

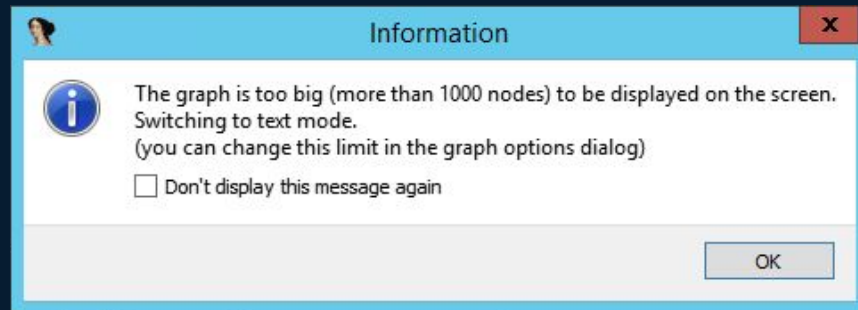
Let's solve a crackme!

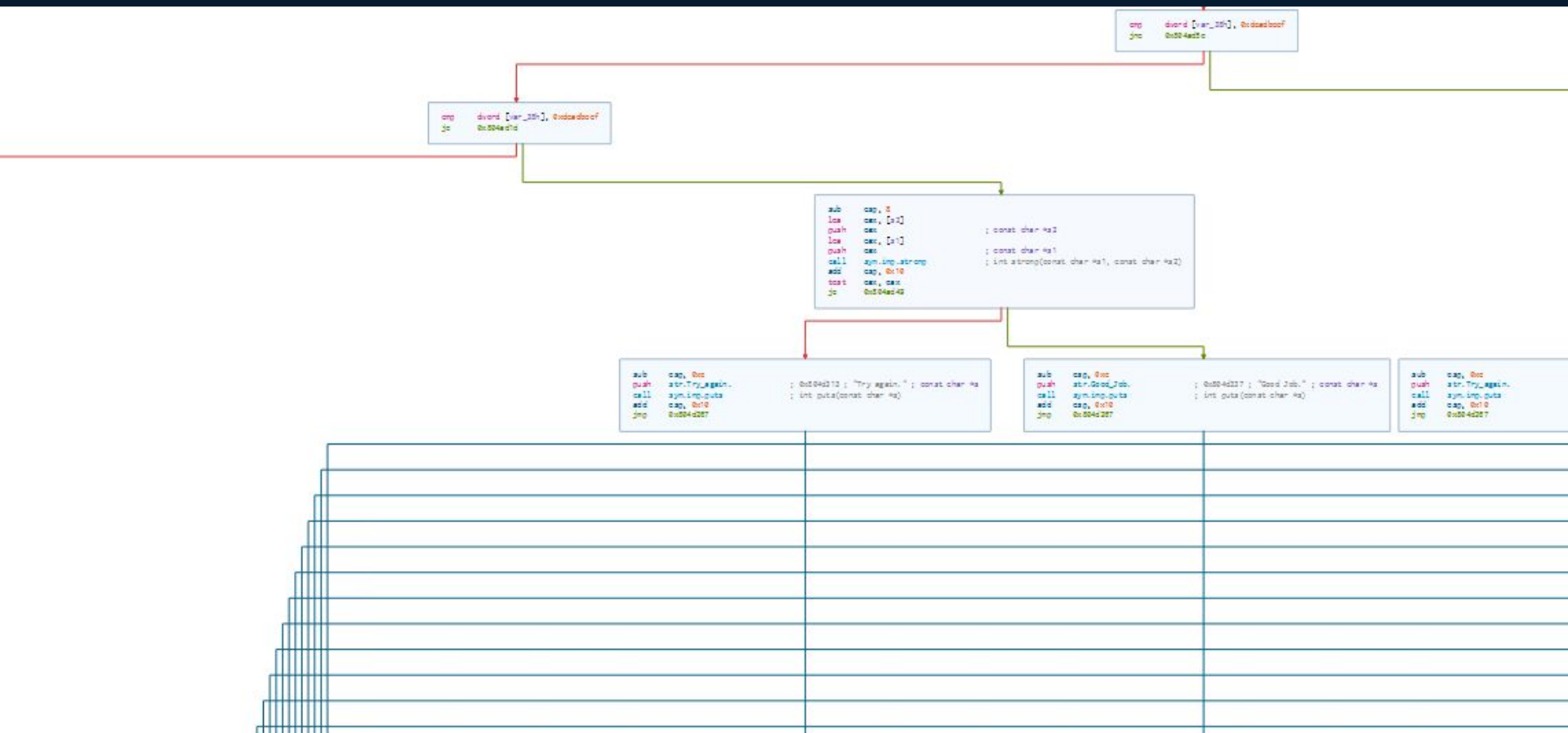


Insomni'Hack 2022
Jannis Kirschner
<https://t.me/learningnets>

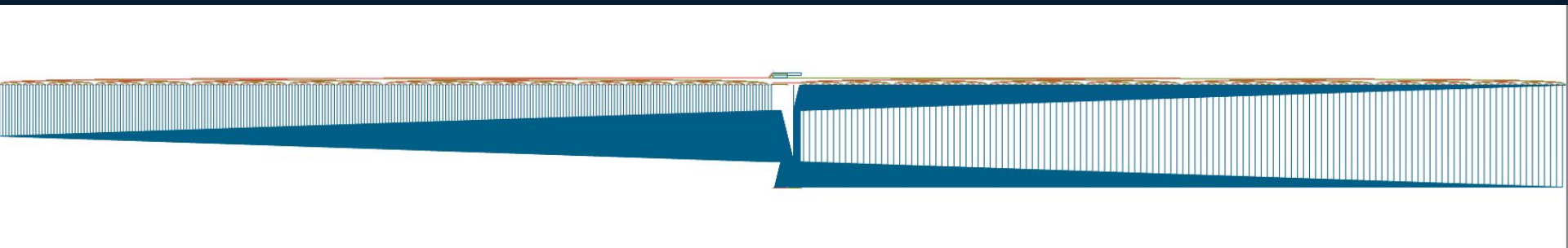


25.03.2022











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

Symbolic Execution Demystified

or

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

```
; 4
sub esp, 0x34
mov eax, ecx
; [0x4:4]=-1
; 8
mov eax, dword [eax + 4]
mov dword [var_1ch], eax
; [0x14:4]=-1
; 20
mov eax, dword gs:[0x14]
mov dword [format], eax
xor eax, eax
call sym.print_msg;[oa]
sub esp, 0xc
; const char *format
; 0x804873e
; "Enter the password: "
push str.Enter_the_password;
; int printf(const char *format)
; call sym.imp.printf;[ob]
add esp, 0x10
sub esp, 8
lea eax, [1]
push eax
; const char *format
...
```

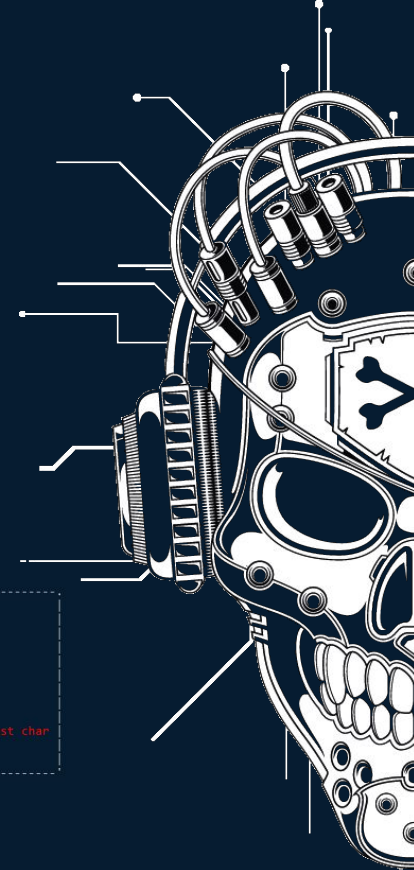
```
0x804864a [og]
; CODE XREF from main (0x8048
...
```

```
0x804861d [of]
; CODE XREF from main (0x8048
lea edx, [1]
mov eax, dword [var_1ch]
add eax, edx
movzx eax, byte [eax]
movsx eax, al
sub esp, 8
push dword [var_1ch]
push eax
call sym.complex_function;[oe]
add esp, 0x10
mov ecx, eax
lea edx, [1]
mov eax, dword [var_1ch]
...
```

```
0x8048650 [oi]
sub esp, 8
; const char *s2
; 0x8048757
; "QMSYJIQP"
push str.QMSYJIQP
lea eax, [1]
; const char *s1
push eax
; int strcmp(const char *s1, const char
call sym.imp.strcmp;[oh]
...
```

```
0x8048668 [ok]
sub esp, 0xc
; const char *s
; 0x8048733
; "Try again."
push str.Try_again.
; int puts(const char
...
```

```
0x804867a [ol]
; CODE XREF from main (0x8048
sub esp, 0xc
; const char *s
; 0x8048760
; "Good Job."
push str.Good_Job.
...
```



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

<https://t.me/learningnets>

What you'll learn today

SYMBOLIC EXECUTION

ANGR

PROBLEM STATE

Section 1: Problem State

Investigating the **Why?**

Section 2: Symbolic Execution

Section 3: Angr

<https://t.me/learningnets>



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 !

Creator : Nofix

Pts: 189

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

Static Analysis

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

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

Soooo...how can we solve such challenge?

$3x^2$

$$L_0 \left(\sqrt{\frac{4K(1+K)}{P_{SF}}} P_K \right)$$

$$21 = (A + \eta)^2 + \kappa^2 \text{ and}$$

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] & 0x3fffffffU) == 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

```
cleaned.txt
~/Insomnihack/00_z3
Save

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] & 0x3fffffffU) == 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] & 0x3fffffffU) == 0x72 &&
37    param_1[0x14] >> 6 == '\x01' &&
38    param_1[7] >> 1 == '/' &&
```

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] & 0x3fffffffU) == 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____rt____

<- lower case t

<- lower case z

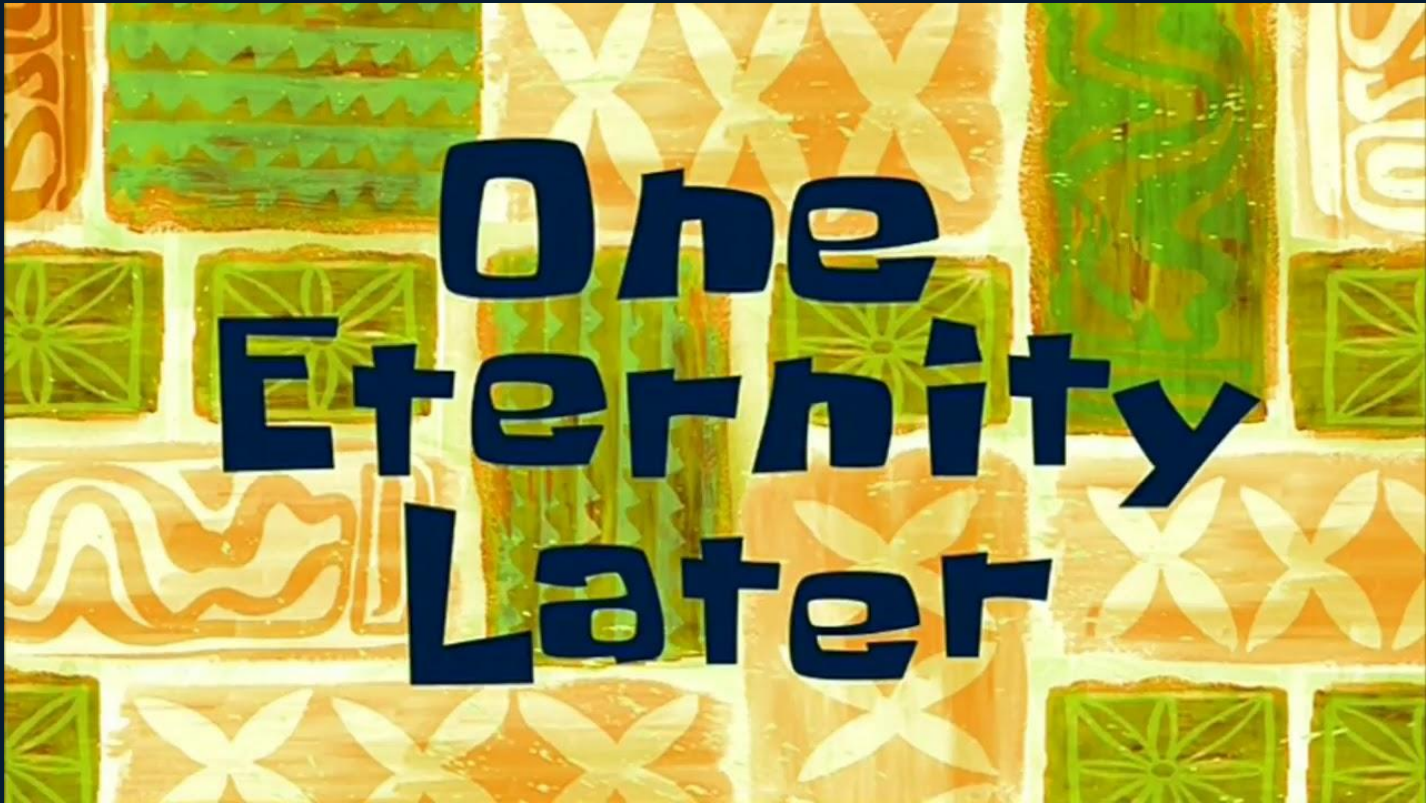
<- 0x13 + 0x5f = 0x72 (lower case r)

<- underscore

<- lower case t

<- lower case s

<- underscore

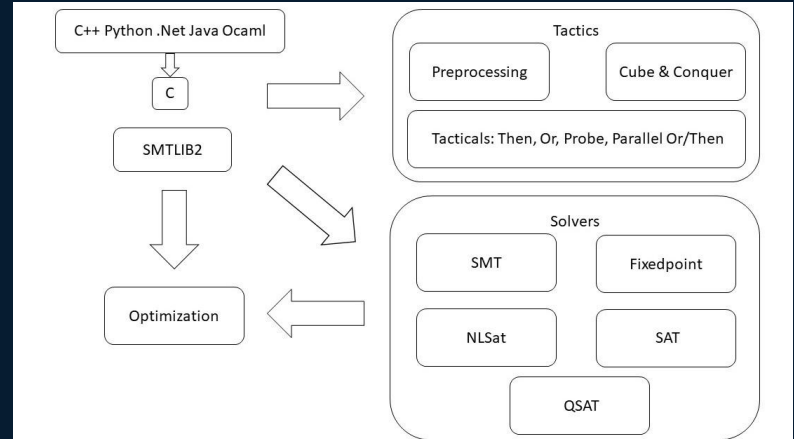


Overview over z3

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

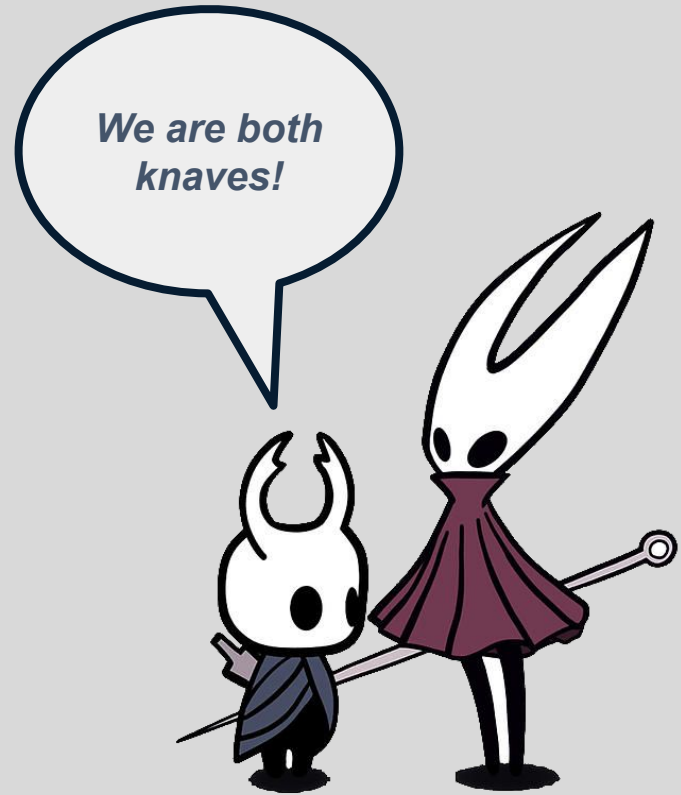


Architecture diagram of z3

A logic puzzle

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

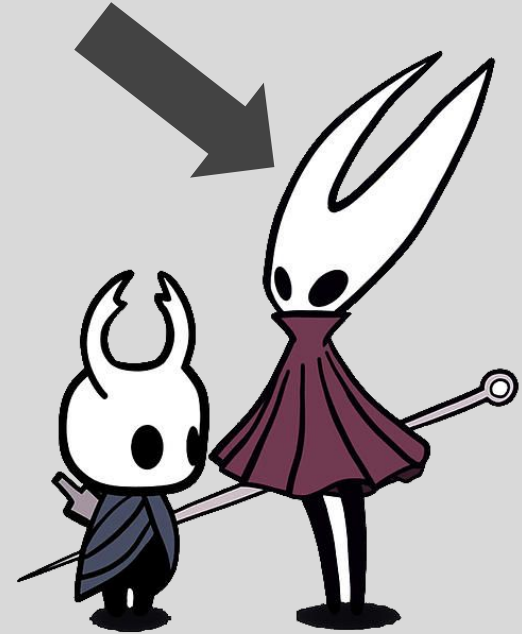
*Two people stand in front of you, **Red** and **Blue**. **Blue** tells you “**we are both knaves**”...who is the knight?*



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:

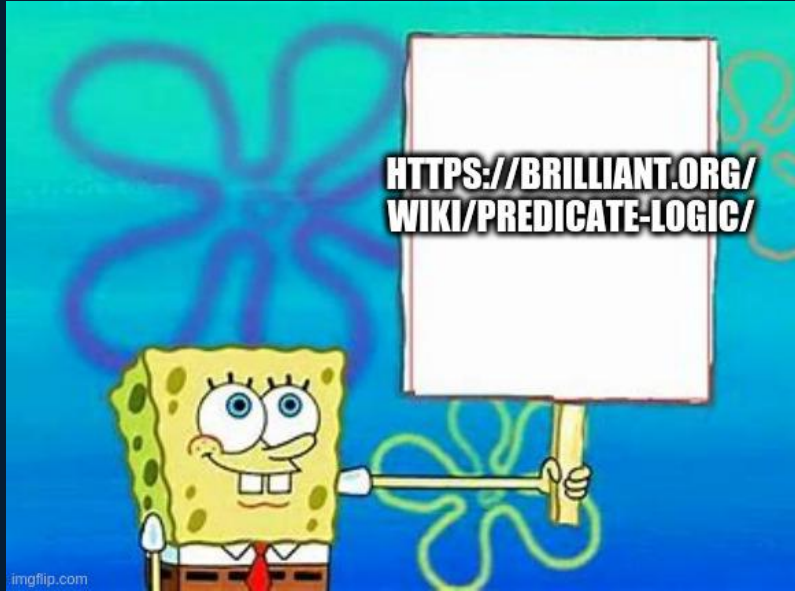
“Given three booleans a,b,c - can the following formula return true: ”
(a and not b) or (not a and c)

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
SAT solvers solve constraints written in propositional logic .	SMT solvers are more powerful and extend them by solving constraints written in predicate (first-order) logic with quantifiers.
Sentences/Statements are propositions (think knights and knaves). Propositional logic studies how they interact irregardless of the contents of the statement -> only logical connections.	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)

If you wanna deep-dive into the maths:



<https://t.me/learningnets>

```
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 --
```

11,20 All

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

```
run.py
Save
Open
Python Tab Width: 8 Ln 105, Col 1 INS

1 # From 03_constraints.py
2
3 # List of BitVec
4 bit = BitVec('bit', 1)
5 s = Solver()
6
7 # Constraints
8 s.add(sat1[0] == sat1[1])
9 s.add(sat1[1] - sat1[2] == -2)
10 s.add(sat1[2] == 10)
11 s.add(sat1[3] == 10)
12 s.add(sat1[4] == 10)
13 s.add(sat1[5] == (sat1[1] & 10) == 1)
14 s.add(sat1[6] == 10)
15 s.add(sat1[7] == 10)
16 s.add(sat1[8] == (sat1[1] & 10) == 1)
17 s.add(sat1[9] == 10)
18 s.add(sat1[10] == (sat1[1] & 10) == 1)
19 s.add(sat1[11] == 10)
20 s.add(sat1[12] == 10)
21 s.add(sat1[13] == 10)
22 s.add(sat1[14] == 10)
23 s.add(sat1[15] == 10)
24 s.add(sat1[16] & 10) == 10)
25 s.add(sat1[17] == 10)
26 s.add(sat1[18] == 10)
27 s.add(sat1[19] == 10)
28 s.add(sat1[20] == 10)
29 s.add(sat1[21] == (sat1[1] & 10) == 1)
30 s.add(sat1[22] == 10)
31 s.add(sat1[23] & 10) == 10)
32 s.add(sat1[24] == 10)
33 s.add(sat1[25] & 10) == 10)
34 s.add(sat1[26] == 10)
35 s.add(sat1[27] == 10)
36 s.add(sat1[28] == 10)
37 s.add(sat1[29] == 1)
38 s.add(sat1[30] == 1)
39 s.add(sat1[31] == 10)
40 s.add(sat1[32] == 1)
41 s.add(sat1[33] & 10) == 10)
42 s.add(sat1[34] == 10)
43 s.add(sat1[35] & sat1[36] == 10)
44 s.add(sat1[37] == 10)
45 s.add(sat1[38] == sat1[39] == 10)
46 s.add(sat1[40] == 10)
47 s.add(sat1[41] == 10)
48 s.add(sat1[42] == 10)
49 s.add(sat1[43] == 10)
50 s.add(sat1[44] == 10)
51 s.add(sat1[45] == 10)
52 s.add(sat1[46] == 10)
53 s.add(sat1[47] & sat1[48] == 10)
54 s.add(sat1[49] == 10)
55 s.add(sat1[50] & sat1[51] == 1)
56 s.add(sat1[52] & (sat1[1] & 10) == 10)
57 s.add(sat1[53] & sat1[54] == 10)
58 s.add(sat1[55] & 10) == 10)
59 s.add(sat1[56] == 10)
60 s.add(sat1[57] == 10)
61 s.add(sat1[58] & 10) == 10)
62 s.add(sat1[59] == 10)
63 s.add(sat1[60] == 10)
64 s.add(sat1[61] == 10)
65 s.add(sat1[62] == 10)
66 s.add(sat1[63] == 10)
67 s.add(sat1[64] == 10)
68 s.add(sat1[65] == 10)
69 s.add(sat1[66] & sat1[67] == 10)
70 s.add(sat1[68] & sat1[69] == 10)
71 s.add(sat1[70] & sat1[71] == 10)
72 s.add(sat1[72] & sat1[73] == 10)
73 s.add(sat1[74] == 10)
74 s.add(sat1[75] == 10)
75 s.add(sat1[76] == 10)
76 s.add(sat1[77] & 10) == 1)
77 s.add(sat1[78] & sat1[79] == 10)
78 s.add(sat1[80] == (sat1[1] & 10) == 1)
79 s.add(sat1[81] == 10)
80 s.add(sat1[82] == 10)
81 s.add(sat1[83] == sat1[84])
82 s.add(sat1[85] == 10)
83 s.add(sat1[86] == 10)
84 s.add(sat1[87] == (sat1[1] & 10) == 1)
85 s.add(sat1[88] == 10)
86 s.add(sat1[89] == 10)
87 s.add(sat1[90] == (sat1[1] & 10) == 1)
88 s.add(sat1[91] == 10)
89 s.add(sat1[92] == 10)
90 s.add(sat1[93] & 10) == 10)
91 s.add(sat1[94] == 10)
92 s.add(sat1[95] == 10)
93 s.add(sat1[96] & sat1[97] == 10)
94 s.add(sat1[98] & sat1[99] == 10)
95 s.add(sat1[100] == 10)
96 s.add(sat1[101] == 10)
97 s.add(sat1[102] == 10)
98 s.add(sat1[103] == 10)
99 s.add(sat1[104] == 10)
100
101
102 is_satisfiable = s.isModel()
103 model = s.getModel()
104 solution_array = [int(i) for i in model[bit]]
105 flag = 'unsatisfiable' if not is_satisfiable else 'satisfiable'
106 print(flag)
107
108 if is_satisfiable:
109     print(model)
110
```



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



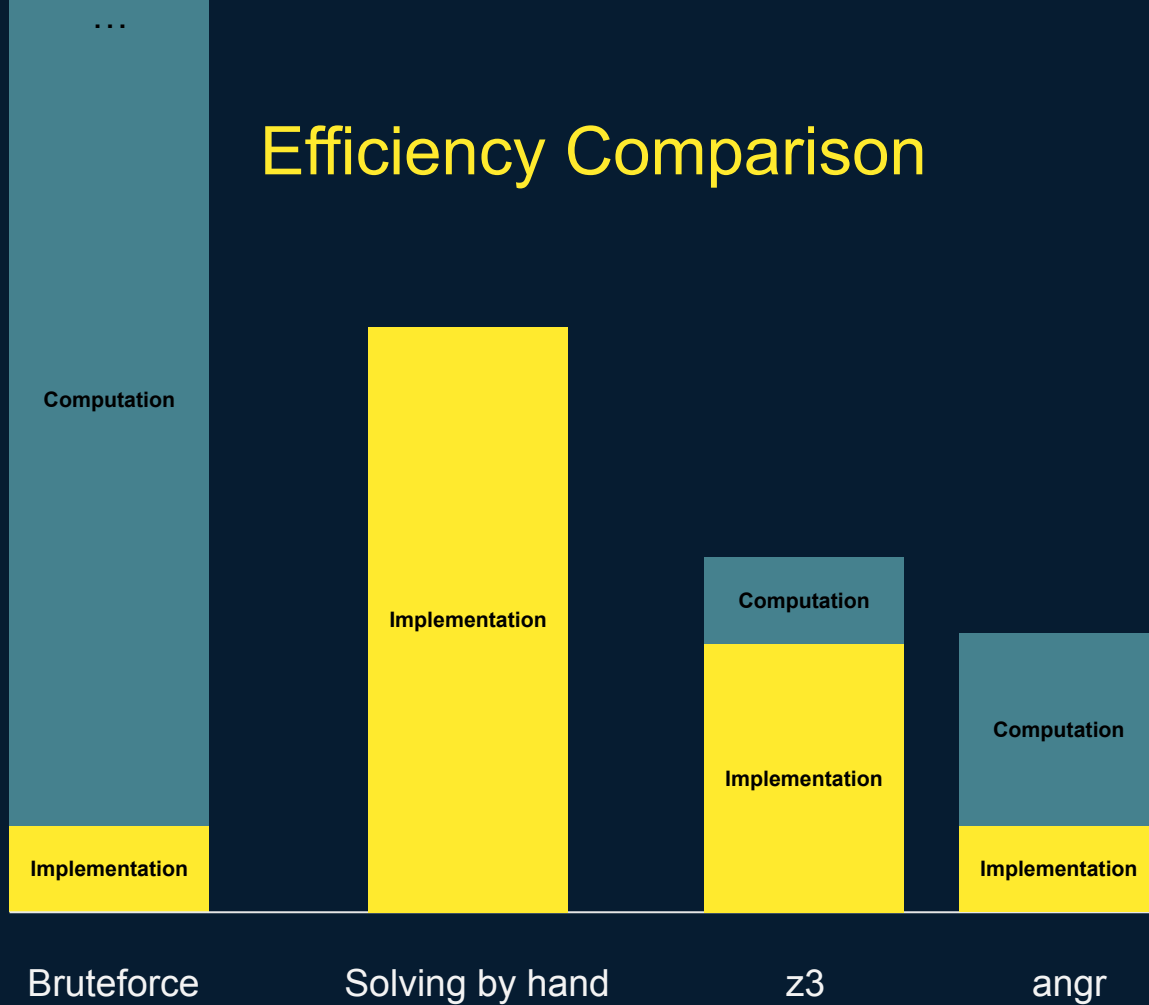
Any guesses to how many lines of code we can reduce it?

We can do the same in about

4

lines of code

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**

Brute force



Solving by hand



SMT Solving



Symbolic Execution

Introducing

Symbolic Execution

Section 2: Symbolic Execution

Illuminating the **What?**

Section 1: Problem Space

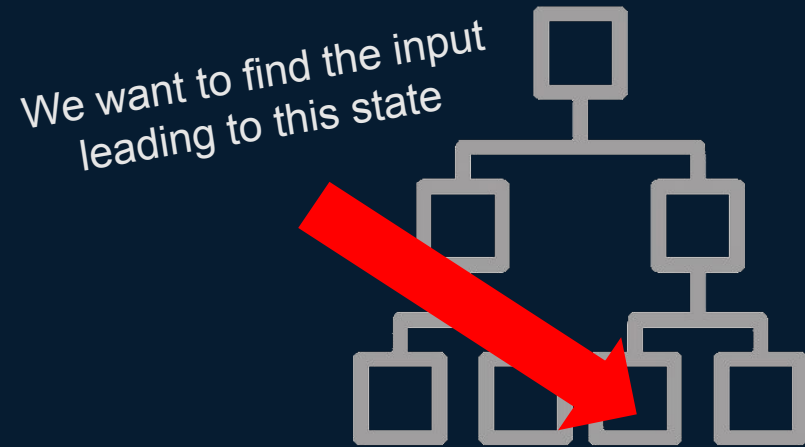
<https://t.me/learningnets>

Section 3: Angr



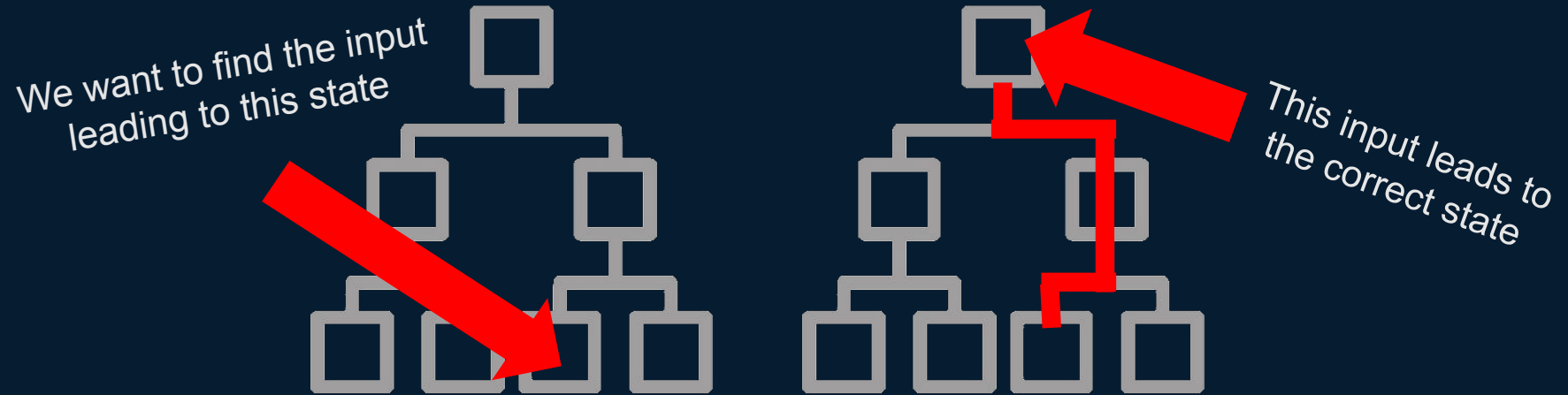
Symbolic Execution is a

“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()      ←  $\lambda$   
    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()      ←  $\lambda$   
    if (size < 5):  
        printf("Works")  ←  $\lambda < 5$   
    else:  
        crash()          ←  $\lambda > 5$   
}
```

“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()      ←  $\lambda$   
    if (size < 5):  
        printf("Works")   ←  $\lambda < 5$   
    else:  
        crash()           ←  $\lambda > 5$   
}
```

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

+

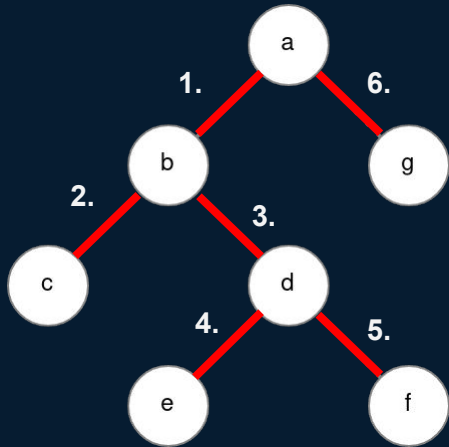
Symbolic Execution

= Concolic Testing

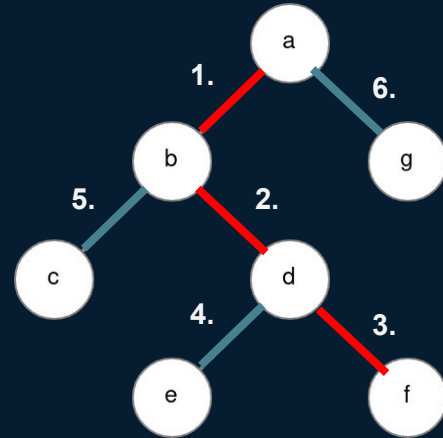
Seed-driven concolic execution is able to favor paths and reach deep into the execution tree

<https://t.me/learningnets>

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")        ←  $\lambda < 5$   
    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")        ←  $\lambda < 5$ 
    else:                        ←  $\neg(\lambda < 5)$ 
        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:                        ← ¬(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")        ←  $\lambda < 5$ 
    else:                        ←  $\neg(\lambda < 5)$ 
        crash()                 ←  $\lambda \geq 5$ 
}
```

Program Validation Tradeoffs

Paths Discovered



“Slushie”
Concolic Testing



“Liquid”
Symbolic Execution



Manual Static Analysis



“Solid”
Concrete Testing

Cost (Computational Resources/Time/Manual Labor)

Different symbolic execution tools

Full System: s2e

User: Angr
Triton
Manticore

Code: KLEE

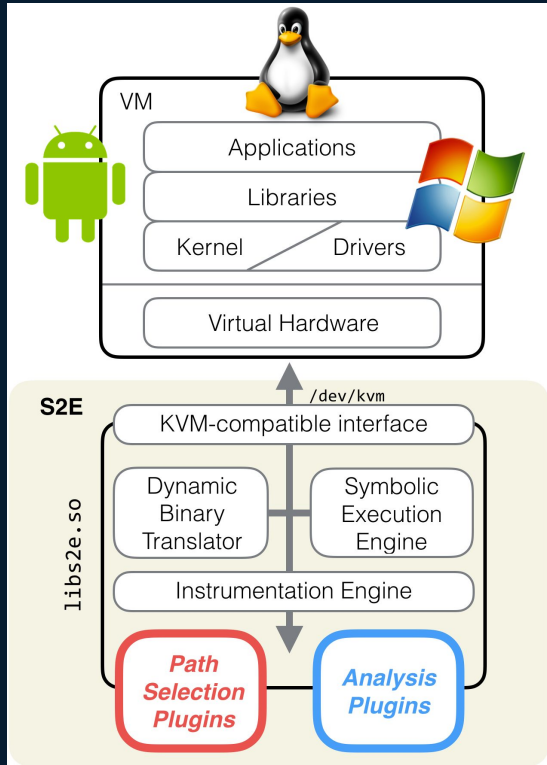
Different symbolic execution tools

Full System: s2e

User: Angr
Triton
Manticore

Code: KLEE

S²E: The Selective Symbolic Execution Platform



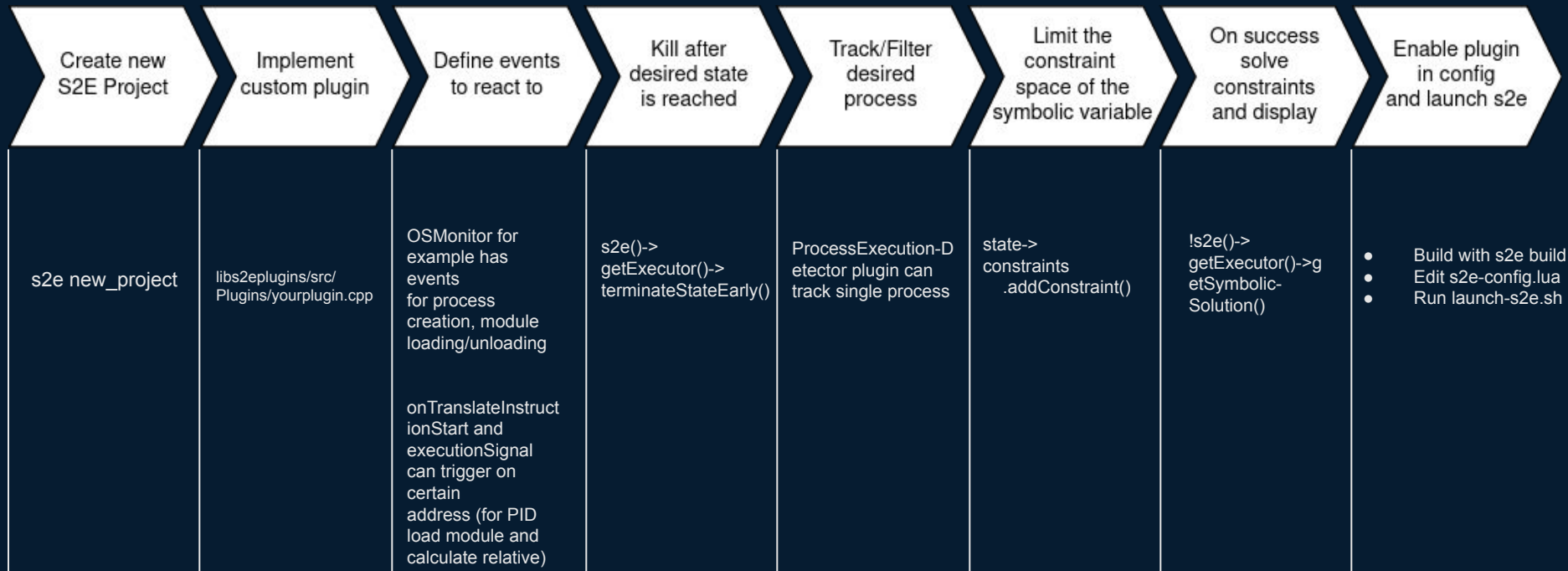
Modular library that enriches virtual machines with symbolic execution & program analysis capabilities.

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 Architecture Diagram

S2E Walkthrough



unbreakable-ctf-s2e/Gooc

https://github.com/adrianherrera/unbreakable-ctf-s2e/blob/master/ 67%

Why GitHub? Team Enterprise Explore Marketplace Pricing

Search Sign in Sign up

adrianherrera / unbreakable-ctf-s2e Public

Notifications Fork 0 Star 2

< Code Issues 1 Pull requests Actions Projects Wiki Security Insights

master unbreakable-ctf-s2e / libs2eplugins / src / s2e / Plugins / GoogleCTFUnbreakable.h Go to file

adrianherrera unbreakable: added missing header Latest commit reass on Apr 25, 2019 History

1 contributor

44 lines (33 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 *m_procDetector;
32
33     void onSymbolicVariableCreation(S2EExecutionState *state, const std::string &name,
34                                   const std::vector<klee::ref<klee::Expr>> &expr, const klee::MemoryObject *mo,
35                                   const klee::Array *array);
36     void onTranslateInstruction(ExecutionSignal *signal, S2EExecutionState *state, TranslationBlock *tb, uint64_t pc);
37     void onSuccess(S2EExecutionState *state, uint64_t pc);
38     void onFailure(S2EExecutionState *state, uint64_t pc);
39 };
40
41 } // namespace plugins
42 } // namespace s2e
43
44 #endif
```

<https://t.me/learningnets>

unbreakable-ctf-s2e/Gooc

https://github.com/adrianherrera/unbreakable-ctf-s2e/blob/master/ 30%

Why GitHub? Team Enterprise Explore Marketplace Pricing

Search Sign in Sign up

adrianherrera unbreakable-ctf-s2e

adrianherrera unbreakable-ctf-s2e

```
1 #ifndef UNBREAKABLE_CTF_S2E_PLUGIN_H
2 #define UNBREAKABLE_CTF_S2E_PLUGIN_H
3
4 #include <s2e/CorePlugin.h>
5 #include <s2e/Plugin.h>
6 #include <s2e/S2E.h>
7
8 namespace s2e {
9 namespace plugins {
10
11 class ProcessExecutionDetector;
12
13 class GoogleCTFUnbreakable : public Plugin {
14     S2E_PLUGIN
15
16 public:
17     GoogleCTFUnbreakable(S2E *s2e) : Plugin(s2e) {
18     }
19
20     void initialize();
21
22 private:
23     ProcessExecutionDetector *m_procDetector;
24
25     void onSymbolicVariableCreation(S2EExecutionState *state, const std::string &name,
26                                   const std::vector<klee::ref<klee::Expr>> &expr, const klee::MemoryObject *mo,
27                                   const klee::Array *array);
28     void onTranslateInstruction(ExecutionSignal *signal, S2EExecutionState *state, TranslationBlock *tb, uint64_t pc);
29     void onSuccess(S2EExecutionState *state, uint64_t pc);
30     void onFailure(S2EExecutionState *state, uint64_t pc);
31 };
32
33 } // namespace plugins
34 } // namespace s2e
35
36 #endif
```

Different symbolic execution tools

Full System: s2e

User: Angr
Triton
Manticore

Code: KLEE

Angr/Triton/Manticore



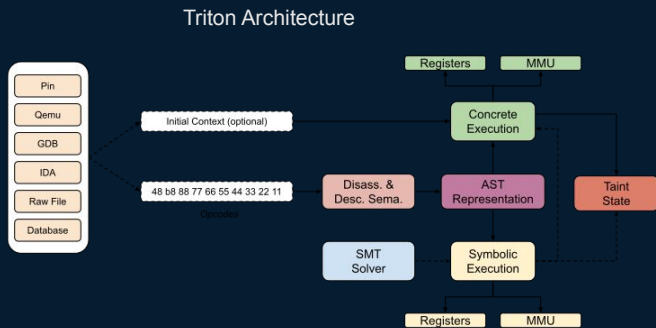
TRILON
Dynamic Binary Analysis



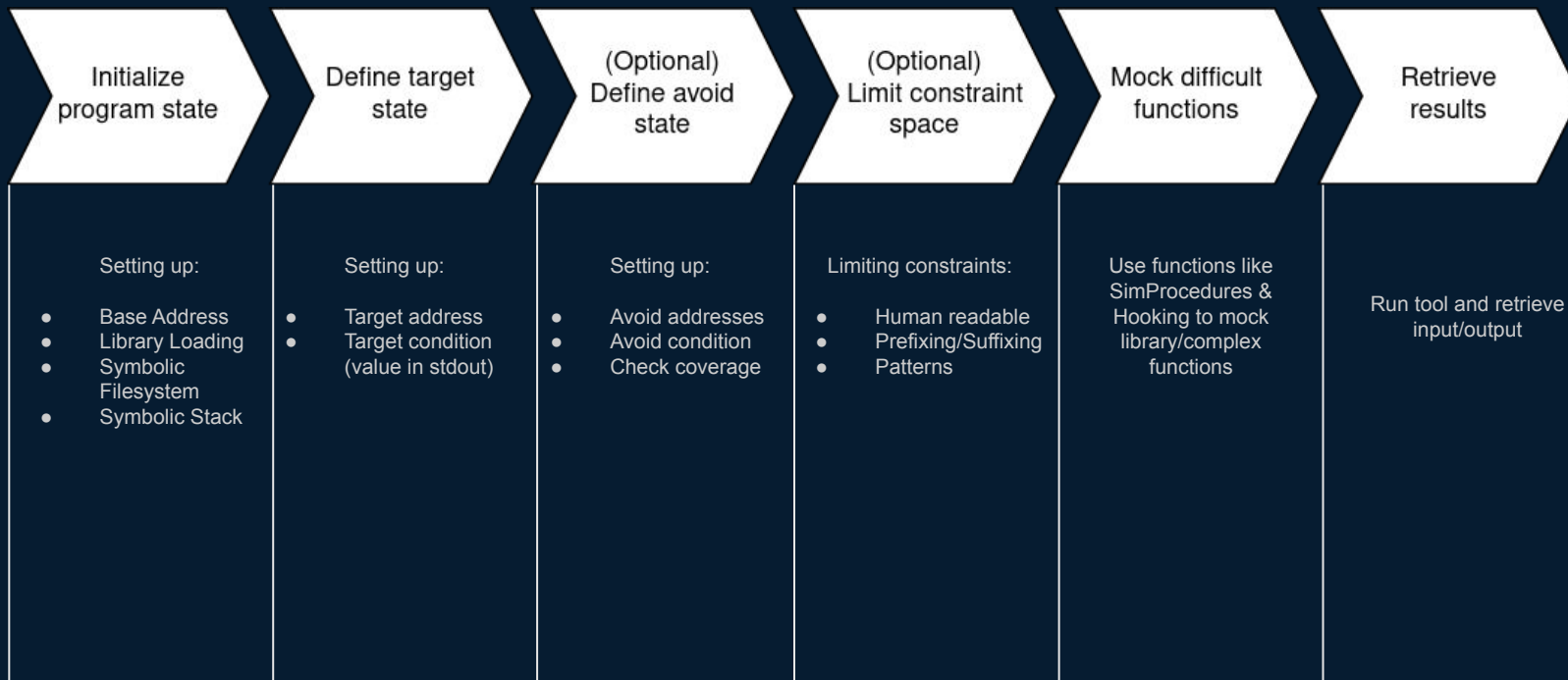
User-level dynamic binary analysis & symbolic execution frameworks (often based on z3).

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 ·

https://github.com/angr/angr-doc/blob/master/examples/google2016_unbreakable_0/solve.py 50%

angr / angr-doc Public

Code Issues 17 Pull requests 1 Actions Projects Wiki Security Insights

master - angr-doc / examples / google2016_unbreakable_0 / solve.py Jump to -

fish Migrate to Python 3 (2011) Latest commit 233550w on Oct 1, 2016 History

4 contributors

Executable File | 55 lines (39 sloc) | 2.58 KB

```
1 #!usr/bin/env python2
2
3 # Author: David Meneuchet@memuchet@protonmail.com
4 # Google 2016 CTF
5 # Challenge: Unbreakable Enterprise Product Activation
6 # Team: Hack-Charlie (http://hack-charlie.team/)
7 # Runtime: ~4.5 seconds (single threaded 65-2000 v3 g 2.9500w on AWS/EC2)
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
15
16     input_size = 0x40 # How large the input is, see 0x0000e.
17
18     argval = claripy.BVS('argval', input_size * 8)
19
20     initial_state = proj.factory.entry_state(args=['./unbreakable-enterprise-product-activation', argval], add_options={angr.options.LAZY_SOLVES})
21     initial_state.libc.buf_symlibic_bytes+input_size + 1 # Thanks to Christopher Salts (Phalix) for pointing this out. By default there's only 60 symlibic 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 argval.chop(8):
25         initial_state.smt_constrain(byte != '\x00') # null
26         initial_state.smt_constrain(byte == '\x20') # '\x20'
27         initial_state.smt_constrain(byte == '\x2f') # '/'
28         # Source: https://www.kernel.org/doc/html/latest/fs/ffs/ffs_3/ffsops/reference/general/integrity-detection-prevention-custom-afk-back-obj-extended-ascii
29         # Thanks to Tom Ravenscroft (@tomravenscroft) for showing me how to restrict to printable characters.
30
31     # We're told that every flag is formatted as "CTF{...}", so we might as well use that information to save processing time.
32     initial_state.smt_constrain(argval.chop(1)[0] == 'C')
33     initial_state.smt_constrain(argval.chop(1)[1] == 'T')
34     initial_state.smt_constrain(argval.chop(2)[2] == '{')
35     initial_state.smt_constrain(argval.chop(2)[3] == 'f')
36     # angr will still find the solution without setting these, but it'll take a few seconds more.
37
38     sm = proj.factory.simulation_manager(initial_state)
39
40     # 0x00000000 = thank you message
41     sm.explore(find=0x00000000, avoid=0x00000000)
42     # 0x00000000 = activation failure
43
44     found = sm.found[0] # In our case, there's only one printable solution.
45
46     solution = found.solver.eval(argval, cast_to_bytes)
47     solution = solution.replace(solution.find(0x2f)+0) # Trim off the null bytes at the end of the flag (if any).
48     return solution
49
50 def test():
51     assert main() == b'CTF{ffw3quik28nandfou4jupq50ver6thefLazylf00d}'
52
53 if __name__ == '__main__':
54     print(Clipart(main()))
```

© 2022 GHSA, Inc. Terms Privacy Security Status Docs Contact GHSA Pricing API Tinting Blog About

https://t.me/learningnets

Different symbolic execution tools

Full System: s2e

User: Angr
Triton
Manticore

Code: KLEE

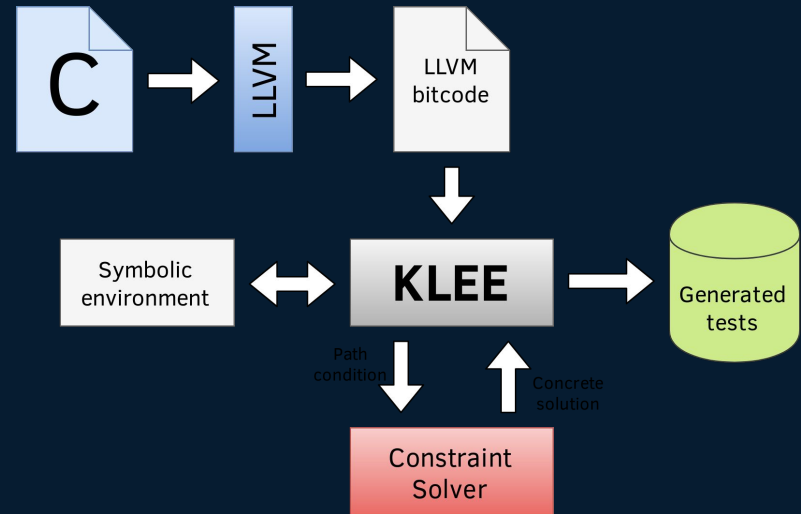
KLEE



LLVM-based symbolic execution engine
for code-level analysis

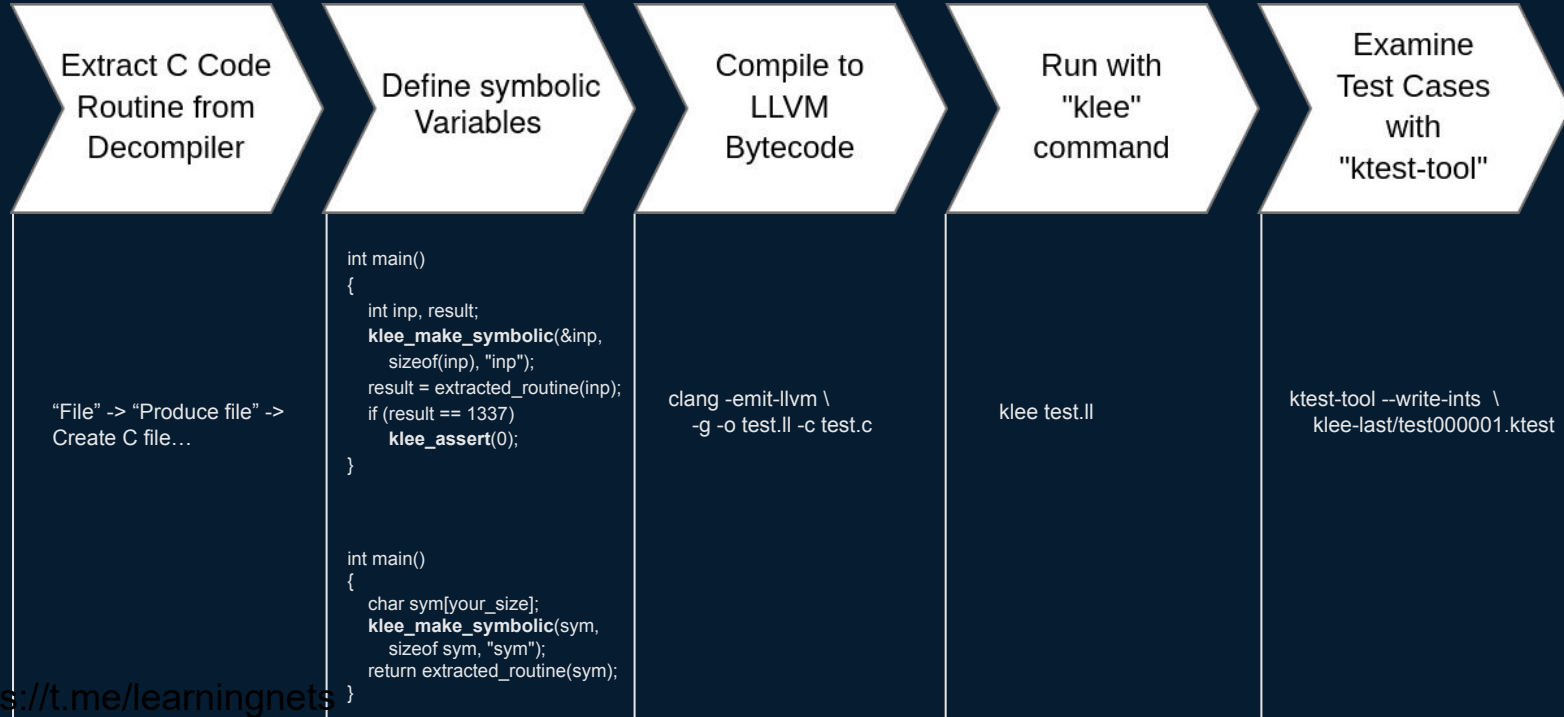
Requires target function to be re-/coded
in C and instrumented

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/master/main.c 67%

GitLab Search GitLab Sign in / Register

Matryoshka-Stage-2

Project information

Repository

Files

Commits

Branches

Tags

Contributors

Graph

Compare

Locked Files

Issues

Merge requests

CI/CD

Deployments

Monitor

Analytics

David Manouchehri · Matryoshka-Stage-2 · Repository

master Matryoshka-Stage-2 / main.c Find file Blame History Permalink

Make Klee friendly.
David Manouchehri authored 5 years ago 08b993e3b

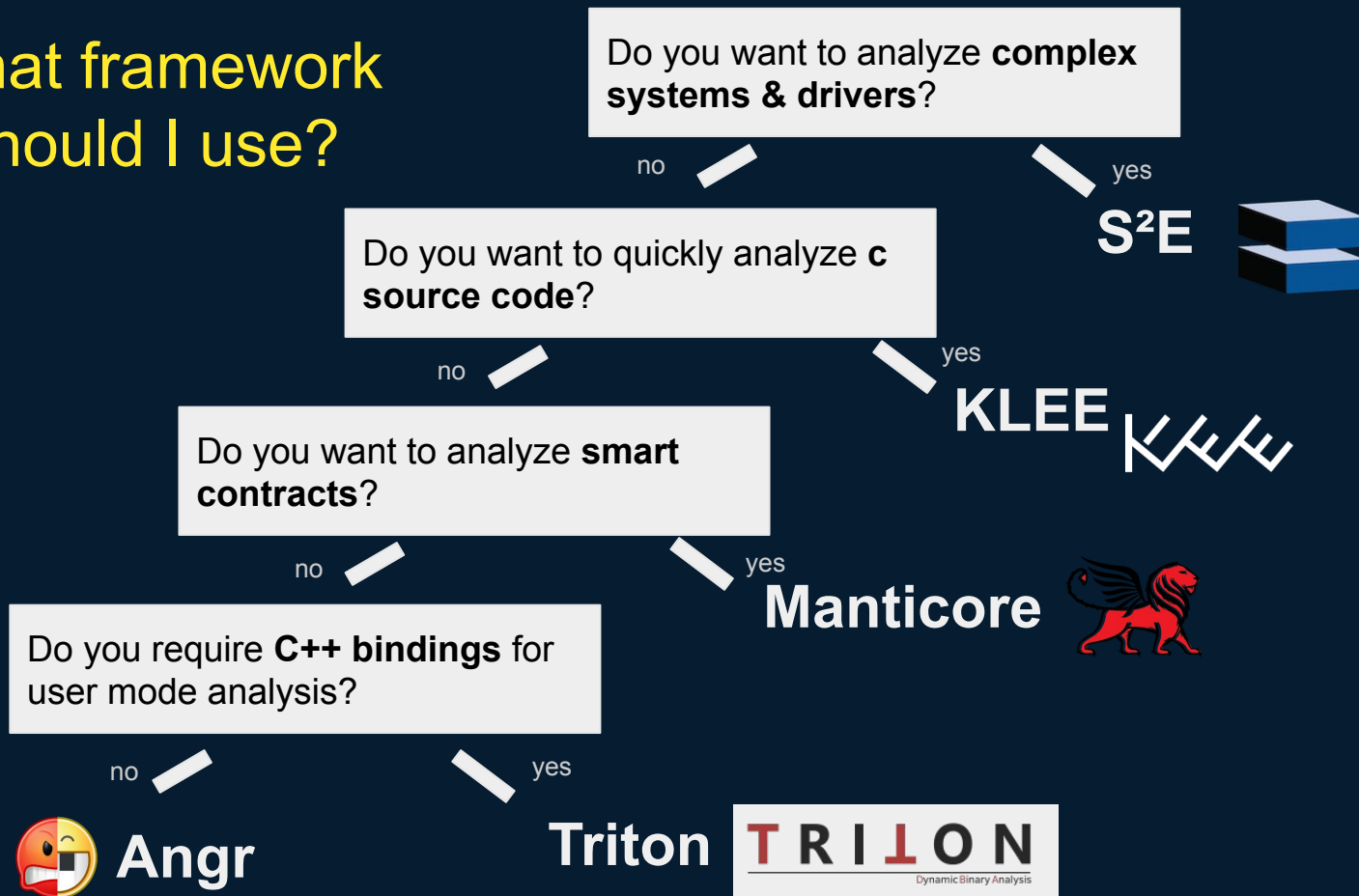
main.c 1.03 KB

```
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; // rdx@10
10    signed int v5; // [sp+1Ch] [bp-14h]@4
11
12    if ( a1 == 2 )
13    {
14        if ( 42 * (strlen(a2[1]) + 1) != 584 )
15            goto LABEL_31;
16        v5 = 1;
17        if ( *a2[1] != 88 )
18            v5 = 0;
19        if ( 2 * a2[1][3] != 208 )
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] )
31            v5 = 0;
32        if ( a2[1][3] != a2[1][9] )
33            v5 = 0;
34        if ( a2[1][4] != 185 )
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            LABEL_31:
46            printf("Try again...\n");
47        }
48        else
49        {
50            printf("Usage: %s <pass>\n", *a2);
51        }
52    }
```

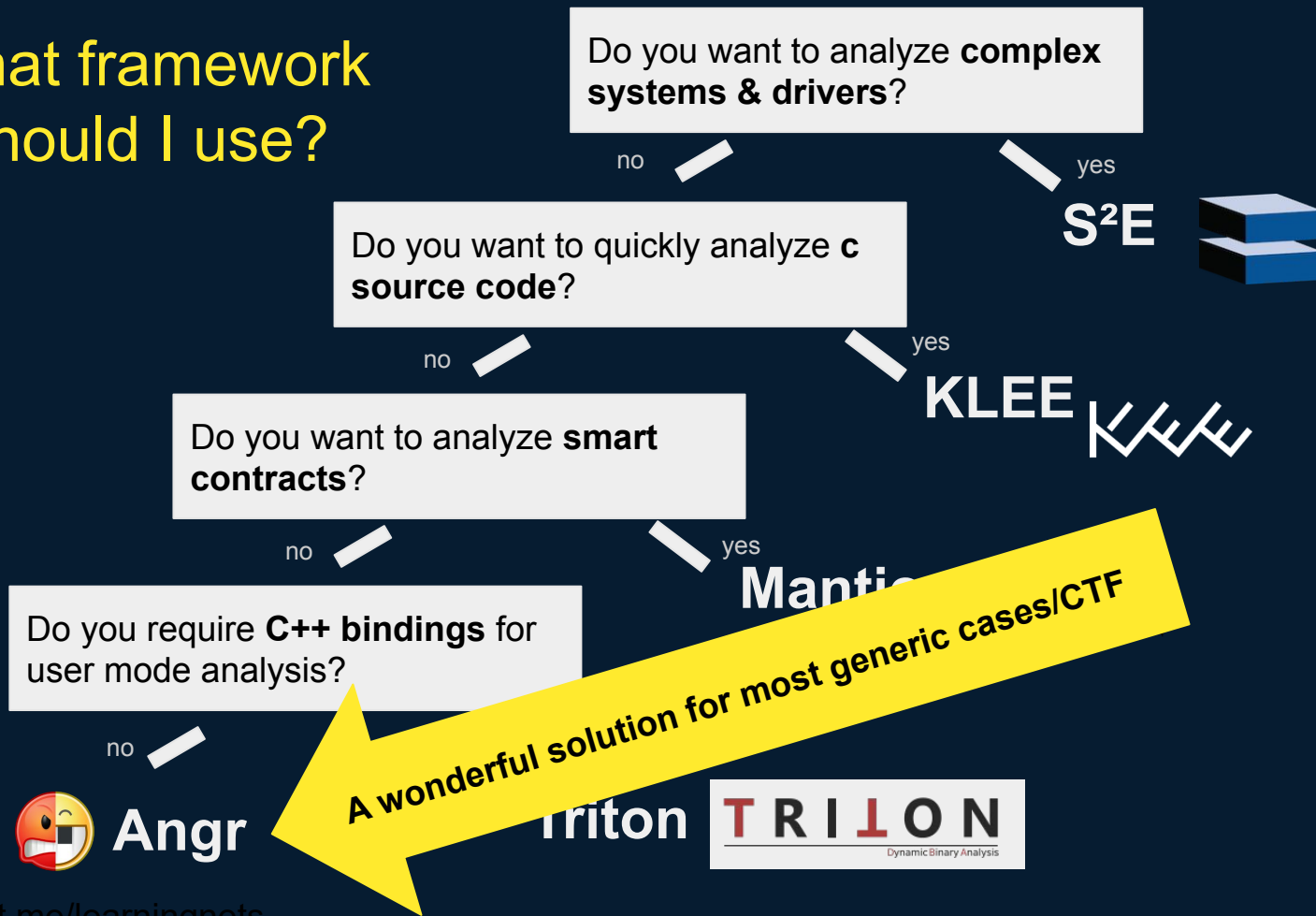
https://t.me/learningnets

« Collapse sidebar

What framework should I use?

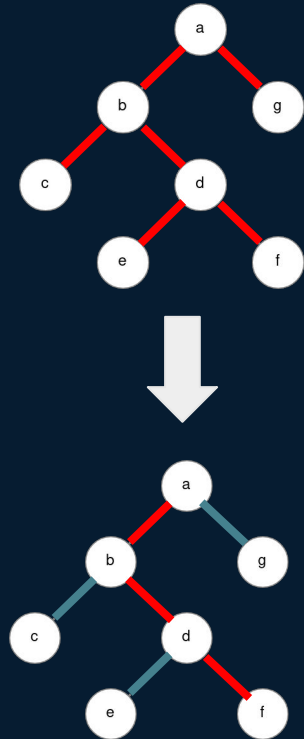


What framework should I use?



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



<https://t.me/learningnets>

Section 3: Angr

Discovering the **How?**

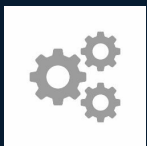
Section 1: Problem Space

Section 2: Symbolic Execution

<https://t.me/learningnets>



angr



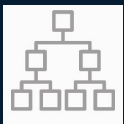
Extensive Binary Analysis Framework



Convenient Python3 Interface

Valgrind

Leverages VEX IR
(x86, ARM, MIPS, PowerPC...)



Symbolic + Concolic Execution



Developed by UCSB

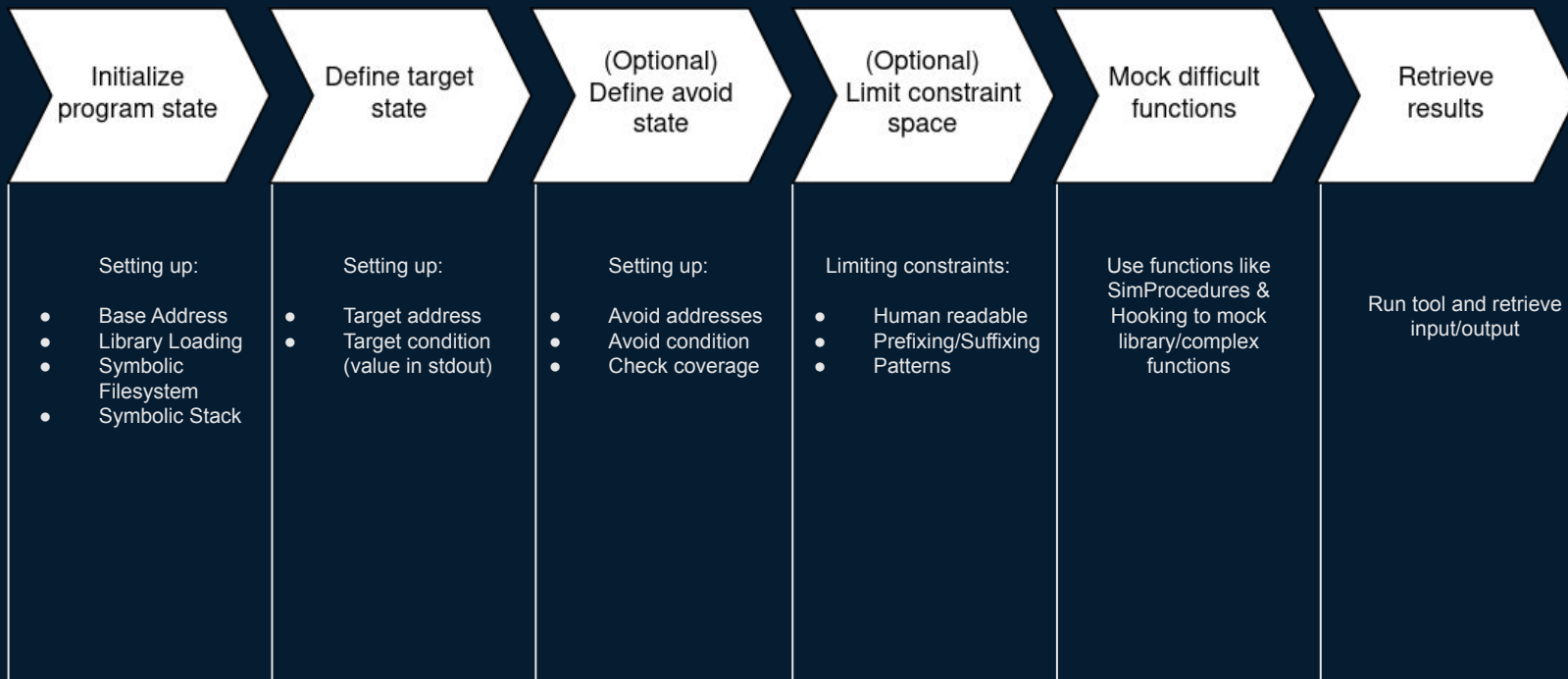
Won 3rd in DARPA
Cyber Grand Challenge

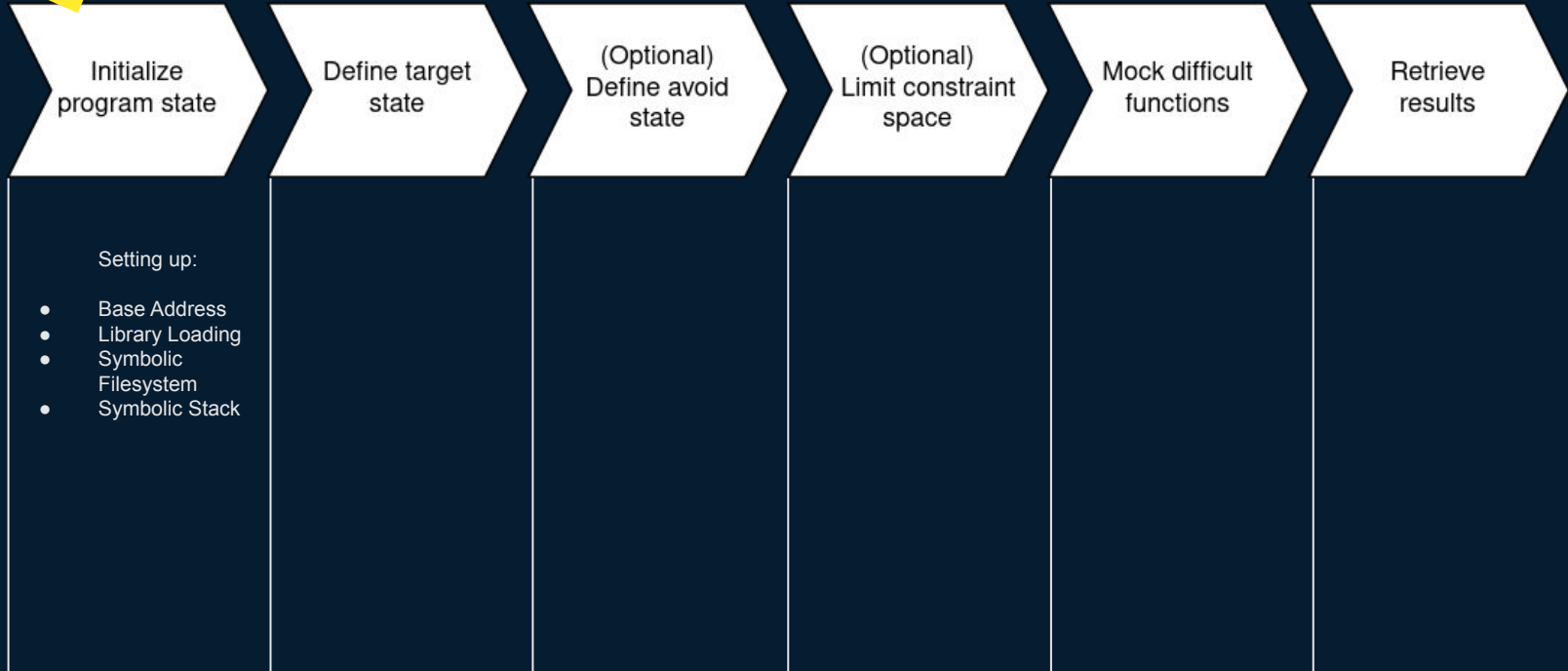
Used for reversing,
rop-chain building,
fuzzing and more



Angr Workflow

<https://t.me/learningnets>





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

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

```
initial_state.regs.ebp = initial_state.regs.esp
```

Initialize stack
frame

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

Define password
bitvectors

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

Align stack pointer

Symbolic Stack

Set start address
after input was
provided

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

```
initial_state.regs.ebp = initial_state.regs.esp
```

Initialize stack
frame

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

Define password
bitvectors

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

Align stack pointer

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

Push password
bitvectors to stack

Symbolic Stack

Set start address
after input was
provided

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

Initialize stack
frame

```
initial_state.regs.ebp = initial_state.regs.esp
```

Define password
bitvectors

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

Align stack pointer

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

Push password
bitvectors to stack

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

Solve bitvectors

```
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)
```

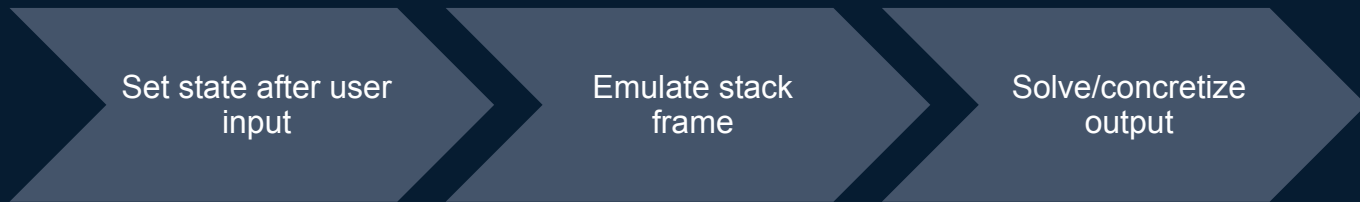
```
padding_length_in_bytes = 0x08  
initial_state.regs.esp -= padding_length_in_bytes
```

```
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))
```



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;
}
```

Symbolic Filesystem

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

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)
```

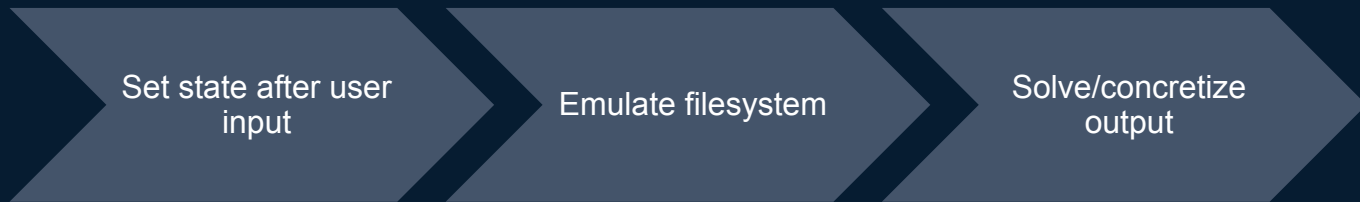
```
password = claripy.BVS('password', sym_file_size * 8)  
symbolic_file_backing_memory.store(0, password)
```

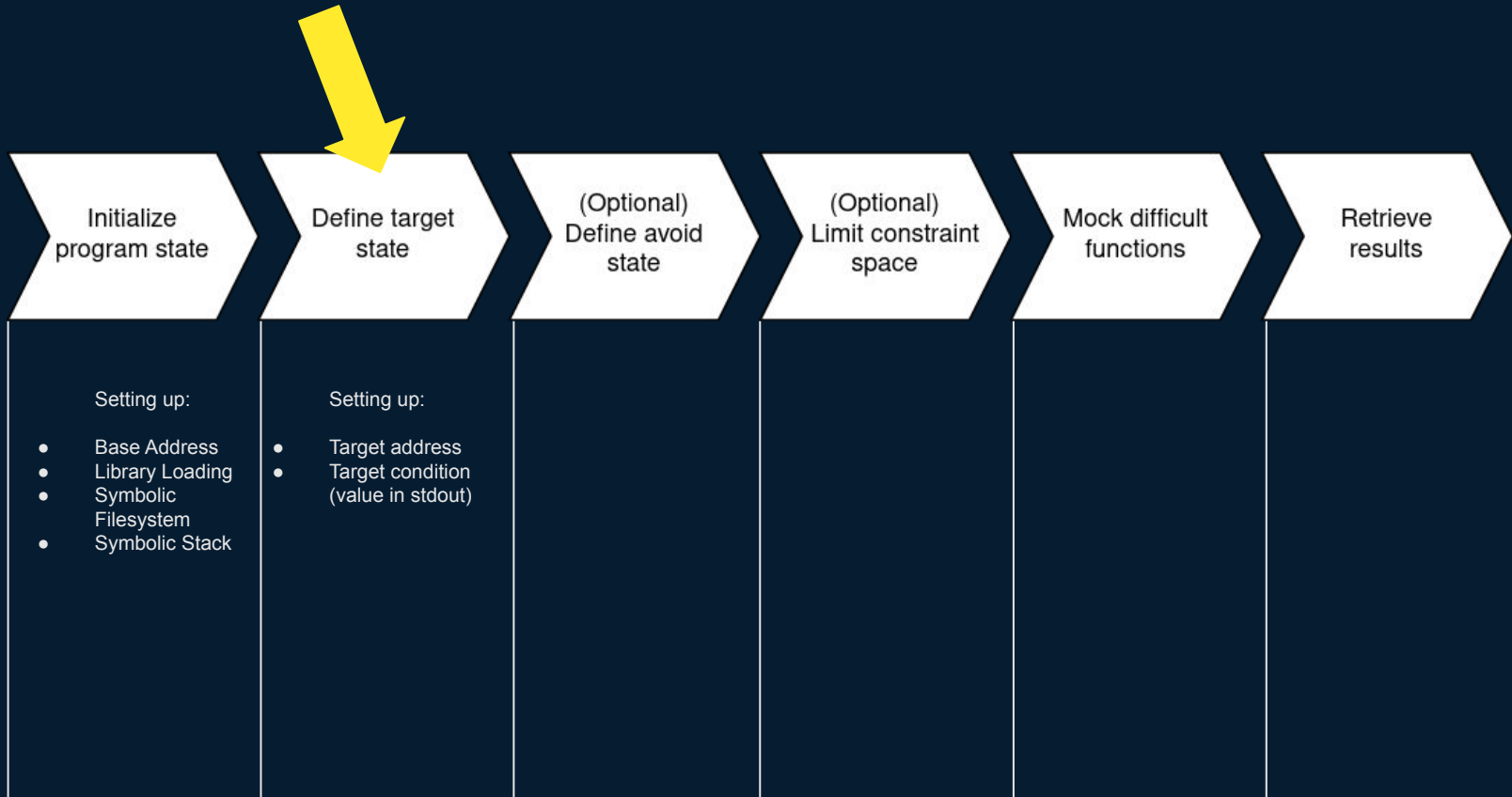
```
password_file = angr.storage.SimFile(filename, 'r',  
    content=symbolic_file_backing_memory,  
    size=symbolic_file_size_bytes  
)
```

```
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)
```





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

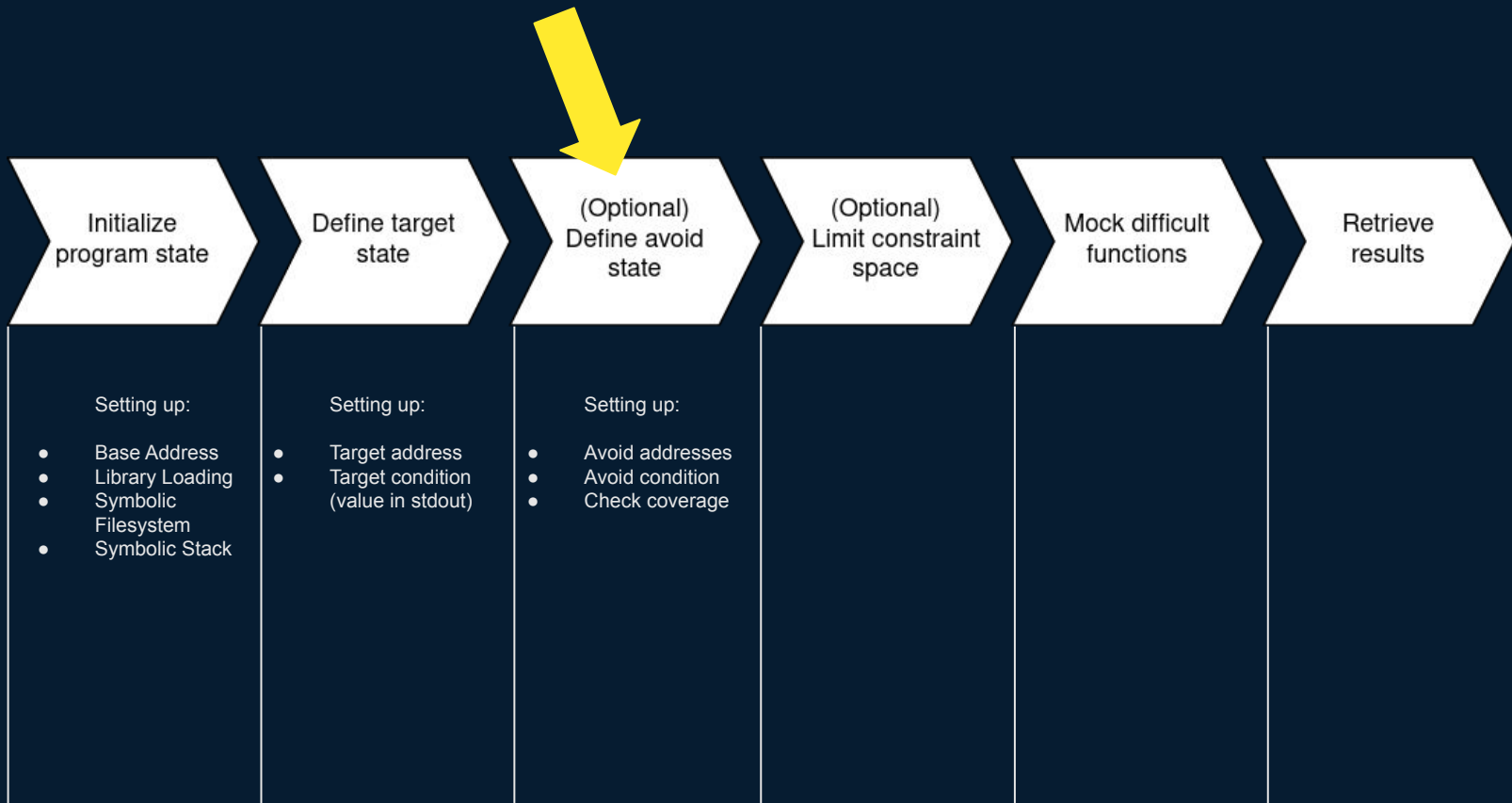
proj = angr.Project('./z3_robot',
                   load_options={"auto_load_libs" : False},
                   main_opts={"base_addr":0}
                   )

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

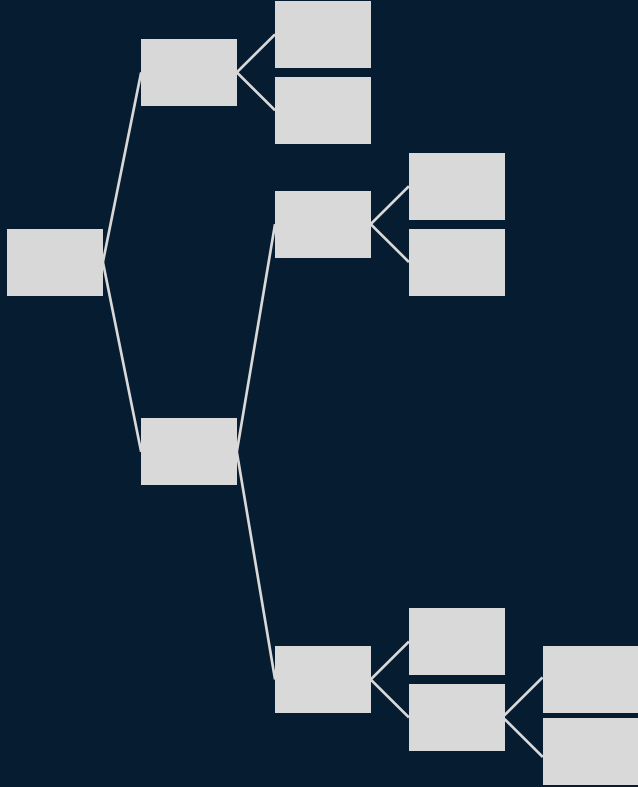
```
import angr, claripy

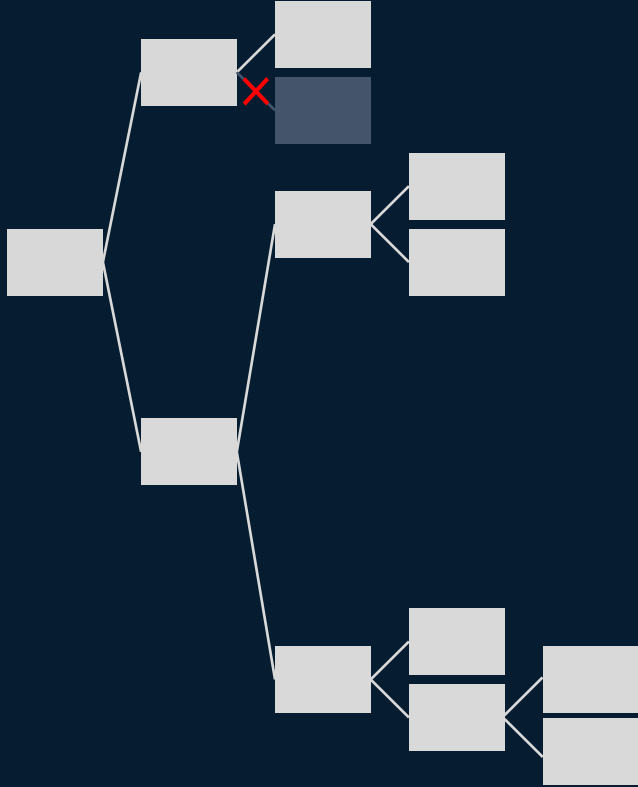
proj = angr.Project('./z3_robot',
                   load_options={"auto_load_libs" : False},
                   main_opts={"base_addr":0})

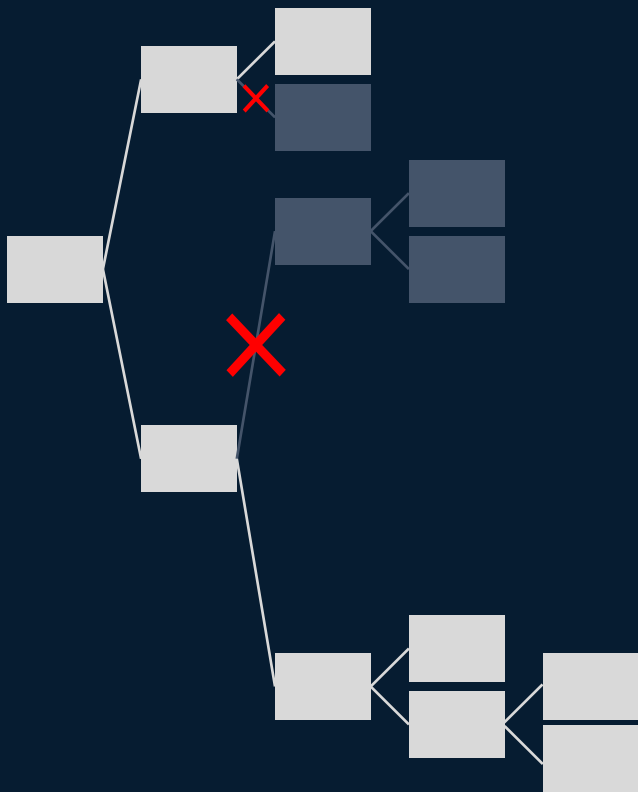
simgr = proj.factory.simgr()

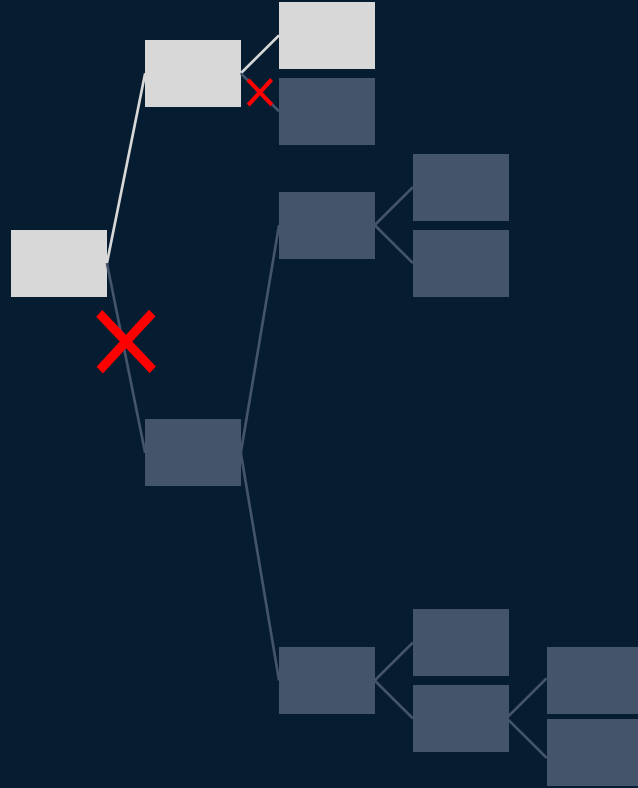
simgr.explore(find=0x00001407, avoid=[0x0000142d])

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









Code Coverage Collection Process

Retrieve Coverage

Export basic block coverage from angr exploration

Fix Bottlenecks

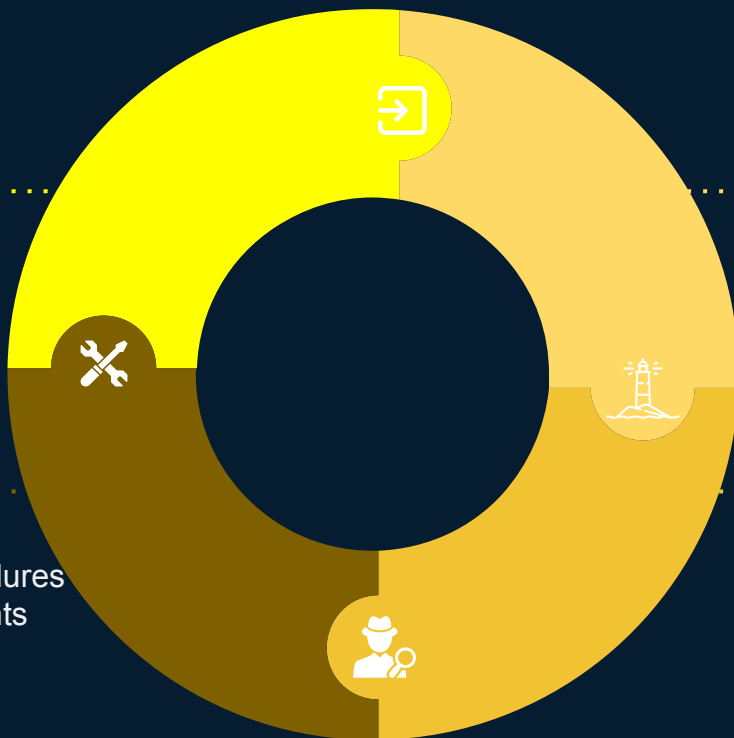
- Avoid bottleneck code
- Use Hooks & SimProcedures
- Manually apply constraints

Load into Lighthouse

Use “File” -> “Load file” -> “Load coverage file/batch...”

Investigate Bottlenecks

Disassemble/Decompile to discover bottlenecks



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)
```

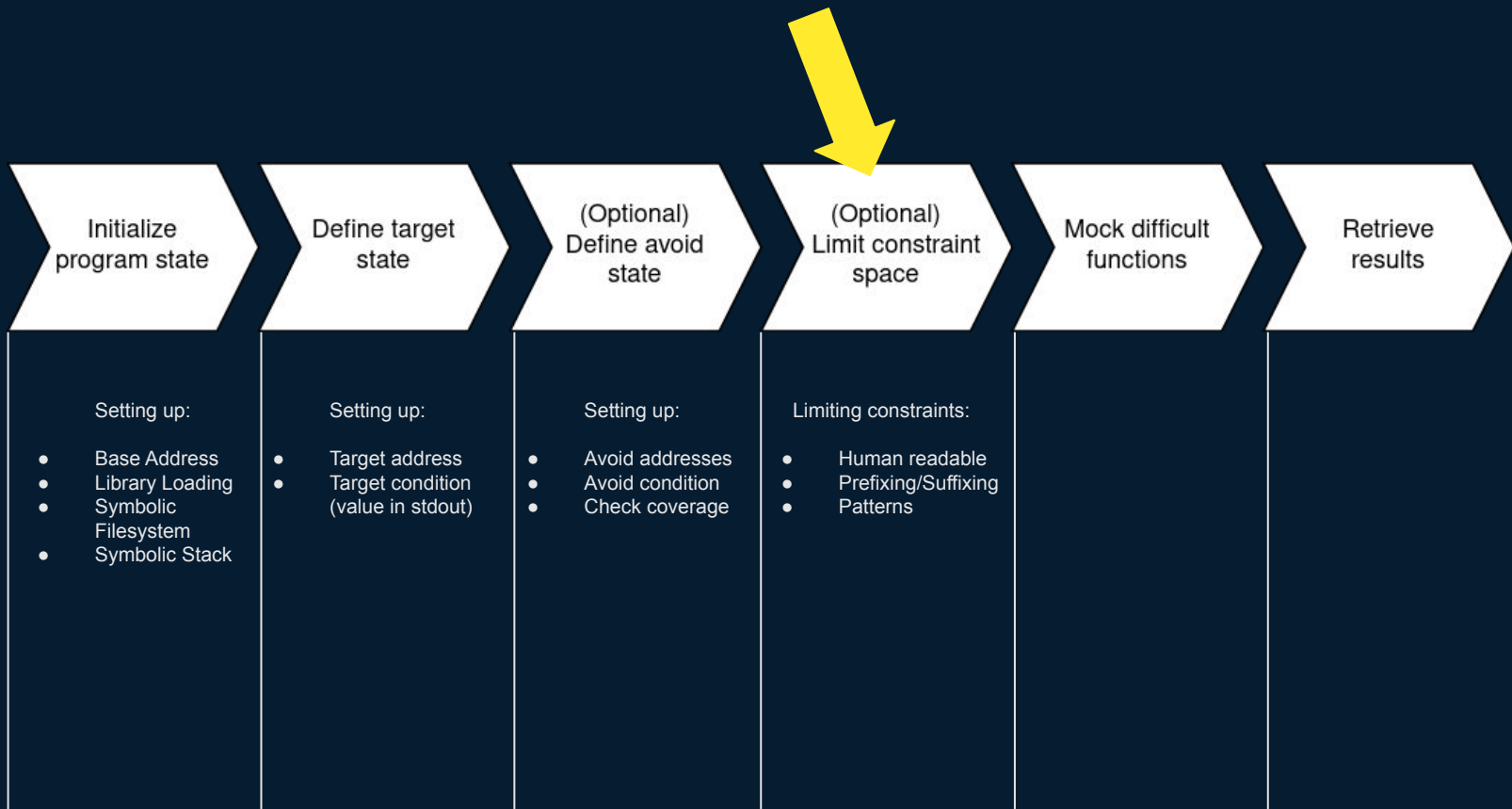
Load into lighthouse to find bottlenecks...

The screenshot displays the IDA Pro interface with the assembly view of the `__fastcall check_flag` function. The assembly code is as follows:

```
000008A8 __fastcall check_flag(char *a1)
{
    return ((unsigned __int8)a1[20] ^ 0x20) == a1[7]
    && a1[21] - a1[3] == -20
    && !a1[2] >> 6)
    && a1[13] == 116
    && 4 * a1[11] == 380
    && a1[7] >> (a1[17] % 8) == 5
    && ((unsigned __int8)a1[6] ^ 0x53) == a1[14]
    && a1[9] == 122
    && a1[5] << (a1[9] % 8) == 392
    && a1[16] - a1[7] == 20
    && a1[7] << (a1[23] % 8) == 190
    && a1[2] - a1[7] == -43
    && a1[21] == 95
    && ((unsigned __int8)a1[2] ^ 0x47) == a1[3]
    && a1 == 99
    && a1[13] == 116
    && (a1[20] & 0x45) == 68
    && (a1[8] & 0x15) == 16
    && a1[2] == 95
    && a1[4] >> 4 == 7
    && a1[13] == 116
    && a1 >> (*a1 % 8) == 12
    && a1[10] == 95
    && (a1[8] & 0xAC) == 40
    && a1[16] == 115
    && (a1[22] & 0x2D) == 24
    && a1[9] == 51
    && a1[5] == 49
    && 4 * a1[19] == 456
    && a1[20] >> 6 == 1
    && a1[7] >> 1 == 47
    && a1[1] == 188
    && a1[3] >> 4 == 7
    && (a1[19] & 0x49) == 64
    && a1[4] == 115
    && ((unsigned __int8)a1[2] + (unsigned __int8)a1[11]) == 20
    && a1 == 99
    && a1[4] + a1[5] == 164
    && a1[15] << 6 == 6880
    && ((unsigned __int8)a1[18] ^ 0x20) == a1[17]
    && ((unsigned __int8)a1[12] ^ 0x2C) == a1[4]
    && a1[19] - a1[21] == 19
    && a1[12] == 95
    && a1[15] >> 1 == 47
    && a1[19] == 114
    && a1[17] + a1[18] == 168
    && a1[22] == 58
    && ((unsigned __int8)a1[23] + (unsigned __int8)a1[21]) == 9
    && a1[6] << (a1[19] % 8) == 396
    && a1[3] + a1[7] == 210
    && a1[22] & 0xED) == 40
    && (a1[12] & 0x6C) == 12
    && ((unsigned __int8)a1[18] ^ 0x60) == a1[15]
```

The Coverage Overview table on the right shows the following data:

Cov %	Func Name	Address	Blocks Hit	Instr. Hit	Func Size	CC
28.57	__init_proc	0x6C0	2 / 3	2 / 7	23	2
0.00	sub_6E0	0x6E0	0 / 1	0 / 2	12	1
100.00	puts	0x6F0	1 / 1	1 / 1	6	1
0.00	__stack_chk_fail	0x700	0 / 1	0 / 1	6	1
100.00	__printf	0x710	1 / 1	1 / 1	6	1
100.00	__strcpy	0x720	1 / 1	1 / 1	6	1
100.00	fgets	0x730	1 / 1	1 / 1	6	1
100.00	fflush	0x740	1 / 1	1 / 1	6	1
0.00	__cxa_finalize	0x750	0 / 1	0 / 1	6	1
8.33	__atexit	0x760	1 / 1	1 / 12	43	1
0.00	deregister_tm_clones	0x790	0 / 4	0 / 13	40	2
21.11	register_tm_clones	0x7D0	2 / 4	2 / 18	37	2
0.00	__do_global_ctors_aux	0x820	0 / 5	0 / 13	51	2
25.00	frame_dummy	0x860	1 / 1	1 / 4	10	1
1.52	check_flag	0x8A8	12 / 95	12 / 789	2765	93
12.07	main	0x1337	1 / 6	7 / 60	274	2
14.71	__libc_csu_init	0x1450	4 / 4	5 / 34	101	3
0.00	__libc_csu_fini	0x14C0	0 / 1	0 / 1	2	1
0.00	__term_proc	0x14C4	0 / 1	0 / 3	9	1
0.00	puts	0x202048	0 / 1	0 / 1	8	1
0.00	__stack_chk_fail	0x202050	0 / 1	0 / 1	8	1
0.00	__printf	0x202058	0 / 1	0 / 1	8	1
0.00	__strcpy	0x202060	0 / 1	0 / 1	8	1
0.00	__libc_start_main	0x202068	0 / 1	0 / 1	8	1
0.00	fgets	0x202070	0 / 1	0 / 1	8	1
0.00	fflush	0x202078	0 / 1	0 / 1	8	1
0.00	__imp_cxa_finalize	0x202080	0 / 1	0 / 1	8	1
0.00	__gmon_start__	0x202090	0 / 1	0 / 1	8	1



```
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'
```

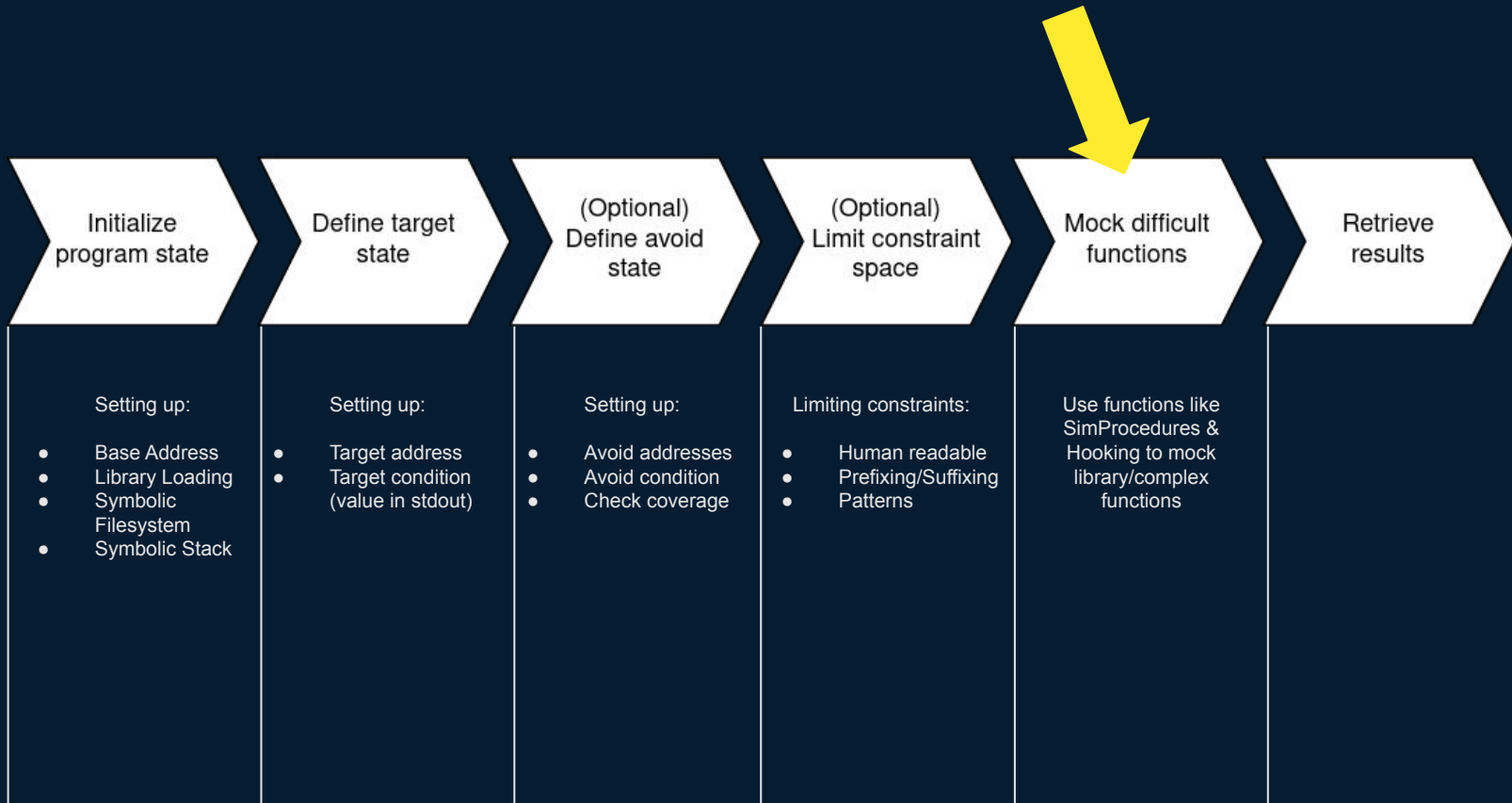
```
# starts with CTF{
```

```
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()
```

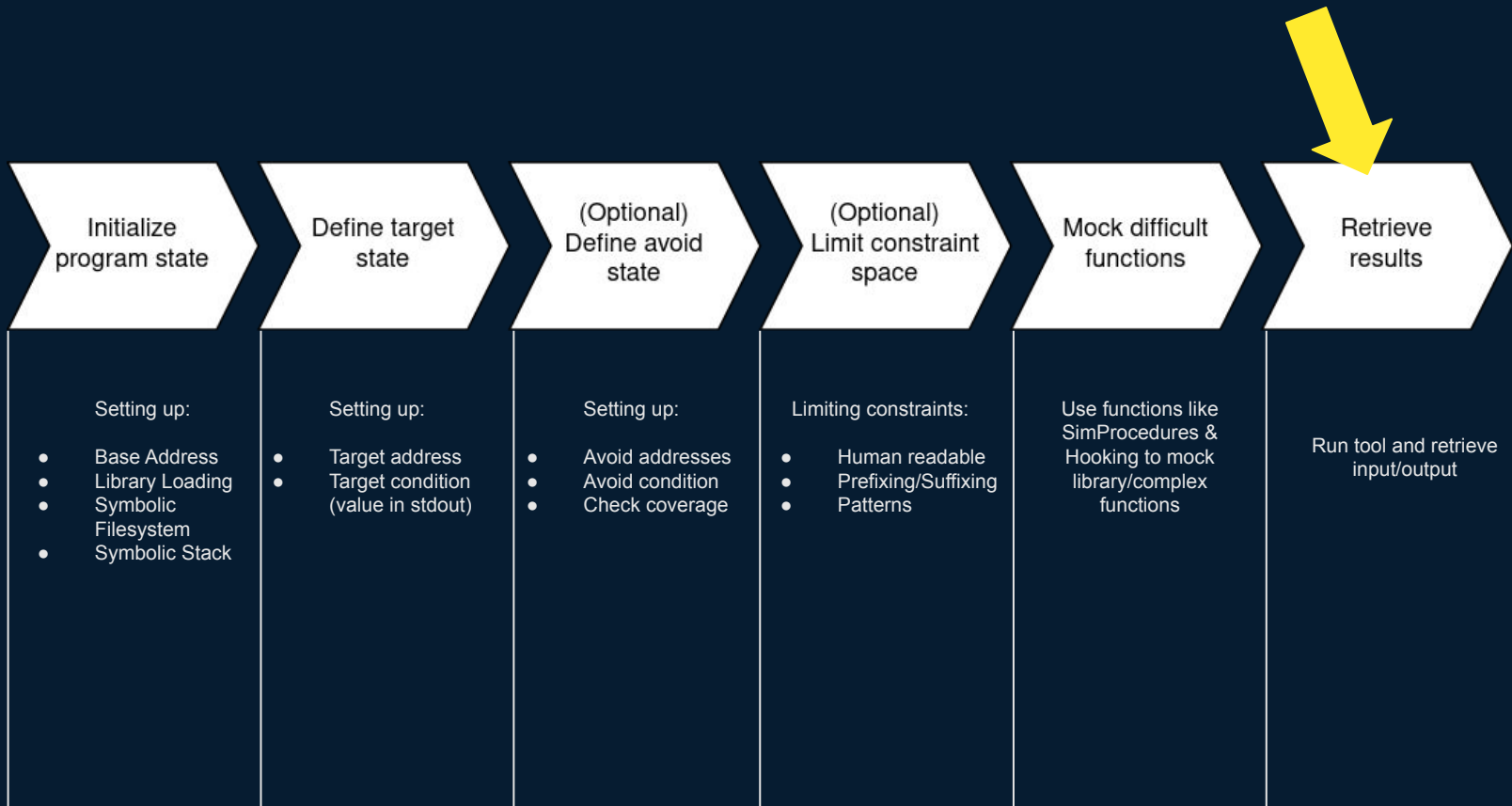
```
# length determines how many bytes get skipped/overwritten
```

```
@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

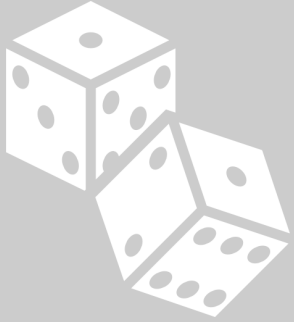
```
simgr.found[0].posix.dumps(0)
```

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

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




Limitations



Non-deterministic
control flow

<https://t.me/learningnets>



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
```

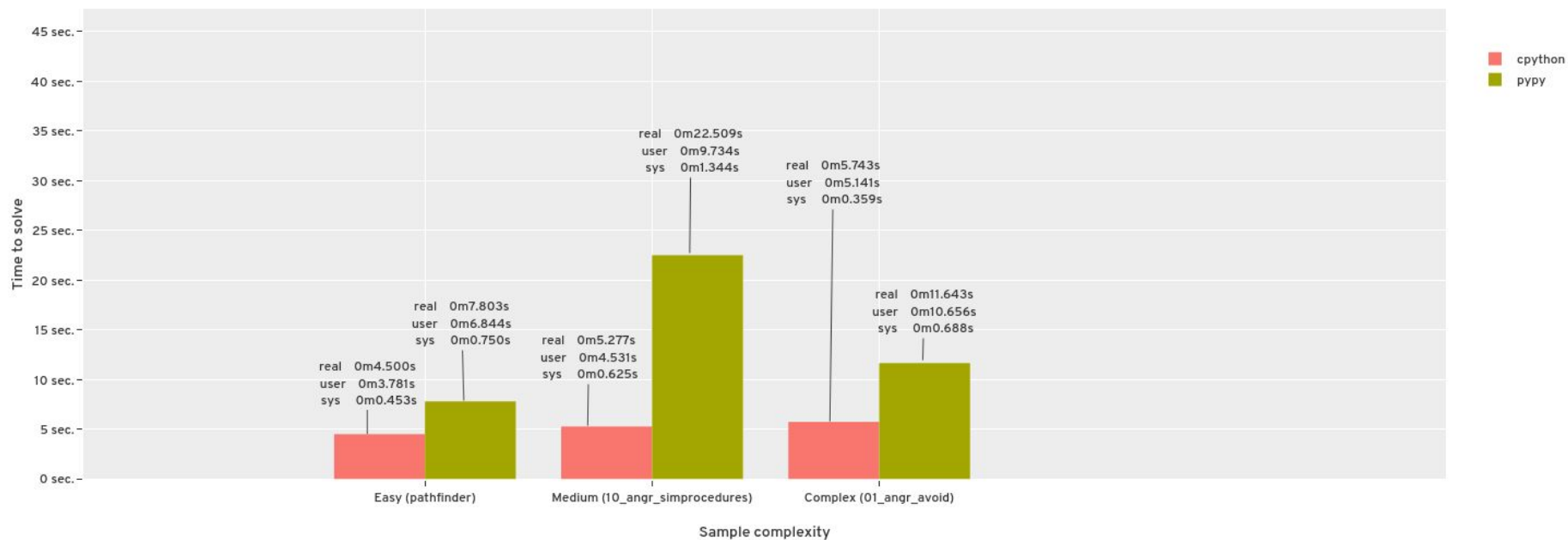
```
proj = angr.Project('./z3_robot',  
                  load_options={"auto_load_libs" : False},  
                  main_opts={"base_addr":0}  
                  )
```

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

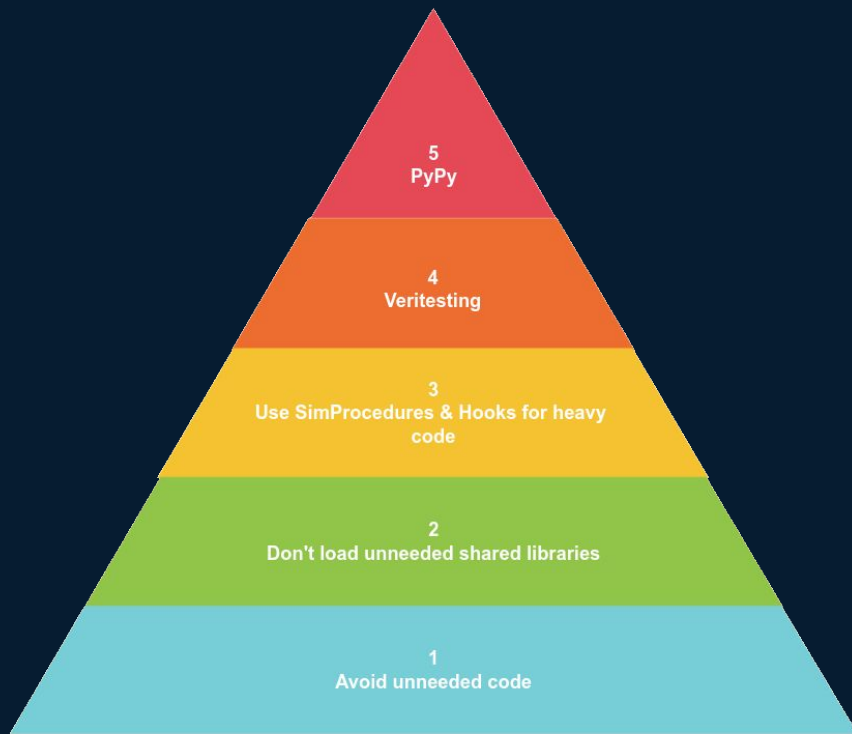
```
simgr.explore(find=0x00001407)
```

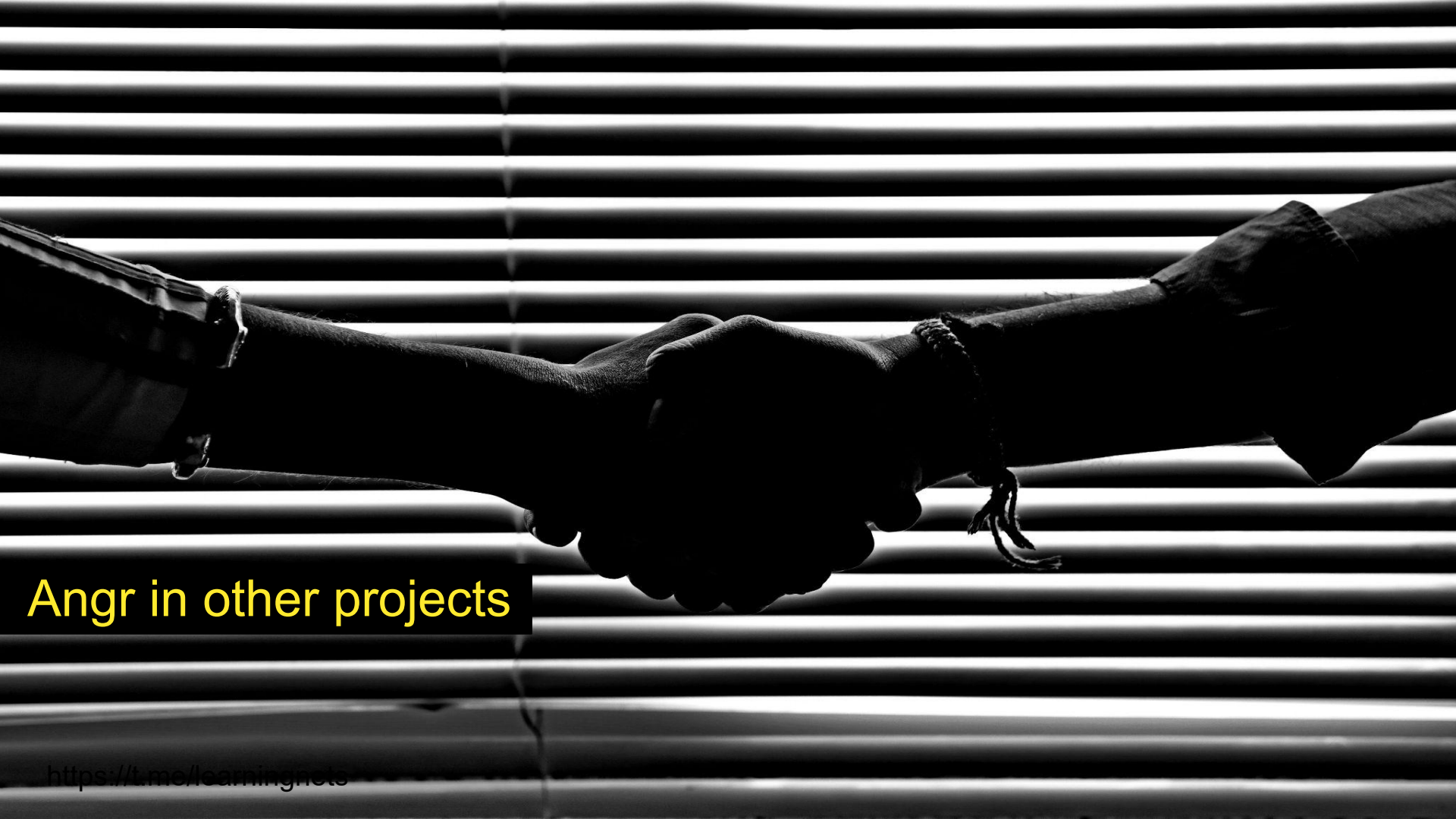
```
print(simgr.found[0].posix.dumps(0))
```

Profiling results



Maslow's Hierarchy of Symbolic Execution





Angr in other projects

Angr-Management

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

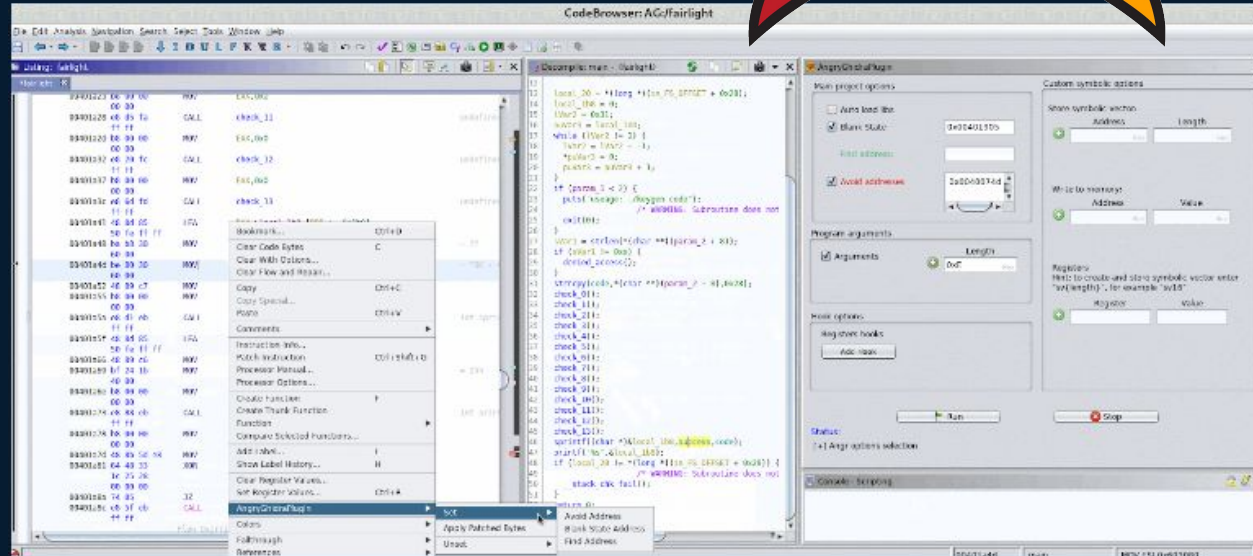
The screenshot displays the Angr-Management GUI with the following components:

- Menu Bar:** File, View, Analyze, Plugins, Help
- Toolbar:** Disassembly, Proximity, Pseudocode, Patches, Symbolic Execution, States, Interaction
- Disassembly Pane:** Shows assembly code for a function (likely a password prompt) with a control flow graph. The code includes instructions like `mov [esp], 0`, `call read`, `puts "Password: "`, and `call puts`. A red box highlights a `call` instruction.
- Hex View:** Displays the raw hex bytes of the code, such as `00485b0 05 b8 00 00 00 00 8b 55 f4 63 35 15 14 00 00 00`.
- Strings List:** Lists strings found in the binary, including "Password:", "Username:", "Go away!", "Welcome to the admin console, trusted user!", "SOSNEAKY", and "/lib/ld-linux.so.2".
- Console Window:** Shows the output of a Python script, including the prompt `In [1]:`.
- Log Window:** Displays error messages from the angr analysis engine, such as `Failed to calculate the stack pointer offset at pc 0x0048655`.

AngryGhidra



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



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

<https://t.me/learningnets>

Driller

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

```
american fuzzy lop 1.86b (test)

process timing |-----| overall results
  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
cycle progress |-----| map coverage
  now processing : 0 (0.00%)
  paths timed out : 0 (0.00%)
stage progress |-----| findings in depth
  now trying : havoc
  stage execs : 1464/5000 (29.28%)
  total execs : 1697
  exec speed : 626.5/sec
fuzzing strategy yields |-----| path geometry
  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%
  levels : 1
  pending : 1
  pend fav : 1
  own finds : 0
  imported : n/a
  variable : 0

[cpu: 92%]
```

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

<https://t.me/learningnets>

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 488b45f0 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 st
; int strcn
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_
0x0040059b e8d0feffff call sym.imp.puts ;[3] ; int pu
; int puts
0x004005a0 eb0c jmp 0x4005ae ;[4]
0x004005a2 488d3dc30000. lea rdi, str.try_Harder_ ; 0x40066c ;
;-- r4ge.avoid:
0x004005a9 e8c2feffff call sym.imp.puts ;[3] ; int pu
; int puts
; JMP XREF from 0x004005a0 (main)
0x004005ae b800000000 mov eax, 0
0x004005b3 c9 leave ; r12; rsp
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: 0x7fff7542d000-0x7fff7542ae80, size: 8576
No Heap section
symbolic address: 0x7fff7542c4cf, size: 15
start symbolic execution, find:0x400594, avoid:['0x4005a9']

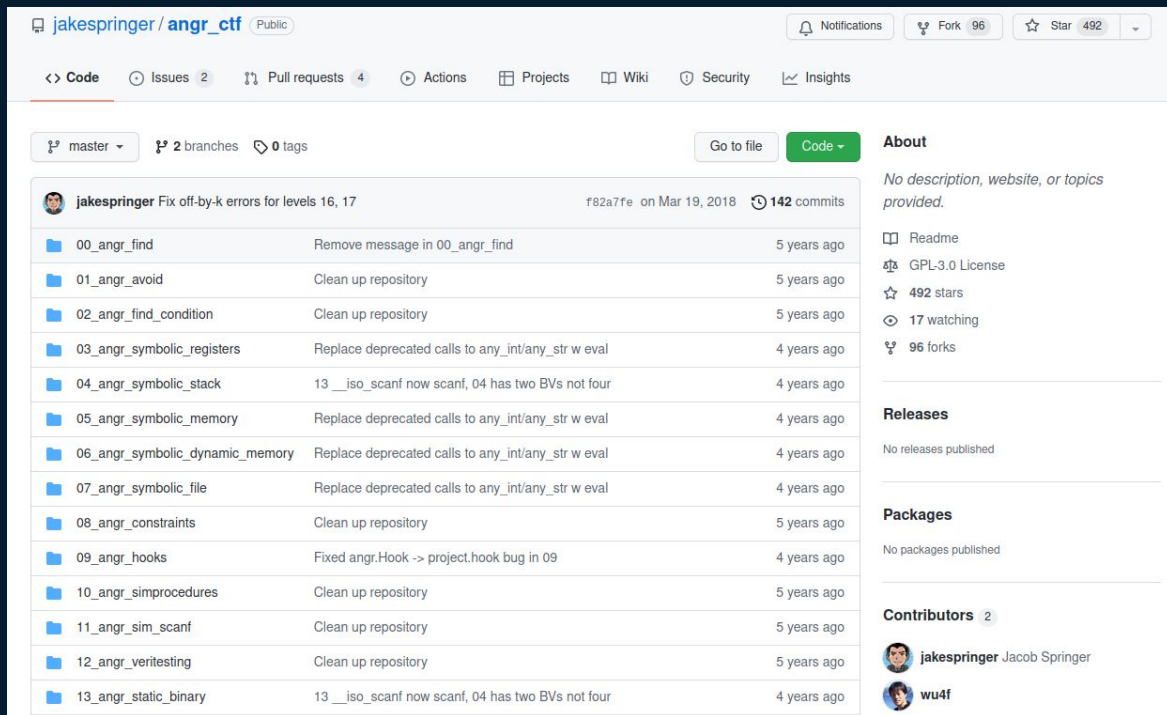
PathGroup Results: <PathGroup with 1 avoid, 1 active, 1 found>
symbolic memory - str: LosFuzzys , hex: 0x4c6f7346757a7a7973000000000000
You want to set debugsession to find address (y/n)? █
```

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

<https://t.me/learningnets>

Further learning

[https://github.com/
jakespringer/angr_ctf](https://github.com/jakespringer/angr_ctf)



Repository: [jakespringer / angr_ctf](#) (Public)

Notifications Fork 96 Star 492

<> Code Issues 2 Pull requests 4 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_scanf 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_veritestest	Clean up repository	5 years ago
13_angr_static_binary	13 __iso_scanf 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
- wu4f**

<https://t.me/learningnets>

Let's solve a crackme!



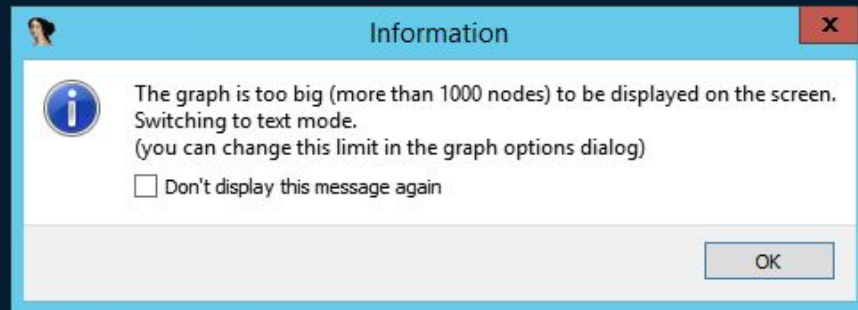
Insomni'Hack 2022
Jannis Kirschner
<https://t.me/learningnets>

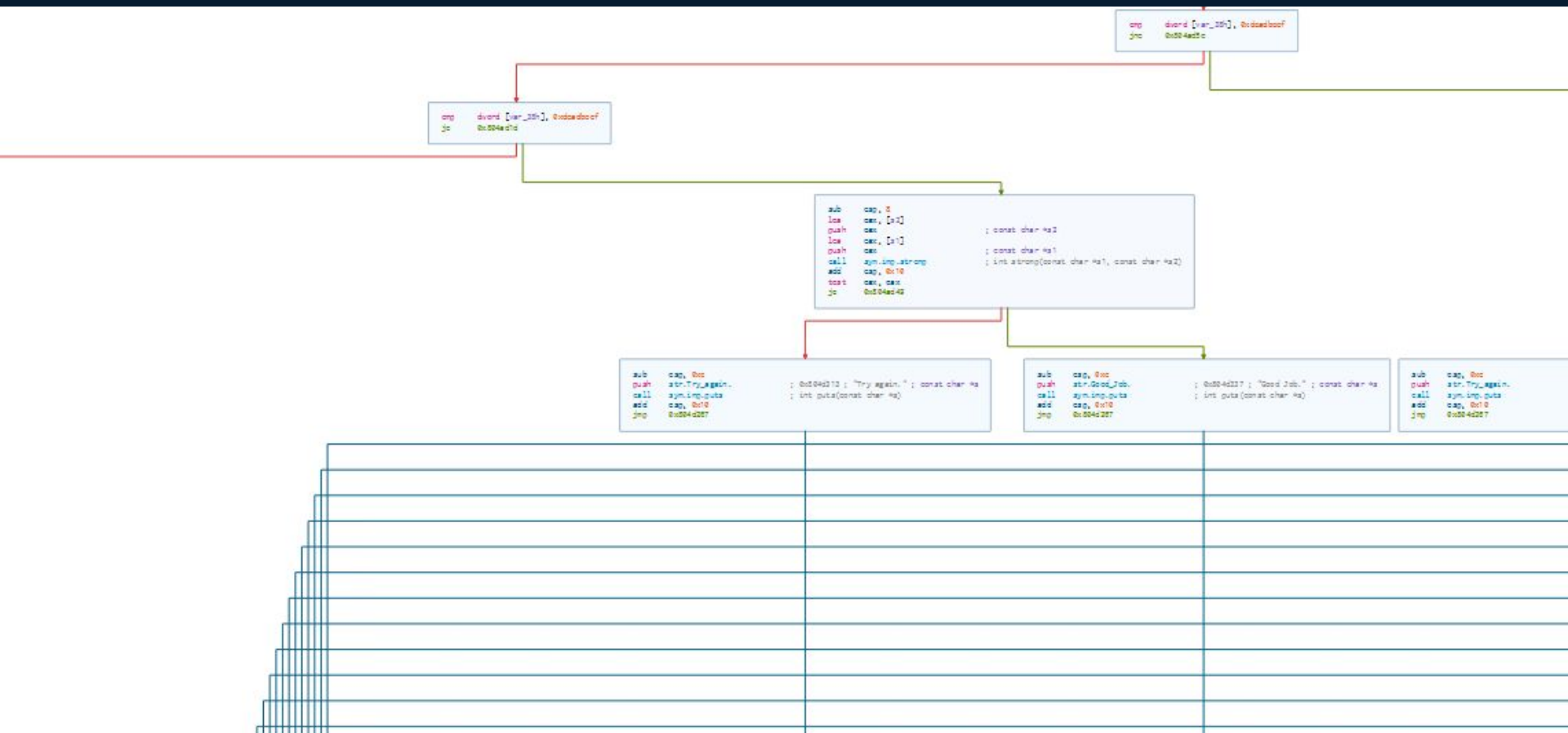


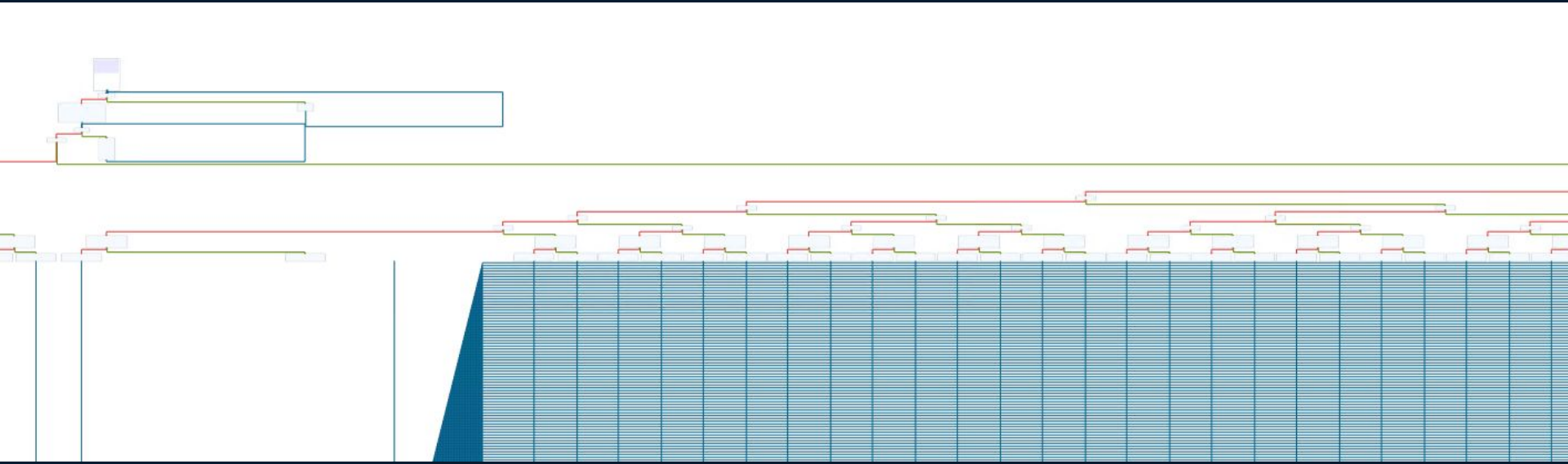
25.03.2022

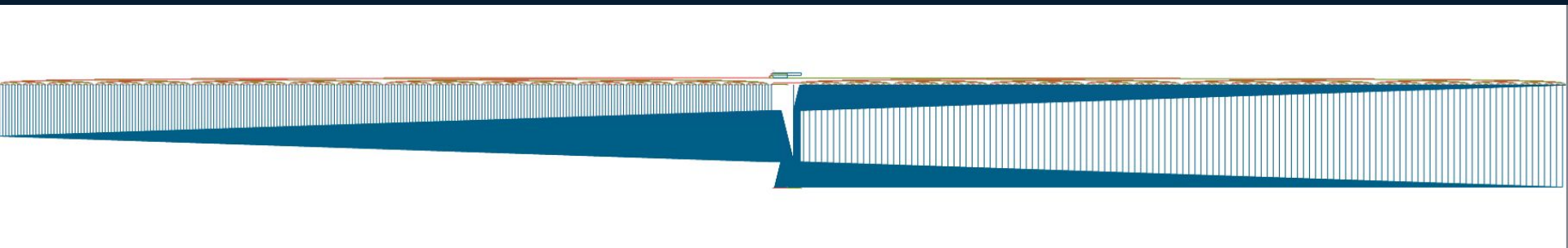
Let's solve a crackme! (Again)











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>
- Work through angr_ctf
- Pwn all the CTFs!!!



@xorkiwi



/in/janniskirschner

