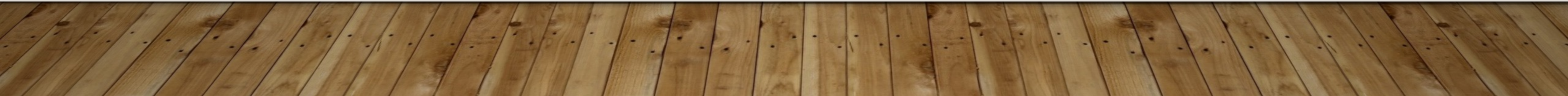
SYMBOLIC EXECUTION

YONGKANG MENG



PROGRAM ANALYSIS TOOL

- dynamic program analysis tool: needs to run the program binary
- gdb valgrind

- static program analysis tool can analyze the program without running it, usually only needs source code
- pylint,clang-tidy, clang-format
- symbolic execution tool also belongs to static program analysis

TESTING: PAIN POINT IN PRODUCTION DEVELOPMENT

- We test because we want to verify our function / component works well and find bugs
- When writing unit test / regression test, we put most of our effort on “make functions work well / return expected type when it gets called with expected parameters”
- We also mock some dummy error case in UT which everyone is smart enough not to trigger them
- However we could barely imagine how could a bug break the system until it occurs
- Tons of assertion – never works when running in production with release build
- Poor UT code coverage

SYMBOLIC EXECUTION

- Walk through all possible reachable branches of a program
- Warns you when there is a possible assertion failure or any dangerous operations(dereferencing a null pointer, divided by 0)
- Auto-generate high coverage unit test
- Does not even requires to run the program

OVERVIEW

JC King, CACM 1976.

Key idea: generalize testing by using unknown symbolic variables in evaluation

Symbolic values, not concrete data

For each condition the analyzer will try to fork a new working flow

STATE DEFINITION

```
1.  void foobar(int a, int b) {
2.     int x = 1, y = 0;
3.     if (a != 0) {
4.        y = 3+x;
5.        if (b == 0)
6.           x = 2*(a+b);
7.     }
8.     assert(x-y != 0);
9.  }
```

Warm-up example: Which values of a and b make the assert fail?

https://blog.csdn.net/weventure

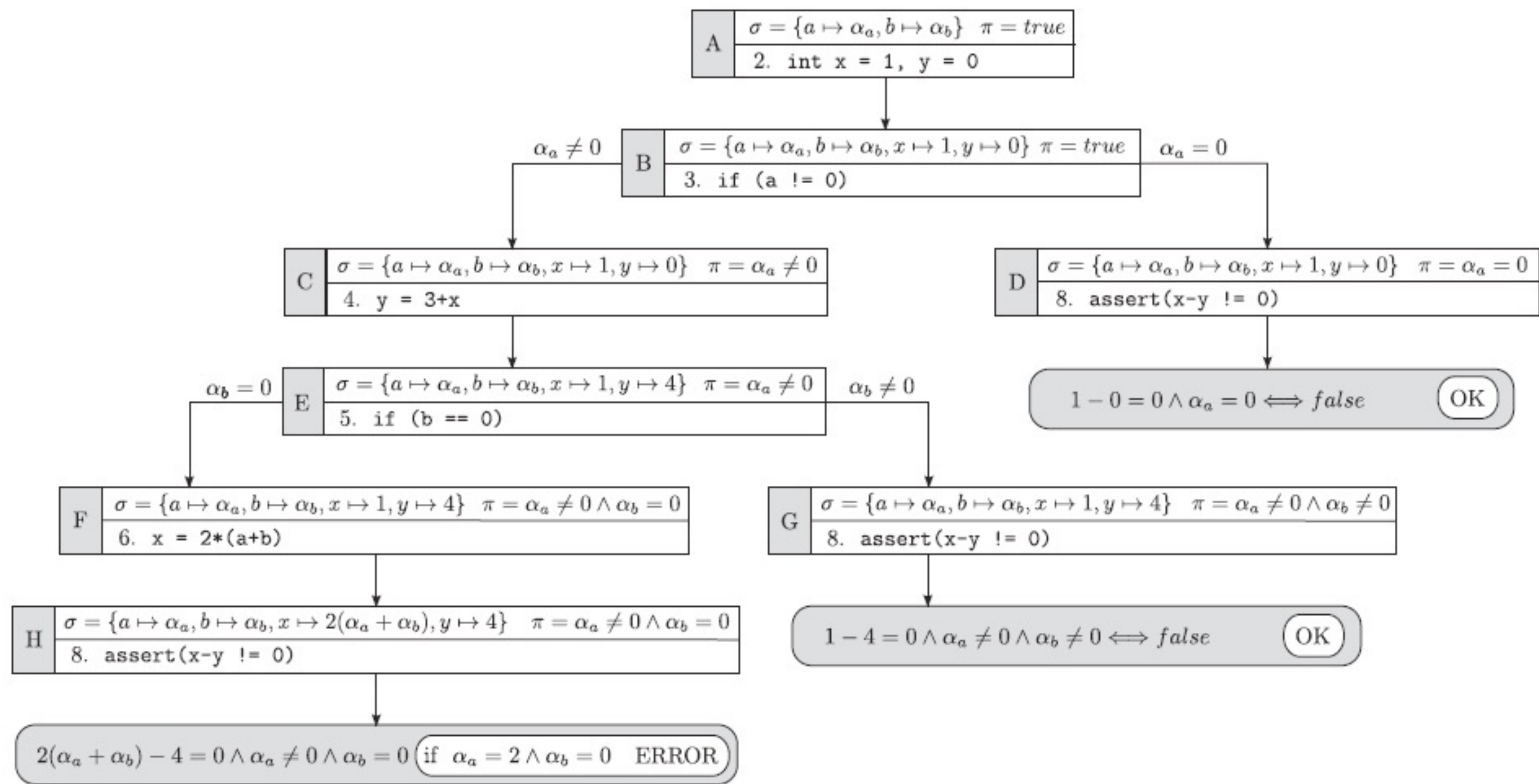


Fig. 2. Symbolic execution tree of function foobar given in Figure 1. Each execution state, labeled with an upper case letter, shows the statement to be executed, the symbolic store σ, and the path constraints π. Leaves are evaluated against the condition in the assert statement.

https://blog.csdn.net/wcventure

STATE DEFINITION

A symbolic execution state comprises of

- The statement that going to be executed
- A mapping (association) between symbolic variable and program variable
- A constraints set that must be met if we need to execute till there

STATE TRANSITION

Depending on $stmt$, the symbolic engine changes the state as follows:

- The evaluation of an assignment $x = e$ updates the symbolic store σ by associating x with a new symbolic expression e_s. We denote this association with $x \mapsto e_s$, where e_s is obtained by evaluating e in the context of the current execution state and can be any expression involving unary or binary operators over symbols and concrete values.
- The evaluation of a conditional branch `if` e `then` s_{true} `else` s_{false} affects the path constraints π. The symbolic execution is forked by creating two execution states with path constraints π_{true} and π_{false}, respectively, which correspond to the two branches: $\pi_{true} = \pi \wedge e_s$ and $\pi_{false} = \pi \wedge \neg e_s$, where e_s is a symbolic expression obtained by evaluating e. Symbolic execution independently proceeds on both states.
- The evaluation of a jump `goto` s updates the execution state by advancing the symbolic execution to statement s.

https://blog.csdn.net/wcventure

KEY CONCEPTS OF SYMBOLIC EXECUTION

- state
- constraints
- constraints solver (SMT, SAT)
- query

QUERY

- Figure out if the path condition is satisfiable
- Is array access a[i] out of bounds?
- Generate concrete inputs that execute the same paths
- Will this divide operation cause zero-division error?

SYMBOLIC EXECUTION

- Selects a state to run and then symbolically execute a single instruction in the context of that state. This loop continues until there are no states remaining, or a user-defined timeout is reached.

- For each conditional statement there will be an exponential increase of number of states, NP question

CHALLENGES

- Path Explosion

Code example: for(int i = 0; i < get_number_from_terminal(); i++)

- Constraint Solving

- Environment mocking

PATH EXPLOSION

- Early branch prune
- Customized searching algorithm: random path selection and coverage-optimized search
- Why these 2 algorithms work?

CONSTRAINT SOLVING

- Current constraint solver can only support simple, linear constraints set
- We should see constraint solver as a black box and do optimization outside it
- What does KLEE do?

KLEE

- Execution Generated Testing: add concrete values to symbolic execution progress

- Expression Rewriting

- Implied Value Concretization

Example: (x + 1 == 10) - > x == 9

KLEE

- constraint independence
- Given the constraint set $\{i < j, j < 20, k > 0\}$, a query of whether $i = 20$ just requires the first two constraints
- Counter-example cache (incremental solving)
- In KLEE all query results are stored in a cache that maps constraint sets to concrete variable assignments (or a special No solution flag if the constraint set is unsatisfiable)

COUNTER-EXAMPLE CACHE

- Historical query result: $(x + y < 10) \wedge (x > 5) \Rightarrow \{x = 6, y = 3\}$
- Subset query: $(x + y < 10)$
- Superset query: $(x + y < 10) \wedge (x > 5) \wedge (y > 2)$: Highly possible that the solution also works here
- For negative result: when a subset of a constraint set has no solution, then neither does the original constraints set. Adding constraints to an unsatisfiable constraint set cannot make it satisfiable.
- Special design: derived from the UBTree structure of Hoffmann and Hoehler, can search for superset and subset quickly.

BUT WHAT IF ALL THESE CAN NOT HELP US OUT

- KLEE: give up the state, goes to other state for searching
- Symbolic execution supporting concolic testing: try with a random 'concrete input'
- That's why we can not reach 100% coverage rate even if each line is reachable

ENVIRONMENT MOCKING

- array
- memory object
- network
- lib function without source code
- system call
- choice: mock at lib level or system call level

MOCK LIB FUNCTIONS

```
1 : ssize_t read(int fd, void *buf, size_t count) {
2 :    if (is_invalid(fd)) {
3 :       errno = EBADF;
4 :       return −1;
5 :    }
6 :    struct klee_fd *f = &fds[fd];
7 :    if (is_concrete_file(f)) {
8 :       int r = pread(f−>real_fd, buf, count, f−>off);
9 :       if (r != −1)
10:          f−>off += r;
11:       return r;
12:    } else {
13:       /* sym files are fixed size: don't read beyond the end. */
14:       if (f−>off >= f−>size)
15:          return 0;
16:       count = min(count, f−>size − f−>off);
17:       memcpy(buf, f−>file_data + f−>off, count);
18:       f−>off += count;
19:       return count;
20:    }
21: }
```

KLEE PERFORMANCE

Coverage (w/o lib)	COREUTILS KLEE tests	COREUTILS Devel. tests	BUSYBOX KLEE tests	BUSYBOX Devel. tests
100%	16	1	31	4
90-100%	40	6	24	3
80-90%	21	20	10	15
70-80%	7	23	5	6
60-70%	5	15	2	7
50-60%	-	10	-	4
40-50%	-	6	-	-
30-40%	-	3	-	2
20-30%	-	1	-	1
10-20%	-	3	-	-
0-10%	-	1	-	30
Overall cov.	84.5%	67.7%	90.5%	44.8%
Med cov/App	94.7%	72.5%	97.5%	58.9%
Ave cov/App	90.9%	68.4%	93.5%	43.7%

Table 2: Number of COREUTILS tools which achieve line coverage in the given ranges for KLEE and developers' tests (library code not included). The last rows shows the aggregate coverage achieved by each method and the average and

SPECIAL USAGE: REGRESSION TEST

```
1 : unsigned mod_opt(unsigned x, unsigned y) {
2 :   if((y & −y) == y) // power of two?
3 :     return x & (y−1);
4 :   else
5 :     return x % y;
6 : }
7 : unsigned mod(unsigned x, unsigned y) {
8 :   return x % y;
9 : }
10: int main() {
11:     unsigned x,y;
12:     make_symbolic(&x, sizeof(x));
13:     make_symbolic(&y, sizeof(y));
14:     assert(mod(x,y) == mod_opt(x,y));
15:     return 0;
16: }
```

Figure 11: Trivial program illustrating equivalence checking. KLEE proves total equivalence when $y \neq 0$.

FOLLOW-UP PAPER/PROJECTS

- S2E: based on KLEE, integrated with qemu, implements 'selective running'
- A survey of Symbolic Execution Techniques
- Symbolic Execution for Software Testing: Three Decades Later
- https://xiongyingfei.github.io/SA/2020/main.htm (Peking University Software Analysis)

THANK YOU