

Symbolic Execution

Ondřej Lengál

SAV'25, FIT VUT v Brně

3 November 2025

Manual Testing

- users try **input vectors**, trying to break a program
- **pros**:
 - ▶ **complete**: a failing input vector **can be “easily” executed**
 - not always easy: concurrency, nondeterministic memory layout, etc.
 - ▶ can be directed to some *corner cases*
- **cons**:
 - ▶ **unsound**: problematic coverage of unexpected corner cases
 - ▶ expensive (testers needed)

Random Testing

- generate *a lot of* **random vectors** and feed them into a program
- **pros:**
 - ▶ can easily create many inputs
- **cons:**
 - ▶ difficult to cover corner cases
 - ▶ many inputs can exercise the same paths through the program
- e.g. QuickCheck for Haskell:

```
prop_RevRev xs = reverse (reverse xs) == xs
```

```
Main> quickCheck prop_RevRev  
OK, passed 100 tests.
```


Static Analysis

Data flow analysis, abstract interpretation, ...:

■ pros:

- ▶ can analyze all possible runs of programs
- ▶ sold by companies (AbsInt, Coverity, GrammaTech, etc.)
- ▶ easy to use (with a catch)

■ cons:

- ▶ often unsound (in practice)
- ▶ *abstraction* $\rightsquigarrow$ **false positives (incomplete)**
 - it can take a lot of effort to sieve through them
- ▶ does not provide concrete failing input vectors

Static Analysis — Example

```
char input[10];
read(fd, input, 10);
int counter = 0;
for (size_t i = 0; i < 10, ++i) {
    if (input[i] == 'B') {
        ++counter;
    }
}
assert(counter != 10);
```

- e.g., abstract interpretation might just say that assert is reachable
- developer needs to assess whether it is true
- abstraction of static analysis can be different than the one used by developer

Symbolic Execution — A middle ground

- **Testing**: works, but each test tries only one possible execution

- ▶ we hope that test cases generalize (no guarantees)

```
assert(f(2) == 21);  
assert(f(3) == 42);  
assert(f(4) == 63);
```

- **Symbolic Execution**: generalizes random testing

- ▶ allows one to assign unknown **symbolic** values to variables, e.g., $y = \alpha$
- ▶ tests may then cover all possible values of the symbolic value

```
assert(f(y) == 21*(y-1));
```

- ▶ if an execution path depends on a symbolic value, **fork** execution

```
unsigned f(unsigned x) {  
    return (x > 0)? 21*(x-1) : 13;  
}
```

Symbolic Execution

- can be seen as an execution of a program in a mixed **symbolic domain**
- similar to abstract interpretation (but with significant differences)

Standard execution semantics:

- in every step, all variables and allocated memory cells have concrete values
 - ▶ concrete state: configuration of a program

Symbolic execution semantics:

- variables and allocated memory cells can also have **symbolic** values
 - ▶ e.g., α , $2 \cdot \beta + 3$, $\gamma + \text{"Hello World"}$, ...
 - ▶ symbolic values are usually introduced to represent **inputs** of the program
- operators need to be extended to be able to work with symbolic values

Symbolic Execution (cntd.)

■ **symbolic state** is a triple $st = (line, store, pc)$ where:

- ▶ $line \in \mathbb{N}$ denotes a program line
- ▶ $store : Mem \rightarrow Sym$ represents (symbolic) values of variables and allocated memory cells
 - Mem : the set of memory locations
 - Sym : the set of symbolic values (it also contains all concrete values)
 - ($\rightarrow$ denotes *partial function*)
- ▶ pc : **path condition**, a formula of first-order logic (over some suitable theory $\mathbb{T}$ that represents program operations and tests) that accumulates conditions that needed to hold to reach st
 - initially set to *true*
 - extended when execution is **forked**: more formulae are appended using **conjunction** $\wedge$

Extending path condition

Let φ be a formula obtained by substituting (symbolic) values of variables into a test

- e.g. if $store = \{x \mapsto \alpha, y \mapsto 2 \cdot \sin \beta, \dots\}$, and there is a test

```
if (3 * x > log(y)) {  
    stmt1;  
    ...  
else {  
    stmt2;  
    ...  
}
```

we obtain for the if branch $\varphi: 3 \cdot \alpha > \log(2 \cdot \sin \beta)$

Extending path condition (cntd.)

- φ is a formula representing a test in a program (e.g. inside an `if` statement)
- suppose pc is $\mathbb{T}$ -satisfiable, then at most one of the following can hold:
 - 1 $pc \Rightarrow_{\mathbb{T}} \varphi$ (the `then` branch)
 - 2 $pc \Rightarrow_{\mathbb{T}} \neg\varphi$ (the `else` branch)where $\Rightarrow_{\mathbb{T}}$ denotes *logical consequence* wrt. theory $\mathbb{T}$
 - ▶ i.e., whether all $\mathbb{T}$ -models of pc are also $\mathbb{T}$ -models of φ (or $\neg\varphi$)
- if one of the logical consequences holds, no forking and extension of pc is required
 - ▶ only one branch is **feasible**
- when neither of the consequences holds, we speak about **forking execution**:
 - ▶ the execution forks because both branches are **feasible**; pc is then extended as:
 - 1 $pc' := pc \wedge \varphi$ (for the `then` branch)
 - 2 $pc' := pc \wedge \neg\varphi$ (for the `else` branch)
- logical consequence is checked using an **SMT Solver**

Example of symbolic execution

```
int power(x, y)
{
1:  int z = 1;
2:  int j = 1;
3:  while (y - j >= 0)
    {
4:      z *= x
5:      ++j;
    }
6:  return z
}
```

line	x	y	z	j	pc

Symbolic execution — high level algorithm

```
1 symState := (line: 0, store:  $\emptyset$ , pc: true) // initial symbolic state
2 workSet := {symState}
3 while workSet  $\neq \emptyset$ :
4     st := workSet.getAndRemove() // many ways to implement
5     st' := symbolically execute from st until a fork to  $l_1$  and  $l_2$  with condition  $\varphi$ , or EXIT,
6             while checking for errors and modifying store accordingly
7     if st'.line == EXIT: continue
8     workSet.add((line:  $l_1$ , store: st'.store, pc: st'.pc  $\wedge \varphi$ ))
9     workSet.add((line:  $l_2$ , store: st'.store, pc: st'.pc  $\wedge \neg \varphi$ ))
```

Symbolic execution tree

paths taken in a symbolic execution can be expressed using a **symbolic execution tree**

- **control points** of the program are nodes
- **statements** are edges
- **tests** that are not logical consequ. of the pc for the branch above them have two outgoing edges:
 - ▶ *true* (for then)
 - ▶ *false* (for else)

properties of the tree:

- for every **terminal leaf** L , there are concrete (non-symbolic) inputs that can navigate execution to L
 - ▶ a terminal leaf corresponds to a finished path
- every two terminal nodes have distinct path conditions, i.e., $pc_1 \wedge pc_2$ is $\mathbb{T}$ -UNSAT

Symbolic execution for verification

program verification:

- every `assume( $\varphi$ )` (in function contracts) will update $pc' := pc \wedge \varphi$
- every `assert( $\varphi$ )` will test whether $pc \Rightarrow_{\mathbb{T}} \varphi$, if not: **report error**
- during execution of a program (or in preprocessing), more statements are added, e.g.:
 - ▶ for a fixed-size array `a` of size `N`, every access `a[x]` where `x` has a symbolic value changes:

`assert(x < N && x >= 0);`

`a[x] = y;      -->      a[x] = y;`

- ▶ every integer division is checked for zero-division:

`assert(x != 0);`

`y = 42 / x;      -->      y = 42 / x;`

- ▶ pointer accesses are checked for `nullptr`:

`assert(x != nullptr);`

`y = *x;      -->      y = *x;`

(checking for dereference of undefined memory locations is more difficult)

- ▶ etc.

Search strategies

given by the implementation of *workSet.getAndRemove()*

- if **stack**: DFS
 - ▶ can easily get stuck in some part of the program
- if **queue**: BFS
 - ▶ usually better, but still not guided by any higher-level knowledge
- more complex strategies:
 - ▶ try to **steer the search** (using priorities) towards assertion failures
 - ▶ reasoning on the *control flow graph* (CFG) of the program
- **randomness**: we don't know which paths to take... why not pick them randomly?
 - 1 pick next path uniformly at random
 - 2 randomly restart search if nothing interesting found for a while
 - 3 when choosing between two paths with the same priority, flip a coin

Search strategies

■ coverage-guided heuristics:

- ▶ try to visit statements not seen before
- ▶ increments statement's score when hit
- ▶ pick a statement with lowest score
- ▶ can be difficult to find how to get to a statement (undecidable)

■ generational search (hybrid of BFS + coverage-guided):

- ▶ **GEN 0**: pick one program path at random, run to completion
- ▶ **GEN $n + 1$** : take *pc* from GEN n and negate one branch condition, repeat
- ▶ *modification*: negate *all* branch conditions, get several paths
- ▶ often used with concolic execution

■ combined search:

- ▶ run multiple searches at once

Issues

- we need to test logical consequence $pc \Rightarrow_{\mathbb{T}} \varphi$ between path conditions and tests
 - ▶ reasoning in some theories is still challenging for SMT solvers
 - e.g., arithmetic over natural numbers, string variables w/ operations, ...
- fixed-size/precision integer and floating-point variables in concrete execution:
 - ▶ are often represented using “ideal” symbolic values from $\mathbb{N}$ or $\mathbb{R}$
 - ▶ more faithful representation uses theory of [FixedSizeBitVectors](#) and [FloatingPoint](#)
- problems modelling **memory**:
 - ▶ checking for invalid memory accesses $a[x]$ where
 - a is an array and
 - x has a symbolic value
 - ▶ unsatisfactory solution:
 - $ite(v(x) = 1, v(a[1]), ite(v(x) = 2, v(a[2]), \dots))$
 - ▶ theory of arrays
 - ▶ even more problems with dynamic data structures
 - model the whole memory as a big array? ... does not scale

■ **path explosion:**

- ▶ when symbolic execution keeps forking
- ▶ e.g. on cycles without a fixed number of iterations
- ▶ cf. bounded model checking (BMC)

■ **imprecision:** reasons

- ▶ pointer manipulation
- ▶ SMT solver limitations
- ▶ complex arithmetic operations (hashing, encryption, etc.)
- ▶ system/library calls (e.g. `libc`):
 - can contain native code
 - very complicated (e.g. call of `malloc`)
 - using a simpler version can be advantageous (e.g., `newlib`, a version of `libc` for embedded systems)
 - need to make a model (a lot of work)

Concolic testing

- **concolic** = **con**crete + **symbolic**
- program is executed at the same time on symbolic and concrete inputs
 - ▶ program is given *concrete inputs* I , which are shadowed by *symbolic values*
 - the symbolic values generalize the concrete inputs
 - ▶ execution of the program is **instrumented**: computation of path condition
 - ▶ when a path terminates
 - choose a **decision point** d in its path condition $pc = \varphi \wedge d \wedge \psi$
 - obtain a new path condition prefix $pc' = \varphi \wedge \neg d$
 - generate new inputs $I' \models pc'$
 - re-run the program with I' as its inputs
- for system calls, use the concrete value
 - ▶ symbolic-ness is lost at such calls
- no need to call SMT solver at conditions

Tools

- **KLEE**: symbolic execution of LLVM bitcode
- **Pex**: symbolic execution for .NET
- **CREST**: concolic testing of C programs
- **SAGE**: targets file parsers (e.g., .doc, .jpeg)
 - ▶ used daily in Microsoft Win, Office, ...
 - ▶ found 100s of bugs in 100s of apps

```
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\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.

Tools

- **Mergepoint**: static analysis + SE
- **Otter**: symbolic execution for C
 - ▶ provide a line number
 - ▶ Otter will try to get there
- **Symbiotic**: symbiosis of several approaches:
 - 1 program instrumentation (adding monitors for various properties)
 - 2 static program slicing (removing statements that are irrelevant to the property)
 - 3 symbolic execution based on KLEE
- **PyEx**: symbolic execution of Python programs

Used materials from

- Jan Strejček, Masaryk University
- Michael Hicks, University of Maryland