Symbolic execution as search, and the rise of solvers

Search and SMT

- Symbolic execution is appealingly **simple and useful**, but **computationally expensive**
- We will see how the effective use of symbolic execution **boils down to a kind of search**
- And also take a moment to see how its feasibility at all has been aided by **the rise of SMT solvers**

Path explosion

- Usually can't run symbolic execution to exhaustion
  - Exponential in branching structure

```
int a = α, b = β, c = γ;    // symbolic
if (a) ... else ...;
if (b) ... else ...;
if (c) ... else ...;
```

    - Ex: 3 variables, 8 program paths
  - Loops on symbolic variables even worse

```
int a = α;    // symbolic
while (a) do ...;
…
```

    - Potentially 2^31 paths through loop!

Compared to static analysis

- Stepping back: Here is a **benefit of static analysis**
  - Static analysis will actually **terminate** even when considering **all possible program runs**
- It does this by approximating multiple loop executions, or branch conditions
  - Essentially **assumes all branches**, and **any number of loop iterations**, are **feasible**
- But can **lead to false alarms**, of course

Basic (symbolic) search

- Simplest ideas: algorithms 101
  - Depth-first search (*DFS*) — **worklist = stack**
  - Breadth-first search (*BFS*) — **worklist = queue**
- Potential drawbacks
  - **Not guided** by any higher-level knowledge
    - Probably a bad sign
  - **DFS could easily get stuck** in one part of the program
    - E.g., it could keep going around a loop over and over again
  - Of these two, BFS is a better choice
    - But more intrusive to implement (can't easily be concolic)

Search strategies

- Need to **prioritize search**
  - Try to steer search towards paths more likely to contain assertion failures
  - Only run for a certain length of time
    - So if we don't find a bug/vulnerability within time budget, too bad
- Think of **program execution as a DAG**
  - Nodes = program states
  - Edge(n_1,n_2) = can transition from state n_1 to state n_2
- We need a kind of **graph exploration algorithm**
  - At each step, pick among all possible paths

Randomness

- We don't know *a priori* which paths to take, so adding some randomness seems like a good idea
  - Idea 1: **pick next path to explore uniformly at random** (*Random Path*, or RP)
  - Idea 2: **randomly restart search** if haven't hit anything interesting in a while
  - Idea 3: **choose among equal priority paths at random**
    - All of these are good ideas, and randomness is very effective
- One drawback of **randomness**: reproducibility
  - Probably good to use pseudo-randomness based on seed, and then record which seed is picked
    - Or bugs may disappear (or reappear) on later runs

Coverage-guided heuristics

- **Idea**: Try to **visit statements we haven't seen before**

- Approach
  - Score of statement = # times it's been seen
  - Pick next statement to explore that has lowest score

- Why might this work?
  - Errors are often in hard-to-reach parts of the program
  - This strategy tries to reach everywhere.

- Why might this *not* work?
  - Maybe never be able to get to a statement if proper precondition not set up

Generational search

- Hybrid of **BFS and coverage-guided**
  - *Generation 0*: pick one program at random, run to completion
  - *Generation 1*: take paths from *gen 0;* negate one branch condition on a path to yield a new path prefix; find a solution for that prefix; then take the resulting path
    - Semi-randomly assigns to any variables not constrained by the prefix
  - *Generation n*: similar, but branching off *gen n-1*
- Also uses a coverage heuristic to pick priority

Combined search

- Run **multiple searches at the same time**
  - Alternate between them; e.g., Fitnext
- Idea: no one-size-fits-all solution
  - Depends on conditions needed to exhibit bug
  - So will be as good as “best” solution, within a constant factor for wasting time with other algorithms
  - Could potentially use different algorithms to reach different parts of the program

SMT solver performance

- SAT solvers are at core of SMT solvers
  - In theory, could reduce all SMT queries to SAT queries
  - In practice, SMT-level optimizations are critical
- Some example extensions/improvements
  - Simple identities (x + 0 = x, x * 0 = 0)
  - Theory of arrays (read(x, write(42, x, A)) = 42)
    - 42 = array index, A = array, x = element
  - Caching (memoize solver queries)
  - Remove useless variables
    - E.g., if trying to show path feasible, only the part of the path condition related to variables in guard are important

Popular SMT solvers

- **Z3** - developed at Microsoft Research
  - http://z3.codeplex.com/
- **Yices** - developed at SRI
  - http://yices.csl.sri.com/
- **STP** - developed by Vijay Ganesh, now @ Waterloo
  - https://sites.google.com/site/stpfastprover/
- **CVC3** - developed primarily at NYU
  - http://www.cs.nyu.edu/acsys/cvc3/

But: Path-based search limited

```
int counter = 0, values = 0;
for (i = 0; i<100; i++) {
  if (input[i] == 'B') {
    counter++;
    values += 2;
  }
}
assert(counter != 75);
```

- This program has 2^{100} possible execution paths.
- Hard to find the bug:
  - $\binom{100}{75} \approx 2^{78}$ paths reach buggy line of code
  - *Pr(finding bug)* = $2^{78} / 2^{100} = 2^{-22}$

Symbolic execution systems

Resurgence

- Two key systems that triggered revival of this topic:
- **DART** — Godefroid and Sen, PLDI 2005
  - Godefroid = model checking, formal systems background
- **EXE** — Cadar, Ganesh, Pawlowski, Dill, and Engler, CCS 2006
  - Ganesh and Dill = SMT solver called STP (used in implementation), Cadar and Engler = systems
- Now on to next-generation systems

SAGE

- **Concolic executor** developed at **Microsoft Research**
  - Grew out of Godefroid's work on DART
  - Uses generational search
- Primarily **targets bugs in file parsers**
  - E.g., JPEG, DOCX, PPT, etc
  - Good fit for concolic execution
    - Likely to terminate
    - Just input/output behavior

SAGE Impact

- **Used on production software at MS**. Since 2007:
  - 500+ machine years (in largest fuzzing lab in the world)
    - Large cluster of machines continually running SAGE
  - 3.4 Billion+ constraints (largest SMT solver usage ever!)
  - 100s of apps, 100s of bugs (missed by everything else…)
    - Ex: *1/3 of all Win7 WEX security bugs found by SAGE*
  - Bug fixes shipped quietly to 1 Billion+ PCs
  - Millions of dollars saved (for Microsoft and the world)
  - SAGE is now used daily in Windows, Office, etc.

http://research.microsoft.com/en-us/um/people/pg/public_psfiles/SAGE-in-1slide-for-PLDI2013.pdf

KLEE

- **Symbolically executes LLVM bitcode**
  - LLVM compiles source file to .bc file
  - KLEE runs the .bc file
  - Grew out of work on EXE
- Works in the style of our basic symbolic executor
  - Uses `fork()` to manage multiple states
  - Employs a variety of search strategies
    - Primarily **random path + coverage-guided**
  - Mocks up the environment to deal with system calls, file accesses, etc.
- **Freely available with LLVM distribution**

KLEE: Coverage for Coreutils

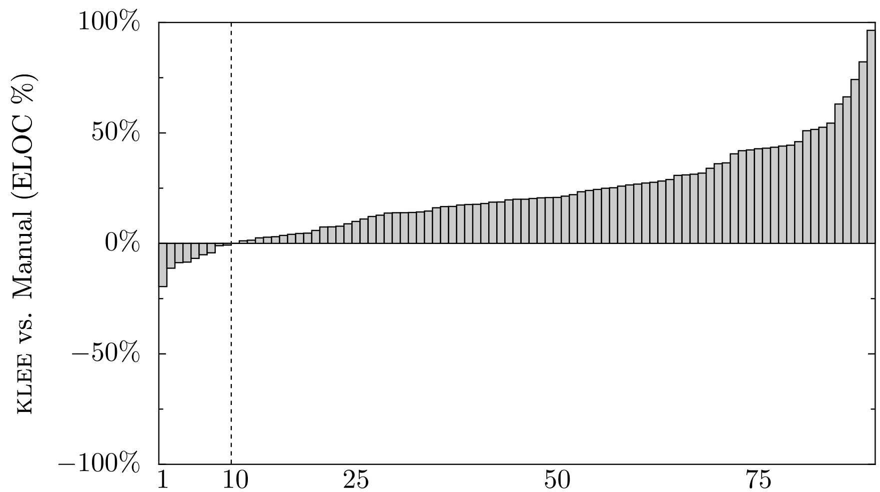

Figure 6: Relative coverage difference between KLEE and the COREUTILS manual test suite, computed by subtracting the executable lines of code covered by manual tests (L_{man}) from KLEE tests (L_{klee}) and dividing by the total possible: $(L_{klee} - L_{man})/L_{total}$. Higher bars are better for KLEE, which beats manual testing on all but 9 applications, often significantly.

Cadar, Dunbar, and Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs, OSDI 2008

KLEE: Coreutils crashes

`paste -d\\ abcdefghijklmnopqrstuvwxyz`
`pr -e t2.txt`
`tac -r t3.txt t3.txt`
`mkdir -Z a b`
`mkfifo -Z a b`
`mknod -Z a b p`
`md5sum -c t1.txt`
`ptx -F\\ abcdefghijklmnopqrstuvwxyz`
`ptx x t4.txt`
`seq -f %0 1`

t1.txt: `"\t \tMD5("`
t2.txt: `"\b\b\b\b\b\b\b\t"`
t3.txt: `"\n"`
t4.txt: `"a"`

Figure 7: KLEE-generated command lines and inputs (modified for readability) that cause program crashes in COREUTILS version 6.10 when run on Fedora Core 7 with SELinux on a Pentium machine.

Cadar, Dunbar, and Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs, OSDI 2008

Mayhem

- Developed at CMU (Brumley et al), **runs on binaries**
- Uses BFS-style search and native execution
  - **Combines best of symbolic and concolic strategies**
- **Automatically generates exploits** when bugs found

Mergepoint

- Extends Mayhem with a technique called **veritesting**
  - ***Combines*** **symbolic execution** with **static analysis**
  - Use static analysis for complete code blocks
  - Use symbolic execution for hard-to-analyze parts
    - Loops (how many times will it run?), complex pointer arithmetic, system calls
- Better **balance** of time **between solver and executor**
  - **Finds bugs faster**
  - **Covers more of the program** in the same time
- Found 11,687 bugs in 4,379 distinct applications in a Linux distribution
  - Including new bugs in highly tested code

Other symbolic executors

- **Cloud9** — Parallel, multi-threaded symbolic execution
  - Extends KLEE (available)
- **jCUTE**, **Java PathFinder** — symbolic execution for Java (available)
- **Bitblaze** — Binary analysis framework (available)
- **Otter** — directed symbolic execution for C (available)
  - Give the tool a line number, and it try to generate a test case to get there
- **Pex** — symbolic execution for .NET

Summary

- **Symbolic execution generalizes testing**
  - Uses static analysis to direct generation of tests that cover different program paths
- Used in practice to find **security-critical bugs** in **production code**
  - SAGE at Microsoft
  - Mergepoint for Linux
- **Many tools freely available**