Variably Interprocedural Program Analysis for Runtime Error Detection

Aaron Tomb
Univ. of California, Santa Cruz
Santa Cruz, CA, USA
atomb@soe.ucsc.edu

Guillaume Brat
RIACS/NASA Ames
Moffett Field, CA, USA
brat@email.arc.nasa.gov

Willem Visser[*]
SEVEN Networks
Redwood City, CA, USA
willem@gmail.com

ABSTRACT

This paper describes an analysis approach based on a combination of static and dynamic techniques to find run-time errors in Java code. It uses symbolic execution to find constraints under which an error (*e.g.*, a null pointer dereference, array out of bounds access, or assertion violation) may occur and then solves these constraints to find test inputs that may expose the error. It only alerts the user to the possibility of a real error when it detects the expected exception during a program run.

The analysis is customizable in two important ways. First, we can adjust how deeply to follow calls from each top-level method. Second, we can adjust the path termination condition for the symbolic execution engine to be either a bound on the path condition length or a bound on the number of times each instruction can be revisited.

We evaluated the tool on a set of benchmarks from the literature as well as a number of real-world systems that range in size from a few thousand to 50,000 lines of code. The tool discovered all known errors in the benchmarks (as well as some not previously known) and reported on average 8 errors per 1000 lines of code for the industrial examples. In both cases the interprocedural call depth played little role in the error detection. That is, an intraprocedural analysis seems adequate for the class of errors we detect.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Symbolic execution*

General Terms

Reliability, Verification

Keywords

Test generation, symbolic execution, defect detection

[*]This author contributed while at RIACS/NASA Ames, Moffett Field, CA

1. INTRODUCTION

As software becomes more complex, and achieves an increasingly critical role in the world's infrastructure, the ability to uncover defects becomes more and more crucial. Software flaws are estimated to cost the United States economy alone tens of billions of dollars annually [24]. In response, many approaches to automated defect detection have been explored.

Program analysis tools aimed at defect detection for procedural or object-oriented languages can be divided into path-sensitive and path-insensitive approaches. Path insensitive approaches, such as those based on abstract interpretation, are traditionally used to show the absence of errors and can produce large numbers of spurious warnings — unless the domain of programs and/or the error classes are suitably restricted, as in the ASTRÉE tool [5]. Path sensitive approaches tend to be focused on finding errors and either use whole program analysis (from some *main* method), or intraprocedural analysis, where each procedure is considered separately. Software model checkers, such as Java PathFinder [30], BOGOR [25], and SPIN [15] take the former approach and static analyzers, such as ESC/Java [12], take the latter approach.

Because we are interested in error detection, we will focus on the path sensitive approach here. In particular we want to investigate the spectrum of analyses that fall between the extremes of whole program analysis and intraprocedural analysis. In addition, since it is often hard to determine the specification a program should adhere to, we will focus on implicit correctness properties that, when violated, result in Java runtime exceptions. Specifically, we look for null pointer dereferences, out-of-bounds array accesses, negative array size exceptions, class cast exceptions, divisions by zero, and assertion violations.

Even when doing a path sensitive analysis, spurious warnings can still occur. The negative impact of such false warnings on the successful uptake of an error detection tool cannot be stressed enough; for every warning, a human must consider *all* contexts in which the code can be executed to determine if the error is real or not, and this is a daunting task. Our goal is to try and reduce this effort as much as possible. More precisely our analysis has zero spurious warnings within the context of the top-level method being analyzed. In other words, if it is possible for the top-level method to be called with the discovered error-inducing inputs, then the error reported is real. Of course it is still possible that, in the context of the rest of the program, the method cannot be called with the inputs that expose the error. However, at the very least, this shows an assumption being made by the method that is not explicitly checked in the code.

In order to reduce spurious warnings, we first ensure that we prune most infeasible paths during symbolic execution by passing the current path condition (*i.e.*, the constraints on the input required to reach the current state) to a decision procedure to check for satisfiability. Next, when we detect a possible error we solve the path condition to obtain test inputs that may expose the error. Finally, we run the code with these inputs and only report an error if the test raises the expected exception.

The Check 'n' Crash tool [8] takes a similar approach. It uses ESC/Java [12] to generate constraints and JCrasher [7] to execute the derived tests. In contrast, we use our own symbolic execution engine to determine the effect of interprocedural analysis on the error detection capability (whereas ESC/Java uses only an intraprocedural analysis). We start by doing an intraprocedural analysis and then in the subsequent analyses follow call chains 1 level deep, 2 levels deep, and so on, while keeping track of the errors detected. We also allow the termination condition for the analysis of a path to be customized: we can either set a bound on the maximum size of the path condition or set a limit on how many times a specific bytecode instruction can be revisited.

We have evaluated our tool on the same small examples used by Check'n'Crash [8] and also on five larger examples which range from 3,000 lines of code (LOC) to 48,000 LOC. For the small examples we discovered all the errors reported by Check'n'Crash, plus a few more. For the larger examples, we found an average of about 8 errors per 1,000 LOC. During the evaluation we calculated a number of statistics on the performance of the technique that resulted in the following interesting observations:

- The level of interprocedural analysis played no noticeable role in the discovery of new errors. However, a more deeply interprocedural analysis did result in more constrained path conditions and thus fewer warnings. Increasing the level of interprocedural analysis greatly increased the execution time.

- For the benchmark examples, using a decision procedure to detect infeasible paths pruned away very few paths in an intraprocedural analysis (less than 10%, on average), but this percentage increased closer to 20% during interprocedural analysis. In fact, the increase is enough to make infeasibility checking indispensable at even one level of interprocedural analysis. For the larger examples, the infeasibility checks were much more important and we found as many as 50% of the paths to be infeasible.

- The size of the path condition played a very small role in discovering errors. That is, analysis that allowed only a small path condition was almost as likely to find a given error as analysis that allowed a larger one. Again, as expected, the larger the allowed path condition length, the longer the execution time.

- The time spent calculating path feasibility dominated the total execution time in almost all cases (although there was at least one exception, in which the generation and execution of the tests took longer).

- Null pointer errors dominated all others. In most cases they accounted for more than 85% of all errors found.

- The number of errors discovered per 1,000 LOC seems to be a good indication of the code quality. For mature and well used code, we discovered around 1 error/1,000 LOC, whereas research prototype code averaged around 10 errors/1,000 LOC.

- The type and number of errors found can indicate coding style. One example had fewer null pointer dereferences since many methods compared their inputs to null before proceeding.

The main conclusion of our work is that an intraprocedural analysis using symbolic execution and test generation seems to be sufficient to find a large number of possible errors. We conjecture, however, that to find deeper behavioral errors this simple approach may not be as effective.

The following section describes variably interprocedural analysis more formally. Section 3 then shows a simple example that illustrate the advantages of our approach. Section 4 describes the implementation of our tool, along with our experience building it, while Section 5 describes its performance on a variety of target programs. In Section 6 we discuss a few optimizations that we'd like to do in the future. Finally, Section 7 gives an overview of related work, and Section 8 contains some concluding remarks.

2. VARIABLY INTERPROCEDURAL ANALYSIS

Variably interprocedural analysis operates on a configurable set M of top-level methods. For whole-program analysis this set is the singleton set containing the entry point of the program. For intraprocedural analysis, it is usually the set of all methods in the program. Other analyses may use different sets.

We associate each method m in M with a call depth CD_m. The call depth CD_m indicates how many levels of sub-calls of m will be explicitly analyzed when beginning with m at the top level.

The process of analyzing a method m with a call depth CD is as follows:

- If $CD < 0$, stop.

- Analyze each statement in m.

- When encountering a call to method m', evaluate m' with maximum call depth $CD - 1$

Because object-oriented programs rarely access instance fields directly, but instead use accessor methods, we hypothesize that setting $CD_m = 1$ for all m in M will yield a significant immediate benefit by making it possible to reason about field values. Other configurations are probably also useful, and the best settings for M and each CD_m may be application-dependent.

3. AN EXAMPLE

As an illustration of some of the advantages of variably interprocedural analysis, consider the program in Figure 1 and the problem of detecting null pointer dereferences. One approach might consist of dataflow analysis, with a lattice

```
01: class Example {
02:   public String hexAbs(int x) {
03:     String result = null;
04:     if(x > 0)
05:       result = Integer.toHexString(x);
06:     else if(x < 0)
07:       result = Integer.toHexString(-x);
08:     return result.toUpperCase();
09:   }
10: }
```

Figure 1: A simple Java program that illustrates some benefits of symbolic execution.

describing null, possibly null, and non-null values for the program variables. Assuming that the analysis does not have access to any specification for the `Integer.toString` method called on lines 5 and 7, it would likely report a possible null pointer dereference on line 8.

Using variably interprocedural symbolic execution, we can do better. If we set the analysis to evaluate all method calls to a depth of 1, we can follow the calls to `Integer.toString`, and determine that they never return null values. Then, because it is a path-sensitive analysis, it can determine that a null pointer dereference can only happen (and must happen) if $x = 0$. Thus, we have ruled out the false positives (the assignments on lines 5 and 7), and given more information about the true error (the missing case for $x = 0$). Given the constraint on x, it is then trivial to construct a test case that will trigger the expected exception.

4. IMPLEMENTATION

We have implemented a tool to apply our technique to the detection of unhandled runtime exceptions in Java bytecode. Our tool builds on the Soot framework for Java bytecode optimization [29]. Soot provides a large body of static analysis code that, while intended for optimization, is also useful for defect detection. The tool consists of approximately 8,000 lines of Java code, of which about 5,300 are devoted to symbolic execution, and around 2,700 to test generation and execution.

4.1 Architecture

Our tool is structured, roughly speaking, as a three-stage pipeline. The first stage performs variably interprocedural symbolic execution, and generates a set of symbolic representations of programs states which, if reached, may result in a runtime exception. The second stage then solves the constraints imposed by these symbolic states to obtain concrete method parameter values suitable for testing. Finally, the third stage uses the Java reflection API to invoke the methods under analysis, with the parameters obtained in the second stage.

4.1.1 Symbolic Execution

The tool begins symbolic execution by analyzing a specified set of methods, instruction by instruction, as illustrated by the Java-like pseudocode in Figure 2.

For each method under analysis, we call **exec** with a reference to the first instruction of the method, and an "empty" symbolic state, representing the lack of any information about the state of the program. The data structure describing this state consists of a path condition and three maps that take local variables, object fields, and array entries, respectively, to symbolic expressions involving only constants and immutable input variables.

As the tool processes each instruction, it updates the symbolic state to reflect the instruction's effects. During this process, instructions that may throw runtime exceptions are handled specially. If the (symbolic) values of the instruction's operands indicate that a runtime exception is possible (*e.g.*, that the base of a field reference or the receiver of a method call may be null, that the denominator of a division operation may be zero (Figure 2, line 16), or that an array index might be out of bounds (line 42)) the tool generates a warning.

In most cases, determining whether a runtime exception is possible requires deciding the satisfiability of an arbitrary formula in first order logic (extended with arithmetic and arrays). To make this decision, we query an external decision procedure, currently CVC-Lite [1]. We denote decision procedure queries in the pseudocode by calls to **satisfiable**.

Method calls can be handled in two different ways (lines 19-38). Each method has an associated call depth (which may be global, affecting all methods, or specified separately for each method). When analyzing a method at the top level, we pass in this depth as a parameter. When we encounter a call instruction, we check the current depth limit (line 25). If it is greater than zero, the **exec** method calls itself recursively with the depth limit decremented by 1 (line 28). Otherwise, the result of the method call is associated with a fresh unknown value (line 35).

Whenever the tool encounters a branch instruction (lines 8-13), it makes two copies of the current state object, adds the branch condition to the path condition of one copy, and adds the negated branch condition to the other copy. It then queries the decision procedure for each of the two new states to determine whether they represent feasible paths. For each feasible path, the symbolic executor recursively invokes itself with the target instruction and associated symbolic state.

Ideally, pruning infeasible paths will reduce the false positive rate as well as execution time. Because we wanted to experiment with how much benefit we get from path pruning, we added the option to disable the decision procedure. In this case, **satisfiable** always returns true when checking branch conditions.

Before passing a path condition to the decision procedure, we do some simple and efficient checks to detect obvious contradictions, so that we can prune some paths without the expense of pipe-based communication with CVC-Lite. This simple analysis occurs even when the decision procedure is disabled.

We use two metrics for ensuring termination in the presence of arbitrary branches, denoted in the pseudo-code by the **terminate** call. The first metric keeps track of how many times each instruction has been encountered. If the number is above a certain instruction visitation threshold (3, by default), our tool stops processing the current path. The second metric keeps track of the number of conjuncts in the path condition, instead. A command-line flag selects which metric to use. A more sophisticated design might identify loops in the control flow graph, and only count jumps to the loop header to keep track of iteration counts. However, the metrics we use are simpler to implement, and seem sufficient.

```
01: SymbolicValue exec(insn, state, depth) {
02:   if(terminate(insn))
03:       return NoValue
04:   switch(insn.type) {
05:     case CondBranch:
06:       tstate = state.clone()
07:       tstate.addPC(insn.cond)
08:       fstate = state.clone()
09:       fstate.addPC(negate(insn.cond))
10:       if(satisfiable(tstate))
11:           exec(insn.ttarget, tstate)
12:       if(satisfiable(fstate))
13:           exec(insn.ftarget, fstate)
14:     case Div:
15:       estate = state.clone()
16:       estate.addPC(eqPred(insn.denom, 0))
17:       if(satisfiable(estate))
18:           handleWarning(estate)
19:     case VirtualCall:
20:       estate = state.clone()
21:       estate.addPC(
22:         eqPred(insn.thisObj, null))
23:       if(satisfiable(estate))
24:           handleWarning(estate)
25:       if(depth > 0) {
26:         cstate = state.clone()
27:         cstate.clearLocals()
28:         result = exec(
29:           insn.calledMethod.first,
30:           cstate, depth - 1)
31:         state.setLocal(
32:           insn.resultLoc,
33:           result)
34:       } else {
35:           state.setLocal(
36:             insn.resultLoc,
37:             new Unknown())
38:       }
39:     case ArrayRead:
40:       estate = state.clone()
41:       estate.addPC(
42:         geqPred(insn.index,
43:           arraySize(insn.base)))
44:       if(satisfiable(estate))
45:           handleWarning(estate)
46:       setLocal(
47:         insnt.resultLoc,
48:         arrayRef(insn.base, insn.idx))
49:     case Return:
50:       return insn.retVal
51:     case ...
52:   }
53: }
54:
55: void handleWarning(state) {
56:   test = solveConstraints(state)
57:   run(test)
58: }
```

Figure 2: Pseudo-Java code for the symbolic execution algorithm. This outline only shows how we handle some of the more interesting bytecode instructions, and is not meant to be complete.

4.1.2 Constraint Solving

For each state that the symbolic executor warns about, the constraint solving stage attempts to find concrete objects or primitive values that, when passed as the parameters or receiver object of the method under analysis, will cause the expected runtime exception (line 56). To solve constraints involving integer arithmetic, we use the POOC solver [27]. For reference equality and inequality constraints we use a union-find data structure to keep track of equivalence classes.

Solving for values of primitive types is fairly easy, given an existing constraint solver. Because object fields and array entries are either primitive values or objects themselves, we can handle them by decomposition.

4.1.3 Test Execution

The test execution stage begins when symbolic execution and constraint solving are completed for a particular method. This stage iterates through the set of solutions produced by the constraint solver and, for each one, attempts to generate appropriate objects for the receiver and parameters of a method call (line 57).

We generate these objects using the Java reflection API, which allows us to produce exactly the primitive values and arrays we desire. To construct objects, we simply invoke an arbitrary constructor in the appropriate class, and then attempt to set its fields to the values obtained during constraint solving (recursing, of course, if those values are objects themselves). We use arbitrary values for any parameters that are unconstrained.

This method of creating objects has a shortcoming. The connection between constructor parameters and a particular field of the resulting object is non-trivial, and may not even exist, hence the choice of an arbitrary constructor. However, it may also be the case that the fields mentioned in a solution are private. Therefore, it may be impossible to directly create the desired program state. This may occur because the state is, in fact, unrealizable. In the case that the state is realizable, however, it may only be reachable through an arbitrarily long sequence of method calls, which we do not attempt to recreate. If these difficulties prevent us from constructing a concrete state to test a given warning, we skip that warning. This may cause us to miss real errors that depend on subtle conditions.

The process just described occurs within the same virtual machine as the analysis, and test cases are generated and executed using reflection. On the other hand, JCrasher, for instance, creates external files containing JUnit test cases. In retrospect, JCrasher's approach seems more robust, and we plan to adopt it in the future.

4.2 Sources of Unsoundness and Incompleteness

Our implementation of symbolic execution has proven to be very effective, and easy to implement. Certain aspects of the design, however, introduce the possibility of both false positives and false negatives. That is, the symbolic executor sometimes warns about potential errors that cannot actually occur during execution of the method in question, or misses errors that may in fact occur. There are several factors contributing to this inaccuracy:

- When we don't follow a method call, there are two possible modes of operation. We can simply assume that the method returns an unknown value (if anything), and doesn't make any changes to global state. This behavior, the default, helps the analysis in cases where the assumption is true, but it may not be true (and definitely isn't in some cases). This can cause us to miss real errors, or to warn about exceptions that cannot actually occur. An alternative is to assume that the method may make arbitrary changes to global state, meaning that, after the call, nothing is known about global state. We can turn this behavior on with a command-line flag, but it also introduces the possibility of both false positives and false negatives.

- Our tool does not have significant support for concurrency. Concurrently executing methods may behave differently from our model, since changes to thread-shared data may occur at any time.

- Our tool does not reason well about floating point numbers or bit-level operations.

- We treat the response "Unknown" from the decision procedure as indicating that a path might be feasible, and we explore it, even though it may actually be infeasible. CVC-Lite will respond with "Unknown" in cases where it knows its analysis to be incomplete, such as when given a query involving multiplication. We also have queries set to abort if they take too long to solve (an upper limit of 20 seconds for the experiments in this paper), and we treat the case of timeouts as if they were "Unknown" responses.

- Return values from unanalyzed methods are considered to be totally unknown, and thus more paths are possible than would be if we knew exactly what the method could return in the given context. Of course, interprocedural execution reduces this problem, but there is always some code we do not analyze, such as that in native methods.

- Access restrictions (such as those enforced by visibility modifiers) may make it so that we cannot construct a test case to reach a given symbolic state, as described in Section 4.1.3. This may happen because the state is actually infeasible (since other code cannot violate the access restrictions, either), or because the program follows the common pattern of making most or all fields private, and setting them within constructors. Because the link between constructor parameters and fields is non-trivial, we may not be able to initialize fields as desired. Because we cannot create test cases for these instances, we skip them and thus do not report them as potential errors.

5. EXPERIMENTAL RESULTS

5.1 Test Subjects

We evaluated our tool on a number of small programs with known bugs that have been used to evaluate previous defect detection tools, along with a collection of larger programs which are stable enough to be in widespread use, but inevitably contain bugs as well.

	Source Lines	Classes	Methods	Known errors
s1	503	1	17	4,1
s1139	462	1	16	3,0
s2120	383	1	17	4,1
s3426	439	1	19	8,0
s8007	376	1	16	1,0
bst	346	2	34	4,0
self	8136	100	510	N/A
cup	11048	37	280	N/A
javafe	48170	229	2017	N/A
cream	3560	33	174	N/A
jpf	38538	382	2458	N/A

Figure 3: The sizes and known error counts for each of our test subjects.

5.1.1 *Small Programs.*

The small programs were originally presented by Christoph Csallner, *et al.*, to evaluate JCrasher [7] and Check'n'Crash [8], and have a set of known errors. The programs consist of:

- Several instances of the P1 program: the responses of various students to a homework assignment, each of which contains at least one bug. These are shown in the results tables by student number: s1, s1139, s2120, s3426, and s8007.

- A binary search tree implementation: bst.

5.1.2 *Large Programs.*

Our collection of larger code bases consists mainly of programs or libraries used in program analysis. These programs inevitably contain bugs, but the exact locations and numbers of true bugs are unknown.

- Self-analysis of the tool's own code.

- CUP, a LALR parser generator for Java (version 0.10k).

- Javafe, a Java parser and type checker, used as part of the ESC/Java project (from ESC/Java2 version 2.0a9).

- Cream, a constraint solver (version 1.2).

- JPF, an explicit-state model checker for Java programs developed at NASA Ames Research Center (version 3.1.2).

The sizes and known error counts (when available) for each of our test subjects are show in Figure 3. The fourth column gives a pair of numbers: the first is the number of null-pointer dereferences, and the second is the number of more complex errors.

5.2 Experimental Setup

We analyzed all of the test programs in a variety of configurations. Each configuration included all of the methods in each program, with a global call depth of 0, 1, or 2. Furthermore we varied the path termination condition: the maximum path condition size was set to 5, 10, 15, 20 and 25 and the instruction revisitation bound was set to 3, 5 and 10. Since the results for the different path termination conditions varied only in very few cases we only present a few representative examples here.

Depth 0

Test	Time	Queries	Pruned	Warnings	Crashes
s1	0m24s	309	7.76%	75	6
s1139	0m5s	351	7.69%	16	8
s2120	0m3s	240	2.08%	11	5
s3426	0m30s	1228	5.04%	35	11
s8007	0m3s	196	1.53%	4	2
bst	0m2s	82	0.00%	15	8

Depth 1

Test	Time	Queries	Pruned	Warnings	Crashes
s1	0m5s	280	15.71%	20	6
s1139	0m6s	600	19.83%	14	8
s2120	0m5s	406	10.34%	11	5
s3426	0m32s	1770	13.55%	35	11
s8007	0m3s	260	11.15%	4	2
bst	0m5s	246	7.31%	13	8

Depth 2

Test	Time	Queries	Pruned	Warnings	Crashes
s1	0m5s	320	18.75%	20	6
s1139	0m7s	670	21.64%	14	8
s2120	0m5s	468	12.39%	11	5
s3426	0m32s	1810	14.14%	35	11
s8007	0m3s	304	15.13%	4	2
bst	0m17s	1084	19.92%	13	8

Figure 4: Analysis results for small programs with Maximum PC Size set to 15

Depth 0

Test	Time	Queries	Pruned	Warnings	Crashes
s1	0m19s	209	10.04%	75	6
s1139	0m6s	250	9.20%	16	8
s2120	0m3s	152	3.28%	12	5
s3426	0m25s	256	7.03%	38	11
s8007	0m2s	124	2.41%	4	2
bst	0m2s	82	0.00%	15	8

Depth 1

Test	Time	Queries	Pruned	Warnings	Crashes
s1	0m28s	517	38.68%	68	5
s1139	0m7s	500	14.60%	14	8
s2120	0m7s	378	9.25%	12	5
s3426	1m0s	798	27.19%	33	11
s8007	0m2s	200	17.50%	4	2
bst	0m5s	246	7.31%	13	8

Depth 2

Test	Time	Queries	Pruned	Warnings	Crashes
s1	0m31s	621	40.25%	68	5
s1139	0m8s	570	17.36%	14	8
s2120	0m8s	464	11.85%	12	5
s3426	0m32s	775	28.25%	30	11
s8007	0m3s	256	22.26%	4	2
bst	0m13s	858	18.76%	13	8

Figure 5: Analysis results for small programs with Maximum Instruction Revisits set to 5

In addition, we experimented with disabling the decision procedure at call depth 0 (purely intraprocedural). The results of our experiments for the small programs appear in Figures 4 and 5, while the results for the larger programs appear in Figures 8 and 9. All experiments were run on a dual-processor 2.66 GHz Pentium running RedHat Enterprise Linux 4 with the JVM given 2GB of memory.

In these tables, the "Queries" entries indicate the number of queries posed to the decision procedure to test path satisfiability, and the "Pruned" entries indicate the percentage of these queries that were found to be unsatisfiable and were not explored further. The "Warnings" entries indicate how many unique possible crashes the symbolic executor discovered, and the "Crashes" entries indicate how many of the possible unique crashes actually occurred when tested.

5.3 Observations

5.3.1 Small Programs

We analyzed the small programs to determine how our technique compared to Check'n'Crash. For these examples we could also study each of the reported warnings and errors to determine how the tool performs when we vary its configuration.

Comparison with Check'n'Crash. We compared our results to those given on the Check'n'Crash website[1]. We found every error discovered by Check'n'Crash that was in an error class supported by our tool, along with a few more:

s1 We found the same ArrayIndexOutOfBoundsException instances and three additional NullPointerException instances.

s1139 We found the same ArithmeticException (due to division by zero) and NegativeArraySizeException, but also an additional ArrayIndexOutOfBoundsException and three additional NullPointerException instances.

s2120 We found the same ArrayIndexOutOfBoundsException instances and three additional NullPointerException instances.

s3426 We found the same ArrayIndexOutOfBoundsException and NegativeArraySizeException instances, but missed the NumberFormatException. We found eight additional NullPointerException instances.

s8007 We found the same NegativeArraySizeException instance and an additional NullPointerException.

bst We found the same two ClassCastException instances and two additional NullPointerException instances.

We were surprised that Check'n'Crash didn't find any of the instances of NullPointerException that we found; either they were suppressed in the reporting or some algorithmic weakness must prevent them from being discovered. We don't currently check for NumberFormatException, so we missed one of the errors caught by Check'n'Crash.

Call Depth. As can be seen from the results, the level of interprocedural analysis did not affect the number of errors

[1]`http://www-static.cc.gatech.edu/grads/c/csallnch/cnc/`

```
int target = ...;
int delta = ...;
foo(int i) {
  if (similar(i,target)) {
     y = 10 / i;
  }
}

boolean similar (int i, int target) {
  if (((target - delta) <= i) &&
       (target + delta) >= i)
    return true;
  return false;
}
```

Figure 6: An example where intraprocedural analysis is sufficient.

```
foo(int m) {
  m = answer(m);
  m = m / (1 - m);
}

int answer(int v) {
  return v == 42 ? 1: 0;
}
```

Figure 7: An example where interprocedural analysis is required.

discovered. However, in some cases, most notably for s1, there was a decrease in warnings going from intraprocedural analysis to interprocedural analysis at depth 1. The code in Figure 6 illustrates the reason for this behavior. Note that depending on the value of *target* and *delta* there could be a division by zero in this code. Assume we pick $target = 100$ and $delta = 10$; in this case we can never have a division by zero.

During an intraprocedural analysis we will get one warning, but no error (since the warning will be for when $i = 0$ and that would make the division unreachable). The reason for this is that the call to *similar* is ignored and a fresh symbolic variable is created to hold the result of the call.

During an interprocedural analysis, however, we will not even get the warning, since the constraints in *similar*, combined with the fact that i should be 0, will be infeasible. The interesting case here is if we pick the values to expose the problem (e.g. change *target* to 1). Now both an intra- and interprocedural analysis expose the error! Note that an intraprocedural analysis also finds the problem simply because we made the statement reachable (by picking *target* and *delta* to expose the problem); thus adding the constraint that i should be 0 to have a possible division by zero is enough to actually find the error.

One can also create an example to show the opposite effect, where obtaining additional constraints exposes errors that would otherwise not have been found. This happens when analyzing the code in Figure 7. Here, an intraprocedural analysis has no additional constraints on the input value m, and the chance that the test generator will randomly pick 42 is almost zero. However, during an interprocedural analysis the constraint that m should be 42 will be recorded, so generating a test case that exposes the error is trivial.

It is easy to see that in general a statement that is potentially buggy can be reached in many more ways that don't expose the error than in ways that do expose the error — if this is not true then the error will probably be found and fixed quickly. In general, the additional constraints obtained by doing an interprocedural analysis serve mostly to reduce the number of (globally) infeasible paths explored, thereby reducing the number of warnings generated, but not affecting the number of errors discovered.

As expected, if the tree of explored paths gets pruned due to additional information from an interprocedural analysis, the execution time is also reduced. This is visible in the results for s1 in Figure 4.

Path Condition Size. Picking a small path condition limit can influence the number of errors detected. In the case of s1 and s2120 (requiring more than 5 constraints in the path condition) and s1139 (requiring more than 10) the minimum value of 5 constraints was not enough to expose all the known errors. When using the number of revisits to an instruction to terminate the search, the minimum number of 3 revisits is always enough the find the known errors. Even for just 3 revisits the path condition can grow quite big. We introduced this termination condition since it allowed us to get a sampling of paths that contain large path conditions.

For example in the case of s1, picking 3 revisits can lead to path conditions of size 34 (leading to a warning), and when considering 10 revisits we see path conditions of size 83 (leading to a warning). Of course, it also has a drawback that one can see in Figure 5 when doing an interprocedural analysis: some errors will never be found (see the depth 1 and 2 cases for s1). This happens since we keep the instruction re-visitation count across multiple paths and thus can prune a path after an instruction was revisited n times on another (uninteresting) path. This is a weakness of our current implementation and we plan to make the instruction re-visitation count path-sensitive in the future.

Interestingly, in an intraprocedural analysis we can go as far as completely eliminating revisits (setting the maximum allowed revisits to zero) and all the known errors in these small examples can still be exposed. This seems to indicate that the errors in these examples are quite simple. However, the number of randomly generated inputs required to uncover the known bugs reported in the JCrasher paper[8] was much higher than the number of tests we required to find the same bugs.

Although not visible in the results presented here, if we increase the maximum allowed path condition, we then see an increase in queries and warnings, and thus in runtime, as we expected.

Pruning Infeasible Paths. We wanted to know how many paths were actually pruned by the infeasibility check. To the best of our knowledge this has not been studied before. We found that, in most cases, the larger the allowed path condition, the larger the percentage of pruned paths. This seems intuitive: adding more constraints increases the number of ways a contradiction may occur. By the same token, the deeper the interprocedural analysis goes, the more paths get pruned (see Figures 4 and 5). Note that we prune very few paths during the intraprocedural analysis (on average less than 10%), due to the lack of constraints from following interprocedural calls. For depth 1 we get an aver-

Depth 0

Test	Time	Queries	Pruned	Warnings	Crashes
self	4m19s	4285	8.40%	734	71
cream	2m7s	55682	49.51%	170	18
CUP	16m16s	5430	8.28%	381	10
javafe	25m5s	123806	43.54%	1372	627
jpf	20m40s	42456	20.97%	2299	538

Depth 1

Test	Time	Queries	Pruned	Warnings	Crashes
self	32m47s	15130	22.46%	792	77
cream	7m32s	163963	49.67%	207	18
CUP	19m57s	32740	3.27%	413	8
javafe	59m23s	152481	39.50%	1487	615
jpf	137m16s	84512	23.49%	2804	538

Depth 2

Test	Time	Queries	Pruned	Warnings	Crashes
self	40m24s	21243	21.88%	768	76
cream	15m47s	243594	50.93%	211	18
CUP	17m32s	21270	9.53%	390	8
javafe	114m39s	87783	27.05%	1419	616
jpf(5)	178m56s	120865	39.53%	2421	539

Figure 8: Larger programs results with Maximum PC size set to 10 (5 for JPF).

Depth 0

Test	Time	Queries	Pruned	Warnings	Crashes
self	2m54s	3146	11.98%	803	74
cream	1m4s	638	9.09%	175	18
CUP	8m5s	2243	7.84%	396	10
javafe	9m49s	11788	10.72%	1466	610
jpf	14m58s	9517	10.44%	2359	541

Depth 1

Test	Time	Queries	Pruned	Warnings	Crashes
self	13m53s	12357	20.81%	851	80
cream	5m5s	2693	16.00%	221	18
CUP	18m22s	6370	17.04%	643	9
javafe	60m16s	48499	19.10%	1625	601
jpf	75m9s	44806	16.23%	2892	544

Depth 2

Test	Time	Queries	Pruned	Warnings	Crashes
self	91m58s	24310	25.18%	868	80
cream	9m13s	4569	20.85%	223	18
CUP	30m9s	10627	23.78%	612	10
javafe	166m42s	100105	22.66%	1577	603
jpf(3)	484m58s	86157	19.65%	2969	546

Figure 9: Larger programs results with Maximum Instruction revisits set to 5 (3 for JPF).

age closer to 16% and for depth 2 we observe an average of around 20% pruned paths.

Given the above result, we wondered whether it was necessary to prune infeasible paths at all. To investigate this we switched off the infeasibility checks and thus moved the analysis burden onto the constraint solving stage (that is, we spent time trying to find solutions even in cases where none exist). In the intraprocedural case, we still found all known errors, sometimes even more quickly than before. This can easily be explained by observing how few paths were pruned anyway during the intraprocedural analysis. Of course there are many more warnings in this case, but the total execution time goes down. The time taken by extra constraint solving is less than the time taken by the omitted feasibility checks.

However, as we increase the level of interprocedural analysis and thus increase the exponential blow-up, disabling the decision procedure quickly becomes impractical. For example, analyzing the s1 case for at level 1 took 5 seconds to complete. Without the decision procedure it didn't complete within 10 minutes (at which point we manually terminated the search).

It is interesting that what seems like a modest increase in the percentage of pruned paths actually has a profound effect on the scalability of the analysis.

5.3.2 Larger Programs

For the larger programs analyzed we don't know the number of true errors and hence cannot say what fraction our tool discovers. We are therefore more interested in whether some of the observations from the small, controlled examples still hold. In addition we also look at some of the performance characteristics we measured: relative cost of each phase of our tool pipeline and the types of errors the tool most frequently finds. We ran the experiments for all the same configurations as for the smaller examples, and as before we only show some representative results (in Figures 8 and 9) and discuss the rest within the text.

Call Depth. In the small examples we saw a decrease in warnings between intraprocedural and one level of interprocedural analysis. We now see something similar, but between level 1 and level 2. For some programs we also see a small increase in the errors found between the intraprocedural analysis and level 1. This seems to indicate that for these more complex programs we need to do at least a level 1 interprocedural analysis. We conjecture the reason for this is that the coding style for object-oriented languages often dictates the use of accessor methods (rather than just getting and setting fields directly) and thus to reason about such programs we need to look at least one level deep. For others we see a decrease in errors found during interprocedural analysis, but this is because our implementation can't always turn warnings into test cases. See Section 6.1 for more details.

Path Condition Size. Since we don't know what all the errors are here, it is hard to judge how much influence the size of the path condition has. We do see a small decrease in the number of errors found when choosing a path condition size bound of 5, but in all cases the number of errors seen is within 2% of the largest number observed — for Cream and CUP we actually find more errors with the smaller path condition. The runtime increase for larger path condition sizes is, however, quite dramatic. Note, for example, that the level 2 analysis of JPF took too long and we only report smaller configurations in the results (path condition bound of 5 instead of 10 and a bound of 3 revisits instead of 5).

Pruning Infeasible Paths. In contrast with the small examples, we now see much higher path pruning percentages, even for intraprocedural analysis. This is likely because the methods are now much bigger and thus more likely to contain contradicting branch conditions than in the small examples. Switching the feasibility check off now causes the search to fail (by running out of memory) for JPF and Javafe during intraprocedural analysis, and the analysis takes at least an order of magnitude longer for the other programs.

Running Times. In order to judge the expense of each of the three stages in our pipeline, we measured time spent doing feasibility checks via CVC-lite, how long it took to generate the tests, and how long it took to execute the tests. We conjectured beforehand that the decision procedure time would dominate. On average this turned out to be the case. However there were a few exceptions as well: for Cream, Javafe and JPF a number of analyses had constraint solving take about 40% of the total time with decision procedure time taking almost all the rest; for CUP it turned out that running the tests dominated the time (on average about 70% of total time) with the rest spent in the decision procedure. These results are clearly very application dependent.

Error Classes. We measured the percentage of errors reported belonging to each of the supported error classes. We found more instances of NullPointerException than ArrayIndexOutOfBoundsException, AssertionError, ClassCastException and NegativeArraySizeException. The analysis performed on the system itself was one of the exceptions, where we found about as many instances of AssertionError as NullPointerException (with a very small percentage of ClassCastException thrown in). As it turns out this was the only program that contained any assertion checks. For Cream the split was about 90% NullPointerException and the rest ArrayIndexOutOfBoundsException. CUP was the only true exception to our rule, where ArrayIndexOutOfBoundsException accounted for about 65% of the errors and NullPointerException the rest. We inspected the code and realized that the developers explicitly added checks for *null* parameters in many of the methods, thus reducing the number of these errors found. For Javafe we saw about 12% of the errors being ArrayIndexOutOfBoundsException with less than 2% ClassCastException and the rest being NullPointerException. Finally, for JPF we saw 85% NullPointerException, 8% ArrayIndexOutOfBoundsException, 6% ClassCastException and less than 1% NegativeArraySizeException.

Code Quality Prediction. Out of the examples we looked at only CUP can be considered a mature product, since it has been available for a number of years. Interestingly this is also the program in which we found the fewest errors — only about 1 for every 1,000 LOC (KLOC). This is quite low by commonly accepted standards (which suggests more like 4 per KLOC). For the academic programs (i.e. all the other larger examples) we discovered between 5 and 12 errors per KLOC. This seems reasonable for research prototype level code. We conjecture that with some more calibration this tool could be used to predict code quality. Note that the number of warnings may not be a good indicator of the quality of the code. For example, we report approximately the same number of warnings per KLOC for Cream and CUP, but we find around 5 times more errors per KLOC for Cream. However, we could interpret these results as an indication that Cream has "shallower" defects.

6. OPTIMIZATIONS

Here we briefly discuss how our tool might be improved.

6.1 Test Generation

The main limitation of our current system lies with the generation of the tests, and in particular, the creation of objects for which private fields need to be set. One can observe from the results that sometimes the number of errors detected goes down between levels. This is due to the randomness in the selection of constructors to call when an object needs to be generated. Currently we always first look for a constructor with no parameters, and, if one does not exist, we randomly choose between those that do exist. Unfortunately, we can pick one that takes another object as input which we may not be able to generate. We pick randomly since we have observed that just picking the first one, for instance, does not uncover as many errors as picking one at random. Of course this scheme should be changed to try all constructors until one is found that can be used to generate the object.That choice could then be recorded for future use.

```
foo(Node n1, Node n2) {
  if (n1 != null && n2 != null) {
    n1.x = 5;
    n2.x = 6;
    assert n1.x == 5 && n2.x == 6;
  }
}
```

Figure 10: An example demonstrating problems with aliasing.

6.2 Environment Generation

The tool sometimes reports a warning that actually indicates a real error, and although it is able to precisely create the required objects to expose the error, the test does not fail because it has complex interactions with the environment. An example (inspired by an actual case from the Next Generation Air Traffic System being developed at NASA) is when an out-of-bounds array access occurs when the number of files in a directory is used as an iteration bound for an array of a fixed size. When executing the test, however, the number of files in the directory might be less than the size of the array, and thus the test does not uncover the error.

Ideally, we could create a test harness under which all environment constraints can be encoded for use during the test. In the example above the test harness will know that the call to get the number of files in the directory should return a predefined value, found during the constraint solving phase. We hope to add such an environment generation feature for the generated tests in future work.

6.3 Aliasing

The tool will not be able to find the error in the code for Figure 10, since it doesn't consider that *n1* and *n2* could be aliased. Adding additional aliasing constraints would cause a blow-up in the constraint size, but would expose errors like these. We believe an experimental study is required to see whether adding such constraints is worthwhile, that is, whether they would expose new errors or just reduce the scalability of the tool.

7. RELATED WORK

Symbolic execution has a long history, and was used for debugging and testing as early as the 1970s [20, 2]. Later work applied similar techniques to Pascal [18], C [26], Fortran [13], and Ada [10], and Zhang explored symbolic execution in the context of pointers and structured data [33]. Use of symbolic execution to generate unit tests fully automatically goes back at least as far as Korel's 1996 paper [22], and has been applied to object-oriented unit tests in the Symstra system [31]. More recently, Engler *et al.* have explored symbolic execution for detecting errors in systems programs, such as file systems [32, 3]. Java Pathfinder, an explicit state model checker, has been extended to support symbolic execution [19].

Our work differs from these primarily in its flexible treatment of procedure calls. To our knowledge, all of the previous work has either used intraprocedural or whole-program analysis.

A similar approach is to do concolic testing [28, 14] where the program is executed with (random) concrete data and in parallel a symbolic execution collects all the constraints that define the input partition the data is from. One of these constraints is then negated, the resulting condition solved and used to rerun the code. This approach allows a systematic analysis of the paths of the program up to some bounds. This differs from our approach since this is a dynamic analysis that is driven by symbolic execution, whereas ours is a static analysis (with symbolic execution) that uses dynamic analysis to refine the results.

Significant research has also gone into automated test generation using methods other than symbolic execution. For example, Cristoph Csallner, *et al.*, have developed a trio of tools for automated test generation in Java. The first, JCrasher [7], generates random JUnit tests for all of the public methods in a given set of classes, using any available constructors or static methods returning the desired type to generate receiver objects and method parameters.

Check'n'Crash [8] refines JCrasher by first passing the program through ESC/Java [12] to find potential errors, and then, as in our work, using a constraint solver to attempt to generate tests to trigger each reported bug, if it exists. The primary difference between our work and Check'n'Crash is that we explore variably interprocedural symbolic execution, whereas ESC/Java is a purely intraprocedural analysis tool.

Finally, DSD [9] further extends Check'n'Crash by integrating Daikon [11], a runtime analysis tool aimed at inferring program invariants. The invariants inferred by Daikon are fed to ESC/Java to improve its analysis. The same idea could be applied to our tool, and is orthogonal to our improvements.

Outside of the realm of symbolic execution and automated test generation, a wide range of other tools have been developed to find bugs in programs. Some of the earliest examples were simple tools such as Lint [17], and the more recent incarnation, Splint [23], which look for suspicious patterns of code in C programs. Similar tools also exist for Java, such as PMD [4] and FindBugs [16]. ESC/Java [12], mentioned earlier, takes a somewhat more rigorous approach, using an axiomatic semantics for Java along with an automated theorem prover to check Java programs against provided specifications. There are also commercial tools such as Coverity [6] and KlocWork [21] that are being used frequently in industry and that also detect errors similar to those our approach can uncover.

The systems in this final set differ from ours in that they only suggest possible bugs, while our tool generates test cases from each warning and runs them to see if crashes actually occur. On the other hand, many of these tools can look for bugs that do not cause the program to crash, while ours only looks for exception-inducing bugs.

8. CONCLUSION

We have presented a technique and accompanying tool that uses symbolic execution of Java code, with variable degrees of interprocedural context, to find possible runtime errors. To reduce false positives, it uses the constraints contained in the state associated with each warning to generate a test that will show the error during execution. We have evaluated the tool on known benchmarks, where it found all the known errors along with a few that were previously unknown. We also evaluated our tool on larger, publicly-available systems. Among these larger examples, we found many errors in research prototypes and considerably fewer in the one more mature system.

Although the main motivation for our work was to reduce the large number of false positives generated by a purely static analysis, and to give better feedback about possible errors, we believe the additional measurements turned out to be equally useful in understanding the power of symbolic execution and interprocedural analysis.

Our aim was to look for "simple" runtime errors (which are the bread and butter of the very successful commercial tools such as Coverity and KlocWork). The results indicate that doing an intraprocedural analysis even for modest path condition sizes (or a small limit on instruction re-visitation) seems to give more than adequate results. We found that doing a more deeply interprocedural analysis reduced the number of warnings but not the number of errors. However, in most cases, this slightly increased precision comes along with a large increase in running time. Using the path condition size as a termination condition has a more immediate effect on the number of errors discovered, but again larger values increase the runtime, and smaller values still seem sufficient to find most errors.

However, our experiments only considered simple runtime exceptions, which for the most part are mechanical errors and not application-specific. Looking for more complex, application-specific errors may require deeper interprocedural analysis. It is unclear whether symbolic execution would be sufficient for these more complex errors, since the executions times in our experiments started to become prohibitively large at 2 levels of interprocedural analysis.

9. REFERENCES

[1] C. Barrett and S. Berezin. CVC Lite: A new implementation of the cooperating validity checker. In R. Alur and D. A. Peled, editors, *Proceedings of the International Conference on Computer Aided Verification*, volume 3114 of *Lecture Notes in Computer Science*, pages 515–518. Springer-Verlag, July 2004.

[2] R. S. Boyer, B. Elspas, and K. N. Levitt. Select — a formal system for testing and debugging programs by symbolic execution. *ACM SIGPLAN Notices*, 10(6):234–245, 1975.

[3] C. Cadar and D. Engler. Execution generated test cases: how to make systems code crash itself. In *Proceedings of the International SPIN Workshop on Model Checking of Software*, 2005.

[4] T. Copeland. *PMD Applied*. Centennial Books, Nov. 2005.

[5] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Min, D. Monniaux, and X. Rival. The ASTRÉE analyser.

In *Proceedings of the European Symposium on Programming*, pages 21–30, 2005.

[6] Coverity, Inc. `http://www.coverity.com`.

[7] C. Csallner and Y. Smaragdakis. JCrasher: An automatic robustness tester for Java. *Software—Practice & Experience*, 34(11):1025–1050, Sept. 2004.

[8] C. Csallner and Y. Smaragdakis. Check 'n' Crash: Combining static checking and testing. In *Proceedings of the International Conference on Software Engineering*, pages 422–431, May 2005.

[9] C. Csallner and Y. Smaragdakis. DSD-Crasher: A hybrid analysis tool for bug finding. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 245–254, July 2006.

[10] L. K. Dillon. Using symbolic execution for verification of Ada tasking programs. *ACM Trans. Prog. Lang. Syst.*, 12(4):643–669, 1990.

[11] M. D. Ernst, A. Czeisler, W. G. Griswold, and D. Notkin. Quickly detecting relevant program invariants. In *Proceedings of the International Conference on Software Engineering*, pages 449–458, June 2000.

[12] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 193–205, 2002.

[13] M. R. Girgis. An experimental evaluation of symbolic execution systems. *IEEE Software Engineering Journal*, 7(4):285–290, 1992.

[14] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 213–223. ACM, 2005.

[15] G. J. Holzmann. *The Spin Model Checker.* Addison-Wesley, 2003.

[16] D. Hovemeyer and W. Pugh. Finding bugs is easy. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2004.

[17] S. C. Johnson. *Lint, a C Program Checker.* Bell Labs, 1978.

[18] R. A. Kemmerer and S. T. Eckman. UNISEX: A UNIX-based symbolic executor for Pascal. *Software — Practice and Experience*, 15(5):439–458, 1985.

[19] S. Khurshid, C. S. Pasareanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2003.

[20] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[21] Klocwork, Inc. `http://www.klocwork.com`.

[22] B. Korel. Automated test data generation for programs with procedures. In *Proceedings of the International Symposium on Software Testing and Analysis*, 1996.

[23] D. Larochelle and D. Evans. Statically detecting likely buffer overflow vulnerabilities. In *Proceedings of the 2001 USENIX Security Symposium*, Aug. 2001.

[24] National Institute of Standards and Technology. The economic impacts of inadequate infrastructure for software testing, May 2002.

[25] Robby, M. B. Dwyer, and J. Hatcliff. Bogor: An extensible and highly-modular model checking framework. In *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 2003.

[26] T. Sakaguchi. *UNISEX-C : a UNIx-based Symbolic EXecutor for standard C.* PhD thesis, University of California, Santa Barbara, 1997.

[27] H. Schlenker and G. Ringwelski. POOC: A platform for object-oriented constraint programming, 2002.

[28] K. Sen, D. Marinov, and G. Agha. "CUTE: A Concolic Unit Testing Engine for C". In *Proceedings of ESEC/SIGSOFT FSE'05*, 2005.

[29] R. Vallée-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co. Soot - a Java optimization framework. In *Proceedings of CASCON 1999*, pages 125–135, 1999.

[30] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engineering Journal*, 10(2), Apr. 2003.

[31] T. Xie, D. Marinov, W. Schulte, and D. Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 3440:365–381, 2005.

[32] J. Yang, C. Sar, P. Twohey, C. Cadar, and D. Engler. Automatically generating malicious disks using symbolic execution. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2006.

[33] J. Zhang. Symbolic execution of program paths involving pointer and structure variables. In *Proceedings of the International Conference on Quality Software*, 2004.