The Design Process of Soot-CSI, or: the Joys and Terrors of Java Static Analysis

Nate Deisinger

Abstract

This document is a summary and explanation of the work I have carried out as an undergraduate working with Professor Ben Liblit over the 2013-2014 academic year at the University of Wisconsin-Madison.

Our initial goal was to take the work of Peter Ohmann, who had developed a novel, lightweight instrumentation framework to aid in debugging C and C++ programs, and apply it to Java. In the process we encountered a variety of issues both abstract (appropriate modeling of CFGs) and practical (the performance intricacies of the JVM) that spoke to the complexity of carrying out good static analysis on Java programs. This document is presented along with a code release of our tools (known as Soot-CSI) as they stand now in the hopes it will be of use to both those carrying on this work directly and anyone interested in related topics of analysis and instrumentation for Java.

1. Introduction

According to Ohmann and Liblit, debugging is difficult and costly, and full tracing of the program's execution is useful for debugging but impractical to deploy. To alleviate this problem, Ohmann and Liblit developed CSI-CC[1]: a specialized C and C++ compiler based off of the LLVM framework which can apply two forms of lightweight tracing: efficient path profiling and call coverage. This information can be recovered upon a crash and used in conjunction with a standard stack trace to provide a reduced view of potentially-executed code and to restrict static slices of the program's program dependence graph (PDG). The details of implementation and the algorithms involved are provided in the associated paper[2]; for the purposes of this document, they will not be re-explained except where modification was necessary.

Ohmann and Liblit's results were highly promising, providing reductions in static slices of 49-78% and an overhead between 0% and 5%, making it feasible to deploy instrumented code in production. These results encouraged us to port the method to the Java language, both for the potential of using it to help Java developers debug their code and to gain a better understanding of the intricacies of Java static analysis and instrumentation. Although some work remains before our tools can be applied in the same manner as Ohmann's, we are now able to instrument Java code to perform path profiling and call coverage. This document will discuss the design choices we made and hurdles we needed to overcome,

---

[1] See https://code.google.com/p/csi-cc/

[2] P. Ohmann and B. Liblit. Lightweight control-flow instrumentation and postmortem analysis in support of debugging. *In 28th International Conference on Automated Software Engineering (ASE 2013)*, Palo Alto, California, Nov. 2013. IEEE and ACM.

and it is our hope that it will be instructive for others attempting static analysis and instrumentation of Java programs.

The remainder of this document will proceed as follows:

- The first three sections focus on the high-level design choices of our implementation. Section 2 discusses the aspects of Java that complicate static instrumentation and analysis. Section 3 discusses the restrictions we placed on our tools. Section 4 explores the various options for instrumentation we considered.
- The following two sections discuss the practicalities of Java and how we implemented our work. Section 5 explains the difficulties and necessity of accounting for exceptional control flow in our instrumentation, while section 6 discusses performance issues unique to Java.
- The final three sections discuss areas of work that are not yet completed. Section 7 discusses in detail the difficulties of creating good Java PDGs; section 8 discusses future work once that hurdle is cleared. Section 9 concludes.

2. Fundamental Java Issues

Java is a managed language; source code is compiled to Java bytecode which is then run in a virtual machine. While convenient to the end user, this introduces several difficulties in performing instrumentation and analysis.

2.1 Crashes Without Crash Dumps

In programs run as native executables, a program may crash due to issues such as accessing invalid memory addresses. When this happens, the kernel will kill the process and dump a snapshot of the program's state; this process is known as *dumping core*. By examining the core dump after the fact, the program's internal variables and stack may be analyzed.

As Java programs are run in a virtual machine (VM), their memory and data structures are abstracted away from the kernel; a Java program that 'crashes' will not cause the kernel to kill the process, but rather will result in the Java VM (JVM) gracefully shutting down. (Issues in the JVM that would cause it to catastrophically fail and dump core are of course possible; however, we are focused on debugging Java programs, rather than the VM itself.)

The Java equivalent of a program crash is an *uncaught exception*. When a Java program performs an illegal operation, the code will throw an exception which is propagated up the stack until an appropriate exception handler is found; if the exception reaches the top of the stack without finding a handler, the JVM prints a stack trace, terminates the program, and closes gracefully.

This presents us with a lack of an equivalent to a core dump. Ohmann and Liblit's technique for instrumentation relies on being able to extract instrumentation trace data from a core dump after a program crashes; in Java, since a crash results in the program being cleanly terminated and the JVM closing normally, there is no core dump from which to read this data. Even in the event the JVM itself were to crash, the Java data structures of the program running in the JVM would be laid out in a platform-specific way and near-impossible to parse correctly.

As such, our instrumentation must be able to dump trace data before the JVM terminates the program. This is complicated by the fact that exceptions unwind the stack as they propagate, making it impossible to access stack-local variables. We will discuss this issue further below.

2.2 Write Once, Run Everywhere

As mentioned above, Java is a managed language; it is compiled to Java *bytecode*, which is fed as input to a JVM. As such, a program may run on one of many different platform-specific or purpose-specific JVMs. In addition to ruling out the possibility of designing tools based off of a JVM core dump, this heavily suggests our tools should adjust program bytecode, so that our instrumented code can run on any JVM.

2.3 Exceptions are not Exceptional

Java exceptions are meant to indicate irregular program flow, generally some sort of error that requires handling. In reality, they are commonly used as specialized control flow and implemented into the critical path of a program. Further complicating matters is that users may define their own exception classes, or choose to create handler code for JVM-inherent exceptions (such as a null pointer dereference).

For us, this means that the problem of exceptional control flow cannot be ignored; if we want an accurate picture of a program's execution or a full PDG, we need to take exceptional control flow into account and ensure our algorithms work with it.

2.4 The Heap is King

In C and C++, nearly any variable can be stack-allocated; storing our trace data there is thus cheap and easy to locate in a core dump. This is the technique used by Ohmann and Liblit.

In Java, only the most primitive types (such as integers, chars, and booleans) are directly stack-allocated; anything more complex, even arrays of the above-mentioned, are ultimately heap-allocated, with only reference variables (managed pointers) left in the stack. We thus must be careful in how we store our trace data to avoid costly heap allocation during runtime. Our current implementation attempts to minimize heap allocation by storing our trace data as individual integers or booleans within a method's stack frame. We then transfer this data to the heap upon an exception arising to avoid it being lost due to the stack unwinding. This will be discussed further in section 6.

A side effect of this heap-focused paradigm is the extreme prevalence of reference variables, which complicates data-flow analysis; this will be discussed in more detail in section 7.

3. Self-imposed Restrictions

With the above in mind, we made three important restrictions on our design:

- **Do not adjust the JVM.** We could have implemented much of our instrumentation and trace data-collection through using a modified JVM, such as an addition to the Jikes RVM[3]. However, in the everyday use case, a user will not want to install a separate JVM to run a single program; indeed, on many platforms, this may be impossible. Thus we restrict our methods to adjusting the bytecode of the program so it will run on as many JVMs as possible.
- **Minimize dependencies for the end user.** Similarly to the above, the more the user needs to adjust their workflow, the less likely they will be willing to run an instrumented program (again, if even able). We considered developing a plugin for the JVM that would have used the Java Debug Interface (JDI) to collect trace data upon a crash; however, this would require the user to install this additional plugin and adjust their JVM environment, and would have added a significant performance overhead. We aim to make instrumented programs that will be transparent to the end user.
- **Keep it in the world of Java.** As Ohmann and Liblit's tools are based off of the LLVM framework, we initially explored if we could carry over their tools directly. LLVM does not have a full-fledged Java frontend or backend as it stands today, and while projects such as RoboVM[4] are in development to convert Java code to x86 executables through LLVM, ultimately this would sacrifice too much compatibility.

Other considerations were also taken into account: we want our instrumented code to remain performant, and we want to minimize the difficulty of instrumenting the code and gathering the trace data for the developer as well.

4. Instrumentation Candidates

With the above restrictions in mind, we considered several possible frameworks for performing our instrumentation.

4.1 WALA

The T. J. Watson Libraries for Analysis (WALA)[5] are a set of tools designed to perform static analysis on Java bytecode; they are primarily packaged as a set of Eclipse plugins.

WALA also includes a bytecode instrumentation library, Shrike, which presents the bytecode of a program as an array that can be manipulated. While undoubtedly powerful and used in several projects, we ultimately decided against WALA both for its reliance on the Eclipse platform and for the unfriendliness of its direct bytecode manipulation; we desired an instrumentation framework that provided a higher-level IR which could be manipulated.

4.2 ASM

ASM[6], like WALA, is a bytecode manipulation framework primarily designed to work directly on Java bytecode. ASM is particularly geared towards runtime use for dynamic class generation and

[3] http://jikesrvm.org/
[4] http://www.robovm.org/
[5] http://wala.sourceforge.net/wiki/index.php/Main_Page

modification, and thus aims to be small and efficient; as a result, it does not provide any sort of intermediate representation or many built-in analyses. While a possible candidate, we hoped to find a framework that provided a higher-level IR and some existing analyses so that we would not need to reinvent the wheel of control flow graphs (CFGs) and other basic analysis structures.

4.3 Soot

Our final candidate for bytecode instrumentation, and the solution we ultimately settled on, is the Soot Java Optimization Framework[7]. Soot is a well-developed, feature-rich framework for static analysis and instrumentation of Java programs at either the source or bytecode level.

What made Soot particularly attractive was its intermediate representation for bytecode, Jimple, which provides a view of code much closer to Java source than direct bytecode, making it easier to reason about and implement our instrumentation. Soot also provided existing analyses to create control-flow graphs and, to a limited extent, call and data-flow graphs. Soot also features a well-documented API[8], tutorials[9], and an active mailing list[10].

For further information on how our tools use Soot, please consult the readme of this document's associated code release. When pertinent, this document will describe features of Soot relevant to our implementation of our analyses.

5. Handling Exceptions and their Handlers

As discussed above, one of the major issues we faced in translating our algorithms to Java was handling exceptions correctly. Since exceptions play an important role in control flow – particularly in paths that lead to a crash – ignoring them would severely limit the usefulness of our analyses. Unfortunately, handling exceptions comes with a new set of challenges:

---

[6] http://asm.ow2.org/
[7] http://www.sable.mcgill.ca/soot/
[8] http://www.sable.mcgill.ca/soot/doc/index.html
[9] http://www.sable.mcgill.ca/soot/tutorial/
[10] https://mailman.cs.mcgill.ca/mailman/listinfo/soot-list/

5.1 Explosion in CFG Size

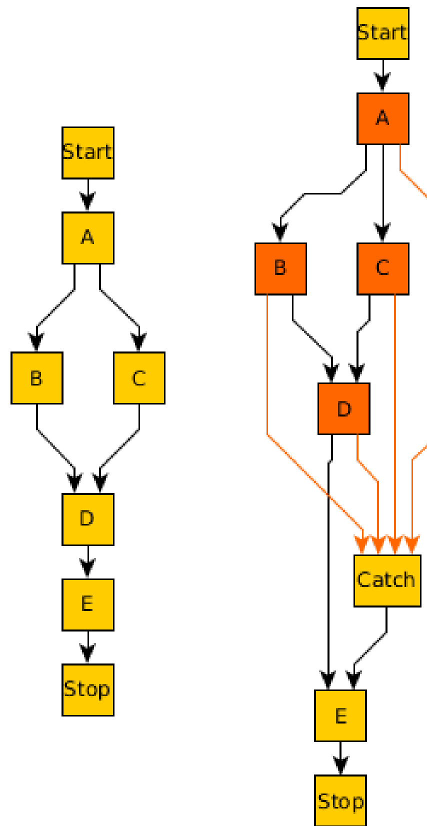

Figure 1: A CFG for a simple method containing one conditional branch.
Figure 2: A CFG for the same method when basic blocks A through D are wrapped in a try/catch block. Exceptional edges are colored orange, while standard control flow edges are colored black.

The immediate challenge of dealing with exceptional control flow is that it greatly expands the number of possible execution paths within a program, particularly if it is handled naïvely.

Consider the above figures. Figure 1 shows a basic block-level CFG for a simple piece of code: an 'if' statement followed by code that executes regardless of which branch is taken. Figure 2 shows that CFG when the branch portion of this code is wrapped in a Java try/catch block. The number of possible execution paths jumps from 2 to 7 immediately; this number would increase further if the try/catch block handled more than one exception, or if a "finally" block was introduced as well.

In other words, the naïve handling of exceptional control flow adds at least one path per CFG node in the try/catch block. This implies additional instrumentation is required to correctly profile all possible execution paths, and complicates the CFG.

5.2 Restricting Exceptional Edges

Java contains two types of exceptions: *checked* and *unchecked*. Checked exceptions are comparatively simple to analyze; the Java language requires that a checked exception be either explicitly caught or that

the method in which it is thrown explicitly declares as such. If the method declares it may throw the exception, this same catch/throw requirement propagates up to callers.

Unchecked exceptions do not carry this same requirement; they may be thrown explicitly or implicitly at any time, and it is the programmer's job to ensure proper handler code exists. Errors such as a null pointer dereference or accessing an array beyond its bounds manifest as unchecked exceptions.

While one could imagine using checked exceptions to restrict the CFG's exceptional edges – it would be possible to analyze exactly which statements (and thus, basic blocks) could potentially throw a checked exception, and use this information to remove spurious edges from the CFG – unchecked exceptions are more difficult to handle. While user-defined unchecked exceptions must be explicitly thrown, JVM-based exceptions (such as null pointer dereferences) are generally raised implicitly.

It may be possible to enumerate these exceptions and the instructions which may throw them and use this information to trim the CFG. One could imagine an analysis which analyzes the instructions of a method, enumerates which exceptions may be thrown, then considers existing exception handlers. Any potentially-thrown-and-uncaught exceptions would be propagated up to the method's callers, where the analysis would repeat. When the process is complete, the information could be used to remove exceptional control-flow edges which would never occur.

At present, we have not implemented such an analysis; however, if naïve exceptional control flow modeling proves restrictive, Soot provides an analysis which gets us halfway there. The UnitThrowAnalysis class of Soot takes a single statement (in Soot parlance, a Unit) and returns a list of exceptions it may throw (explicitly or implicitly). Using this, building a basic-block level throw analysis would be simple, and some work could allow for interprocedural throw analysis as well.

5.3 Working With Exceptions in Soot

The Ball-Larus path profiling algorithm aims to associate a unique value with each possible acyclic execution path in a method. In order to collect this data, the algorithm selects certain edges in the CFG which will update a path counter that, upon function exit or a backedge, will contain the unique value corresponding to the acyclic path taken to that point. In modeling the CFG with exceptional control flow, it is natural that some of the instrumented edges will be exceptional ones.

Unfortunately, adding instrumentation to an exceptional control flow edge is non-trivial. Whilst ordinary jumps in the CFG can be handled by redirecting the jump to a piece of instrumentation code that increments the path counter the appropriate amount, exceptional flow has no explicit jumps.

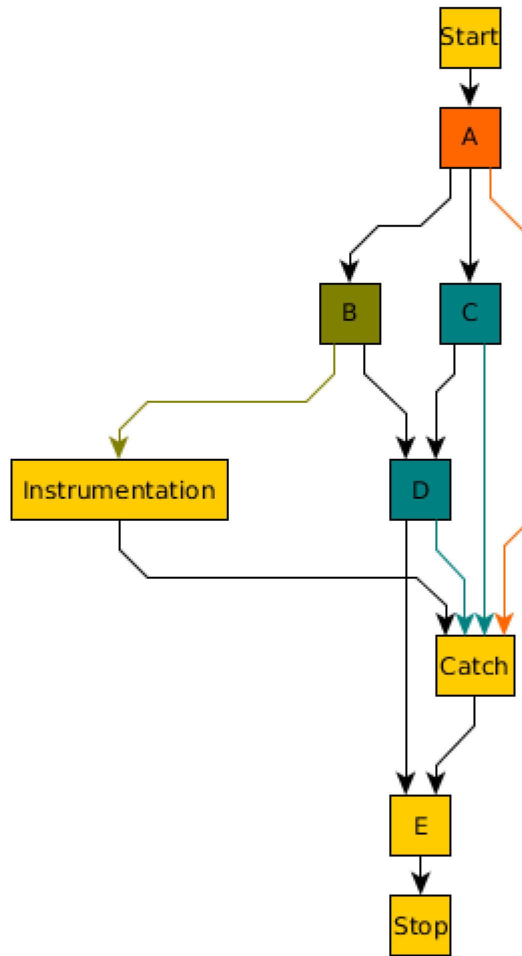

Figure 3: The CFG of the method from figures 1 and 2 when the exceptional edge between block B and the catch block is instrumented. Different colors indicate distinct traps and their associated exception handler edges. One trap contains block A, one trap contains block B, and one trap contains blocks C and D, as they are contiguous in the function.

In Soot, exceptional flow is defined in terms of 'traps', which specify a range of basic blocks where, if an exception is caught, execution will redirect to a handler statement (in a separate basic block.) In our current model, we consider each block inside a trap to have an edge to the handler block, as any block could potentially throw an exception and cause execution to jump to the handler block. To instrument one of these edges, we cannot simply redirect the handler unit of the trap, as this would result in multiple blocks erroneously pointing to the same instrumentation code. Instead, we must split the trap into multiple traps covering smaller ranges of blocks.

Our current implementation checks the range of blocks being trapped and splits the trap into one, two, or three traps depending on which blocks are being trapped. If only the instrumented block is trapped, a single trap is created; if the first or last block is trapped, two traps are created; otherwise, three traps are created. In the event multiple traps are created, one trap contains the block whose exceptional CFG edge is being instrumented; its associated handler unit becomes instrumentation code that increments the path counter and then jumps to the original handler code. The other traps handle

the blocks before or after this instrumented block, and retain the original handler behavior. Figure 3 shows the results of this.

6. Keeping our Instrumentation Performant

As mentioned in section 2, Java has a number of peculiarities, such as a heavy focus on heap allocation, that make it difficult to remain performant. We describe our strategies for high-performance instrumentation below:

6.1 Keep Data in the Stack

Ohmann and Liblit's design allowed for arrays of booleans to indicate call coverage and arrays of integers to indicate paths taken over time. Java does not allow for stack-allocated arrays; we thus need to dispose of the array abstraction and store all instrumentation data (such as path profiling values and call coverage information) in individual stack-allocated booleans and integers. While not as immediately elegant, Soot enables us to keep track of these variables and insert code to dump them upon a crash in a reasonable fashion.

Related to this issue is the problem of keeping global information. Roughly, the Java equivalent of a global variable is a static class variable. These are initialized upon use of a class (or explicitly through use of the Java Class class)[11] and are more likely to be less performant than stack-allocated variables, as they will be held in memory rather than registers. If the JVM is intelligent enough, it may move these variables to registers when they are being frequently accessed, but there is no guarantee. Thus we aim to avoid overuse of static variables; however, for accurate global call coverage, this is unavoidable.

6.2 Avoid Method Calls and Initializations

The JVM may or may not perform inlining on Java methods automatically during execution; nonetheless, in our tests, we found making method calls a common part of our instrumentation caused an unacceptable drop in performance. (For example, a dummy method that simply adds two numbers inserted before all function returns causes a performance overhead of [%].)

A key part of our instrumentation design for path tracing involves the existence of a shadow stack of instrumentation objects ("PathTrace"s) which can store a method signature and the path counter value representing what execution path has been taken through it. Recall that an exception in Java unwinds the stack as it propagates up. In the process, all stack-allocated variables are discarded and become impossible to access. In order to ensure we do not lose our trace data as the stack unwinds, it is necessary to move it to one of these PathTrace objects when a method terminates due to an uncaught exception. This shadow stack then retains the data even as the program stack unwinds. If an exception escapes the program, we are able to unwind this shadow stack and print the associated trace data for each method that was on the stack when the uncaught exception was thrown.

---

[11] http://docs.oracle.com/javase/specs/jls/se7/html/jls-12.html#jls-12.4.1

Zealously creating, updating, and removing these objects as the stack grows or shrinks proved to be prohibitively expensive. Thus, we made the following design choices:

- **Pre-allocate these objects.** When the PathTrace class is initialized, it automatically allocates 50 PathTrace objects, placing a single concentrated allocation cost at program start, rather than many allocation costs throughout execution.
- **Delay initialization as long as possible.** Although our instrumentation adds a local PathTrace object to each method, this object is initialized to null. Only when a method throws an uncaught exception does it call a method to take one of the pre-allocated PathTrace objects and initialize it with its method signature, path value, and call coverage.
- **Eliminate getters and setters.** While standard object-oriented practices call for the use of getter and setter methods, it is more performant for us to make our instrumentation fields public and set them directly.
- **Make eliminating old data quick and only done when necessary.** Consider the case where a method throws an exception which is caught and handled correctly higher up the stack. In this case, we must conservatively collect information about the methods that do not handle the exception, but we will want to remove this old data when we successfully return from the method that handled the exception. We maintain a boolean to ensure we only do this cleaning in this case and not on every method return, and we perform cleaning by simply moving the pointer of the first 'available' PathTrace object and allowing new data to overwrite the old. In this fashion, the shadow stack matches the unwinding performed by the caught exception.

We make use of similar methods for storing call coverage information, though its focus on global information requires making heavier use of global variables, as mentioned above. In the case of call coverage, we allocate static booleans corresponding to each call site in the program at instrumentation time; upon a crash, we print these global values alongside local coverage data from the current stack.

7. The Problems of PDGs

Recall that a major contribution of CSI-CC was the ability to use its trace data to reduce the size of static slices of a PDG. We hoped to carry out the same idea with Java, only to run into an unfortunate lack of support: good Java PDGs are nearly impossible to find.

7.1 Challenges of Java PDGs

Generating good Java PDGs is a non-trivial problem. Recall the following issues about Java:

- **Almost everything is pointer-like.** Since anything more complicated than a boolean, integer, or char in Java is stored in a reference variable, we must be able to perform pointer-like data flow analysis if we wish to track many kinds of Java variables.
- **Exceptional control flow is not exceptional.** Proper PDGs need to handle control and data flow for exceptions appropriately. In conservative cases, this can mean control edges from every statement within a try block to a handler statement.

- **Java call graphs may be very large.** As an object-oriented language, Java makes use of many instance and static methods as well as dynamic dispatch due to inheritance, resulting in a large call graph which takes more time to analyze and complicates interprocedural data flow. Good PDGs should also be context-sensitive with respect to call sites to ensure proper interprocedural data flow.
- **Java exists at multiple levels.** While not an inherent issue, recall that Java may be considered either as bytecode or source; choosing which interpretation to build a PDG off of raises questions of precision and translation for later tools to use.

With these in mind, we looked for a good Java PDG solution. Ideally, we would like a PDG generation tool that is interprocedural, context-sensitive, handles object-related data flow accurately, and can work at the source level. It suffices to say that such a tool does not currently exist, or else this section would be much shorter than it is. Described below are several candidates:

7.2 JPDG and JavaPDG

In 2013, a group led by Professor Andy Podgurski at Case Western Reserve University released a tool, JavaPDG, which could analyze Java class files and create bytecode-level PDGs. Initially promising, we found it to be limited due to a reliance on external tools and its strictly bytecode-level PDG representations.

At present, Tim Henderson, with the same group, is working on a revised version of the tool, JPDG. JPDG uses Soot as the backbone of its analysis framework, and exports its graphs to a JSON-serialized format. JPDG is still under heavy development, and has many limitations, presenting only intraprocedural PDGs with data-flow analysis of local, stack-allocated variables only. Early experiments building off of its code to export graphs in a format friendly to our tools were promising, and code related to this endeavor may be found in the "uwmadison.jpdg" package of our code release. However, the aforementioned limitations of JPDG led us to investigate other options.

7.3 Soot and Heros

Soot contains limited built-in support for data and control flow analysis. While control flow is well-modeled – we will explain Soot's CFG representations in more detail below – there is only limited support for data flow and dependency analysis. Soot's built-in classes are capable of performing simple intraprocedural reaching definitions analyses on local variables, which can then be built upon to create data dependency graphs. This is the method currently used by JPDG.

The Heros framework for Soot extends its data flow capabilities to provide a model for solving interprocedural, finite, distributive subset (IFDS) and interprocedural distributive environment (IDE) data-flow problems. As part of its examples, it contains a reaching-definitions analysis. The solution provided improves on Soot's built-in data flow analysis significantly by adding intraprocedural support for instance fields, static fields, and array references, as well as some interprocedural support for data flow through arguments and return values. More precisely, checking the statement corresponding to the formal-in of a function will give a list of the possible definitions for it from its callers. Similarly,

checking a statement corresponding to an assignment from a function will show definitions based on the possible return values of the caller. Intraprocedural exceptional flow is also handled well.

Unfortunately, this analysis also has blind spots. No points-to analysis is performed; thus, aliasing and object fields are not correctly handled. Interprocedural support for static variables is not provided (each function acts as though it has its own copies). Finally, the analysis is context-insensitive.

It is possible Heros' provided framework could be built upon to resolve some or all of these issues; however, in our research we have not undertaken this work.

7.3.1 Soot's CFG Representation

Although Soot may not have full PDG support, it does have well-defined support for creating intraprocedural CFGs, which we have used to implement our path tracing framework.

Soot defines two types of CFGs: BlockGraphs and UnitGraphs. The former classes provide CFGs at the level of the basic block; the latter provide them at the level of the unit.

In our analysis, we make use of the ZonedBlockGraph subclass. This class defines a CFG in which exception boundaries are taken into consideration when defining basic blocks; that is, the first unit handled by some exception handler becomes the start of a new basic block. Similarly, any unit that is the last one handled by an exception handler marks the end of a basic block.

It is important to note that this style of handling exceptions does not account for *when* in the basic block an exception is thrown. If an exceptional edge is taken, one can only say that the block in question was reached, not that any of it was successfully executed.

If a more explicit accounting of exceptional edges is desired, the ExceptionalBlockGraph class breaks each block in a try/catch construct into blocks a single unit long; this significantly increases the size of the CFG, but provides a more precise graph.

7.4 VASCO

VASCO[12] is a data-flow solver framework designed to provide support for solving data-flow problems that cannot be modeled as IDFS or IDE problems. This category includes such problems as context-sensitive call graphs and points-to analyses, and VASCO in its current state provides some existing example analyses of these problems as extensions of Soot.

VASCO is also a much more complicated framework than Soot's built-in functionality or Heros, and its analyses are more expensive in time and memory than Soot's. Still, using VASCO it is possible to get the points-to set for a given variable at a given execution point (one must be careful to disable culling of intermediate results, or the provided analysis will only provide data at the end of a method.)

There are still some limitations present in VASCO; notably, it does not seem to handle points-to analyses correctly for static variables (the reason for this is unknown.) The existing analysis also still

[12] https://github.com/rohanpadhye/vasco

needs code created to adapt its internal ContextSensitiveTransitionTable to a context-sensitive call graph compatible with Soot's interfaces. (Soot defines a ContextSensitiveCallGraph interface but does not natively implement it.)

VASCO was originally the work of Rohan Padhye and Dr. Uday Khedker at the Indian Institute of Technology Bombay; Rohan is now at IBM, and the code has been passed on to Alefiya Lightwala. Rohan has been very helpful in explaining VASCO and its current state, and if this route is pursued, the three can be contacted at rohanpadhye@gmail.com, uday@cse.iitb.ac.in, and alefiyalightwala@cse.iitb.ac.in, respectively.

7.5 WALA

WALA was discussed above when considering instrumentation frameworks. As our investigation at the time was primarily focused on frameworks that could perform bytecode instrumentation at a relatively high level, we passed over WALA in favor of using Soot.

Upon revisiting the problem of Java PDGs, we were pleasantly surprised to find WALA contains some built-in support for Java System Dependence Graphs (SDGs) – that is, full interprocedural PDGs. Unfortunately, due to time constraints, we have not fully explored this option. There are some particulars to keep in mind when working with WALA:

- The project is primarily structured as a set of Eclipse plugins, though it is possible to build and use as standalone libraries.
- WALA's default behavior attempts to analyze any and all libraries on the program classpath; this includes the Java standard libraries. Excluding certain classes is possible by providing a list of packages to ignore, but due to how WALA performs its type analyses some large, commonly-used classes such as java.lang.Object must be included.
- WALA creates PDGs at the bytecode level, though there is some support for mapping these to source line statements.
- WALA has significant time and memory requirements for performing its analysis – far greater than Soot or VASCO. SDGs created in this manner may also be difficult to work with due to their large size.
- WALA's documentation does not always seem to be accurate, and may be difficult to access due to permissions errors. (Consider instead their Google Group[13].)

Nonetheless, WALA is a promising lead for the question of how to create good Java PDGs to slice against, and I regret not considering it for this purpose earlier.

8. Future Work

Recall that our goal was to provide a framework for Ohmann and Liblit's tools and techniques to work with Java code. We have provided a set of tools which perform instrumentation and output trace data in the same format as their work. However, several challenges remain:

[13] https://groups.google.com/forum/#!forum/wala-sourceforge-net

8.1 Support for Multi-Threaded Programs

Our current tools operate on the assumption that the programs being analyzed are single-threaded. Support for multi-threaded programs would allow for analysis related to concurrency and other bugs. Adding functionality would involve not only ensuring that our existing data structures are thread-safe, but also that we are capable of correctly collecting and printing trace data related to multiple threads of execution. One could imagine per-thread side stacks of PathTrace objects that provide data on the execution paths taken by a particular thread.

8.2 Proper Java PDG Construction and Analysis

As section 7 discussed in detail, the problem of creating a precise Java PDG is not trivially solved. At present, the WALA framework shows promise for this purpose; however, hurdles of space and time complexity as well as mapping from bytecode to statement- or basic block-based PDGs still need to be overcome. If WALA does not meet our purposes, it may be necessary to create Java PDG generation software using one of the other discussed methods.

However, we are confident that once PDG support is obtained, working with the PDGs should be comparatively simple; Ohmann and Liblit's framework makes use of the CodeSurfer software, and PDGs and trace data provided in the proper format should theoretically be compatible in spite of having their origins in Java code.

8.3 Java-Specific Instrumentation Strategies

The path profiling and call coverage strategies we have implemented have had their worth shown in Ohmann and Liblit's work. However, as Java is an object-oriented language, ostensibly encouraging of code modularization rather than monolithic functions, it is unclear now how effective they will prove in the world of Java. Other, Java-specific instrumentation, such as recording object field accesses, may prove to provide better trace data for determining program execution.

8.4 Non-PDG Applications

Though the focus of Ohmann and Liblit's work is on slicing against PDGs, the trace data collected could be used for other purposes as well. In particular, Ohmann and Liblit's work describes an algorithm to restrict CFGs based off of this information; one could imagine using this simpler algorithm in conjunction with Soot's CFGs to provide programmers a listing of program nodes that must have or may have executed.

8.5 Optimization and Elegance

A major goal of Ohmann and Liblit's original work was to perform *light-weight* instrumentation; while in our tests we see a fairly small overhead (roughly 2 to 5% with call coverage, and 5 to 10% with path tracing), there is room for improvement. Call coverage in particular relies heavily on references to heap-allocated static variables enumerated at instrumentation time, making its instrumentation monolithic and raising potential performance issues. Implementing a callback system which allows classes to register static booleans for global coverage upon initialization may allow for more flexibility in how our analyses can be applied, as well as avoid maintaining coverage data for classes that go unused.

9. Conclusion

We have successfully applied the instrumentation techniques of Ohmann and Liblit's work to Java through use of the Soot framework, in the process discovering many challenges unique to Java both in terms of practice and theory. The model of running programs in the JVM raises issues of performance and requires more intricate methods of collecting and reporting trace data than waiting for a core dump; language features such as exceptions complicate both instrumentation and program flow modeling.

It is our hope that this document has provided a good general overview of the complications posed by Java static analysis and provided a list of possible solutions, both implemented and proposed. It is also our hope that our discussions will prove at least somewhat applicable for other languages as well, be that support for analyzing C++ exceptions or for work in other managed languages such as C# or Python.

Acknowledgments

Thanks must be given first and foremost to my advisor, Dr. Ben Liblit, who has put up with my repeatedly running into a different brick wall once a week for an entire academic year. His good humor and support has been invaluable.

Peter Ohmann created the work which my research built off of, and throughout the year was consistently helpful in explaining or clarifying concepts and in providing information on issues of data formatting and tools.

Alisa Maas has consistently provided insightful feedback, as well as come up with half a dozen scenarios to break whatever advancement I thought I might have that week, the sign of a truly great researcher. She also pointed me towards WALA's PDG support when it had completely slipped my mind.

Ben, Peter, and Alisa also lent their comments to this document, helping transform it from a jumble of words to a slightly-less-chaotic jumble of words.

Dr. Andy Podgurski and Tim Henderson at Case Western Reserve University have been extremely helpful in explaining their work in creating Java PDGs. Even if we ultimately choose to use WALA for our PDG generation, I hope this document is of use to them.

Dr. Uday Khedker and Rohan Padhye were similarly responsive and helpful with questions about their VASCO framework and how it might be extensible to a PDG creation system.

The people of the Soot mailing list, particularly Dr. Eric Bodden, were invaluable in providing suggestions and feedback on how to use Soot for our instrumentation and analysis.