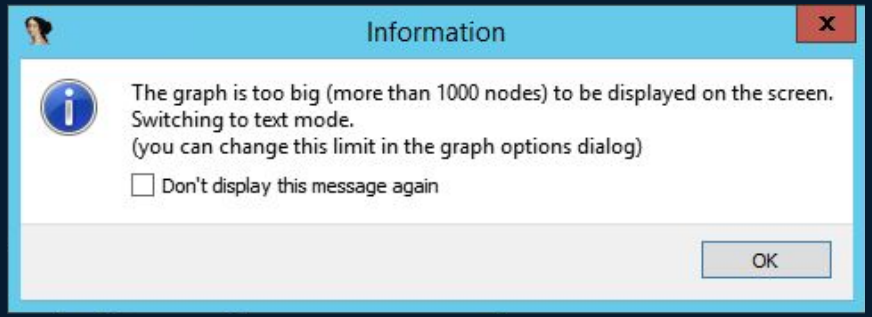
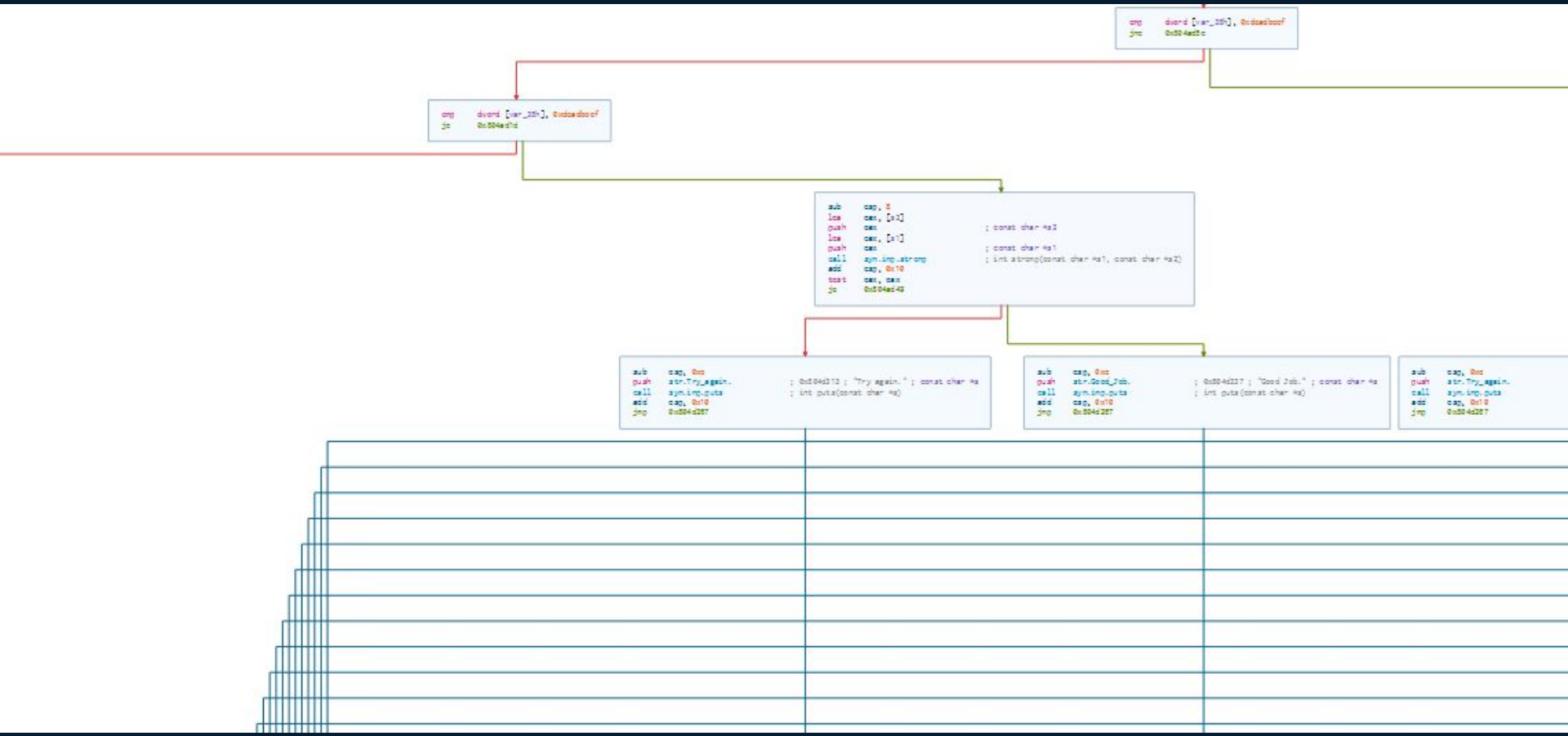
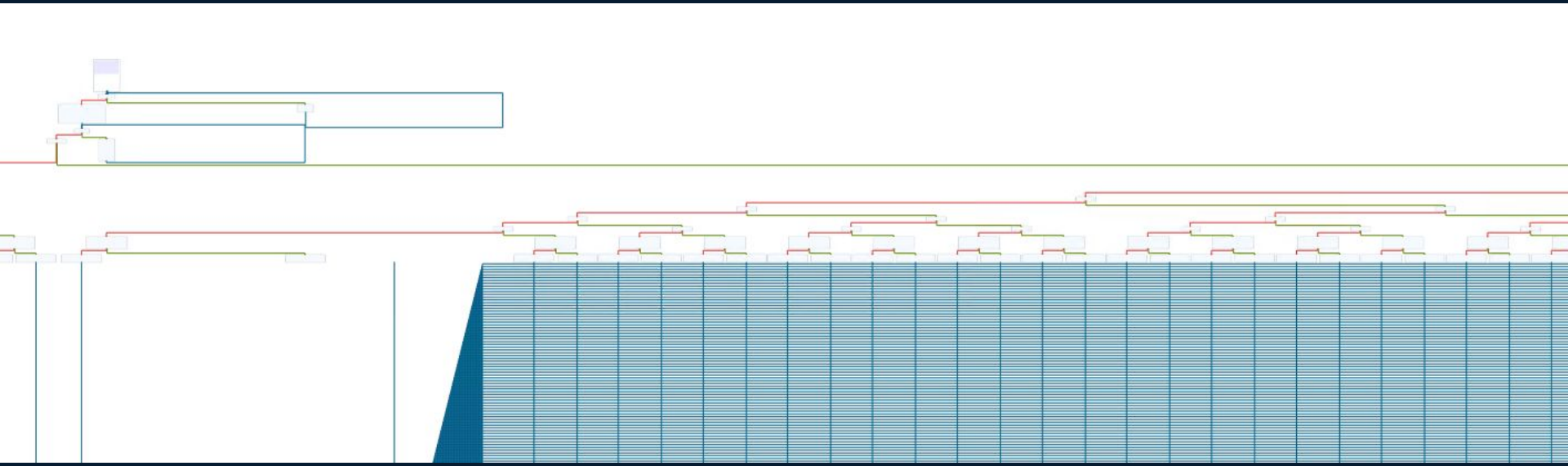
Let’s solve a crackme!

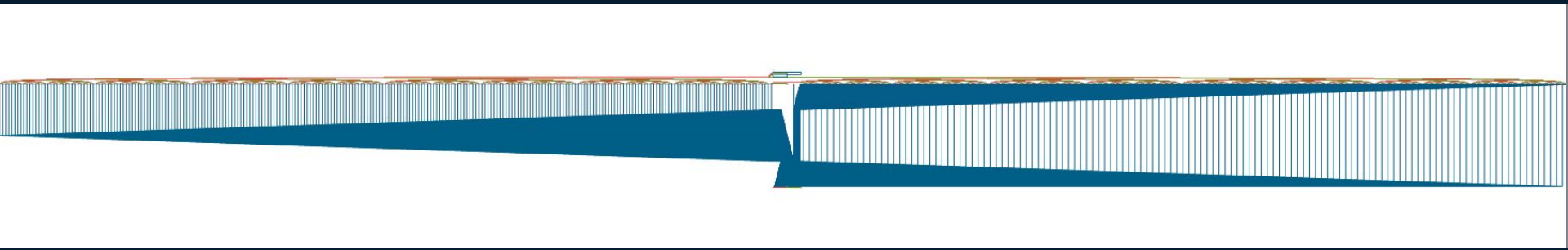
Insomni’Hack 2022
Jannis Kirschner

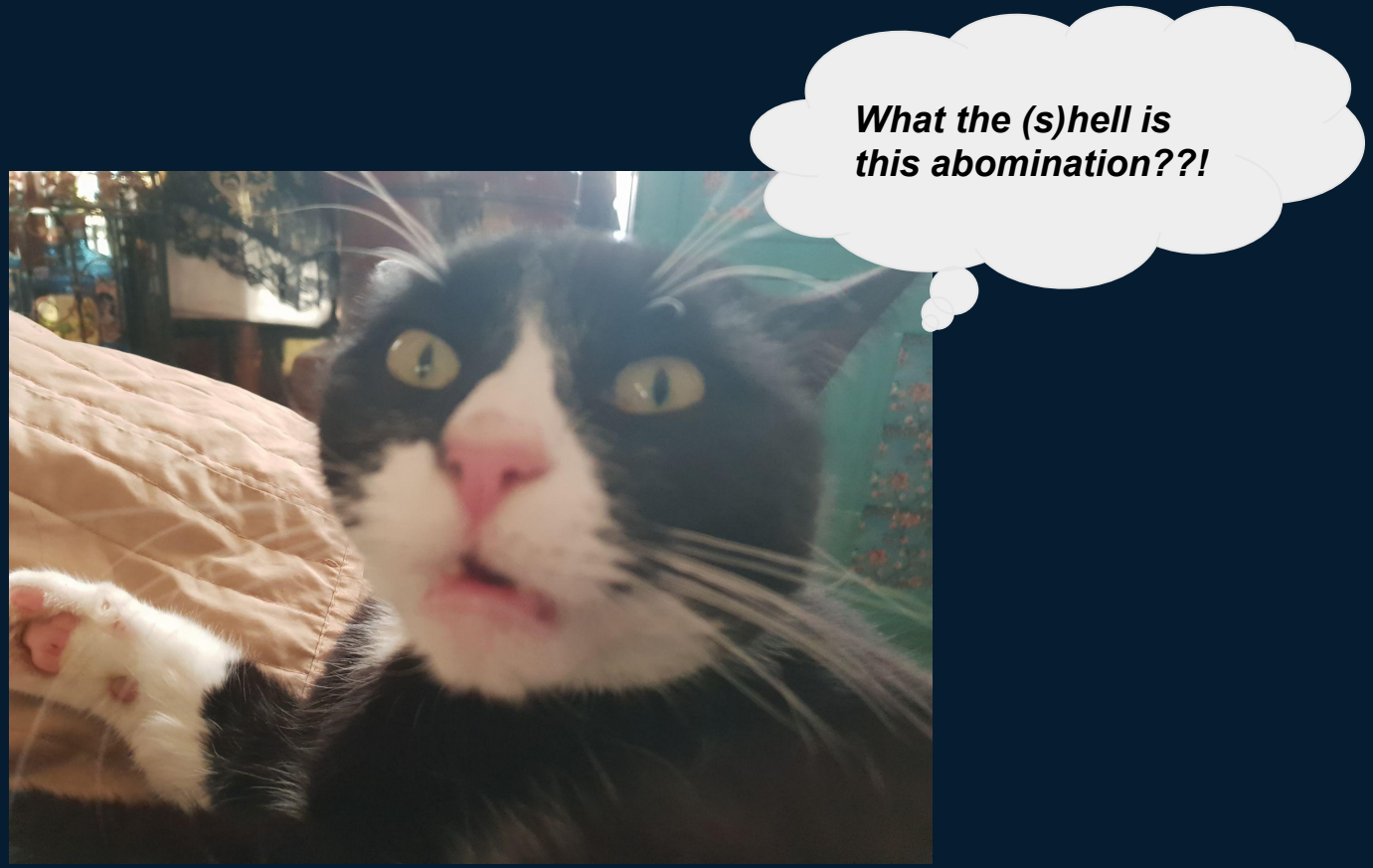
25.03.2022







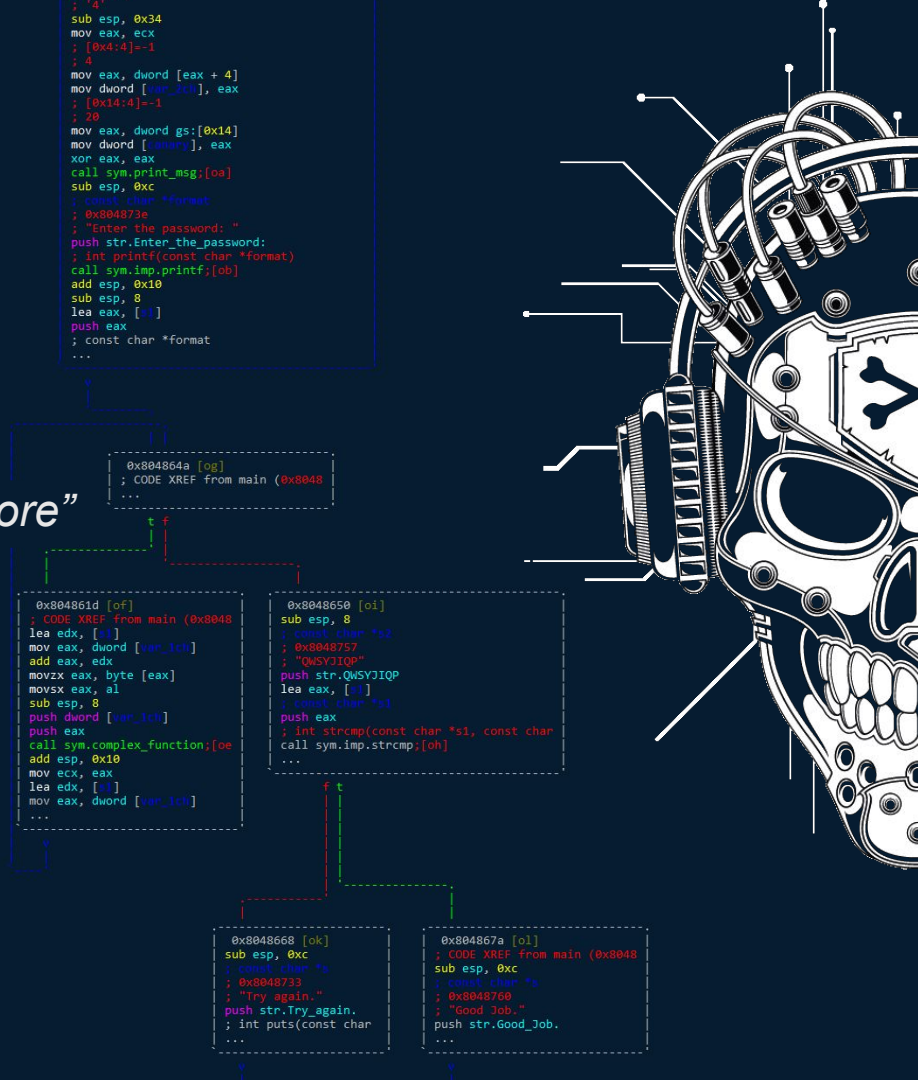




Symbolic Execution Demystified

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

Section 1: Problem State

Investigating the **Why?**

Section 2: Symbolic Execution

Section 3: Angr





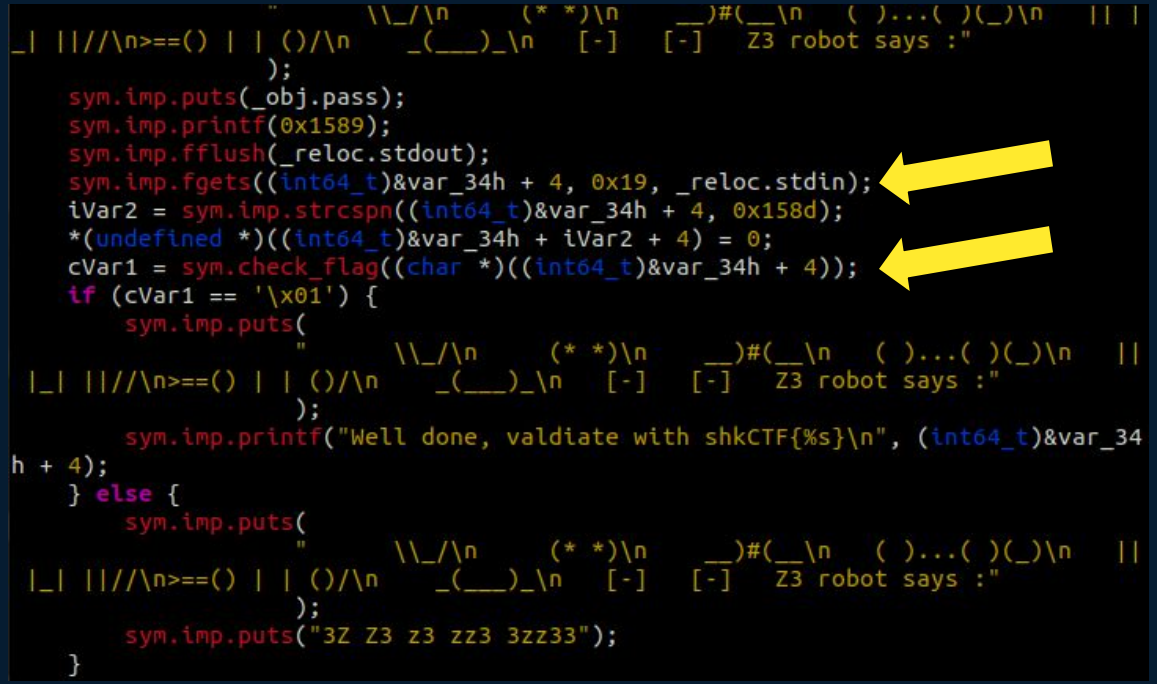
Example: z3_robot (SharkyCTF2020)

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

Creator : Nofix
Pts: 189

https://ctftime.org/event/1034

Static Analysis



x86_64 ELF Binary

Not Stripped

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

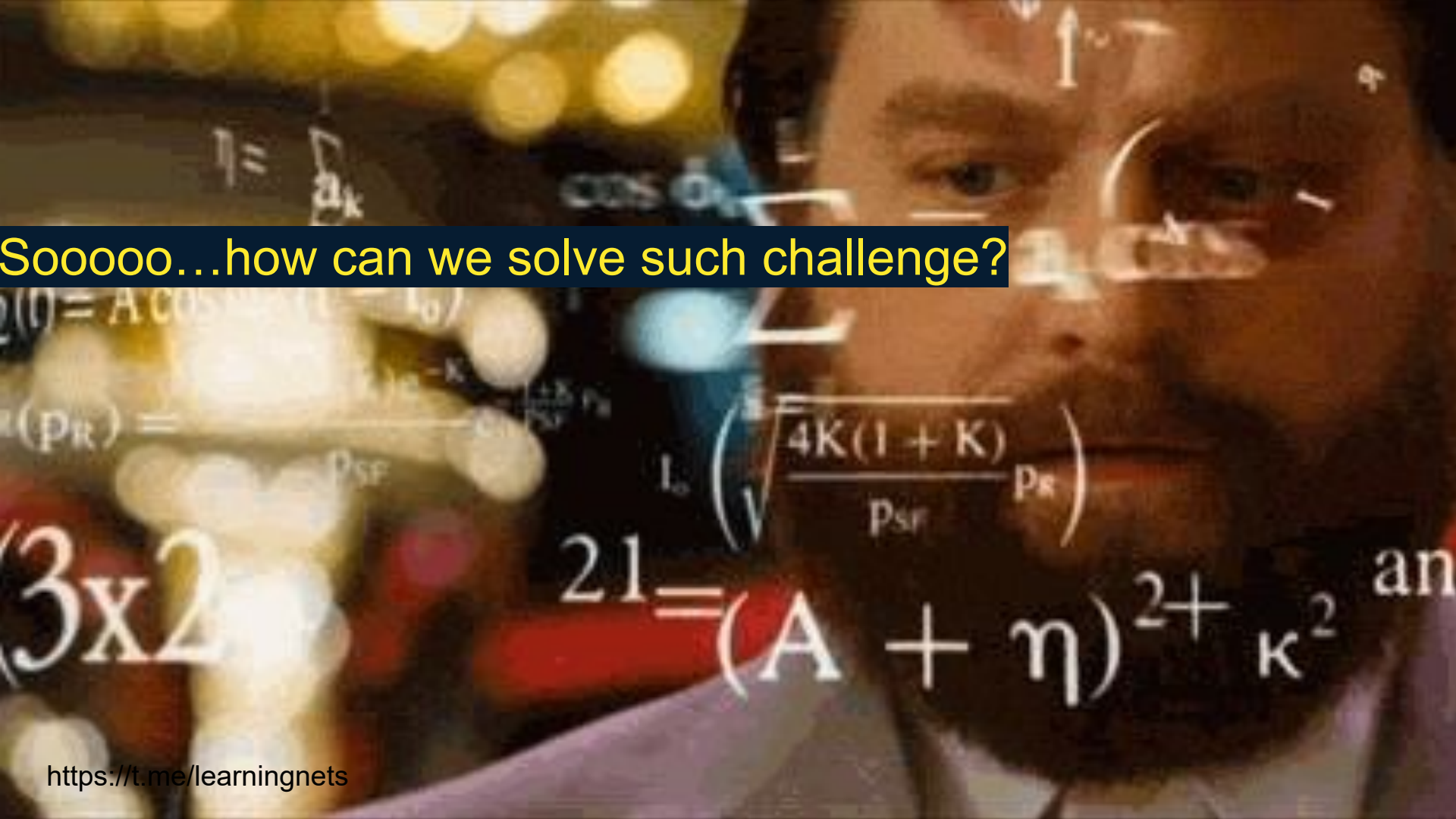
Trying to bruteforce

Binary asks for a 24 characters long passphrase

Brute-forcing it would be infeasible!

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

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?
https://t.me/learningnets

Solving it manually

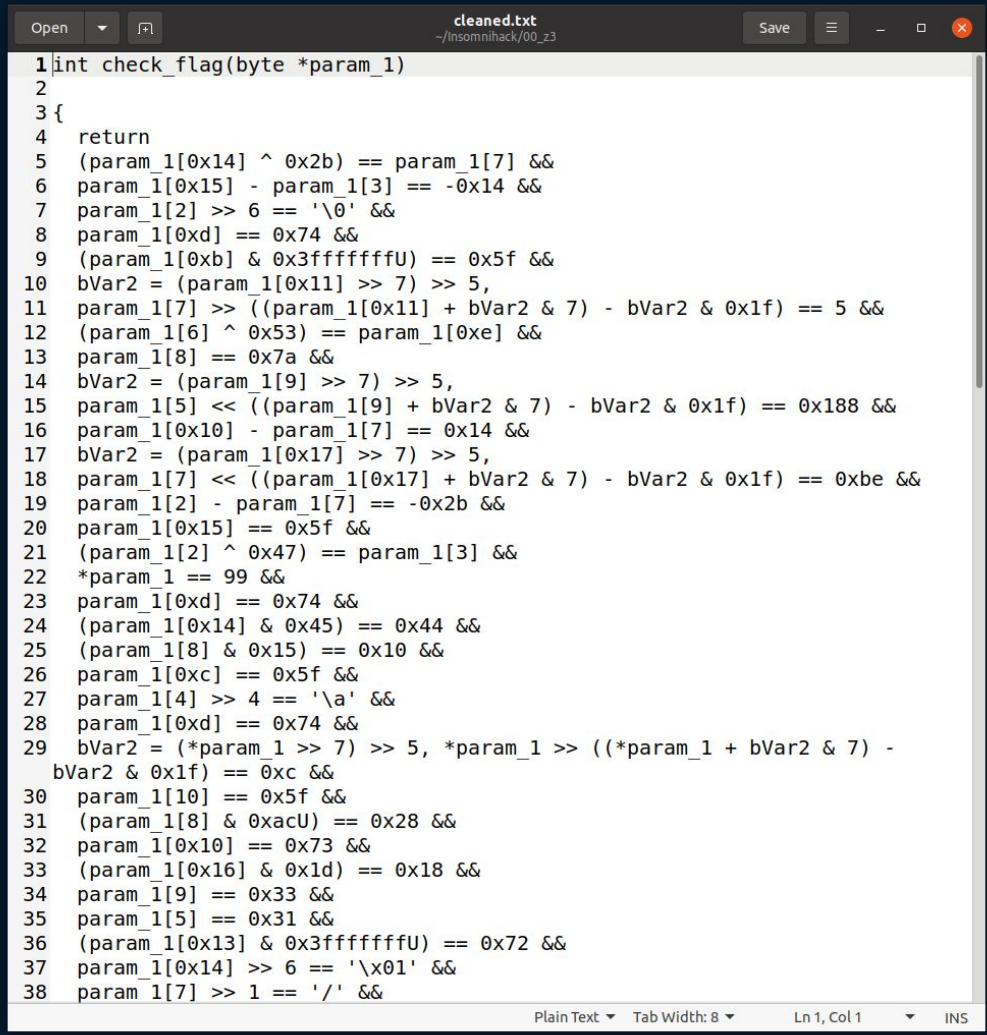
```
[0x00000760]> pdg @ sym.check_flag

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

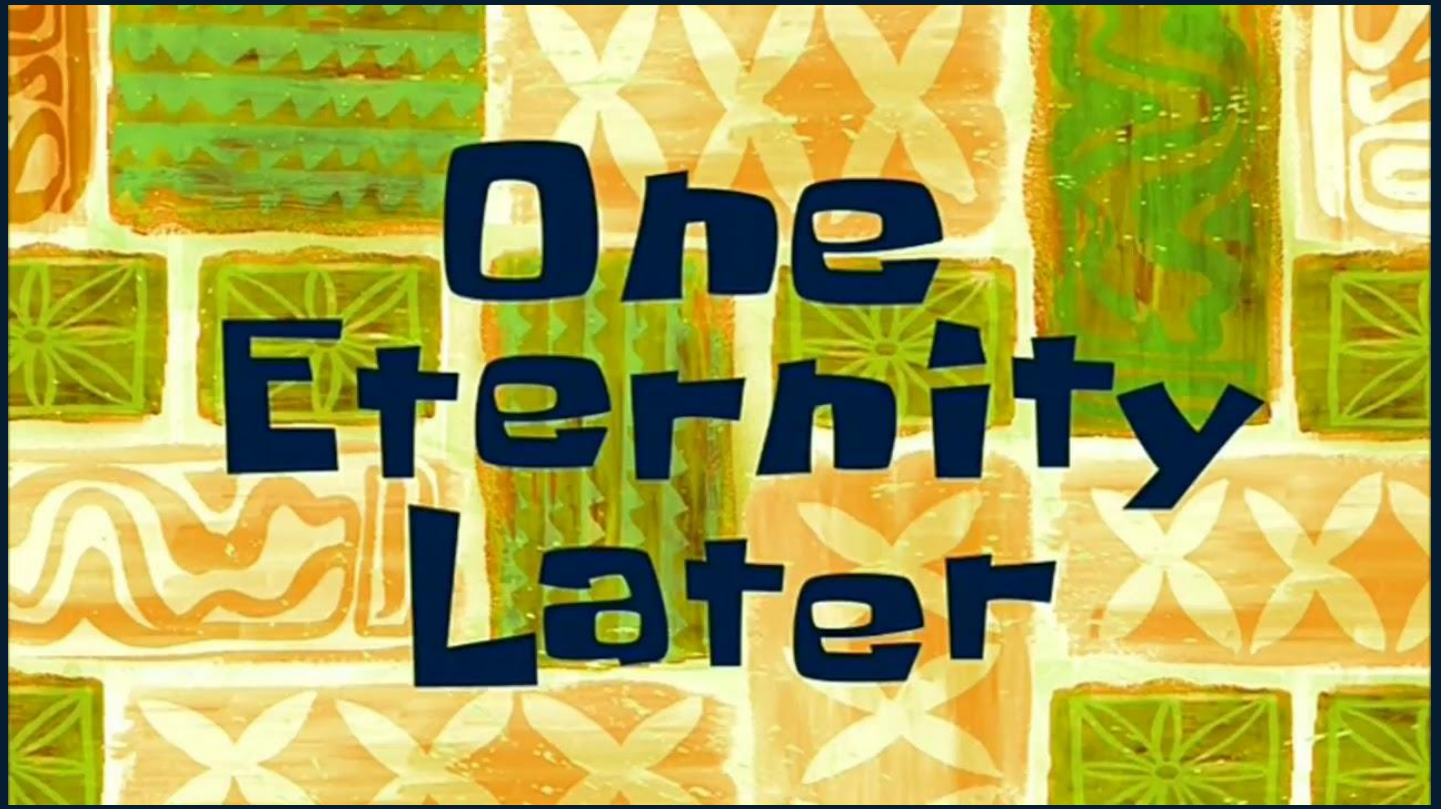
Now we got a nice list with all the constraints that we can work with



Solving it manually

_ _ _ _ s _ _ _ z _ _ _ _ _ t _ _ _ _ _ _ r t _ _ _

```
(param_1[0x14] ^ 0x2b) == param_1[7]
param_1[0x15] - param_1[3] == -0x14
param_1[2] >> 6 == '\0'
param_1[0xd] == 0x74                          <- lower case t
(param_1[0xb] & 0x3fffffffU) == 0x5f
(param_1[6] ^ 0x53) == param_1[0xe]
param_1[8] == 0x7a                            <- lower case z
param_1[0x10] - param_1[7] == 0x14
param_1[0x13] - param_1[0x15] == 0x13          <- 0x13 + 0x5f = 0x72 (lower case r)
param_1[0xc] == 0x5f                          <- underscore
param_1[0xf] >> 1 == '/'
param_1[0x14] == 0x74                         <- lower case t
param_1[4] == 0x73                            <- lower case s
(param_1[0x17] ^ 0x4a) == *param_1
(param_1[6] ^ 0x3c) == param_1[0xb]
param_1[0x15] == 0x5f                         <- underscore
```

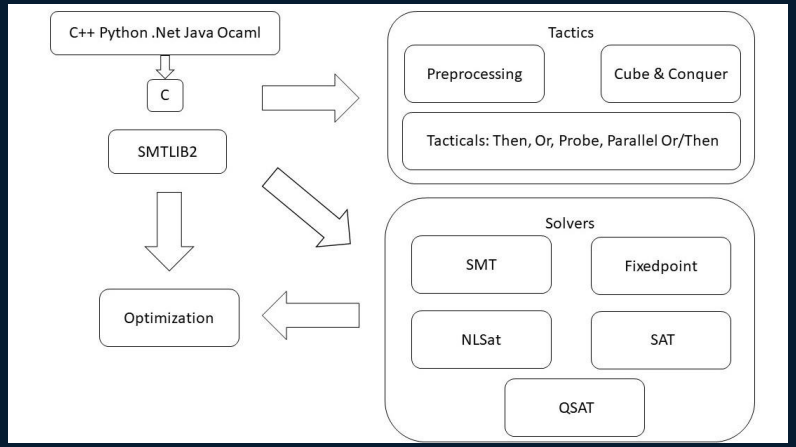


Overview over z3

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

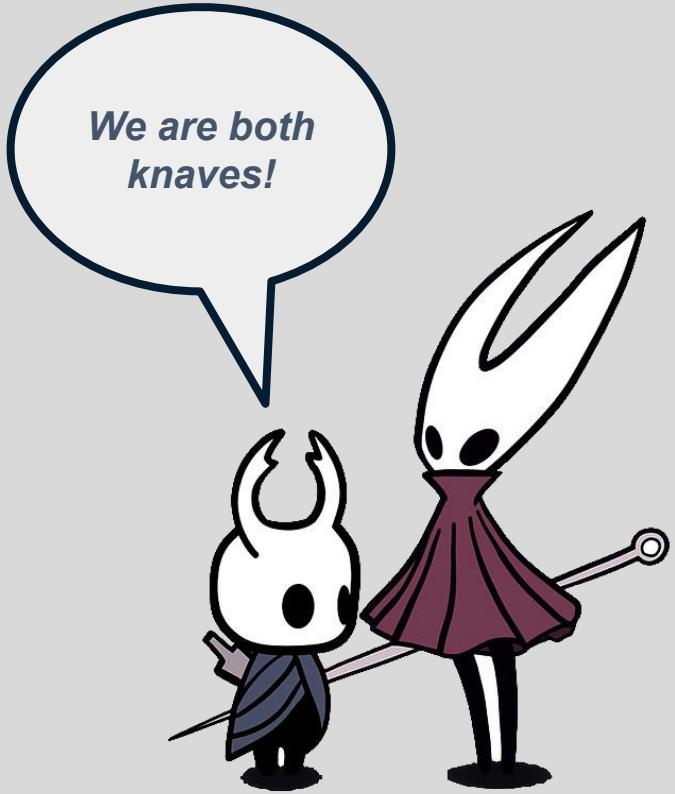


Architecture diagram of z3

A logic puzzle

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

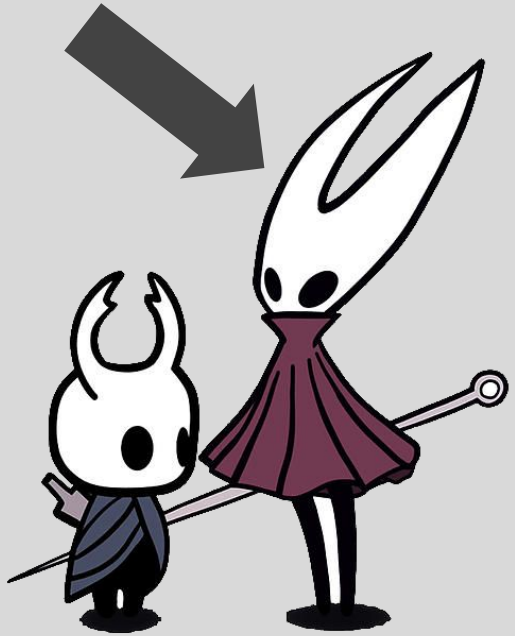
*Two people stand in front of you, Red and Blue. Blue tells you "**we are both knaves**"...who is the knight?*



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:

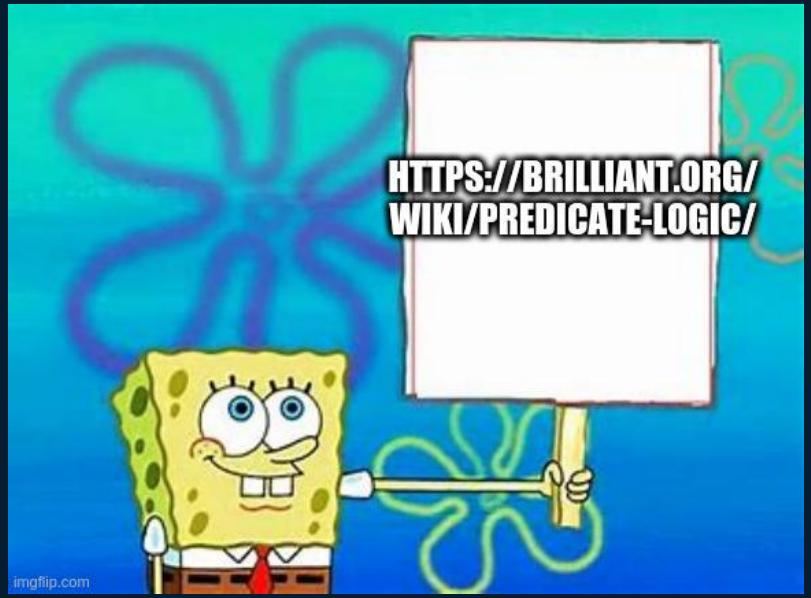
“Given three booleans a,b,c - can the following formula return true: ”
(a and not b) or (not a and c)

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:



```
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

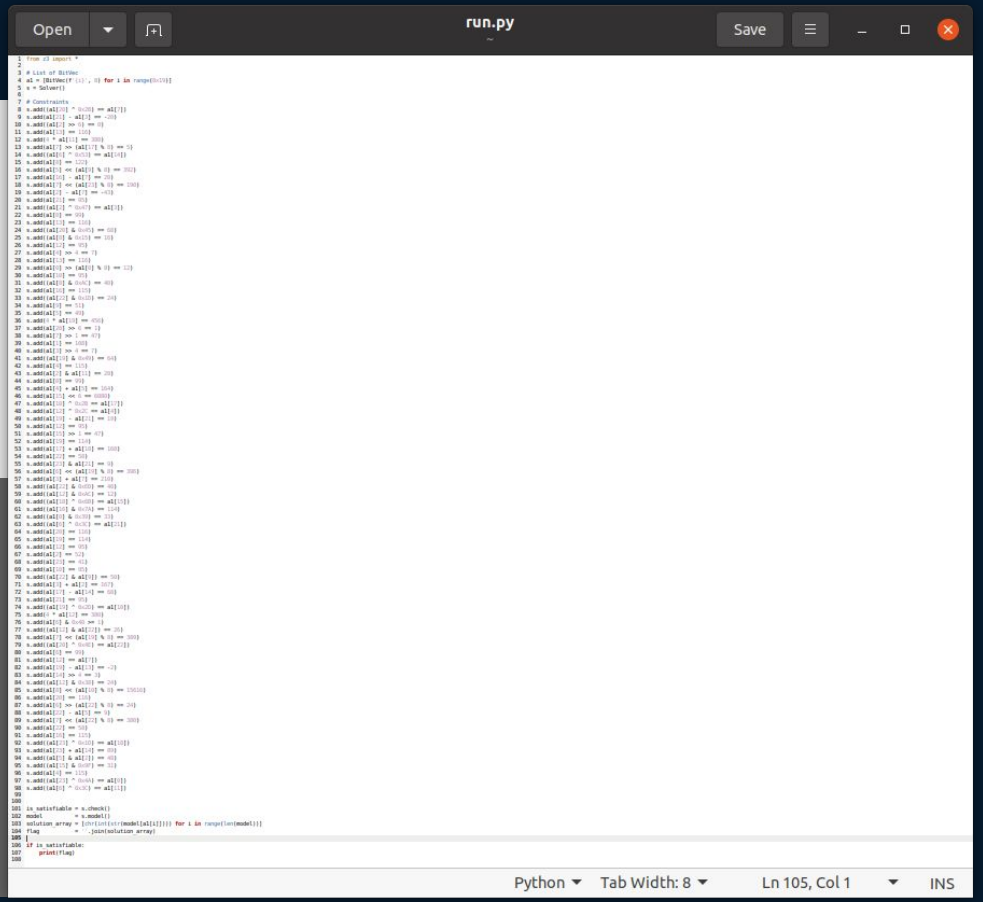
Compute model and convert solved bitvector integers to characters

Display flag

Solution script

~100 Lines of Code

91 Constraints



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

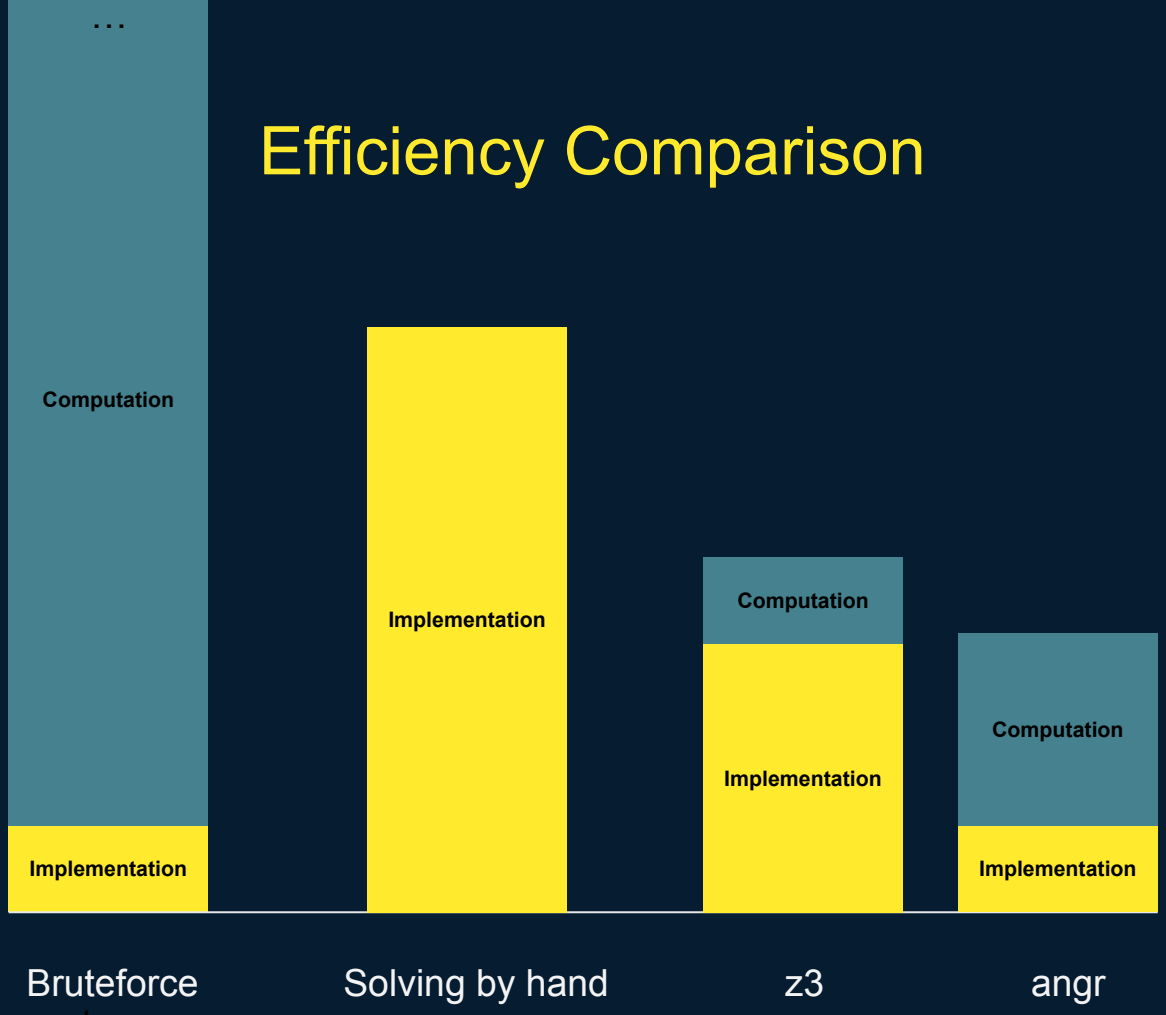
Any guesses to how many lines of code we can reduce it?

We can do the same in about

4

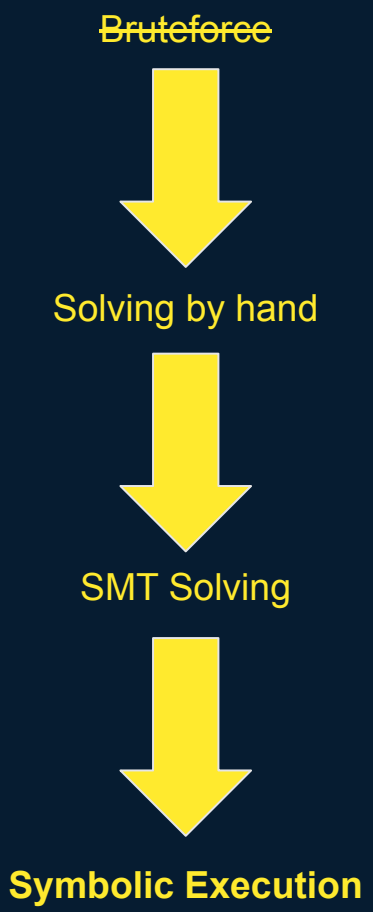
lines of code

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**



Introducing

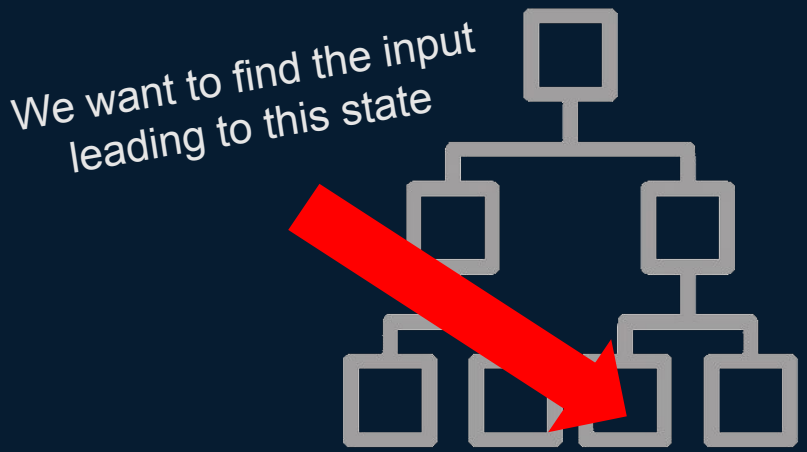
Symbolic Execution

Section 2: Symbolic Execution

Illuminating the **What?**

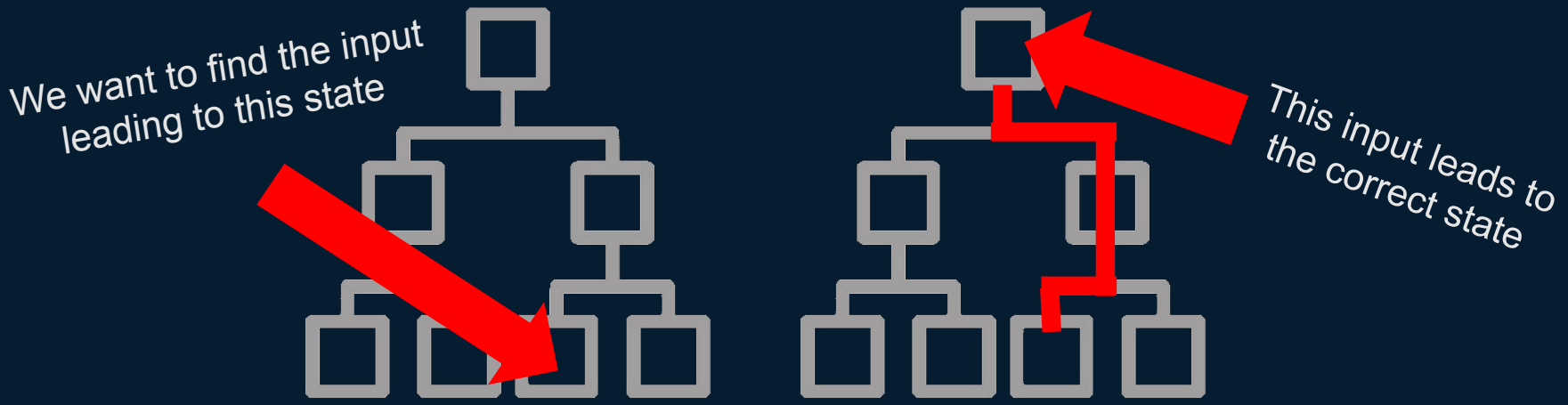
Symbolic Execution is a

"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**
+
Symb**olic** Execution

= Concolic Testing

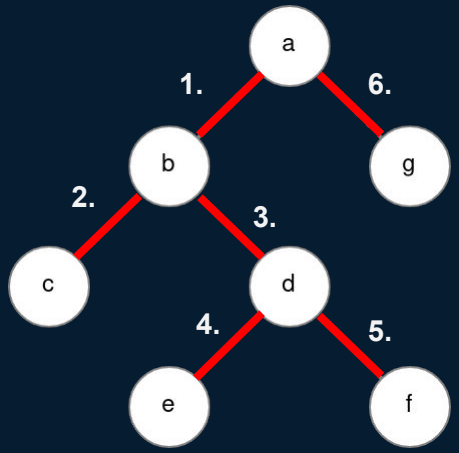
“Dynamic” Concolic Testing

Concrete **Testing**
+
Symb**olic** Execution

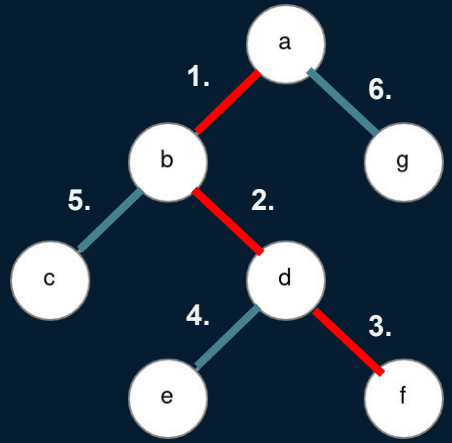
= Concolic Testing

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:                          ←¬(λ < 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

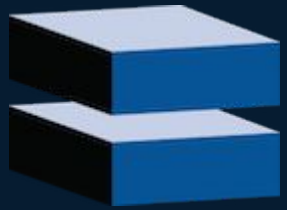
```
void ValidSize()                          
{                                         
        var size = read()          ← 4    
        if (size < 5):             ←True  
                printf(“Works”)    ←λ < 5 
        else:                      ←¬(λ < 5)
                crash()            ←λ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

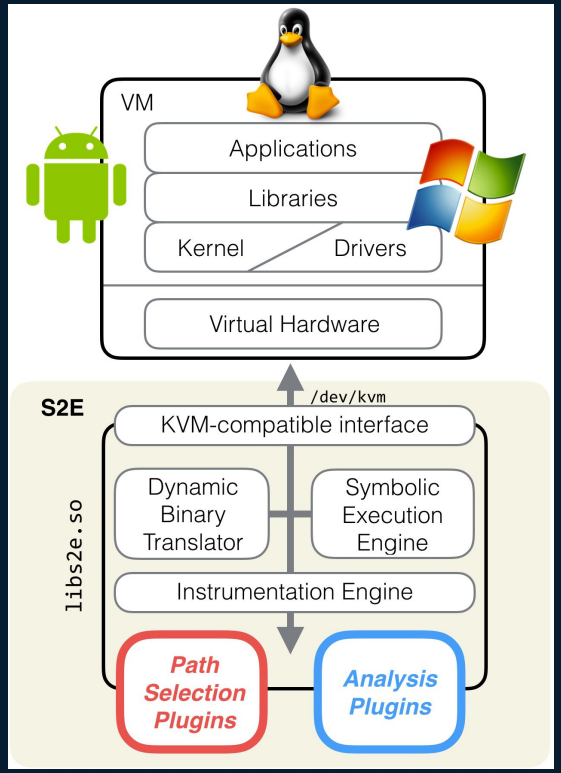
Program Validation Tradeoffs

Different symbolic execution tools

Different symbolic execution tools

S²E: The Selective Symbolic Execution Platform





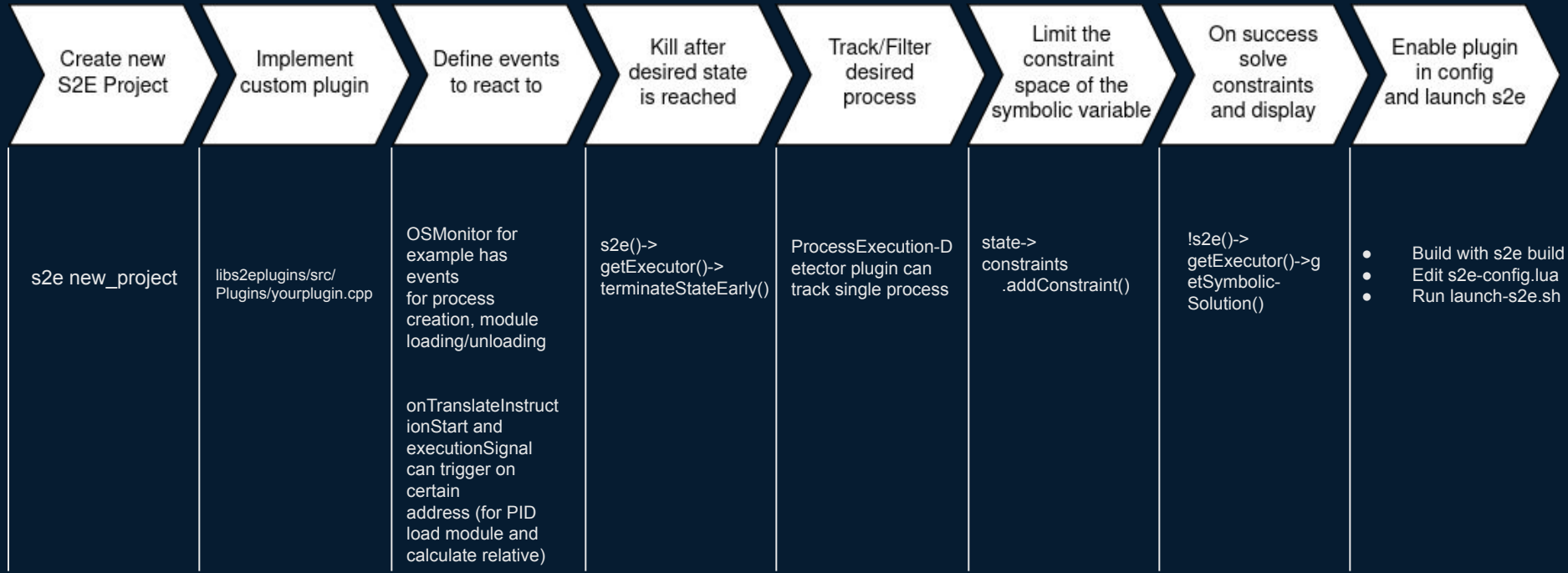
S2E Architecture Diagram

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



unbreakable-ctf-s2e/Goo

https://github.com/adrianherrera/unbreakable-ctf-s2e/blob/master/l 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 [illegible] on Apr 25, 2019 History

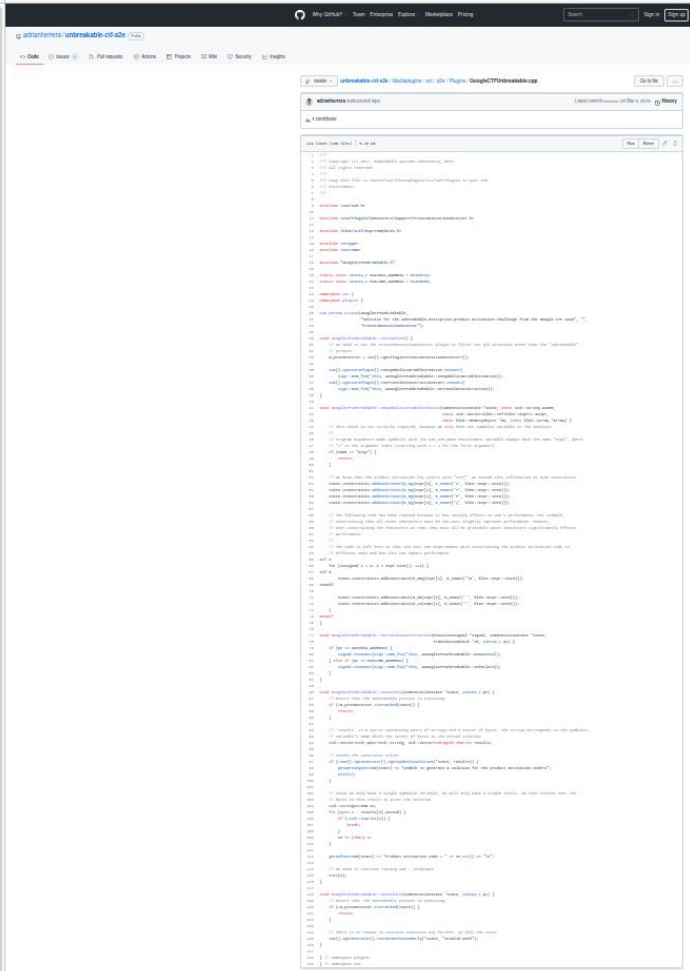
1 contributor

44 lines (33 sloc) 1.16 KB Raw Blame

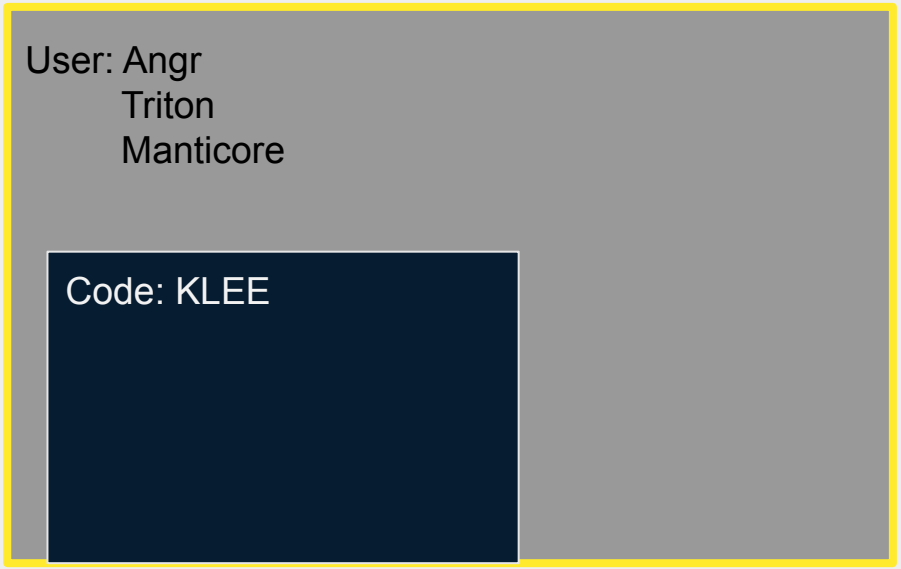
```
///
/// Copyright (C) 2017, Dependable Systems Laboratory, EPFL
/// All rights reserved.
///
/// Copy this file to source/s2e/libs2eplugins/src/s2e/Plugins in your S2E
/// environment.
///

#ifndef S2E_PLUGINS_GOOGLE_CTF_UNBREAKABLE_H
#define S2E_PLUGINS_GOOGLE_CTF_UNBREAKABLE_H

#include <s2e/CorePlugin.h>
#include <s2e/Plugin.h>
#include <s2e/S2E.h>

namespace s2e {
namespace plugins {

class ProcessExecutionDetector;

class GoogleCTFUnbreakable : public Plugin {
    S2E_PLUGIN

public:
    GoogleCTFUnbreakable(S2E *s2e) : Plugin(s2e) {
    }

    void initialize();

private:
    ProcessExecutionDetector *m_procDetector;

    void onSymbolicVariableCreation(S2EExecutionState *state, const std::string &name,
                                    const std::vector<klee::ref<klee::Expr>> &expr, const klee::MemoryObject *mo,
                                    const klee::Array *array);
    void onTranslateInstruction(ExecutionSignal *signal, S2EExecutionState *state, TranslationBlock *tb, uint64_t pc);
    void onSuccess(S2EExecutionState *state, uint64_t pc);
    void onFailure(S2EExecutionState *state, uint64_t pc);
};

} // namespace plugins
} // namespace s2e

#endif
```


unbreakable-ctf-s2e/Goo

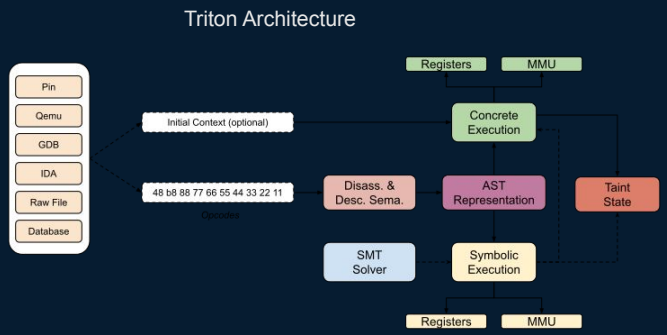
https://github.com/adrianherrera/unbreakable-ctf-s2e/blob/ma 30%



Different symbolic execution tools



Angr/Triton/Manticore

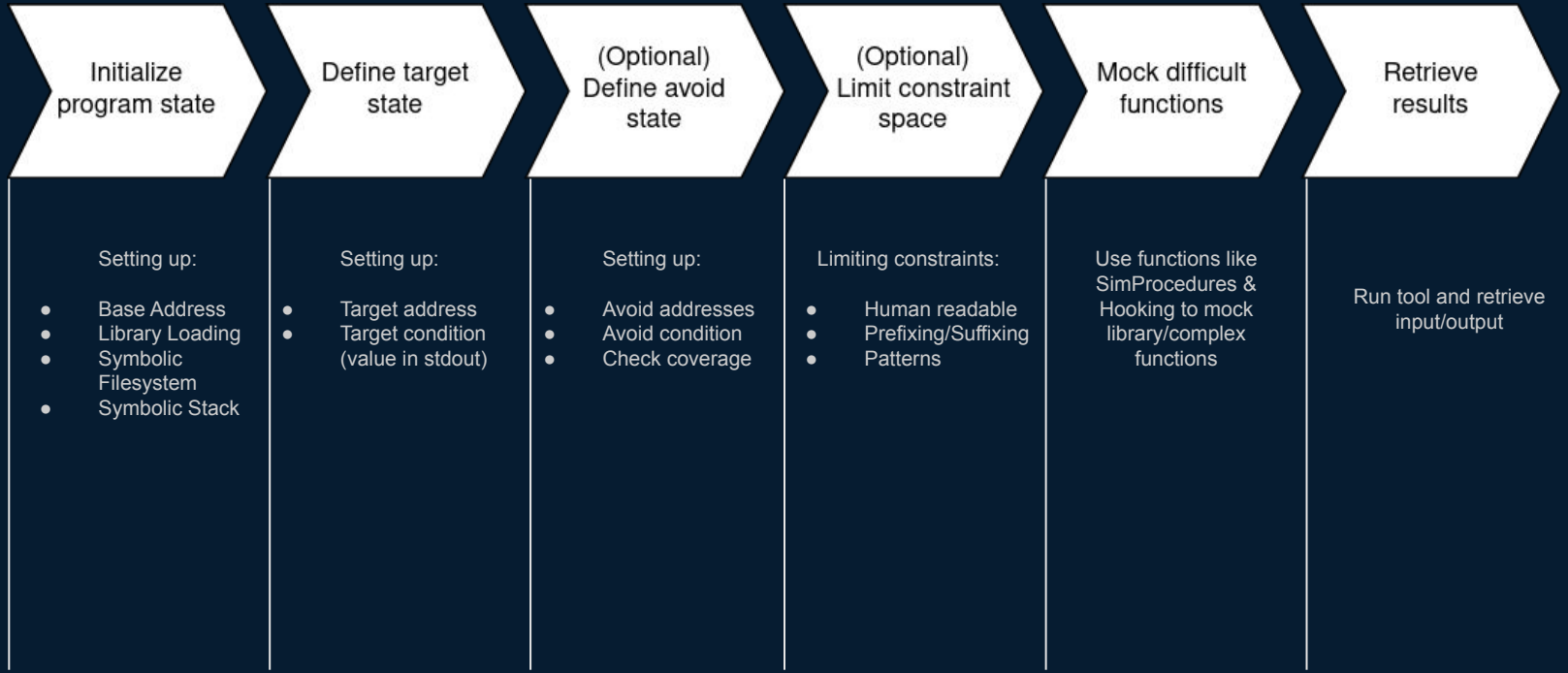


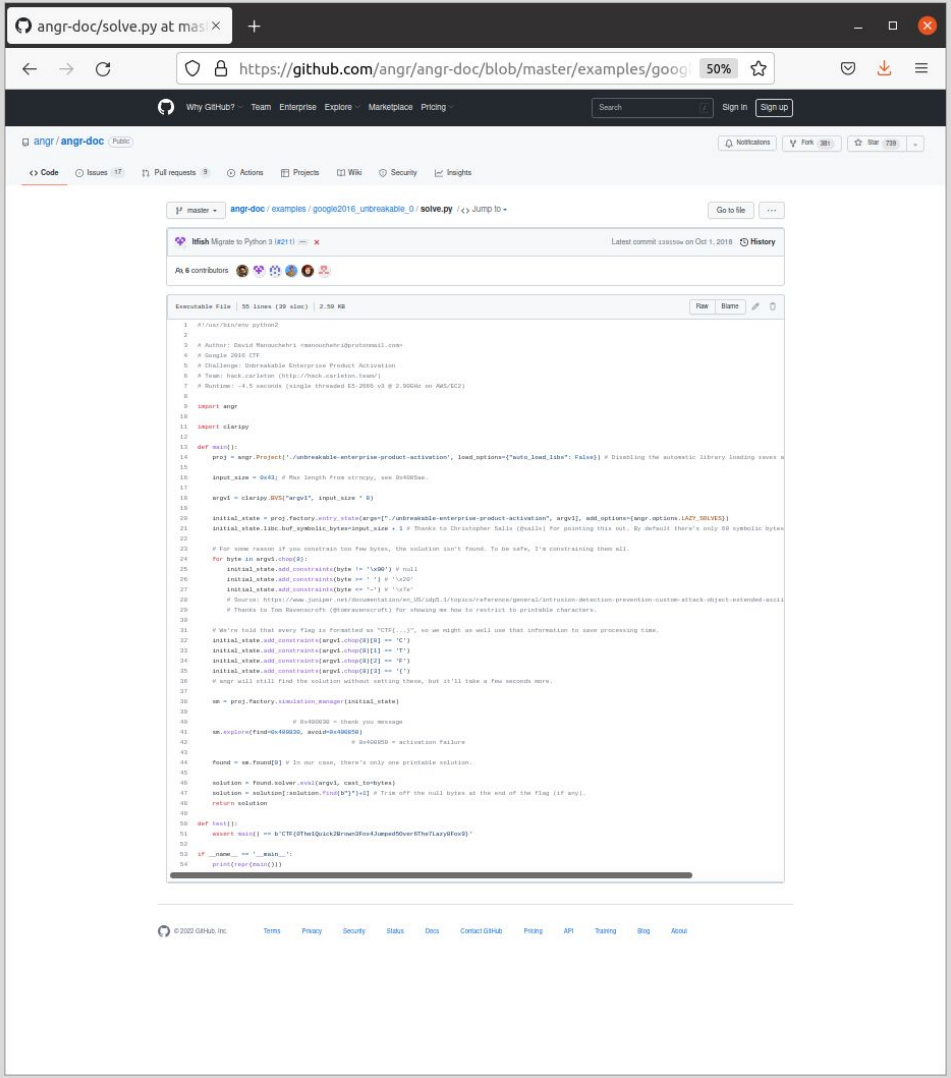
User-level dynamic binary analysis & symbolic execution frameworks (often based on z3).

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





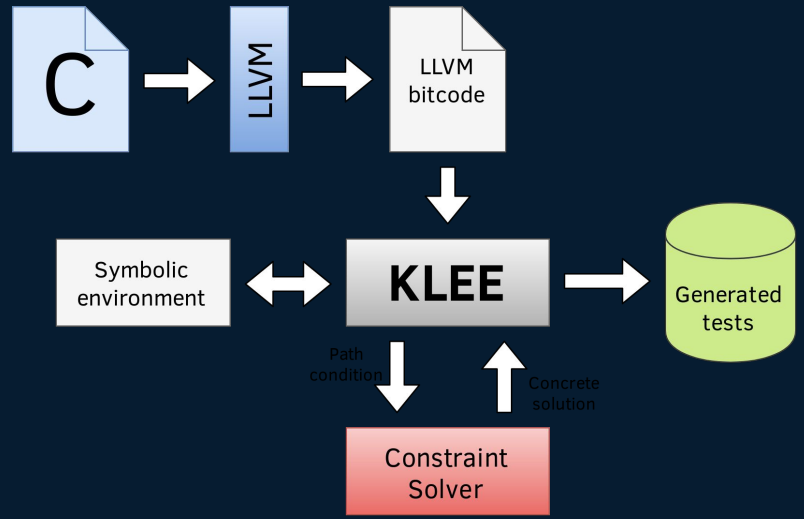
Different symbolic execution tools

KLEE

LLVM-based symbolic execution engine for code-level analysis

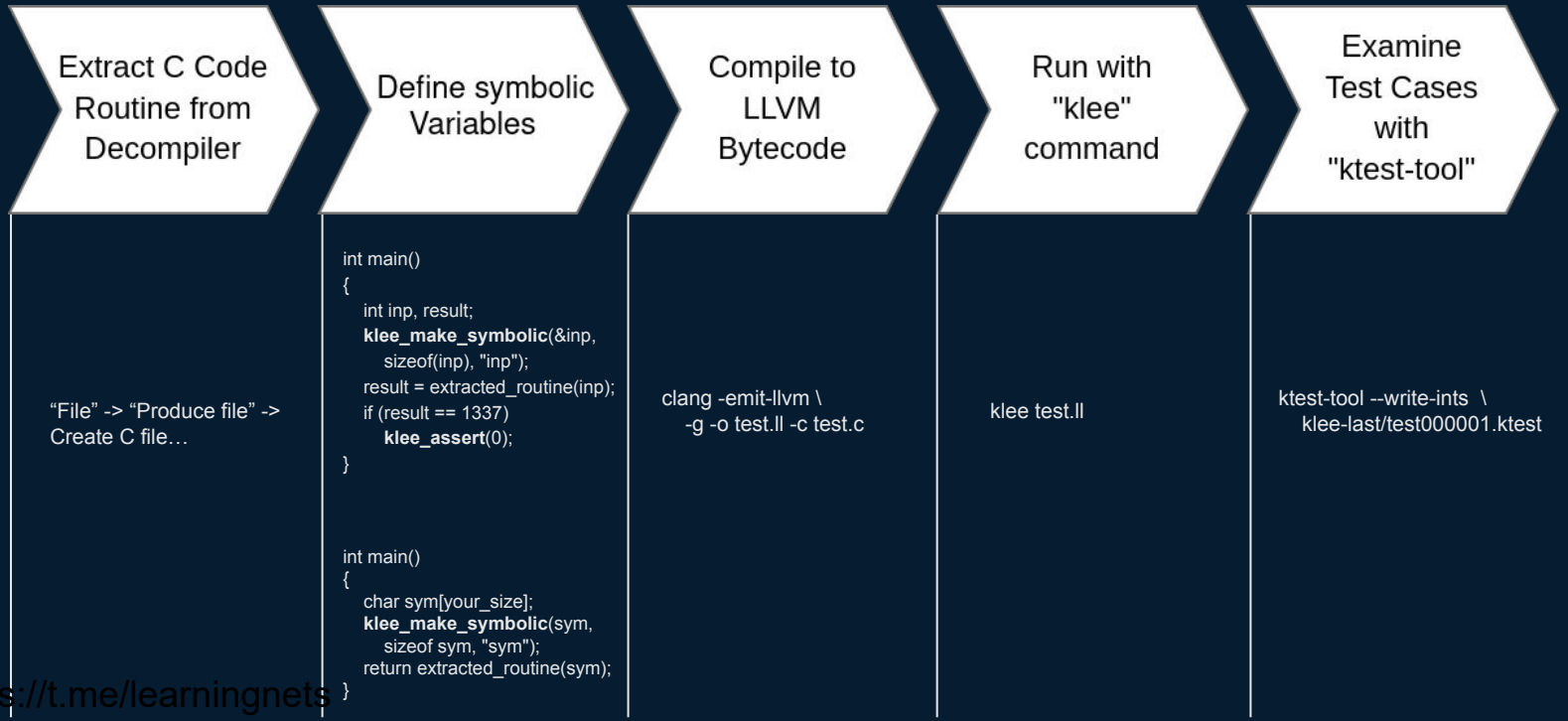
Requires target function to be re-/coded in C and instrumented

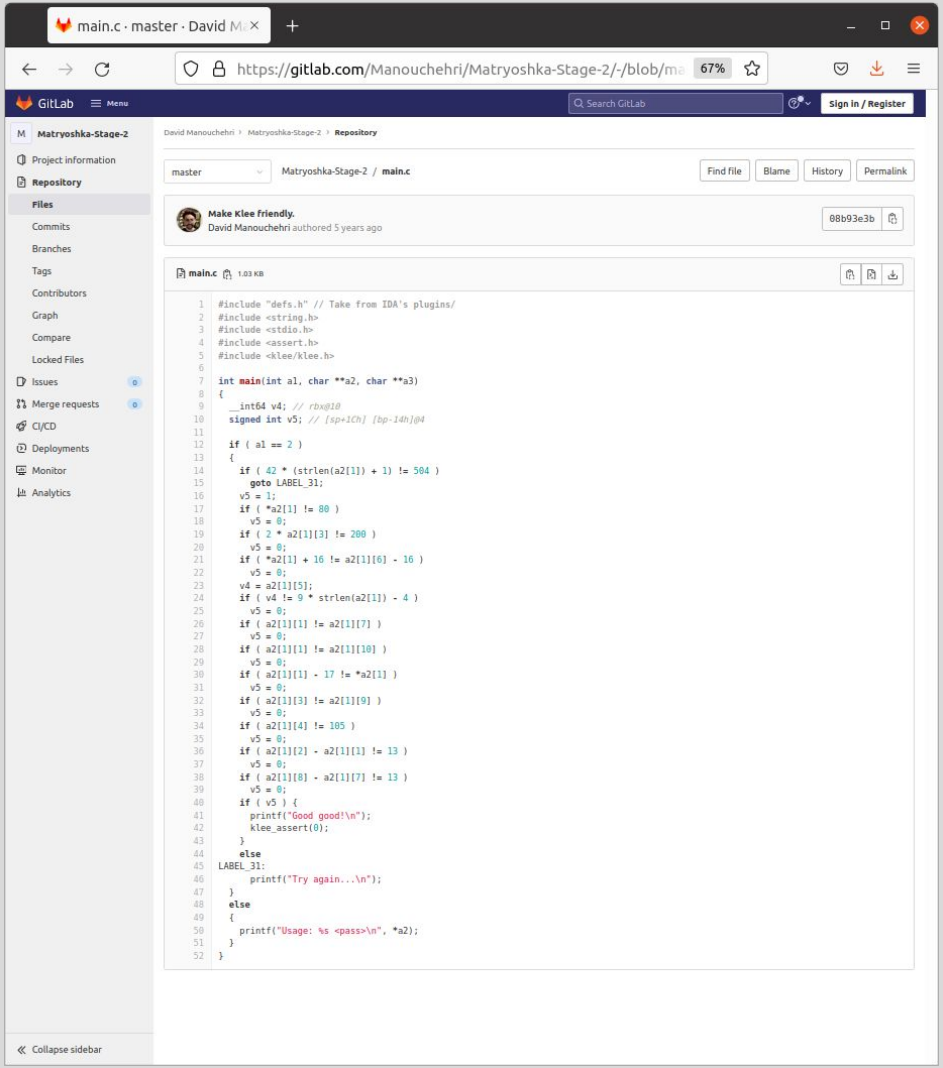
High performance due to smaller overhead compared with other frameworks, as well as nifty features such as coverage, test case and path exporting



KLEE Architecture Diagram

KLEE Walkthrough





What framework should I use?

Do you want to analyze **complex systems & drivers**?

- no → Do you want to quickly analyze **c source code**?
- yes → **S²E** 

Do you want to quickly analyze **c source code**?

- no → Do you want to analyze **smart contracts**?
- yes → **KLEE** 

Do you want to analyze **smart contracts**?

- no → Do you require **C++ bindings** for user mode analysis?
- yes → **Manticore** 

Do you require **C++ bindings** for user mode analysis?

- no →  **Angr**
- yes → **Triton** 

What framework should I use?

Do you want to analyze **complex systems & drivers**?

no

yes

S²E

Do you want to quickly analyze **c source code**?

no

yes

KLEE

Do you want to analyze **smart contracts**?

no

yes

Manti

Do you require **C++ bindings** for user mode analysis?

no

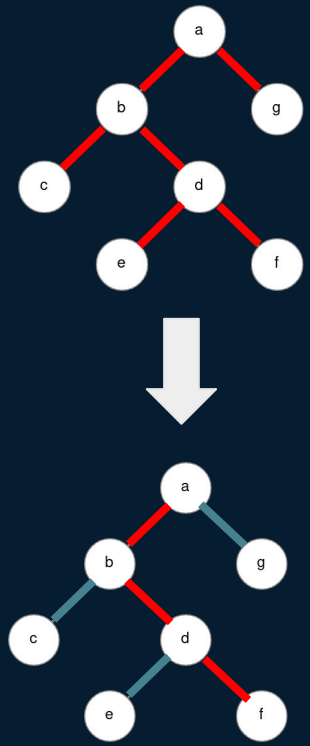
Angr

A wonderful solution for most generic cases/CTF

riton

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?**

angr

Extensive Binary Analysis Framework

Convenient Python3 Interface

Valgrind

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

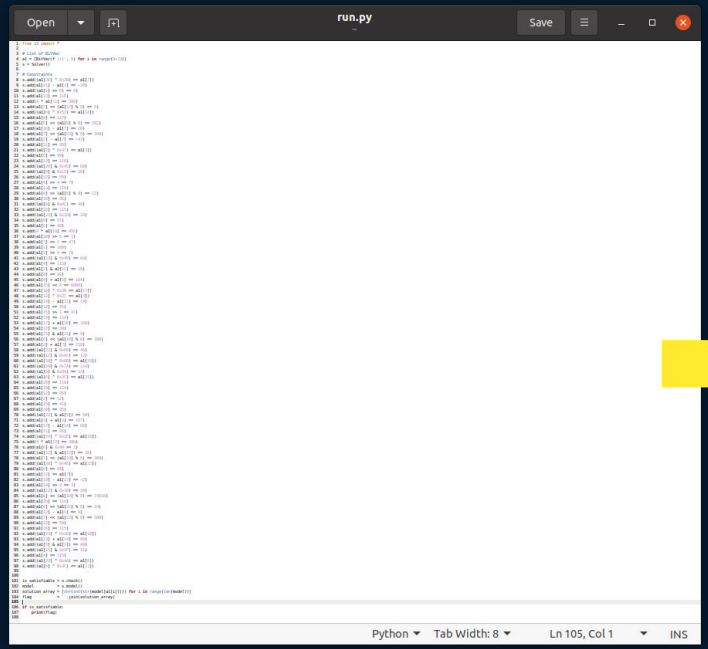
Symbolic + Concolic Execution

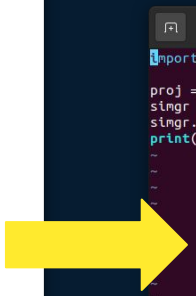
Developed by UCSB

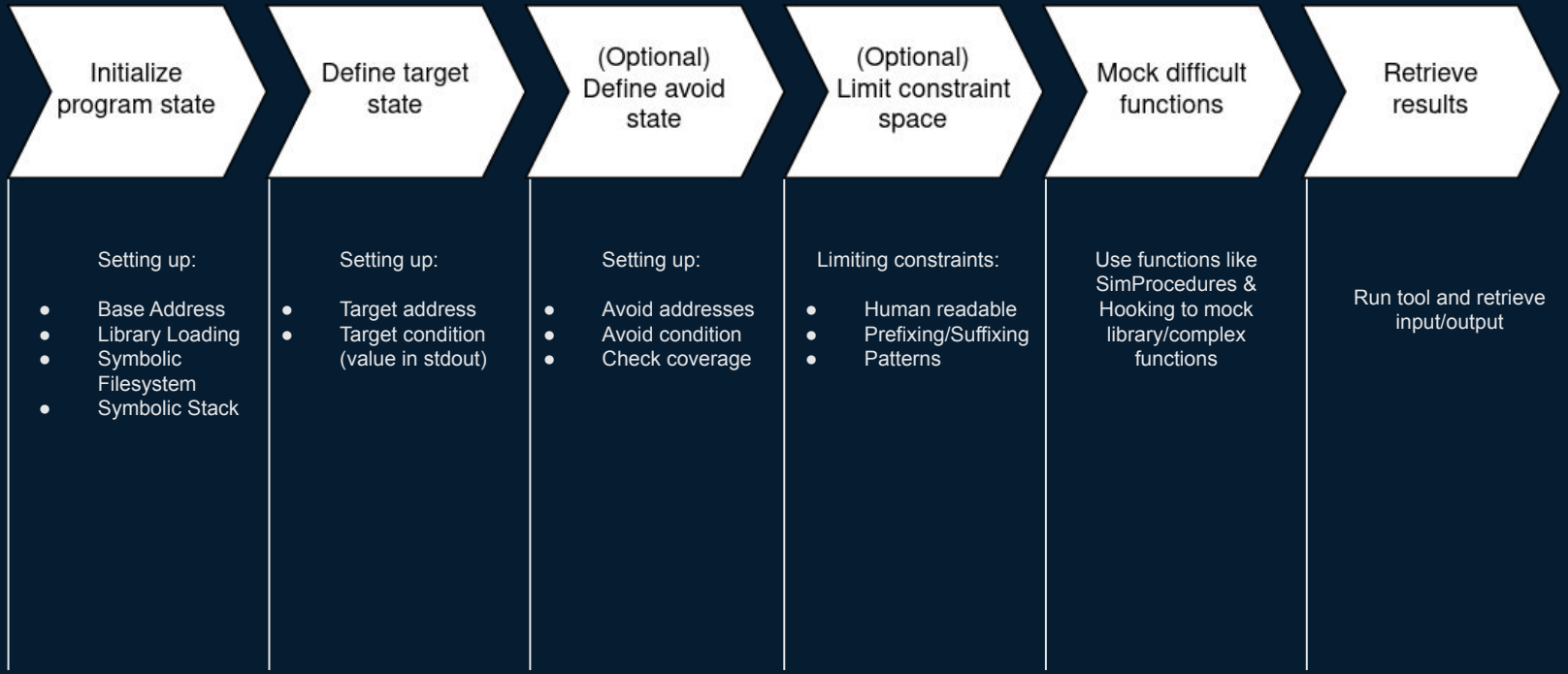
Won 3rd in DARPA
Cyber Grand Challenge

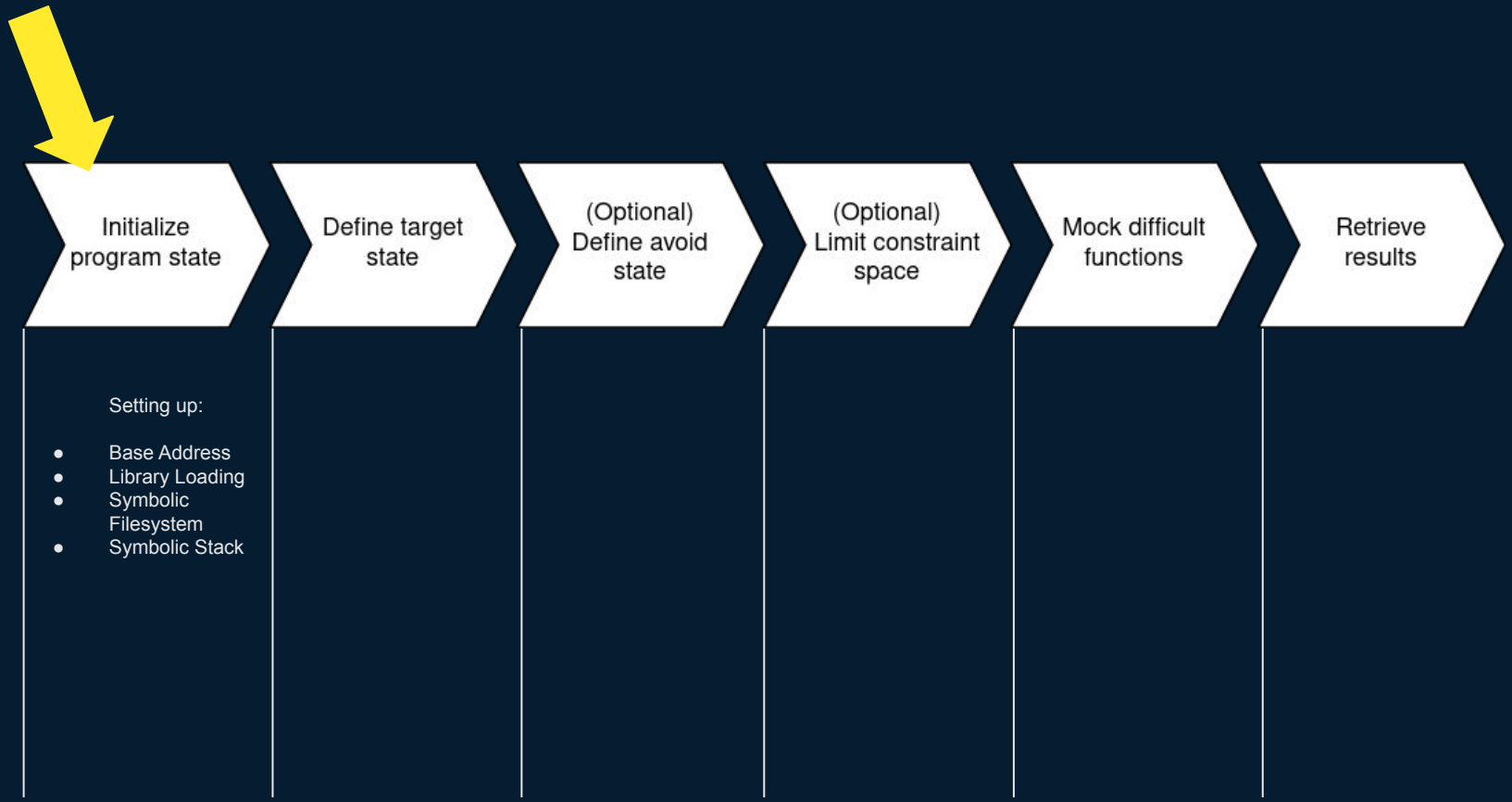
Used for reversing,
rop-chain building,
fuzzing and more

Let's recap for a second









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

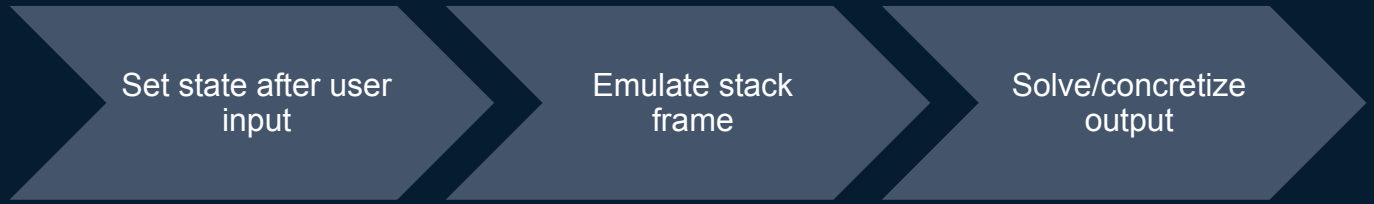
- Set start address after input was provided
- Initialize stack frame
- Define password bitvectors
- Align stack pointer
- Push password bitvectors to stack
- Solve bitvectors

```
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))
```



Symbolic Filesystem

Provide input via file
Validate input (constraint check function)
Print result

```c
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

Set start address after input

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

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

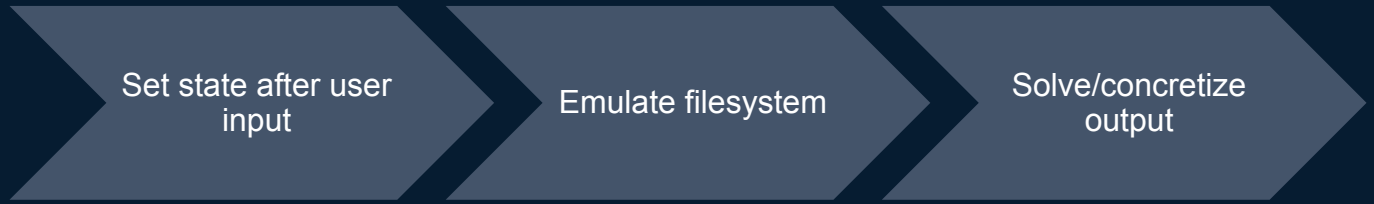
- Set start address after input
- Define symbolic memory
- Add password bitvector to symbolic memory
- Add a symbolic file (SimFile) based on symbolic memory
- Define symbolic filesystem and add SimFile
- Solve symbolic memory

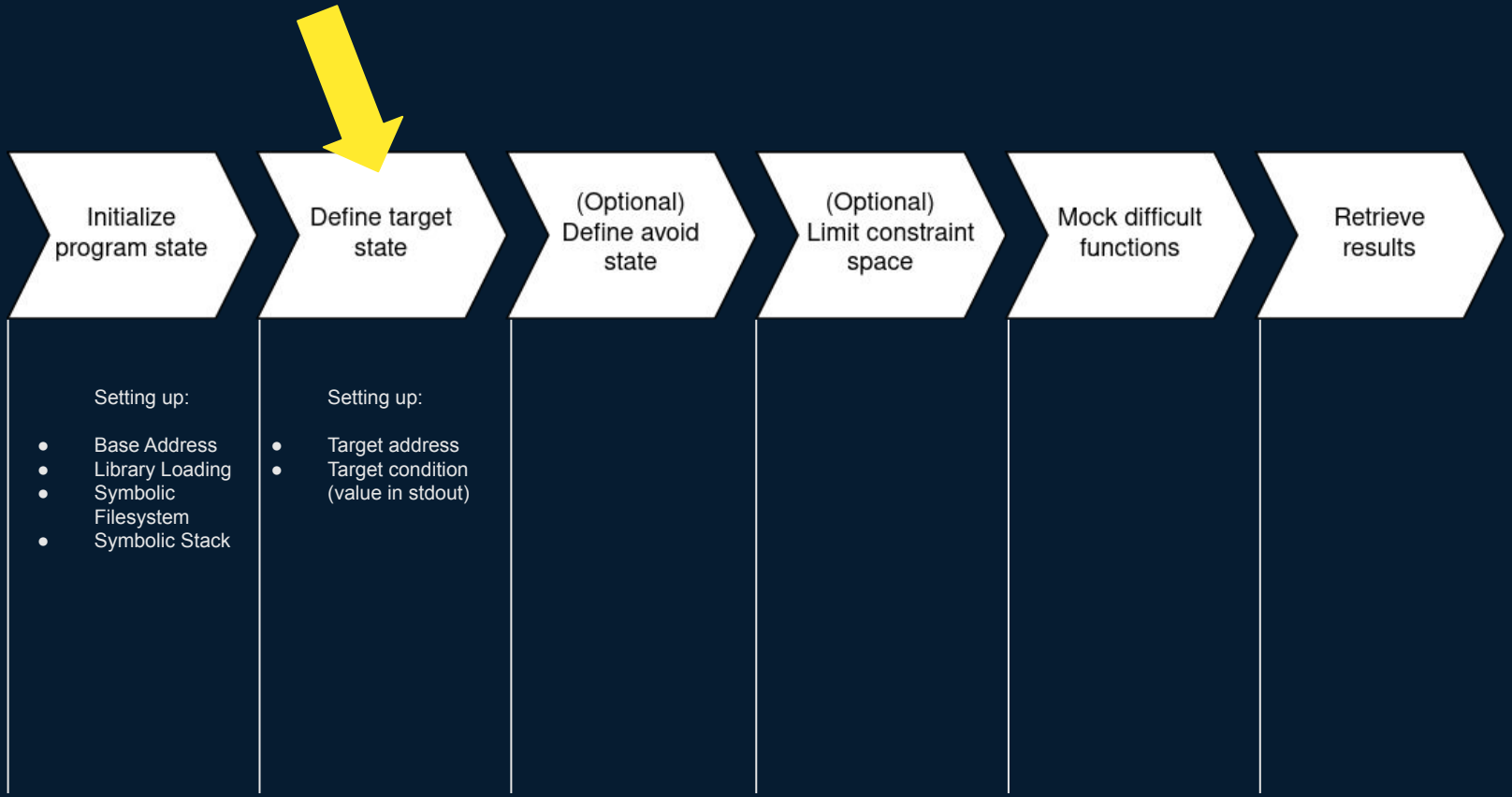
```
start_addr = 0x000013cc
initial_state = proj.factory.blank_state(addr=start_addr)

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
)
```

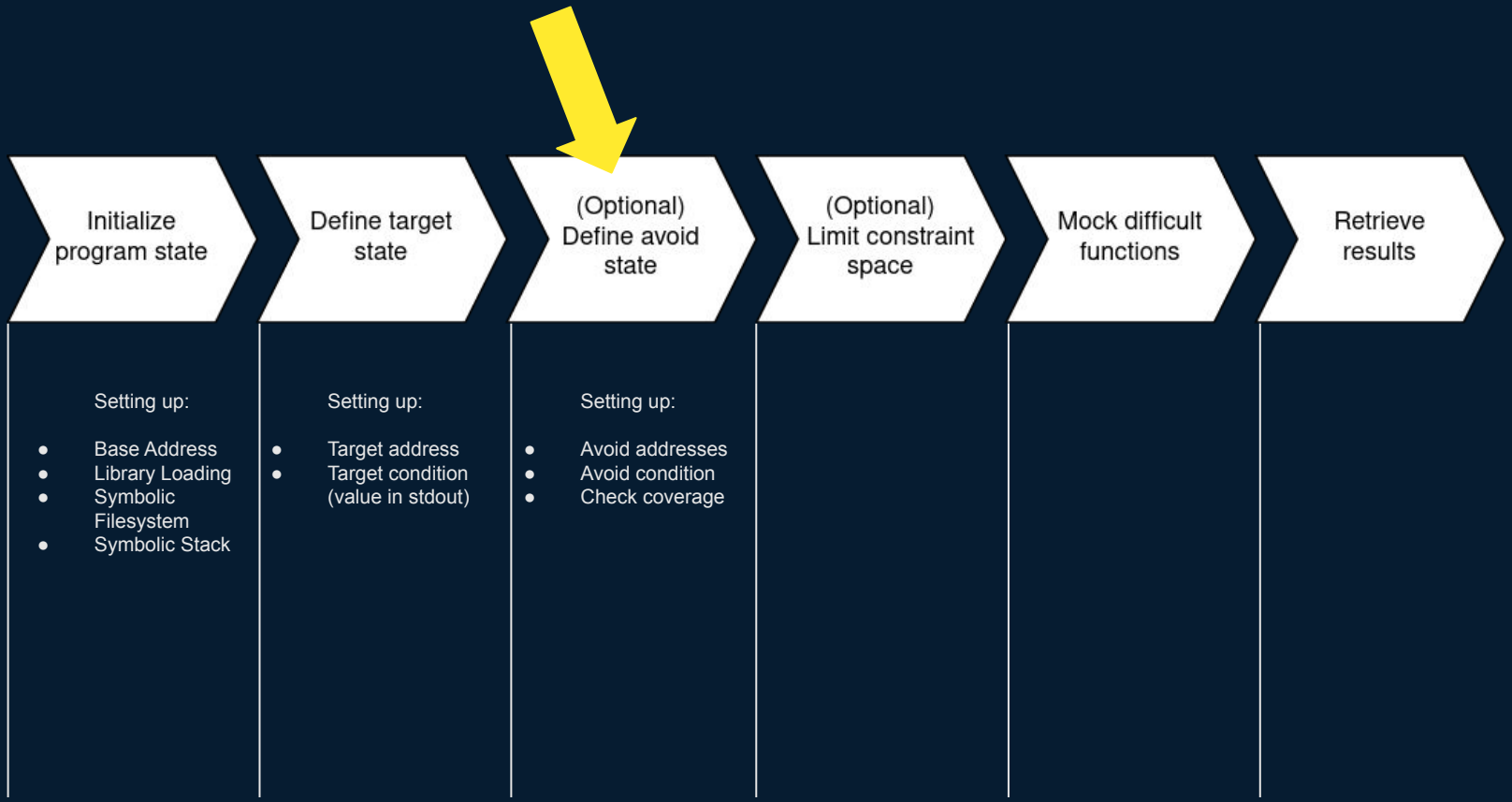
Can also be value

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

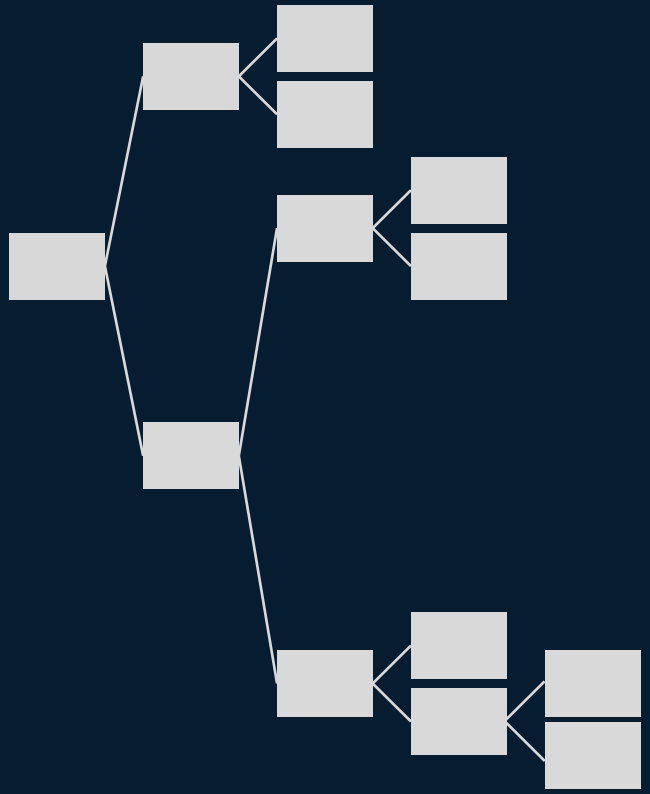
```
simgr.explore(find=0x00001407, avoid=[0x0000142d])
```

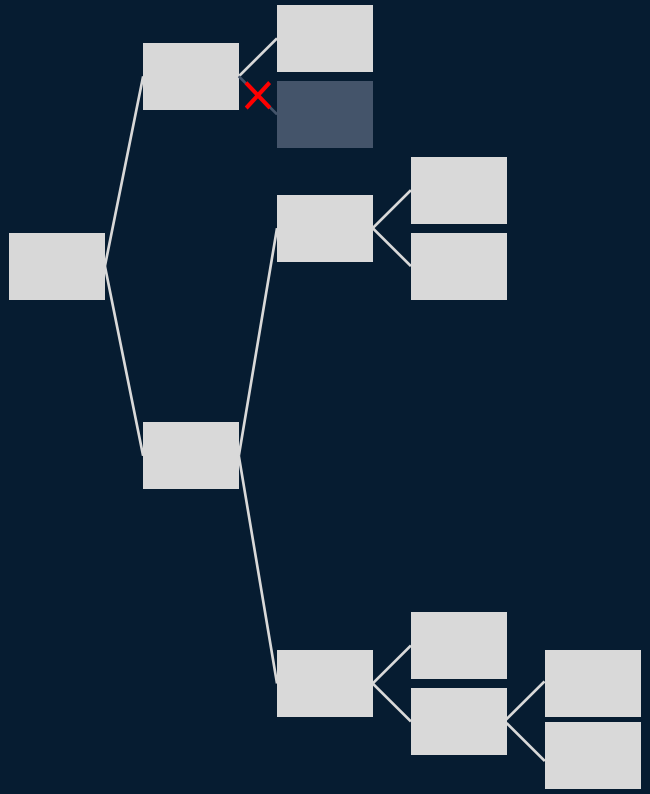
Just exclude, it’s easy

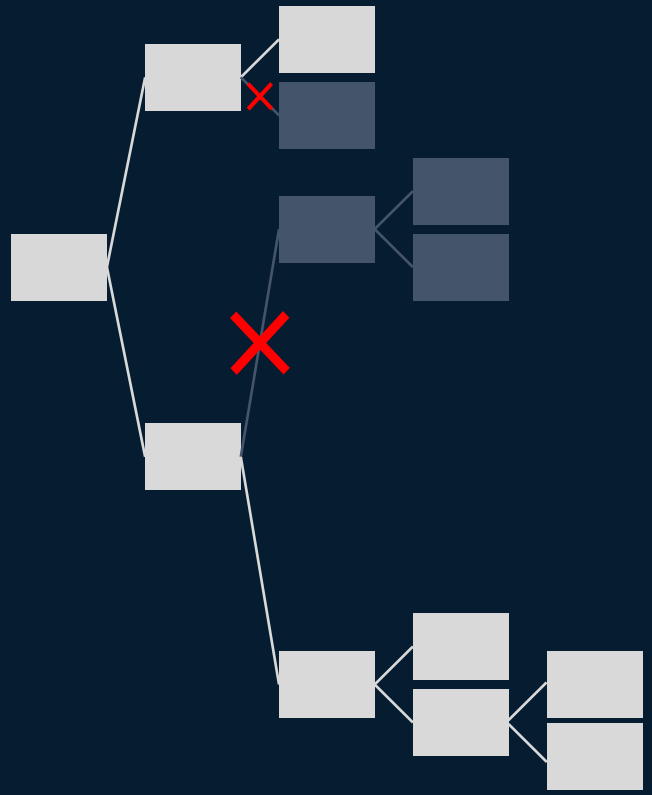
A great way to reduce complexity is by entirely avoiding unneeded paths

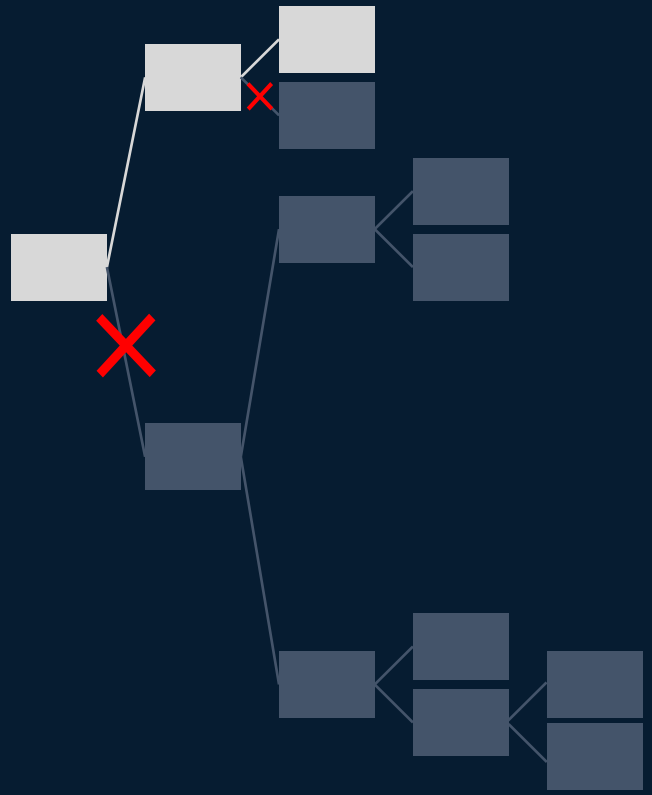
Selecting those paths works best with reverse engineering & human intuition

```
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))
```









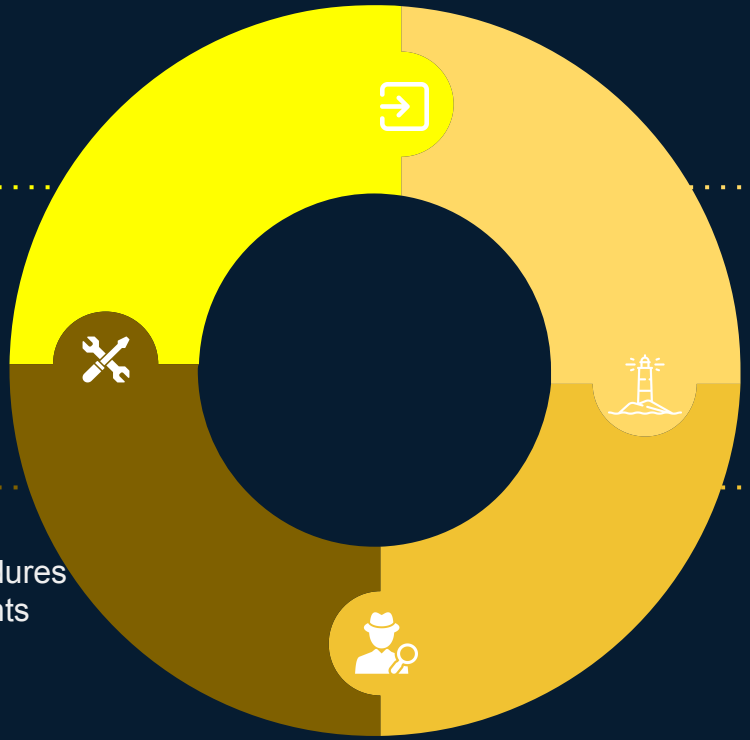
Code Coverage Collection Process

Retrieve Coverage

Export basic block coverage from angr exploration

Load into Lighthouse

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



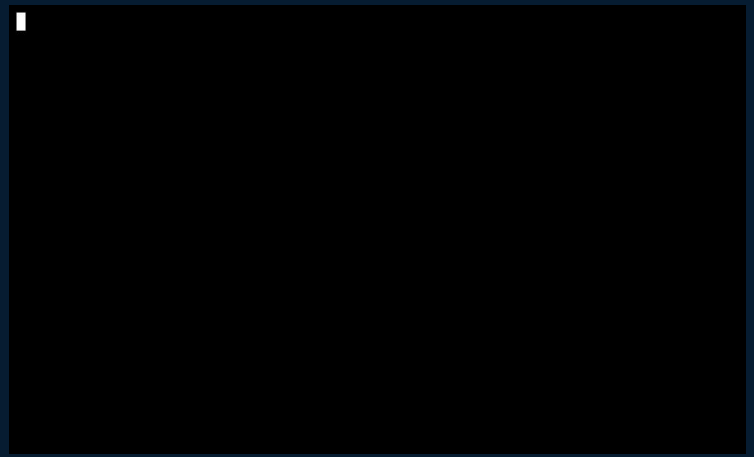
Investigate Bottlenecks

Disassemble/Decompile to discover bottlenecks

Fix Bottlenecks

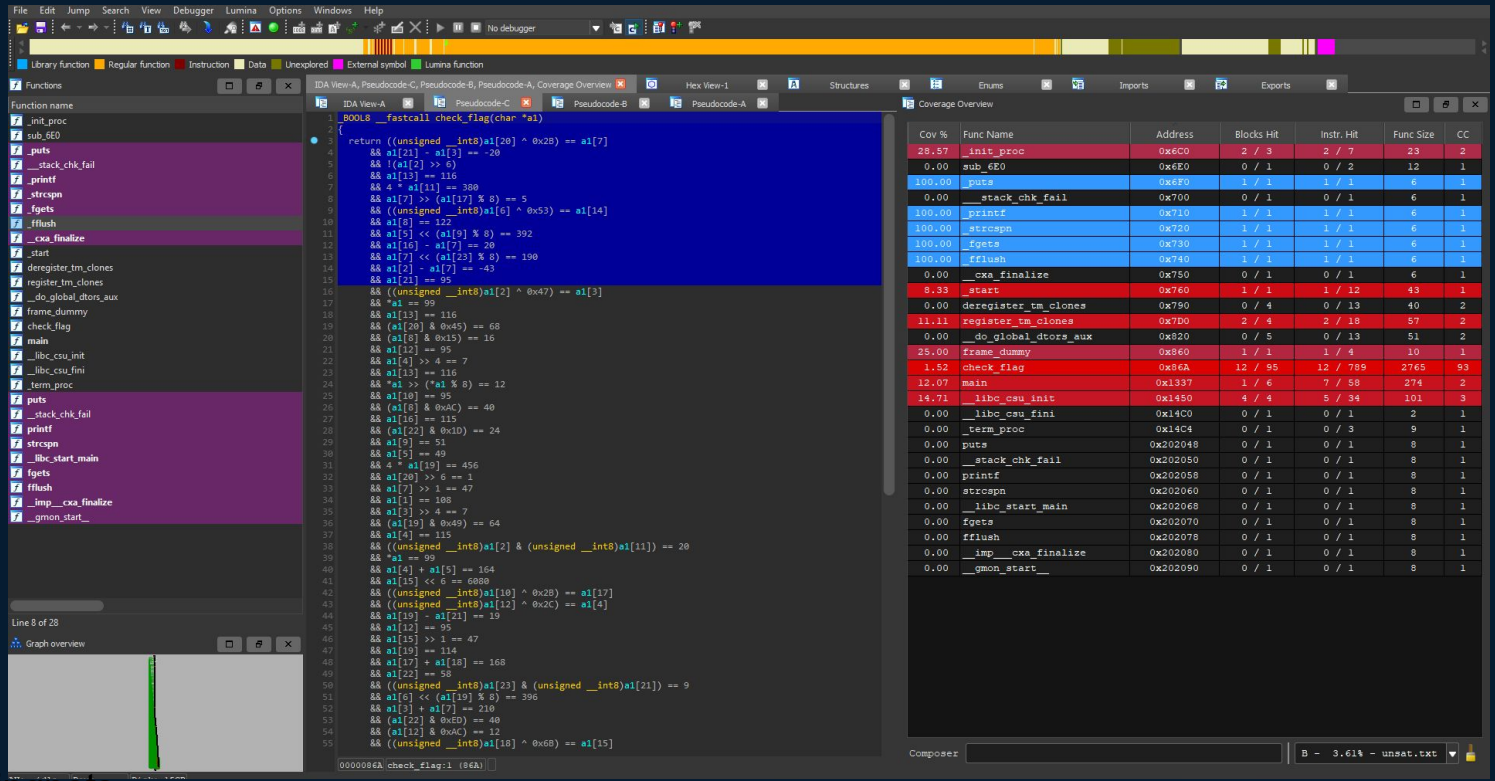
- Avoid bottleneck code
- Use Hooks & SimProcedures
- Manually apply constraints

Code Coverage

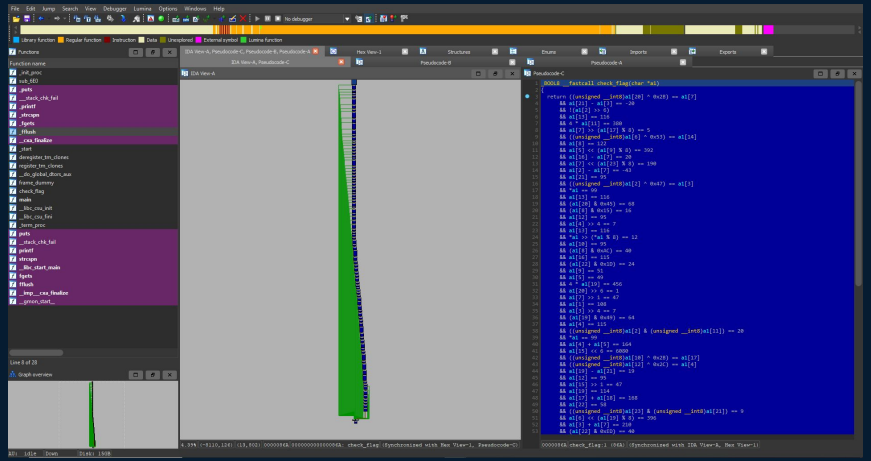


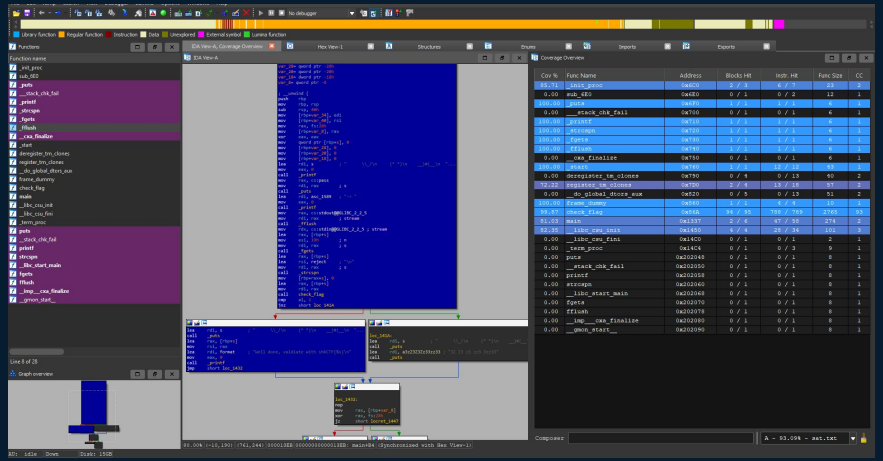
```
def get_small_coverage(*args, **kwargs):
    sm = args[0]
    stashes = sm.stashes
    i = 0
    for simstate in stashes["active"]:
            state_history = ""

            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)
```

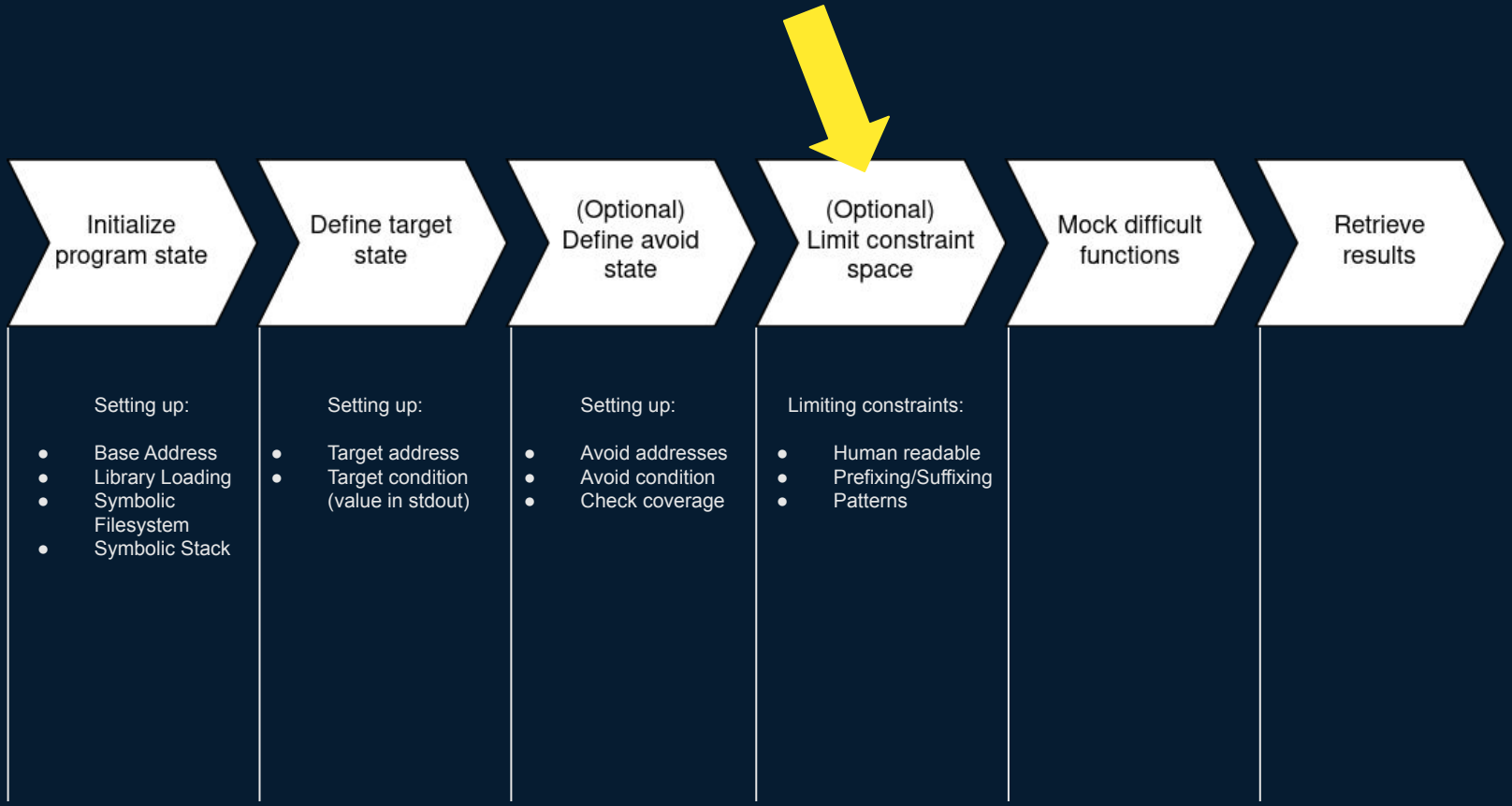
Load into lighthouse to find bottlenecks…



…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

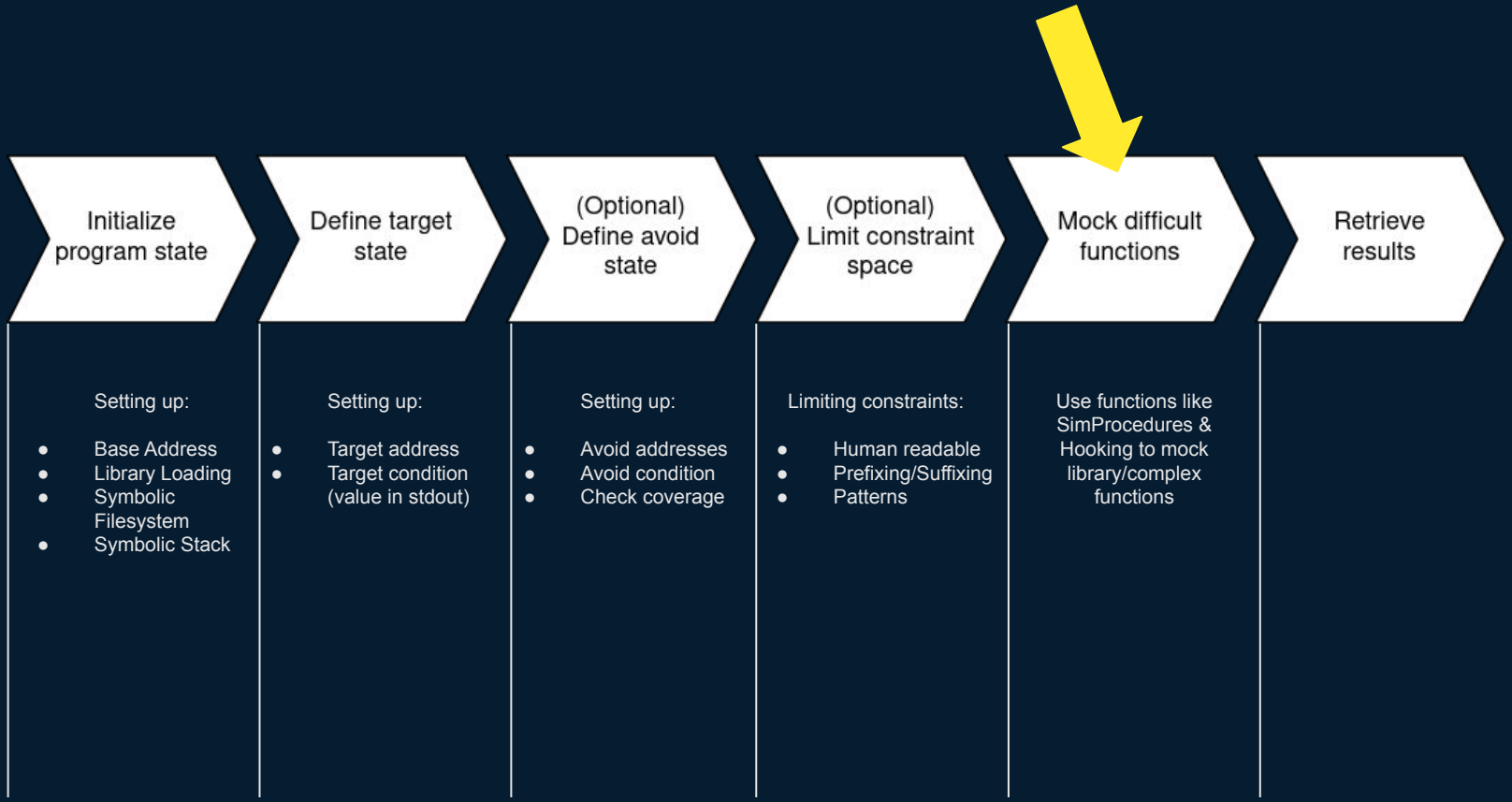
Create new symbolic password bitvector

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

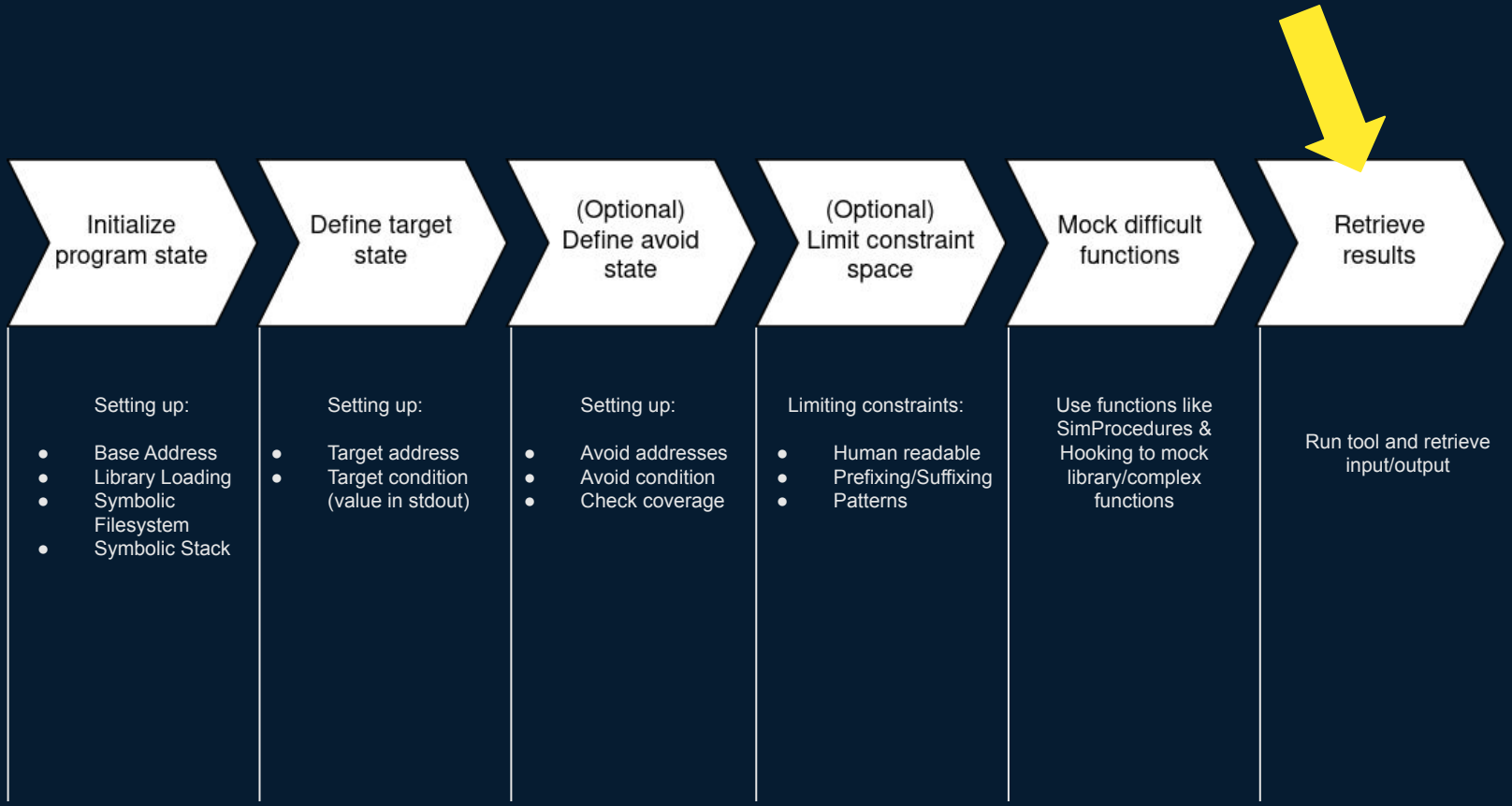
```
# length determines how many bytes get skipped/overwritten
@proj.hook(0x1337, length=5)
def set_rax(state):
  state.regs.rax = 1
```

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)
```

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(inital_state, veritesting=True)
```

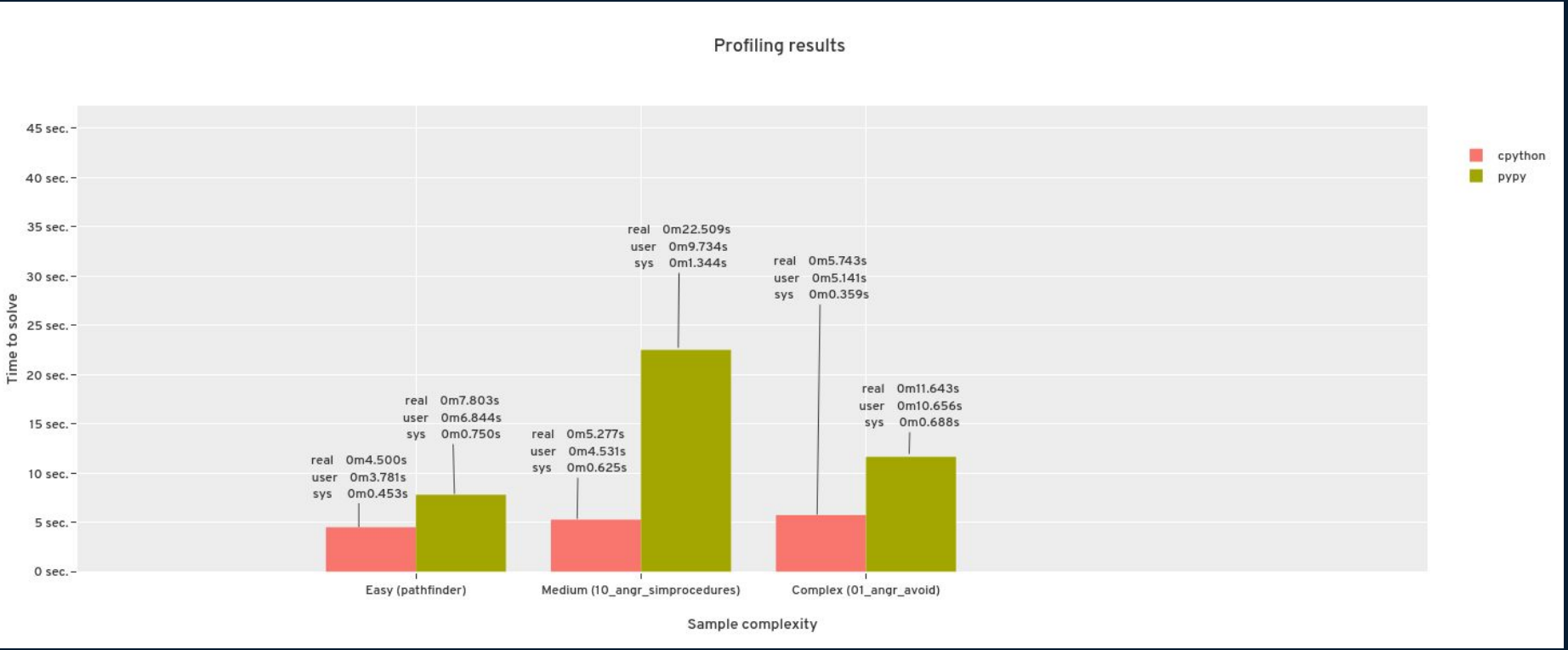
Veritesting

Algorithm to automatically reduce state explosions

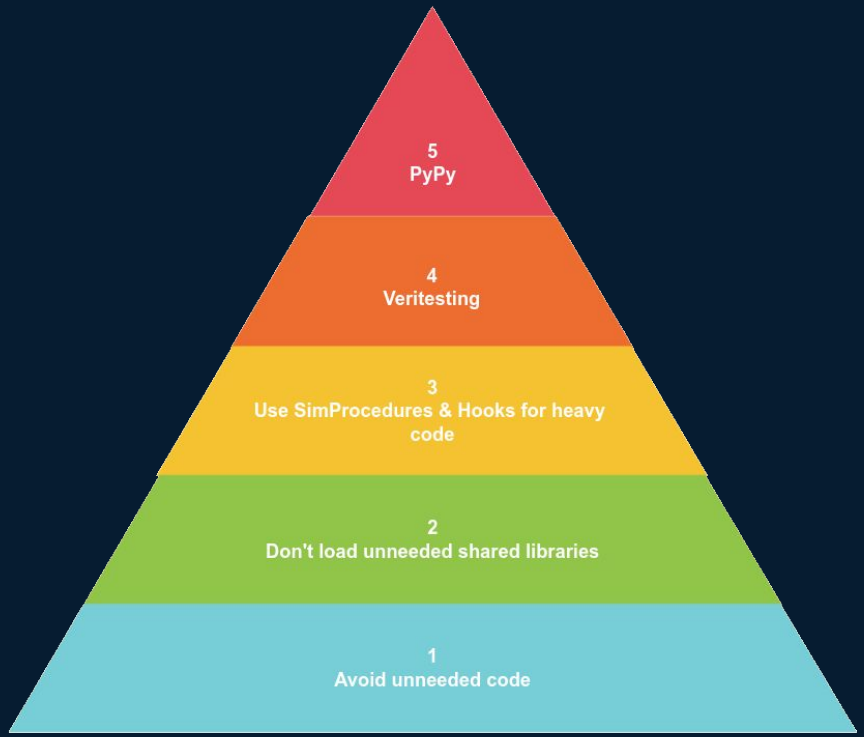
Relies on heuristics to merge states

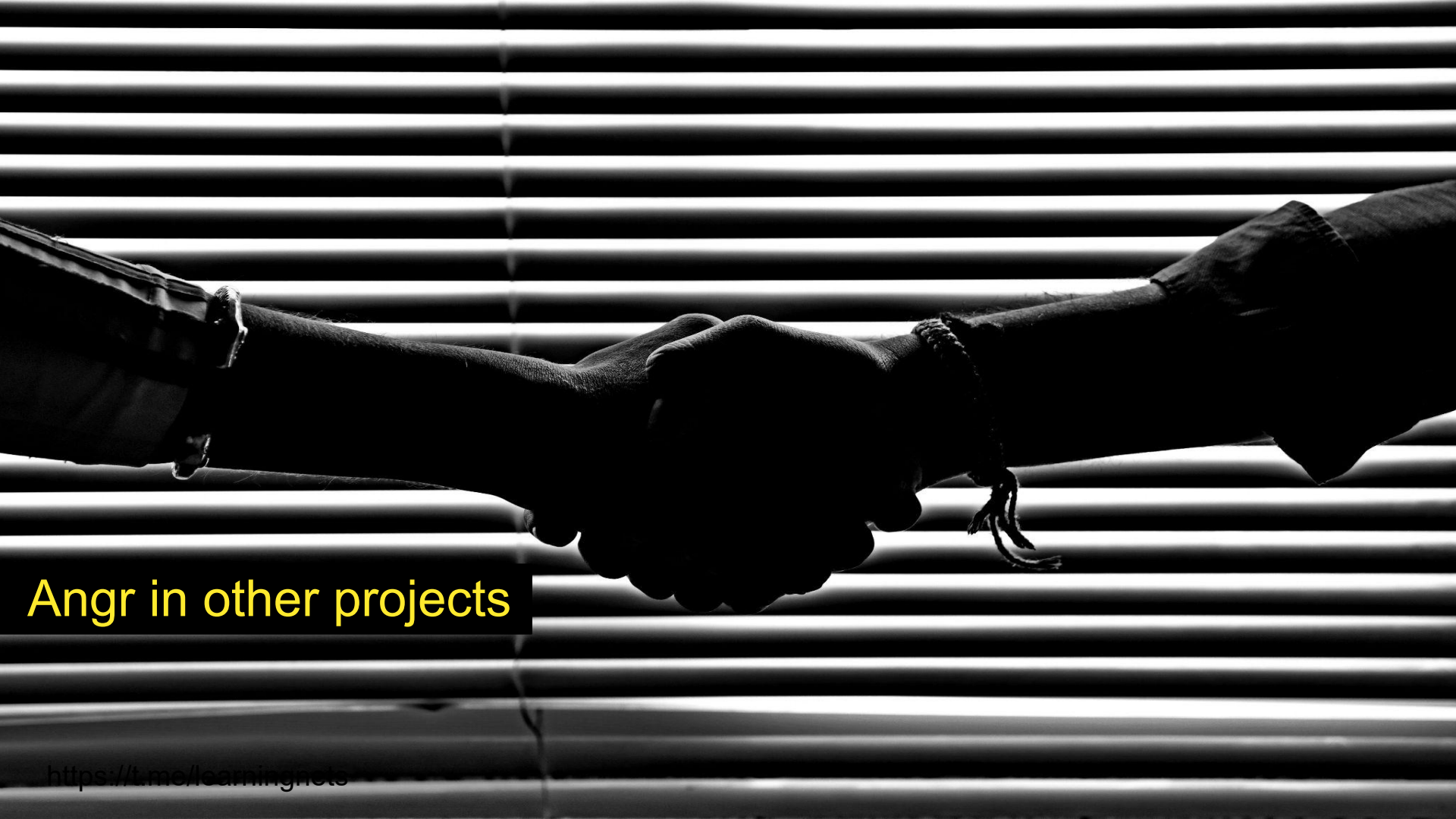
```
import angr, claripy

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(inital_state, veritesting=True)

simgr.explore(find=0x00001407)

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

PyPy



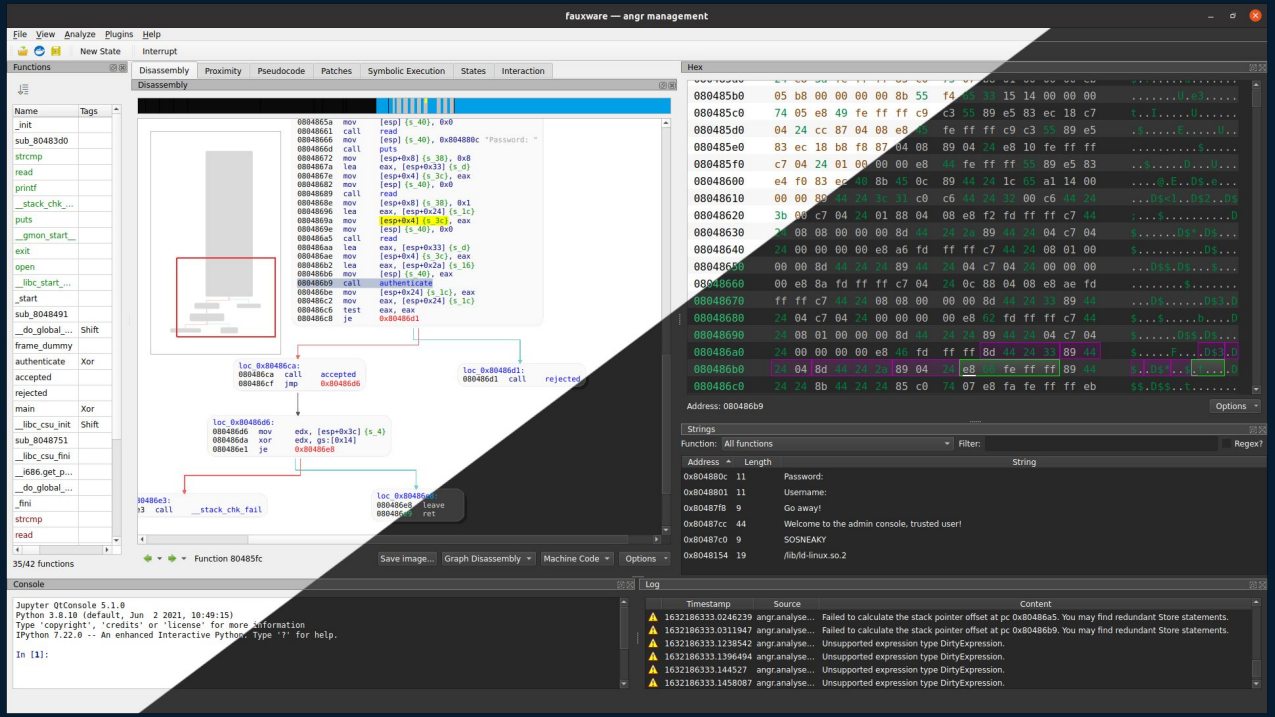
Maslow’s Hierarchy of Symbolic Execution





Angr-Management

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

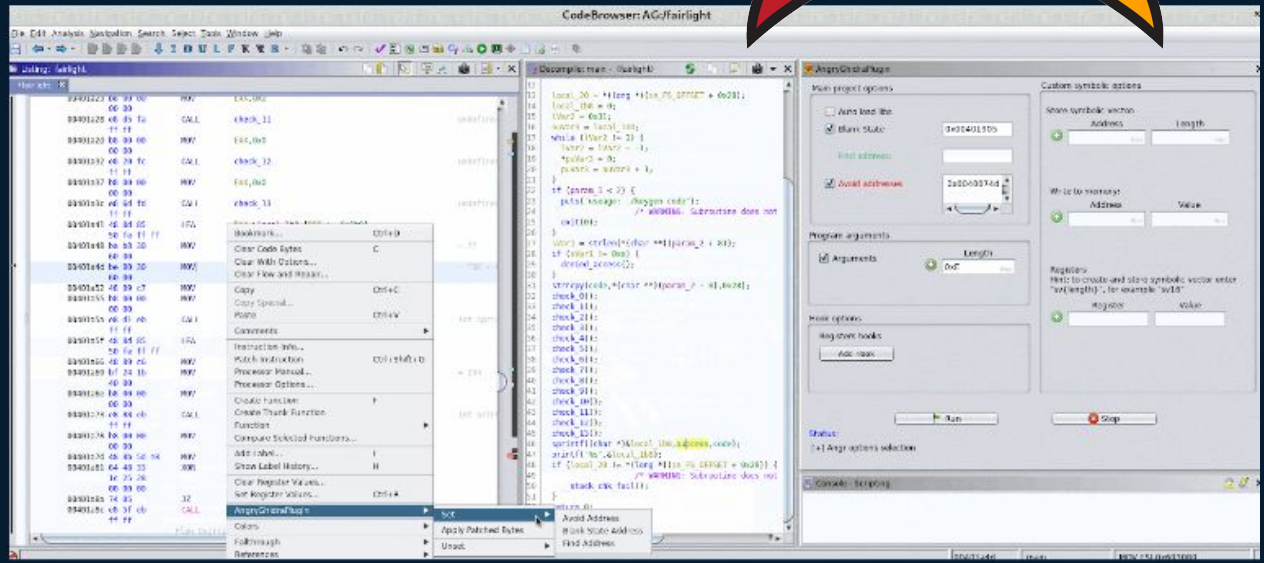


https://t.me/learningnets

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

AngryGhidra

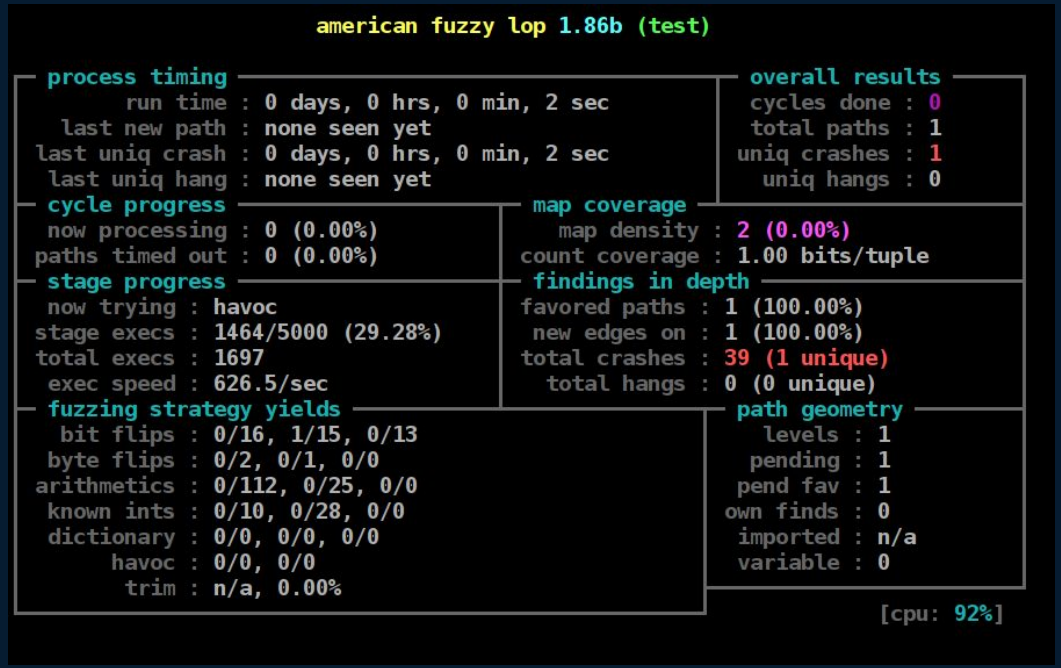
A plugin that combines the convenience of ghidra with the power of the angr framework



https://github.com/Nalen98/AngryGhidra

Driller

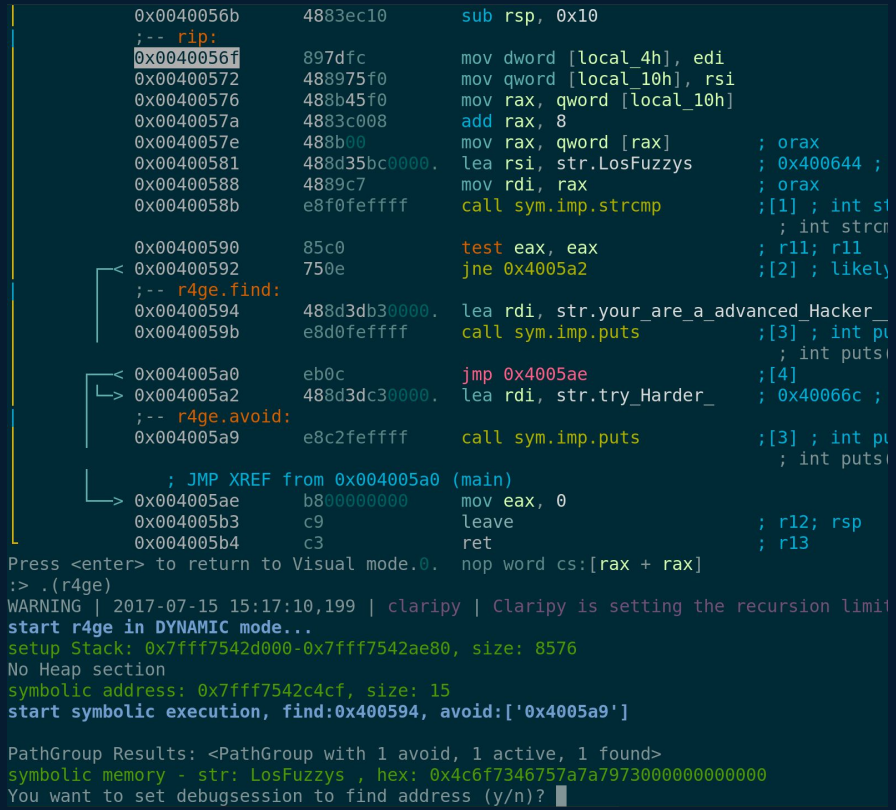
Augments the afl-fuzz capabilities with symbolic execution to discover new, interesting paths



https://github.com/shellphish/driller

r4ge

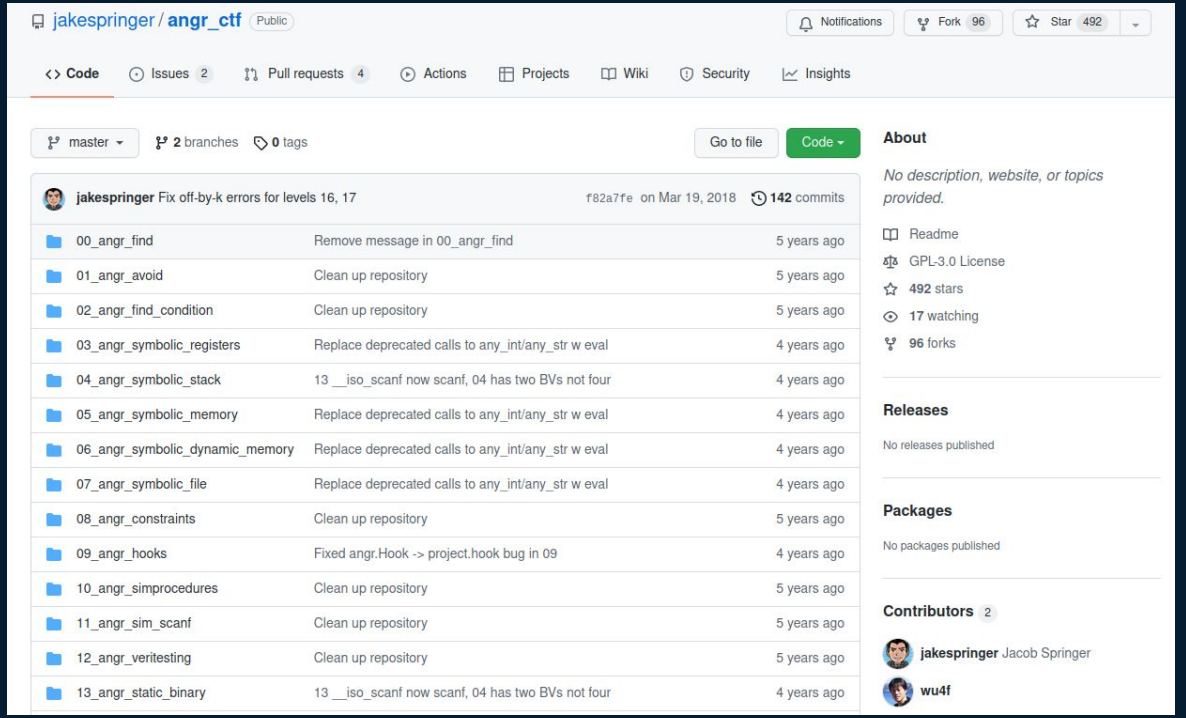
We all like radare2/rizin, now you can use angr functionalities straight from your favorite reverse engineering framework



https://github.com/gast04/r4ge

Further learning

https://github.com/jakespringer/angr_ctf



Let’s solve a crackme!

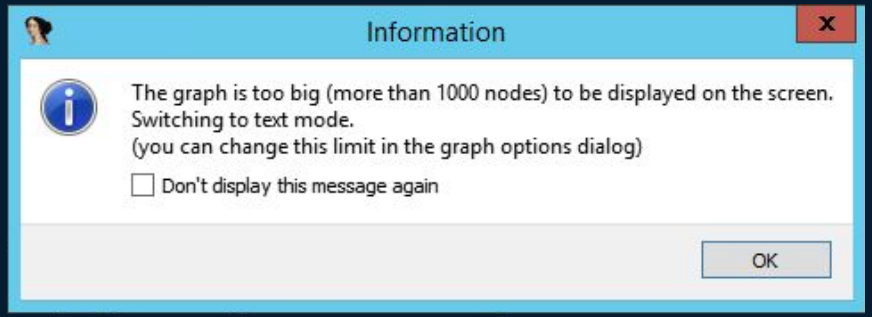
Insomni’Hack 2022
Jannis Kirschner

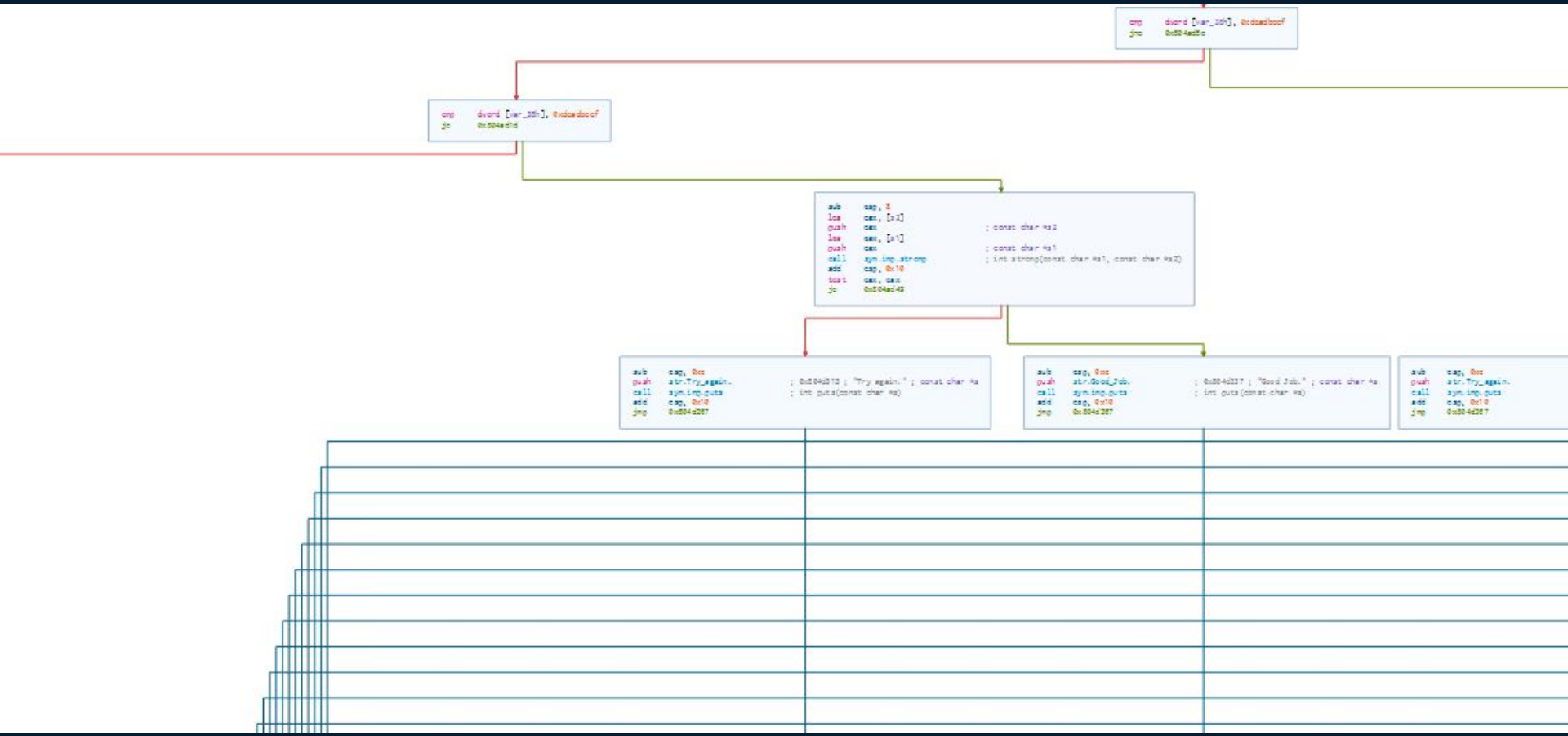
25.03.2022

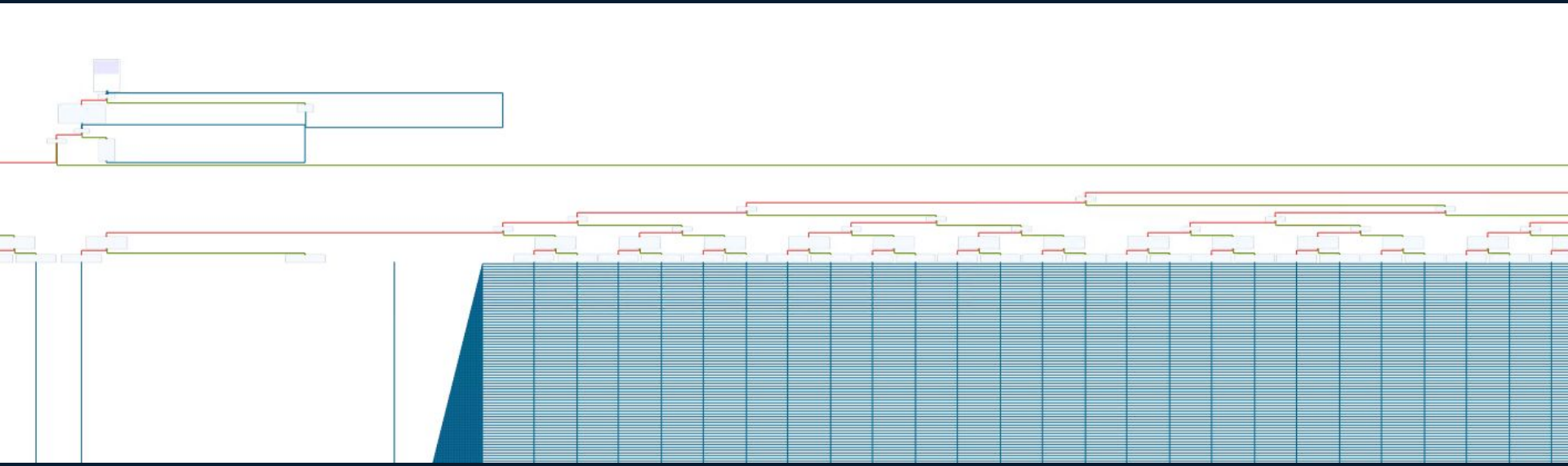
Let’s solve a crackme! (Again)

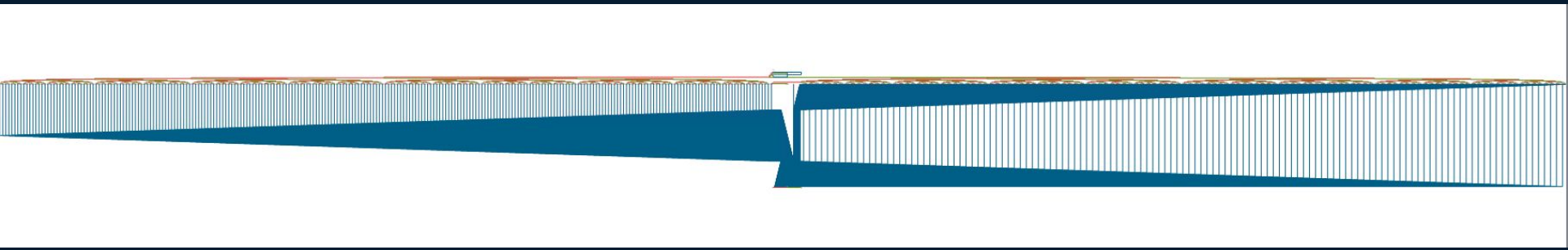
Insomni’Hack 2022
Jannis Kirschner

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

→ Work through angr_ctf

→ Pwn all the CTFs!!!

@xorkiwi

/in/janniskirschner