

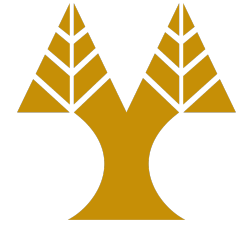
CS451 – Software Analysis

Lecture 16

Constraint Solving with Z3

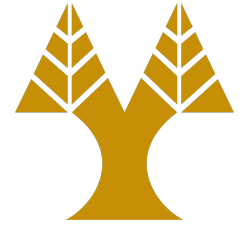
Elias Athanasopoulos
elathan@cs.ucy.ac.cy

Constraint solving

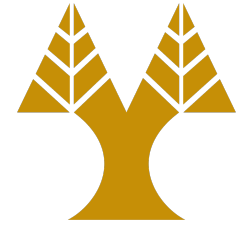


- During a symbolic execution, the analysis may produce several constraints
- Solving a large set of formulas is not trivial
 - Boolean satisfiability is NP-complete, while the SMT problem is NP-hard
- There are specific constraint-solving tools, based on mathematics
 - Boolean satisfiability problem (SAT) solvers
 - Satisfiability modulo theories (SMT) solvers
- Symbolic engines use a separate constraint solver
 - Most engines allow multiple constraint solvers to be plugged

Z3



- Z3 is an open-source SMT solver developed by Microsoft Research
- Z3 is just the constraint solver, but can be connected to symbolic execution engines
- Available bindings for C/C++ and Python
- Command-line interface



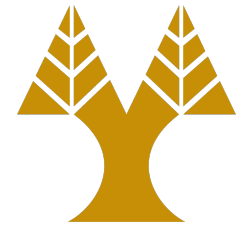
Example

```
x = int(argv[0])  
y = int(argv[1])  
z = x + y
```

```
if (x >= 5)  
    foo(x, y, z)  
    y = y + z  
    if (y < x)  
        baz(x, y, z)  
    else  
        qux(x, y, z)  
else  
    bar(x, y, z)
```

Is baz() reachable and for which values?

We are going to use Z3 to answer this question.



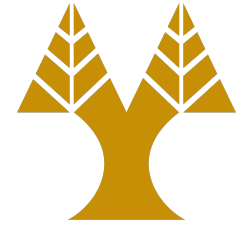
Declaring variables in Z3

- We can run Z3 from the command line and declare the variables of the program as *constants*

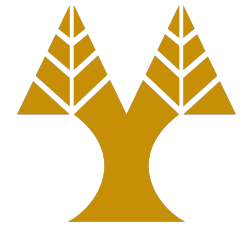
```
$ z3 -in  
(declare-const x Int)  
(declare-const y Int)  
(declare-const z Int)  
(declare-const y2 Int)
```

- Z3 models variables as constants and tries to find a solution
 - This is different with executing a program, where accessing variables is ordered by the way instructions are executed

Static single assignment (SSE)



- Z3 attempts to *solve* the constraints through a model
 - The model does not encapsulate the computational aspects, i.e., it doesn't matter if x becomes 5 before y becomes 4
- This has an implication that a double assignment may produce a non-solvable problem
 - If y is 5 and then becomes 4, then there will be two conflicting constraints introduced in the model, where y should be 4 and 5
- SSE assigns each variable only once and uses additional variables for new assignments
 - We do that with y_2 , when y is updated to become $y + z$ in the program



Adding constraints

- Further to declaring constants, we can add constraints, which are called assertions in Z3
 - Z3 uses Polish notation, which means that the operator comes before the operands
 - $x + y$ becomes $+ x y$

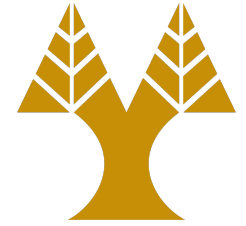
```
(assert (= z (+ x y)))
```

```
(assert (>= x 5))
```

```
(assert (= y2 (+ y z)))
```

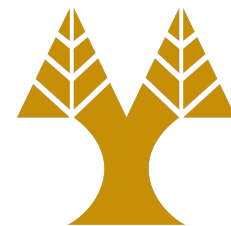
```
(assert (< y2 x))
```

Checking satisfiability and getting the model



```
(check-sat)
sat
(get-model)
(
  (define-fun y () Int
    (- 1))
  (define-fun x () Int
    5)
  (define-fun y2 () Int
    3)
  (define-fun z () Int
    4)
)
```

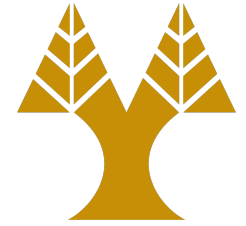
This means that the system of constraints is solvable (sat) and a solution for reaching baz() is $y = -1$ and $x = 5$.



Proving unreachability

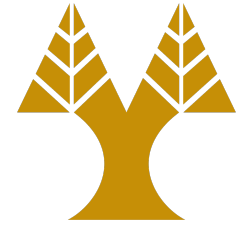
```
(declare-const x Int)
(declare-const y Int)
(declare-const z Int)
(declare-const y2 Int)
(assert (>= x 0))
(assert (>= y 0))
(assert (= z (+ x y)))
(assert (>= x 5))
(assert (= y2 (+ y z)))
(assert (< y2 x))
(check-sat)
unsat
```

Modeling constraints for machine code



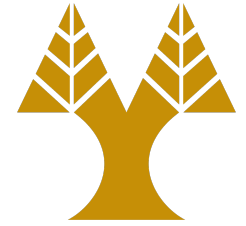
- Z3 uses mathematics and considers arbitrary precision of numbers
 - But binary code supports specific capacities (in bits) for arithmetic operations
- Z3 offers bitvectors, which are fixed-width integers
 - Z3 offers `bvadd`, `bvsub`, `bvmul`, etc., for performing arithmetic operations, instead of the typical `+`, `-` and `*`, etc.
- Z3 allows the definition of bitvectors
 - `(_ bv10 32)` creates a 32-bit bitvector equal to 10

Opaque predicate example



- Opaque predicates are expressions that will always evaluate to true or false
 - Although the outcome of this expression is known at priori, the expression is computed at run-time
- These are expressions that can be used to confuse the reverse engineer
 - The expression is connected with a branch that executes some (dead) code
 - The expression is known to always computer to false, but this is not visible

Example and solution with Z3



```
if (x + x*x) % 2 != 0) foo();
```

- This is false for any x, and therefore foo() will never be called
- However the expression will always be computed at run-time and serve as an obfuscation technique for the analyst

• Z3 solution

```
(declare-const x (_ BitVec 64))  
(assert (not (= (bvsmod (bvadd (bvmul x x) x) (_ bv2 64)) (_ bv0 64))))  
(check-sat)  
unsat
```