Incremental Symbolic Execution

Sarmad Makhdoom
2012-03-0019

Supervisors
Dr Junaid Haroon Siddiqui
Dr Arif Zaman
Introduction

- What is software testing?
  - It’s all about finding bugs
- Why do we need to test?
  - Bugs fixing cost is increasing
- How do we test?
  - Static vs Dynamic analysis

- Famous failures
  - Ariane 5 Explosion ($500m) [http://www.ima.umn.edu/~arnold/disasters/ariane.html]
  - NASA Mars Climate Orbiter ($125m) [http://mars.jpl.nasa.gov/msp98/news/mco991110.html]
Outline

- Background
- Motivation
- Technique
- Evaluation
- Related Work
- Conclusion
Background
Concrete Execution

```c
int mid(int x, int y, int z) {
    if (x<y){
        if (y<z)
            return y;
        else{
            if (x<z)
                return z;
            else
                return x;
        }
    }else{
        if (x<z){
            return x;
        }else{
            if (y<z)
                return z;
            else
                return y;
        }
    }
}
```

Concrete test

- `x=1`
- `y=3`
- `z=2`
Symbolic Execution Demo

```c
int mid(int x, int y, int z) {
  if (x < y) {
    if (y < z) {
      return y;
    } else {
      if (x < z)
        return z;
      else
        return x;
    }  
  } else {
    if (x < z)
      return x;
    else {
      if (y < z)
        return z;
      else
        return y;
    }  
  }
}
```

Path condition (feasible) -> Solve

Concrete Test: x=1, y=3, z=2
Symbolic Execution [CACM’76]

- Concrete execution fixes input variables and exercises one path per input
- Symbolic execution uses symbols with no restrictions other than type
- Both branches of every condition in the program are explored
- A path condition is built for every path and contains the constraints required to take this path

“Symbolic Execution and Program Testing” – CACM 1976
Ranged Symbolic Execution [OOPSLA’12]

Scaling symbolic execution using ranged analysis. ACM-OOPSLA 2012
Ranged Symbolic Execution [OOPSLA’12]

Scaling symbolic execution using ranged analysis. ACM-OOPSLA 2012
KLEE [OSDI’08]

• An open source symbolic executor based on LLVM
  ▫ Generate high coverage test cases
• LLVM (Low Level Virtual Machine)
  ▫ Converts code into bytecode
• Constraint Solving
  ▫ STP (fast SMT solver)
Incremental Testing

Test Suite

Code v1.0 ➔ Modification ➔ Code v1.1
Incremental Testing

Test Suite

Modification

Code v1.0

Code v1.1
Incremental Testing

Test Suite

Modification

Code v1.0

Code v1.1
Incremental Testing

Test Suite

Code v1.0

Final Test Suite

Modification

Test Suite

Code v1.1
Motivation
Motivation

- All execution paths may be very large and may not be of interest
- Only execution paths that differ between two versions are of interest
- Test only program changes not the whole program
- Incremental symbolic execution is useful in bug finding and regression testing
Technique
Key Ideas

- Majority of search space is invalid
- Solving path conditions is expensive
- Comparing and validating of path conditions is cheap
- One way is to compare both CFGs [DiSE'11]
  - Static analysis
  - Inexact (in-depth node changes are problematic)
  - Scalability issues
Technique

- Full symbolic execution with initial version:
  - Generates inputs for each distinct path

- Incremental symbolic execution on subsequent versions
  - On exploration divide tests based on each branch condition
  - Compare and validate tests
    - If test is valid, don’t use solver
    - If test is invalid, explore the program for new states
Algorithm

**Input:** A finite set $TestSuite = \{t_1, t_2, \ldots, t_n\}$ of test cases

1. **for each** $b$ in $BranchCondition$ **do**
   2. $T_{true} \leftarrow \text{split}(TestSuite, b)$; $T_{false} \leftarrow \text{split}(TestSuite, \neg b)$
   3. $T_{invalid} \leftarrow TestSuite - (T_{true} \cup T_{false})$
   4. **if** $T_{invalid} \neq \emptyset$ **then**
      5. **if** $T_{true} = \emptyset$ **then**
         6. $\text{exploreAndSolve}(T_{true})$
         7. $\text{IncrementalExplore}(T_{false})$
      8. **if** $T_{false} = \emptyset$ **then**
         9. $\text{exploreAndSolve}(T_{false})$
         10. $\text{IncrementalExplore}(T_{true})$
   11. **else**
      12. $\text{IncrementalExplore}(T_{true})$
      13. $\text{IncrementalExplore}(T_{false})$

Algorithm to explore new ranges and not solving the present path conditions in the new program
Technique

Evaluate & Compare

\[ t_1, t_2, t_3, t_4, t_5, t_6, \ldots, t_n \]
Technique

t₁, t₂, t₃, t₄, t₅, t₆,..., tn

t₁, t₂, t₃, t₄

t₁, t₂, t₃

t₁, t₂

t₁, t₂

Evaluate & Compare
Technique

\( t_1, t_2, t_3, t_4, t_5, t_6, \ldots, t_n \)
Technique

$t_1, t_2, t_3, t_4, t_5, t_6, \ldots, t_n$

Invalidates
Technique

Evaluate & Compare

$t_1, t_2, t_3, t_4, t_5, t_6, \ldots, t_n$

$t_5, t_6, t_7, t_8$

$t_5, t_6, t_7$

$t_6, t_7$

$t_8$
Technique

$t_1, t_2, t_3, t_4, t_5, t_6, \ldots, t_n$

Range need to be explored
What if there is some infeasible path become feasible now?

"New code in two consecutive valid test cases"
Explanation

Change

Valid Test 1

Infeasible area

Valid Test 2
Evaluation
Evaluation

• Incremental testing on two different GNU Coreutils Suite
  ▫ Minor update release (v7.1 → v7.2)
  ▫ Major update release (v7.2 → v8.1)

• 85 stand-alone (i.e. excluding wrappers) apps
  ▫ File system management: ls, mkdir, chmod, etc.
  ▫ Management of system properties: hostname, printenv, etc.
  ▫ Text file processing : sort, wc, od, etc.
  ▫ ...

• Tests performed on Lonestar Linux cluster at TACC
  (http://tacc.utexas.edu/)
Results of Evaluation (v7.1 → v7.2 – Running Time)

71% Time Saved
Results of Evaluation (v7.1→v7.2 – Solver Time)

79% Time Saved
Results of Evaluation (v7.2 → v8.1 – Running Time)

73% Time Saved
Results of Evaluation (v7.2→v8.1 – Solver Time)

78% Time Saved
Directed Incremental Symbolic Execution [PLDI’11]

- Compute affected location by comparing CFGs
- Their technique is based on static analysis and reachability analysis
- Perform symbolic execution on only modified CFG

- Out technique perform dynamic analysis
- Generate test suite for whole new program not only for modified area
KATCH [FSE’13]

- Combines static and dynamic analysis for increased coverage
- Katch select tests from manual generated test suite
- Perform heuristics based dynamic analysis to increase chance of hitting modified code
- Execute modified code symbolically for test suite generation

- We don’t use static analysis or manual test suite for incremental testing
- Our technique can also explore in deeper depth in less time
Memoized Symbolic Execution [ISSTA’12]

- Cache based symbolic execution technique
- Store the results in trie-based data structure
- Re-use results from previous run by maintaining and updating trie
- Saving relies on position on change

- Our proposed technique based on ranged analysis is more effective
- Low cost of storing tests

Memoized Symbolic Execution. ISSTTA 2012
**Green**: Reducing, Reusing and Recycling Constraints [FSE’12]

- An interface between analyzer and solver
- Benefits
  - Reuse within an analysis run
  - Reuse Across Programs, Analyses, Solvers
- Procedure
  - Path Conditioning Slicing
  - Canonization
  - Storage
  - The Green Solver Interface

*Green: Reduce, reuse and recycle constraints in program analysis, in FSE. ACM, 2012*
Green: Reduce, reuse and recycle constraints in program analysis, in FSE. ACM, 2012
Conclusion
Conclusion

- Symbolic execution allows us to reason about multiple concrete executions
- Ranged symbolic execution allows dividing the problem of symbolic execution
- The proposed Incremental technique enables more *effective* and *efficient* testing of code using symbolic execution
  - Fully dynamic approach to find changes
  - Incremental testing in much less time
- Results show time saving for the same programs running incrementally on their new versions
Questions