Advanced Cybersecurity Topics

–

Constraint Solving & Symbolic Execution

20-21

credits to: Marco Bonelli, Armando Solar-Lezama

Symbolic execution

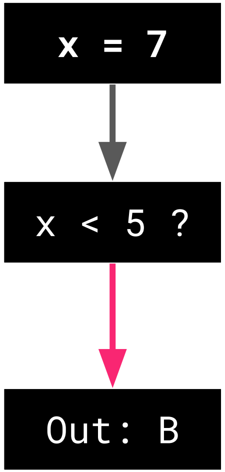
Symbolic execution is a powerful approach in binary analysis which consists of exploring all the possible paths a program can take by using **symbolic values** as input.

Execution starts normally, and whenever a decision whether to take branch or not has to be made, the current state of the program is **duplicated** and both branches are executed, keeping track of the **conditions** needed by each one.

Symbolic execution: example

```c
int main(void) {
  unsigned x;
  scanf("%d", &x);

  if (x < 5)
    puts("A");
  else
    puts("B");

  return 0;
}
```

Normal execution



Program is run with a fixed input.

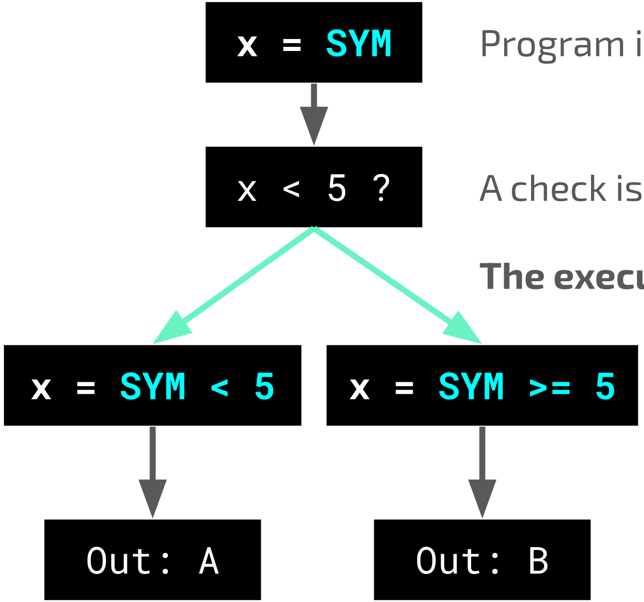
A check is made.

x >= 5, so the branch is not taken.

The output is "B".

Symbolic execution: example

```c
int main(void) {
  unsigned x;
  scanf("%d", &x);

  if (x < 5)
    puts("A");
  else
    puts("B");

  return 0;
}
```



Program is run with a **symbolic** input.

A check is made, and **detected**.

The execution is split in two.

Two different constraints are created for the input.

Now we know which input can lead us to A, and which to B.

Symbolic execution: example

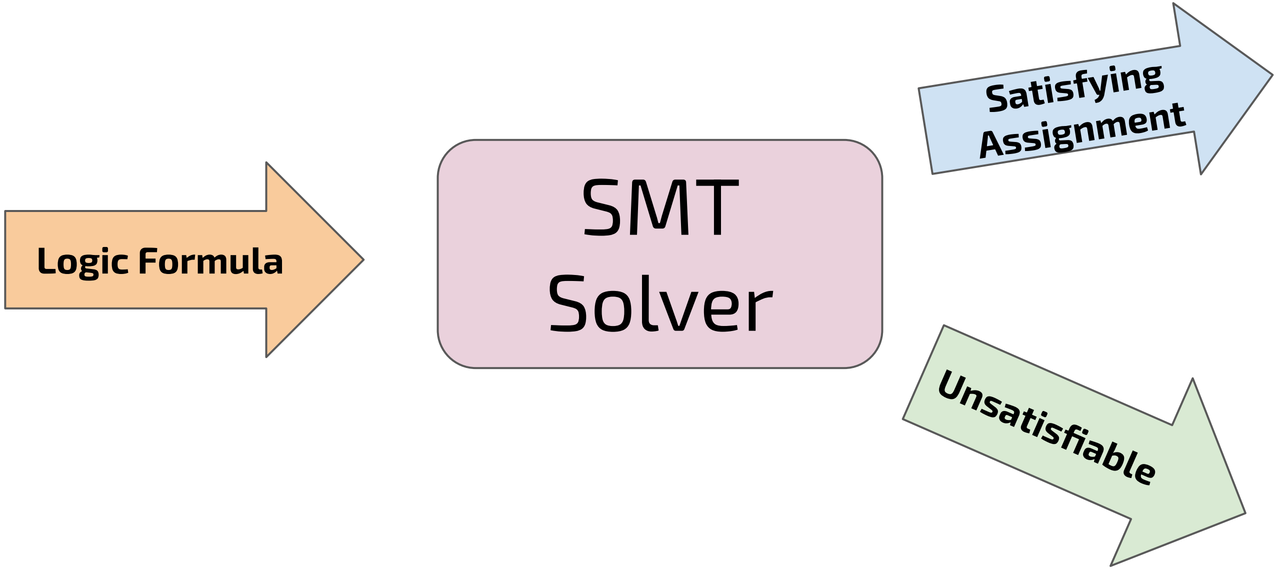
```c
int main(void) {
  unsigned x;
  scanf("%d", &x);

  if (x == 0xa5f4e321)
    puts("A");
  else
    puts("B");

  return 0;
}
```

Constraint programming

In short, "constraint programming" means we do **not** specify a **step** or sequence of steps to execute, **but** rather the **properties** of a solution to be found.

This technique is very useful in reverse-engineering: specific sets of constraints often need to be satisfied in order to "crack" a program.

SMT solvers

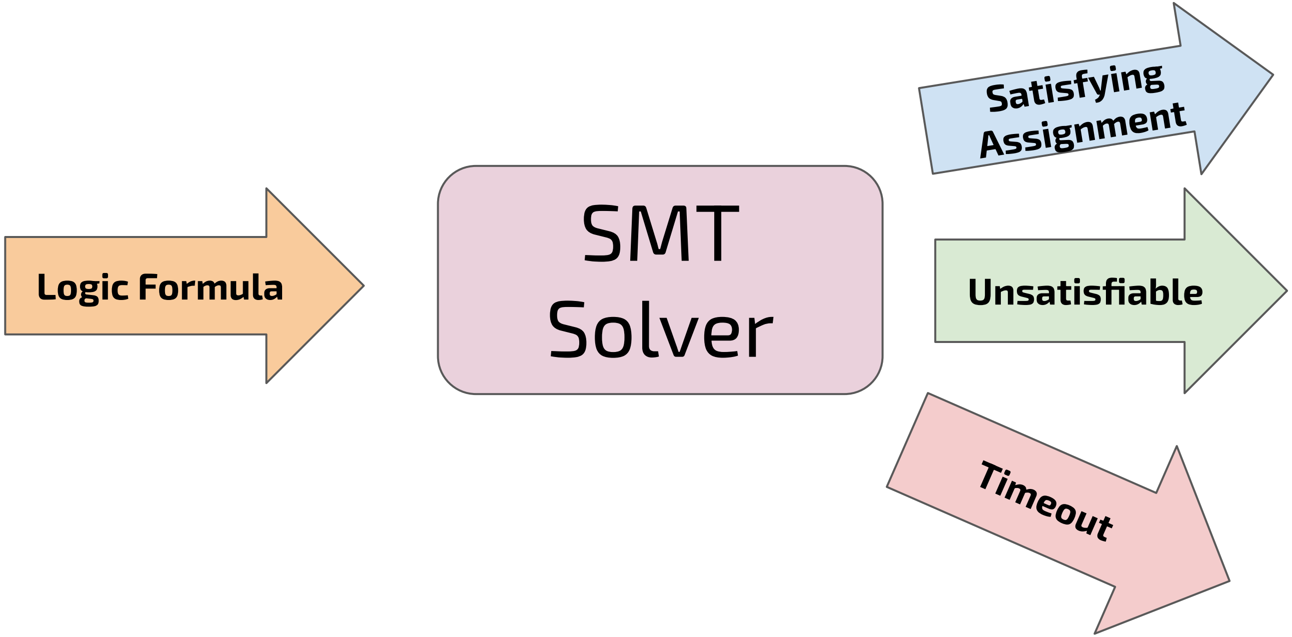

Logic Formula
SMT Solver
Satisfying Assignment
Unsatisfiable

SMT solvers

SMT (Satisfiability Modulo Theories) is a generalization of the boolean satisfiability problem. In **boolean logic** we only have two possible values (0, 1), in the SMT more complex values can be expressed as multiple boolean values.

An SMT solver is a program which can automatically determine if a certain **set of constraints** (expressed in first-order logic) is **satisfiable**, and if so, find the solution(s).

SMT solvers - NP Problem



SMT solver

Boolean SAT + **Theories**:

BitVector

Integer

Uninterpreted Function

Array

SAT Solver

Conjunctive Normal Form

$x1 \lor \neg x2 \lor x3 \lor \neg x4$

$x1 \lor x2$

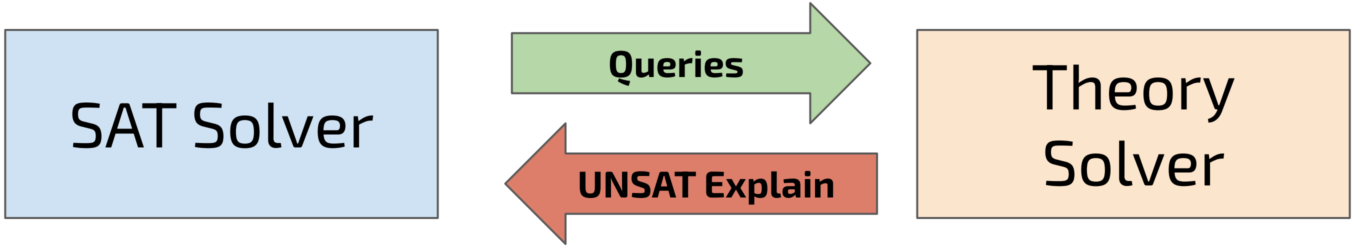
$x1 \lor x4 \lor x3$

SAT Solver + Theories = SMT

$$x > 5 \wedge y < 5 \wedge (y > x \vee y > 3)$$

$F1 \wedge F2 \wedge (F3 \vee F4)$

$x > 5, y < 5, y > x$



$\neg(F1 \wedge F2 \wedge F3)$

The Z3 SMT solver

Z3 is a powerful SMT theorem prover and solver.

We will take a look at its Python3 API and go through the main kind of problems it can solve:

- **Linear integer**/**real** arithmetic equations.
- **Non-linear integer/real** arithmetic equations.
- **Optimization** problems.

Z3: basic variable types

`z3.Int('x')`	Unbounded integer variable.
`z3.Real('x')`	Unbounded real variable.
`z3.Bool('x')`	Boolean variable.
`z3.BitVec('x', <length>)`	Bit vector: ordered sequence of bits.
`z3.FP('x', <type>)`	Floating point variable.
`z3.String('x')`	String variable (not really a classic string).

Z3: expressions

Every variable created with Z3 can be used *symbolically* to create expressions:

```
>>> x = z3.Int('x')
>>> y = z3.Int('y')

>>> x + y
x + y

>>> x - y < 4
x - y < 4
```

```
>>> type(x)
z3.z3.ArithRef

>>> type(x + y)
z3.z3.ArithRef

>>> type(x - y < 4)
z3.z3.BoolRef
```

Z3: constraint solver

A set of expressions (aka constraints) can be checked for satisfiability and, if satisfiable, evaluated:

```python
x = z3.Int('x')
y = z3.Int('y')
z = z3.Int('z')

solver = z3.Solver()
solver.add(x > y)
solver.add(y > z)
solver.add(z >= 3)
solver.add(z <= 5)
```

```python
>>> solver.check()
sat
>>> model = solver.model()
>>> model.eval(x)
5
>>> model.eval(y)
4
>>> model.eval(z)
3
```

Z3: simple non-linear equation system

$$
\begin{cases}
y = x^2 - 4 \\
y - z = 10 \\
x + 3y + z = -6
\end{cases}
$$

```python
x = z3.Real('x')
y = z3.Real('y')
z = z3.Real('z')

solver = z3.Solver()

solver.add(y == x**2 - 4)
solver.add(y - z == 10)
solver.add(x + y + z == -6)

solver.check()
m = solver.model()

print(m)
```

```
[
    y = 0.4713602229,
    x = 2.1145591083,
    z = -9.5286397770
]
```

Z3: simple optimization

$$\text{Constraints:} \begin{cases} x < 4 \\ y - x < 2 \end{cases}$$

$$\text{value}(x, y) = x + 2y$$

```python
x = z3.Real('x')
y = z3.Real('y')
value = z3.Real('value')

opt = z3.Optimize()

opt.add(x < 4)
opt.add(y - x < 2)
opt.add(value == x + 2*y)

opt.maximize(value)
opt.check()

print(opt.model())
```

```
[
    y = 4,
    x = 3,
    value = 11
]
```

Z3: conditional logic

```python
def f(a, b):
    if a < 1:
        return a
    if b > 1:
        return a + b
    return b
```

```python
def f(a, b):
    if2 = z3.If(b > 1, a + b, b)
    if1 = z3.If(a < 1, a, if2)
    return if1
```

```python
solver = z3.Solver()
x = z3.Int('x')
y = z3.Int('y')

solver.add(f(x, y) == 6)

solver.check()
print(solver.model())
```

```
[y = 5, x = 1]
```

Z3: useful resources

Here's some random links, in case you don't want to waste time Googling:

Introduction and some tutorials:

- https://github.com/ericpony/z3py-tutorial
- https://rise4fun.com/z3/tutorialcontent/guide
- https://theory.stanford.edu/~nikolaj/programmingz3.html

Z3py API reference:

https://z3prover.github.io/api/html/namespacez3py.html

angr: introduction

Angr is a **binary analysis framework** for both static analysis and symbolic execution. It's written in Python, which makes it simple to use

Python3

Check out angr.io for documentation, tutorials and working examples! Really, DO IT, there's much more to know...

angr: introduction

The **Project**: every time you work with angr, you'll need to create a "project". A project takes a binary and loads it, along with the needed libraries, and extracts basic information.

```
>>> proj = angr.Project('/bin/true')
>>> proj.arch
<Arch AMD64 (LE)>
>>> proj.entry
0x401670
>>> proj.filename
'/bin/true'
```

angr: introduction

The **SimulationManager**: once a project is created, we can create an instance of a sim. manager from it. The SM is the core of angr, and controls how the binary is executed and dynamically analyzed. The SM takes a ***state*** from which to start execution as argument (by default the entry point).

```
>>> sm = proj.factory.simgr()
>>> sm
<SimulationManager with 1 active>
>>> sm.active
[<SimState @ 0x401670>]
```

angr: introduction

The **SimState**: it's a class representing a simulation state, which is a snapshot of the state of the binary at a certain point of execution: **registers, processor flags, memory, etc.**

States can be created or modified to alter the program execution. They are organized by the SM in different **stashes**, with different meanings. The most important are the active stash and the deadended stash.

angr: introduction

Create a SimState (for example from the entry point):

```
>>> state = proj.factory.entry_state()
<SimState @ 0x401670>
```

Customize the state:

```
>>> state.regs.rbp = state.regs.rsp
>>> state.mem[0x1000].uint64_t = state.regs.rdx
>>> ...
```

angr: introduction

Before starting the execution, and whenever a branch is encountered, a new SimState is put in the ***active* stash**.

States in the active stash are **executed in "parallel"** (**one basic block** at the time), until they reach a dead end (for example return from main(), exit(), etc.).

When a state reaches a dead end and can no longer advance, it is put in the ***deadended* stash**.

angr: symbolic execution basics

The workflow when working with angr for symbolic execution is more or less always the same:

Create a **Project**.
Create or choose a **SimState** to start from.
Create a **SimulationManager** (pass the state to it).
Simulate using .step(), .explore() or .run().
Optional: ***monitor*** the stashes during the simulation.

angr: symbolic execution basics

SM.**step**(…)

Steps a stash of states (by default, the active stash) forward one basic block.

SM.**explore**(*n*=…, *find*=…, *avoid*=…, …)

Steps a stash at most *n* steps forward, avoiding addresses listed in *avoid*, until any of the addresses listed in *find* is found (*find* can also be a function which checks the current SimState and returns True or False).

SM.**run**(…)

Similar to explore(), but more rarely used.

angr: symbolic execution basics

To create symbolic inputs we use **claripy**, which is a library used by angr that inherits all basic types from Z3, plus has some more utilities.

```
argv = []
for i in range(10):
    x = claripy.BVS('arg_' + str(i), 8) # Symbolic BitVector
                                        # Same as z3.BitVector()
    argv.append(x)

state = proj.factory.entry_state(args=argv) # 10 args 1 byte each
sim   = proj.factory.simulation_manager(state)
# ...
```

angr: symbolic execution basics

Once all is set up, we can start simulating:

```python
sim = proj.factory.simulation_manager(initial_state)

while len(sim.active) > 0:
    print(sim, sim.active)

    # Simulate one block at a time (n=1)
    # Or until we find at least 1 address listed in find=...
    sim.explore(find=0x4007B6, n=1, num_find=1)

    if len(sim.found) > 0: # Check if we got where we wanted
        break
```

angr: state explosion

From `malloc.c`

```c
bck = victim->bk;
size = chunksize(victim);
mchunkptr next = chunk_at_offset(victim, size);

if ((size <= 2 * SIZE_SZ)
 || (size > av->system_mem))
   malloc_printerr("...");
if ((chunksize_nomask(next) < 2 * SIZE_SZ)
 || (chunksize_nomask(next) > av->system_mem))
   malloc_printerr("...");
if ((prev_size(next) & ~(SIZE_BITS)) != size)
   malloc_printerr("...");
if ((bck->fd != victim)
 || (victim->fd != unsorted_chunks(av)))
   malloc_printerr("...");
if (prev_inuse (next))
   malloc_printerr("...");
```

What happens if we are doing symbolic execution and our program does a call to a very complex library function like for example `malloc()`?

Many branches grow the stash of active simulation states exponentially! **Keeping track of all the active states becomes impossible.** The simulation becomes so slow that it never terminates (or even advances).

angr: state explosion

For common library functions, angr already handles it: it does not analyze their execution, instead it simulates it.

For anything else we need to be careful and control the flow of execution by specifying what to **avoid**.

```python
sim = proj.factory.simulation_manager(initial_state)
to_find  = [0x4007b6, 0x40103a, 0x400883]
to_avoid = [0x402033, 0x40076f]

sim.explore(find=to_find, avoid=to_avoid) # Avoid these addresses or we might
                                           # end up with too many states
if sim.found:
    print('Success!')
```

angr: constraints, input, output

We can apply symbolic values to a state in many different ways, but the most common are:

- Symbolic arguments
- Symbolic standard input
- Symbolic memory or registers

And for the output, usually we want to see what's on the standard output.

angr: constraints, input, output

Applying a symbolic value to the **arguments**:

```python
argv = ['./prog']
argv.append(claripy.BVS('arg1', 20*8)) # symbolic first argument

state = proj.factory.entry_state(args=argv)
simgr = proj.factory.simulation_manager(state)

simgr.explore(find=0xAAAAAA, avoid=0xBBBBBB) # explore...

if simgr.found:
    found = simgr.found[0]
    print(found.solver.eval(argv[1])) # eval
```

angr: constraints, input, output

Applying a symbolic value to **standard input**:

```python
chars = [claripy.BVS('c%d' % i, 8) for i in range(20)]    # 20 bytes
input_str = claripy.Concat(*chars + [claripy.BVV(b'\n')]) # + \n
initial_state = proj.factory.entry_state(stdin=input_str) # use as stdin

for c in chars: # make sure all chars are printable
    initial_state.solver.add(c >= 0x20, c <= 0x7e)

simgr = proj.factory.simulation_manager(initial_state)
simgr.explore(find=0xAAAAAA)

if simgr.found:
    print(simgr.found[0].posix.dumps(0)) # dump content of stdin
```

angr: constraints, input, output

Applying a symbolic value to **memory**:

```
initial_state = proj.factory.entry_state(addr=0xXXXXXX)

var = claripy.BVS('var', 20*8)                # 20 symbolic bytes
initial_state.memory.store(0x401337, var) # store in memory

simgr = proj.factory.simgr(initial_state)
simgr.explore(find=0xAAAAAA, avoid=0xBBBBBB) # explore...

if simgr.found:
    found = simgr.found[0]
    print(found.solver.eval(var).to_bytes(20, 'big')) # eval
```

angr: more advanced usage

Angr by default emulates ALL the memory of a process (including memory of libraries). Sometimes this can slow down the simulation, so if we don't need it, we can avoid it by setting ***auto_load_libs*** to **False**.

```
proj = angr.project.Project('./prog', auto_load_libs=False)
```

angr: more advanced usage

Sometimes we want to get some more control on the simulation, or maybe we want to check some register or value in memory at some point of simulation.

To do this, angr makes it possible to **hook certain addresses or symbols** with our own functions (also called `SimProcedures`).

angr: more advanced usage

We can hook a certain **address**, so when it is reached our function is executed:

```python
def myfunction(state):
    print('rax:', state.regs.rax) # print contents of some regs
    print('rbx:', state.regs.rbx)
    print('rcx:', state.regs.rcx)
    state.regs.rdx = claripy.BVV(0x1234, 64)  # set rdx = 0x1234

proj = angr.project.Project('./myprog')
proj.hook(0x401337, myfunction)
```

angr: more advanced usage

Or, we can create a `SimProcedure` and hook a **symbol**:

```
class FakeRand(angr.SimProcedure): # create a custom SimProc
    def run(self):
        # always return 42
        return self.state.solver.BVV(42, 64)

proj = angr.project.Project('./myprog')

# Hook it to the libc rand() function:
# NB: we need replace=True because angr already hooks rand
proj.hook_symbol('rand', FakeRand(), replace=True)
```