figure out how the unexpected value came to be or where it originated from. For our discussions, we define information flow as follows:

Definition 3.1: An information flow from X to Y happens when Y observes a change to its value which is directly or indirectly dependent on the value of X over the course of a program execution.

An information flow in a program can happen either explicitly or implicitly. An explicit information flow is the one that happens through assignment statements. For example, in

the statement x = y, the value of program variable y flows into the program variable x. An explicit flow can happen indirectly through intermediate assignments, as the following code snippet shows:

```
1 x = 1;
2 y = x;
3 z = y;
```

In this example, the value of x (in this case 1) flows into z through the intermediate assignment statement y = x.

An explicit flow can also happen when there is partial usage of some value. For example, in the statement x = y + z, there is a flow from y to x, even though the values may not be the same.

An implicit information flow happens as a result of control flow instructions. For our discussions, we ignore implicit information flow.

A. Shadow Graph

We represent the information flow using a variant of Dynamic Depedence Graph (DDG) [12], [13]. A DDG is a directed graph where each node represents an occurrence of a program statement during the execution. An edge between two nodes represent either the data or control dependency between those two nodes. The information flow tracking engine records every step of execution in the form of a sub-graph which becomes part of the whole program information flow graph. Each sub-graph represents the executed AST node and the value defined by that node. The nodes provide some meta-data about what existed at runtime. Therefore, we call this graph as Shadow Graph. In this section, we demonstrate the structure of the shadow graph.

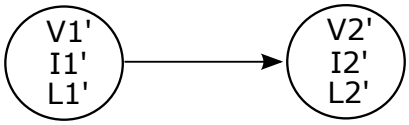
B. Nodes

A node in the shadow graph corresponds to a value defined at a particular point of execution. The node provides meta-data about the creation of the value, as a result, we call it as Shadow Node. A shadow node provides two pieces of information: 1) the instruction which defines the current value, 2) the memory location where the current value resides. In our implementation, we keep track of the instruction using a pointer to the AST node which defined the current value. The AST node provides information regarding the source section to which it belongs. As for the memory location, the meta information could differ among different types of memory locations. For example, for an instance variable, the meta information could be the identifier of the instance object and the name of the variable, whereas for a local variable, the meta information is the identifier of the frame in which the variable is defined and the name of the local variable.

C. Edges

The edges in the shadow graph represents the flow of information. Fig. 1 shows a sample graph for a program where: 1) the AST node I1 generates the value V1 in the memory location L1, 2) the AST node I2 generates the value V2 in the memory location L2, and 3)I2 uses the value residing in L1.

Fig. 1. The graph representing information flow



V1′, I1′, L1′, V2′, I2′, and L2′ represent the meta-data about V1, I1,L1, V2, I2 and L2, respectively.

IV. Truffle Framework

Truffle is a library for writing AST interpreters [14], part of Oracle Labs open source Graal platform [15] for implementing programming languages. Graal enables high performance implementations that share a large amount of language-independent code and services. Truffle-based languages currently include Ruby, Javascript, R, and Python, among others.

The *Truffle Instrumentation and Debugging Framework* provides a language-independent runtime API for low-overhead access to *execution events*. It is designed to simplify the construction of a wide variety of developer tools [16], and can support any number of independent tools as *clients*. The framework currently supports debugging and profiling services in the Graal runtime, as well as experimental tools inside and outside the company.

A Truffle event describes the execution of a single node in a Truffle AST, together with the static context (AST node and corresponding source code), dynamic environment (stack, etc.), and returned value. Clients of the framework *subscribe* to events at *instrumented nodes* by providing a *filter*. Events are delivered synchronously with extremely low overhead [17] through dynamic insertion of *wrapper* nodes that capture execution state at AST nodes where subscriptions are active.

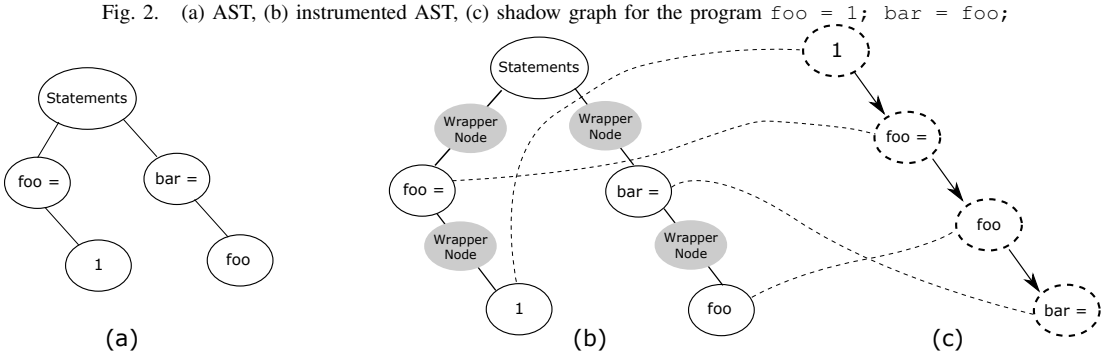
Instrumentation clients rely on Truffle language implementors to *mark up* AST nodes that correspond to significant program elements such as blocks, statements, and expressions. One kind of markup is a SourceSection, which identifies the specific text corresponding to the node, together with a description (including language) of the source that contains the text. Another kind of markup is a set of *tags*: symbolic names that identify the kind of program element represented by a node. For example, the debugging services halt while stepping only at nodes tagged as ”statements”. Subscription filters can be expressed using any combination of sources, languages, text locations, and tags.

As an example, consider the following code snippet:

```
1 foo = 1;
2 bar = foo;
```

An AST for this program could be like the one shown in Fig. 2-a. There are different types of AST nodes in this tree: two nodes for assignment, a node for loading a variable, and a node for a literal. To get the specific details about each node at runtime, the language developer can tag different AST node classes based on their functionality. The instrumentation framework adds the appropriate wrapper nodes, leading to the AST shown in Fig. 2-b. The wrapper nodes can intercept the

Fig. 2. (a) AST, (b) instrumented AST, (c) shadow graph for the program `foo = 1; bar = foo;`



access to the wrapped node by invoking event listeners before and after the node is visited. The shadow graph representing the information flow for this example is shown in Fig. 2-c. To build the shadow graph, for each use of a memory location, say `L`, by an AST node, say `I`, we need to add an edge from the shadow node corresponding to the last store into `L`, to the newly created shadow node for the value defined by `I`. For example, the access to variable foo in the second statement should be connected to the definition in the first statement.

V. Approach

As we said earlier, our goal is to have a language-independent source-level information flow tracking engine. The information flow happens once an instruction gets executed. In terms of AST interpreters of Truffle languages, each step of execution can be seen as executing an AST node. An AST node may use some runtime values located in some memory locations (such as a local variable), it performs some computation on the values, and it may define some new values in some memory locations. From the point of view of an information flow tracking engine, there is a flow of information from the origins to the target memory location, through the executed AST node.

The instrumentation framework can inform the engine about each executed AST node and the value generated by that node. In order to keep the information flow graph consistent, we need to connect the shadow node corresponding to the newly executed AST node to its origins within the current shadow graph. The challenge is how to look up in the current shadow graph for a shadow node representing the last definition of the memory location used by the newly executed AST node. Remember that each shadow node represents a value defined at runtime by an AST node for a specific memory location. So, we can uniquely find the shadow node, if we store the shadow nodes in a hierarchy similar to the runtime memory of the program. But, the problem is that the hierarchy of memory for different languages is not necessarily the same. In section V-A, we show how we abstract the hierarchy of memory locations for different languages, leading to a language-independent classification with a clear hierarchy within each class.

Given such a classification and the hierarchy within each class, to find the shadow node corresponding to a value, we need specific details about the runtime location of that value. We use this information as the key for mapping to the corresponding shadow node. We can expect the AST node to provide the details on the memory location to which it depends. However, we need to deal with different languages, each having many different AST node classes. So, Just as in the case of an Instruction Set Architecture (ISA) where different instructions may fall into the same instruction format, we need to identify different kinds of AST node classes and the memory locations on which they perform. Once we have such a model, we can implement it as an engine with limited cases to deal with. Section V-B shows how to use the classification of the memory locations for defining different kinds of AST node classes.

A. Memory Locations

An AST node can use a set of memory locations (use-set) and define a set of memory locations (def-set). A specific memory location at runtime resides in a memory region such as stack or heap, and the program usually can refer to the location using a name binding (for operand stack slots, there is no name binding normally). While there are different types of memory regions at the runtime of a program execution, from which the values are accessed, these regions are a small finite set which is usually common among different programming languages. Besides, for each type of AST node, we can determine the regions for its use and def set statically.

While different languages can have different memory regions for their runtime environment, but we find the following regions quite common among many popular programming languages:

1) Local space: This is the region where the information local to a method call is kept. For a stack-based computational model, it goes by Call Stack. It is private to the thread which runs the invocation. It normally consists of intermediate values, local variables table, argument values, and possibly a return value. For each invocation of the method, a new set of memory locations are reserved to keep the values. The local variables table keeps track of the variables which are defined within that lexical scope, though not declared to be static. A return value keeps track of the value returned by the current invocation of the method.

2) Heap space: This is the space where dynamically allocated data structures reside. For an object oriented language,

this includes the objects created at runtime. This region is normally shared among all threads of execution. The lifetime of an object residing in this region, for a managed runtime environment with automatic memory management, depends on its reachability.

3) Static space: The space where the variables whose addresses are determined statically, reside. The name of such a variable must be bound to a lexical context such as a class, a module, or a method definition. As a result, it differs from global variables whose names are not bound to any specific lexical context. The lifespan of a static variable is equal to the whole program run.

4) Global space: The global space resembles static space in the sense that addresses are determined statically, but as opposed to static variables, there is no lexical context to which the name of these variables belong and they are accessible from everywhere within the program. The lifespan of a global variable is equal to the whole program run.

The knowledge about the region to which a memory location belongs narrows down the search space within the shadow graph for the corresponding shadow nodes, though, it is not enough. To uniquely locate a shadow memory location, we need extra information about the location. Such extra information is not the same for different regions, but once we know about the region, we can expect the AST node to provide those information. That is why we need to classify the AST nodes based on the memory regions they read from and write to. The discussion in the next section is based on this observation.

B. AST node classes

As we said, we need to get some extra information from each executed AST node to uniquely locate the shadow memory locations corresponding to its origins and targets, within the shadow graph. The expected information differs from one memory region to the other, but it is the same for a single region. For example, for a read from a local variable, the name of the variable and the activation in which the variable is defined are required, whereas, for a read from a global variable, only the name matters. As a result, we abstract the behavior of AST nodes by just focusing on the regions of their use/def set, regardless of the language to which they belong. We can now tag different types of AST nodes, based on how they move data between these regions. A tag is a combination of the the name of the region(s) where the operands are coming from and the name of the regions the output are written to. This implies that , in theory, the number of different tags should be the size of the set corresponding to the Cartesian product of different regions for the use-set and def-set. However, in practice, many of the combinations are not realistic for an AST node, and only a small set of them are possible. For example, for an AST node which is aimed at reading a local variable, it is not common to write the read value in the heap space, but normally it is returned as an intermediate value.

Table II shows the tags and their meaning. There are couple of points about the content of the table. First, the Truffle languages use an AST interpreter model. So, the intermediate values are passed around as the return value of the method which evaluates the AST node. In our model, we use the notion of operand stack for representing intermediate values. We assume each method invocation has an operand stack, and each intermediate value computed at runtime is stored in a slot of the operand stack. Second, the column `NoL` specifies the case where no memory location is involved. Third, for the rows corresponding to read and write into an instance variable, the requirements are followed by a question mark. The reason for that is that this requirement is not consistent among different language implementations. We discuss the details in section VII-E.

VI. Engine Implementation

The implementation of the information flow tracking engine is composed of a set of event listeners. Each listener corresponds to one of the tags which we already identified in the previous section. The instrumentation framework, upon the execution of an AST node, delivers an event to its corresponding event listener. The event listener is responsible to update the shadow graph with the information about the newly executed AST node. This happens through creating a shadow node for the newly executed AST node and attaching it to its origins within the current shadow graph.

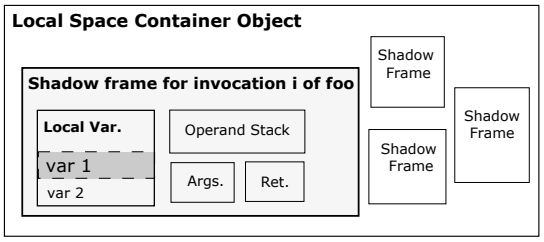
In order to locate the shadow nodes in the graph, we store each shadow node in a container object corresponding to the memory region of the value defined in that node. It implies that we have four container objects, representing the four different memory regions we identified in section V-A. While this narrows down the search space, it is not sufficient. To uniquely find a shadow node, we need to use the specific information required for each region, as shown in the last column of the table II. So, for each tag, we introduce a Java interface which contains methods to get the specific information to that tag from the AST node. It means that tagging an AST node class requires implementing the methods declared for the interface corresponding to that tag.

We use an example to explain the behavior of an event listener. Assume an AST node which reads from a local variable gets executed. The instrumentation framework invokes the event listener corresponding to the tag for the nodes using local variable (forth row of the table II). Such a listener targets the local-space container object. The local space container object is a set of Shadow Frame Objects, as shown in Fig. 3. A shadow frame object represents meta information about a runtime frame. The instrumentation framework provides the runtime frame object for all the event listeners. We map from the runtime frame object, say the frame for the invocation `i` of method `foo`, to a shadow frame object in the local space container object. The shadow frame object contains all the local-space-specific memory locations. Within the shadow frame object, we look up in the local variables table using the name of the variable. The AST node provides the name of the local variable, say `var1`, given the interface of its tag.

TABLE II
TAGS FOR DIFFERENT KINDS OF AST NODES. EACH ROW REPRESENTS ONE TAG. THE REQUIREMENTS COLUMN SPECIFY THE INFORMATION WHICH THE NODES LABELED WITH THAT TAG SHOULD PROVIDE. OS = OPERAND STACK, LV = LOCAL VARIABLE, A = ARGUMENT, RV = RETURN VALUE, IV = INSTANCE VARIABLE, CV = CLASS VARIABLE, GV = GLOBAL VARIABLE, NOL = NO MEMORY LOCATION

OS	LV	A	RV	IV	CV	GV	NoL	Comment	Requirements
D							U	AST nodes that read no memory location, and define a value onto the operand stack, e.g.: a literal node	
U/D								AST nodes that pop at least one value from top of the operand stack and define a value onto the operand stack, e.g.: an operator node	
D	U							AST nodes that read a local variable and push the value onto the operand stack.	Frame, Name of local variable
U	D							AST nodes that pop a value from the operand stack and write the value into a local variable	Frame, Name of local variable
D		U						AST nodes that read an argument value and push the value onto the local operand stack	Index of argument
U			D					AST nodes that pop a value from the operand stack and define the return value for the current method, e.g.: a return node	
U				D				AST nodes that pop values from the operand stack and write a value into an instance field within the heap space	Field name?
U/D				U				AST nodes that pop values from the operand stack to find an object within the heap, read the value of the instance field and push the value onto the operand stack	Field name?
D					U			AST nodes that read the value of a statically scoped variable and push the value onto the operand stack	Field name, static scope id
U					D			AST nodes that read a value form the operand stack and assign it to statically scoped variable	Field name, static scope id
D						U		AST nodes that read a global value and push the value on top of the operand stack	Variable name
U						D		AST nodes that pop a value from the operand stack and assign the value to a global variable	Variable name

Fig. 3. The structure of a local space container object



VII. EVALUATION

Initially, we plugged the engine to Simple Language. As the name suggests, it is a simple language for demonstration purposes of Truffle. We tagged all the AST node classes of the language using the tags we discussed in Table II. The next step was to try the engine for another language. This time we targeted TruffleRuby. TruffleRuby is an implementation of Ruby on Truffle. While we tagged the most essential features of the language, it is not covering all the language. The reason we did not tag all AST nodes is that it requires the knowledge about the behavior of each AST node class. Usually, it is the language developer who has this knowledge for all the nodes. The current implementation of TruffleRuby includes 244 AST node classes directly extending the top level class `RubyNode` in the class hierarchy. We identified 21 AST node classes whose behavior was obvious from their names and tagged them accordingly. The tagging of each AST node class boils down to 10 lines of code on average, so for the full language coverage, roughly 2500 lines of code are required, where most of the code is repeating. We also suspect that most of the AST nodes which we did not tag fall in the category of the nodes dealing only with the operand stack. In the following, we discuss a case study on the comparison of our approach versus the bytecode instrumentation approach.

A. Case Study

In this section, we discuss on the comprehensibility of the dependencies detected through our approach on Truffle for TruffleRuby vs an alternative approach based on bytecode instrumentation for JRuby. For the latter case, we use JRuby compiler to generate bytecodes. We go over different language features and extract the shadow graph for those features in both cases. Table III shows the results. We provide two types of metrics. For each language feature, the *Element Count* column specifies the number of AST nodes, and the number of bytecodes generated in the case of Truffle and Java-Bytecodes, respectively. The other two columns show the size of the sliced shadow graph in each case. The sliced shadow graph is computed with respect to the final result. For example, for a store, we find the slice for the already stored value. This provides more fair results, because not all the bytecodes are involved in a specific computaiton. Yet, we provide the element count because it is an indicator of the number of elements which need to be instrumented. The more elements to be instrumented, the larger the final code is. Due to the lack of space, we cannot go through all the results in detail. Instead, we discuss some of them in details.

TABLE III
COMPARISON OF OUR APPROACH ON TRUFFLE VS JAVA BYTECODE INSTRUMENTATION ON JVM

	Element Count		Shadow Graph Size	
Language Feature	Bytecode	AST	Bytecode	AST
Load Literal	11	1	8	1
Load local Var.	6	1	6	1
Store Local Var.	6	1	6	1
Method Invocation	31	2	NA	2
Binary Operator	14	3	NA	3
Load Instance Var.	21	3	13	3
Store Instance Var.	21	4	13	4
Load Class Var.	6	1	6	1
Store Class Var.	8	1	8	1
Load Global Var.	2	1	2	1
Store Global Var.	4	1	4	1

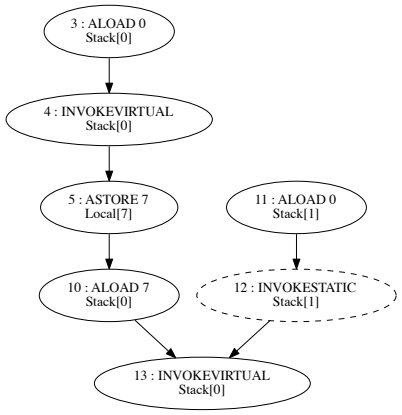
B. Storing a local variable

Consider the simple assignment statement `var = 1`. The bytecode array generated by the JRuby compiler for this assignment statement is as follows:

```
1  public static RUBY$script(Lorg/jruby/runtime/ThreadContext
       ;Lorg/jruby/parser/StaticScope;Lorg/jruby/runtime/
       builtin/IRubyObject;[Lorg/jruby/runtime/builtin/
       IRubyObject;Lorg/jruby/runtime/Block;Lorg/jruby/
       RubyModule;Ljava/lang/String;)Lorg/jruby/runtime/
       builtin/IRubyObject;
2    L0
3     ALOAD 0
4     INVOKEVIRTUAL org/jruby/runtime/ThreadContext.
       getCurrentScope ()Lorg/jruby/runtime/DynamicScope;
5     ASTORE 7
6     NOP
7     NOP
8    L1
9     LINENUMBER 1 L1
10    ALOAD 7
11    ALOAD 0
12    INVOKESTATIC locals.fixnum0 (Lorg/jruby/runtime/
       ThreadContext;)Lorg/jruby/RubyFixnum;
13    INVOKEVIRTUAL org/jruby/runtime/DynamicScope.
       setValueZeroDepthZeroVoid (Lorg/jruby/runtime/builtin/
       IRubyObject;)V
```

The code starts with a determination of the current dynamic scope. The scope determines the value of local variables that are accessible at any point of execution. As a result, for any access to a local variable, the runtime needs to first determine the current dynamic scope (bytecode index 4). In case of Truffle, the instrumentation framework provides the frame for each event listener and the frame contains the variables active in that scope. The invocation at bytecode index 12 is for loading the right-hand side value (literal). The actual store happens through another method invocation at bytecode index 13 using the currently found dynamic scope. The sliced shadow graph for this simplified scenario is shown in Fig. 4. Each node shows the executed bytecode and the memory location defined by that bytecode. Remember that the final store to the local variable is happening through the library call and we do not see it in the code. So, we provide a slice with respect to the bytecode index 13, assuming that it stores the value to the appropriate local variable. We do not trace the body of library methods, because it increases the trace size even more with details on the library execution. The graph has 7 nodes, but 6 of them are involved in storing and the dashed node is for loading the literal.

Fig. 4. The shadow graph for an assignment to a local variable in JRuby



In TruffleRuby, the AST for the same code snippet, i.e., `var = 1` has two nodes: a node for loading the right hand side, and another node for storing the right hand side value into a local variable. The resulting shadow graph for such an AST has also 2 nodes, one representing the load of the literal to a slot in the operand stack, and one node corresponding to using the value at the top of the operand stack and storing it into the local variable `var`. In fact, the store into the local is shown by a single shadow node. Not only the shadow graph size gets much smaller in this case, but also, the nodes in the shadow graph have a corresponding element in the concrete source code. This makes it more intuitive for the programmer to reason about the graph. The three invocations, in case of bytecode, plus all indirections make the results of a query language quite misleading.

C. Method Invocation

For a simple method invocation with no arguments, say `foo`, the compiler generates 31 bytecodes. Due to the lack of space, we do not show the code here. The generated bytecode array consists of four parts. 1) registering the already compiled method `foo` using the runtime helper function `defCompiledInstanceMethod`, 2) invoking a synthetic method (a method introduced by the compiler that does not have a corresponding construct in the source code) containing the code for finding the call site for method `foo`, 3)invoking a runtime helper function called `newVariableCachingCallSite` to find the call site for method `foo`, and 4) invoking the method `call` on the already resolved call site object which ultimately ends up in calling the method `foo`. As can be seen, for a simple method invocation within a Ruby source code, three other irrelevant invocations of the runtime library needs to be done before the actual invocation takes place. Even though we do not trace the body of library methods, but this is already a lot indirection which happens inside the bytecode. In the case of Truffle, the AST

for the same Ruby code snippet consists of just a single node, namely `RubyCallNode` and this node has the `SelfNode` as its child. The `SelfNode` loads the current object on which the method is to be invoked. In the case of byteocode, we cannot provide the shadow graph for an invocation, because the actual invocation happens dynamically from within a library method and given that we do not trace library methods, our shadow graph for the invocation becomes detached from its origins.

D. Operators

In Ruby, every data item is an object, so all operations on these data items happen through method invocations on objects. Given the need to a method invocation, the generated bytecode array has similarities to the bytecode array generated for a normal method invocation in the source. This implies that the resulting value of applying an operator on some operands can not be traced backward. However, the AST for an operator is composed of a specific node responsible for invoking library methods. Given that this node is specific, we can instrument this node and connect its operands to its return value.

E. Messages for the language developers

In this section, we provide some key messages for the language developers based on our experience.

1) A well-defined hierarchy of classes for AST nodes. The reason is that tagging can be done much easier if the AST nodes with the same behavioral semantics inherit from a common base class.

2) For some of AST node classes, the number of operands and the order of evaluating them are quite language-specific. For example, for a store into an instance variable, usually there are three operands, namely the receiver object, the right hand side value and the name of the field. These three can be evaluated in any order for different languages. The instrumentation cannot tell in which order the operands reside on the operand stack, once it encounters a store instance variable node. Besides, the number of operands may also differ from language to language. For example, in SL, there are three dynamic operands to be evaluated at runtime, whereas for TruffleRuby, there are two operands and the name is a compile-time constant. We solve this problem by extending the interface of the tag with methods that return an index (into operand stack) that identify where particular information is to be found. If such a method returns -1 then the framework will assume that this is the two-argument case and handle accordingly. The question mark in the last column of the Table II refers to this problem.

3) As we said, for each tag, there is a corresponding Java interface which determines the specific requirements to be provided by the AST nodes labeled with that tag. Currently, tagging an AST node and implementing the interface are independent from each other. In other terms, tagging as AST node class does not force the language developer to implement the corresponding interface. We learned from our experience that the instrumentation framework should be updated such that tagging happens through implementing the specific interface of the tag. In this way, the event handler can safely down cast the AST node object to the specific tag interface and get the required information out of it.

4) For any set of statements which can be grouped together lexically, such as the body of a method or statements in a module, it is always a good idea to wrap them under a common AST node. This can specify the boundary of entrance and exit to that module. The instrumentation framework can use this boundary for initialization and disposal purposes.

VIII. Related Work

Dynamic program analysis is crucial to program comprehension, because it has the potential to provide an accurate picture of the actual behavior of a program at runtime [18]. In this paper, we focused on dynamic analyses centered around program dependencies and information flow.

The Program Dependence Graph (PDG) was introduced by Ferrante et al. [19] as a program representation that provides a unified framework for applying program optimization techniques. Horwitz et. al. [2] discuss how the PDG as a language independent program representation, can provide the basis for powerful program comprehension tools.

Program slicing is a technique that can benefit from dependence graphs. The term program slicing was originally coined by Weiser [3] as the technique which prunes a program such that those parts that could not have contributed to the failure are ruled out. Weiser's approach was to solve a sequence of static dataflow-analysis problems. Ottenstein and Ottenstein [20] proposed another approach based on PDG traversals. Korel and Laski [21] extended Weiser's static approach for the dynamic case. Agrawal and Horgan [5] discuss several approaches for computing dynamic slicing using a Dynamic Dependence Graph (DDG). They also introduce a Reduced DDG as an economical variation. In this paper, we used a variation of DDG for representing a program's execution.

The notion of information-flow is used for program comprehension. Bergeretti and Carré [22] use information-flow relations to support program comprehension tasks in writing, debugging and updating a program. Their approach is based on syntactic information, whereas ours is purely dynamic. Lienhard et al. [23] introduce the notion of object-flow analysis for identifying and comprehending dependencies between features. Fine-grained dynamic analysis traces the transfer of object references at runtime. Our approach, in addition to tracing all transfers of object references, tracks flows through primitive values. Yazdanshenas and Moonen [24] present a technique that supports system-wide tracking of information flow in component-based systems. The technique is shown to be useful in software development and software certification.

Dependence graphs play an important role in modern debugging tools. WhyLine [9] is based on causal relationships between the output and the program's execution, using both static and dynamic analysis. The developer can ask "`why something happened/did not happen`". The information flows detected by our approach can feed a tool to provide similar responses about an execution, though only

for "`why`" questions. The "`Why did not`" questions require some static analysis, which we do not support. The dynamic analysis tool presented by [10] uses a variation of DDGs to represent a program's execution. It enables the querying about an execution history, rather than a specific state.

IX. Conclusion

We present a language-independent approach for tracking information flow within a program execution, motivated by the industry trend towards shared frameworks for implementing languages. We believe such a framework should also include tools for use with those languages. Access to tools increases the usability and thus the popularity of languages, As soon as a new language is released, mature tools will be available for assisting programmers with program comprehension tasks.

We implemented an engine for tracking information flow that can be adapted to a language by labeling AST node classes of the language. We represent information flow using a variant of dynamic dependence graph called shadow graph, which can be the basis for many program comprehension techniques such as program slicing, query, and debugging. Our case study compared our approach for TruffleRuby, a Truffle implementation of Ruby, against an approach based on Java-bytecode for JRuby. In order to extract the shadow graph out of the program execution, we instrumented the ASTs and bytecode arrays for the two approaches. Case study results show that our approach provides information closer to source code constructs, thus more user comprehensible. We are considering applying this approach to other Truffle languages.

Acknowledgment

The authors are indebted to members of the Virtual Machine Research Group at Oracle Labs and the Institute of System Software at the Johannes Kepler University Linz for creating the language implementation technologies that make this work possible. We thank group member Christian Humer in particular for his contributions to parts of this work.

References

[1] A. Sarimbekov, L. Stadler, L. Bulej, A. Sewe, A. Podzimek, Y. Zheng, and W. Binder, "Workload characterization of jvm languages," *Software: Practice and Experience*, 2015.

[2] S. Horwitz and T. Reps, "The use of program dependence graphs in software engineering," in *Proceedings of the 14th International Conference on Software Engineering*, ser. ICSE '92. New York, NY, USA: ACM, 1992, pp. 392–411.

[3] M. Weiser, "Program slicing," in *Proceedings of the 5th international conference on Software engineering*, ser. ICSE '81. Piscataway, NJ, USA: IEEE Press, 1981, pp. 439–449.

[4] ——, "Programmers use slices when debugging," *Commun. ACM*, vol. 25, no. 7, pp. 446–452, Jul. 1982.

[5] H. Agrawal and J. R. Horgan, "Dynamic program slicing," *SIGPLAN Not.*, vol. 25, no. 6, pp. 246–256, Jun. 1990.

[6] K. Chen and V. Rajlich, "Case study of feature location using dependence graph," in *Program Comprehension, 2000. Proceedings. IWPC 2000. 8th International Workshop on*. IEEE, 2000, pp. 241–247.

[7] A. Rohatgi, A. Hamou-Lhadj, and J. Rilling, "An approach for mapping features to code based on static and dynamic analysis," in *The 16th IEEE International Conference on Program Comprehension*. IEEE, 2008, pp. 236–241.

[8] T. Eisenbarth, R. Koschke, and D. Simon, "Locating features in source code," *IEEE Transactions on software engineering*, vol. 29, no. 3, pp. 210–224, 2003.

[9] A. J. Ko and B. A. Myers, "Finding causes of program output with the java whyline," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 2009, pp. 1569–1578.

[10] M. R. Azadmanesh and M. Hauswirth, "Blast: Bytecode-level analysis on sliced traces," in *Proceedings of the Principles and Practices of Programming on The Java Platform*. ACM, 2015, pp. 152–158.

[11] D. E. Denning, "A lattice model of secure information flow," *Commun. ACM*, vol. 19, no. 5, pp. 236–243, May 1976.

[12] S. Horwitz, J. Prins, and T. Reps, "Integrating noninterfering versions of programs," *ACM Trans. Program. Lang. Syst.*, vol. 11, no. 3, pp. 345–387, Jul. 1989.

[13] B. P. Miller and J.-D. Choi, "A mechanism for efficient debugging of parallel programs," in *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, ser. PLDI '88, New York, NY, USA, 1988, pp. 135–144.

[14] T. Würthinger, A. Wöß, L. Stadler, G. Duboscq, D. Simon, and C. Wimmer, "Self-optimizing ast interpreters," in *Proceedings of the 8th Symposium on Dynamic Languages*, ser. DLS '12, New York, NY, USA, 2012, pp. 73–82.

[15] T. Würthinger, C. Wimmer, A. Wöß, L. Stadler, G. Duboscq, C. Humer, G. Richards, D. Simon, and M. 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*, ser. Onward! 2013, New York, NY, USA, 2013, pp. 187–204.

[16] M. L. Van De Vanter, "Building debuggers and other tools: We can have it all. A Position Paper," in *Proceedings of the Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, ser. ICOOOLPS '15. ACM Press, 2015.

[17] C. Seaton, M. L. Van De Vanter, and M. Haupt, "Debugging at full speed," in *Proceedings of the Workshop on Dynamic Languages and Applications*, ser. DYLA '14, New York, NY, USA, 2014, pp. 2:1–2:13.

[18] B. Cornelissen, A. Zaidman, A. van Deursen, L. Moonen, and R. Koschke, "A systematic survey of program comprehension through dynamic analysis," *IEEE Trans. Softw. Eng.*, vol. 35, no. 5, pp. 684–702, Sep. 2009.

[19] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 9, no. 3, pp. 319–349, 1987.

[20] K. J. Ottenstein and L. M. Ottenstein, "The program dependence graph in a software development environment," *SIGPLAN Not.*, vol. 19, no. 5, pp. 177–184, Apr. 1984.

[21] B. Korel and J. Laski, "Dynamic program slicing," *Inf. Process. Lett.*, vol. 29, no. 3, pp. 155–163, Oct. 1988.

[22] J.-F. Bergeretti and B. A. Carré, "Information-flow and data-flow analysis of while-programs," *ACM Trans. Program. Lang. Syst.*, vol. 7, no. 1, pp. 37–61, Jan. 1985.

[23] A. Lienhard, O. Greevy, and O. Nierstrasz, "Tracking objects to detect feature dependencies," in *Program Comprehension, 2007. ICPC'07. 15th IEEE International Conference on*. IEEE, 2007, pp. 59–68.

[24] A. R. Yazdanshenas and L. Moonen, "Tracking and visualizing information flow in component-based systems," in *Program Comprehension (ICPC), 2012 IEEE 20th International Conference on*. IEEE, 2012, pp. 143–152.