

Lesson 5: Constraint Solving and Symbolic Execution

We'll have some fun

Luca Di Bartolomeo Leonardo Galli

flagbot (CTF@VIS)

November 2, 2021



Table of Contents

Constraint Solving

General

Defining Variables

Defining the Domain

Defining Constraints

Solving for Constraints

Angr

Demo

Tips and tricks

Troubleshooting

Other tools

Challenge

Constraint Solving

General



Problem: Annoying Reverse Challenge

- ▶ Already reversed good amount of challenge
- ▶ Now you know what conditions every byte of flag must fulfill

```
char vals[] = {0xe2, 0x37, 0xcf, 0xe4, 0xc2, 0x3a, 0x42, 0x6c, 0x6e, 0x92,  
              0x5, 0x3a, 0xc5, 0xe6, 0xdf, 0x5c, 0x1f, 0x7, 0xe7, 0xd7, 0xd9, 0x1a,  
              0xc7, 0xda, 0x63, 0x70, 0x7b, 0xf1, 0xf0, 0xf7, 0xf6, 0xf5};  
int main(int argc, const char* argv[]) {  
    char input[32];  
    gets(input); // lets just imagine this removing newlines  
    for (int i = 0; i < 32; i++) {  
        char a = input[i] ^ (input[i] << 2);  
        char b = (input[i] - i) ^ (input[i] + 20);  
        if ((a ^ b) != vals[i]) return 1;  
    }  
    return 0;  
}
```



Solution: Constraint Solving

1. Define variables (usually input we control, in example `char input[32]`)
2. Define domain of variables (usually printable characters)
3. Define constraints, i.e. first-order logic formulas with equality (figured out by reversing)
4. Use a tool (such as z3) to solve for your variables



Time Complexity

How long does a solver theoretically take?



Time Complexity

How long does a solver theoretically take?

Running Time of Constraint Solvers

It is very similar to the SAT problem. It comes to no surprise, that it is an NP-Complete Problem as well! Theoretically, it would take exponential time to solve!

In practice, we have a small enough search space and independent parts. Additionally, specialized libraries have optimized code for solving these problems.



z3 Installation

- ▶ z3 does the heavy lifting of constraint solving for you
- ▶ usually you work with its python bindings, Z3Py
- ▶ installation should be easy via pip: `pip3 install z3-solver`
- ▶ **do not install z3!**



Constraint Solving

Defining Variables



Considerations

- ▶ similar to variables when programming, we need to specify the type
- ▶ usually, libraries support:
 - ▶ integers
 - ▶ real numbers
 - ▶ even functions!
- ▶ however, computers use neither integers or real numbers, but rather machine numbers
 - ▶ often called `BitVector`
 - ▶ allows you to specify how many bits your machine number should have
- ▶ usually, support for array types is either non existant or very limited
 - ▶ this also applies to strings!



Working with Arrays and Strings

How can we define an array?



Working with Arrays and Strings

How can we define an array?

- ▶ we define a sequence of variables
- ▶ since we will be scripting with python anyways, we can use arrays in python

Should we do the same for strings?



Working with Arrays and Strings

How can we define an array?

- ▶ we define a sequence of variables
- ▶ since we will be scripting with python anyways, we can use arrays in python

Should we do the same for strings?

- ▶ depends on the library, but usually yes (use python array of 8-bit BitVectors)
- ▶ for angr's implementation, it is usually more effective to define an $(8n)$ -BitVector for a string of length n



Defining Variables with Z3Py

```
import z3

x = z3.Int('x') # all variables need a name
y = z3.Real('y')

flag = []
for i in range(32): # we know flag is at most 32 chars
    flag.append(z3.BitVec(f'flag_{i}', 8)) # char is 8 bits
```



Constraint Solving

Defining the Domain



Considerations

- ▶ flag is always made out of printable characters:
 - ▶ special characters: `' !"#\$\%&\'()*+,-./'`, 32 (0x20) – 47 (0x2f),
`':;<=>?@'`, 58 (0x3a) – 64 (0x40),
`'[\]^_`'`, 91 (0x5b) – 96 (0x60),
`'{|}~'`, 123 (0x7b) – 126 (0x7e)
 - ▶ digits: `'0123456789'`, 48 (0x30) – 57 (0x39)
 - ▶ uppercase: `'ABCDEFGHIJKLMNOPQRSTUVWXYZ'`, 65 (0x41) – 90 (0x5a)
 - ▶ lowercase: `'abcdefghijklmnopqrstuvwxyz'`, 97 (0x61) – 122 (0x7a)
- ▶ try keeping your domain as small as possible!
- ▶ but, if exact length is unknown, some characters might be 0!
- ▶ other types can be restricted like normal
 - ▶ keep in mind - by default - numbers are signed!
 - ▶ i.e. $x < 100$ allows $x = -200$



Defining Domains with Z3Py

- ▶ Z3Py has no real concept of domains, instead we just add constraints!
- ▶ for this, we need a `Solver`
 - ▶ stores constraints on variables
 - ▶ will be used to solve these constraints
- ▶ for now, we just add a few constraints



Defining Domains with Z3Py

```
s = z3.Solver() # create our solver

s.add(x < 100) # allows x = -200!
s.add(y < 100)
s.add(y > -100)

for c in flag:
    s.add(c >= ' ') # space is first printable character
    s.add(c <= '~') # tilde is last printable character
```



Constraint Solving

Defining Constraints



Considerations

- ▶ when using BitVectors, there is no need for manual masking (e.g. `x & 0xff`, ensuring only 8 bits used)
- ▶ usually, individual constraints are ANDed together
 - ▶ if you need OR, create one constraint that is an OR of the individual constraints
- ▶ keep your constraint count as low as possible, while also ensuring constraints are as “tight” as possible
 - ▶ the less possible values your variables can take, the faster solving is
 - ▶ for example, constrain flag to flag format, i.e. `flag[:8] == 'flagbot'`
 - ▶ the more constraints to fulfill, the slower solving is
- ▶ when working with BitVectors, pay attention to signedness of operation
 - ▶ by default, operations are signed



Common Operations in Z3Py

- ▶ arithmetic operations (unsigned counterparts):

`+, -, *, / (UDiv), % (URem)`

- ▶ bitwise operations: `|, &, ^, ~`

- ▶ boolean operations:

`Or(a, b, ...), And(a, b, ...), Not(a), Xor(a, b), Implies(a, b)`

- ▶ comparison: `<= (ULE), < (ULT), > (UGT), >= (UGE), ==`

- ▶ shifts: `<<, >> (LShR), RotateLeft, RotateRight`

- ▶ concatenate multiple values (a will occupy bits starting at 0, b will follow after a, etc.): `Concat(a, b, ...)`

- ▶ extract bits from BitVector: `Extract(high, low, val)`

See [Official Z3Py Documentation](#) for more!



Defining Constrains in Z3Py

```
for c in flag: # change our domain to allow 0
    s.add(z3.Or(c >= ord(' '), c == 0), c <= ord('~'))
# if one character is null, all following must be as well!
for i in range(len(flag)-1):
    s.add(z3.Implies(flag[i] == 0, flag[i+1] == 0))

z = z3.Int('z') # find prime smaller than 100
s.add(z3.ForAll([z], z3.Implies(z3.And(1 < z, z < x), x % z != 0)), 1 < x)
s.add(z3.ForAll([z], z3.Implies(z3.And(1 < z, z < y),
    z3.ToInt(y) % z != 0)), 1 < y)
vals = [0xe2, ..., 0xf5] # values extracted via reversing
for i, c in enumerate(flag): # add actual constraints
    a = c ^ (c << 2)
    b = (c - i) ^ (c + 20)
    s.add(a ^ b == vals[i])
```



Constraint Solving

Solving for Constraints



Solving with Z3Py

Depends a lot on your library!

```
print(s.check()) # check() tries to find values satisfying all constraints
# prints 'sat' if values found, 'unsat' if not
print(s.model()) # model() gives you the actual values
# prints [flag_23 = 97,
# ...
# flag_19 = 102]
print("".join([chr(s.model().eval(c).as_long()) for c in flag]))
# prints 'flagbot{z3_makes_life_easy}\x00\x00\x00\x00\x00'
```



Angr



Installation

▶ `mkvirtualenv angr`

▶ `pip install angr`

OR

▶ `docker run -it angr/angr`



What are we talking about

- ▶ **Claripy** - a data abstraction library
- ▶ **angr** - a concolic execution engine



What are we talking about

- ▶ **Claripy** - a data abstraction library
- ▶ **angr** - a concolic execution engine

Around 100k lines of python





UC Santa Barbara + Arizona State University





UC Santa Barbara + Arizona State University

For the DARPA Cyber Grand challenge



What does concolic mean

*“Concolic testing (a portmanteau of concrete and symbolic) is a hybrid software verification technique that performs **symbolic execution**, a classical technique that treats program variables as symbolic variables, along a **concrete execution** (testing on particular inputs) path”*


Wikipedia



What does concolic mean

An Illustrative Example

```
int foo(int v) {  
    return 2*v;  
}  
  
void test_me(int x, int y) {  
    int z = foo(y);  
    if (z == x)  
        if (x > y+10)  
            ERROR;  
}
```



Concrete Execution		Symbolic Execution
concrete state	symbolic state	path condition
$x = 2$	$x = x_\theta$	$2*y_\theta == x_\theta$
$y = 1$	$y = y_\theta$	
$z = 2$	$z = 2*y_\theta$	$x_\theta \leq y_\theta + 10$
Solve: $(2*y_\theta == x_\theta)$ and $(x_\theta > y_\theta + 10)$		



What does concolic mean

Symbolic execution

For each basic block, calculate all possible successors and all constraints necessary to get to a given successor

Full control over the execution

Quite slow

Concrete execution

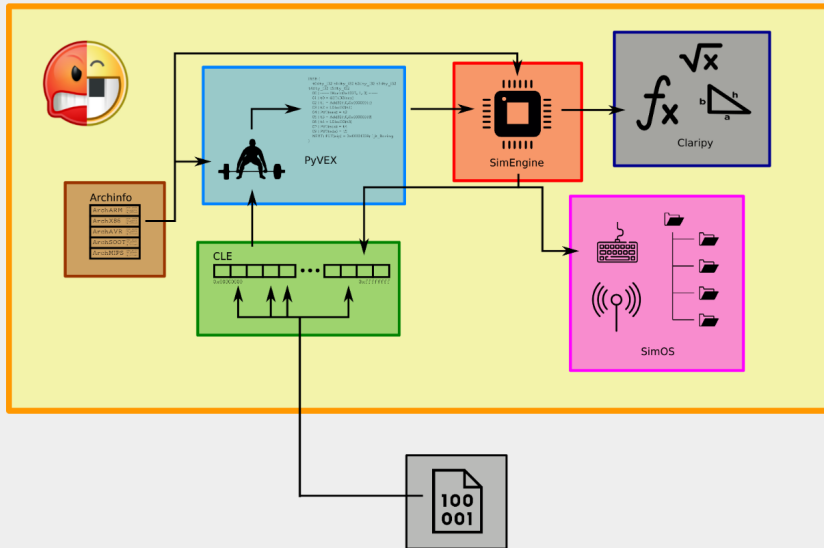
For each basic block, just execute it with your own damn CPU

Same execution control you would have with a debugger

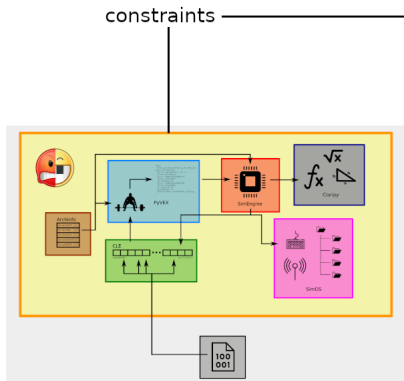
Many orders of magnitude faster



Overview



Actually, it's more like this



Z3

Documentation

Angr's documentation is like every cool recent state-of-the-art infosec tool



Documentation

Angr's documentation is like every cool recent state-of-the-art infosec tool

it is basically non-existent



Documentation

Angr's documentation is like every cool recent state-of-the-art infosec tool

it is basically non-existent

Your best bet is to have a look at what is *pretending* to be the official documentation and a set of examples they provide on the Angr website:

- ▶ <https://docs.angr.io/>
- ▶ <https://docs.angr.io/examples>



Documentation

Angr's documentation is like every cool recent state-of-the-art infosec tool

it is basically non-existent

Your best bet is to have a look at what is *pretending* to be the official documentation and a set of examples they provide on the angr website:

- ▶ <https://docs.angr.io/>
- ▶ <https://docs.angr.io/examples>

And here again, you will find yourself having to look at the source code to understand how stuff works. Only this time it's Python, not C, so maybe it's a little better, I guess? Not sure though, honestly.



Let's get our hands dirty

```
import angr
import claripy

project = angr.Project("./crackme") # load a binary

# This alone will take from 3 to 10 seconds
# If you think this is slow, oh boy, are you gonna change your mind
```



Let's get our hands dirty

```
import angr
import claripy

project = angr.Project("./crackme")
flag = claripy.BVS("flag", 8*100) # create a symbolic value

# first argument: name (does not really concern you)
# second argument: size in BITS (so here we have 100 chars)

# You can also use claripy.BVV() instead for a concrete (fixed) value
```



Let's get our hands dirty

```
import angr
import claripy

project = angr.Project("./crackme")
flag = claripy.BVS("flag", 8*50)
state = project.factory.full_init_state(stdin=flag)

# Here, we create an initial "state". There are many ways to do this:
# - full_init_state : quickly go over loading libs and go to main
# - entry_state      : bare-bones state corresponding to binary entry point
# - blank_state      : void state. Set starting address yourself.
```



Let's get our hands dirty

```
import angr
import claripy

project = angr.Project("./crackme")
flag = claripy.BVS("flag", 8*50)
state = project.factory.full_init_state(stdin=flag)
sm = project.factory.simulation_manager(state)
sm.explore(find=good_address, avoid=bad_address)

# Now, go and try to find desirable states!
# Arguments to 'find' and 'avoid' can be single addresses,
# lists of addresses or predicates on states

# A state can be in one of the following stashes:
# found - active - avoid - unsat - errored - deadended - unconstrained
```



Let's get our hands dirty

```
import angr
import claripy

project = angr.Project("./crackme")
flag = claripy.BVS("flag", 8*50)
state = project.factory.full_init_state(stdin=flag)
sm = project.factory.simulation_manager(state)
sm.explore(find=good_address, avoid=bad_address)
print (sm.found[0].solver.eval_upto(flag, 4, cast_to=bytes))

# Having found one (or more?) "good" states, we tell z3 to solve the
# constraints and give us up to 4 possible valid values for the
# "flag" symbolic variable
```



Demo



Demo

Demo time!



Angr limitations

- ▶ Path explosion
- ▶ Single-threaded
- ▶ It cannot cheat complex algos (e.g. crypto)
- ▶ You actually need to reverse part of the binary



Tips and tricks



Trick of the trade no. 1

Use PYPY!

```
pypy -m ensurepip  
pypy -m pip install angr
```



Trick of the trade no. 1

Use PYPY!

```
pypy -m ensurepip  
pypy -m pip install angr
```

Depends on the case, but in my experience it gets you a 2x-8x speedup



Trick of the trade no. 2

Give UNICORN a go!

```
state = project.factory.blank_state(add_options=angr.options.unicorn)
```



Trick of the trade no. 2

Give UNICORN a go!

```
state = project.factory.blank_state(add_options=angr.options.unicorn)
```

If you have to do a lot of concrete execution, this helps a lot



Trick of the trade no. 3

You can load COREDUMPS in angr!

```
proj = angr.Project("./coredump")
```



Trick of the trade no. 4

Symbolize ARBITRARY memory!

```
flag = claripy.BVS("flag", 8*8)
state.memory.store(flag, 0x800000)
```



Trick of the trade no. 5

Keep track of REGISTERS!

```
def lol(lsm):  
    print(lsm.active[0].regs.rip)  
  
sm.explore(find=address, avoid=address, step_func=lol)
```



Trick of the trade no. 6

Use symbolic ARGUMENTS!

```
argv = [project.filename]  
argv.append(sym_arg)  
state = project.factory.entry_state(args=argv)
```



Trick of the trade no. 7

Impose your own CONSTRAINTS!

```
flag = claripy.BVS("flag", 8*100)
for byte in flag.chop(8):
    state.add_constraints(byte >= '\x20') # ' '
    state.add_constraints(byte <= '\x7e') # '~'
```



Trick of the trade no. 8

Implement stuff YOURSELF!

```
class fixpid(angr.SimProcedure):  
    def run(self):  
        return 0x30  
  
project.hook(0x4008cd, fixpid())
```



Trick of the trade no. 9

Tell angr's warnings to SHUT THE FUCK UP!

```
state = project.factory.blank_state(  
    add_options={angr.options.ZERO_FILL_UNCONSTRAINED_MEMORY,  
                angr.options.ZERO_FILL_UNCONSTRAINED_REGISTERS}))
```



Trick of the trade no. 9

Tell angr's warnings to SHUT THE FUCK UP!

```
state = project.factory.blank_state(  
    add_options={angr.options.ZERO_FILL_UNCONSTRAINED_MEMORY,  
                angr.options.ZERO_FILL_UNCONSTRAINED_REGISTERS}))
```

- ▶ Actually useful in some cases, not just to make the output less annoying!



Tell angr's warnings to SHUT THE FUCK UP!

```
state = project.factory.blank_state(  
    add_options={angr.options.ZERO_FILL_UNCONSTRAINED_MEMORY,  
                angr.options.ZERO_FILL_UNCONSTRAINED_REGISTERS}))
```

- ▶ Actually useful in some cases, not just to make the output less annoying!
- ▶ Usually, we can expect memory and registers to be zeroed initially. Being certain about it helps prevent path explosion (and generally makes things easier for angr)



Tell angr's warnings to SHUT THE FUCK UP!

```
state = project.factory.blank_state(  
    add_options={angr.options.ZERO_FILL_UNCONSTRAINED_MEMORY,  
                angr.options.ZERO_FILL_UNCONSTRAINED_REGISTERS}))
```

- ▶ Actually useful in some cases, not just to make the output less annoying!
- ▶ Usually, we can expect memory and registers to be zeroed initially. Being certain about it helps prevent path explosion (and generally makes things easier for angr)
- ▶ Some library functions that initialize memory to zero, such as `explicit_bzero()`, aren't recognized by angr at the time of writing



Trick of the trade no. 10

Be LAZY!

```
state = ...  
state.options |= {LAZY_SOLVES}  
# you can also use the 'add_options' argument when creating the state
```



Trick of the trade no. 10

Be LAZY!

```
state = ...  
state.options |= {LAZY_SOLVES}  
# you can also use the 'add_options' argument when creating the state
```

- ▶ By default, angr runs z3 to check states for **satisfiability** at every simulation step.
 - ▶ Might be good to avoid explosion by quickly throwing out impossible states
 - ▶ but can be super slow



Trick of the trade no. 10

Be LAZY!

```
state = ...  
state.options |= {LAZY_SOLVES}  
# you can also use the 'add_options' argument when creating the state
```

- ▶ By default, angr runs z3 to check states for **satisfiability** at every simulation step.
 - ▶ Might be good to avoid explosion by quickly throwing out impossible states
 - ▶ but can be super slow
- ▶ `angr.sim_options.LAZY_SOLVES` **defers** checking satisfiability
“until absolutely necessary” [<https://docs.angr.io/appendix/options>]
 - ▶ can speed up execution by 10x, maybe even more!



Trick of the trade no. 10

Be LAZY!

```
state = ...  
state.options |= {LAZY_SOLVES}  
# you can also use the 'add_options' argument when creating the state
```

- ▶ By default, angr runs z3 to check states for **satisfiability** at every simulation step.
 - ▶ Might be good to avoid explosion by quickly throwing out impossible states
 - ▶ but can be super slow
- ▶ `angr.sim_options.LAZY_SOLVES` **defers** checking satisfiability “until absolutely necessary” [<https://docs.angr.io/appendix/options>]
 - ▶ can speed up execution by 10x, maybe even more!
 - ▶ works well if a “good” path (along which to gather constraints) is easy to find and “bad” branches are easy to **avoid**
 - ▶ probably a bad idea if control flow is obfuscated (branches that are never taken, etc.)



Troubleshooting



Dealing with Symbolic Strings

- ▶ angr's SimProcedures of string functions such as `strlen` assume symbolic strings to be **at most 60 bytes long** by default
- ▶ If a string needs to be longer than that, or you specifically constrain it to be longer than 60, you get an **unsatisfiable** state :(



Dealing with Symbolic Strings

- ▶ angr's SimProcedures of string functions such as `strlen` assume symbolic strings to be **at most 60 bytes long** by default
- ▶ If a string needs to be longer than that, or you specifically constrain it to be longer than 60, you get an **unsatisfiable** state :(
- ▶ Easy fix:

```
buf_size = 128
flag = claripy.BVS("flag", 8*buf_size)

state = ...
state.libc.buf_symbolic_bytes = buf_size
state.libc.max_str_len = buf_size
# might want to use max() instead
# to make sure you're not making anything smaller
```



Unsatisfiable States

- ▶ A state becomes unsatisfiable when its constraints **contradict** each other
 - ▶ informally: “this cannot possibly happen on a machine”



Unsatisfiable States

- ▶ A state becomes unsatisfiable when its constraints **contradict** each other
 - ▶ informally: “this cannot possibly happen on a machine”
 - ▶ constraints may be path constraints derived from control flow, or ones you manually added

```
void foo(int x) {  
    if (x == 42) {  
        puts("forty-two");  
        if (x == 43) {  
            // states here are always unsat  
            // (unless you manually mess with them)  
            puts("this is fine");  
            puts("absolutely no bitflips from cosmic radiation");  
            system("sudo rm -rf /");  
        }  
    }  
}
```



Debugging Unsatisfiable States

- ▶ You can access a state's constraints (e.g. in a Python debugger) using `state.solver.constraints`



Debugging Unsatisfiable States

- ▶ You can access a state's constraints (e.g. in a Python debugger) using `state.solver.constraints`
 - ▶ but that may be a lot of constraints, so which ones are actually wrong?



Debugging Unsatisfiable States

- ▶ You can access a state's constraints (e.g. in a Python debugger) using `state.solver.constraints`
 - ▶ but that may be a lot of constraints, so which ones are actually wrong?
- ▶ `unsat_core()` gives you a **subset of contradicting constraints**
 - ▶ actually how I found out about the issue with symbolic string lengths from slide 52



Debugging Unsatisfiable States

- ▶ You can access a state's constraints (e.g. in a Python debugger) using `state.solver.constraints`
 - ▶ but that may be a lot of constraints, so which ones are actually wrong?
- ▶ `unsat_core()` gives you a **subset of contradicting constraints**
 - ▶ actually how I found out about the issue with symbolic string lengths from slide 52
 - ▶ need to enable an option to use:

```
state = ...
state.options |= {angr.sim_options.CONSTRAINT_TRACKING_IN_SOLVER}
# you can also use 'add_options'

sm = project.factory.simulation_manager(state)
sm.explore(...)

# assume you have an unsat state that *should* be satisfiable
print("one of these is false: ", sm.unsat[0].solver.unsat_core())
```



Other tools



Other tools

- ▶ Triton
- ▶ KLEE
- ▶ S2E



Other tools

- ▶ Triton
- ▶ KLEE
- ▶ S2E

- ▶ Less suitable for a quick hack
- ▶ More stable; more documented
- ▶ Actually used by many companies → will probably be supported for a long time

Exception: Manticore



Challenge



Challenge

Angry bomb

This one's easy - it's the famous reversing bomb, 6 stages (or more?) of pure fun disarming. But wait, it's with a twist! Now you actually need to solve each phase with angr. No manual reversing allowed!

Hints: Whenever you feel really angry – scream, it will help. Source: am Italian

Files: bomb.zip

Author: CMU Labs

