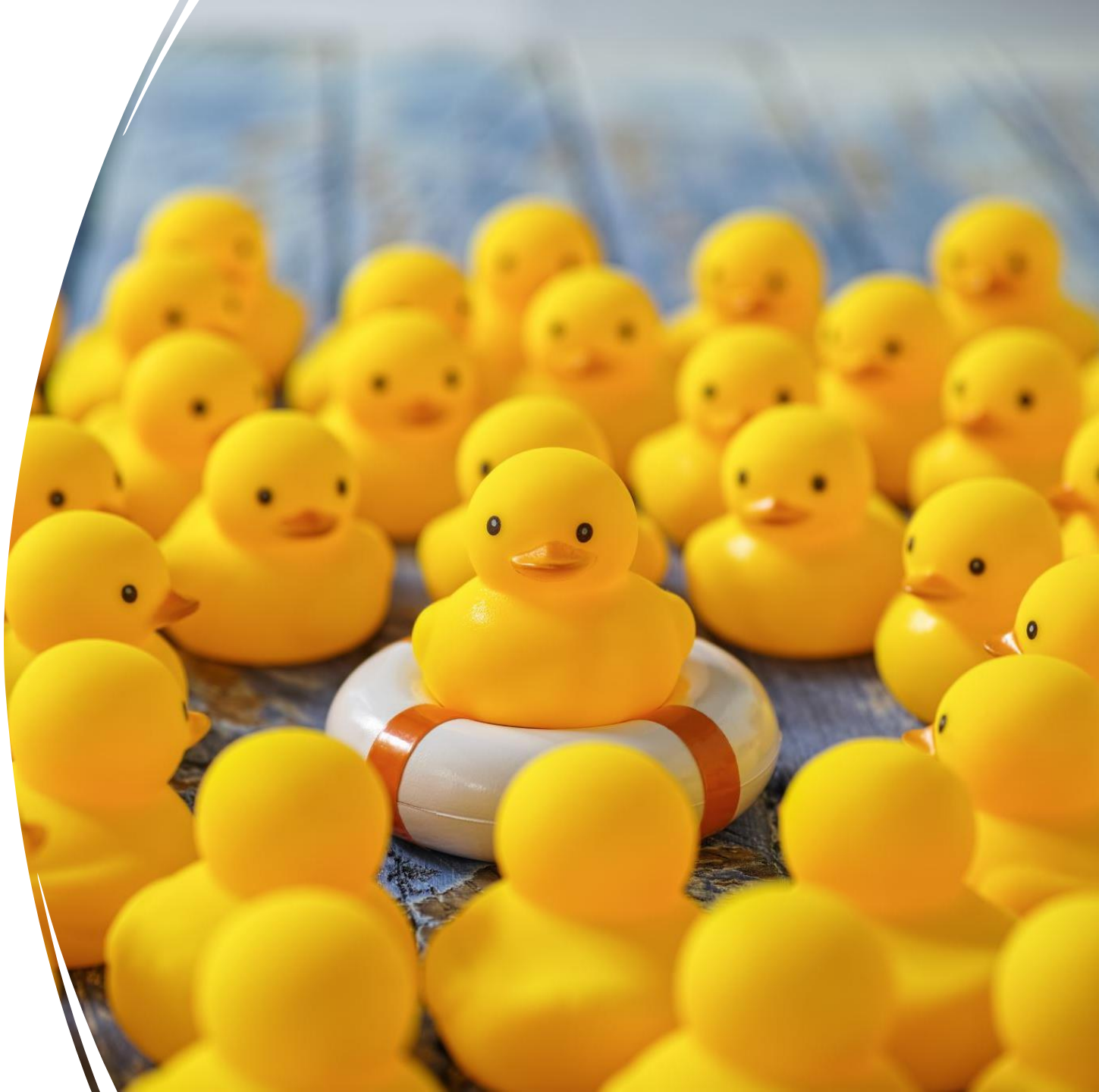
Z3

Constraint solver

Introduction

- Constraint solver
- Created by Microsoft Research
- Can use the specialized SMTLIB language, but also bindings for most popular programming languages
  - I will use Python, as is the most common among CTF-players
- Give it a set of variables and constraints, and it will try to find values for the variables that satisfy the given constraints.



Constraint problem

Consists of

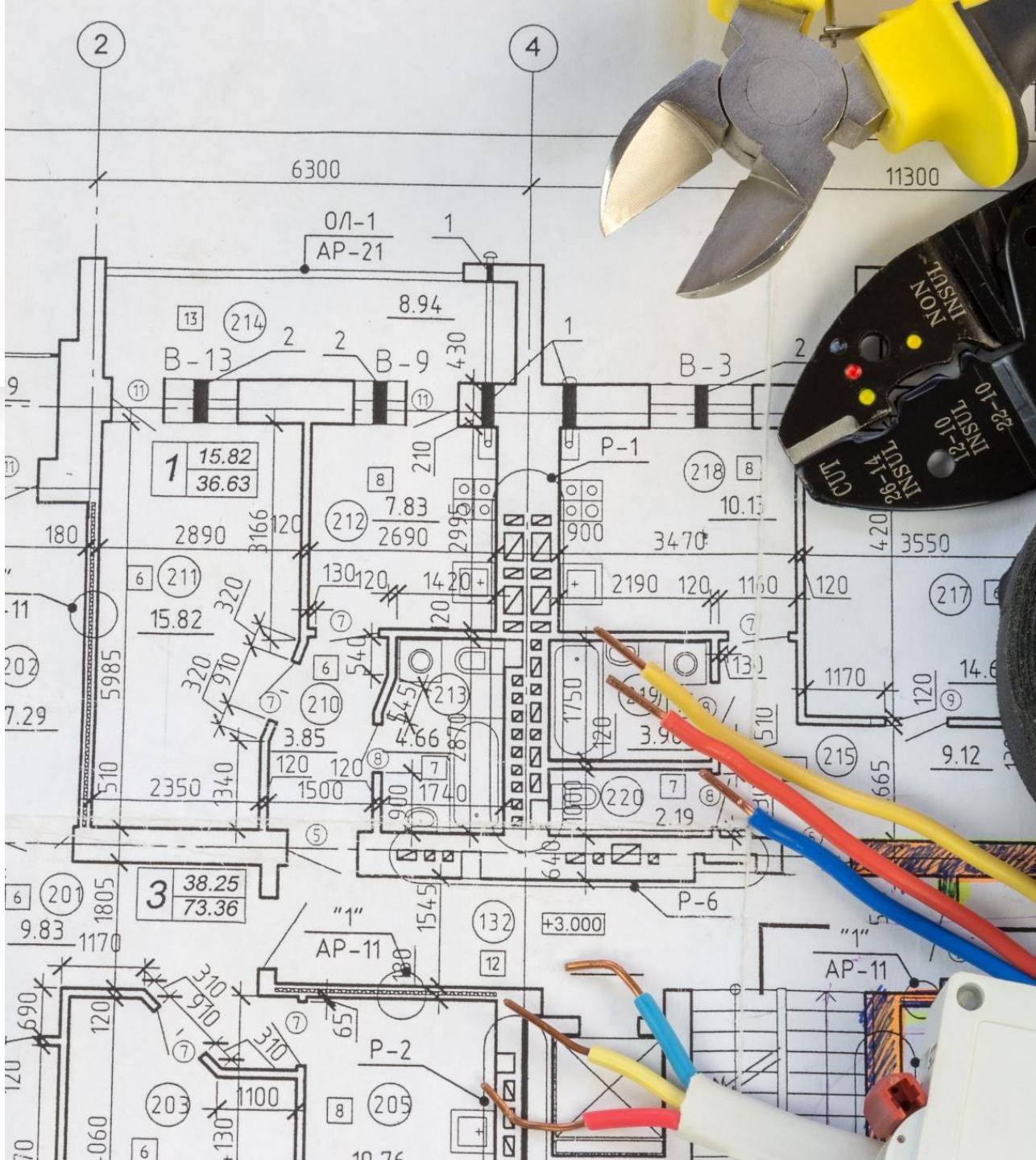
- Variables
- Constraints on these variables

Examples

- Mathematical
  - Set of equations/unequalities
- Logical puzzles
- Sudoku
- Scheduling
- In general any problem in NP can be redused to CSP

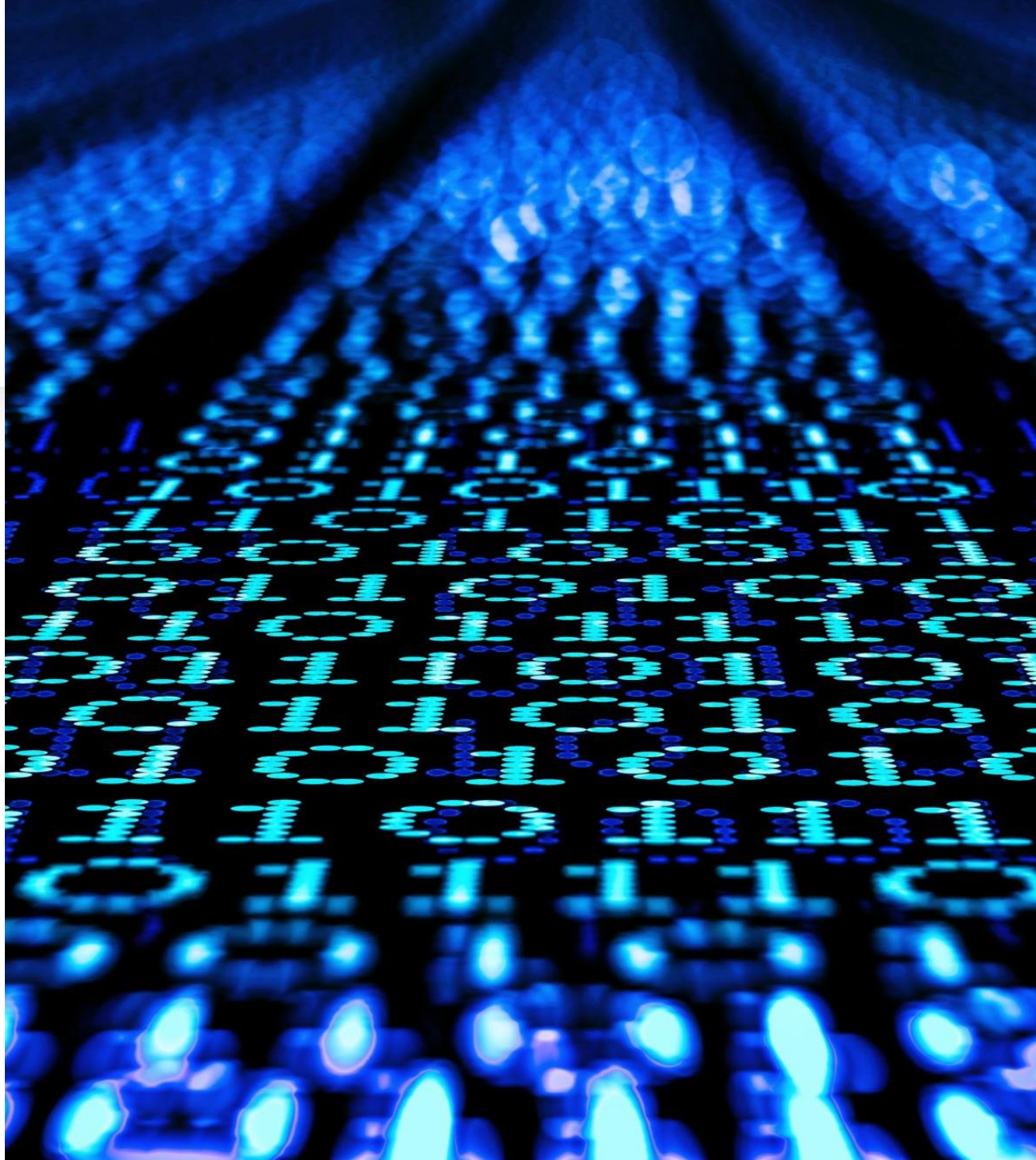
Installing

- Kali
  - sudo apt install z3
    - Or download: https://github.com/Z3Prover/z3/releases
  - pip install z3-solver



Datatypes

- Boolean
- Int
- Real
- BitVector
- Functions
- Strings



The setup of a Z3 program

- Setup variables
- Setup constraints
- Ask the solver to satisfy the constraints

- solver.check()
  - sat = it has a solution
  - unsat = it could not find a solution

```python
import z3

solver = z3.Solver()
x = z3.Int('x')
y = z3.Int('y')
solver.add(x + y == 10)
solver.add(x * y == 24)
print(solver.check())
print(solver.model())
```

```
PROBLEMS 1    OUTPUT    DEBUG CONSOLE

└─$ python3 test.py
sat
[x = 4, y = 6]
```

Int

- Arbitrary precision integer
- Good for arithmetic operarions
- Int("name")

BitVec

- Fixed size
- Good for bit operations
- BitVec("name", size) for symbolic
- BitVecVal(value, size) for non-symbolic
- Important to choose correct size!

```
>>> z3.Int("a") ^ 5
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    z3.Int("a") ^ 5
    ~~~~~~~~~~~~^~~
TypeError: unsupported operand type(s) for ^: 'ArithRef' and 'int'
>>> z3.BitVec("a", 8) ^ 5
a ^ 5
>>> z3.Int("a") + 1000
a + 1000
>>> z3.BitVec("a", 8) + 1000
a + 232
```

Operators

Logical

- And
- Or
- Not

Sequences

- Extract / Substring
- Length
- Contains

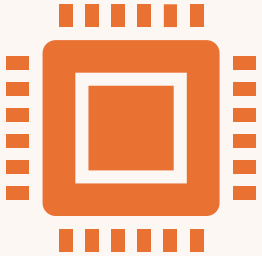
Bitwise

- Xor
- LshR
- RotateLeft
- RotateRight

Einstein's puzzle

- There are five houses.
- The Englishman lives in the red house.
- The Spaniard owns the dog.
- Coffee is drunk in the green house.
- The Ukrainian drinks tea.
- The green house is immediately to the right of the ivory house.
- The Old Gold smoker owns snails.
- Kools are smoked in the yellow house.
- Milk is drunk in the middle house.
- The Norwegian lives in the first house.
- The man who smokes Chesterfields lives in the house next to the man with the fox.
- Kools are smoked in the house next to the house where the horse is kept.
- The Lucky Strike smoker drinks orange juice.
- The Japanese smokes Parliaments.
- The Norwegian lives next to the blue house.
- Who owns the Zebra?

Pros and cons (personal)



Pros:

Easy to set up, especially in Python

Practical when there is no simple algorithm to solve a problem

- Typically many different operations mixed

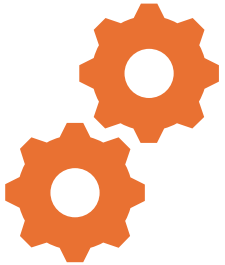
Cons:

Easy to make small mistakes that are difficult to find.

I never know how long it will take for Z3 to solve it (or if it is able to solve it at all)

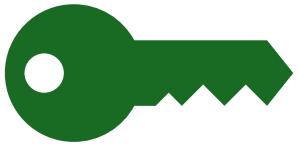
- Constraint solving is NP-hard in general, so Z3 cannot give any general guarantees

Use-cases



Reverse engineering

Trying to satisfy the constraints that the program requires.



Cryptography

Is good on things that are on bit-level, like XOR.

Worse when it comes to arithmetic on big numbers (but it sometimes works there as well!)

Dealing with floating point numbers

- Very practical for solving PRNGs such as xorshift128+ and Mersienne twister
  - In these, random states are generated by bit-operations
  - But random.random() converts these bit-states to floating point numbers
- fpBVToFP converts bitvectors to floating points

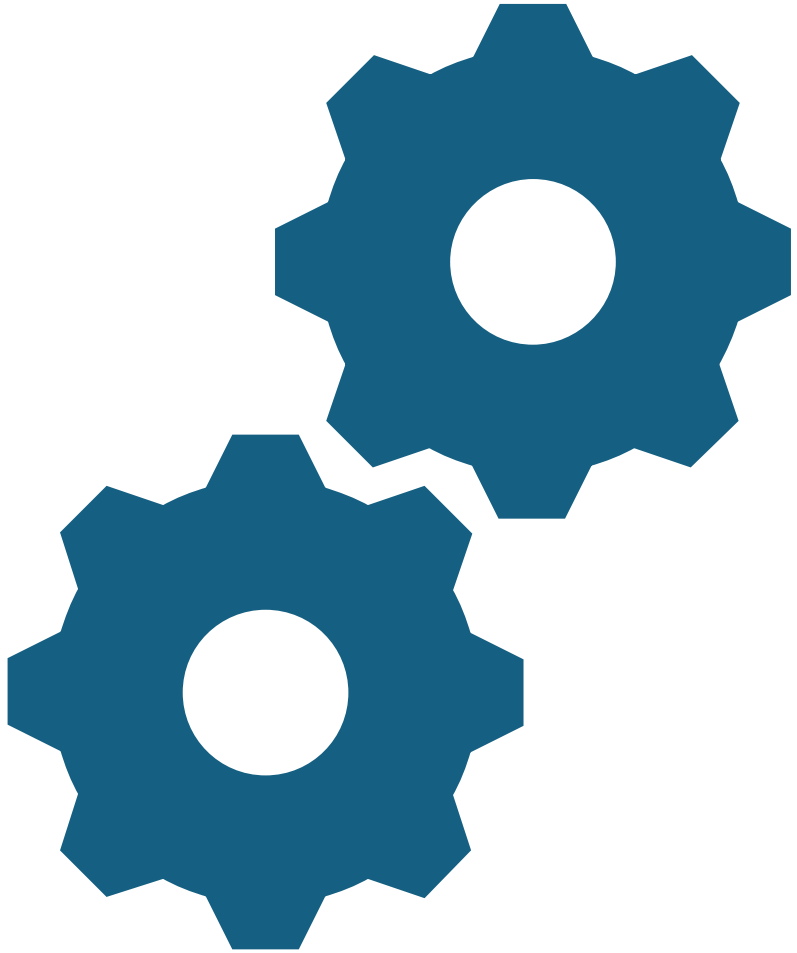
Exercises:

- Create a sudoku-solver using z3
- Solve "example_task" using z3
- Solve "promise.html" using z3 (challenge!)
- Write your own Z3-challenge

Z3 too tedious?

Automate constraint solving for binaries

Angr



Symbolic execution

- Program execution based on symbolic variables rather than actual imput values.

```
1   x = Int()
2
3   if  x < 10:
4       print("x < 10")
5   else:
6       print("x >= 10")
```

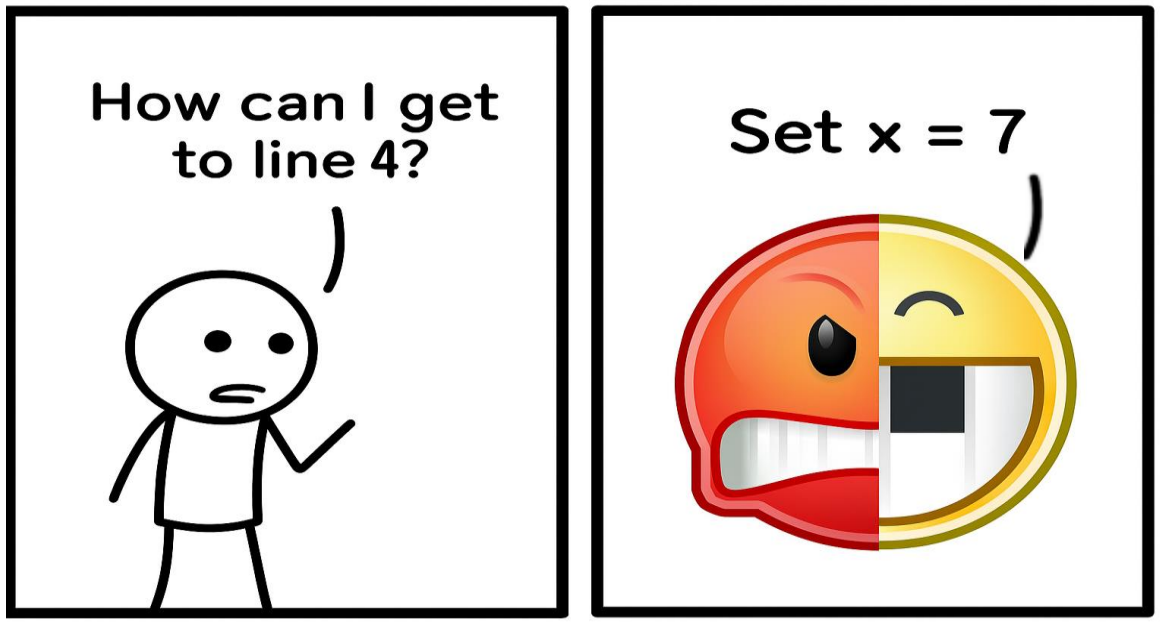
<- x as symbolic variable

<- State: constraint: x < 10
program counter: 4

<- State: constraint: x >= 10
program counter: 6

- Instead of choosing one of the two branches, like normal execution, it instead creates one state for each branch, with constraints on symbolic variables necessary for taking said branch.

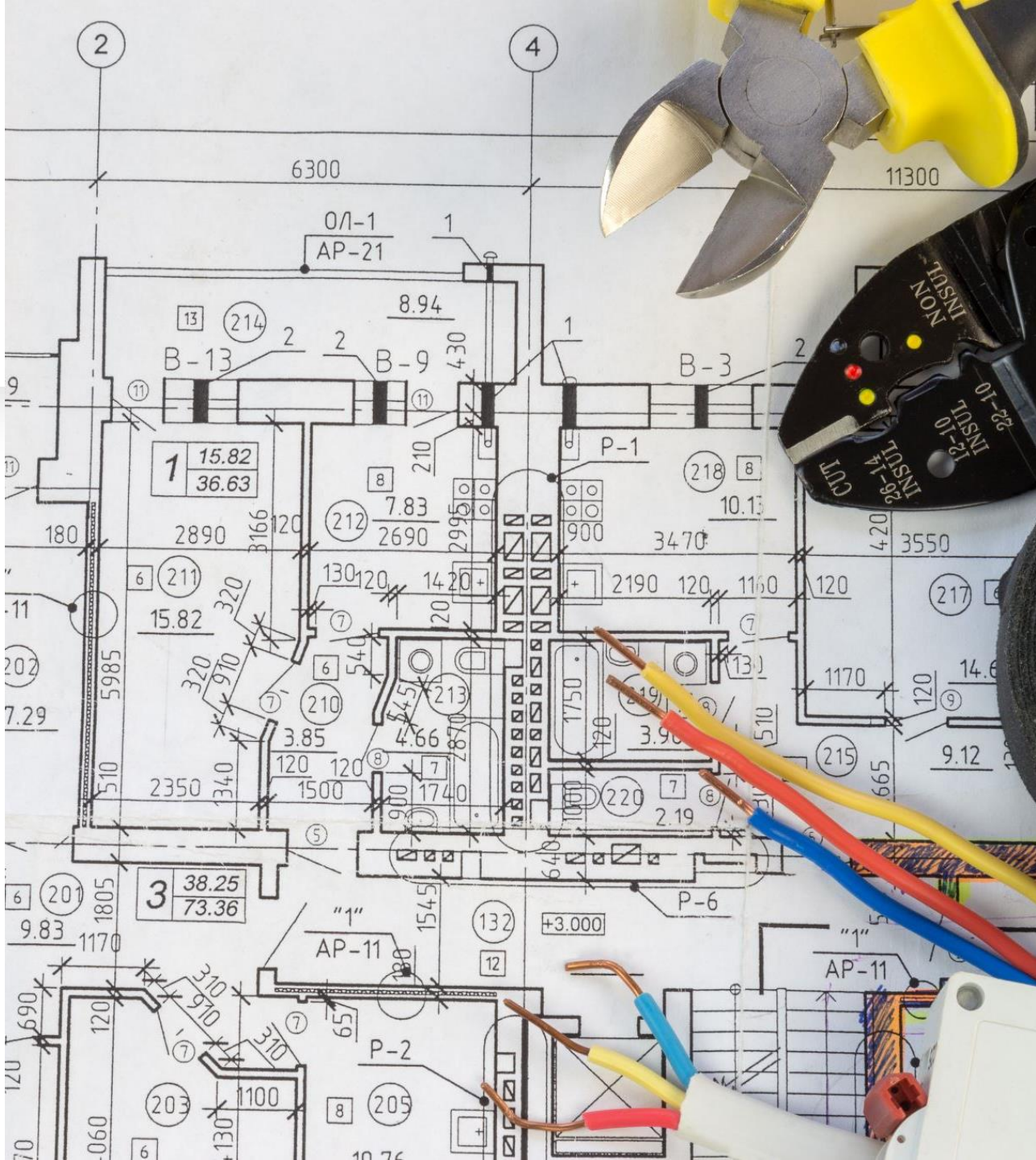
So where does angr come into play?




```
1    x = Int()
2
3    if  x < 10:
4        print("x < 10")
5    else:
6        print("x >= 10")
```

Installing

- pip install angr

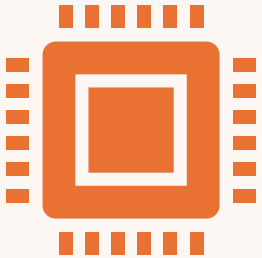


Simple example

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main() {
    puts("Write the password:");
    char *buf = malloc(20);
    fgets(buf, 20, stdin);
    size_t len = strlen(buf);
    if (len > 0 && buf[len - 1] == '\n') {
        buf[len - 1] = '\0';
    }
    if (strcmp(buf, "S3cretPassw0rd") == 0) {
        puts("Access granted.");
    } else {
        puts("Access denied.");
    }
    free(buf);
    return 0;
}
```

```python
import angr

project = angr.Project("./simple")

state = project.factory.entry_state()

simgr = project.factory.simgr(state)

def is_success(state):
    return b"Access granted." in state.posix.dumps(1)

def is_failure(state):
    return b"Access denied." in state.posix.dumps(1)

simgr.explore(find=is_success, avoid=is_failure)

if simgr.found:
    found_state = simgr.found[0]
    solution = found_state.posix.dumps(0)
    print("Recovered password:", solution.decode('utf-8', errors='ignore'))
else:
    print("Failed to find a valid password.")
```

Pros and cons (personal)



Pros:

Could save you a lot of work and effort

Practical when there are are a lot of branches

- But the number of states cannot grow too fast

Cons:

Difficult to know in beforehand if it is able to solve it, or how long time it will take.

A lot of stuff is unintuitive and poorly documented

Create project

```
project = angr.Project("./maybe_checker", main_opts={'base_addr': 0x400000})
```

- Select which binary to use
- Select loading options both for binary and libraries
- Select if external libraries should be loaded
  - Libraries (e.g Libc) can have functions that can be simplified

Create entry state

```
state = project.factory.entry_state()
```

- `.blank_state()` constructs a “blank slate” blank state, with most of its data left uninitialized. When accessing uninitialized data, an unconstrained symbolic value will be returned.
- `.entry_state()` constructs a state ready to execute at the main binary’s entry point.
- `.full_init_state()` constructs a state that is ready to execute through any initializers that need to be run before the main binary’s entry point, for example, shared library constructors or preinitializers. When it is finished with these it will jump to the entry point.
- `.call_state()` constructs a state ready to execute a given function.

- Can set things like address, stdin and argv as keyword arguments

SimulationManager

```
simgr = project.factory.simgr(state)
```

- Controls the state search
- Can ask different tings:
  - Step in the program
  - Find a certain state

Explore

```
simgr.explore(find=0x101229, avoid=0x10123a)
```

```
def is_success(state):
    return b"Access granted." in state.posix.dumps(1)

def is_failure(state):
    return b"Access denied." in state.posix.dumps(1)

simgr.explore(find=is_success, avoid=is_failure)
```

- Can be address(es) or functions

Path explosion

- When the number of different paths grows too fast
  - Too many states/branches to handle
- Angr does not handle well when the for loop uses a symbolic variable e.g.

Found states

- If simgr managed to get to the wanted state, it is in the found-array.
- From there you can get values for symbolic variables

```
if simgr.found:
    found_state = simgr.found[0]
    solution = found_state.posix.dumps(0)
    print("Recovered password:", solution.decode('utf-8', errors='ignore'))
else:
    print("Failed to find a valid password.")
```

Hooks

- Add or replace a part of the program
- Length = skip this number of bytes in the actual program
- You can read or change the current state of the program
- project.hook_symbol if you want to hook a symbol

```
@project.hook(0x101156, length=0)
def change_string(state):
    global_s_ptr = project.loader.main_object.get_symbol("global_s").rebased_addr
    global_s_addr = state.mem[global_s_ptr].uint64_t.resolved
    state.memory.store(global_s_addr, b"newstring\0")
```

On states: Posix

- You can get stdin, stdout (and more) by looking at the posix

```
def is_success(state):
    return b"Access granted." in state.posix.dumps(1)

def is_failure(state):
    return b"Access denied." in state.posix.dumps(1)

simgr.explore(find=is_success, avoid=is_failure)

if simgr.found:
    found_state = simgr.found[0]
    solution = found_state.posix.dumps(0)
    print("Recovered password:", solution.decode('utf-8', errors='ignore'))
```

On states: Memory

```
global_s_addr = s.mem[global_s_ptr].uint64_t.resolved
s.memory.store(global_s_addr, b"newstring\0")
```

- You can read and write to memory
- s.memory.store(addr, val) or s.memory.load(addr, size)
- s.mem[addr] for quick access
- Can load and store symbolic or actual values

On states: Registers

```
@project.hook(0x10116a, length=0)
def change_register(state):
    print("RSI was", state.regs.rsi, "changing to", hex(0x1337))
    state.regs.rsi = 0x1337
```

- Can read and write registers
- Can use both symbolic and actual values

On states: Solvers

- Can add constraints to variables.
- Can evaluate variables, i.e. find values for variables that satisfy constraints.

```
@project.hook(0x10116a, length=0)
def change_register(state):
    print("RSI was", state.regs.rsi, "changing to", rsi)
    state.regs.rsi = rsi
    state.solver.add(rsi == 0x1337)
```

```
endstate = simgr.found[0]
print("global_s:", endstate.solver.eval(global_s, cast_to=bytes))
print("rsi:", hex(endstate.solver.eval(rsi, cast_to=int)))
```

Extras: SimFile

- Can create symbolic files for the program to use
- state.fs.insert('filename', symbolic_file), state.fs.get, state.fs.delete

```
flag_len = 20
flag = claripy.BVS('flag', flag_len * 8)
simfile = angr.SimFile('stdin', content=flag, size=flag_len)

# Create the initial program state, passing our symbolic stdin
state = project.factory.entry_state(
    stdin=simfile
)
```

Extras: SimProcedure

- More advanced type of hook
  - `ret(expr)`: Return from a function
  - `jump(addr)`: Jump to an address in the binary
  - `exit(code)`: Terminate the program
  - `call(addr, args, continue_at)`: Call a function in the binary
  - `inline_call(procedure, *args)`: Call another SimProcedure in-line and return the results

```
class NotVeryRand(SimProcedure):
    def run(self, return_values=None):
        rand_idx = self.state.globals.get('rand_idx', 0) % len(return_values)
        out = return_values[rand_idx]
        self.state.globals['rand_idx'] = rand_idx + 1
        return out

project.hook_symbol('rand', NotVeryRand(return_values=[413, 612, 1025, 1111]))
```

Extra: Java

- Angr can also symbolically execute Java
- Eg. Jar-files, apk-files
- Examples in their repository
- Seems a bit outdated? I had some trouble with setting it up?

Exercises:

- Do angr_ctf: https://github.com/jakespringer/angr_ctf
- Solve "example_task" using angr
- Solve "maybe_checker" using angr (challenge!)
- Try out some challenges at https://github.com/angr/angr-examples
- Write your own angr-challenges