CS 250 Software Security

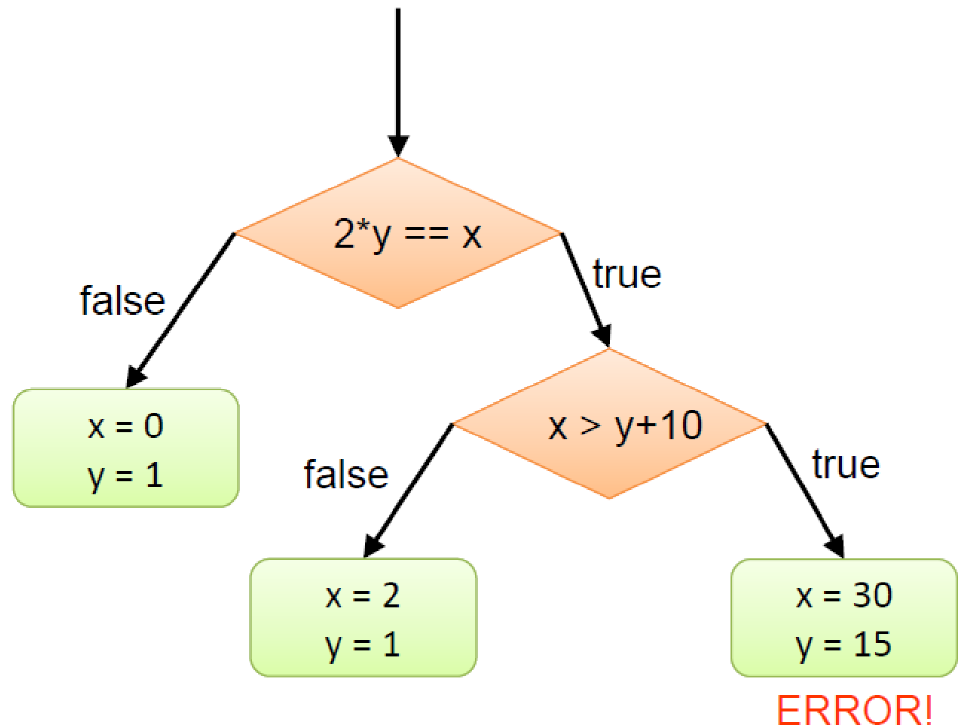
Symbolic Execution

Classic Symbolic Execution

```
int twice (int v) {
        return 2*v;
}

void testme (int x, int y) {
        z = twice (y);
        if (z == x) {
                if (x > y+10)
                        ERROR;
                }
        }
}

/* simple driver exercising testme() with
int main() {
        x = sym_input();
        y = sym_input();
        testme(x, y);
        return 0;
}
```



First paper: 1976 Symbolic Execution and Program Testing

Problem 1: Infinite execution path

```
void testme_inf () {
        int sum = 0;
        int N = sym_input();
        while (N > 0) {
            sum = sum + N;
            N = sym_input();
        }
}
```

Figure 3. Simple example to illustrate infinite number of execution paths.

Problem 2: Unsolvable formulas

```
int twice (int v) {
        return (v*v) % 50;
}
```

Figure 4. Simple modification of the example in Figure 1. The function twice now performs some non-linear computation.

Problem 3: Symbolic modeling

- External function calls and system calls are hard to model
- For efficiency, symbolic execution systems often model libc function calls.
  - File system related
  - String operations

Concolic Testing

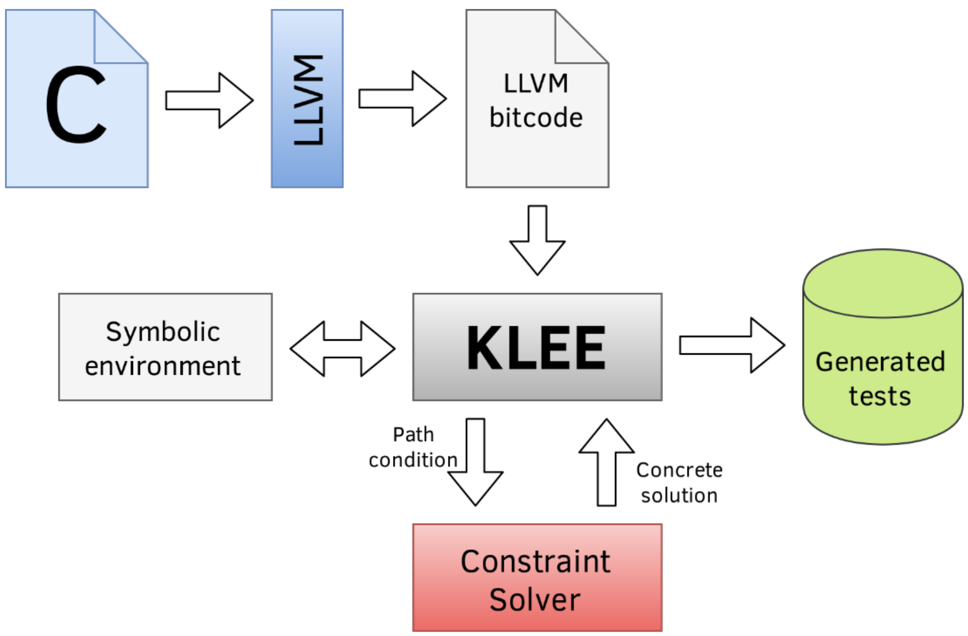
- Performs symbolic execution dynamically, while the program is executed on some concrete input values.
- Generate some random input: x=22, y=7 and execute the program both concretely and symbolically
- The concrete execution take the “else” branch on Line 7 and the symbolic execution generates the path constraint x != 2y
- Negates a conjunct in the path constraint and solves x==2y and get a new test input x=2, y=1
- Test the program with the new input

Concolic Testing: What is the benefit?

- Solve complex formulas
  - x == (y*y) mod 50, unsolvable if both x and y are symbolic
  - if we concretize y to its concrete value, now solvable
- External library call and system call
  - E.g., fd = open(filename)
  - Set filename to its concrete value “/tmp/abc.txt”
  - Execute the system call concretely
  - Set fd to be concrete after the system call return

How to implement it?

- Let's start with KLEE



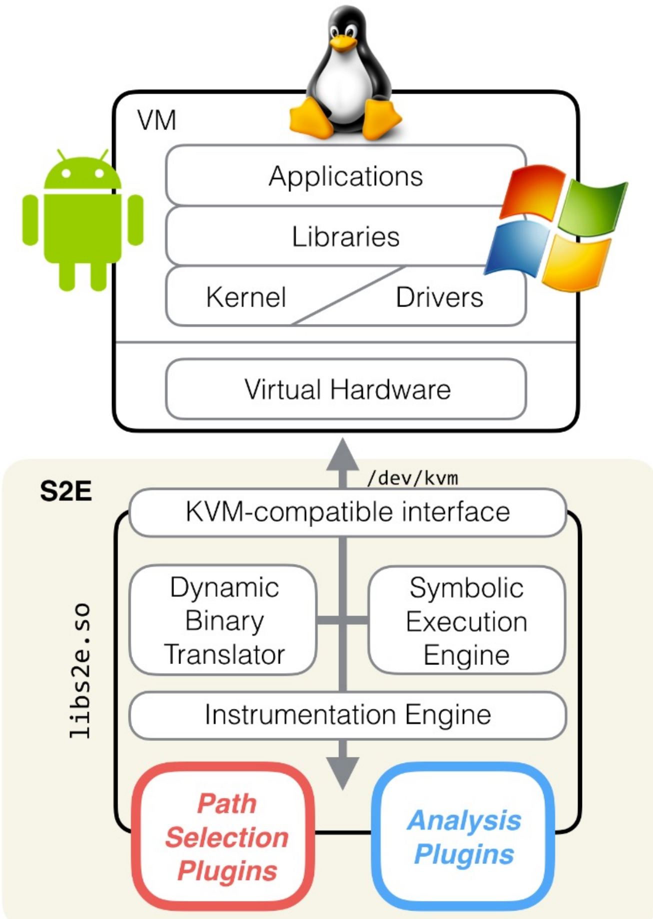
- Symbolically Interpret and Concretely Execute LLVM IR
- Full Symbolic Environment Modeling
- State Forking
- Simple State Scheduling: Random/Coverage-Optimized

https://klee.github.io/

Angr: Symbolic Execution for Binary

- https://angr.io/
- Follows the similar design as Klee
- Klee: C code -> LLVM bitcode, interpret LLVM bitcode
- Angr: Binary -> VEX IR, interpret VEX IR in Python!
  - So it is slow!

S2E: Selective Symbolic Execution for Binary



- https://s2e.systems/
- Symbolically execute a software component in the VM
- Concretely execute the rest
- Based on QEMU
- QEMU TCG IR -> LLVM IR -> KLEE backend

Still not good enough!

- In DARPA CGC, most of the vulnerabilities are found by fuzzing!
- Too slow: Constraint collection + Constraint solving
- State explosion problem
- Complete environment modeling is hard

QSYM: A fast and scalable concolic execution engine for binary

- https://github.com/sslab-gatech/qsym

- Big idea:
  - Sacrifice soundness for efficiency

- It will be paired up with a fuzzer, so efficiency is way more important than soundness

QSYM: Get rid of IRs

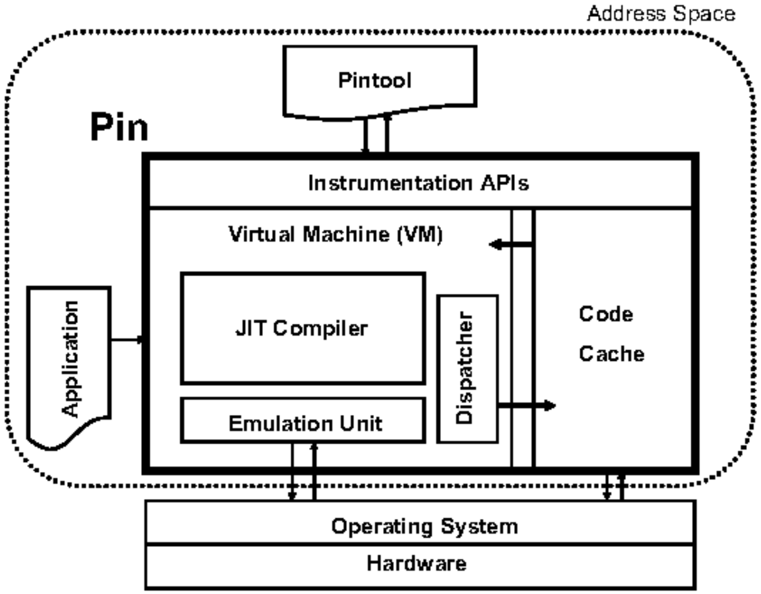
Executor	chksum	md5sum	sha1sum	md5sum(mosml)
Native	0.008	0.014	0.014	0.001
KLEE	26.243	32.212	73.675	0.285
angr	-	-	-	462.418

Why Intermediate Representations (Irs)?

- Pros
  - Faithfully capture the instruction semantics
  - Provide architecture-independent interpretation
- Cons
  - IR statements are 4-5 timers larger than instructions
  - Emulating/Interpreting IR is slow

- QSYM's design decision
  - Directly extract symbolic expressions/constraints from instructions
  - May not deal with complex instructions
  - Hard to support multiple architectures
  - Sacrifice soundness for efficiency

QSYM: Symbolic Emulation



- Workflow:
  - Pintool-based dynamic binary instrumentation
  - For each instruction, checks if any operand is symbolic
  - If so, pass this instruction to symbolic backend
- Problems:
  - Pin is closed source
  - Support only one arch
  - Shadow value analysis in Pin is expensive
  - A better alternative: QEMU

QSYM: Re-execution vs. State Forking

- State forking
  - No need to re-execute (just recover from the snapshot)
  - State in concolic execution = program state + kernel state
  - Forking program state is trivial, but forking kernel state is not
  - Expensive to manage the states
  - Requires perfect environment modeling
- Re-execution
  - No state management
  - May not be that slow
  - Time vs. Space trade-off
  - Concrete environment

QSYM: Models minimal system calls

- Only model system calls that are relevant to user interactions
  - Standard input, file read, …
- Other system calls: just use concrete values
  - Execute them concretely
- It will result in incomplete constraints
  - Yes, QSYM only models simple instructions anyway
- Concretization needs to over-constrained analysis

QSYM: Strict Branch Flipping Policy

- Look at current branch and last branch
- Flip the current branch if this pair is new
- It can solve state/path explosion problem, but may also miss important branches

QSYM: Constraint Solving

```
// @funcs.c:221 in file v5.6
if ((ms->flags & MAGIC_NO_CHECK_COMPRESS) == 0) {
  m = file_zmagic(ms, &b, inname); // zlib decompress
  ...
}

// other interesting code
```

```
// @funcs.c:177 in file v5.6
// looks_ascii()
if (ch >= 0x20 && ch < 0x7f)
  ...
// file_tryelf()
if (ch == 0x7f)
  ...
```

Figure 3: The first example shows that collecting complete constraints for complicated routines such as `file_zmagic()` could prohibit finding new paths. The second example shows that if a given concrete input follows a true path of `looks_ascii()`, it over-constrains the path not to find a true path of `file_tryelf()`.

- Full path constraints
  - Too expensive to collect
  - Sometimes over-constrained
- Nested Branch Solving
  - Only include constraints that have data dependencies with the last branch
- Optimistic Solving
  - Only solve the last branch condition

QSYM: Basic Block Pruning

- Some loop bodies can be executed repeatedly to generate symbolic constraints
- Long execution and complex constraints
- If a basic block is executed too frequently, stop generating constraints for them
- Exponential back-off

QSYM is great! Is that it?

- Even faster symbolic emulation
  - For Source code:
    - Symbolic execution with SymCC: Don't interpret, compile!, in the 29th USENIX Security Symposium, August 2020
    - SymSan: Time and Space Efficient Concolic Execution via Dynamic Data-Flow Analysis, *in the 31st USENIX Security Symposium*, August 2022.
  - For Binary code:
    - Compilation-based symbolic execution for binaries, in the ISOC Network and Distributed System Security Symosium, February 2021.
    - Our Work in submission
- Faster constraint solving
  - JIGSAW: Efficient and Scalable Path Constraints Fuzzing, *in the 43rd IEEE Symposium on Security and Privacy*, May 2022.
- More intelligent branch flipping
  - Marco: A Stochastic and Asynchronous Concolic Explorer, *to appear in the 46th International Conference on Software Engineering*, April 2024.

What else can be done?

- Let's brainstorm!