

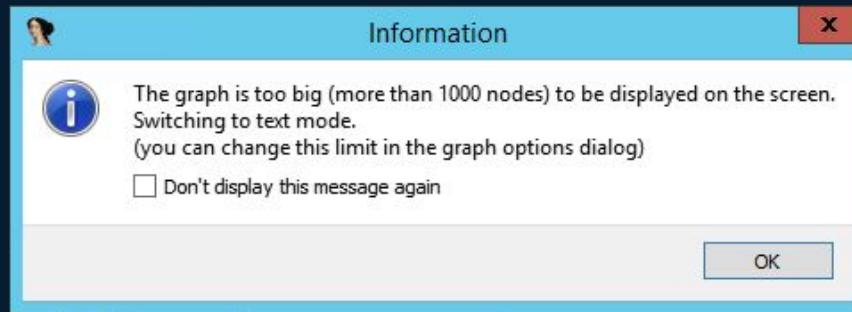
Let's solve a crackme!

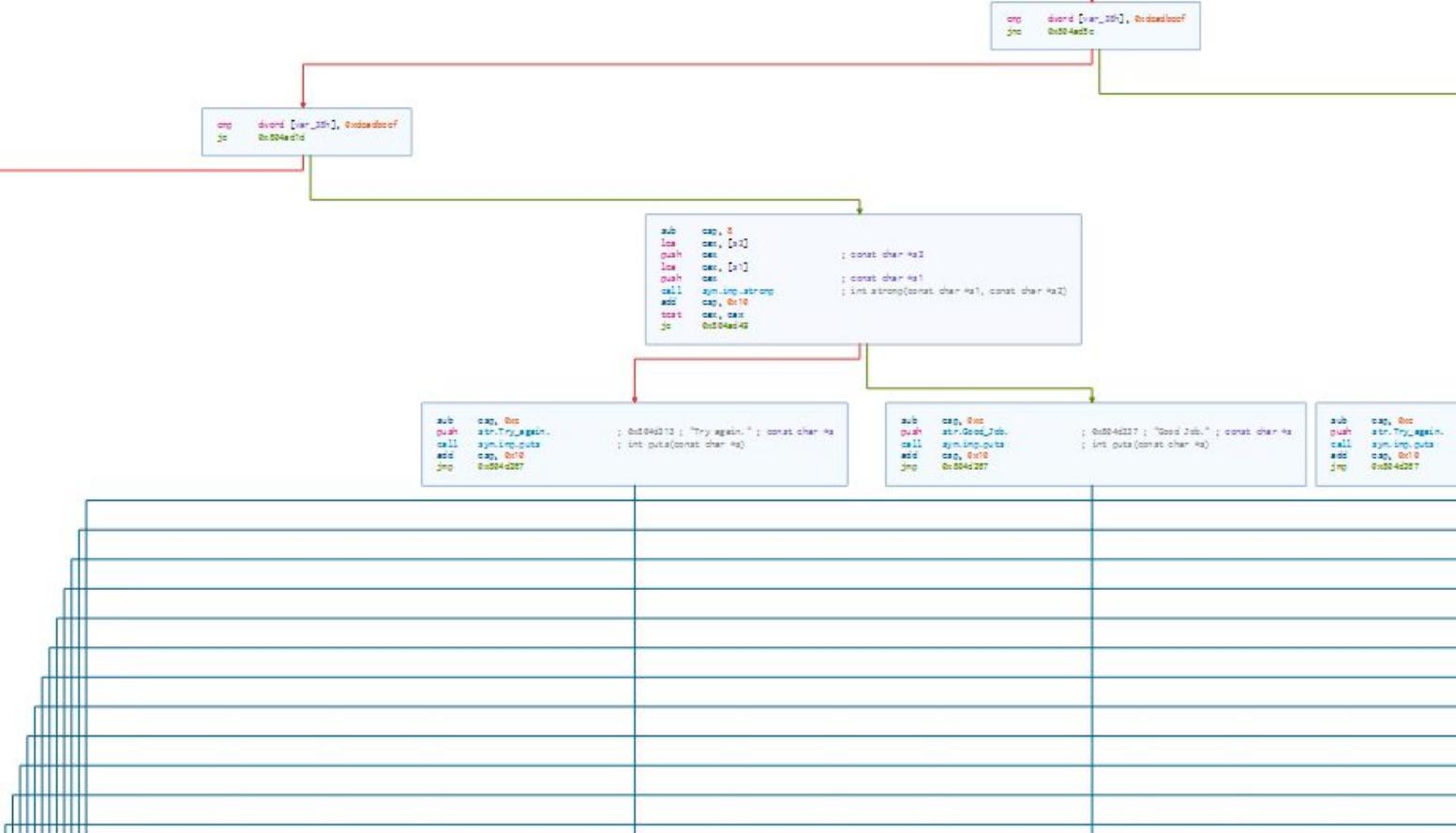


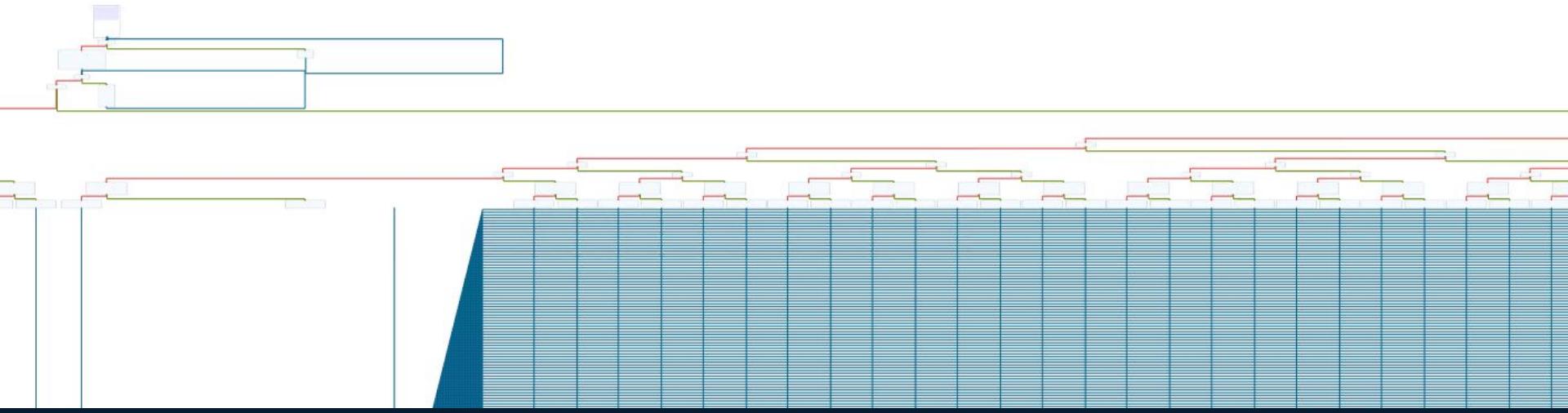
InsomniHack 2022
Jannis Kirschner

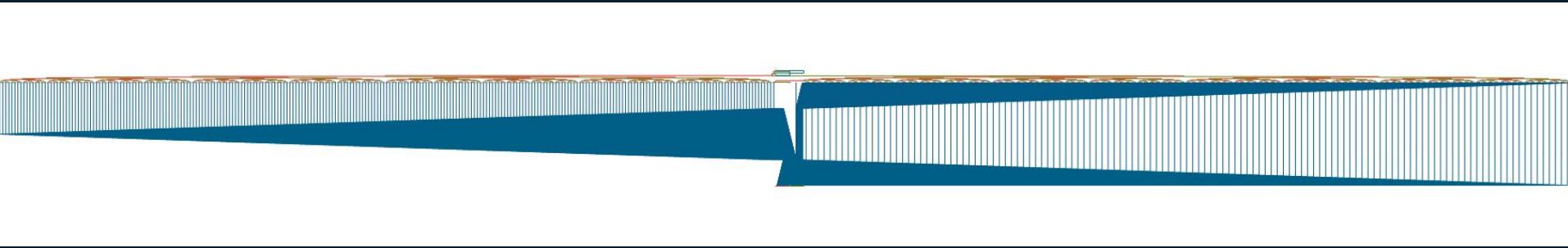


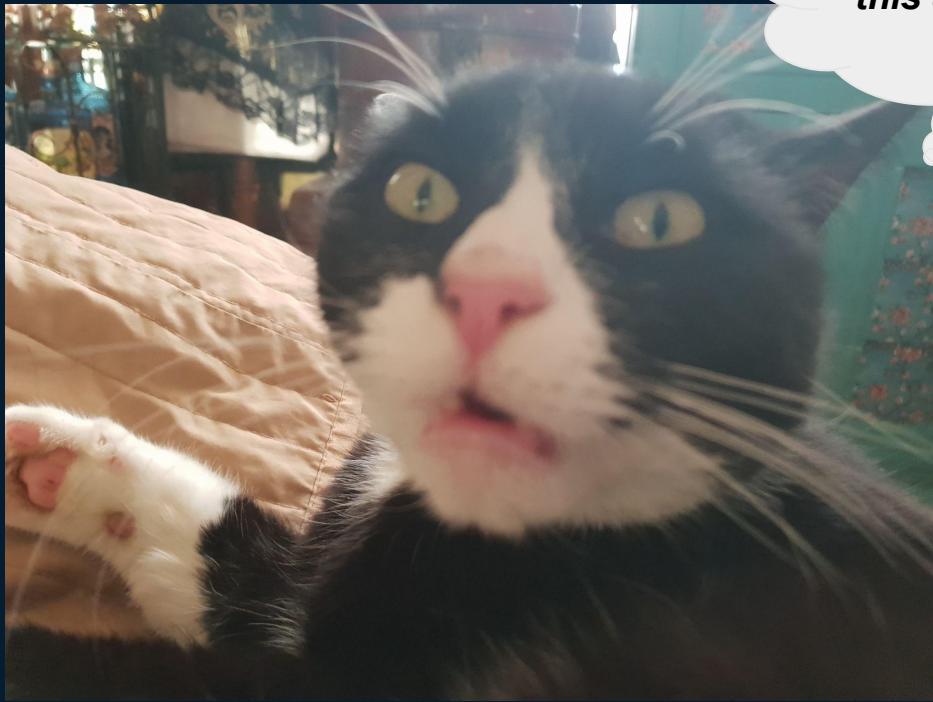
25.03.2022











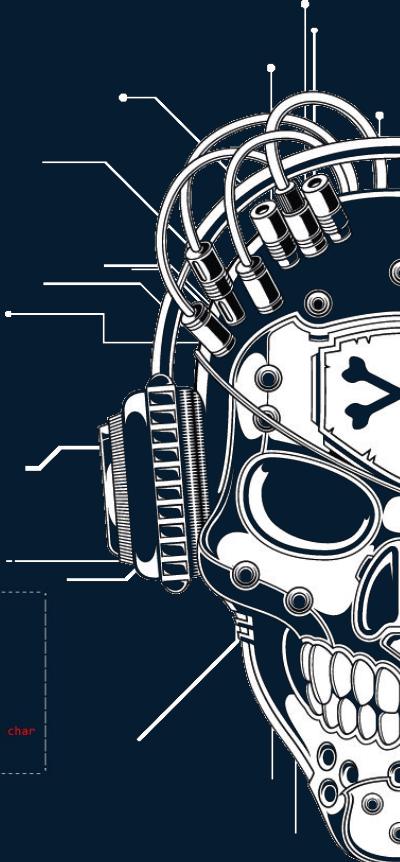
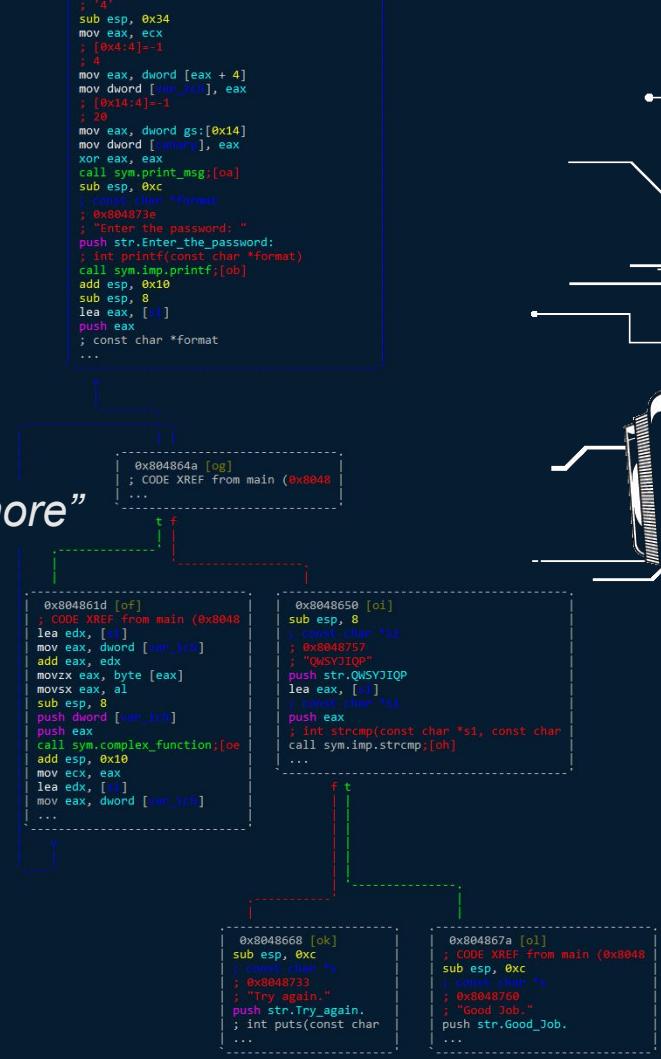
*What the (s)hell is
this abomination??!*

Symbolic Execution Demystified

or

“Why huge call-graphs don’t scare me anymore”

Insomni’Hack 2022
Jannis Kirschner



Jannis Kirschner

- Independent **Vulnerability Researcher**
- **Reverse Engineer & Exploit Developer**
- Passionate **CTF Player**

- Found major vulns in **e-voting systems**,
wifi routers and **embedded systems** with
my research team **suid.ch**



Views are my own and not related to my employer

 @xorkiwi

 /in/janniskirschner



What you'll learn today

ANGR

SYMBOLIC EXECUTION

PROBLEM STATE

Section 1: Problem State

Investigating the **Why?**

Section 2: Symbolic Execution



Section 3: Angr



Example: z3_robot (SharkyCTF2020)



I made a robot that can only communicate with "z3". He locked himself and now he is asking me for a password !

<https://ctftime.org/event/1034>

Creator : Nofix

Pts: 189

Static Analysis

```
_ | //\\n>==() || ()\\n  (* *)\\n  [-] )#(\\n  ( )...(\_)\\n  || |
_| ||//\\n>==() || ()\\n  (* *)\\n  [-] )#(\\n  ( )...(\_)\\n  || |
|_) ;  
sym.imp.puts(_obj.pass);  
sym.imp.printf(0x1589);  
sym.imp.fflush(_reloc.stdout);  
sym.imp.fgets((int64_t)&var_34h + 4, 0x19, _reloc.stdin);  
iVar2 = sym.imp.strcspn((int64_t)&var_34h + 4, 0x158d);  
*(undefined *)((int64_t)&var_34h + iVar2 + 4) = 0;  
cVar1 = sym.check_flag((char *)((int64_t)&var_34h + 4));  
if (cVar1 == '\\x01') {  
    sym.imp.puts(  
        "  \\_/_\\n  (* *)\\n  [-] )#(\\n  ( )...(\_)\\n  || |  
|_) ||//\\n>==() || ()\\n  (* *)\\n  [-] )#(\\n  ( )...(\_)\\n  || |  
|_) ;  
    sym.imp.printf("Well done, validate with shkCTF{%s}\\n", (int64_t)&var_34  
h + 4);  
} else {  
    sym.imp.puts(  
        "  \\_/_\\n  (* *)\\n  [-] )#(\\n  ( )...(\_)\\n  || |  
|_) ||//\\n>==() || ()\\n  (* *)\\n  [-] )#(\\n  ( )...(\_)\\n  || |  
|_) ;  
    sym.imp.puts("3Z Z3 z3 zz3 zz33");  
}
```

x86_64 ELF Binary

Not Stripped

Main function reads 24
chars via stdin and
passes to “check_flag”
function for validation

Trying to bruteforce

```
sym.imp.fgets((int64_t)&var_34h + 4, 0x19, _reloc.stdin);
```

Binary asks for a 24
characters long passphrase

Brute-forcing it would be
infeasible!

Password Length	Numerical 0-9	Upper & Lower case a-Z	Numerical Upper & Lower case 0-9 a-Z	Numerical Upper & Lower case Special characters 0-9 a-Z %\$
1	instantly	instantly	instantly	instantly
2	instantly	instantly	instantly	instantly
3	instantly	instantly	instantly	instantly
4	instantly	instantly	instantly	instantly
5	instantly	instantly	instantly	instantly
6	instantly	instantly	instantly	20 sec
7	instantly	2 sec	6 sec	49 min
8	instantly	1 min	6 min	5 days
9	instantly	1 hr	6 hr	2 years
10	instantly	3 days	15 days	330 years
11	instantly	138 days	3 years	50k years
12	2 sec	20 years	162 years	8m years
13	16 sec	1k years	10k years	1bn years
14	3 min	53k years	622k years	176bn years
15	26 min	3m years	39m years	27tn years
16	4 hr	143m years	2bn years	4qdn years
17	2 days	7bn years	148bn years	619qdn years
18	18 days	388bn years	9tn years	94qtn years
19	183 days	20tn years	570tn years	14sxn years
20	5 years	1qdn years	35qdn years	2sptn years

Sooooo...how can we solve such challenge?

$$2l = (A + \eta)^2 + \kappa^2 \text{ and}$$

Solving it manually

```
[0x00000760]> pdg @ sym.check_flag

undefined8 sym.check_flag(char *arg1)
{
    undefined8 uVar1;
    uint8_t uVar2;
    char *var_8h;

    if (((((((((uint8_t)(arg1[0x14] ^ 0x2bU) == arg1[7]) && ((int32_t)arg1[0x1
5] - (int32_t)arg1[3] == -0x14)) &&
           (arg1[2] >> 6 == '\0')) && ((arg1[0xd] == 't' && (((int32_t)arg1[0
xb] & 0xffffffffU) == 0x5f)))))) &&
        ((uVar2 = (uint8_t)(arg1[0x11] >> 7) >> 5,
         (int32_t)arg1[7] >> ((arg1[0x11] + uVar2 & 7) - uVar2 & 0x1f) == 5
&&
        (((uint8_t)(arg1[6] ^ 0x53U) == arg1[0xe] && (arg1[8] == 'z')))))
&&
        ((uVar2 = (uint8_t)(arg1[9] >> 7) >> 5, (int32_t)arg1[5] << ((arg1[9]
+ uVar2 & 7) - uVar2 & 0x1f) == 0x188
&& (((((int32_t)arg1[0x10] - (int32_t)arg1[7] == 0x14 &&
           (uVar2 = (uint8_t)(arg1[0x17] >> 7) >> 5,
            (int32_t)arg1[7] << ((arg1[0x17] + uVar2 & 7) - uVar2 & 0x1f)
== 0xbe)) &&
           ((int32_t)arg1[2] - (int32_t)arg1[7] == -0x2b)) &&
```

“check_flag” routine contains a lot of constraints to check for flag validity

We can extract them by hand

Solving it manually

Now we got a nice list with all the constraints that we can work with

```
1 int check_flag(byte *param_1)
2
3 {
4     return
5     (param_1[0x14] ^ 0x2b) == param_1[7] &&
6     param_1[0x15] - param_1[3] == -0x14 &&
7     param_1[2] >> 6 == '\0' &&
8     param_1[0xd] == 0x74 &&
9     (param_1[0xb] & 0xffffffffU) == 0x5f &&
10    bVar2 = (param_1[0x11] >> 7) >> 5,
11    param_1[7] >> ((param_1[0x11] + bVar2 & 7) - bVar2 & 0x1f) == 5 &&
12    (param_1[6] ^ 0x53) == param_1[0xe] &&
13    param_1[8] == 0x7a &&
14    bVar2 = (param_1[9] >> 7) >> 5,
15    param_1[5] << ((param_1[9] + bVar2 & 7) - bVar2 & 0x1f) == 0x188 &&
16    param_1[0x10] - param_1[7] == 0x14 &&
17    bVar2 = (param_1[0x17] >> 7) >> 5,
18    param_1[7] << ((param_1[0x17] + bVar2 & 7) - bVar2 & 0x1f) == 0xbe &&
19    param_1[2] - param_1[7] == -0x2b &&
20    param_1[0x15] == 0x5f &&
21    (param_1[2] ^ 0x47) == param_1[3] &&
22    *param_1 == 99 &&
23    param_1[0xd] == 0x74 &&
24    (param_1[0x14] & 0x45) == 0x44 &&
25    (param_1[8] & 0x15) == 0x10 &&
26    param_1[0xc] == 0x5f &&
27    param_1[4] >> 4 == '\a' &&
28    param_1[0xd] == 0x74 &&
29    bVar2 = (*param_1 >> 7) >> 5, *param_1 >> ((*param_1 + bVar2 & 7) -
   bVar2 & 0x1f) == 0xc &&
30    param_1[10] == 0x5f &&
31    (param_1[8] & 0xacU) == 0x28 &&
32    param_1[0x10] == 0x73 &&
33    (param_1[0x16] & 0x1d) == 0x18 &&
34    param_1[9] == 0x33 &&
35    param_1[5] == 0x31 &&
36    (param_1[0x13] & 0xffffffffU) == 0x72 &&
37    param_1[0x14] >> 6 == '\x01' &&
38    param_1[7] >> 1 == '/' &&
```

Plain Text ▾ Tab Width: 8 ▾

Ln 1, Col 1 ▾

INS

Solving it manually

```
(param_1[0x14] ^ 0x2b) == param_1[7]
param_1[0x15] - param_1[3] == -0x14
param_1[2] >> 6 == '\0'
param_1[0xd] == 0x74
(param_1[0xb] & 0xffffffffU) == 0x5f
(param_1[6] ^ 0x53) == param_1[0xe]
param_1[8] == 0x7a
param_1[0x10] - param_1[7] == 0x14
param_1[0x13] - param_1[0x15] == 0x13
param_1[0xc] == 0x5f
param_1[0xf] >> 1 == '/'
param_1[0x14] == 0x74
param_1[4] == 0x73
(param_1[0x17] ^ 0x4a) == *param_1
(param_1[6] ^ 0x3c) == param_1[0xb]
param_1[0x15] == 0x5f
```

----- s ----- z ----- t ----- r t -----

<- lower case t

<- lower case z

<- 0x13 + 0x5f = 0x72 (lower case r)

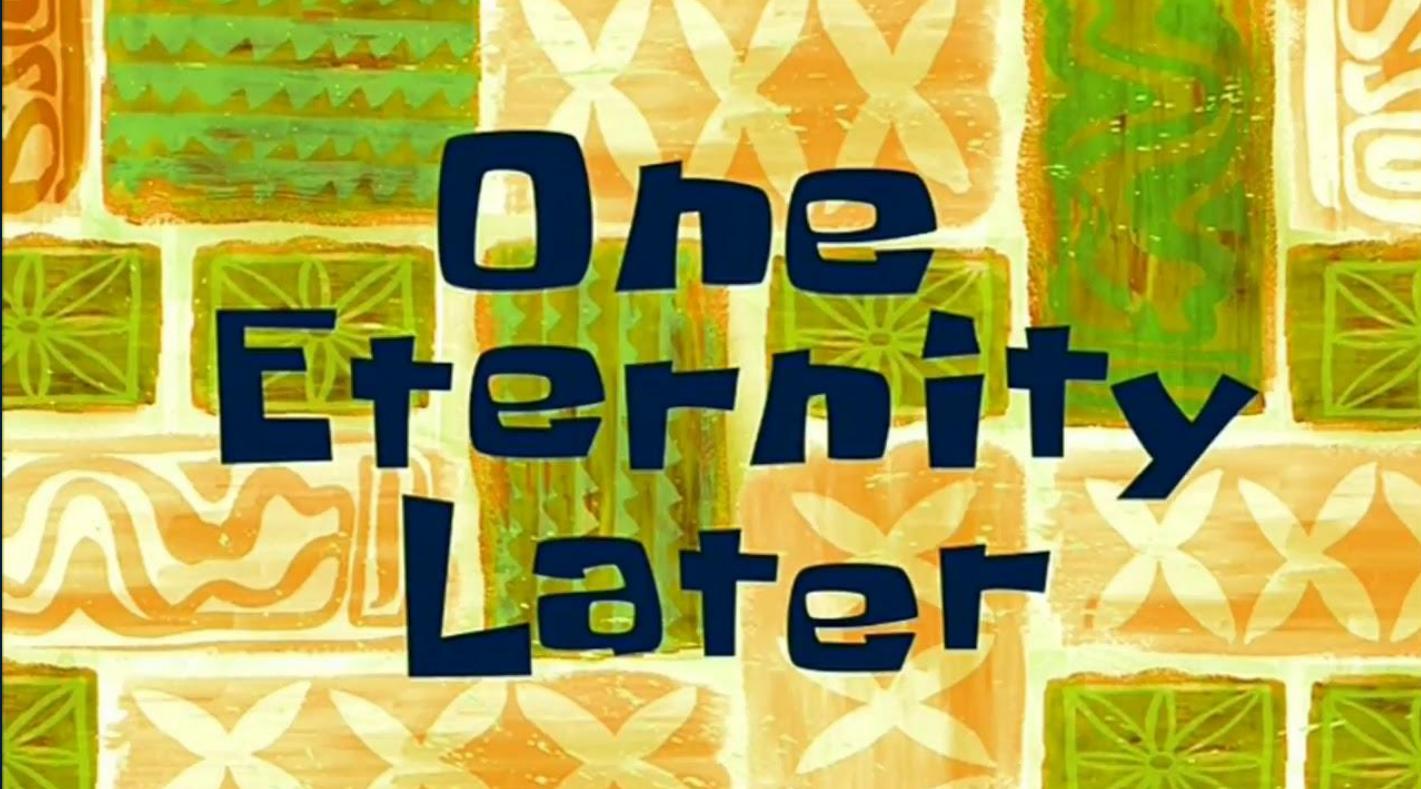
<- underscore

<- lower case t

<- lower case s

<- underscore





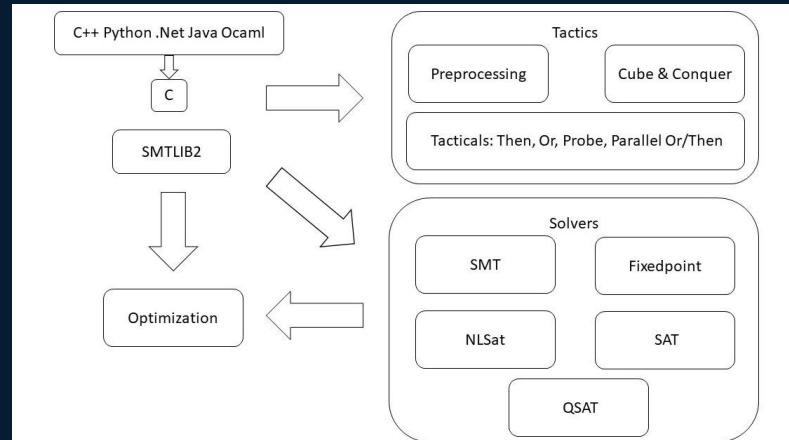
**One
Eternity
Later**

Overview over z3

The z3 theorem prover is an open source SMT solver developed by Microsoft Research

It's used to try and determine whether a mathematical formula is satisfiable using the boolean satisfiability (SAT) problem

SMT solving builds the bases for most modern symbolic execution frameworks

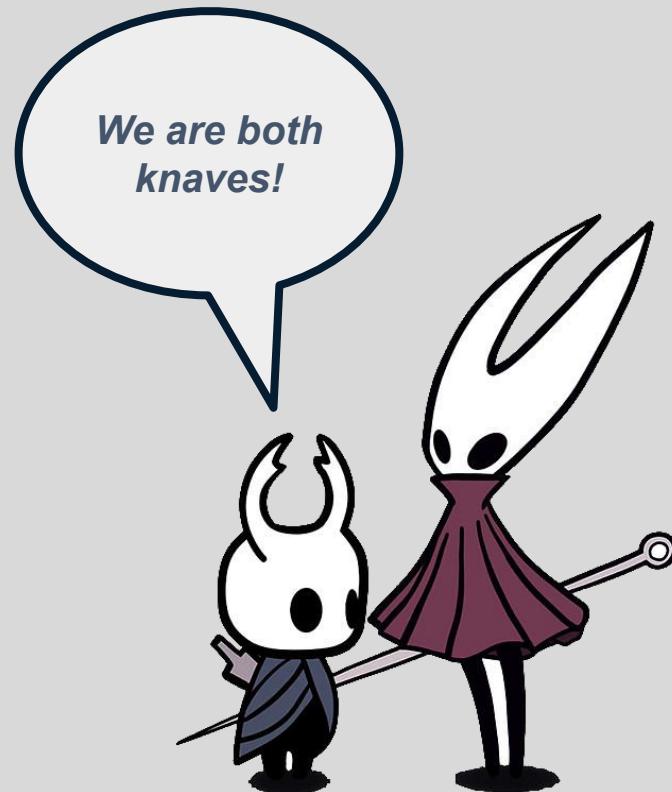


Architecture diagram of z3

A logic puzzle

There is an island inhabited by knights and knaves. Knights always tell the truth while knaves always lie.

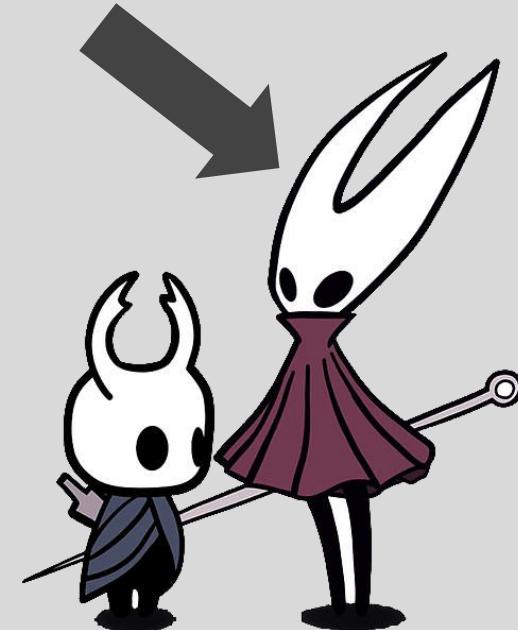
*Two people stand in front of you, Red and Blue. Blue tells you “**we are both knaves**”...who is the knight?*



A logic puzzle

Blue cannot be the knight. If blue was a knight he would've told a **lie** which is **infeasible** since knights cannot lie.

Our Knight



SAT/SMT solving

We can ask them questions like:

“Given three booleans a,b,c - can the following formula return true: ”

(a and not b) or (not a and c)

SAT/SMT solving

We can ask them questions like:

“Given three booleans a,b,c - can the following formula return true: ”
(a and not b) or (not a and c)

SAT: Fills a,b,c with **ones and zeroes** to prove SAT

SAT/SMT solving

We can ask them questions like:

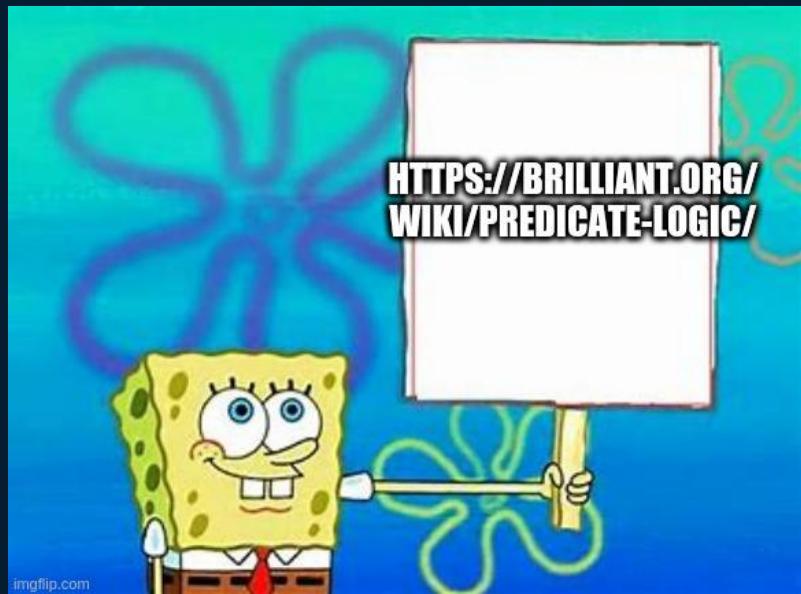
“Given three booleans a,b,c - can the following formula return true: ”
(a and not b) or (not a and c)

SAT: Fills a,b,c with **ones and zeroes** to prove SAT

SMT: Fills a,b,c with **new formulas** using integers, strings & new functions

SAT Solving	SMT Solving
<p>SAT solvers solve constraints written in propositional logic.</p>	<p>SMT solvers are more powerful and extend them by solving constraints written in predicate (first-order) logic with quantifiers.</p>
<p>Sentences/Statements are propositions (think knights and knaves). Propositional logic studies how they interact irregardless of the contents of the statement -> only logical connections.</p>	<p>Predicate logic extends propositional logic but replaces atomical elements (propositional letters) by properties to better describe the subject of a sentence. A quantified predicate is a proposition (assigned values to variables)</p>

If you wanna deep-dive into the maths:



pip install z3-solver

Automating with SMT Solvers

```
from z3 import *

a1 = [BitVec(f'{i}', 8) for i in range(0x19)]
s = Solver()

s.add((a1[20] ^ 0x2B) == a1[7])
s.add(a1[21] - a1[3] == -20)
s.add((a1[2] >> 6) == 0)
s.add(a1[13] == 116)
s.add(4 * a1[11] == 380)
s.add(a1[7] >> (a1[17] % 8) == 5)

.
.
.

-- INSERT --
```

Creating bitvectors
for keyspace

Placing all the
extracted constraints
by hand

Automating with SMT Solvers

```
s.add(a1[14] >> 4 == 3)
s.add((a1[12] & 0x38) == 24)
s.add(a1[8] << (a1[10] % 8) == 15616)
s.add(a1[20] == 116)
s.add(a1[6] >> (a1[22] % 8) == 24)
s.add(a1[22] - a1[5] == 9)
s.add(a1[7] << (a1[22] % 8) == 380)
s.add(a1[22] == 58)
s.add(a1[16] == 115)
s.add((a1[23] ^ 0x1D) == a1[18])
s.add(a1[23] + a1[14] == 89)
s.add((a1[5] & a1[2]) == 48)
s.add((a1[15] & 0x9F) == 31)
s.add(a1[4] == 115)
s.add((a1[23] ^ 0x4A) == a1[0])
s.add((a1[6] ^ 0x3C) == a1[11])

is_satisfiable = s.check()
model         = s.model()
solution_array = [chr(int(str(model[a1[i]]))) for i in range(len(model))]
flag          = ''.join(solution_array)
```



-- INSERT --

105,1

Bot

Check if constraints
are satisfiable

Compute model and
convert solved
bitvector integers to
characters

Display flag

Solution script

~100 Lines of Code

91 Constraints

```
1 from z3 import *
2
3 # List of Strings
4 al1 = [String('a1[%d]' % i) for i in range(0,11)]
5 v = Solver()
6
7 # Constraints
8
9 a.add(al1[0] <= al1[1])
10 a.add(al1[1] >= al1[0])
11 a.add(al1[2] >= al1[1])
12 a.add(al1[1] >= al1[2])
13 a.add(al1[2] >= al1[3])
14 a.add(al1[3] >= al1[2])
15 a.add(al1[4] >= al1[3])
16 a.add(al1[3] >= al1[4])
17 a.add(al1[5] >= al1[4])
18 a.add(al1[4] >= al1[5])
19 a.add(al1[6] >= al1[5])
20 a.add(al1[5] >= al1[6])
21 a.add(al1[7] >= al1[6])
22 a.add(al1[6] >= al1[7])
23 a.add(al1[8] >= al1[7])
24 a.add(al1[7] >= al1[8])
25 a.add(al1[9] >= al1[8])
26 a.add(al1[8] >= al1[9])
27 a.add(al1[10] >= al1[9])
28 a.add(al1[9] >= al1[10])
29 a.add(al1[0] >= v)
30 a.add(al1[1] >= v)
31 a.add(al1[2] & v == 1)
32 a.add(al1[3] & v == 1)
33 a.add(al1[4] & v == 1)
34 a.add(al1[5] & v == 1)
35 a.add(al1[6] & v == 1)
36 a.add(al1[7] & v == 1)
37 a.add(al1[8] & v == 1)
38 a.add(al1[9] & v == 1)
39 a.add(al1[10] & v == 1)
40 a.add(al1[0] >= al1[1])
41 a.add(al1[1] >= al1[0])
42 a.add(al1[1] >= al1[2])
43 a.add(al1[2] >= al1[1])
44 a.add(al1[2] >= al1[3])
45 a.add(al1[3] >= al1[2])
46 a.add(al1[4] >= al1[3])
47 a.add(al1[3] >= al1[4])
48 a.add(al1[5] >= al1[4])
49 a.add(al1[4] >= al1[5])
50 a.add(al1[6] >= al1[5])
51 a.add(al1[5] >= al1[6])
52 a.add(al1[7] >= al1[6])
53 a.add(al1[6] >= al1[7])
54 a.add(al1[8] >= al1[7])
55 a.add(al1[7] >= al1[8])
56 a.add(al1[9] >= al1[8])
57 a.add(al1[8] >= al1[9])
58 a.add(al1[10] >= al1[9])
59 a.add(al1[9] >= al1[10])
60 a.add(al1[0] >= al1[1])
61 a.add(al1[1] >= al1[0])
62 a.add(al1[1] >= al1[2])
63 a.add(al1[2] >= al1[1])
64 a.add(al1[2] >= al1[3])
65 a.add(al1[3] >= al1[2])
66 a.add(al1[4] >= al1[3])
67 a.add(al1[3] >= al1[4])
68 a.add(al1[5] >= al1[4])
69 a.add(al1[4] >= al1[5])
70 a.add(al1[6] >= al1[5])
71 a.add(al1[5] >= al1[6])
72 a.add(al1[7] >= al1[6])
73 a.add(al1[6] >= al1[7])
74 a.add(al1[8] >= al1[7])
75 a.add(al1[7] >= al1[8])
76 a.add(al1[9] >= al1[8])
77 a.add(al1[8] >= al1[9])
78 a.add(al1[10] >= al1[9])
79 a.add(al1[9] >= al1[10])
80 a.add(al1[0] >= al1[1])
81 a.add(al1[1] >= al1[0])
82 a.add(al1[1] >= al1[2])
83 a.add(al1[2] >= al1[1])
84 a.add(al1[2] >= al1[3])
85 a.add(al1[3] >= al1[2])
86 a.add(al1[4] >= al1[3])
87 a.add(al1[3] >= al1[4])
88 a.add(al1[5] >= al1[4])
89 a.add(al1[4] >= al1[5])
90 a.add(al1[6] >= al1[5])
91 a.add(al1[5] >= al1[6])
92 a.add(al1[7] >= al1[6])
93 a.add(al1[6] >= al1[7])
94 a.add(al1[8] >= al1[7])
95 a.add(al1[7] >= al1[8])
96 a.add(al1[9] >= al1[8])
97 a.add(al1[8] >= al1[9])
98 a.add(al1[10] >= al1[9])
99 a.add(al1[9] >= al1[10])
100
101 ix.satisfiable = s.check()
102 if ix.satisfiable == unsat:
103     solution_array = [int(i.to_int()>>model[al1[i]])) for i in range(len(model))]
104     flag = " ".join(solution_array)
105 else:
106     if ix.satisfiable:
107         print(flag)
108
109 if ix.satisfiable:
110     print("Satisfiable")
111 else:
112     print("Unsatisfiable")
```



Another Random Twitter User

@somedog

...

I saw a guy reversing a crackme today.
No symbolic execution.
No dynamic binary instrumentation.
No instruction counting.
He just sat there.
Extracting constraints by hand.
Like a Psychopath.



These materials may have been obtained through hacking

12:00 PM · Jun 10, 2021 · Twitter Web App

40.3K Retweets **11.3K** Quote Tweets **196.9K** Likes



Any guesses to how many lines of code we can reduce it?

We can do the same in about

4

lines of code

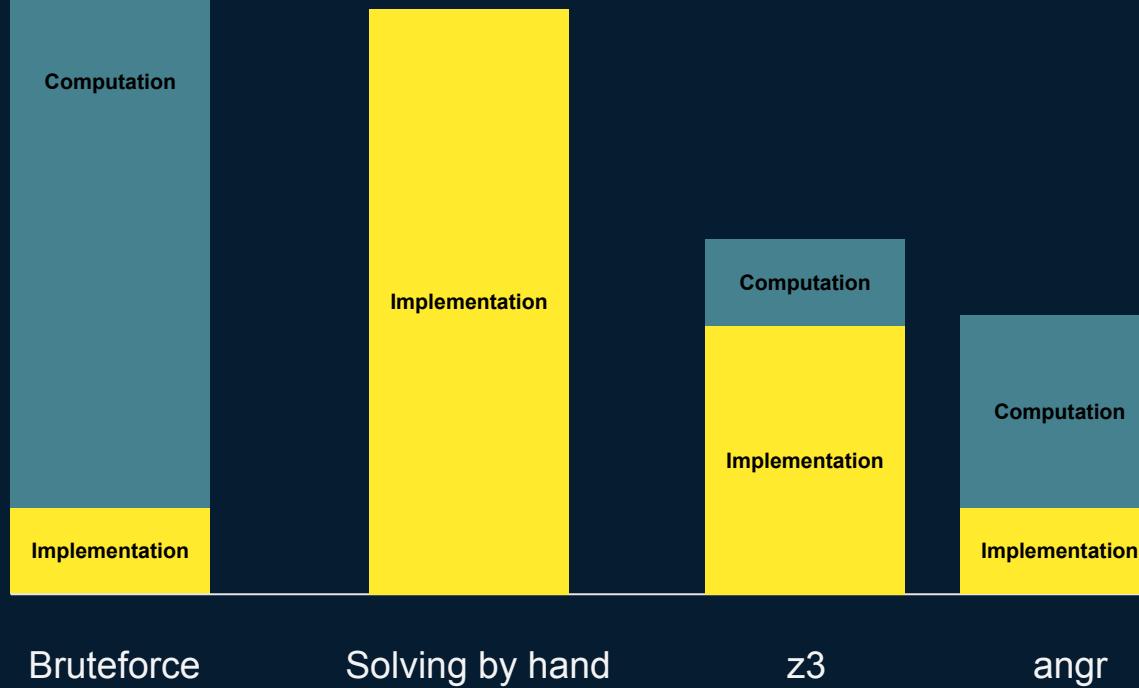
```
kiwi@doghouse: ~/Insomnihack/00_z3

import angr, claripy

proj = angr.Project('./z3_robot', load_options={"auto_load_libs": False}, main_opts={"base_addr": 0})
simgr = proj.factory.simgr()
simgr.explore(find=0x00001407, avoid=0x0000142d)
print(simgr.found[0].posix.dumps(0))

~
```

Efficiency Comparison



Problem State Recap

- Crackme input has to meet a lot of **constraints**
- Brute-force is infeasible
 - We extracted constraints and **manually searched** for matches
- This is slow and time consuming
 - We automated the constraint solving with **SMT solvers**
- Extracting constraints by hand takes a long time
 - We also automated this step with **symbolic execution**

Bruteforce



Solving by hand



SMT Solving



Symbolic Execution

Introducing

Symbolic Execution

Section 2: Symbolic Execution

Illuminating the **What?**

Section 1: Problem Space

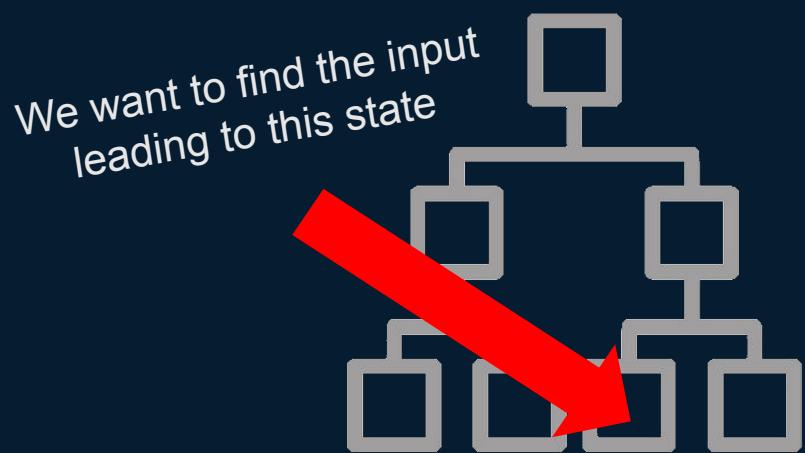


Section 3: Angr



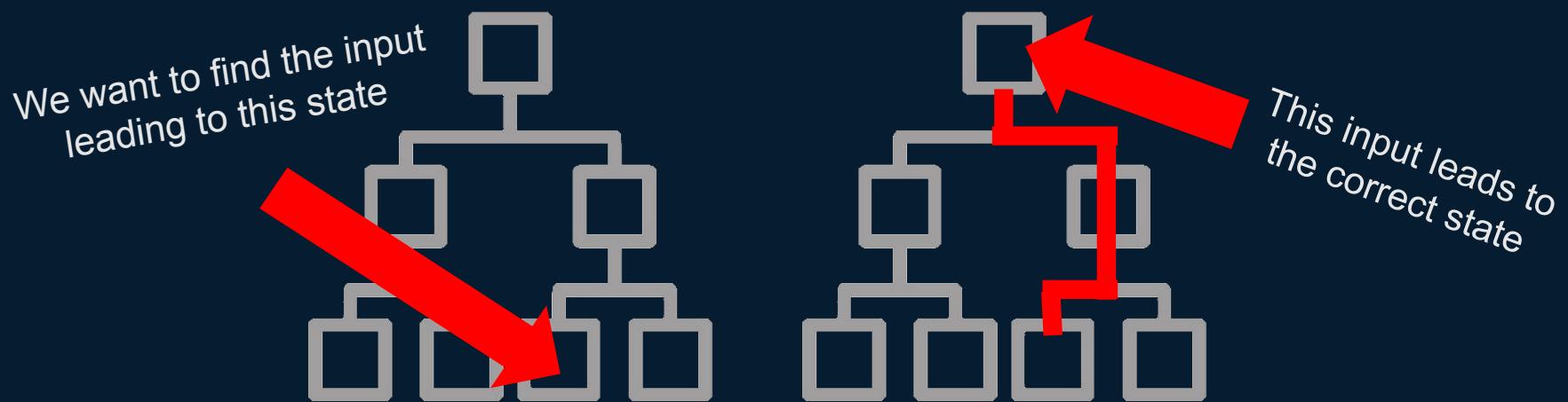
Symbolic Execution is a

“System that walks through all possible paths of a program to determine what inputs cause each of them to execute”



Symbolic Execution is a

“System that walks through all possible paths of a program to determine what inputs cause each of them to execute”



Concrete Execution

Program reads concrete input value to size

Input gets used for conditional branch and evaluated

Either a string is written to stdout or the crash function is called

```
void ValidSize()
{
    var size = read()
    if (size < 5):
        printf("Works")
    else:
        crash()
}
```

Concrete Execution

Program reads concrete input value to size

Input gets used for conditional branch and evaluated

Either a string is written to stdout or the crash function is called

```
void ValidSize()
{
    var size = read()      ← 4
    if (size < 5):
        printf("Works")
    else:
        crash()
}
```

Concrete Execution

Program reads concrete input value to size

Input gets used for conditional branch and evaluated

Either a string is written to stdout or the crash function is called

```
void ValidSize()
{
    var size = read()      ← 4
    if (size < 5):        ← True
        printf("Works")
    else:
        crash()
}
```

Concrete Execution

Program reads concrete input value to size

Input gets used for conditional branch and evaluated

Either a string is written to stdout or the crash function is called

```
void ValidSize()
{
    var size = read()      ← 4
    if (size < 5):        ← True
        printf("Works")   ← Executed
    else:
        crash()
}
```

“Static” Symbolic Execution

Instead of concrete input symbolic value is assigned to size

Symbolic value can take any value so proceeds with both paths by “forking”

After crash/normal termination computes concrete value by smt solving the accumulated path constraints

```
void ValidSize()
{
    var size = read()
    if (size < 5):
        printf("Works")
    else:
        crash()
}
```

“Static” Symbolic Execution

Instead of concrete input symbolic value is assigned to size

Symbolic value can take any value so proceeds with both paths by “forking”

After crash/normal termination computes concrete value by smt solving the accumulated path constraints

```
void ValidSize()
{
    var size = read()           ← λ
    if (size < 5):
        printf("Works")
    else:
        crash()
}
```

“Static” Symbolic Execution

Instead of concrete input
symbolic value is assigned to
size

Symbolic value can take any
value so proceeds with both
paths by “forking”

After crash/normal termination
computes concrete value by
smt solving the accumulated
path constraints

```
void ValidSize()
{
    var size = read()           ← λ
    if (size < 5):
        printf("Works")      ← λ < 5
    else:
        crash()              ← λ > 5
}
```

“Static” Symbolic Execution

Instead of concrete input
symbolic value is assigned to
size

Symbolic value can take any
value so proceeds with both
paths by “forking”

After crash/normal termination
computes concrete value by
smt solving the accumulated
path constraints

```
void ValidSize()
{
    var size = read()           ← λ
    if (size < 5):
        printf("Works")      ← λ < 5
    else:
        crash()              ← λ > 5
}
```

The problem with static symbolic execution...

It's difficult for static symbolic execution to reach deep into the execution tree

Path selection heuristics might choose paths that won't advance propagation

For example in a loop depending on a symbolic variable it might not find the exit



“Dynamic” Concolic Testing

Concrete Testing
+
Symbolic Execution

= Concolic Testing

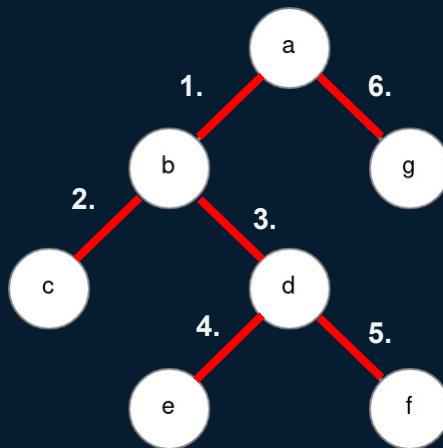
“Dynamic” Concolic Testing

Concrete Testing
+
Symbolic Execution

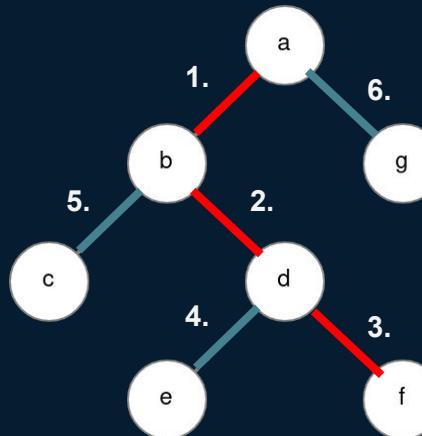
= Concolic Testing

Seed-driven concolic execution is able to favor paths and reach deep into the execution tree

Symbolic vs Concolic Execution



- Main Path
- Adjacent Paths



Explores **all** possible paths in a binary

Explores **adjacent** paths along a main branch based on seed input

“Dynamic” Concolic Testing

Run program with a concrete
(random) seed input

Collect the path constraint

Negate the last (not already
negated) constraint

SMT solve to inverse the latest
branch and discover a new path

Repeat until no new paths are
found

```
void ValidSize()
{
    var size = read()
    if (size < 5):
        printf("Works")
    else:
        crash()
}
```

“Dynamic” Concolic Testing

Run program with a concrete
(random) seed input

Collect the path constraint

Negate the last (not already
negated) constraint

SMT solve to inverse the latest
branch and discover a new path

Repeat until no new paths are
found

```
void ValidSize()
{
    var size = read()           ← 4
    if (size < 5):             ← True
        printf("Works")
    else:
        crash()
}
```

“Dynamic” Concolic Testing

Run program with a concrete
(random) seed input

Collect the path constraint

Negate the last (not already
negated) constraint

SMT solve to inverse the latest
branch and discover a new path

Repeat until no new paths are
found

```
void ValidSize()
{
    var size = read()           ← 4
    if (size < 5):             ← True
        printf("Works")       ← λ < 5
    else:
        crash()
}
```

“Dynamic” Concolic Testing

Run program with a concrete
(random) seed input

Collect the path constraint

Negate the last (not already
negated) constraint

SMT solve to inverse the latest
branch and discover a new path

Repeat until no new paths are
found

```
void ValidSize()
{
    var size = read()           ← 4
    if (size < 5):             ← True
        printf("Works")       ← λ < 5
    else:
        crash()               ← ¬(λ < 5)
}
```

“Dynamic” Concolic Testing

Run program with a concrete
(random) seed input

Collect the path constraint

Negate the last (not already
negated) constraint

SMT solve to inverse the latest
branch and discover a new path

Repeat until no new paths are
found

```
void ValidSize()
{
    var size = read()           ← 4
    if (size < 5):             ← True
        printf("Works")       ← λ < 5
    else:                      ← ¬(λ < 5)
        crash()                ← λ >= 5
}
```

“Dynamic” Concolic Testing

Run program with a concrete
(random) seed input

Collect the path constraint

Negate the last (not already
negated) constraint

SMT solve to inverse the latest
branch and discover a new path

Repeat until no new paths are
found

```
void ValidSize()
{
    var size = read()           ← 4
    if (size < 5):             ← True
        printf("Works")       ← λ < 5
    else:                      ← ¬(λ < 5)
        crash()                ← λ >= 5
}
```

Program Validation Tradeoffs



Cost (Computational Resources/Time/Manual Labor)

Different symbolic execution tools

Full System: s2e

User:
Angr
Triton
Manticore

Code: KLEE

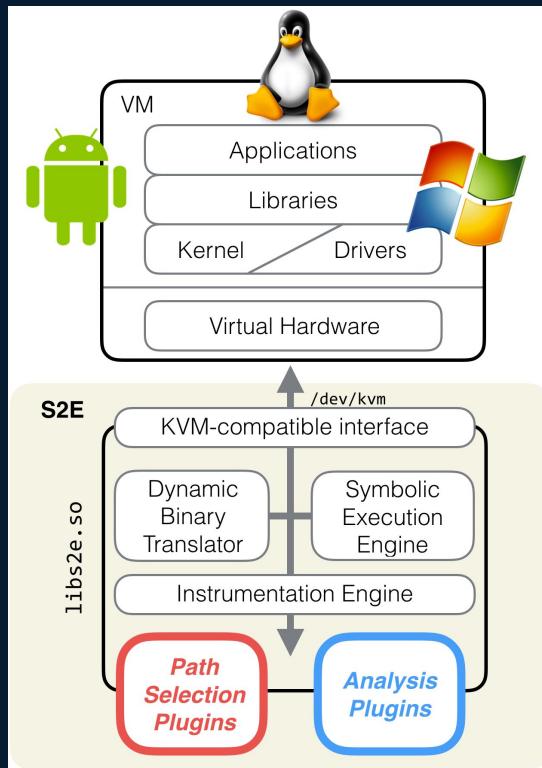
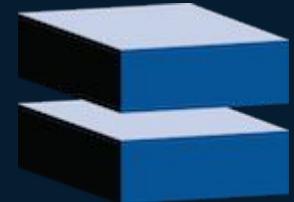
Different symbolic execution tools

Full System: s2e

User:
Angr
Triton
Manticore

Code: KLEE

S²E: The Selective Symbolic Execution Platform



S2E Architecture Diagram

Modular library that enriches virtual machines with symbolic execution & program analysis capabilities.

Runs entire software stack including applications, libraries, kernel, firmware and drivers (full system emulation).

Extensible and able to analyze large, complicated software like device drivers that have a lot of complex interactions.

S2E Walkthrough

Create new S2E Project	Implement custom plugin	Define events to react to	Kill after desired state is reached	Track/Filter desired process	Limit the constraint space of the symbolic variable	On success solve constraints and display	Enable plugin in config and launch s2e
<code>s2e new_project</code>	<code>libs2eplugins/src/Plugins/yourplugin.cpp</code>	<p>OSMonitor for example has events for process creation, module loading/unloading</p> <p>onTranslateInstructionStart and executionSignal can trigger on certain address (for PID load module and calculate relative)</p>	<code>s2e()>getExecutor()>terminateStateEarly()</code>	ProcessExecution-Detector plugin can track single process	<code>state->constraints.addConstraint()</code>	<code>!s2e()>getExecutor()>getSymbolicSolution()</code>	<ul style="list-style-type: none">• Build with s2e build• Edit <code>s2e-config.lua</code>• Run <code>launch-s2e.sh</code>

Why GitHub? Team Enterprise Explore Marketplace Pricing

Search Sign in Sign up

adrianherrera/unbreakable-ctf-s2e [Public]

Code Issues Pull requests Actions Projects Wiki Security Insights

master · unbreakable-ctf-s2e / libs2eplugins / src / s2e / Plugins / GoogleCTFUnbreakable.h

adrianherrera unbreakable: added missing header

Latest commit raaasea on Apr 25, 2019 History

1 contributor

44 lines (38 sloc) | 1.16 KB

Raw Blame

```
1 /**
2 * Copyright (C) 2017, Dependable Systems Laboratory, EPFL
3 * All rights reserved.
4 */
5 /// Copy this file to source/s2e/libs2eplugins/src/s2e/Plugins in your S2E
6 /// environment.
7 /**
8 *
9 #ifndef S2E_PLUGINS_GOOGLE_CTF_UNBREAKABLE_H
10 #define S2E_PLUGINS_GOOGLE_CTF_UNBREAKABLE_H
11
12 #include <s2e/CorePlugin.h>
13 #include <s2e/Plugin.h>
14 #include <s2e/S2E.h>
15
16 namespace s2e {
17     namespace plugins {
18
19         class ProcessExecutionDetector;
20
21     class GoogleCTFUnbreakable : public Plugin {
22         S2E_PLUGIN
23
24     public:
25         GoogleCTFUnbreakable(S2E *s2e) : Plugin(s2e) {
26             }
27
28         void initialize();
29
30     private:
31         ProcessExecutionDetector *_procDetector;
32
33         void onSymbolicVariableCreation(S2EEExecutionState *state, const std::string &name,
34                                         const std::vector<klee::Expr> &expr, const klee::MemoryObject *mo,
35                                         const klee::Array *array);
36
37         void onTranslateInstruction(ExecutionSignal *signal, S2EEExecutionState *state, TranslationBlock *tb, uint64_t pc);
38         void onSuccess(S2EEExecutionState *state, uint64_t pc);
39         void onFailure(S2EEExecutionState *state, uint64_t pc);
40     };
41
42 } // namespace plugins
43 } // namespace s2e
44 #endif
```

unbreakable-ctf-s2e/Goo

Code Issues Pull requests Actions Projects Wiki Security Insights

File unbreakable-ctf-s2e/Unbreakable.cpp

```
#!/usr/bin/python

# This file contains the code for the unbreakable encryption problem accusation challenge from the simple web user. It is a modified version of the original code provided by the challenge author, with some changes made to make it more challenging.

# The code is designed to be run on a Linux system with Python 2 installed. It uses the pwn library to interact with the program's memory space.

# The program takes a single argument, which is the path to the executable file. The executable file is located in the same directory as this script.

# The program reads the input from the standard input stream (stdin). It expects the user to enter a string of characters, which will be encrypted using the Caesar cipher algorithm.

# The program then prints the encrypted output to the standard output stream (stdout). The user can then copy and paste the output into a text editor or terminal window to view the encrypted message.

# The program also includes a function called "check_answer" which takes two arguments: the user's input and the expected output. It compares the two strings and prints a message indicating whether the answer is correct or incorrect.

# The program also includes a function called "generate_random_string" which generates a random string of characters using the ASCII character set. This function is used to generate a random password for the user to enter.

# The program also includes a function called "hash_password" which takes a password and returns a hashed version of it. This function is used to store the user's password in a secure manner.

# The program also includes a function called "check_password" which takes a password and returns a boolean value indicating whether the password is correct or not. This function is used to verify the user's password when they log in.
```

Different symbolic execution tools

Full System: s2e

User: Angr
Triton
Manticore

Code: KLEE

Angr/Triton/Manticore



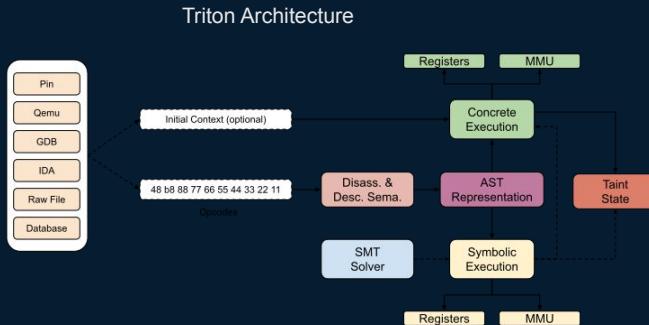
TRITON
Dynamic Binary Analysis



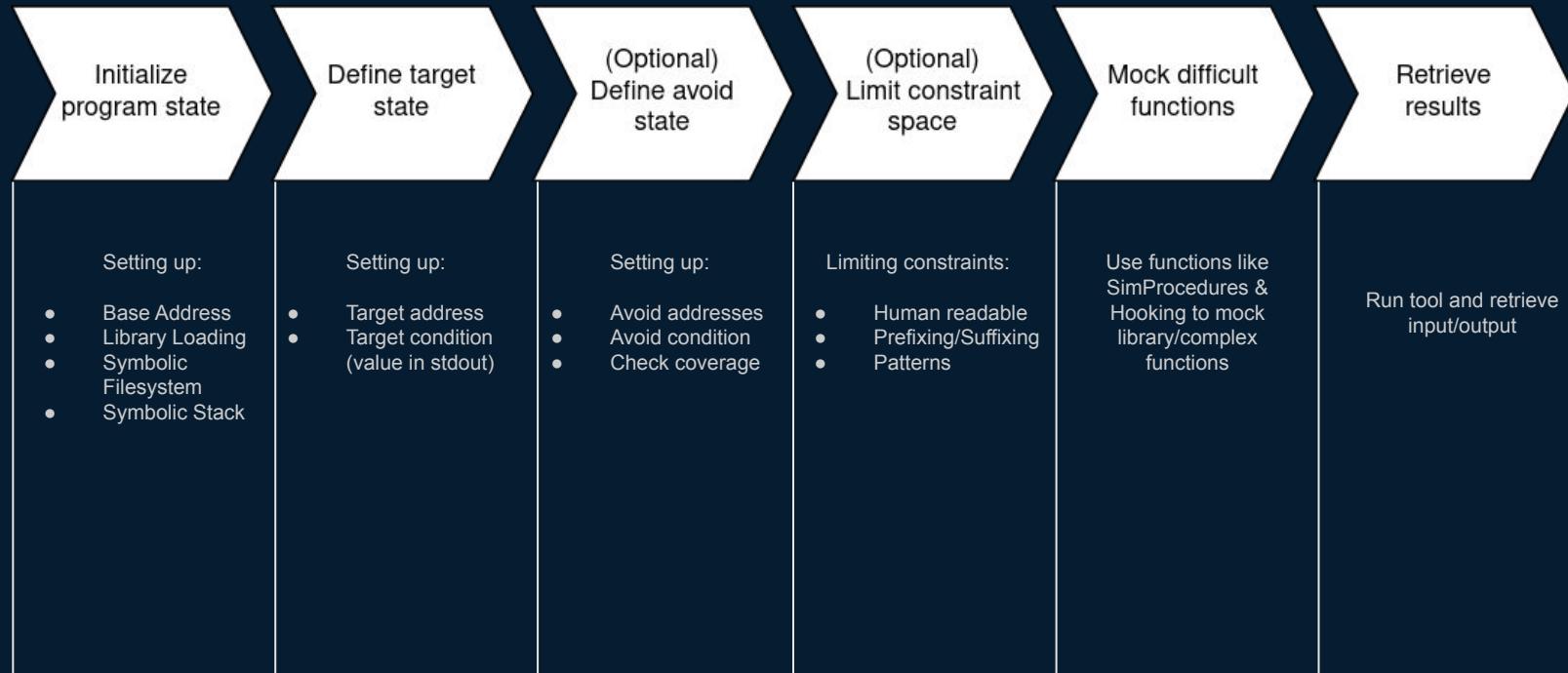
User-level dynamic binary analysis & symbolic execution frameworks (often based on z3).

Able to lift & instrument a number of binary architectures like x86, x86-64, AArch64, EVM Smart Contracts, ARM, MIPS, WASM, PowerPC (yes, even BrainFuck)

Great mix between convenience, speed and instrumentability - perfect for CTF



User-Level Workflow



angr-doc/solve.py at master · angr/angr-doc · GitHub

https://github.com/angr/angr-doc/blob/master/examples/google2016/unbreakable_0/solve.py

Why GitHub? Team Enterprise Explorers Marketplace Pricing

Search Sign in Sign up

angr / angr-doc Public

Code Issues Pull requests Actions Projects Wiki Security Insights

Jump to · Go to file · ...

Initial Migrate to Python 3 (x21) — x Latest commit 1e39000 on Oct 1, 2018 History

As 6 contributors

Executable File | 55 lines (39 slim) | 2.50 KB

```
1 #!/usr/bin/env python2
2
3 # Author: David Marnachchi <dmarnachchi@protonmail.com>
4 # Google 2016 CTF
5 # Challenge: Unbreakable Enterprise Product Activation
6 # Team: HackCaravan (http://hacckaravan.tweru.net)
7 # Runtime: ~4.5 seconds (single threaded 12-2000 v3 @ 2.50GHz on AMD K20)
8 #
9 import angr
10
11 import claripy
12
13 def main():
14     proj = angr.Project("./unbreakable-enterprise-product-activation", load_options={"auto_load_libs": False}) # Disabling the automatic library loading saves a few seconds
15
16     input_size = 0x100 # Max length from strcpy, see DoSEED.
17
18     argv1 = claripy.BVS("argv1", input_size * 8)
19
20     initial_state = proj.factory.entry_state(args=["./unbreakable-enterprise-product-activation", argv1], add_options={angr.options.LAZY_SOLVES})
21     initial_state.libc.buf_symbolic_bytes=input_size * 8 # Thanks to Christopher Talls (@salt11) for pointing this out. By default there's only 0x8 symbolic bytes
22
23     # For some reason if you constrain too few bytes, the solution isn't found. To be safe, I'm constraining them all.
24     for byte in range(0, input_size):
25         initial_state.add_constraints(argv1[byte] == '\x00') # null
26         initial_state.add_constraints(argv1[byte] == '\x41') # 'A'
27         initial_state.add_constraints(argv1[byte] == '\x7f') # '?'
28
29     # Source: https://www.juniper.net/documentation/JUNOS14.1/topics/reference/general/introspection-prevention-custom-attack-object-extended-ascii/
30     # Thanks to the Juniper Security Research team for showing me how to restrict to printable characters.
31
32     # We want to make sure that every flag is formatted as "CTF{...}", so we might as well use that information to save processing time.
33     initial_state.add_constraints(argv1[0] == 'C')
34     initial_state.add_constraints(argv1[1] == 'T')
35     initial_state.add_constraints(argv1[2] == 'F')
36     initial_state.add_constraints(argv1[3] == '{')
37     initial_state.add_constraints(argv1[4] == '}')
38
39     initial_state.add_constraints(argv1[5:input_size] == ' ')
40
41     initial_state.add_constraints(argv1[-1] == '\x00') # null bytes at the end of the flag (if any).
42
43     sm = proj.factory.simulation_manager(initial_state)
44
45     sm.explore(find=0x400030, avoid=0x400020)
46
47     sm.found[0].add_pc(0x400030)
48
49     print(sm.found[0].pc)
50
51     assert sm.found[0].pc == 0x400030
52
53     if __name__ == "__main__":
54         print(repr(sm.found[0]))
```

© 2022 GitHub, Inc. Terms Privacy Security Status Docs Contact GitHub Pricing API Training Blog About

Different symbolic execution tools

Full System: s2e

User:
Angr
Triton
Manticore

Code: KLEE

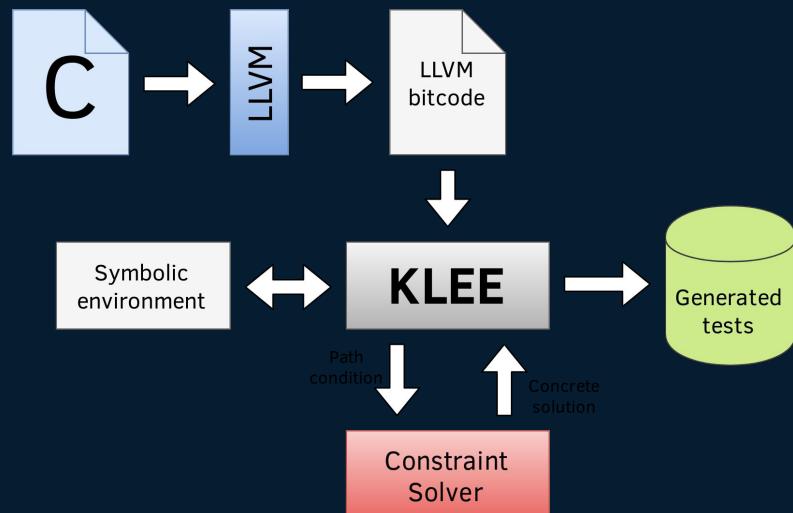
KLEE



LLVM-based symbolic execution engine
for code-level analysis

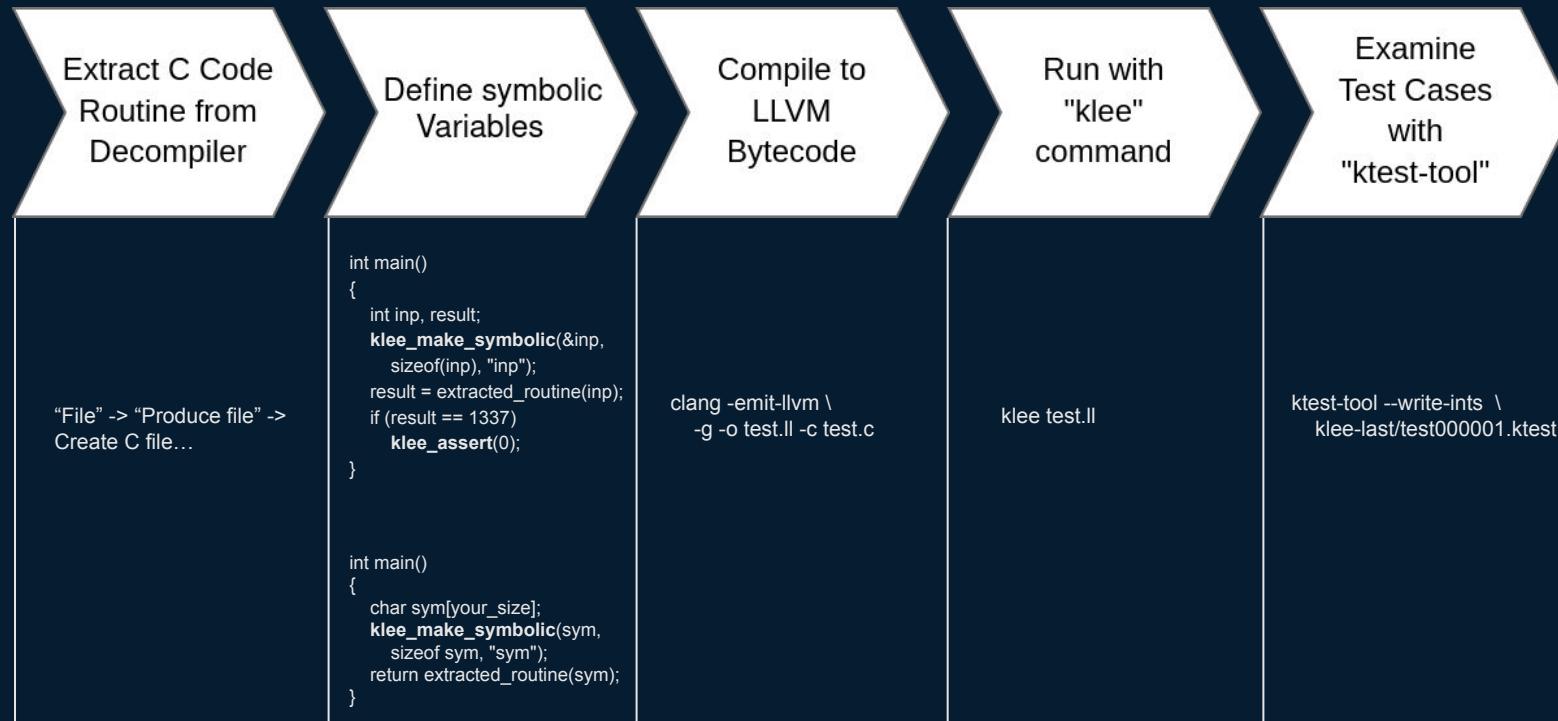
Requires target function to be re-coded
in C and instrumented

High performance due to smaller
overhead compared with other
frameworks, as well as nifty features
such as coverage, test case and path
exporting



KLEE Architecture Diagram

KLEE Walkthrough



main.c · master · David M · X

https://gitlab.com/Manouchehri/Matryoshka-Stage-2/-/blob/main.c 67% ⚡

GitLab Search GitLab Sign in / Register

Matryoshka-Stage-2 · Repository

master · Matryoshka-Stage-2 / main.c

Find file Blame History Permalink

Make Klee friendly.

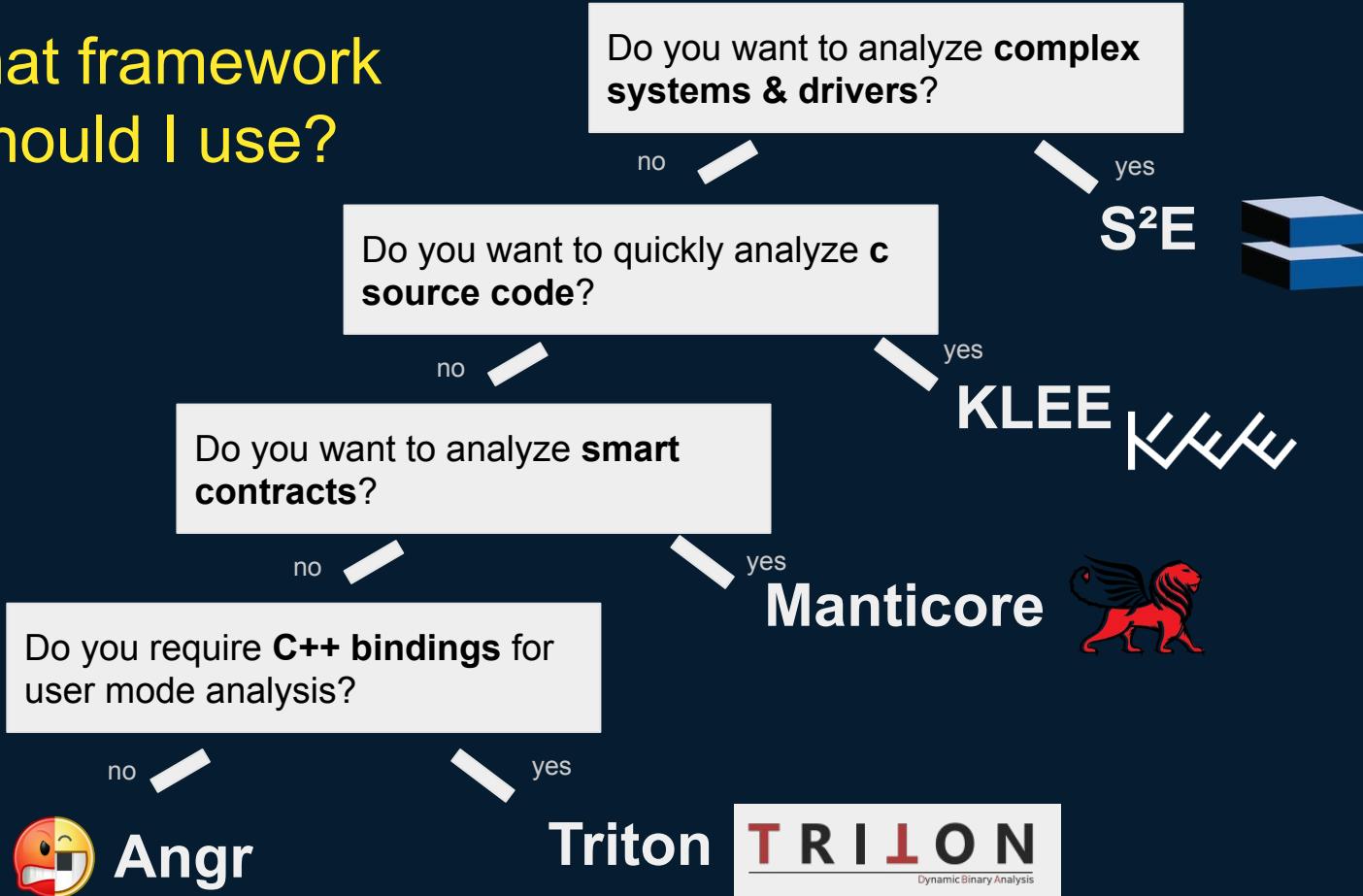
David Manouchehri authored 5 years ago

main.c 1.03 kB

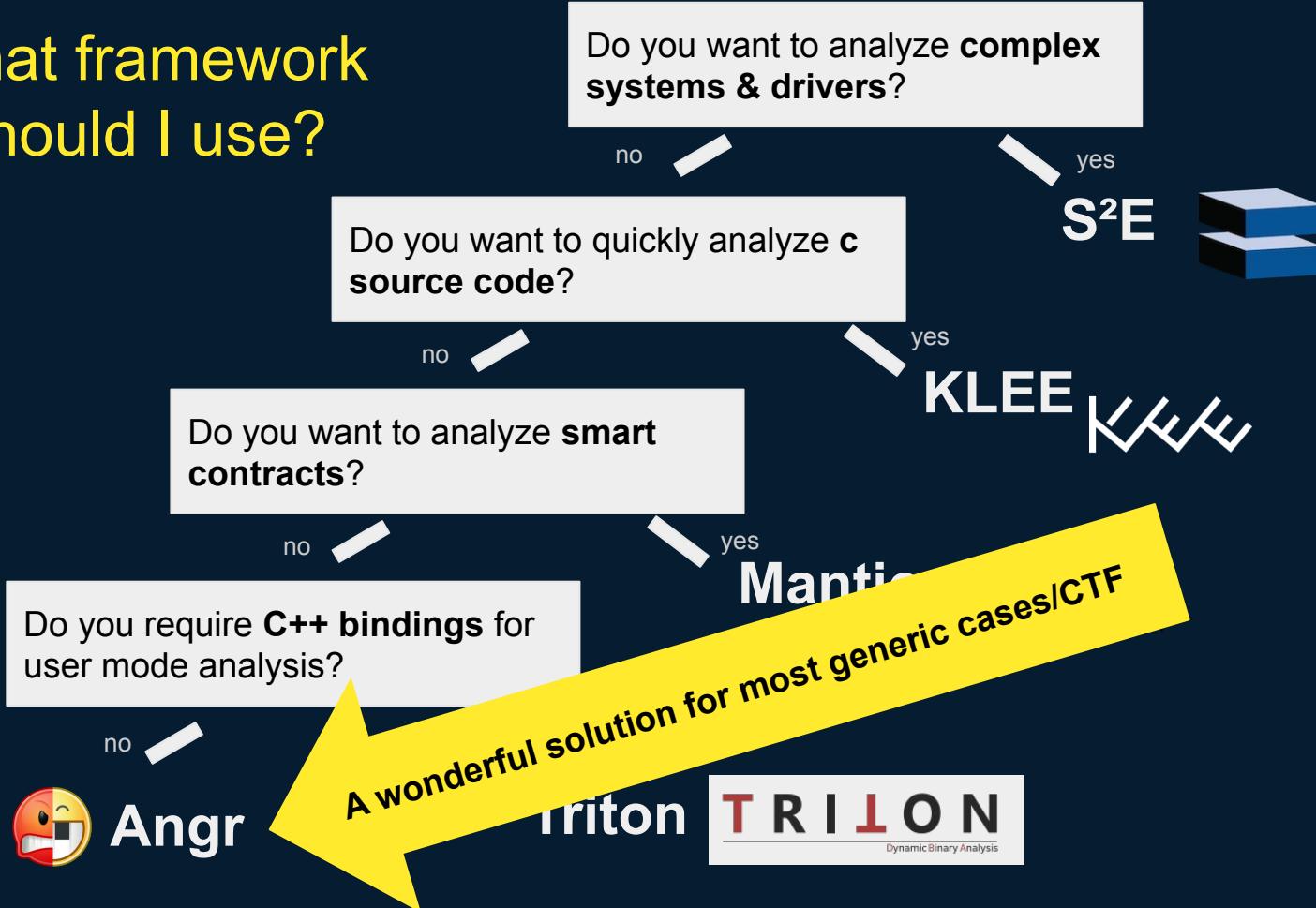
```
1 #include "defs.h" // Take from IDA's plugins/
2 #include <string.h>
3 #include <stdio.h>
4 #include <assert.h>
5 #include <klee/klee.h>
6
7 int main(int a1, char **a2, char **a3)
8 {
9     __int64 v4; // r8x10
10    signed int v5; // [sp+1Ch] /bp-14h/04
11
12    if ( a1 == 2 )
13    {
14        if ( 42 * (strlen(a2[1]) + 1) != 504 )
15            goto LABEL_31;
16        v5 = 1;
17        if (*a2[1] != 80 )
18            v5 = 0;
19        if ( 2 * a2[1][3] != 200 )
20            v5 = 0;
21        if ( *a2[1] + 16 != a2[1][6] - 16 )
22            v5 = 0;
23        v4 = a2[1][5];
24        if ( v4 != 9 * strlen(a2[1]) - 4 )
25            v5 = 0;
26        if ( a2[1][1] != a2[1][7] )
27            v5 = 0;
28        if ( a2[1][1] != a2[1][10] )
29            v5 = 0;
30        if ( a2[1][1] - 17 != a2[1][1] )
31            v5 = 0;
32        if ( a2[1][3] != a2[1][9] )
33            v5 = 0;
34        if ( a2[1][4] != 105 )
35            v5 = 0;
36        if ( a2[1][2] - a2[1][1] != 13 )
37            v5 = 0;
38        if ( a2[1][8] - a2[1][7] != 13 )
39            v5 = 0;
40        if ( v5 ) {
41            printf("Good good!\n");
42            klee_assert(0);
43        }
44        else
45            printf("Try again..\n");
46    }
47    else
48    {
49        printf("Usage: %s <pass>\n", *a2);
50    }
51}
52}
```

<< Collapse sidebar

What framework should I use?

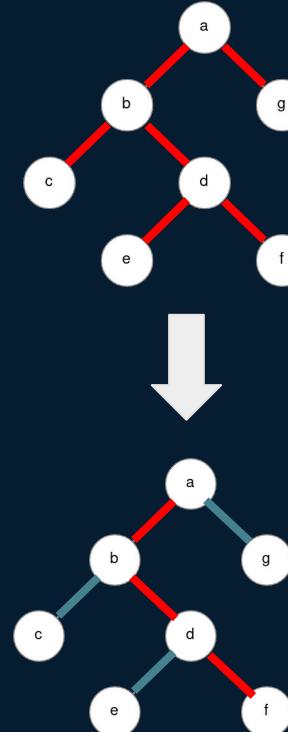


What framework should I use?



Symbolic Execution Recap

- Symbolic execution tries to find inputs that cause a program part to execute
- It works by:
 - traversing an execution tree
 - accumulating constraints at each branch
 - solving them using an SMT solver
- Concolic execution is seed-driven symbolic execution that trades higher performance for potential coverage loss
- There are many symbolic execution frameworks: angr is the best for most CTF challenges



Further Readings



Section 3: Angr

Discovering the **How?**

Section 1: Problem Space

Section 2: Symbolic Execution



angr

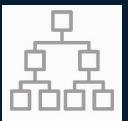


Extensive Binary Analysis Framework



Convenient Python3 Interface

Valgrind



Symbolic + Concolic Execution



Developed by UCSB

Won 3rd in DARPA
Cyber Grand Challenge

Used for reversing,
rop-chain building,
fuzzing and more

A photograph of a waterfall in a dense tropical forest. The waterfall flows down a rocky cliff into a pool of water. The surrounding area is filled with lush green foliage, including various ferns and large leafy plants. The lighting is natural, filtering through the trees.

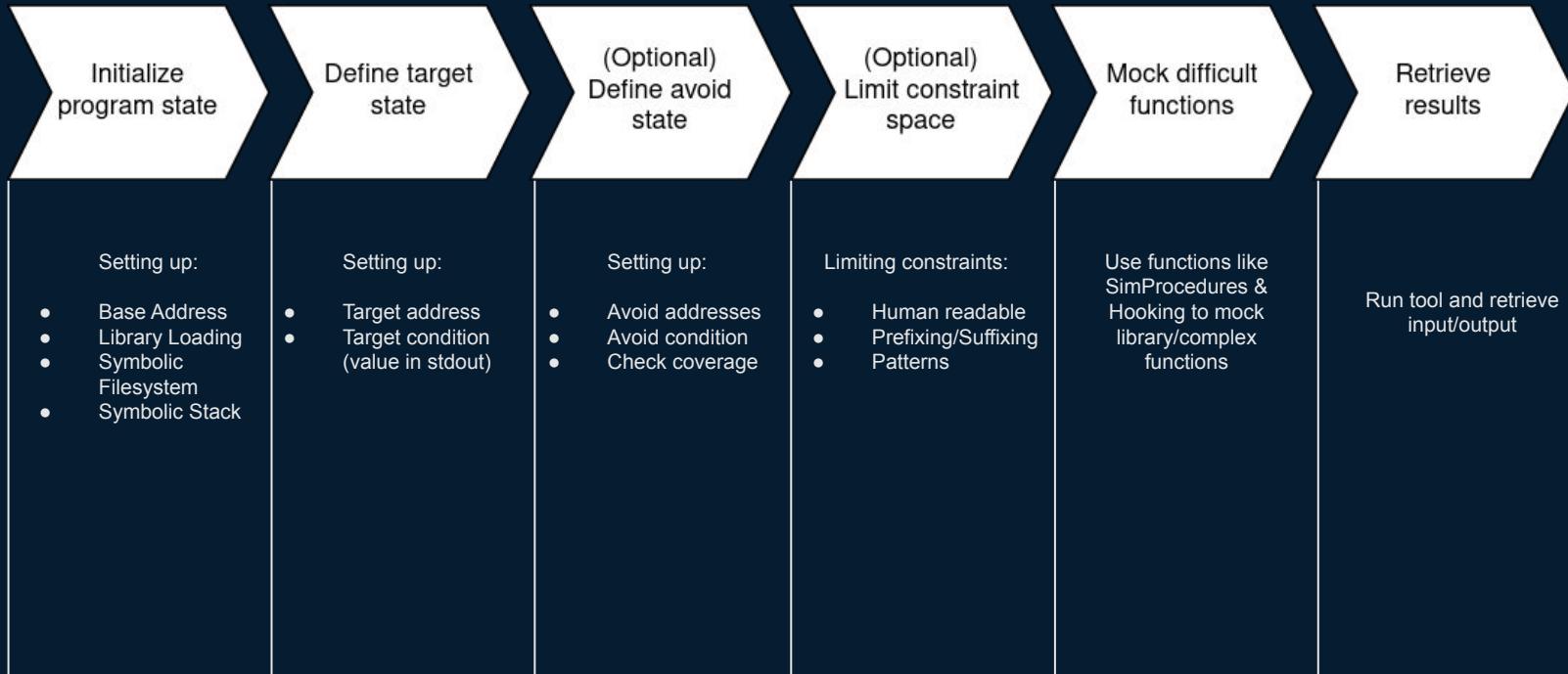
Angr Workflow

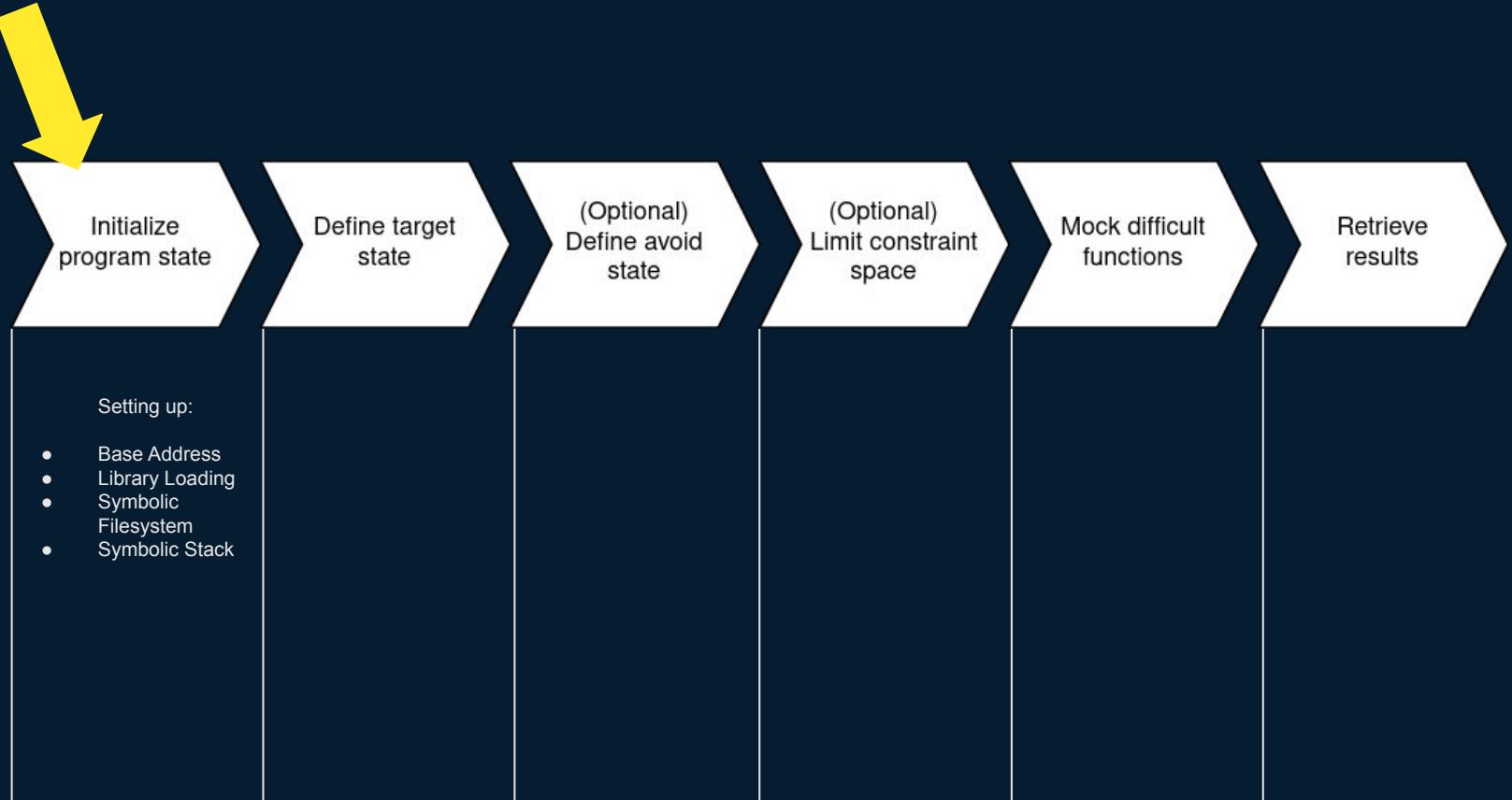
Let's recap for a second



```
kiwi@doghouse: ~/InsomniHack/00_z3
import angr, claripy

proj = angr.Project('./z3_robot', load_options={"auto_load_libs": False}, main_opts={"base_addr": 0})
simgr = proj.factory.simgr()
simgr.explore(find=0x00000140, avoid=0x00000142d)
print(simgr.found[0].posix.dumps()
```





Basic example

Provide input

Validate input
(constraint check
function)

Print result

```
int main()
{
    char input[0x19];
    sym.imp.fgets(input, 0x19, _reloc.stdin);

    int result = check_flag(input);

    if (result == 0) { puts("Solved"); }
    else { puts("Nope"); }

    return 0;
}
```

Basic example

Initialize project

```
import angr, claripy

proj = angr.Project('./z3_robot',
    load_options={"auto_load_libs": False},
    main_opts={"base_addr": 0}
)
```

Basic example

Initialize project

Initialize simulation manager

```
import angr, claripy

proj = angr.Project('./z3_robot',
                    load_options={"auto_load_libs" : False},
                    main_opts={"base_addr":0}
                    )

simgr = proj.factory.simgr()
```

Basic example

Initialize project

Initialize simulation manager

Explore until required address

```
import angr, claripy

proj = angr.Project('./z3_robot',
                    load_options={"auto_load_libs" : False},
                    main_opts={"base_addr":0}
                    )

simgr = proj.factory.simgr()

simgr.explore(find=0x00001407)
```

Basic example

Initialize project

Initialize simulation manager

Explore until required address

Print concretized result

```
import angr, claripy

proj = angr.Project('./z3_robot',
    load_options={"auto_load_libs" : False},
    main_opts={"base_addr":0}
)

simgr = proj.factory.simgr()

simgr.explore(find=0x00001407)

print(simgr.found[0].posix.dumps(0))
```

Managing state

Provide input

Validate input
(constraint check
function)

Print result

```
int main()
{
    char input[0x19];
    sym.imp.fgets(input, 0x19, _reloc.stdin);

    int result = check_flag(input);

    if (result == 0) { puts("Solved"); }
    else { puts("Nope"); }

    return 0;
}
```

Managing state

Time waste
function

Provide input

Validate input
(constraint check
function)

Print result

```
int main()
{
    complicated_timewaste_function(); //sleeps forever

    char input[0x19];
    sym.imp.fgets(input, 0x19, _reloc.stdin);

    int result = check_flag(input);

    if (result == 0) { puts("Solved"); }
    else { puts("Nope"); }

    return 0;
}
```

Managing state

Up to now the initial state was always defined as the binary entry point

We can also specify a custom start address to speed up execution:

- Save time by directly running main
- Skip large function
- Define custom input

```
start_addr = 0x00001337
initial_state = proj.factory.blank_state(addr=start_addr)
simgr = proj.factory.simgr(initial_state)
```

Managing state

Up to now the initial state was always defined as the binary entry point

We can also specify a custom start address to speed up execution:

- Save time by directly running main
- Skip large function
- Define custom input

```
import angr, claripy

proj = angr.Project('./z3_robot',
    load_options={"auto_load_libs": False},
    main_opts={"base_addr":0}
)

start_addr = 0x00001337
initial_state = proj.factory.blank_state(addr=start_addr)
simgr = proj.factory.simgr(initial_state)

simgr.explore(find=0x00001407)

print(simgr.found[0].posix.dumps(0))
```

What if input is...

...complex format string?

...consisting of multiple parameters?

...over memory/file/network?

Custom Symbol Injection

```
password = claripy.BVS('password', 8*input_length)
```

Registers:

```
initial_state.regs.eax = password  
initial_state.regs.ebx = password  
initial_state.regs.edx = password
```

Memory:

```
initial_state.memory.store(password_address, password, endness=project.arch.memory_endness)
```

Stack:

```
initial_state.stack_push(password)
```

Argv:

```
initial_state = project.factory.entry_state(args=["binary_name", password])
```

Symbolic Stack

Provide complex
format string
input

Validate input 1
function

Validate input 2
function

Print result

```
int main()
{
    int input1;
    int input2;

    scanf("%x %x", &input1, &input2);

    int result1 = check_flag1(input1);
    int result2 = check_flag2(input2);

    if ( (result1 == 0) && (result2 == 0) ) { puts("Solved"); }
    else { puts("Nope"); }

    return 0;
}
```

Symbolic Stack

Set start address
after input was
provided

```
start_addr = 0x000013cc
initial_state = proj.factory.blank_state(addr=start_addr)
```

Symbolic Stack

Set start address
after input was
provided

Initialize stack
frame

```
start_addr = 0x000013cc
initial_state = proj.factory.blank_state(addr=start_addr)
initial_state.regs.ebp = initial_state.regs.esp
```

Symbolic Stack

Set start address
after input was
provided

Initialize stack
frame

Define password
bitvectors

```
start_addr = 0x000013cc
initial_state = proj.factory.blank_state(addr=start_addr)

initial_state.regs.ebp = initial_state.regs.esp

password0 = claripy.BVS('password0', 4*8)
password1 = claripy.BVS('password1', 4*8)
```

Symbolic Stack

Set start address
after input was
provided

Initialize stack
frame

Define password
bitvectors

Align stack pointer

```
start_addr = 0x000013cc
initial_state = proj.factory.blank_state(addr=start_addr)

initial_state.regs.ebp = initial_state.regs.esp

password0 = claripy.BVS('password0', 4*8)
password1 = claripy.BVS('password1', 4*8)

padding_length_in_bytes = 0x08
initial_state.regs.esp -= padding_length_in_bytes
```

Symbolic Stack

Set start address
after input was
provided

Initialize stack
frame

Define password
bitvectors

Align stack pointer

Push password
bitvectors to stack

```
start_addr = 0x000013cc
initial_state = proj.factory.blank_state(addr=start_addr)
initial_state.regs.ebp = initial_state.regs.esp
```

```
password0 = claripy.BVS('password0', 4*8)
password1 = claripy.BVS('password1', 4*8)

padding_length_in_bytes = 0x08
initial_state.regs.esp -= padding_length_in_bytes
```

```
initial_state.stack_push(password0)
initial_state.stack_push(password1)
```

Symbolic Stack

Set start address
after input was
provided

Initialize stack
frame

Define password
bitvectors

Align stack pointer

Push password
bitvectors to stack

Solve bitvectors

```
start_addr = 0x000013cc
initial_state = proj.factory.blank_state(addr=start_addr)

initial_state.regs.ebp = initial_state.regs.esp

password0 = claripy.BVS('password0', 4*8)
password1 = claripy.BVS('password1', 4*8)

padding_length_in_bytes = 0x08
initial_state.regs.esp -= padding_length_in_bytes

initial_state.stack_push(password0)
initial_state.stack_push(password1)

simgr = proj.factory.simgr(initial_state)
simgr.explore(find=0x00001407)

solution0 = (simulation.found[0].solver.eval(password0))
solution1 = (simulation.found[0].solver.eval(password1))

print("{0},{1}".format(solution0,solution1))
```

Symbolic Stack

Set start address
after input was
provided

Initialize stack
frame

Define password
bitvectors

Align stack pointer

Push password
bitvectors to stack

Solve bitvectors

```
import angr, claripy

proj = angr.Project('./z3_robot',
    load_options={"auto_load_libs": False},
    main_opts={"base_addr":0}
)

start_addr = 0x000013cc
initial_state = proj.factory.blank_state(addr=start_addr)

initial_state.regs.ebp = initial_state.regs.esp

password0 = claripy.BVS('password0', 4*8)
password1 = claripy.BVS('password1', 4*8)

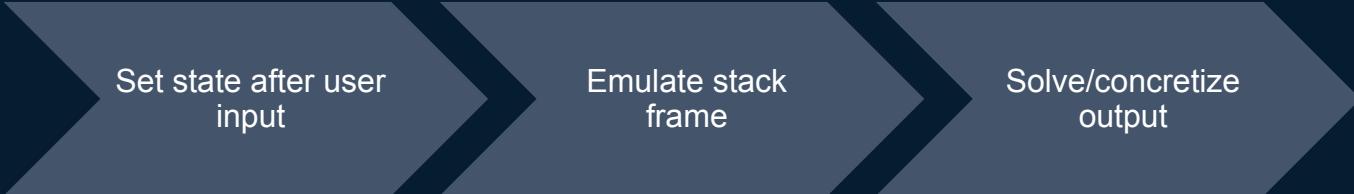
padding_length_in_bytes = 0x08
initial_state.regs.esp -= padding_length_in_bytes

initial_state.stack_push(password0)
initial_state.stack_push(password1)

simgr = proj.factory.simgr(initial_state)
simgr.explore(find=0x00001407)

solution0 = (simulation.found[0].solver.eval(password0))
solution1 = (simulation.found[0].solver.eval(password1))

print("{0},{1}".format(solution0,solution1))
```



Set state after user
input

Emulate stack
frame

Solve/concretize
output

Symbolic Filesystem

Provide input via file

Validate input
(constraint check function)

Print result

```
int main()
{
    FILE *fp;
    char input[0x19];

    fp = fopen("./inputfile.txt", "r");
    fgets(input, 0x19, (FILE*)fp);
    fclose(fp);

    int result = check_flag(input);

    if (result == 0) { puts("Solved"); }
    else { puts("Nope"); }

    return 0;
}
```

```
start_addr = 0x000013cc  
initial_state = proj.factory.blank_state(addr=start_addr)
```

Symbolic Filesystem

Set start address after input

Symbolic Filesystem

Set start address after input

Define symbolic memory

```
start_addr = 0x000013cc
initial_state = proj.factory.blank_state(addr=start_addr)

filename = 'inputfile.txt'
sym_file_size = 64

symbolic_file_backing_memory =
    angr.state_plugins.SimSymbolicMemory()
symbolic_file_backing_memory.set_state(initial_state)
```

Symbolic Filesystem

Set start address after input

Define symbolic memory

Add password bitvector to
symbolic memory

```
start_addr = 0x000013cc
initial_state = proj.factory.blank_state(addr=start_addr)

filename = 'inputfile.txt'
sym_file_size = 64

symbolic_file_backing_memory =
    angr.state_plugins.SimSymbolicMemory()
symbolic_file_backing_memory.set_state(initial_state)

password = claripy.BVS('password', sym_file_size * 8)
symbolic_file_backing_memory.store(0, password)
```

Symbolic Filesystem

Set start address after input

Define symbolic memory

Add password bitvector to
symbolic memory

Add a symbolic file (SimFile)
based on symbolic memory

```
start_addr = 0x000013cc
initial_state = proj.factory.blank_state(addr=start_addr)

filename = 'inputfile.txt'
sym_file_size = 64

symbolic_file_backing_memory =
    angr.state_plugins.SimSymbolicMemory()
symbolic_file_backing_memory.set_state(initial_state)

password = claripy.BVS('password', sym_file_size * 8)
symbolic_file_backing_memory.store(0, password)

password_file = angr.storage.SimFile(filename, 'r',
content=symbolic_file_backing_memory,
size=symbolic_file_size_bytes
)
```

Symbolic Filesystem

Set start address after input

Define symbolic memory

Add password bitvector to
symbolic memory

Add a symbolic file (SimFile)
based on symbolic memory

Define symbolic filesystem and
add SimFile

```
start_addr = 0x000013cc
initial_state = proj.factory.blank_state(addr=start_addr)

filename = 'inputfile.txt'
sym_file_size = 64

symbolic_file_backing_memory =
    angr.state_plugins.SimSymbolicMemory()
symbolic_file_backing_memory.set_state(initial_state)

password = claripy.BVS('password', sym_file_size * 8)
symbolic_file_backing_memory.store(0, password)

password_file = angr.storage.SimFile(filename, 'r',
    content=symbolic_file_backing_memory,
    size=symbolic_file_size_bytes
)

symbolic_filesystem = {
    filename : password_file
}
initial_state.posix.fs = symbolic_filesystem
```

Symbolic Filesystem

Set start address after input

Define symbolic memory

Add password bitvector to
symbolic memory

Add a symbolic file (SimFile)
based on symbolic memory

Define symbolic filesystem and
add SimFile

Solve symbolic memory

```
start_addr = 0x000013cc
initial_state = proj.factory.blank_state(addr=start_addr)

filename = 'inputfile.txt'
sym_file_size = 64

symbolic_file_backing_memory =
    angr.state_plugins.SimSymbolicMemory()
symbolic_file_backing_memory.set_state(initial_state)

password = claripy.BVS('password', sym_file_size * 8)
symbolic_file_backing_memory.store(0, password)

password_file = angr.storage.SimFile(filename, 'r',
    content=symbolic_file_backing_memory,
    size=symbolic_file_size_bytes
)

symbolic_filesystem = {
    filename : password_file
}
initial_state.posix.fs = symbolic_filesystem

simgr = proj.factory.simgr(initial_state)
simgr.explore(find=0x00001407)

solution = (simulation.found[0].solve.eval(password,cast_to=str))
print(solution)
```

Symbolic Filesystem

Set start address after input

Define symbolic memory

Add password bitvector to
symbolic memory

Add a symbolic file (SimFile)
based on symbolic memory

Define symbolic filesystem and
add SimFile

Solve symbolic memory

```
...
start_addr = 0x000013cc
initial_state = proj.factory.blank_state(addr=start_addr)

filename = 'inputfile.txt'
sym_file_size = 64

symbolic_file_backing_memory =
    angr.state_plugins.SimSymbolicMemory()
symbolic_file_backing_memory.set_state(initial_state)

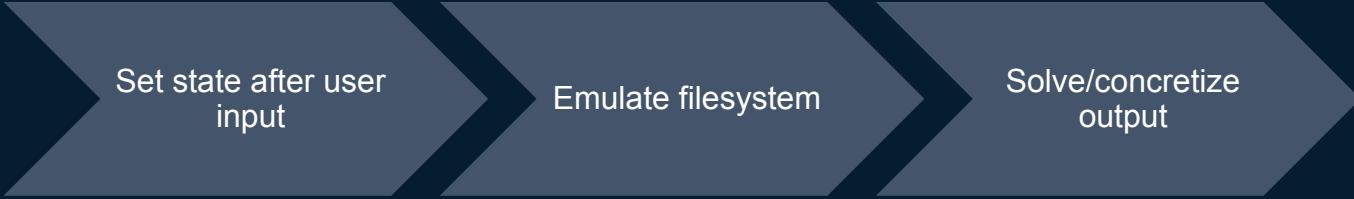
password = claripy.BVS('password', sym_file_size * 8)
symbolic_file_backing_memory.store(0, password)

password_file = angr.storage.SimFile(filename, 'r',
    content=symbolic_file_backing_memory,
    size=symbolic_file_size_bytes
)

symbolic_filesystem = {
    filename : password_file
}
initial_state.posix.fs = symbolic_filesystem

simgr = proj.factory.simgr(initial_state)
simgr.explore(find=0x00001407)

solution = (simulation.found[0].solve.eval(password,cast_to=str))
print(solution)
```

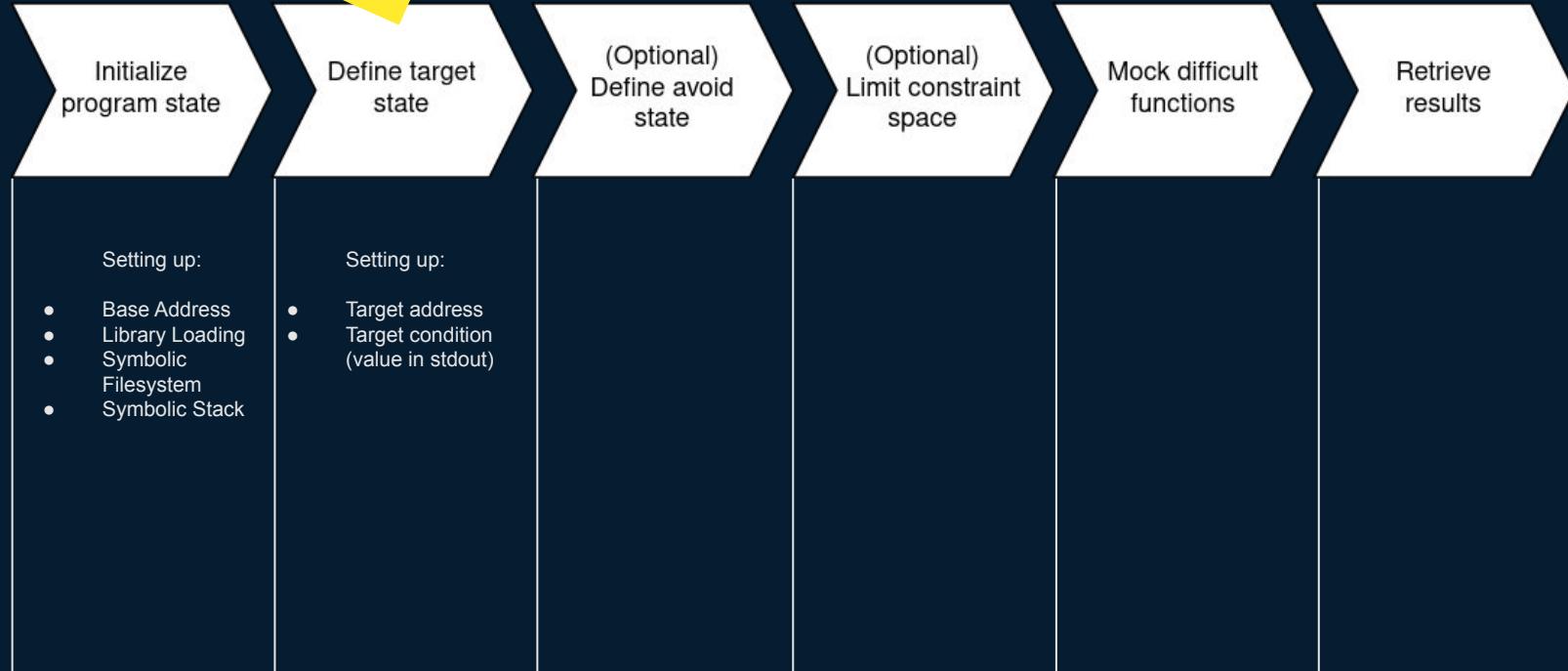


```
graph LR; A[Set state after user input] --> B[Emulate filesystem]; B --> C[Solve/concretize output]
```

Set state after user input

Emulate filesystem

Solve/concretize output



Target state definition

Define target address(es)

Explore until solution is found
or whole graph was explored

```
simgr = proj.factory.simgr()  
simgr.explore(find=0x00001407)
```

Target state definition

Define target address(es)

Explore until solution is found
or whole graph was explored

```
import angr, claripy

proj = angr.Project('./z3_robot',
    load_options={"auto_load_libs": False},
    main_opts={"base_addr": 0}
)

simgr = proj.factory.simgr()
simgr.explore(find=0x00001407)

print(simgr.found[0].posix.dumps(0))
```

Can also be value

Sometimes your target is not necessarily an address

You can also specify arbitrary conditions for finding/avoiding conditions

A common use-case is setting your target based on values written to stdout

```
def is_successful(state):
    stdout_output = state.posix.dumps(sys.stdout.fileno())
    return 'Solved' in stdout_output

simgr.explore(
    find=is_successful
)
```

Can also be value

Sometimes your target is not necessarily an address

You can also specify arbitrary conditions for finding/avoiding conditions

A common use-case is setting your target based on values written to stdout

```
import angr, claripy

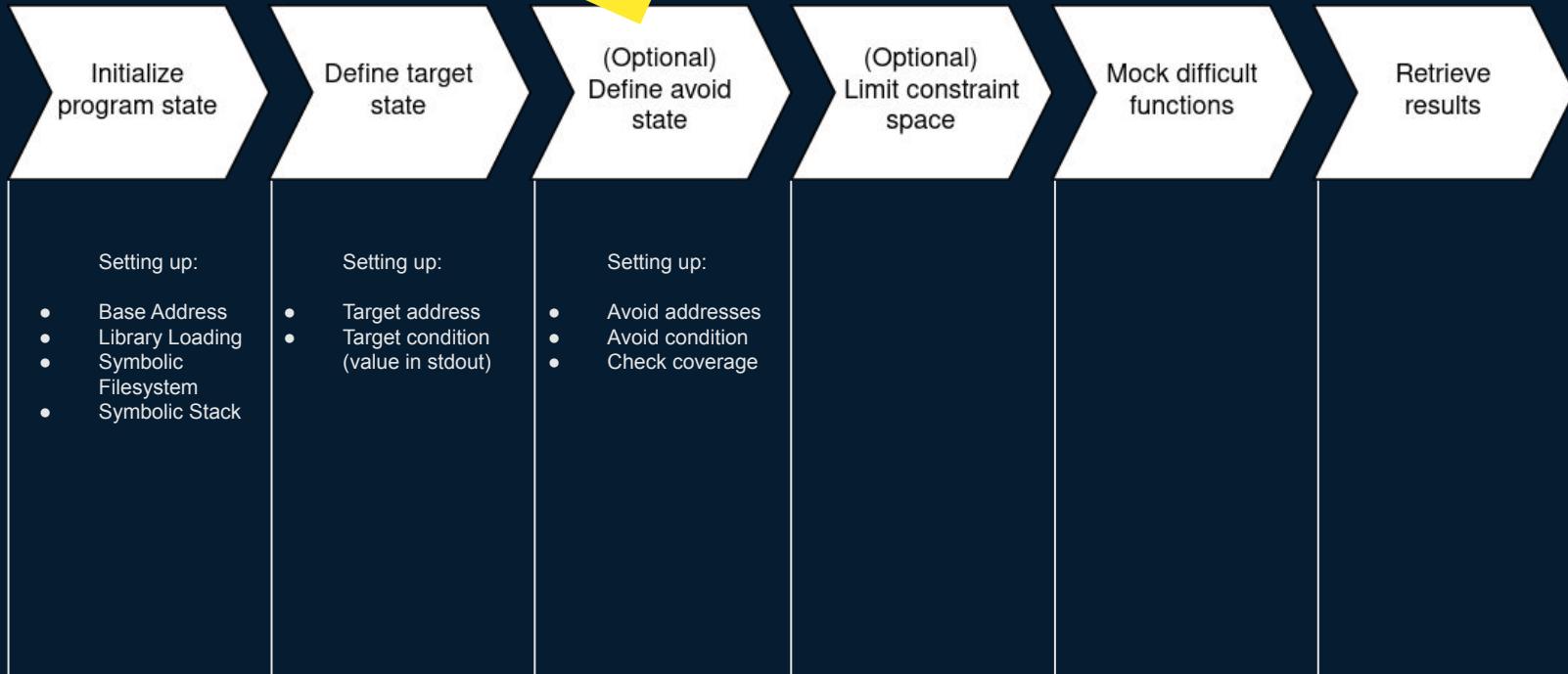
proj = angr.Project('./z3_robot',
    load_options={"auto_load_libs": False},
    main_opts={"base_addr": 0}
)

simgr = proj.factory.simgr()

def is_successful(state):
    stdout_output = state.posix.dumps(sys.stdout.fileno())
    return 'Solved' in stdout_output

simgr.explore(
    find=is_successful
)

print(simgr.found[0].posix.dumps(sys.stdin.fileno()))
```



State Explosion



Branches double per condition

Growth of problem is exponential
relating to program size

Slows down symbolic execution

Just exclude, it's easy

A great way to reduce complexity
is by entirely avoiding unneeded
paths

Selecting those paths works best
with reverse engineering & human
intuition

```
simgr.explore(find=0x00001407, avoid=[0x0000142d])
```

Just exclude, it's easy

A great way to reduce complexity
is by entirely avoiding unneeded
paths

Selecting those paths works best
with reverse engineering & human
intuition

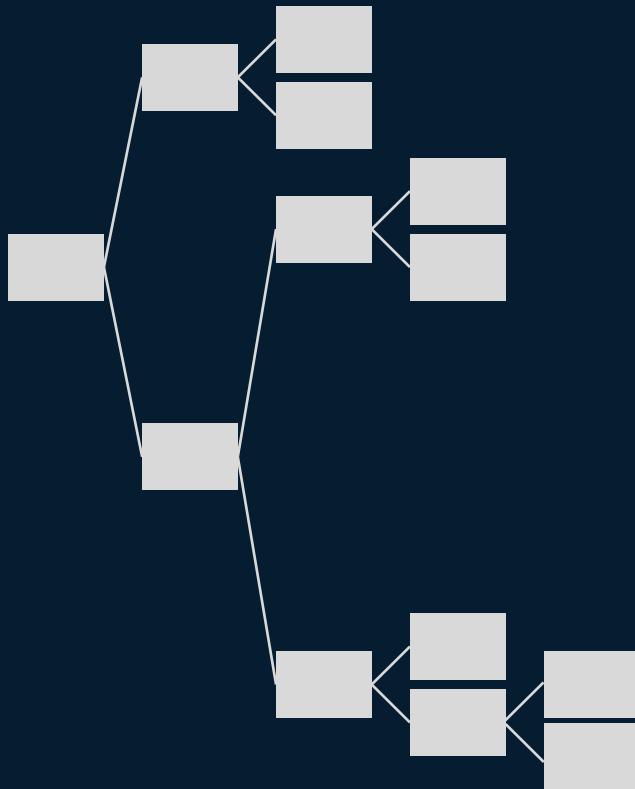
```
import angr, claripy

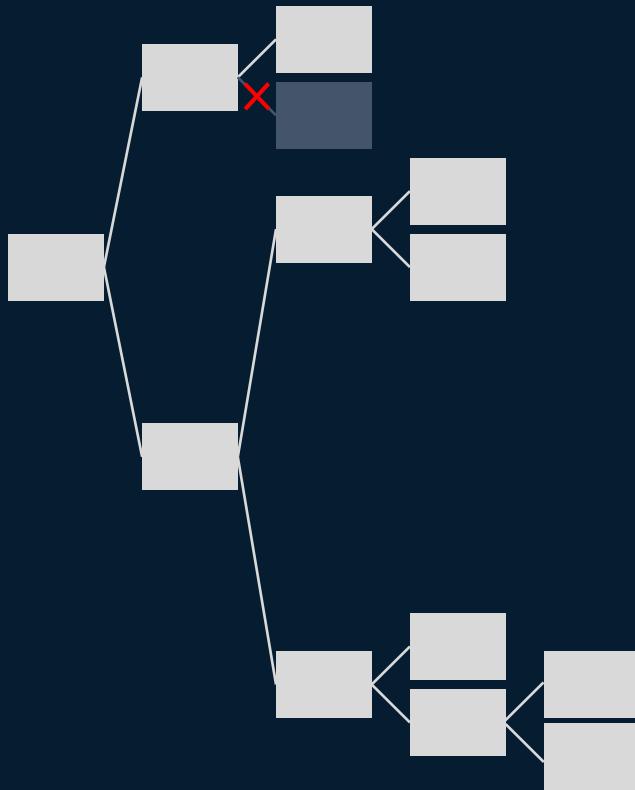
proj = angr.Project('./z3_robot',
    load_options={"auto_load_libs": False},
    main_opts={"base_addr": 0})

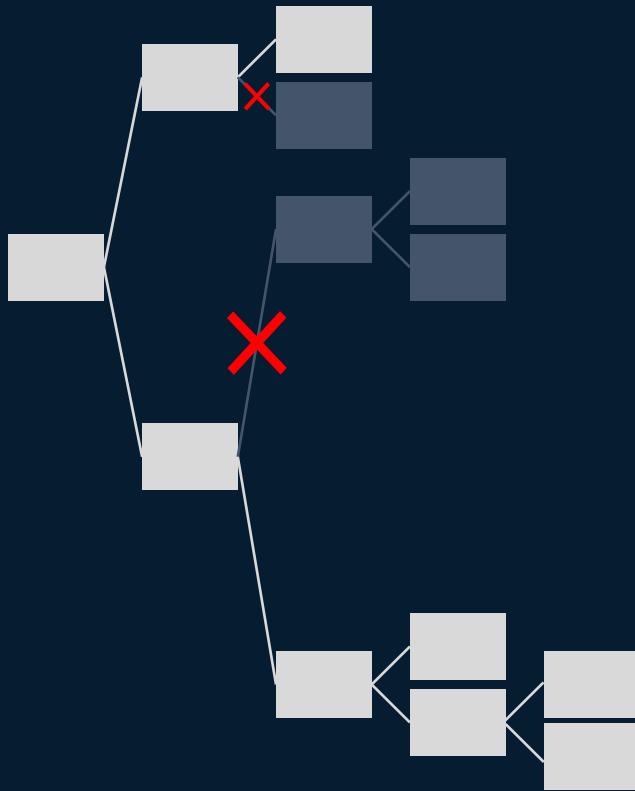
simgr = proj.factory.simgr()

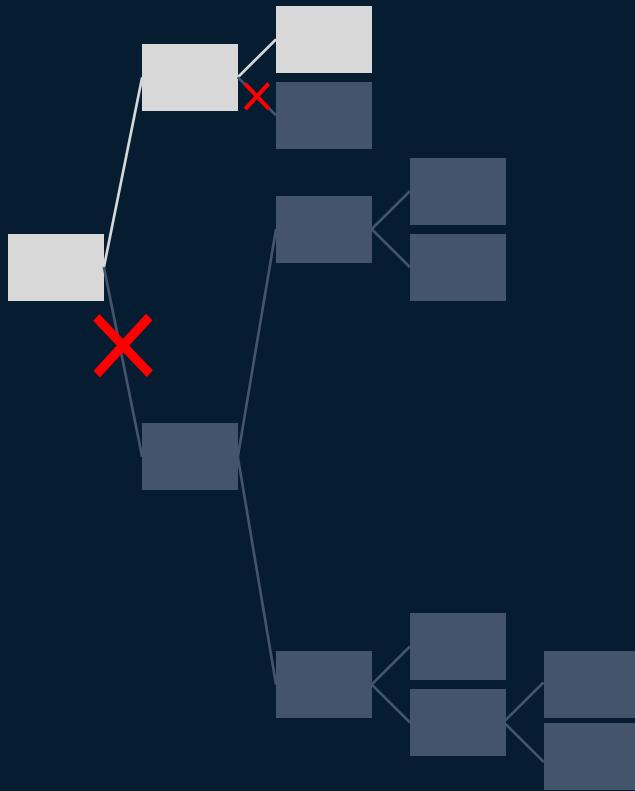
simgr.explore(find=0x00001407, avoid=[0x0000142d])

print(simgr.found[0].posix.dumps(0))
```

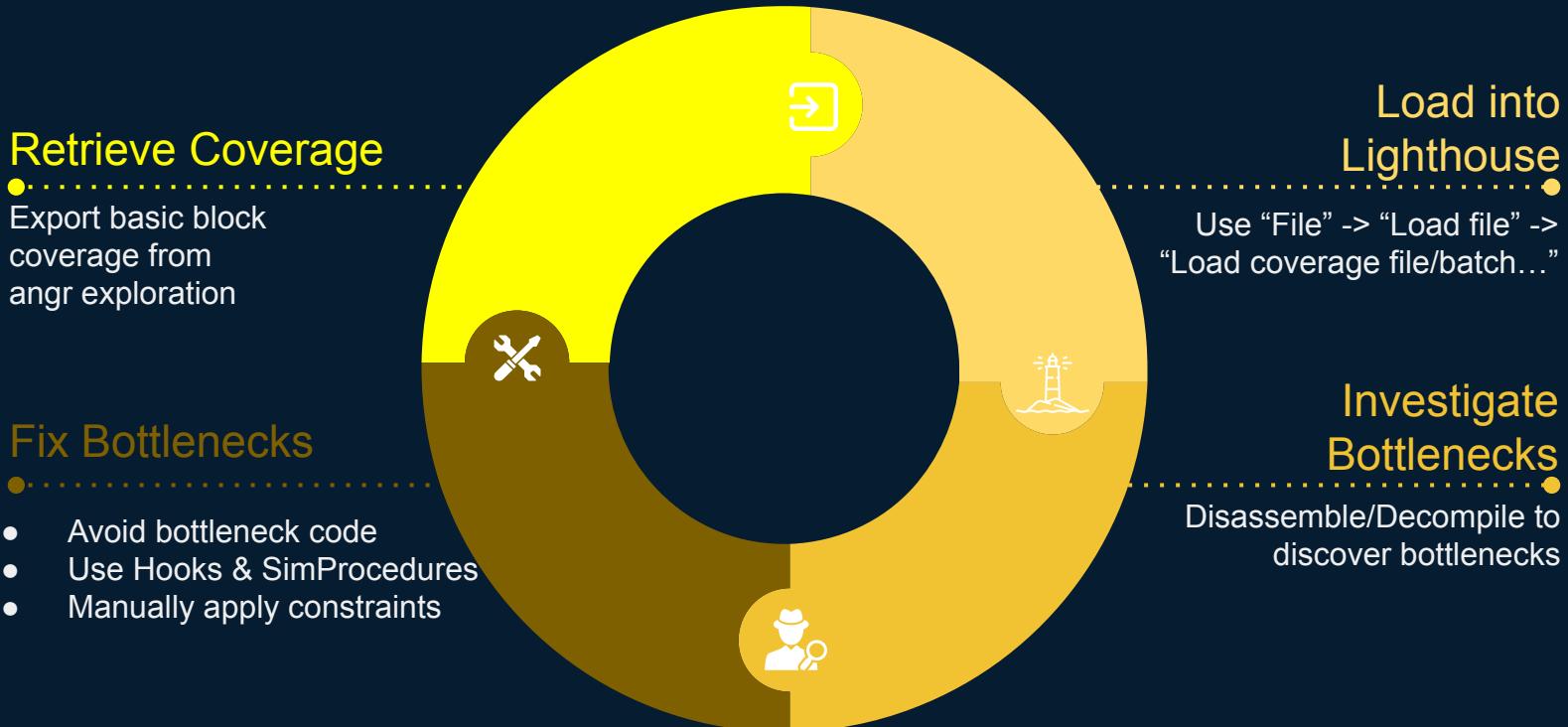








Code Coverage Collection Process



Code Coverage

```
def get_small_coverage(*args, **kwargs):
    sm = args[0]
    stashes = sm.stashes
    i = 0
    for simstate in stashes["active"]:
        state_history = ""

        for addr in simstate.history.bbl_addrs.hardcopy:
            write_address = hex(addr)
            state_history += "{0}\n".format(write_address)
        raw_syminput = simstate.posix.stdin.load(0, state.posix.stdin.size)

        syminput = simstate.solver.eval(raw_syminput, cast_to=bytes)
        print(syminput)
        ip = hex(state.solver.eval(simstate.ip))
        uid = str(uuid.uuid4())
        sid = str(i).zfill(5)
        filename = "{0}_active_{1}_{2}_{3}".format(sid,syminput, ip, uid)

        with open(filename, "w") as f:
            f.write(state_history)
        i += 1

    simgr.explore(find=0x00001407, step_func=get_small_coverage)
```

Load into lighthouse to find bottlenecks...

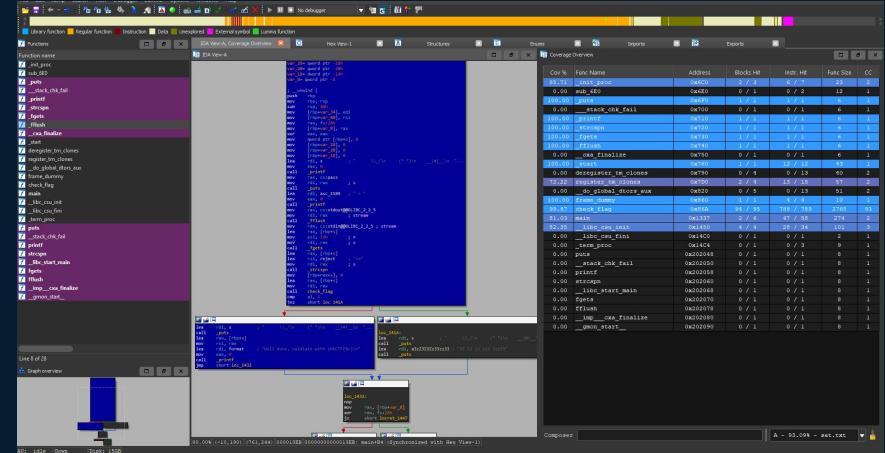
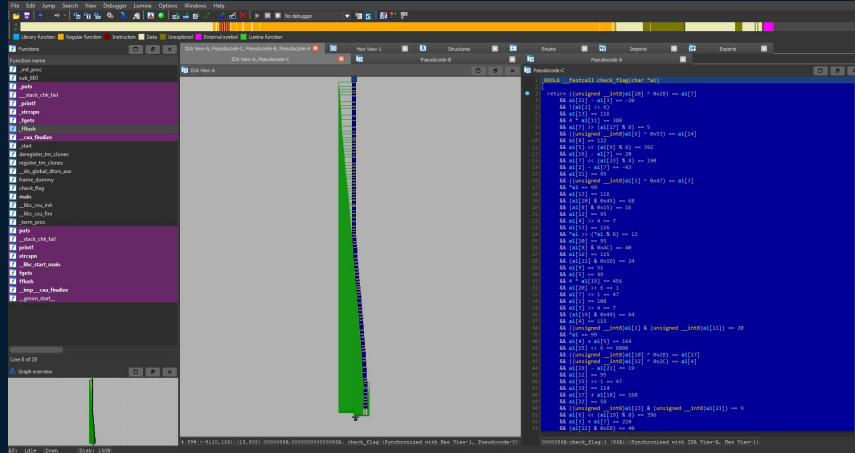
The screenshot shows the IDA Pro interface with several windows open. The main window displays assembly code for the `_fastcall check_flag(char *)` function. The code uses a series of `88` and `88 4` instructions to manipulate memory at addresses `a1[20]`, `a1[13]`, `a1[11]`, `a1[7]`, `a1[14]`, `a1[16]`, `a1[17]`, `a1[23]`, `a1[10]`, `a1[12]`, `a1[19]`, `a1[18]`, and `a1[15]`. The assembly code includes various arithmetic operations like addition, subtraction, multiplication, and division.

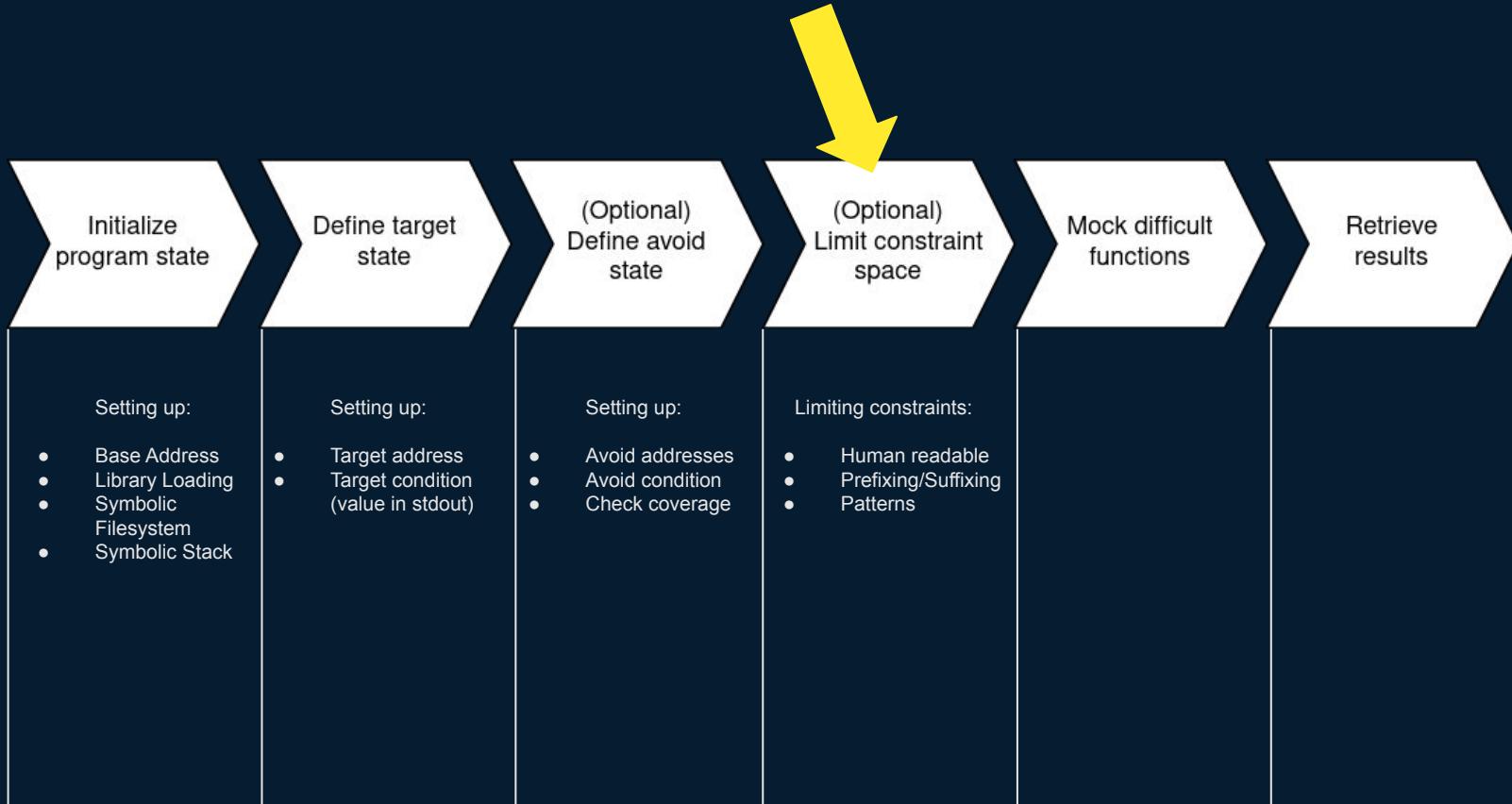
Below the assembly window, the `Graph overview` window shows a complex flowchart with many nodes and connections, indicating the control flow of the program.

The right side of the interface features a `Coverage Overview` window. This window lists 55 functions along with their coverage statistics. The columns include Cov %, Func Name, Address, Blocks Hit, Instr. Hit, Func Size, and CC. The functions listed include `init_proc`, `puts`, `_stack_chk_fail`, `printf`, `strcspn`, `fgets`, `fflush`, `_cxa_finalize`, `_start`, `deregister_tm_clones`, `register_tm_clones`, `_do_global_dtors_aux`, `frame_dummy`, `check_flag`, `main`, `_libc_csu_init`, `_libc_csu_fini`, `_termproc`, `puts`, `_stack_chk_fail`, `printf`, `strcspn`, `_libc_start_main`, `fgets`, `fflush`, `_imp_cxa_finalize`, and `_gmon_start_`. The `check_flag` function has the highest coverage at 95%, while `main` has the lowest at 1%.

At the bottom of the interface, there are status bars for `Composer` and `B - 3.61% - unsat.txt`.

...and guide angr into resolving them





import angr, claripy



Limiting Constraints

Import the constraint solver engine

Limiting Constraints

Import the constraint solver engine

Create new symbolic password bitvector

Create state and pass bitvector to it (argv,
symbolic stack, symbolic file...)

```
import angr, claripy
```

```
proj = angr.Project('./z3_robot',
    load_options={"auto_load_libs": False},
    main_opts={"base_addr": 0}
)
```

```
password = claripy.BVS("password", 8*8) #8 chars
initial_state = proj.factory.entry_state(args=["crackme", password])
```

Limiting Constraints

Import the constraint solver engine

Create new symbolic password bitvector

Create state and pass bitvector to it (argv,
symbolic stack, symbolic file...)

Add custom constraints to bitvector:

- Only printable characters

```
import angr, claripy
```

```
proj = angr.Project('./z3_robot',
    load_options={"auto_load_libs": False},
    main_opts={"base_addr": 0}
)
```

```
password = claripy.BVS("password", 8*8) #8 chars
initial_state = proj.factory.entry_state(args=["crackme", password])
```

```
# only printable characters
for byte in password.chop(8):
    initial_state.add_constraints(byte != '\x00') # null
    initial_state.add_constraints(byte >= ' ') # '\x20'
    initial_state.add_constraints(byte <= '~') # '\x7e'
```

Limiting Constraints

Import the constraint solver engine

Create new symbolic password bitvector

Create state and pass bitvector to it (argv, symbolic stack, symbolic file...)

Add custom constraints to bitvector:

- Only printable characters
- Password starts with "CTF{"

```
import angr, claripy
```

```
proj = angr.Project('./z3_robot',
    load_options={"auto_load_libs": False},
    main_opts={"base_addr": 0}
)

password = claripy.BVS("password", 8*8) #8 chars
initial_state = proj.factory.entry_state(args=["crackme", password])

# only printable characters
for byte in password.chop(8):
    initial_state.add_constraints(byte != '\x00') # null
    initial_state.add_constraints(byte >= ' ') # '\x20'
    initial_state.add_constraints(byte <= '~') # '\x7e'

# starts with CTF{
initial_state.add_constraints(password.chop(8)[0] == 'C')
initial_state.add_constraints(password.chop(8)[1] == 'T')
initial_state.add_constraints(password.chop(8)[2] == 'F')
initial_state.add_constraints(password.chop(8)[3] == '{')
```

Limiting Constraints

Import the constraint solver engine

Create new symbolic password bitvector

Create state and pass bitvector to it (argv, symbolic stack, symbolic file...)

Add custom constraints to bitvector:

- Only printable characters
- Password starts with "CTF{"

Solve bitvector to get password

```
import angr, claripy
```

```
proj = angr.Project('./z3_robot',
    load_options={"auto_load_libs": False},
    main_opts={"base_addr": 0}
)
```

```
password = claripy.BVS("password", 8*8) #8 chars
initial_state = proj.factory.entry_state(args=["crackme", password])
```

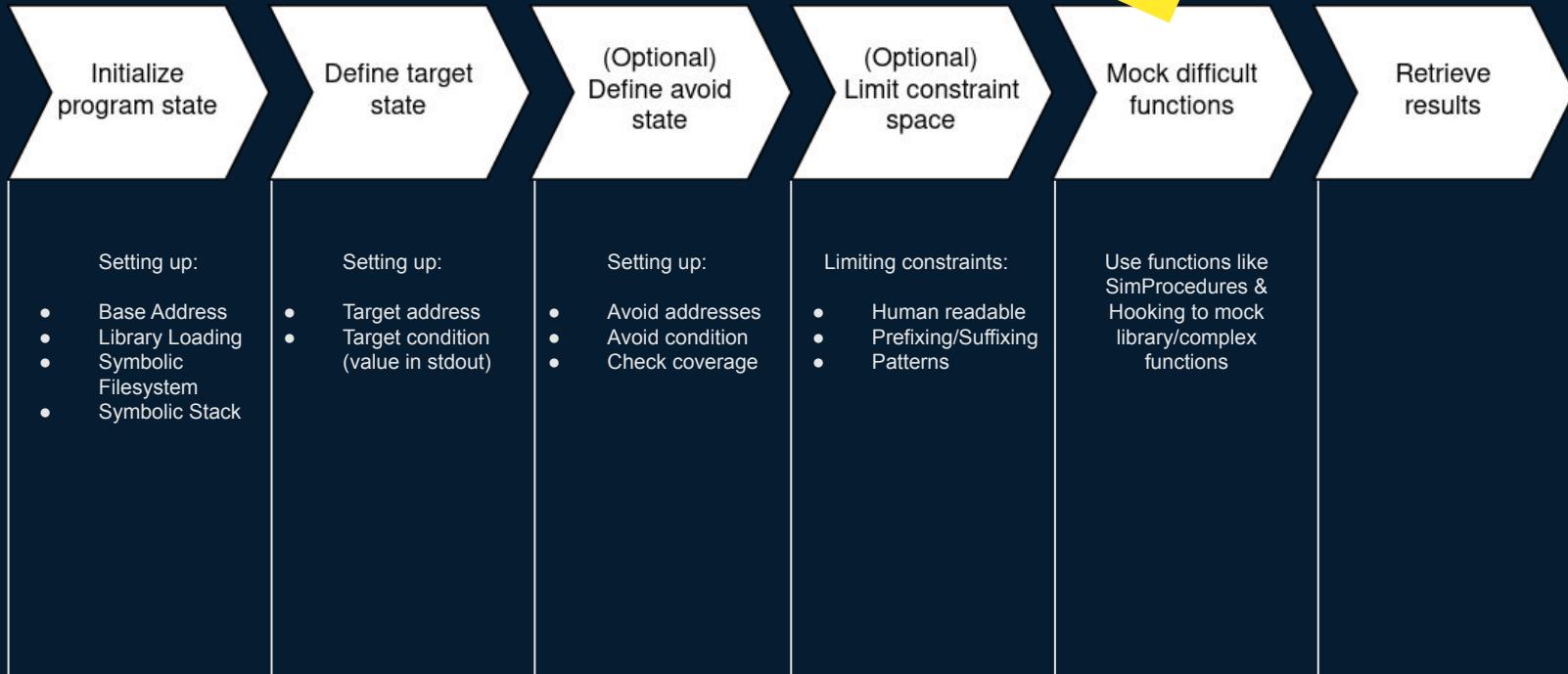
```
# only printable characters
for byte in password.chop(8):
    initial_state.add_constraints(byte != '\x00') # null
    initial_state.add_constraints(byte >= ' ') # '\x20'
    initial_state.add_constraints(byte <= '~') # '\x7e'
```

```
# starts with CTF{
initial_state.add_constraints(password.chop(8)[0] == 'C')
initial_state.add_constraints(password.chop(8)[1] == 'T')
initial_state.add_constraints(password.chop(8)[2] == 'F')
initial_state.add_constraints(password.chop(8)[3] == '{')
```

```
simgr = proj.factory.simgr(initial_state)
```

```
simgr.explore(find=0x00001407)
```

```
print(simgr.found[0].solver.eval(password, cast_to=bytes))
```



SimProcedures

You can use SimProcedures to
overwrite binary functions with
python code

This helps with controlling
complicated, low-level library
functions

For example useful to overwrite
secure PRNG with insecure
implementation/static values

```
class NewOverwrittenFunc(angr.SimProcedure):
    # arguments automatically extracted
    def run(self, argc, argv):
        if argc > 0:
            print("This is python code now {0}".format(argv[0]))
        return 0
    return 1
```

```
proj.hook_symbol('function_to_overwrite', NewOverwrittenFunc())
```

SimProcedures

You can use SimProcedures to
overwrite binary functions with
python code

This helps with controlling
complicated, low-level library
functions

For example useful to overwrite
secure PRNG with insecure
implementation/static values

```
import angr, claripy
```

```
class NewOverwrittenFunc(angr.SimProcedure):
    # arguments automatically extracted
    def run(self, argc, argv):
        if argc > 0:
            print("This is python code now {0}".format(argv[0]))
        return 0
    return 1
```

```
proj = angr.Project('./z3_robot',
                    load_options={"auto_load_libs" : False},
                    main_opts={"base_addr":0}
                    )
proj.hook_symbol('function_to_overwrite', NewOverwrittenFunc())
```

```
simgr = proj.factory.simgr()
```

```
simgr.explore(find=0x00001407)
```

```
print(simgr.found[0].posix.dumps(0))
```

User Hooks

User Hooks can be used if
overwriting a whole function
seems to extensive
(SimProcedure)

Just specify at what address to
hook and how many bytes to
skip

```
# length determines how many bytes get skipped/overwritten
@proj.hook(0x1337, length=5)
def set_rax(state):
    state.regs.rax = 1
```

User Hooks

User Hooks can be used if overwriting a whole function seems to extensive (SimProcedure)

Just specify at what address to hook and how many bytes to skip

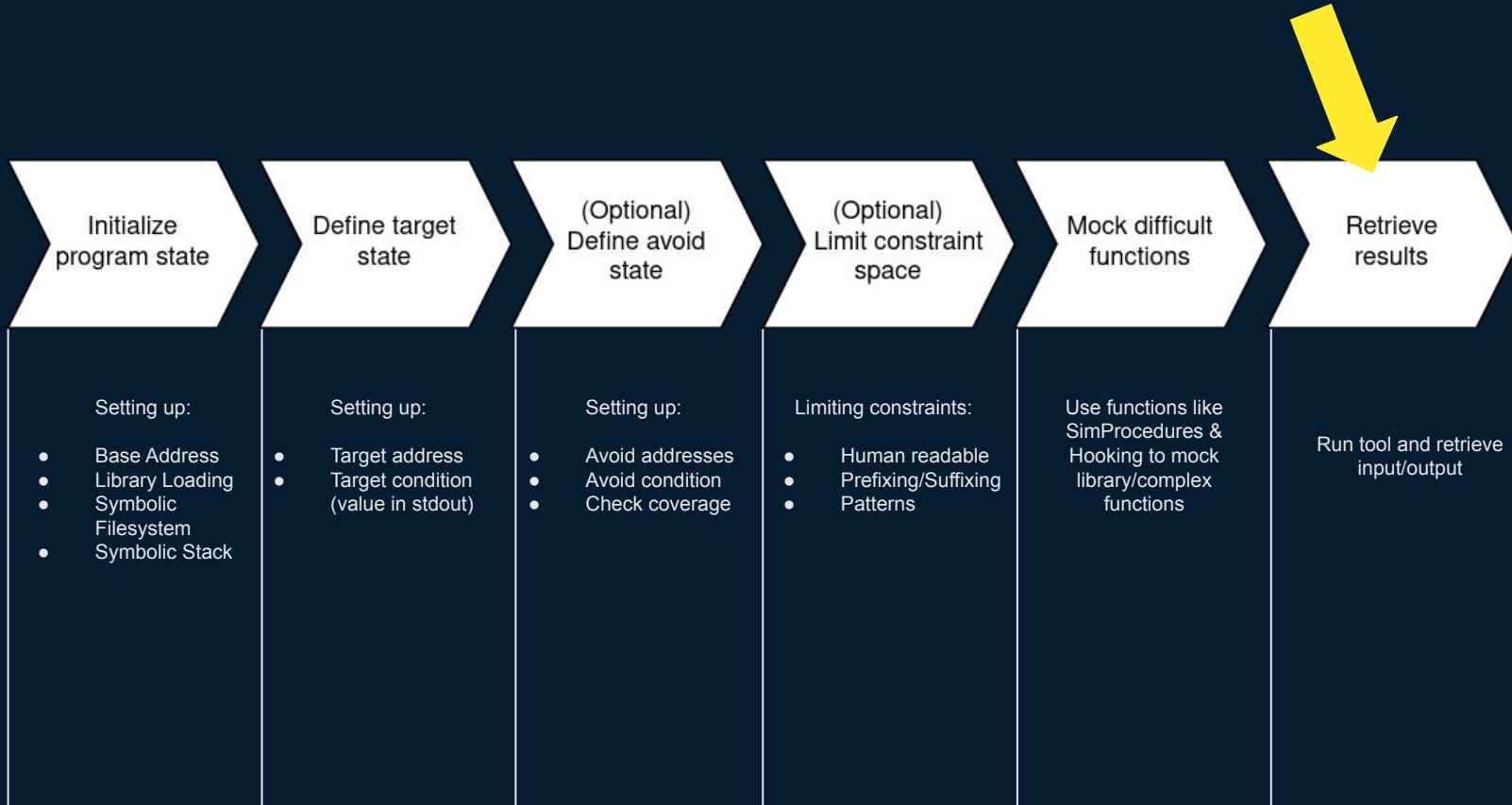
```
import angr, claripy

proj = angr.Project('./z3_robot',
    load_options={"auto_load_libs": False},
    main_opts={"base_addr": 0}
)

simgr = proj.factory.simgr()

# length determines how many bytes get skipped/overwritten
@proj.hook(0x1337, length=5)
def set_rax(state):
    state.regs.rax = 1

simgr.explore(find=0x00001407)
print(simgr.found[0].posix.dumps(0))
```



Concretizing results

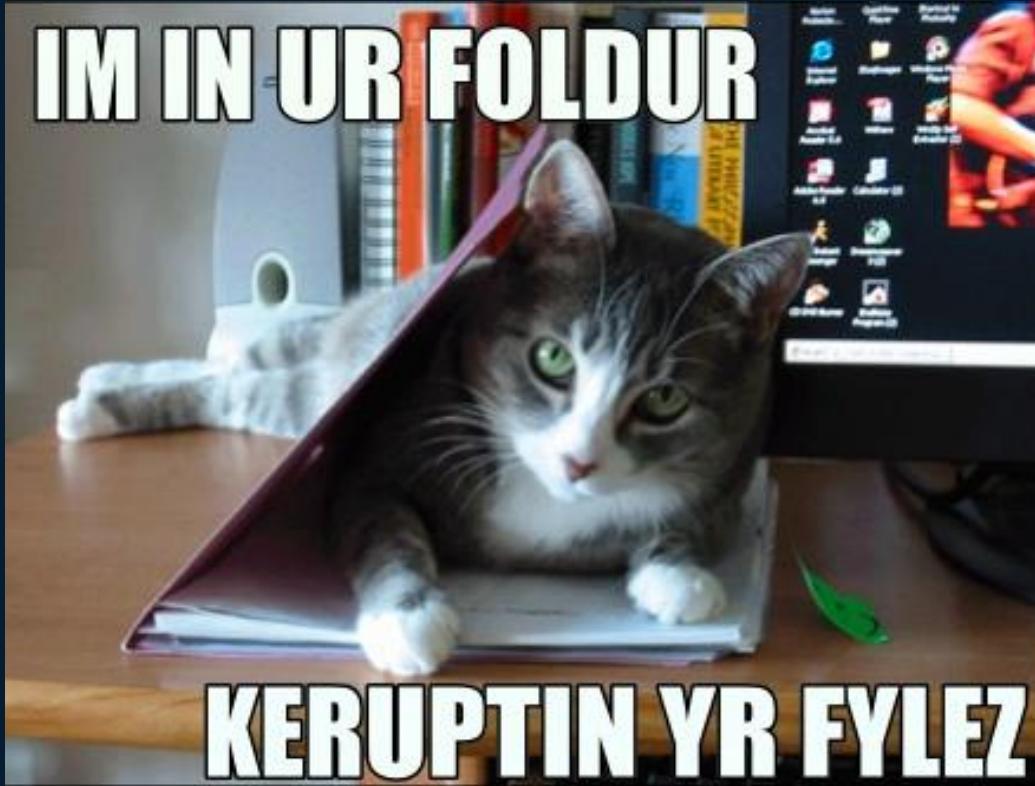
After simulation manager has found a satisfied result you can dump stdin or evaluate your symbolic bitvector

```
simgr.found[0].posix.dumps(0)
```

```
simgr.found[0].posix.dumps(sys.stdin.fileno())
```

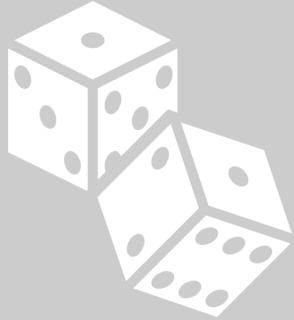
```
simgr.found[0].solver.eval(your_bitvector, cast_to=bytes)
```


IM IN UR FOLDUR



KERUPTIN YR FYLEZ

Limitations



Non-deterministic
control flow



State explosion
causing exponential
growth



Cryptographic
primitives are still
valid



Improve Performance

Veritesting

Algorithm to
automatically reduce
state explosions

Relies on heuristics to
merge states

```
simgr = proj.factory.simgr(initial_state, veritesting=True)
```

Veritesting

Algorithm to
automatically reduce
state explosions

Relies on heuristics to
merge states

```
import angr, claripy

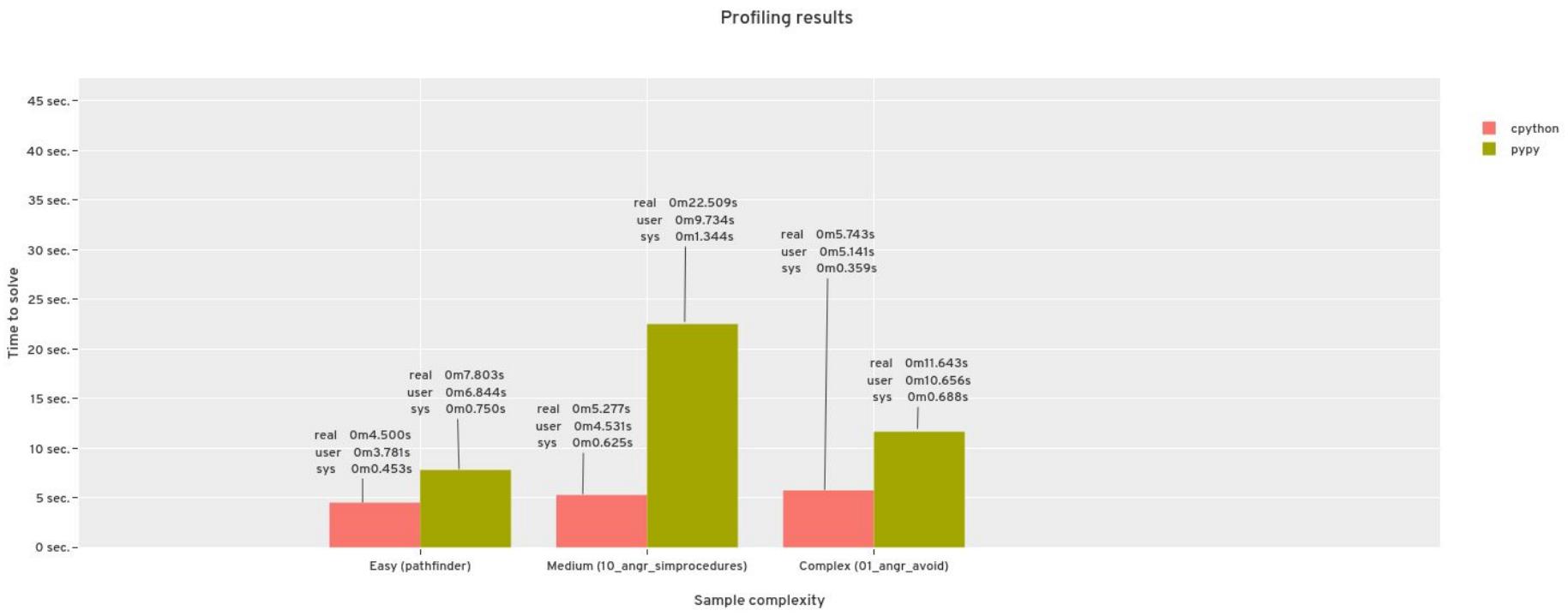
proj = angr.Project('./z3_robot',
    load_options={"auto_load_libs": False},
    main_opts={"base_addr": 0}
)

initial_state = project.factory.entry_state()
simgr = proj.factory.simgr(initial_state, veritesting=True)

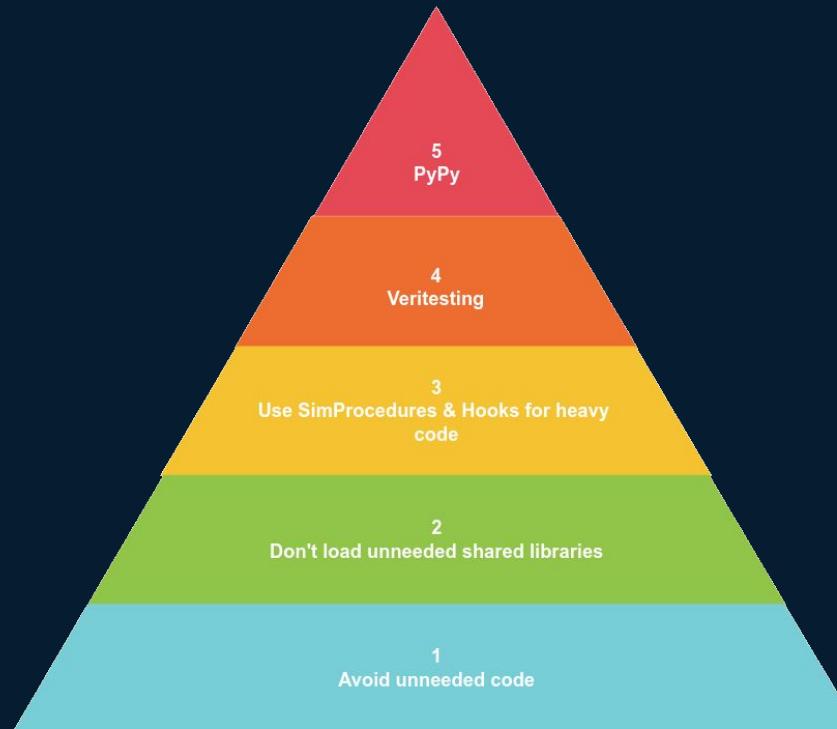
simgr.explore(find=0x00001407)

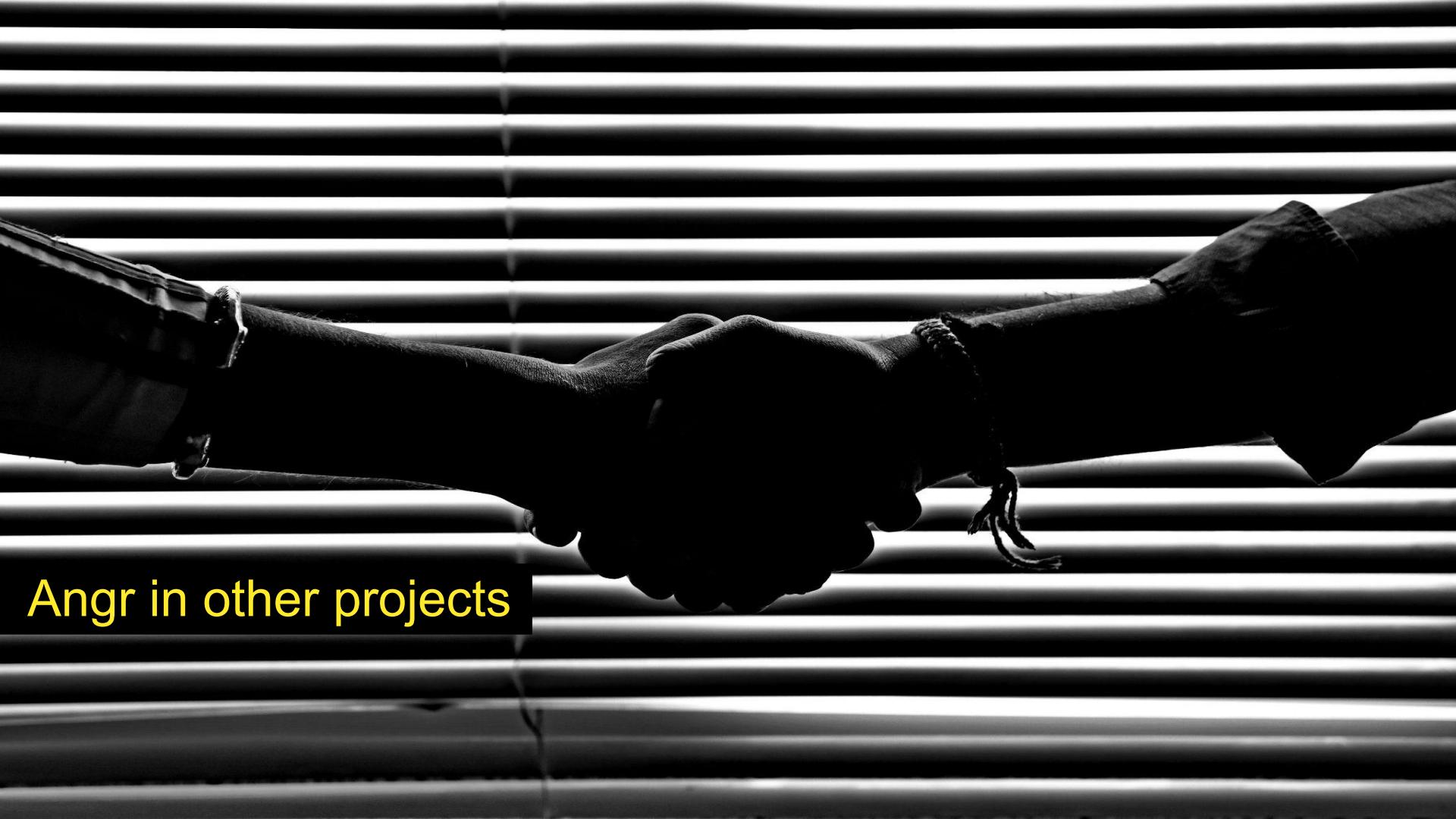
print(simgr.found[0].posix.dumps(0))
```

PyPy



Maslow's Hierarchy of Symbolic Execution

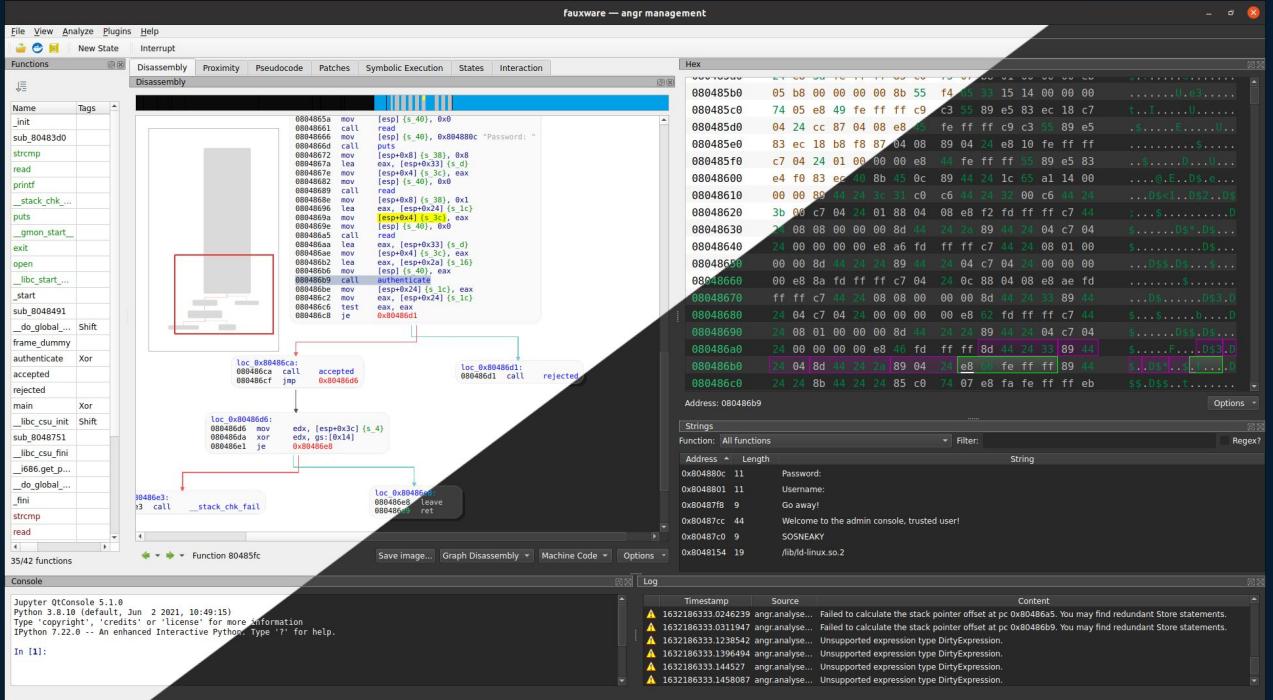




Angr in other projects

Angr-Management

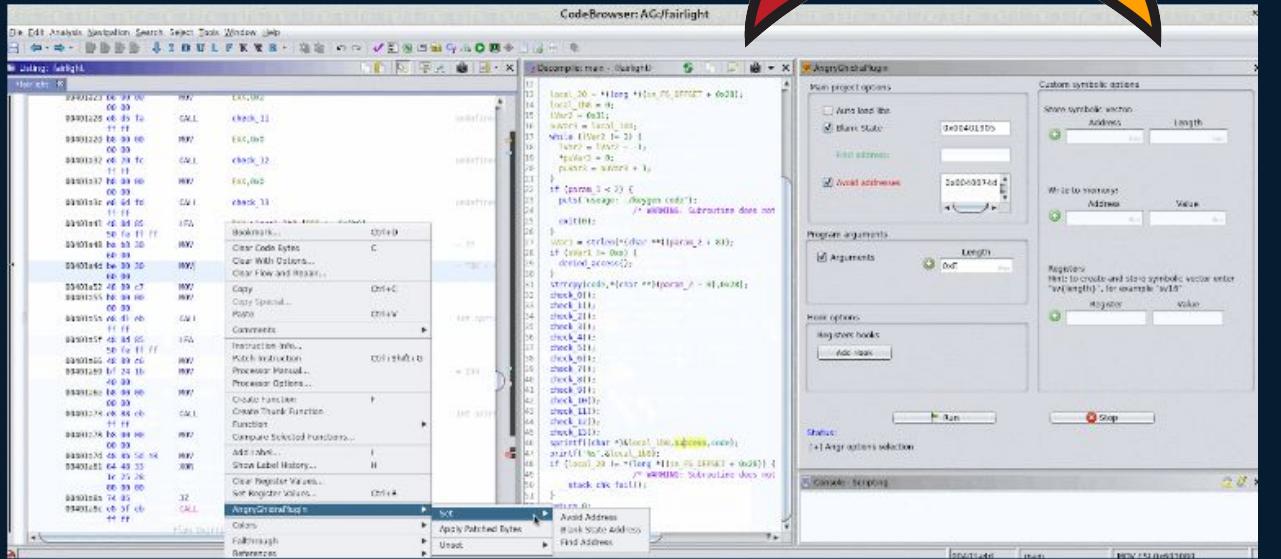
The official GUI to angr,
useful for reverse
engineering and binary
analysis



<https://github.com/angr/angr-management>

AngryGhidra

A plugin that combines the convenience of ghidra with the power of the angr framework



<https://github.com/Nalen98/AngryGhidra>

Driller

Augments the afl-fuzz capabilities with symbolic execution to discover new, interesting paths

american fuzzy lop 1.86b (test)	
process timing run time : 0 days, 0 hrs, 0 min, 2 sec last new path : none seen yet last uniq crash : 0 days, 0 hrs, 0 min, 2 sec last uniq hang : none seen yet	overall results cycles done : 0 total paths : 1 uniq crashes : 1 uniq hangs : 0
cycle progress now processing : 0 (0.00%) paths timed out : 0 (0.00%)	map coverage map density : 2 (0.00%) count coverage : 1.00 bits/tuple
stage progress now trying : havoc stage execs : 1464/5000 (29.28%) total execs : 1697 exec speed : 626.5/sec	findings in depth favored paths : 1 (100.00%) new edges on : 1 (100.00%) total crashes : 39 (1 unique) total hangs : 0 (0 unique)
fuzzing strategy yields bit flips : 0/16, 1/15, 0/13 byte flips : 0/2, 0/1, 0/0 arithmetics : 0/112, 0/25, 0/0 known ints : 0/10, 0/28, 0/0 dictionary : 0/0, 0/0, 0/0 havoc : 0/0, 0/0 trim : n/a, 0.00%	path geometry levels : 1 pending : 1 pend fav : 1 own finds : 0 imported : n/a variable : 0
[cpu: 92%]	

<https://github.com/shellphish/driller>

r4ge

We all like radare2/rizin, now
you can use angr functionalities
straight from your favorite
reverse engineering framework

```
0x0040056b    4883ec10    sub rsp, 0x10
;-- rip:
0x0040056f    897dfc      mov dword [local_4h], edi
0x00400572    488975f0      mov qword [local_10h], rsi
0x00400576    4888b45f0     mov rax, qword [local_10h]
0x0040057a    4883c008     add rax, 8
0x0040057e    488b00      mov rax, qword [rax]      ; orax
0x00400581    488d35bc0000  lea rsi, str.LosFuzzys ; 0x400644 ;
0x00400588    4889c7      mov rdi, rax      ; orax
0x0040058b    e8f0feffff   call sym.imp.strcmp ;[1] ; int strcmp
0x00400590    85c0        test eax, eax      ; r11; r11
0x00400592    750e        jne 0x4005a2      ;[2] ; likely
;-- r4ge.find:
0x00400594    488d3db30000  lea rdi, str.your_are_a_advanced_Hacker_ ;[3] ; int puts
0x0040059b    e8d0feffff   call sym.imp.puts      ; int puts
;-- r4ge.avoid:
0x004005a0    eb0c        jmp 0x4005ae      ;[4]
0x004005a2    488d3dc30000  lea rdi, str.try_Harder_ ; 0x40066c ;
0x004005a9    e8c2feffff   call sym.imp.puts      ;[3] ; int puts
; JMP XREF from 0x004005a0 (main)
0x004005ae    b800000000  mov eax, 0      ; r12; rsp
0x004005b3    c9          leave      ; r13
0x004005b4    c3          ret       ; r13
Press <enter> to return to Visual mode.0.  nop word cs:[rax + rax]
:> .(r4ge)
WARNING | 2017-07-15 15:17:10,199 | claripy | Claripy is setting the recursion limit
start r4ge in DYNAMIC mode...
setup Stack: 0xffff7542d000-0xffff7542ae80, size: 8576
No Heap section
symbolic address: 0xffff7542c4cf, size: 15
start symbolic execution, find:0x400594, avoid:['0x4005a9']

PathGroup Results: <PathGroup with 1 avoid, 1 active, 1 found>
symbolic memory - str: LosFuzzys , hex: 0x4c6f7346757a7a7973000000000000
You want to set debugsession to find address (y/n)? █
```

<https://github.com/gast04/r4ge>

Further learning

https://github.com/jakespringer/angr_ctf

Code Issues Pull requests Actions Projects Wiki Security Insights

master 2 branches 0 tags Go to file Code

jakespringer Fix off-by-k errors for levels 16, 17 f82a7fe on Mar 19, 2018 142 commits

00_angr_find	Remove message in 00_angr_find	5 years ago
01_angr_avoid	Clean up repository	5 years ago
02_angr_find_condition	Clean up repository	5 years ago
03_angr_symbolic_registers	Replace deprecated calls to any_int/any_str w eval	4 years ago
04_angr_symbolic_stack	13_iso_scand now scanf, 04 has two BVs not four	4 years ago
05_angr_symbolic_memory	Replace deprecated calls to any_int/any_str w eval	4 years ago
06_angr_symbolic_dynamic_memory	Replace deprecated calls to any_int/any_str w eval	4 years ago
07_angr_symbolic_file	Replace deprecated calls to any_int/any_str w eval	4 years ago
08_angr_constraints	Clean up repository	5 years ago
09_angr_hooks	Fixed angr.Hook -> project.hook bug in 09	4 years ago
10_angr_simprocedures	Clean up repository	5 years ago
11_angr_sim_scanf	Clean up repository	5 years ago
12_angr_veritesting	Clean up repository	5 years ago
13_angr_static_binary	13_iso_scand now scanf, 04 has two BVs not four	4 years ago

About

No description, website, or topics provided.

Readme GPL-3.0 License 492 stars 17 watching 96 forks

Releases No releases published

Packages No packages published

Contributors 2

jakespringer Jacob Springer
wu4

Let's solve a crackme!



InsomniHack 2022
Jannis Kirschner



25.03.2022

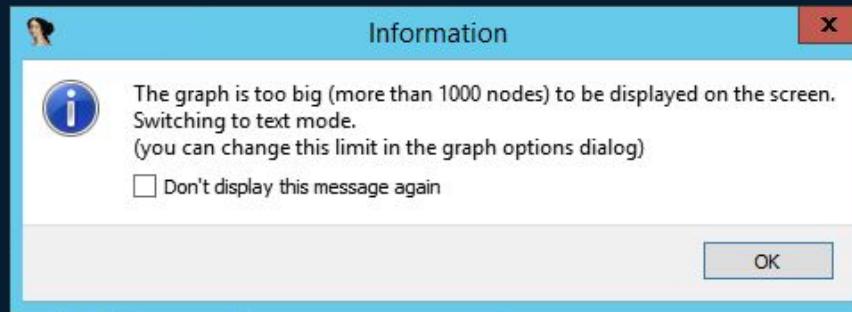
Let's solve a crackme! (Again)

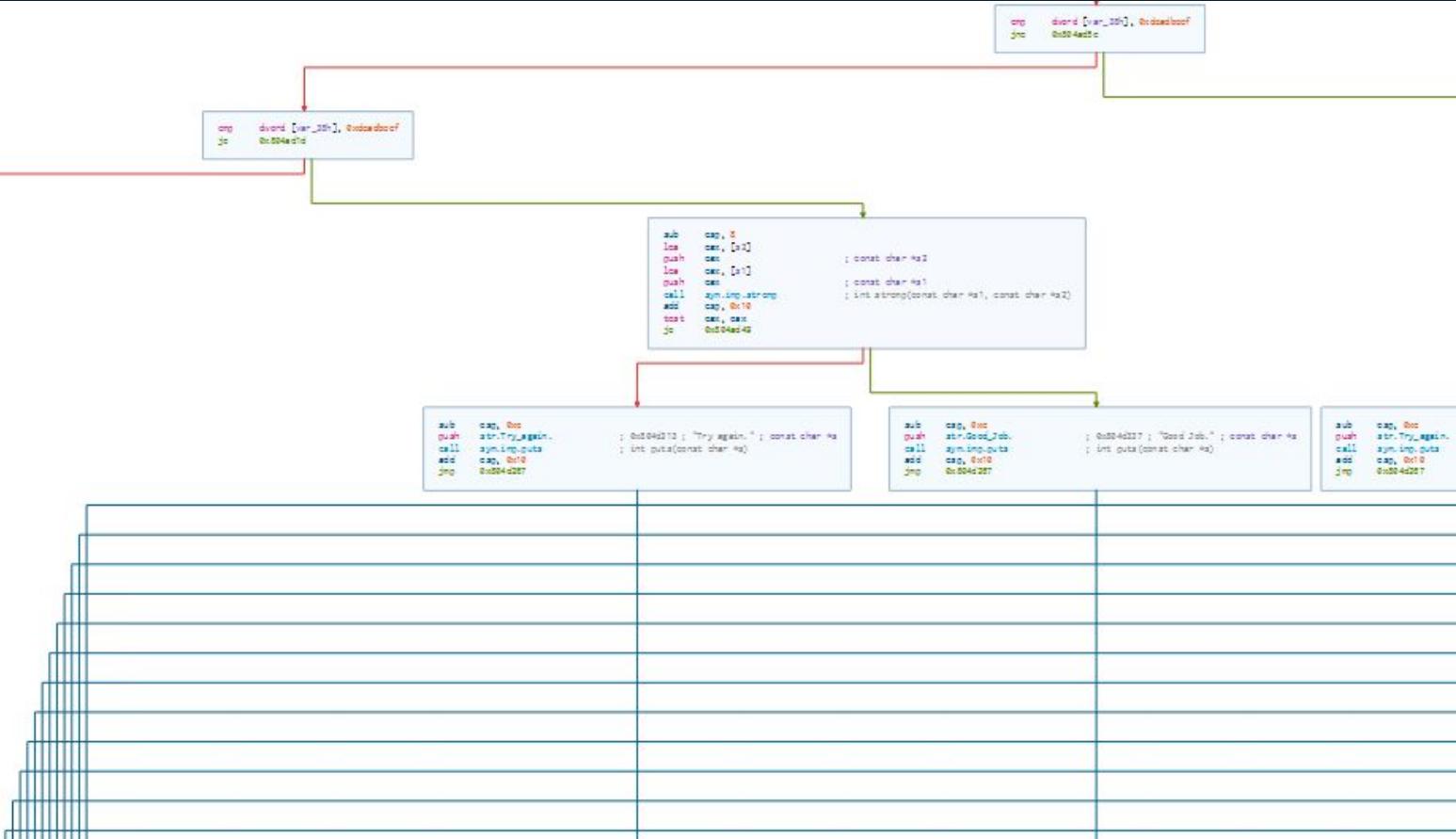


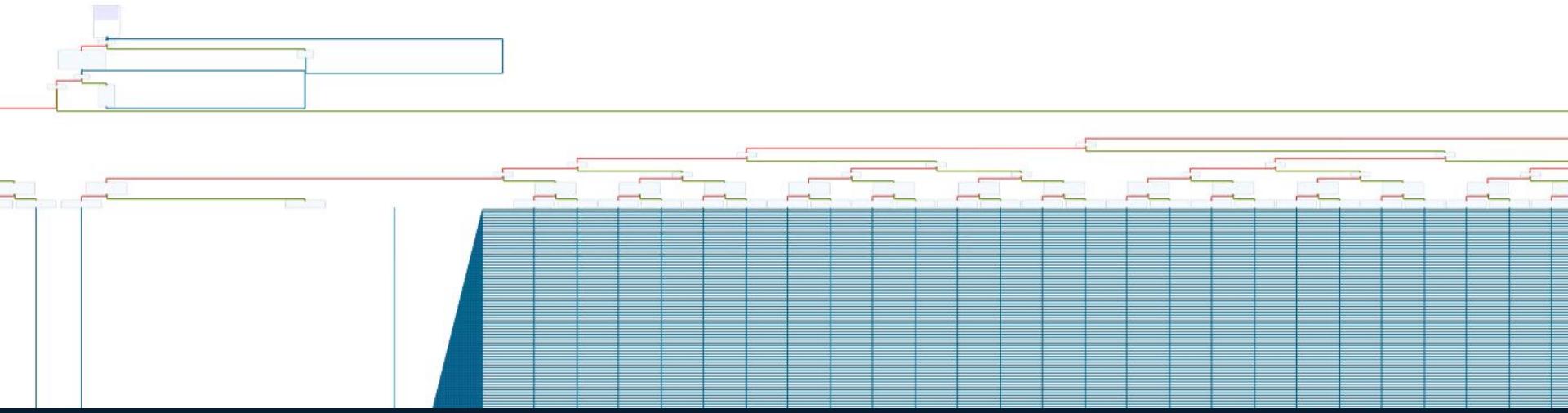
Insomni'Hack 2022
Jannis Kirschner

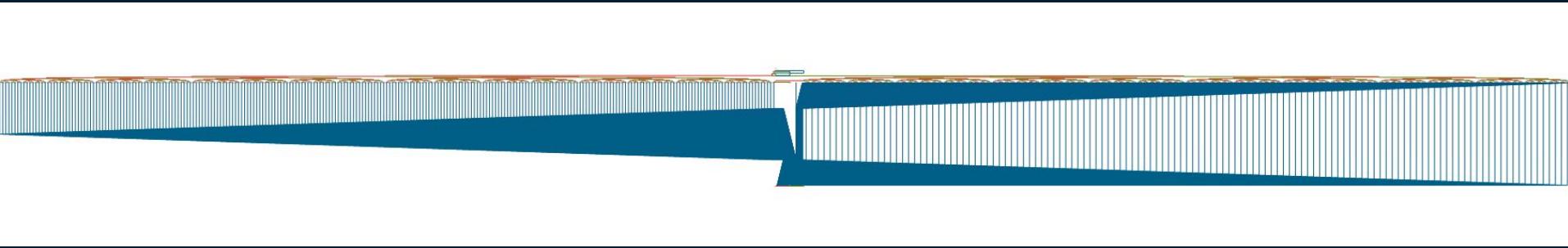


25.03.2022









Angr Recap

- The angr framework features a nice python3 api
- To reach your desired condition you'll need to reduce state explosion
 - You can avoid code, hook, and manually guide the framework
- Angr is incorporated into many tools from advanced fuzzers to modern binary analysis suites
- Symbolic Execution isn't magic though
 - We have to keep performance limitations in mind

Build “Symbolic Execution Harness”



Continuously monitor and improve performance (avoiding, hooking, manual constraints...)



Run to retrieve your flag

Demo



Your princess is in another castle

- Download slides from here: [https://github.com/JannisKirschner/
SymbolicExecutionDemystified](https://github.com/JannisKirschner/SymbolicExecutionDemystified)
- Work through angr_ctf
- Pwn all the CTFs!!!



 @xorkiwi

 /in/janniskirschner

Section 1: Problem Space

Section 2: Symbolic Execution

Section 3: Angr

