Symbiotic: finding bugs in C programs

Marek Chalupa, Masaryk University

DevConf 2019, 25. 1. 2019
Nowadays, we usually use testing to find bugs.

It is hard to write tests that reveal bugs.

Automatic test generation can be used.
Symbolic Execution

- Given a program with inputs:
  - use symbols instead of inputs,
  - execute the program with these symbols,
  - fork the execution on branchings.

- This way we can enumerate all possible paths in the program.
void foo(int a, int b, int c)
{
    if (a > 0) {
        c = 2;
        if (b < 0) {
            c = b;
        }
    } else {
        c = -2;
    }
    if (a + b + c <= 0)
        error();
}
Symbolic execution

a := α, b := β, c := γ

\[\alpha \leq 0\]

\[\alpha > 0\]

\[c = -2\]

\[c = 2\]

\[b < 0;\]

\[[\beta \geq 0]\]

\[[\beta < 0]\]

\[\text{if (a + b + c <= 0);}\]

\[\text{error();}\]

\[\text{if (a + b + c <= 0);}\]

\[\text{error();}\]

\[\alpha > 0\]

\[\text{if (a + b + c <= 0);}\]

\[\text{error();}\]

\[\alpha \leq 0\]

\[\alpha + \beta - 2 > 0\] \[\alpha + \beta - 2 \leq 0\]

\[\alpha > 0\]

\[\alpha + \beta + \beta > 0\] \[\alpha + \beta + \beta \leq 0\]

\[\alpha + \beta - 2 > 0\] \[\alpha + \beta - 2 \leq 0\]
- Open-source symbolic executor for LLVM bitcode.
- http://klee.github.io/
Symbolic execution is computationally very demanding.
Symbolic execution is computationally very demanding.

Preprocess the code before giving it to KLEE.
Symbiotic

LLVM → KLEE

c

c

c

8 / 16
Symbiotic

LLVM → KLEE

. c

c ode

optimizations

t est
t est
Symbiotic

LLVM → KLEE

- Code optimizations
- Program slicing
Program Slicing

- Compute dependencies between instructions.
- We say that instruction A depends on instruction B if:
  - instruction A uses values generated by instruction B, or
  - instruction A is not executed if we go some other way at (branching) B.
- Slicing: keep only the instructions on which the error (transitively) depends.
int zeroing(char *buf, size_t size)
{
    int n = input();
    for (int i = 0; i < n; ++i) {
        assert(i < size && "Out of bounds");
        buf[i] = 0;
    }

    return 0;
}
int zeroing(char *buf, size_t size)
{
    int n = input();
    for (int i = 0; i < n; ++i) {
        assert(i < size && "Out-of-bounds");
        buf[i] = 0;
    }

    return 0;
}
Symbiotic - cont.

LLVM → .c

program slicing

→ KLEE

code optimizations

test test
Symbiotic - cont.

- LLVM
  - .c
  - code optimizations

- find possible errors
- program slicing

- KLEE
  - test
  - code
Finding possible errors

- Symbiotic performs light-weight static analysis before slicing.
- The static analysis looks for chosen errors (integer overflow, double free, dangling pointer dereference,...).
- Instructions that *may* exhibit an error are set as slicing criteria.
Symbiotic - cont.

LLVM → .c → LLVM → find possible errors → program slicing → KLEE

code optimizations
Symbiotic - cont.

LLVM → KLEE

- Code optimizations
- Find possible errors
- Program slicing

KLEE →

- Test
Apart from the already mentioned steps, Symbiotic:
- automatically marks memory symbolic,
- replaces undefined functions with symbolic stubs.

**Limits:** no C++ (exceptions), no threads (yet).

Unfortunately, Symbiotic still does not scale to large programs.
Future Directions

- **Scalability**
  - slicing (faster analyses),
  - symbolic execution (abstraction),
  - different back-ends than KLEE.

- Better modelling of the environment (POSIX).

- C++ and threads.
Conclusion

- Symbiotic is a tool for finding bugs in C programs.
- Combines fast static analysis with program slicing and symbolic execution.
- Runs on sequential C code.
- Still needs some work
  - scalability issues,
  - no C++ and threads.

https://github.com/staticafi/symbiotic

Thank you!
Conclusion

- Symbiotic is a tool for finding bugs in C programs.
- Combines fast static analysis with program slicing and symbolic execution.
- Runs on sequential C code.
- Still needs some work
  - scalability issues,
  - no C++ and threads.

https://github.com/staticafi/symbiotic

Thank you!