



# Variably Interprocedural Program Analysis for Runtime Error Detection

---

Aaron Tomb, Guillaume Brat, and Willem Visser  
Presented by Mark Smith



# Contents

---

- Introduction
- Example - Variably Interprocedural Analysis
- Implementation
- Experimental Results
- Optimizations
- Related Work
- Conclusion



# Introduction

---

- As software becomes more complex, the ability to uncover defects becomes more and more critical
- Software flaws
  - estimated to cost the United States economy alone tens of billions of dollars annually.
- In response, many approaches to automated defect detection have been explored.



# Introduction

---

- This paper describes an analysis approach combining static and dynamic techniques to find run-time errors in Java code.
  - Employs symbolic execution to find constraints under which an error (e.g., a null pointer dereference, array out of bounds access) may occur
  - Solves constraints to find test inputs for exposing the error.
  - user alerted when it detects the expected exception



# Variably Interprocedural Analysis- Example

---

```
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: }
```



# Implementation

---

- The tool applies their technique to the detection of unhandled runtime exceptions in Java bytecode.
- Builds on the Soot framework
  - Soot provides a large body of static analysis code that is useful for defect detection.
- Tool consists of approx. 8,000 lines of Java code



# Implementation

---

## Architecture

*Three-stage pipeline.*

- first stage (Symbolic Execution)
  - performs variably interprocedural symbolic execution
  - generates a set of symbolic representations of programs states which may result in a runtime exception.
  
- second stage (Constraint Solving)
  - solves constraints imposed by these symbolic states to obtain concrete method parameter values for testing.
  
- third stage (Test Execution)
  - uses the Java reflection API to invoke the methods under analysis, with the parameters obtained in second stage.



# Implementation

---

## Symbolic Execution

- Tool begins symbolic execution by analyzing a specified set of methods, instruction by instruction.
- See pseudo-code





# Implementation

---

```
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:     }
```



# Implementation

---

## Constraint Solving

- 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.
  
- For solving constraints involving integer arithmetic:
  - POOC solver.
  
- For reference equality and inequality constraints:
  - use a union-find data structure to keep track of equivalence classes.



# Implementation

---

## Test Execution

- This stage begins when symbolic execution and constraint solving are completed for a particular method.
- 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
- Objects are generated using the Java reflection API (can pose a shortcoming)



# Implementation

---

## Sources of Unsoundness/Incompleteness

- Certain aspects of the design introduce the possibility of both false positives and false negatives.
- When a method call is not followed, two possible modes of operation:
  - Assume the method returns an unknown value and doesn't make any changes to global state.
  - Assume that the method may make arbitrary changes to global state, meaning that, after the call, nothing is known about global state.



# Implementation

---

## Sources of Unsoundness/Incompleteness (cont.)

- ❑ Tool does not have significant support for concurrency.
- ❑ Tool does not reason well about floating point numbers or bit-level operations.
- ❑ Access restrictions may make it so that we cannot construct a test case to reach a given symbolic state



# Experimental Results

---

## Test Subjects

- Small programs
  - Programs of various students for a homework assignment (contain bugs)
  - A binary search tree implementation: bst.
- Large programs
  - Programs, libraries (contain bugs; exact location and # unknown)
  - Self-analysis of the tool's own code.
  - CUP, Javafe (ESC/Java project), Cream, JPF (NASA)

# Experimental Results

---

## Test Subjects

	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



# Experimental Results

---

## Observations

- Small programs
  - analyzed the small programs to determine how the technique compared to Check'n'Crash.
  - found every error discovered by Check'n'Crash that was in an error class supported by the tool
  - See tables of analysis



# Experimental Results

---

## Observations

- Small programs
  - Call Depth
    - the level of interprocedural analysis did not affect the number of errors discovered.
  - Path Condition Size
    - Picking a small path condition limit can influence the number of errors detected.
    - When using the number of revisits to an instruction to terminate the search, the minimum number of 3 revisits is enough to find the known errors



# Experimental Results

---

## Observations

- Small programs (cont.)
  - Pruning infeasible paths
    - in most cases, the larger the allowed path condition, the larger the percentage of pruned paths.
      - This is intuitive: adding more constraints increases the number of ways a contradiction may occur.
    - Likewise, the deeper the interprocedural analysis goes, the more paths get pruned



# Experimental Results

---

## Observations

- Large programs
  - more interested in whether some of the observations from the small, controlled examples still hold.
  - Focused on some of the performance characteristics measured:
    - relative cost of each phase of our tool pipeline
    - the types of errors the tool most frequently finds



# Experimental Results

---

## Observations

- Large programs
  - Call Depth
    - (similar to small examples) Decrease in warnings between intraprocedural and one level of interprocedural analysis.
  - Path Condition Size
    - a small decrease in the number of errors found when choosing a path condition size bound of 5
    - runtime increase for larger path condition sizes is quite dramatic.



# Experimental Results

---

## Observations

- Large programs (cont.)
  - Pruning infeasible paths
    - In contrast with the small examples, now much higher path pruning percentages, even for intraprocedural analysis.



# Experimental Results

---

## Observations

- Large programs (cont.)
  - Running Times
    - measured:
      - time spent doing feasibility checks via CVC-lite
      - time spent generating the tests
      - timer spent to execute the tests.
    - (Correctly) Determined beforehand that the decision procedure time would dominate



# Optimizations

---

## Test Generation

- 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
- 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.



# Optimizations

---

## Environment Generation

- tool sometimes reports a warning that indicates a real error, and although able to precisely create the required objects to expose the error, the test does not fail
  - Due to complex interactions with the environment.
  
- could create a test harness under which all environment constraints can be encoded for use during the test.
  - 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.



# Optimizations

---

## Aliasing

```
1: foo(Node n1, Node n2) {  
2:   if (n1 != null && n2 != null) {  
3:     n1.x = 5;  
4:     n2.x = 6;  
5:     assert n1.x == 5 && n2.x == 6;  
6:   }  
7: }
```

- The tool will not be able to find the error
  - 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.



# Related Work

---

- Symbolic execution has a long history (1970's)
- Java Pathfinder, an explicit state model checker
  - extended to support symbolic execution
- Similar Approach - concolic testing
  - 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



# Related Work

---

- automated test generation using methods other than symbolic execution.
  - JCrasher, Check'n'Crash, DSD
- Other tools (outside realm of symbolic execution) explicit state model checker
  - Lint (C), Splint (C), FindBugs (Java), PMD (Java)



# Conclusion

---

- Paper presented a technique and accompanying tool that uses symbolic execution of Java code, with variable degrees of interprocedural context, to find possible runtime errors.
- aim was to look for “simple” runtime errors
- results indicate
  - Doing an intraprocedural analysis even for modest path condition sizes seems to give more than adequate results.
  - Interprocedural analysis reduced the number of warnings but not the number of errors.
  - Using the path condition size as a termination condition has a more immediate effect on the number of errors discovered
    - but larger values increase the runtime
    - smaller values still seem sufficient to find most errors.



# Conclusion

---

- experiments only considered simple runtime exceptions (for the most part are mechanical errors and not application-specific).
- looking for more complex, application-specific errors may require deeper interprocedural analysis.

Questions?

---





# 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.
- [4] T. Copeland. PMD Applied. Centennial Books, Nov. 2005.
- [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.
- [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.
- [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.
- [20] J. C. King. Symbolic execution and program testing. Communications of the ACM, 19(7):385–394, 1976.
- [24] National Institute of Standards and Technology. The economic impacts of inadequate infrastructure for software testing, May 2002.
- [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. Vall’ee-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.