Symbolic Execution, Dynamic Analysis

http://d3s.mff.cuni.cz

Department of Distributed and Dependable Systems

Pavel Parízek

CHARLES UNIVERSITY IN PRAGUE

faculty of mathematics and physics

Symbolic execution

Key concepts

- Symbolic values used for program variables
  - `v: x+2, x: i0+i1-3, y: 2*i1`
- Program inputs: variable names
- Other variables: functions over symbolic inputs
- Path condition (PaC)
  - Set of constraints over symbolic input values that hold in the current program state

Example program

```
input: x,y
 1: u = x - y;
 2: if (x > 1)
 3:    u = u + x;
 4: if (y > x)
 5:    u = y - x;
 6: assert (u >= 0);
```

Symbolic execution and program verification

- Symbolic program state
  - Symbolic values of program variables
  - Path condition (PaC)
  - Program counter (PC)
- Symbolic state = a set of concrete states
- Symbolic execution tree = state space
  - Tree of symbolic program states
  - Transitions labeled with the PC

Department of Distributed and Dependable Systems

Symbolic execution and program verification

- Path condition updated at each branching point in the program code
  - Different constraints added for each branch
  - Example: `if-else` with a boolean condition *C*
    - Formula *C* added for the `if` branch
    - Formula *not C* added for the `else` branch

- State space traversal
  - Satisfiability of *PaC* checked in each symbolic state
  - *PaC* == false → symbolic state not reachable
    - Verification tool backtracks and explores different branches

Department of Distributed and Dependable Systems D3S

Symbolic execution: possible applications

Do you have some ideas ?

Symbolic execution: possible applications

- Automatically generating test inputs
  - From path conditions in symbolic program states
- Find inputs that trigger a specific error
- Systematic testing of open systems
  - Examples: isolated procedures, components
  - Programs with unspecified concrete inputs
- Checking programs with inputs from unbounded domains (integers, floats, strings)

Symbolic execution: limitations

What are the limitations ?

Symbolic execution: limitations

- Handling loops with many iterations
- Stateless exploration (no state matching)
- Undecidable and complex path conditions
- State explosion (too many paths)
- Concurrent accesses from multiple threads

Department of Distributed and Dependable Systems

Loops with many iterations

```
x = input();
i = 1000;
while (true) {
   if (x > i) ...
   i--;
   ...
}
```

Loops: practical approach

- Unrolling loops to a specific depth
  - Limited number of loop iterations explored
- Exploring data structures up to a given bounded size

Concolic execution

Concolic execution

<u>con</u>crete + symbo<u>lic</u> = concolic

- How it works
  - Performs concrete execution on random inputs
  - Tracks symbolic values of program variables
  - Gathers constraints forming a path condition along the single executed path

Concolic execution: applications

- Dynamic test generation
  - Path condition for the single explored path defines corresponding test inputs
  - Negating constraints (clauses) for branching points
    - Find test inputs that drive program execution along different paths (control-flow)

KLEE: Symbolic Virtual Machine

- Symbolic execution tool for system code
  - Used to detected many real bugs in Linux/Unix core system utilities (`ls`, `chmod`, ...)
  - Models interaction with complex environment (files, networking, unix syscalls)
  - Highly optimized (performance, scalability)

- Built upon the LLVM compiler infrastructure

- Web: http://klee.github.io/
- Further information (recommended)
  - http://llvm.org/pubs/2008-12-OSDI-KLEE.pdf

PEX: White-box unit testing for .NET

- Dynamic test generation (concolic execution)
- Generates unit tests with high code coverage

- Availability
  - Visual Studio 2010 Power Tools, command-line

- Web sites
  - https://www.microsoft.com/en-us/research/project/pex-and-moles-isolation-and-white-box-unit-testing-for-net/

- Live demo: Code Digger
  - Visual Studio 2012 extension based on Pex

- IntelliTest extension for Visual Studio 2015
  - https://docs.microsoft.com/en-us/visualstudio/releasenotes/vs2015-rtm-vs#intellitest

SAGE: Scalable Automated Guided Execution

- Automated whitebox fuzz testing for security
- Systematic dynamic generation of unit tests

- How it works
  - concolic execution + solving negated conditions to infer new test inputs

- Main author: Patrice Godefroid
  - https://patricegodefroid.github.io/

- Further information (selected papers)
  - https://patricegodefroid.github.io/public_psfiles/ndss2008.pdf
  - https://patricegodefroid.github.io/public_psfiles/icse2013.pdf

Department of Distributed and Dependable Systems

Other tools

- https://playground.diffblue.com/
  - Automatically generating tests for Java code
  - Registration needed (or sign-in via FB/Google)

Dynamic analysis

Dynamic analysis

- Goal: analyze behavior of the program based on concrete execution of a single path

- Input: binary executable

Q: How can we get the data about one path?

Collecting information about single path

- Instrumentation
  - Target: binary executable, source code
- Runtime monitoring
  - manual inspection of huge log files
- Custom libraries

- Events
  - field accesses on shared heap objects
  - locking (acquisition, release, attempts)
  - procedure calls (e.g., user-defined list)

Benefits

- Precision
  - Complete information about program state
  - Recording only events that really happen

- Tool support
  - Errors: deadlocks, race conditions, atomicity
  - Languages: Java, C/C++, C# ?

Limitations

- Coverage
  - Single execution path
  - Few related paths
- Overhead
  - Compared with plain concrete execution
  - Range: 50 % - 1000 % (!)
  - Possible remedy: sampling

Department of Distributed and Dependable Systems

Selected tools

- Pin
  - Runtime binary instrumentation platform for Linux (32-bit x86, 64-bit x86, ARM)
  - Custom tools written in C/C++ using rich Pin API
  - Important features
    - efficient dynamic compilation (JIT)
    - process attaching, transparency
- Valgrind
  - Heavyweight dynamic binary instrumentation framework again for Linux (x86, PPC)
  - Tools: memory checker, thread checkers, some profilers
- RoadRunner

Department of Distributed and Dependable Systems

Applications

- Detecting bugs of all kinds
- Concolic execution
  - Adding new symbolic constraints into PaC
- Discovering likely invariants
- Predicting race conditions

Predicting data race conditions

- Algorithm
  1) Run dynamic analysis tool to record events about one particular execution trace
  2) Check the given trace for data race conditions
  3) If we find some errors, then stop immediately
  4) Generate feasible interleavings of events from the given single trace
  5) Check each generated interleaving for data races
  6) Report all detected possible races to the user

Predicting data race conditions

Q: How can we generate feasible interleavings ?

Generating feasible interleavings

- All possible interleavings of events from different threads

- Use the happens-before order between synchronization events
  - Conflicting field accesses not ordered → interleave

Discovering likely invariants

- Algorithm
  - Run the dynamic analysis tool several times (on selected inputs, test suite) to get a set of traces
  - Find properties over variables and data structures that hold for all/most traces in the set
  - Drop all inferred properties that do not satisfy additional tests (e.g., statistical relevance)
  - What remains are the likely invariants

Q: Looks good but there is a small catch

Department of Distributed and Dependable Systems

Discovering likely invariants

- Limitations
  - Precision depends on the test suite quality (inputs)
  - Cannot guarantee soundness and completeness
- Benefits
  - It is actually useful: checking implicit assumptions about program behavior, rediscovering formal specifications, documentation, etc
- Tool support: Daikon
  - Predefined templates instantiated with variables
  - http://plse.cs.washington.edu/daikon/

Department of Distributed and Dependable Systems

Further reading

- C.S. Pasareanu and W. Visser. **A Survey of New Trends in Symbolic Execution for Software Testing and Analysis**. STTT, 11(4), 2009
- P. Godefroid, N. Klarlund, and K. Sen. **DART: Directed Automated Random Testing**. PLDI 2005
- C. Cadar, D. Dunbar, and D. Engler. **KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs**. OSDI 2008
- P. Godefroid, M. Levin, and D. Molnar. **Automated Whitebox Fuzz Testing**. NDSS 2008
- E. Bounimova, P. Godefroid, and D. Molnar. **Billions and Billions of Constraints: Whitebox Fuzz Testing in Production**. ICSE 2013
- N. Tillmann, J. de Halleux, and T. Xie. **Transferring an Automated Test Generation Tool to Practice: from Pex to Fakes and Code Digger**. ASE 2014
- N. Nethercote and J. Seward. **Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation**. PLDI 2007
- C. Wang, R. Limaye, M. Ganai, and A. Gupta. **Trace-Based Symbolic Analysis for Atomicity Violations**. TACAS 2010
- M.D. Ernst, J.H. Perkins, P.J. Guo, S. McCamant, C. Pacheco, M.S. Tschantz, and C. Xiao. **The Daikon System for Dynamic Detection of Likely Invariants**. Sci. Comput. Program., 69(1-3), 2007

Department of Distributed and Dependable Systems