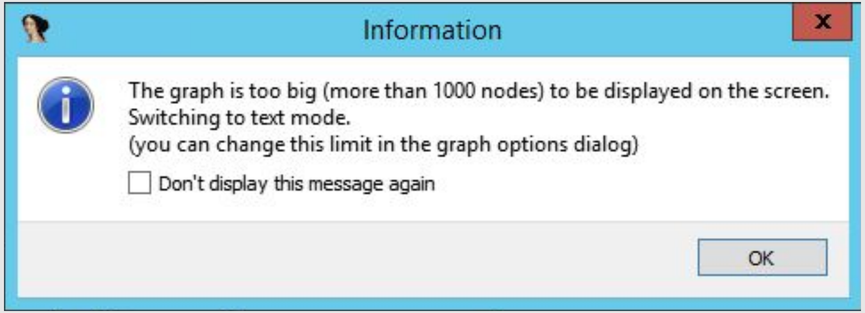
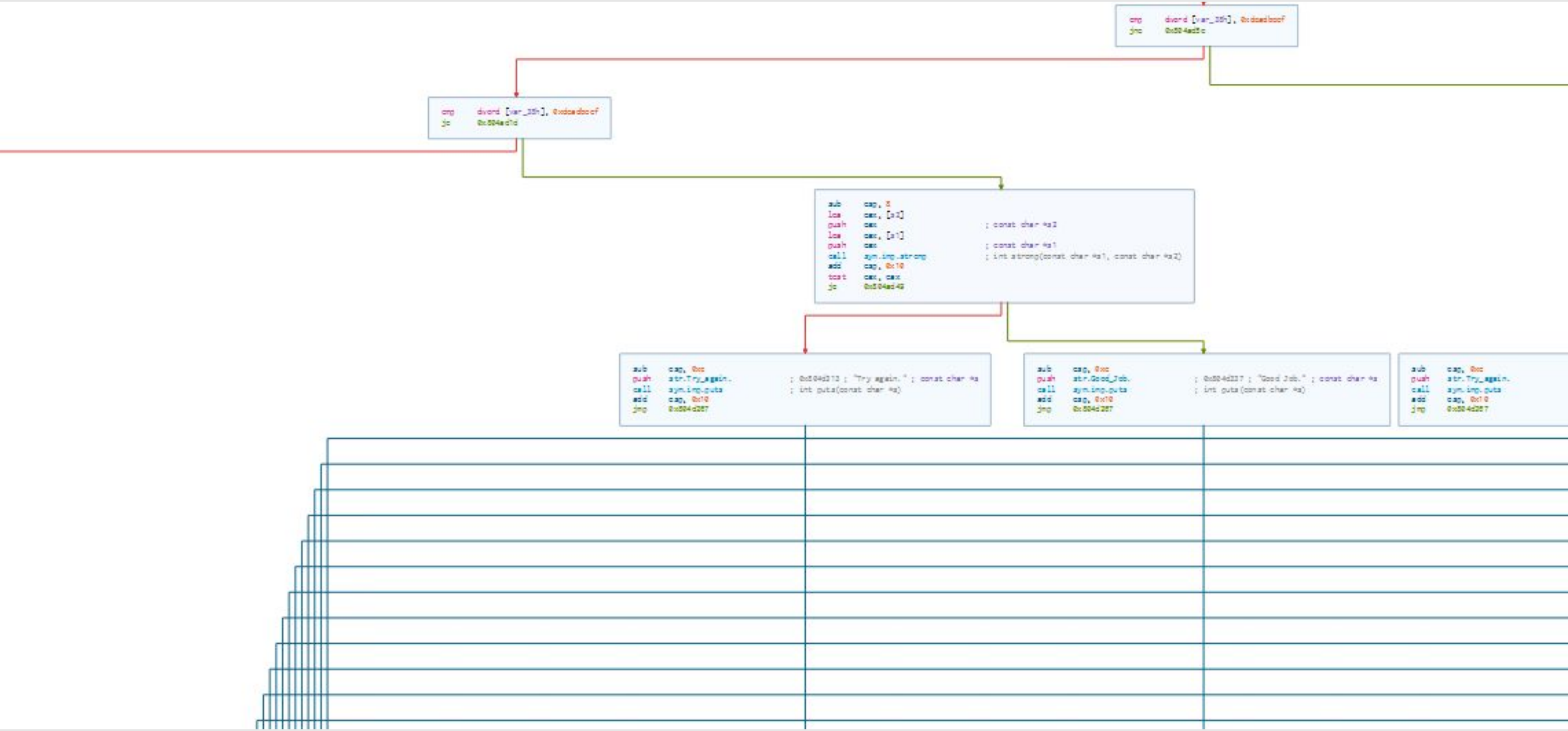
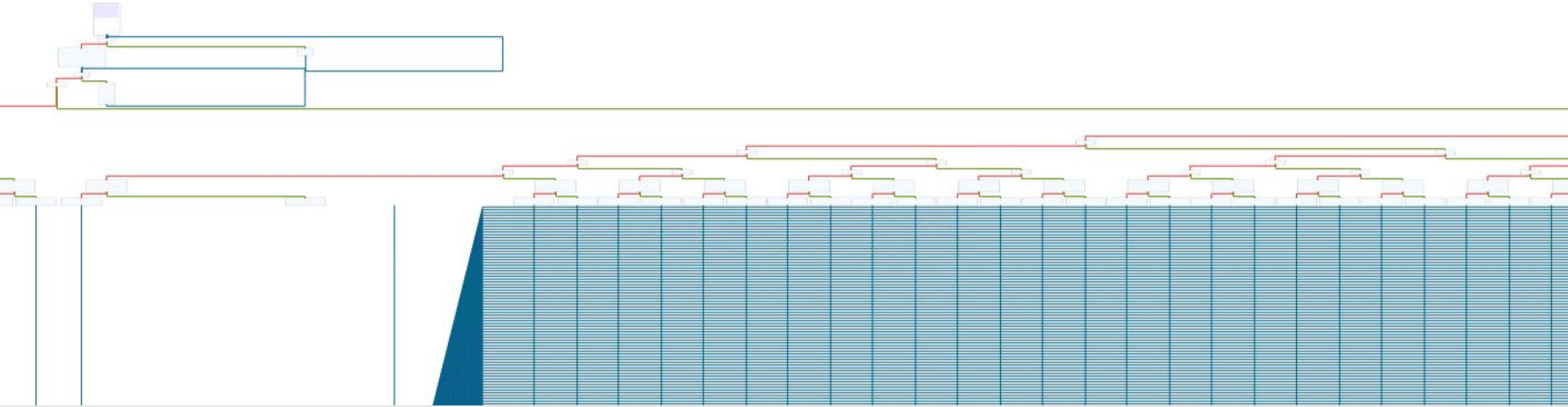
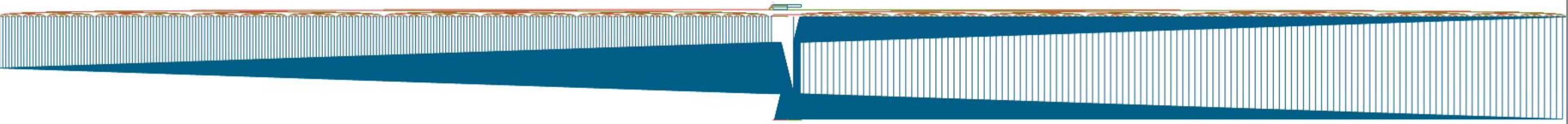

Information
The graph is too big (more than 1000 nodes) to be displayed on the screen.
Switching to text mode.
(you can change this limit in the graph options dialog)
Don't display this message again
OK



cmp dword [var_38h], 0xdeadbeef
sub esp, 8
push eax ; const char *s2
push eax ; const char *s1
call sym.imp.strcmp
add esp, 0x10
test eax, eax
sub esp, 0xc
push str.Try_again. ; "Try again." ; const char *s
call sym.imp.puts ; int puts(const char *s)
add esp, 0x10
push str.Good_Job. ; "Good Job." ; const char *s
call sym.imp.puts ; int puts(const char *s)
add esp, 0x10
push str.Try_again.
call sym.imp.puts
add esp, 0x10






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

Reverse engineering of **black-box binaries** with **symbolic** and **concolic execution techniques**

or

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

REcon Montreal 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 devices*** with my research team ***suid.ch***

Views are my own and not related to my employer

@xorkiwi

/in/janniskirschner

Who are you?

Example: z3_robot (SharkyCTF2020)

https://ctftime.org/event/1034

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

Static Analysis

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

x86_64 ELF Binary

Not Stripped

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

Trying to bruteforce

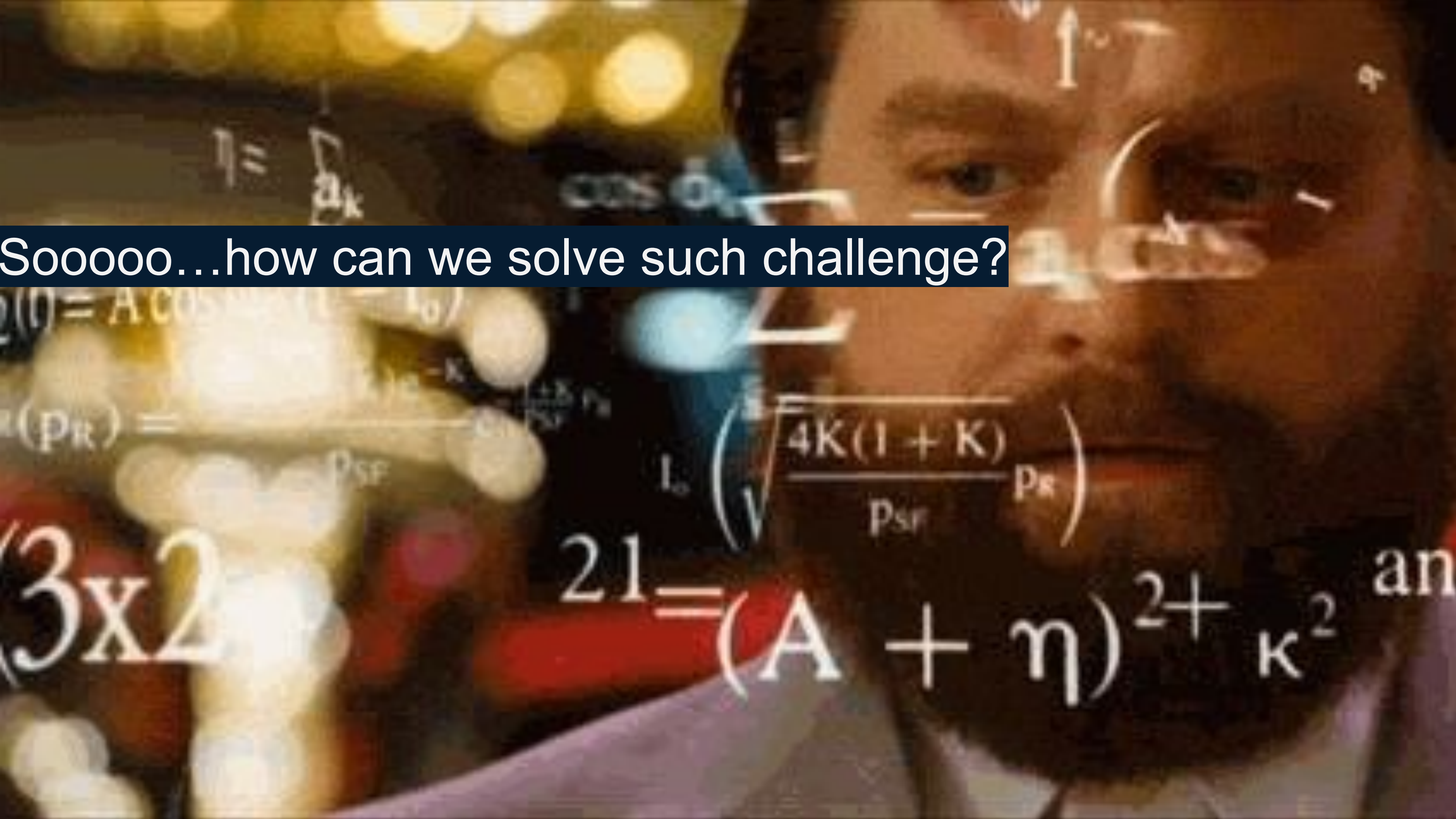
```
sym.imp.fgets((int64_t)&var_34h + 4, 0x19, _reloc.stdin);
```

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

Binary asks for a 24 characters long passphrase

Brute-forcing it would be infeasible!

Sooooo…how can we solve such challenge?



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

All constraints extracted from decompiled pseudocode

cleaned.txt
~/Insomnihack/00_z3

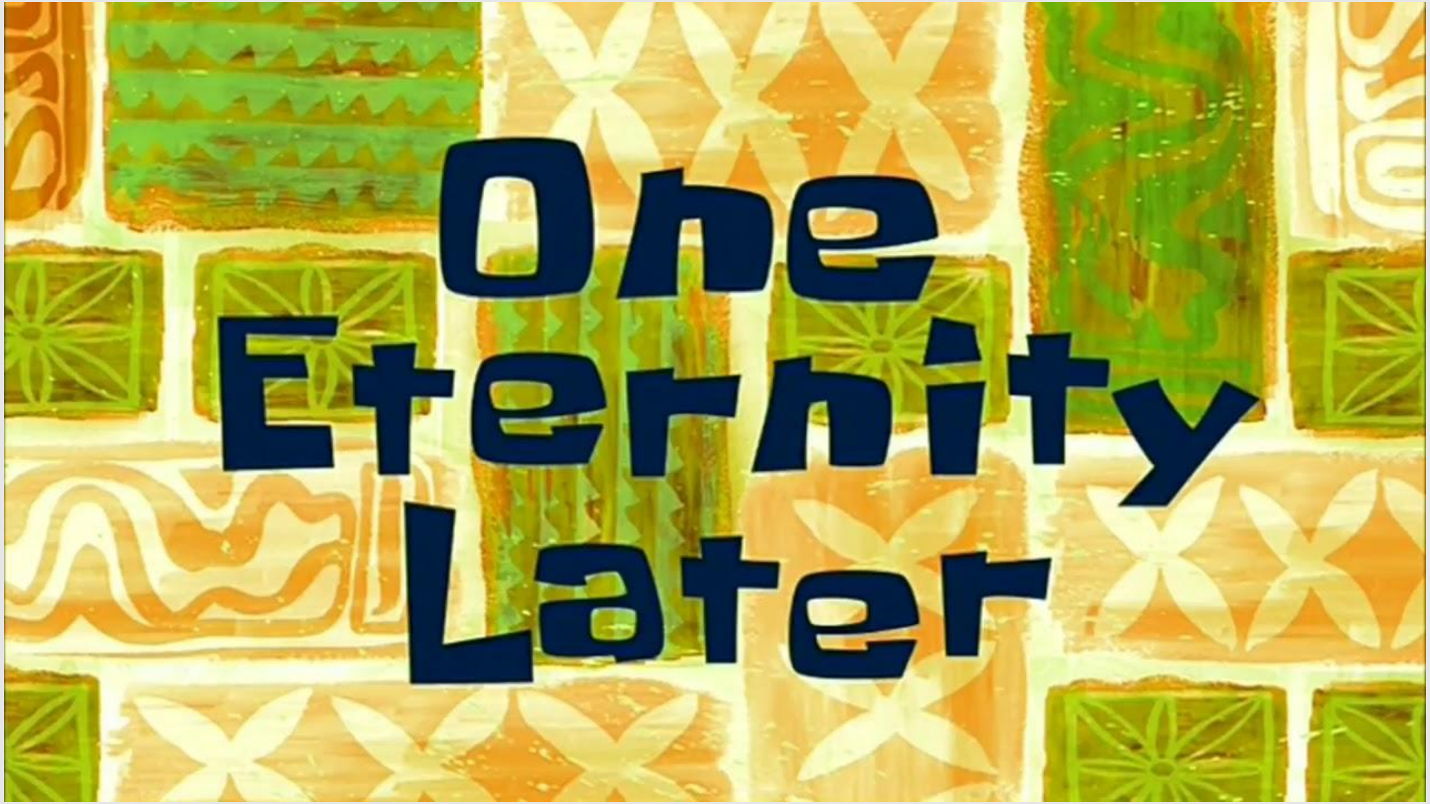
```
int check_flag(byte *param_1)

{
  return
  (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 &&
  bVar2 = (param_1[0x11] >> 7) >> 5,
  param_1[7] >> ((param_1[0x11] + bVar2 & 7) - bVar2 & 0x1f) == 5 &&
  (param_1[6] ^ 0x53) == param_1[0xe] &&
  param_1[8] == 0x7a &&
  bVar2 = (param_1[9] >> 7) >> 5,
  param_1[5] << ((param_1[9] + bVar2 & 7) - bVar2 & 0x1f) == 0x188 &&
  param_1[0x10] - param_1[7] == 0x14 &&
  bVar2 = (param_1[0x17] >> 7) >> 5,
  param_1[7] << ((param_1[0x17] + bVar2 & 7) - bVar2 & 0x1f) == 0xbe &&
  param_1[2] - param_1[7] == -0x2b &&
  param_1[0x15] == 0x5f &&
  (param_1[2] ^ 0x47) == param_1[3] &&
  *param_1 == 99 &&
  param_1[0xd] == 0x74 &&
  (param_1[0x14] & 0x45) == 0x44 &&
  (param_1[8] & 0x15) == 0x10 &&
  param_1[0xc] == 0x5f &&
  param_1[4] >> 4 == '\a' &&
  param_1[0xd] == 0x74 &&
  bVar2 = (*param_1 >> 7) >> 5, *param_1 >> ((*param_1 + bVar2 & 7) -
bVar2 & 0x1f) == 0xc &&
  param_1[10] == 0x5f &&
  (param_1[8] & 0xacU) == 0x28 &&
  param_1[0x10] == 0x73 &&
  (param_1[0x16] & 0x1d) == 0x18 &&
  param_1[9] == 0x33 &&
  param_1[5] == 0x31 &&
  (param_1[0x13] & 0x3fffffffU) == 0x72 &&
  param_1[0x14] >> 6 == '\x01' &&
  param_1[7] >> 1 == '/' &&
```

Plain Text Tab Width: 8 Ln 1, Col 1 INS

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**

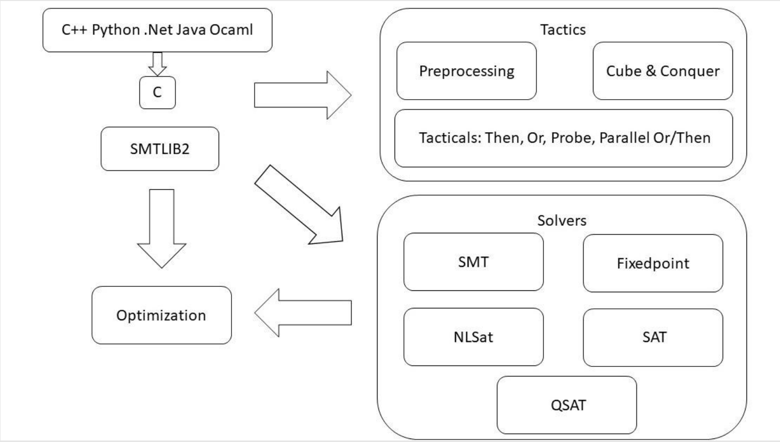

One
Eternity
Later

Overview over z3

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

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

SMT solving builds the bases for most modern symbolic execution frameworks

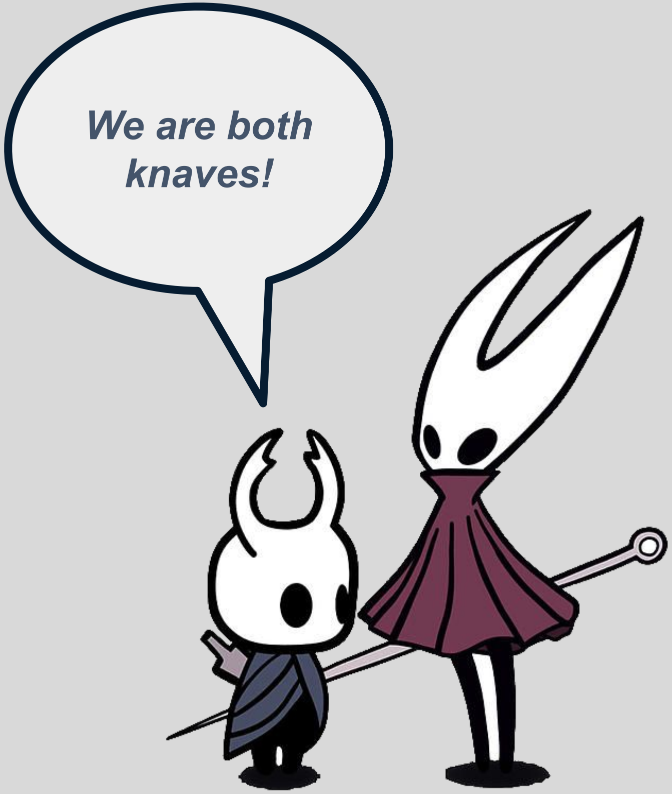


Architecture diagram of z3

A logic puzzle

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

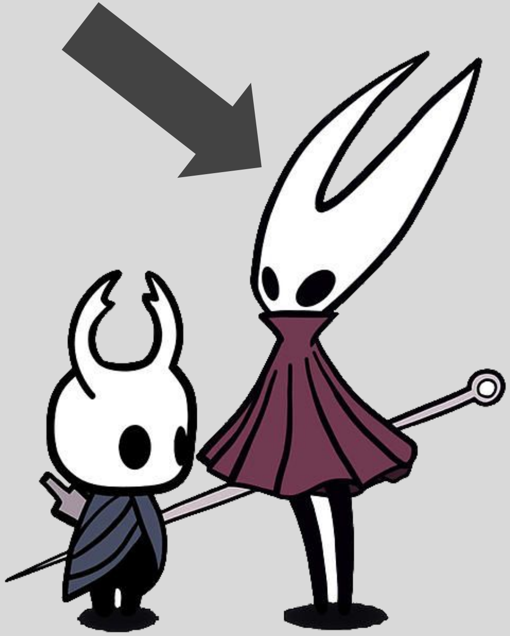
*Two people stand in front of you, Red and Blue. Blue tells you "**we are both knaves**"...who is the knight?*



A logic puzzle

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

Our Knight



SAT/SMT solving

We can ask them questions like:

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

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

SAT/SMT solving

We can ask them questions like:

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

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

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

SAT/SMT solving

We can ask them questions like:

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

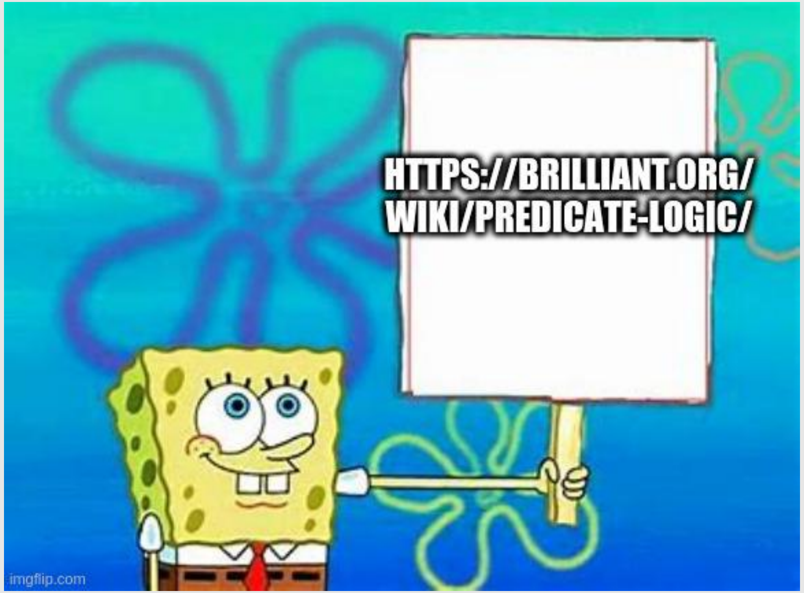
(a and not b) or (not a and c)

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

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

SAT Solving	**SMT Solving**
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

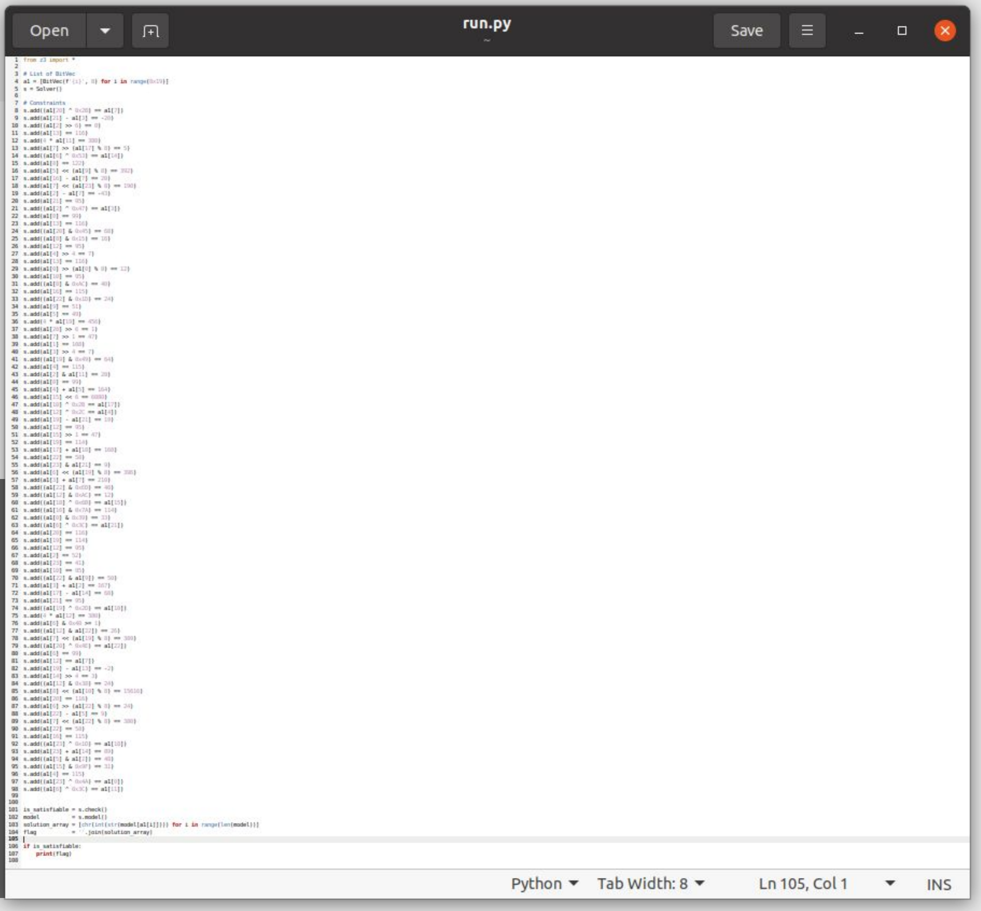
Compute model and convert solved bitvector integers to characters

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

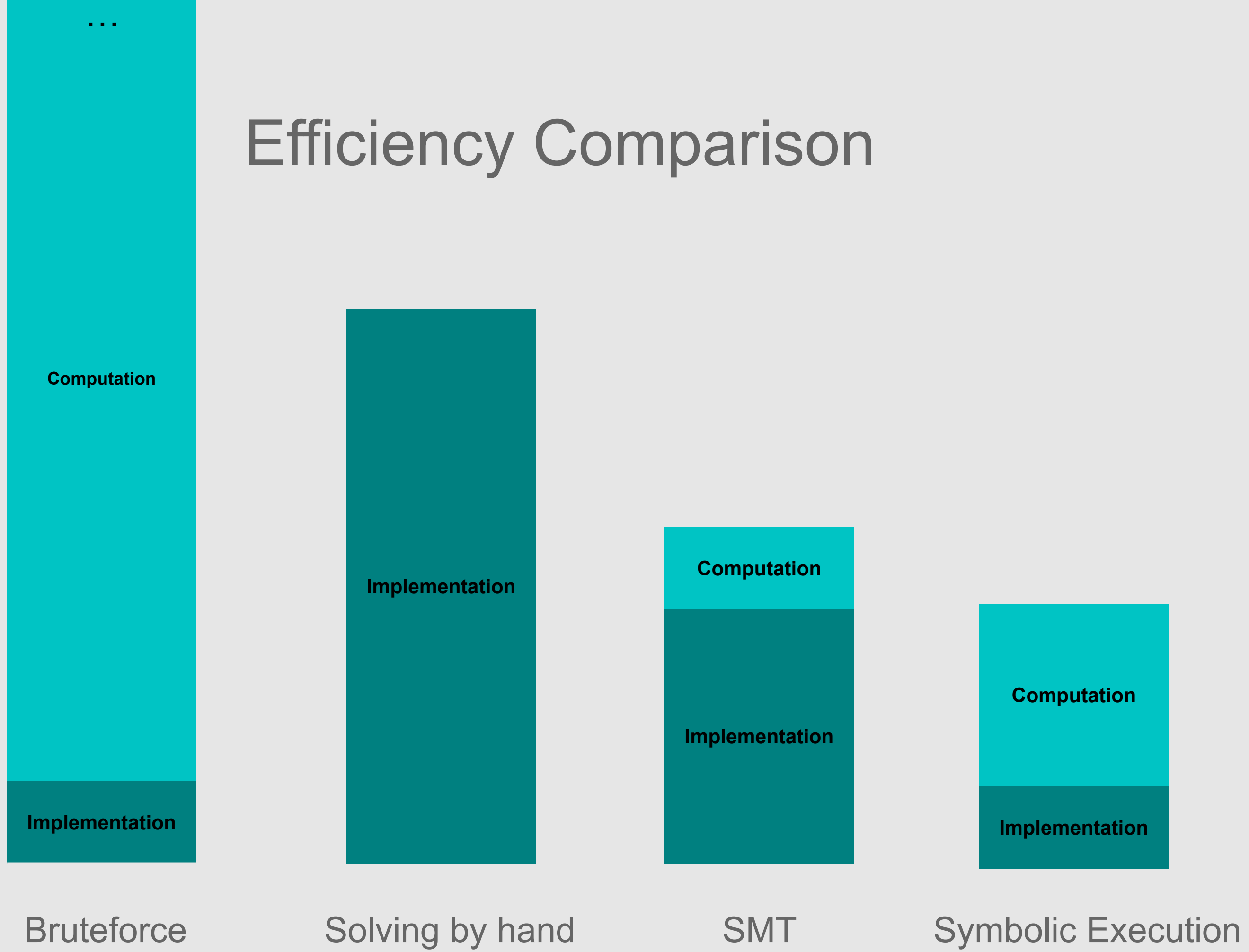
4

lines of code

kiwi@doghouse: ~/Insomnihack/00_z3

```
import angr, claripy

proj = angr.Project('./z3_robot', load_options={"auto_load_libs": False}, main_opts={"base_addr": 0})
simgr = proj.factory.simgr()
simgr.explore(find=0x00001407, avoid=0x0000142d)
print(simgr.found[0].posix.dumps(0))
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
                                                                                1,1           All
```

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 additionally automated constraint extraction with **symbolic execution**

~~Bruteforce~~

Solving by hand

SMT Solving

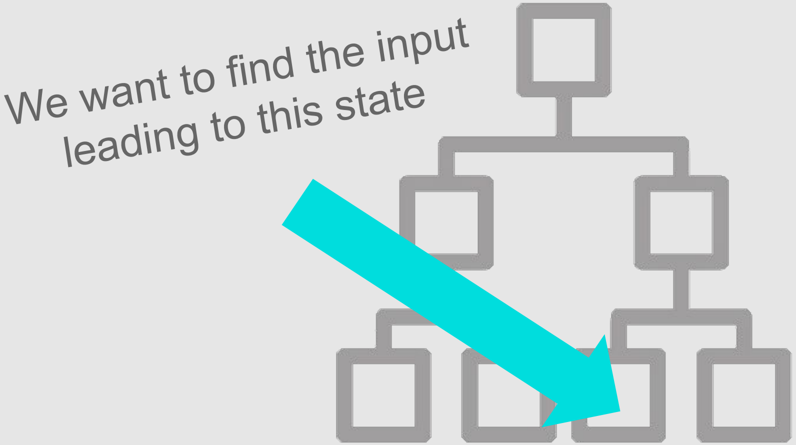
Symbolic Execution

Symbolic Execution

Introducing

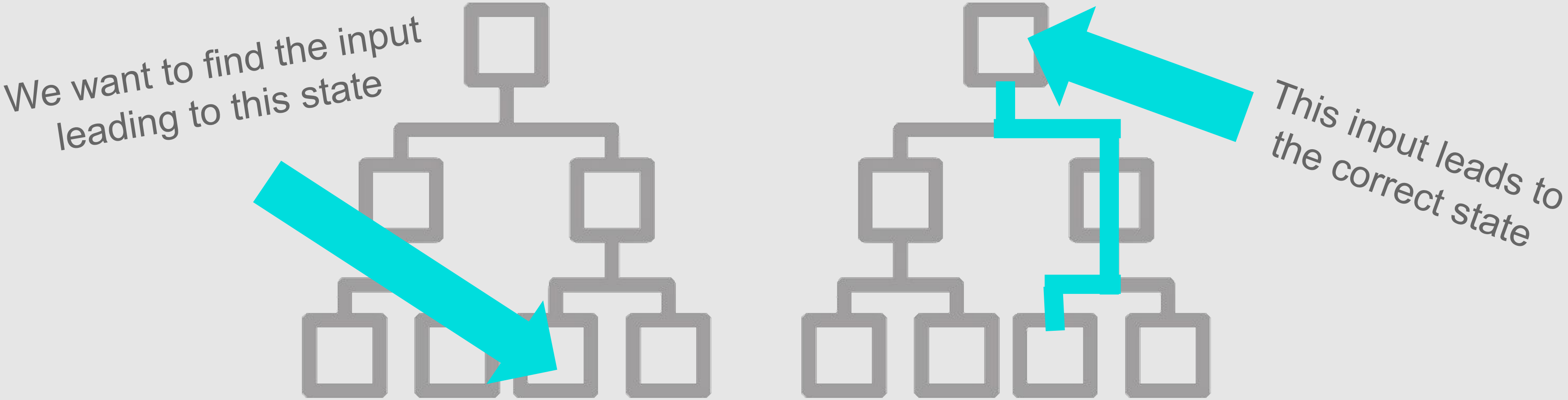
Symbolic Execution is a

“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

“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

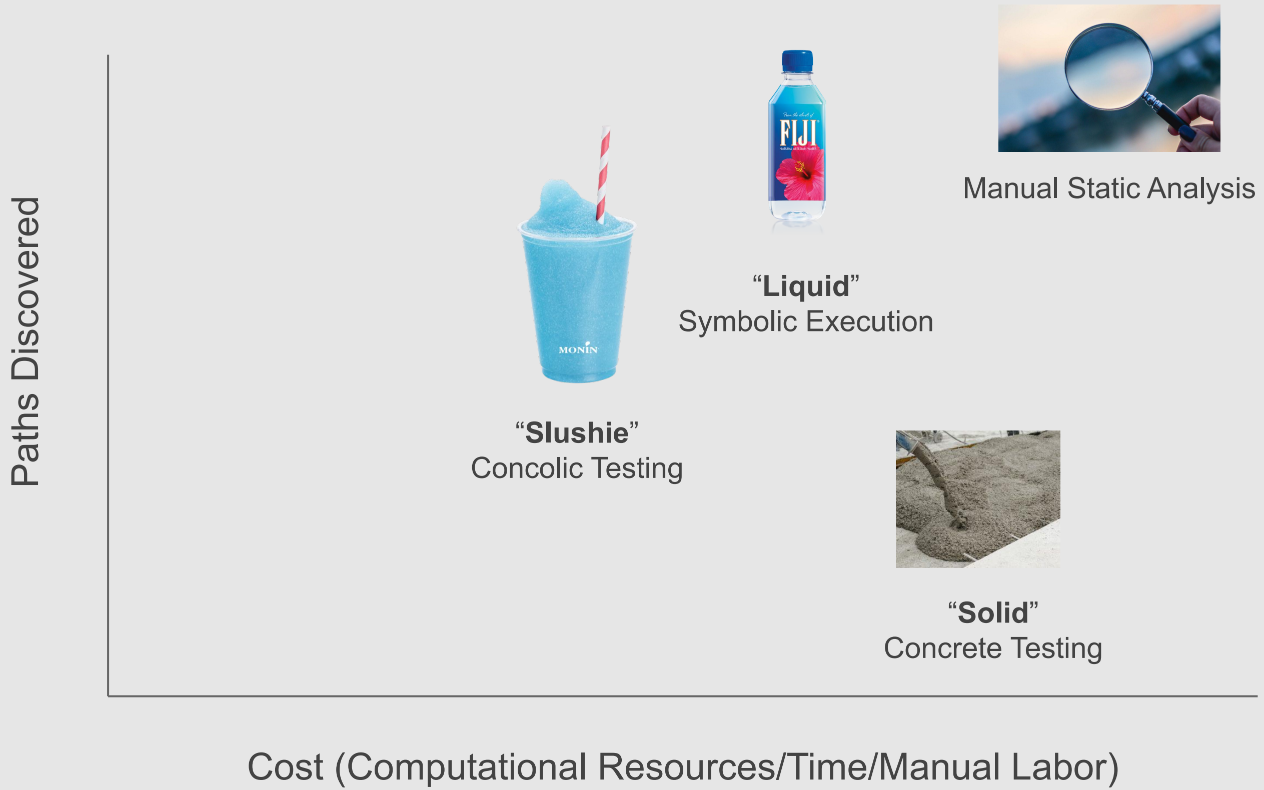
Negate the last (not already negated) constraint

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



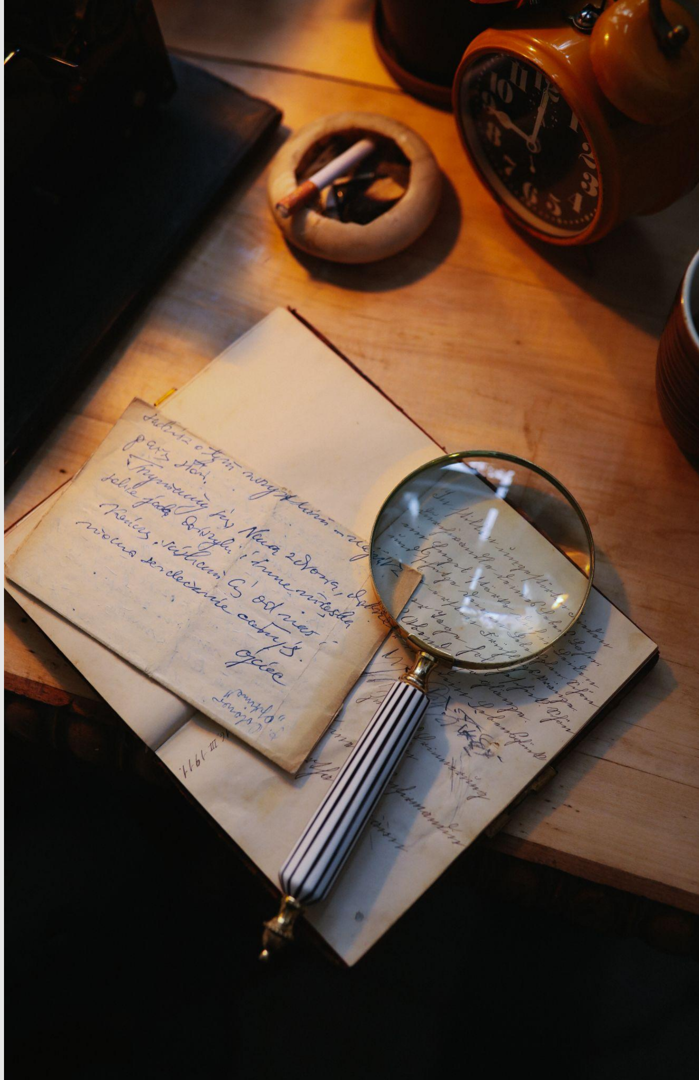
Where is Symbolic Execution Used?

Symbolic Execution **Frameworks**

Novel **Tools** (GUI's, Attack Surface Analysis, Taint Analysis, Rop Chain Generation)

Integrations into your favorite reverse engineering software

Augmented **Fuzzers**



Different symbolic execution frameworks

Full System: s2e

User: Angr
Triton
Manticore

Code: KLEE

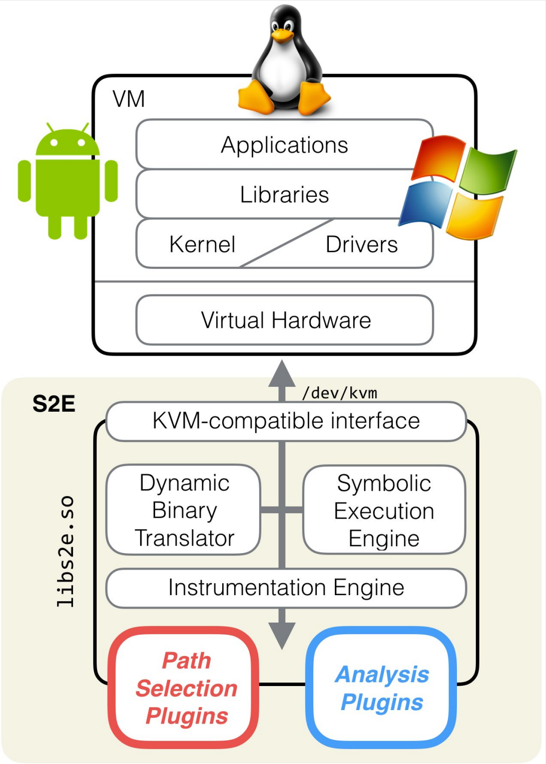
Different symbolic execution frameworks

Full System: s2e

User: Angr
Triton
Manticore

Code: KLEE

S²E: The Selective Symbolic Execution Platform



S2E Architecture Diagram

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

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

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

S2E Walkthrough

Create new S2E Project	Implement custom plugin	Define events to react to	Kill after desired state is reached	Track/Filter desired process	Limit the constraint space of the symbolic variable	On success solve constraints and display	Enable plugin in config and launch s2e
s2e new_project	libs2eplugins/src/ Plugins/yourplugin.cpp	OSMonitor for example has events for process creation, module loading/unloading onTranslateInstructionStart and executionSignal can trigger on certain address (for PID load module and calculate relative)	s2e()-> getExecutor()-> terminateStateEarly()	ProcessExecution-Detector plugin can track single process	state-> constraints .addConstraint()	!s2e()-> getExecutor()->getSymbolic- Solution()	• Build with s2e build • Edit s2e-config.lua • Run launch-s2e.sh

adrianherrera unbreakable: added missing header Latest commit f88d350 on Apr 25, 2019 History

1 contributor

44 lines (33 sloc) | 1.16 KB Raw Blame

```cpp
///
/// 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

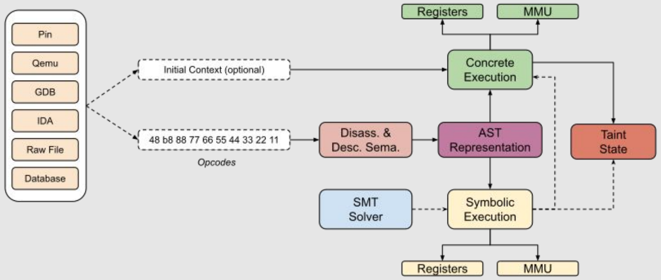
Full System: s2e

User: Angr
Triton
Manticore

Code: KLEE

Angr/Triton/Manticore

Triton Architecture

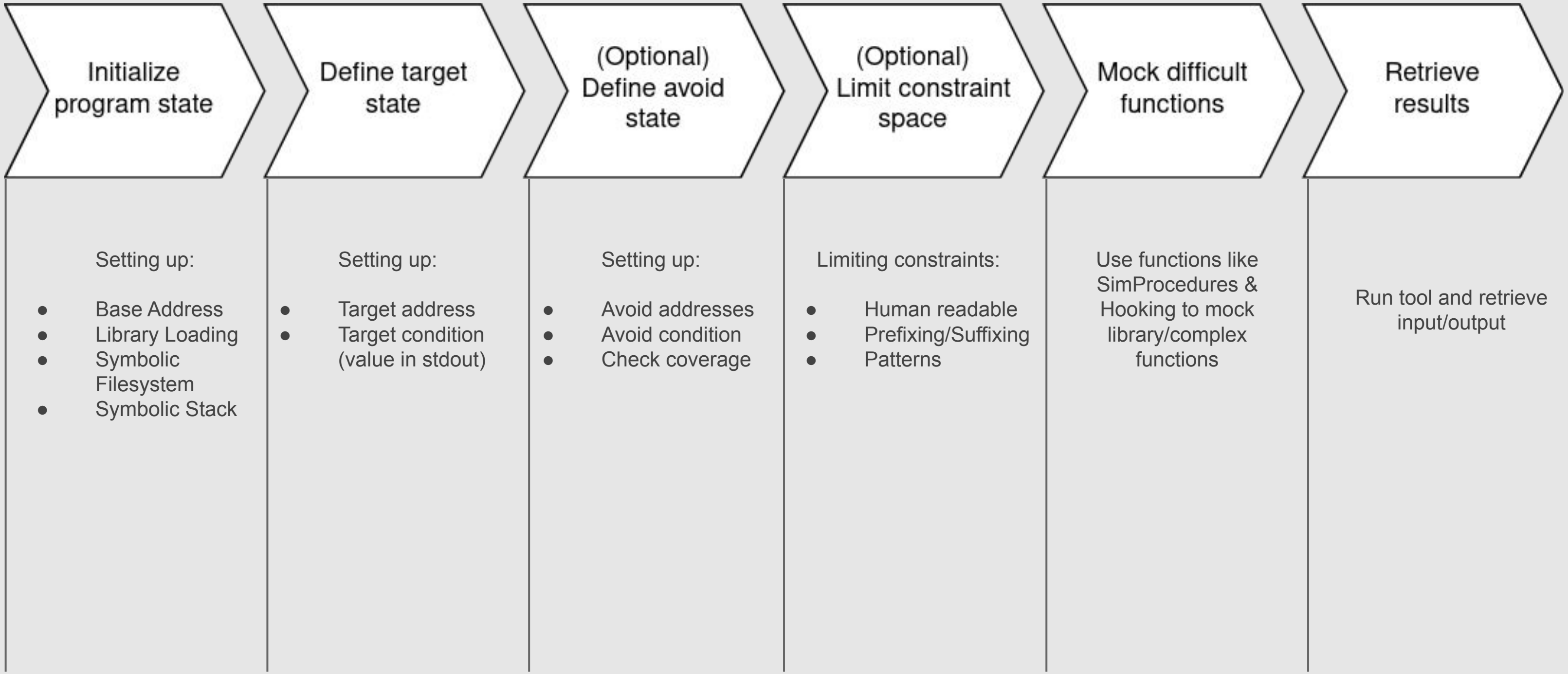


User-level dynamic binary analysis & symbolic execution frameworks (often based on z3).

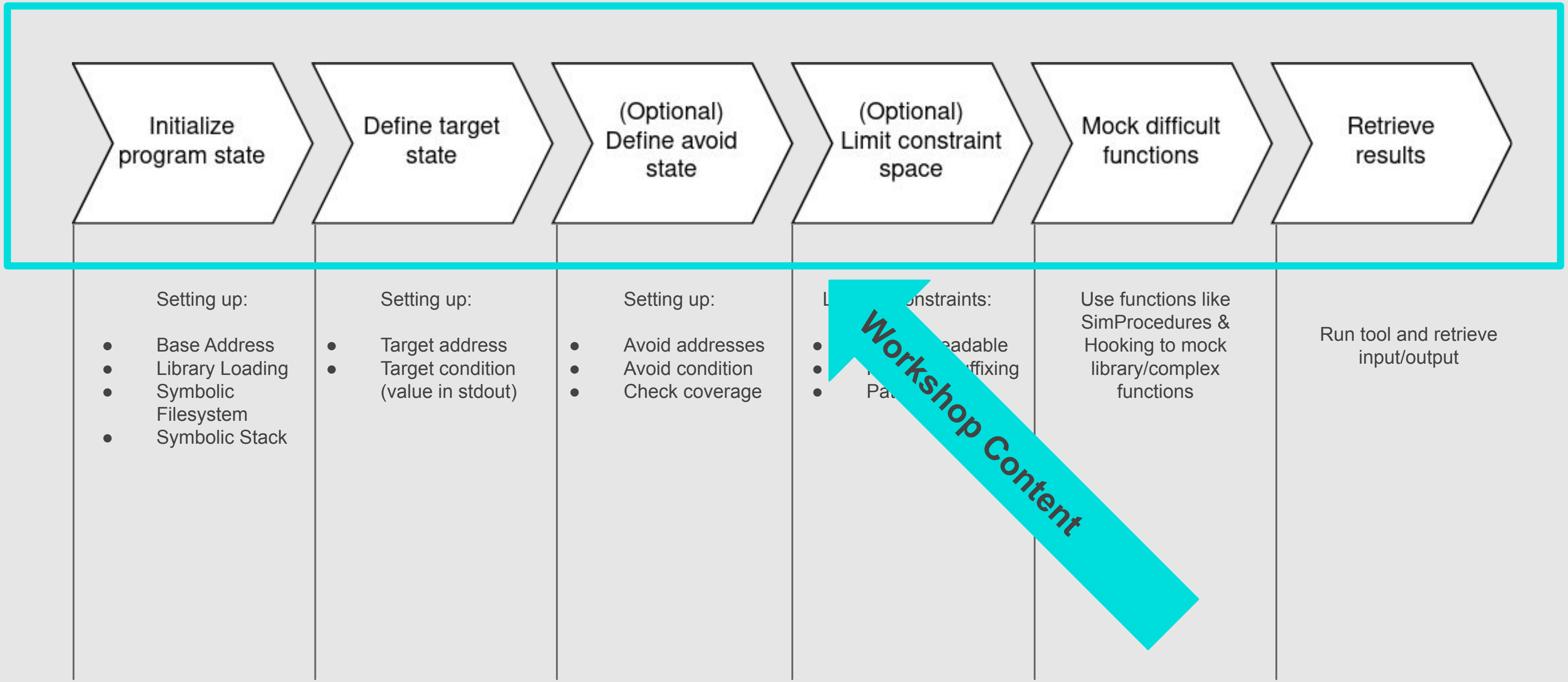
Able to lift & instrument a number of binary architectures like x86, x86-64, AArch64, EVM Smart Contracts, ARM, MIPS, WASM,PowerPC (yes, even BrainFuck)

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

User-Level Workflow



User-Level Workflow



ltfish Migrate to Python 3 (#211) Latest commit 139159e on Oct 1, 2018 History

6 contributors

Executable File | 55 lines (39 sloc) | 2.59 KB Raw Blame

```
#!/usr/bin/env python2

# Author: David Manouchehri <manouchehri@protonmail.com>
# Google 2016 CTF
# Challenge: Unbreakable Enterprise Product Activation
# Team: hack.carleton (http://hack.carleton.team/)
# Runtime: ~4.5 seconds (single threaded E5-2666 v3 @ 2.90GHz on AWS/EC2)

import angr

import claripy

def main():
    proj = angr.Project('./unbreakable-enterprise-product-activation', load_options={"auto_load_libs": False}) # Disabling the automatic library loading saves a

    input_size = 0x43; # Max length from strncpy, see 0x4005ae.

    argv1 = claripy.BVS("argv1", input_size * 8)

    initial_state = proj.factory.entry_state(args=["./unbreakable-enterprise-product-activation", argv1], add_options={angr.options.LAZY_SOLVES})
    initial_state.libc.buf_symbolic_bytes=input_size + 1 # Thanks to Christopher Salls (@salls) for pointing this out. By default there's only 60 symbolic bytes

    # For some reason if you constrain too few bytes, the solution isn't found. To be safe, I'm constraining them all.
    for byte in argv1.chop(8):
        initial_state.add_constraints(byte != '\x00') # null
        initial_state.add_constraints(byte >= ' ') # '\x20'
        initial_state.add_constraints(byte <= '~') # '\x7e'
        # Source: https://www.juniper.net/documentation/en_US/idp5.1/topics/reference/general/intrusion-detection-prevention-custom-attack-object-extended-ascii
        # Thanks to Tom Ravenscroft (@tomravenscroft) for showing me how to restrict to printable characters.

    # We're told that every flag is formatted as "CTF{...}", so we might as well use that information to save processing time.
    initial_state.add_constraints(argv1.chop(8)[0] == 'C')
    initial_state.add_constraints(argv1.chop(8)[1] == 'T')
    initial_state.add_constraints(argv1.chop(8)[2] == 'F')
    initial_state.add_constraints(argv1.chop(8)[3] == '{')
    # angr will still find the solution without setting these, but it'll take a few seconds more.

    sm = proj.factory.simulation_manager(initial_state)

                         # 0x400830 = thank you message
    sm.explore(find=0x400830, avoid=0x400850)
                                     # 0x400850 = activation failure

    found = sm.found[0] # In our case, there's only one printable solution.

    solution = found.solver.eval(argv1, cast_to=bytes)
    solution = solution[:solution.find(b"}")+1] # Trim off the null bytes at the end of the flag (if any).
    return solution

def test():
    assert main() == b'CTF{0The1Quick2Brown3Fox4Jumped5Over6The7Lazy8Fox9}'

if __name__ == '__main__':
    print(repr(main()))
```

Different symbolic execution tools

Full System: s2e

User: Angr
Triton
Manticore

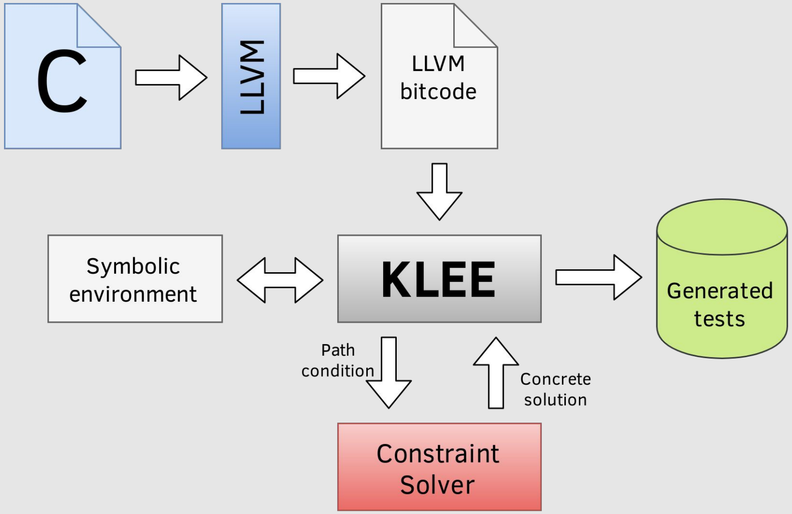
Code: KLEE

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

Extract C Code Routine from Decompiler

“File” -> “Produce file” -> Create C file…

Define symbolic Variables

```
int main()
{
    int inp, result;
    klee_make_symbolic(&inp,
        sizeof(inp), "inp");
    result = extracted_routine(inp);
    if (result == 1337)
        klee_assert(0);
}
```

```
int main()
{
    char sym[your_size];
    klee_make_symbolic(sym,
        sizeof sym, "sym");
    return extracted_routine(sym);
}
```

Compile to LLVM Bytecode

```
clang -emit-llvm \
   -g -o test.ll -c test.c
```

Run with "klee" command

```
klee test.ll
```

Examine Test Cases with "ktest-tool"

```
ktest-tool --write-ints  \
   klee-last/test000001.ktest
```

main.c · master · David M

https://gitlab.com/Manouchehri/Matryoshka-Stage-2/-/blob/ma 67%

GitLab Menu Search GitLab Sign in / Register

M Matryoshka-Stage-2

- Project information
- Repository
  - Files
  - Commits
  - Branches
  - Tags
  - Contributors
  - Graph
  - Compare
  - Locked Files
- Issues 0
- Merge requests 0
- CI/CD
- Deployments
- Monitor
- Analytics

Collapse sidebar

David Manouchehri > Matryoshka-Stage-2 > **Repository**

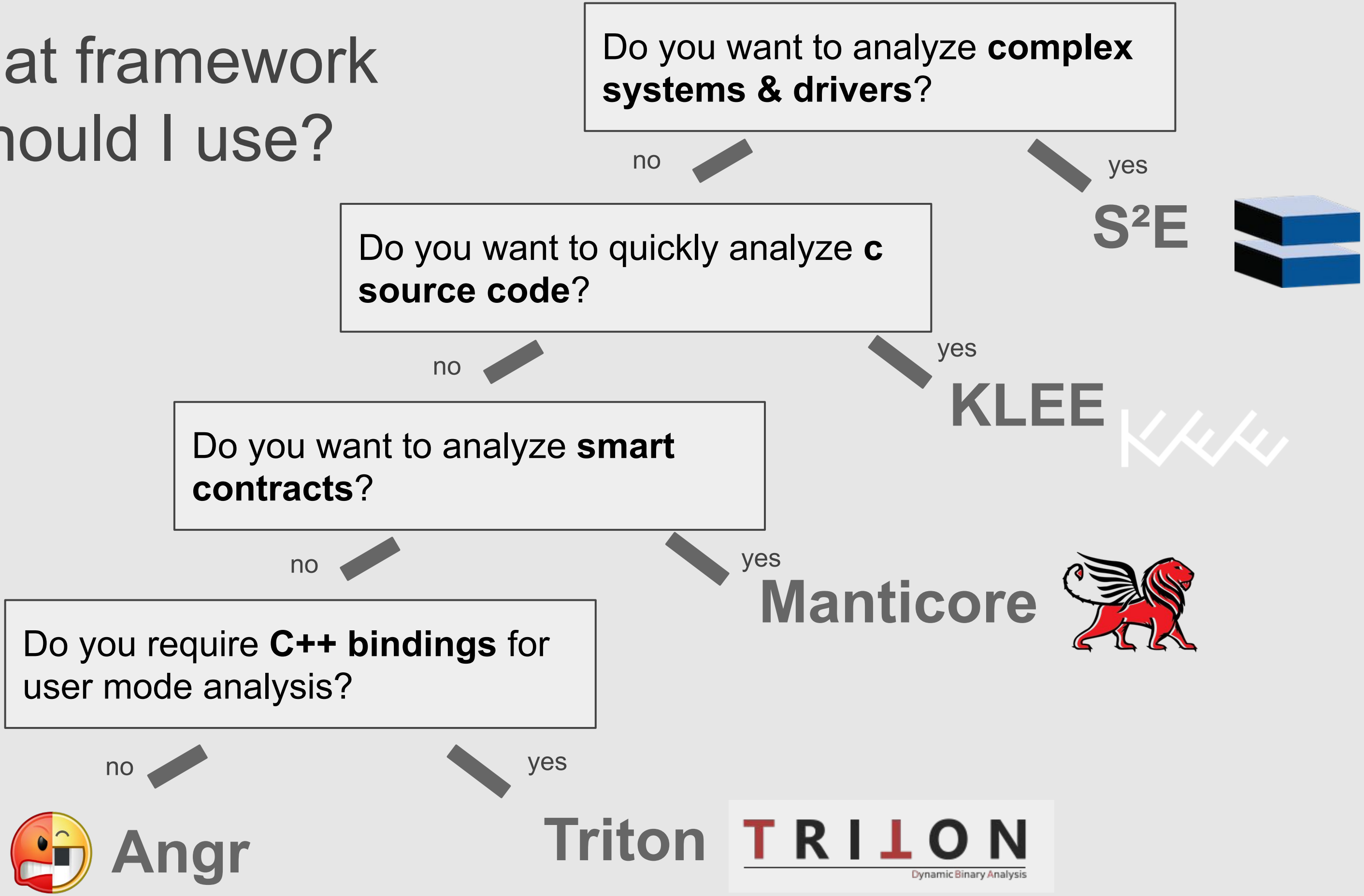
master | Matryoshka-Stage-2 / **main.c** | Find file | Blame | History | Permalink

Make Klee friendly.
David Manouchehri authored 5 years ago | 08b93e3b

main.c 1.03 KB

```c
#include "defs.h" // Take from IDA's plugins/
#include <string.h>
#include <stdio.h>
#include <assert.h>
#include <klee/klee.h>

int main(int a1, char **a2, char **a3)
{
  __int64 v4; // rbx@10
  signed int v5; // [sp+1Ch] [bp-14h]@4

  if ( a1 == 2 )
  {
    if ( 42 * (strlen(a2[1]) + 1) != 504 )
      goto LABEL_31;
    v5 = 1;
    if ( *a2[1] != 80 )
      v5 = 0;
    if ( 2 * a2[1][3] != 200 )
      v5 = 0;
    if ( *a2[1] + 16 != a2[1][6] - 16 )
      v5 = 0;
    v4 = a2[1][5];
    if ( v4 != 9 * strlen(a2[1]) - 4 )
      v5 = 0;
    if ( a2[1][1] != a2[1][7] )
      v5 = 0;
    if ( a2[1][1] != a2[1][10] )
      v5 = 0;
    if ( a2[1][1] - 17 != *a2[1] )
      v5 = 0;
    if ( a2[1][3] != a2[1][9] )
      v5 = 0;
    if ( a2[1][4] != 105 )
      v5 = 0;
    if ( a2[1][2] - a2[1][1] != 13 )
      v5 = 0;
    if ( a2[1][8] - a2[1][7] != 13 )
      v5 = 0;
    if ( v5 ) {
      printf("Good good!\n");
      klee_assert(0);
    }
    else
LABEL_31:
      printf("Try again...\n");
  }
  else
  {
    printf("Usage: %s <pass>\n", *a2);
  }
}
```

What framework should I use?



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?

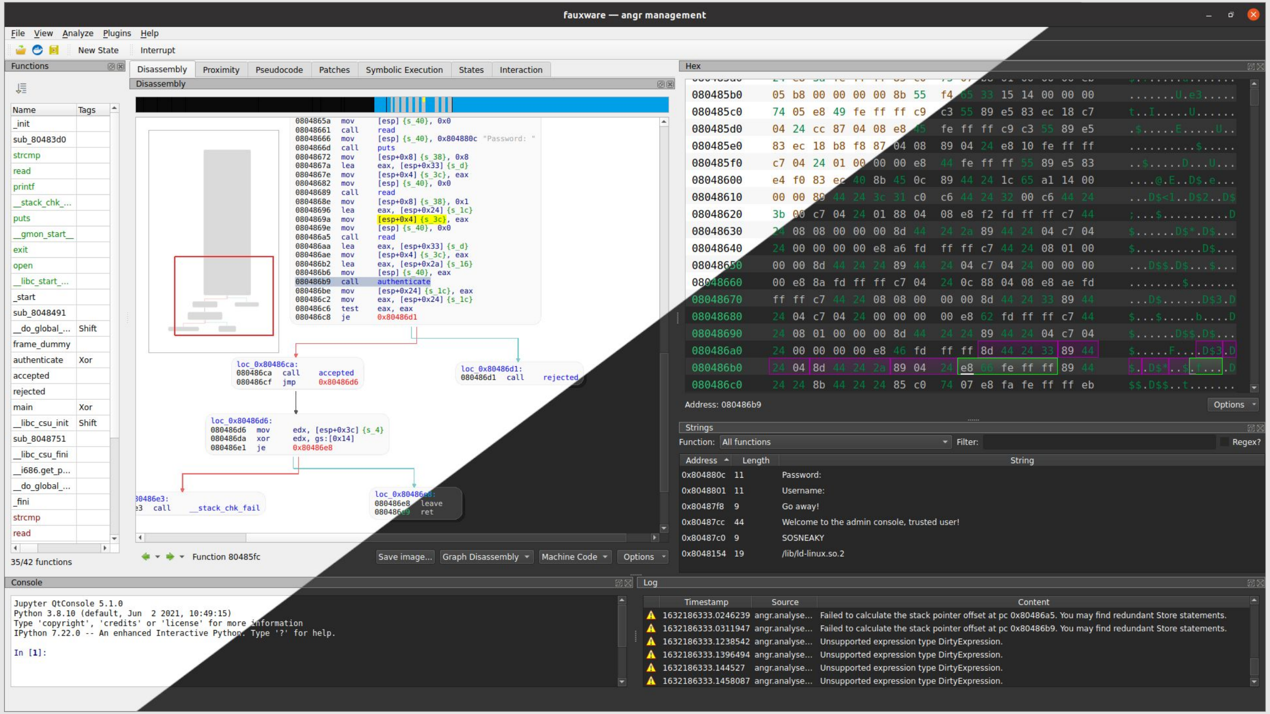
no → Angr

yes → Triton

A wonderful solution for most generic cases & CTF

Tools/Angr-Management

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



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

Tools/one_gadget

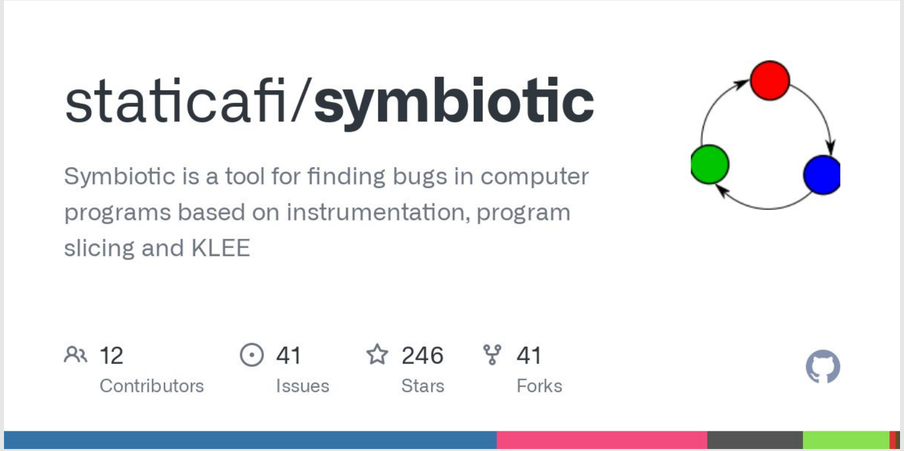
Search for magic gadgets/one gadgets in a target binary. (Single rop gadget to *execve('/bin/sh', NULL, NULL)*)

```
→ one_gadget /lib/x86_64-linux-gnu/libc.so.6
0x4f2c5 execve("/bin/sh", rsp+0x40, environ)
constraints:
  rcx == NULL

0x4f322 execve("/bin/sh", rsp+0x40, environ)
constraints:
  [rsp+0x40] == NULL

0x10a38c execve("/bin/sh", rsp+0x70, environ)
constraints:
  [rsp+0x70] == NULL
```

https://github.com/david942j/one_gadget

Tools/symbiotic

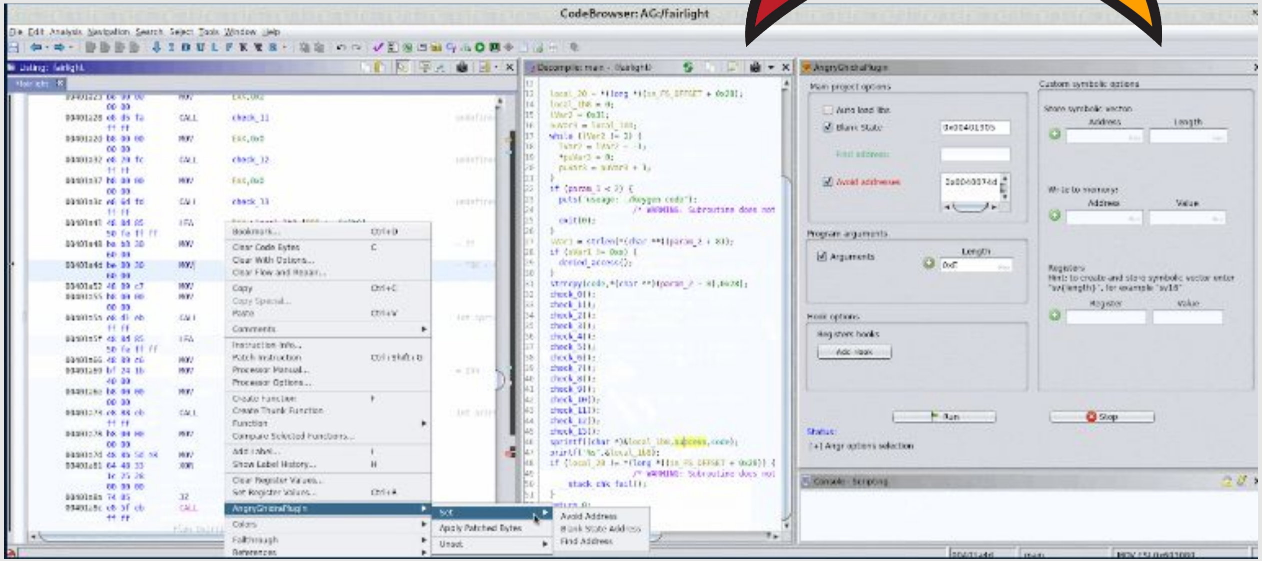
Program validation and vulnerability discovery (assertion violations, invalid pointer dereference, double free, memory leaks, etc…) using the KLEE framework



https://github.com/staticafi/symbiotic

Integrations/AngryGhidra

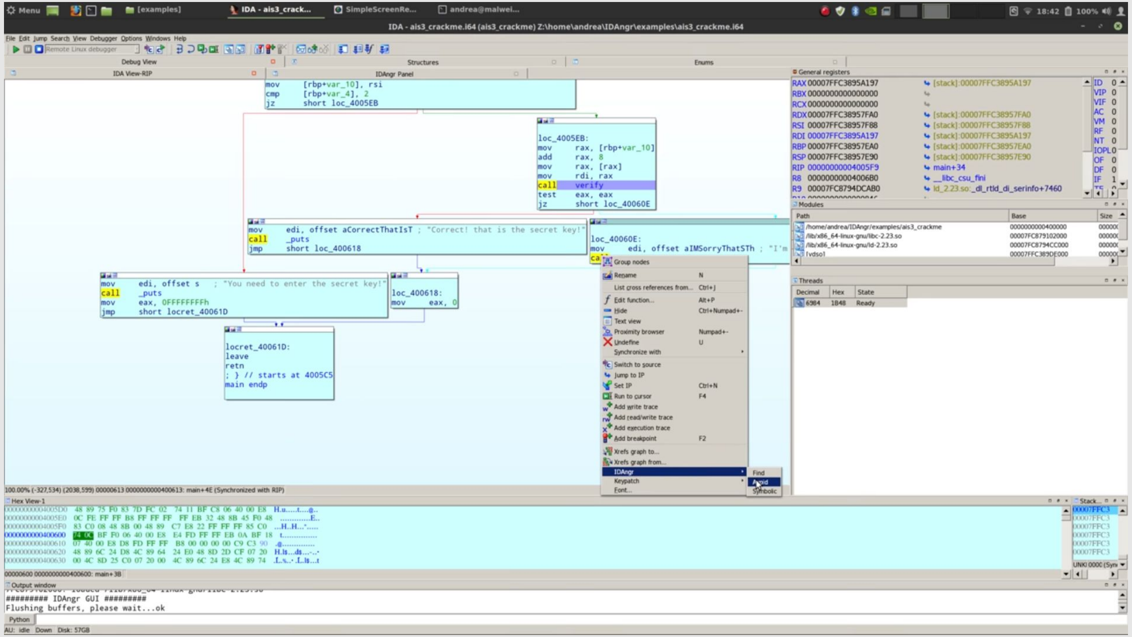
A plugin that combines the convenience of ghidra with the power of the angr framework



https://github.com/Nalen98/AngryGhidra

Integrations/IDAngr

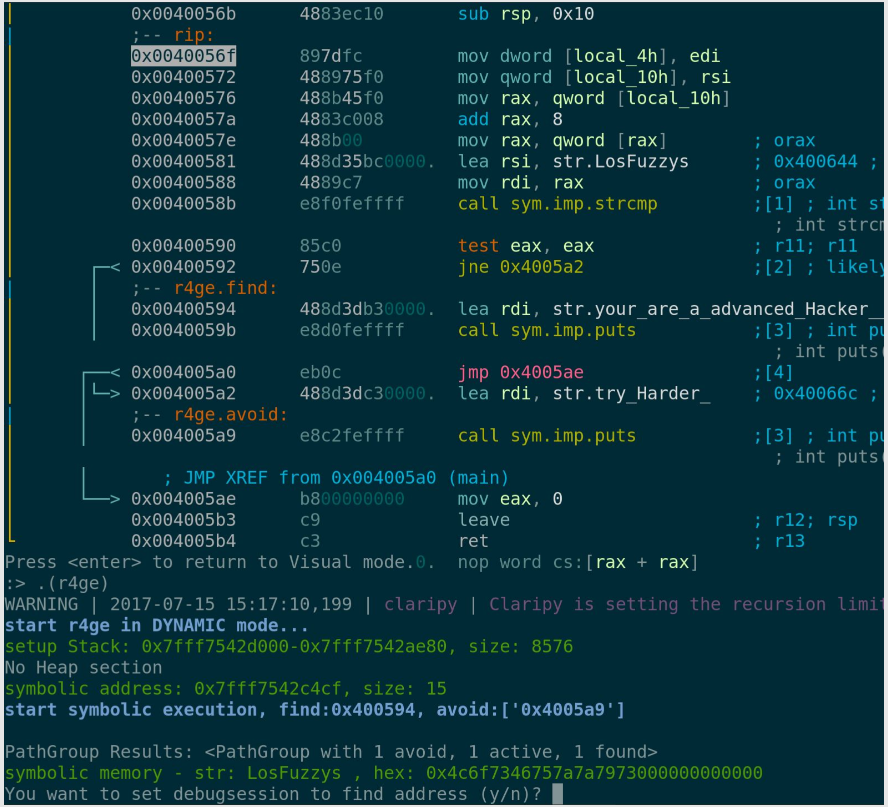
Load and explore IDA debugger state into angr (using angrdbg)



https://github.com/andreafioraldi/IDAngr

Integrations/r4ge

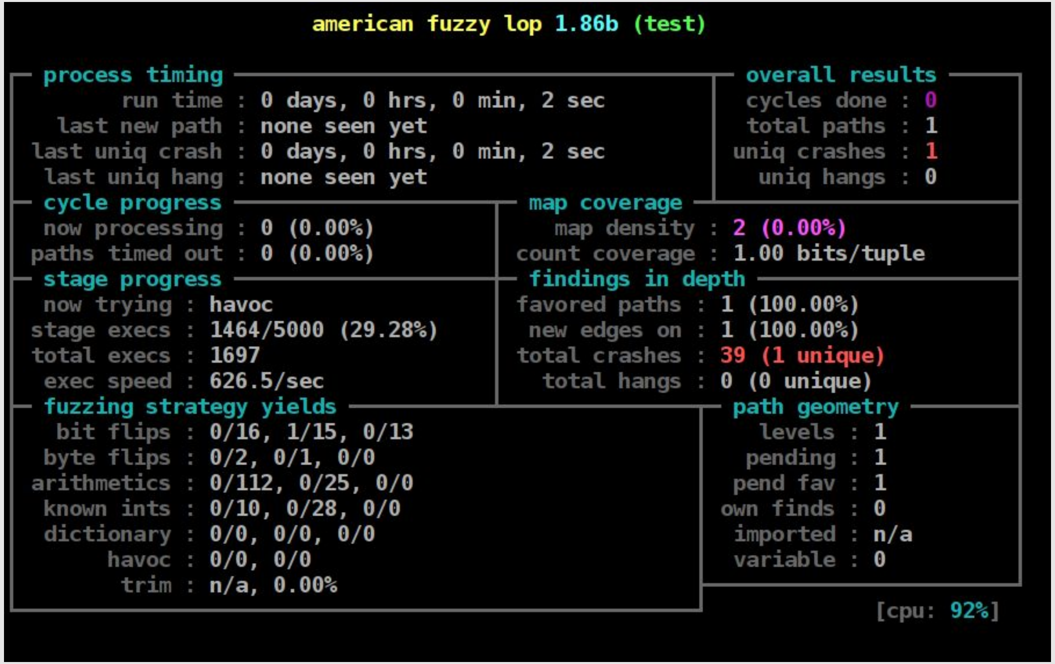
We all like radare2/rizin, now you can use angr functionalities straight from your favorite reverse engineering framework



https://github.com/gast04/r4ge

Fuzzing/Driller

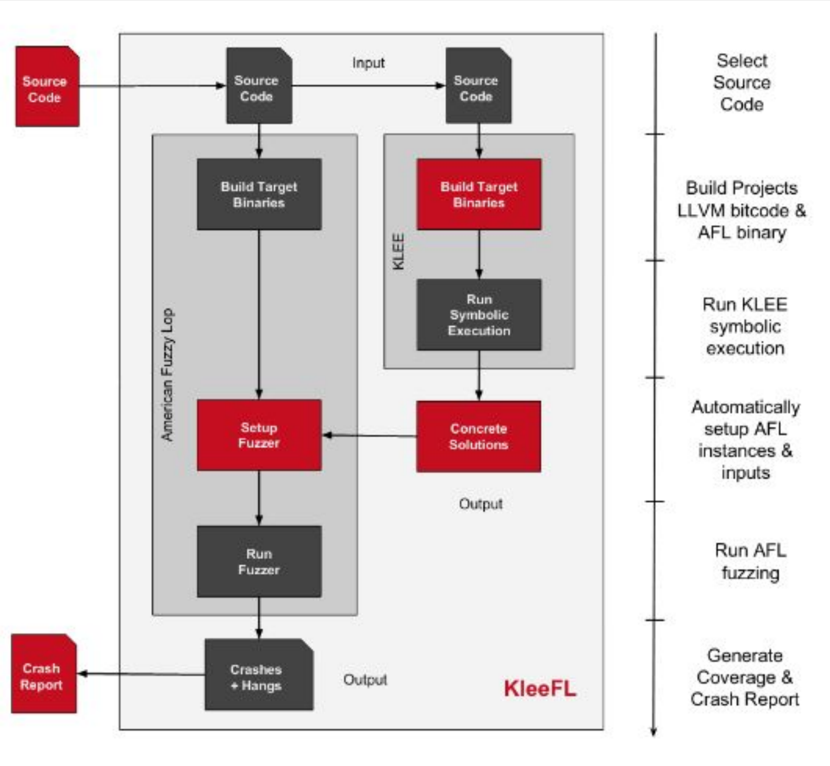
Augments the afl-fuzz capabilities with symbolic execution to discover new, interesting paths



https://github.com/shellphish/driller

Fuzzing/KleeFL

Similar to Driller but with KLEE as the symbolic execution provider



https://github.com/julieeen/kleefl

Fuzzing/LibKluzzer

A LibFuzzer extension using symbolic execution via the KLEE framework

LLVM-based Hybrid Fuzzing with LibKluzzer (Competition Contribution)

Hoang M. Le

Insitute of Computer Science
University of Bremen, Germany
hle@uni-bremen.de

Abstract. LibKluzzer is a novel implementation of hybrid fuzzing, which combines the strengths of coverage-guided fuzzing and dynamic symbolic execution (a.k.a. whitebox fuzzing). While coverage-guided fuzzing can discover new execution paths at nearly native speed, whitebox fuzzing is capable of getting through complex branch conditions. In contrast to existing hybrid fuzzers, that operate directly on binaries, LibKluzzer leverages the LLVM compiler framework to work at the source code level. It employs LibFuzzer as the coverage-guided fuzzing component and KLUZZER, an extension of KLEE, as the whitebox fuzzing component.

Keywords: Hybrid Fuzzing · Coverage-guided Fuzzing · Symbolic Execution · LLVM.

https://link.springer.com/content/pdf/10.1007/978-3-030-45234-6_29.pdf

Limitations

Non-deterministic control flow

State explosion causing exponential growth

Cryptographic primitives are still valid

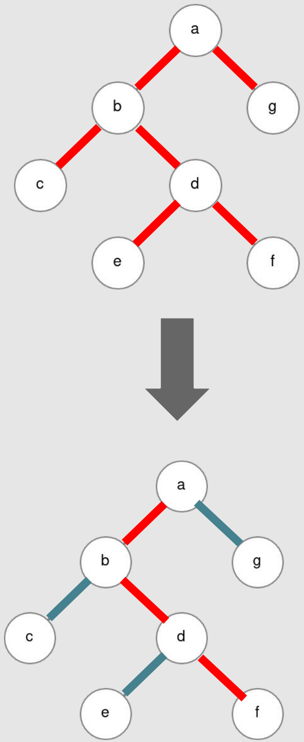
Symbolic Execution Recap

- Symbolic execution tries to find inputs that cause a program part to execute

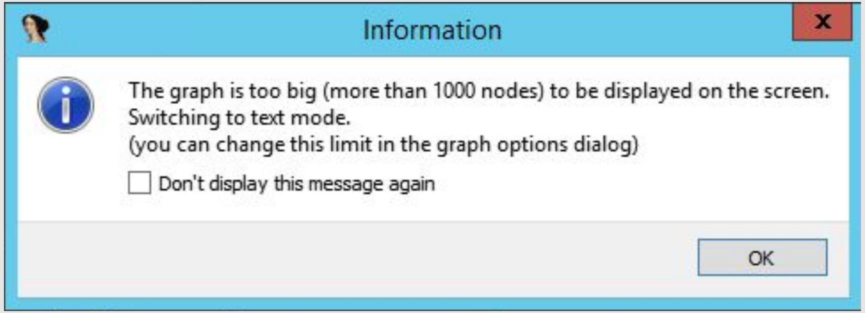
- It works by:
  - traversing an execution tree
  - accumulating constraints at each branch
  - solving them using an SMT solver

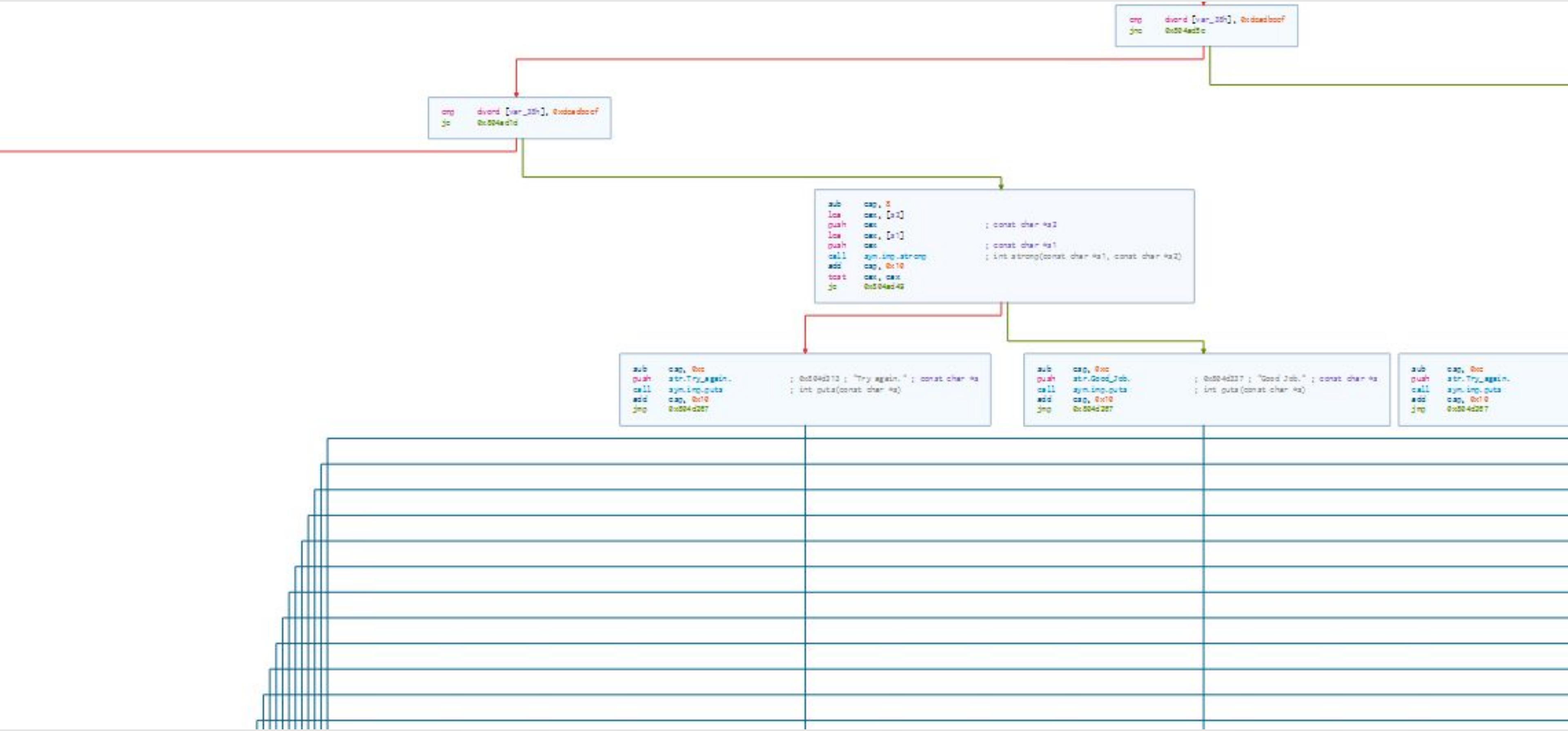
- Concolic execution is seed-driven symbolic execution that trades higher performance for potential coverage loss

- There are many symbolic execution frameworks, integrations and tools

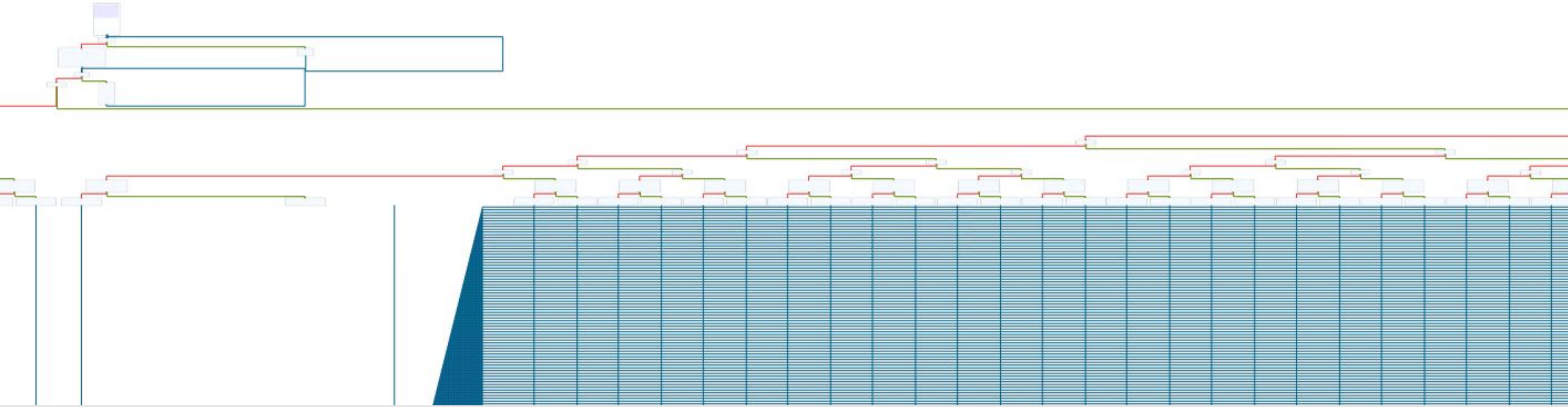


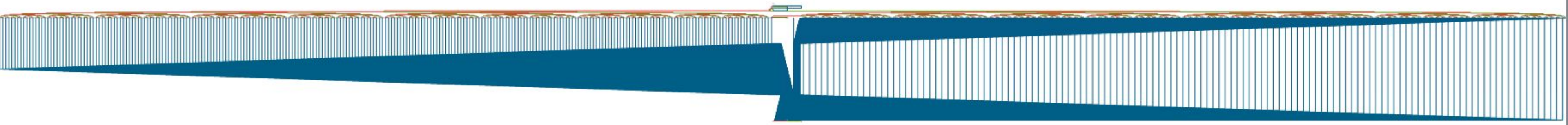
Further Readings


Information
The graph is too big (more than 1000 nodes) to be displayed on the screen.
Switching to text mode.
(you can change this limit in the graph options dialog)
Don't display this message again
OK



cmp dword [var_38h], 0xdeadbeef
jne
cmp dword [var_38h], 0xdeadbeef
je
sub esp, 8
lea eax, [s2]
push eax ; const char *s2
lea eax, [s1]
push eax ; const char *s1
call sym.imp.strcmp ; int strcmp(const char *s1, const char *s2)
add esp, 0x10
test eax, eax
je
sub esp, 0xc
push str.Try_again. ; "Try again." ; const char *s
call sym.imp.puts ; int puts(const char *s)
add esp, 0x10
jmp
sub esp, 0xc
push str.Good_Job. ; "Good Job." ; const char *s
call sym.imp.puts ; int puts(const char *s)
add esp, 0x10
jmp
sub esp, 0xc
push str.Try_again.
call sym.imp.puts
add esp, 0x10
jmp





Demo


Do we get to mess with him now?
No. We wait until the worst possible moment.
CI/CD
Demo


DEAR DEMO GODS
WE HUMBLY REQUEST ONE FINAL DEMO
memegenerator.net

What we'll learn at the **Workshop**

- The **user-level symbolic execution workflow** in-depth
- Solve **practical challenges** using the **angr framework**
- How to tackle **performance** issues
- Gaining a **CTF** edge via **implicit constraints**
- Exporting **code coverage** from angr runs

Complete slides will be shared at the workshop :)

@xorkiwi

/in/janniskirschner

Reverse engineering of **black-box binaries** with **symbolic** and **concolic execution techniques**

or

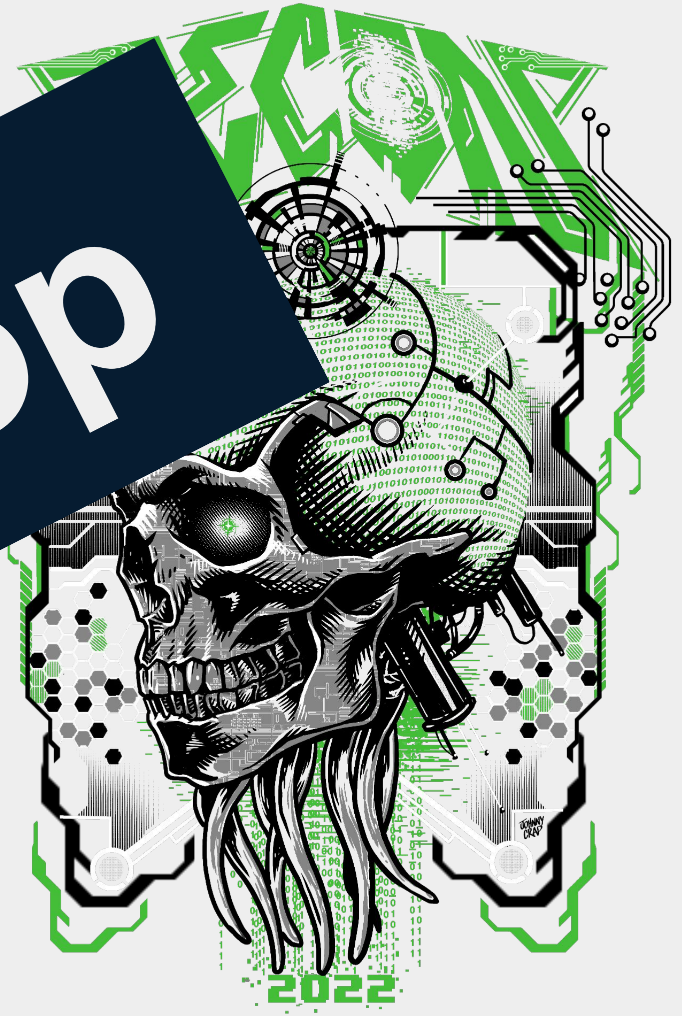
“Why huge call-graphs don’t scare me anymore”

REcon Montreal 2022 | Jannis Kirschner

Reverse engineering of
black-box binaries
symbolic and
executi

Workshop

"W anymore"



REcon Montr | Jannis Kirschner

angr

Extensive Binary Analysis Framework

Convenient Python3 Interface

Valgrind

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

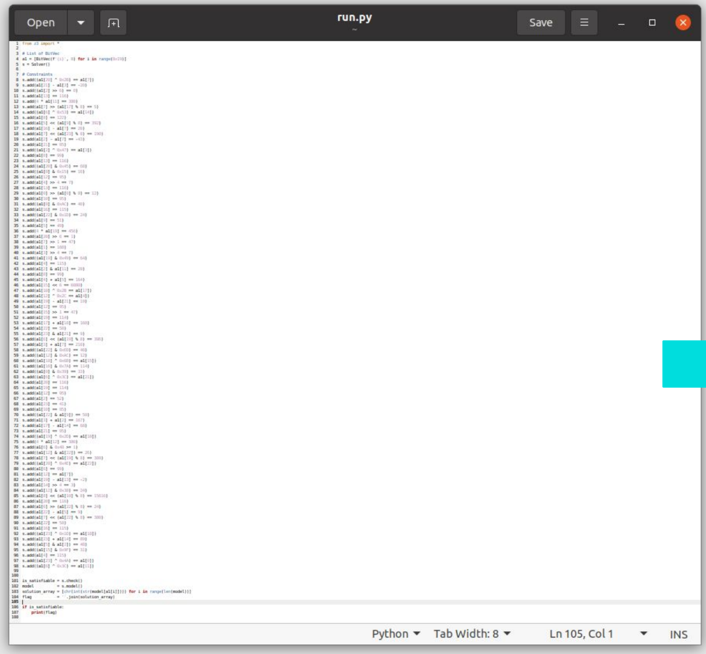
Symbolic + Concolic Execution

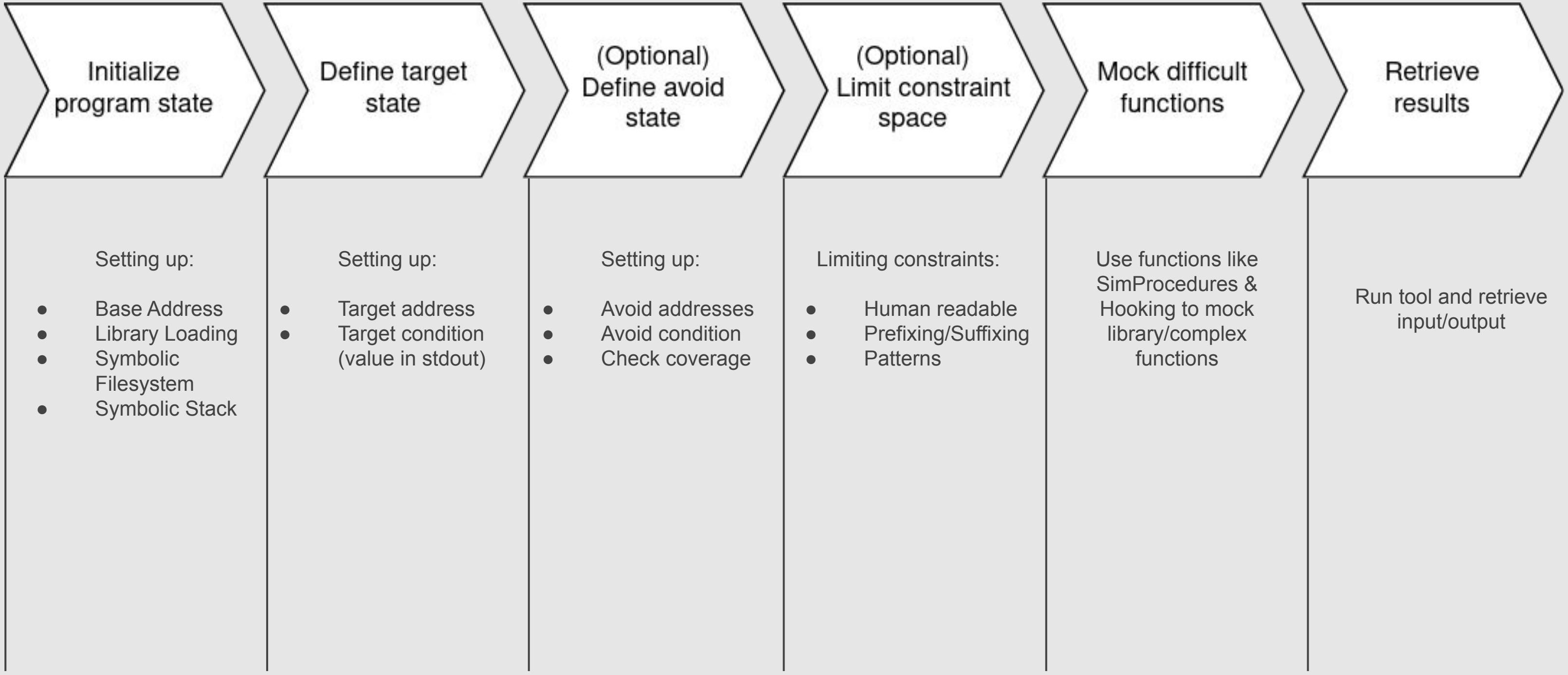
Developed by UCSB

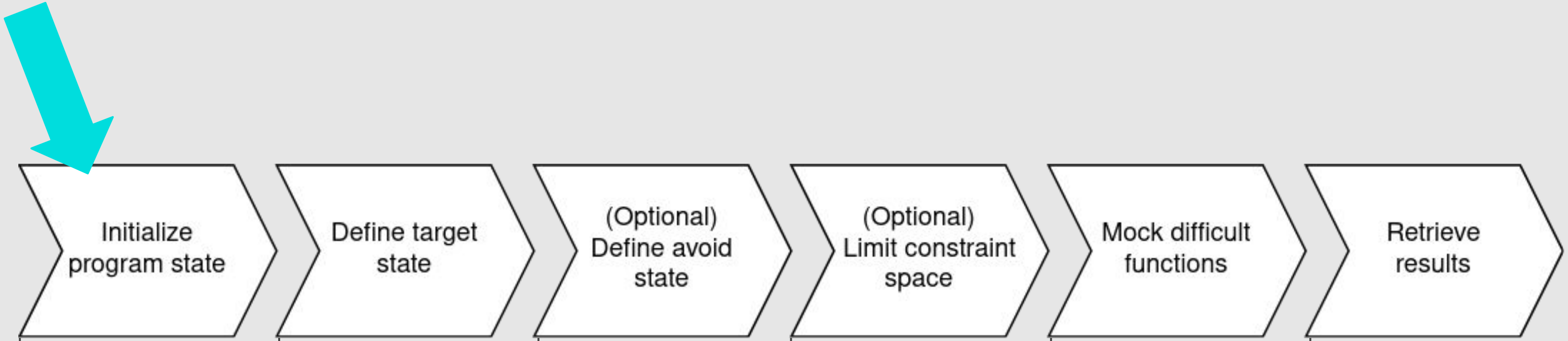
Won 3rd in DARPA Cyber Grand Challenge

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

Let’s recap for a second




Initialize program state
Define target state
(Optional) Define avoid state
(Optional) Limit constraint space
Mock difficult functions
Retrieve results
Setting up:
• Base Address
• Library Loading
• Symbolic Filesystem
• Symbolic Stack
Setting up:
• Target address
• Target condition (value in stdout)
Setting up:
• Avoid addresses
• Avoid condition
• Check coverage
Limiting constraints:
• Human readable
• Prefixing/Suffixing
• Patterns
Use functions like SimProcedures & Hooking to mock library/complex functions
Run tool and retrieve input/output



Setting up:

- Base Address
- Library Loading
- Symbolic Filesystem
- Symbolic Stack

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

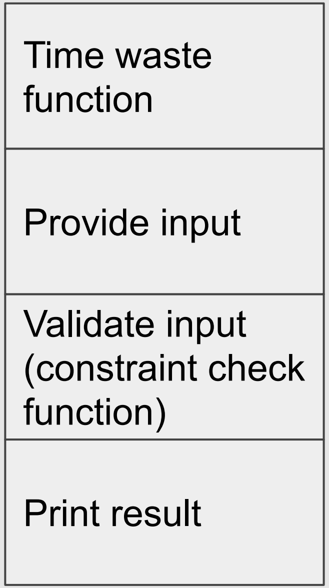
```
import angr, claripy

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




```
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

```python
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 = (simgr.found[0].solver.eval(password0))
solution1 = (simgr.found[0].solver.eval(password1))

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

Symbolic Stack

Set start address after input was provided

Initialize stack frame

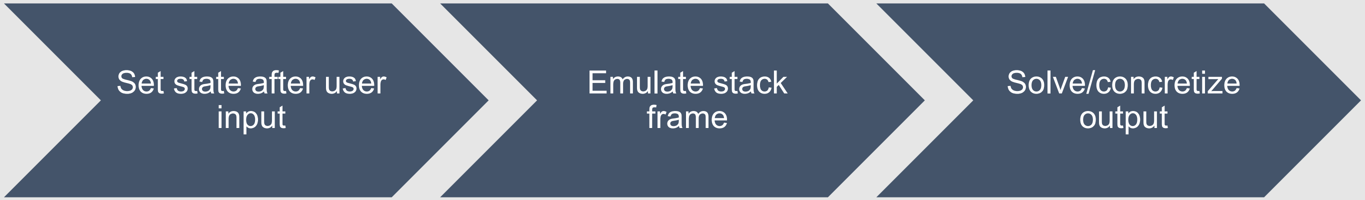
Define password bitvectors

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

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


Set state after user input
Emulate stack frame
Solve/concretize output

Symbolic Filesystem

Provide input via file

Validate input (constraint check function)

Print result

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

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

   int result = check_flag(input);

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

   return 0;
}
```

Symbolic Filesystem

Set start address before input

```
start_addr = 0x000013cc
```

Symbolic Filesystem

Set start address before input

Define symbolic bitvector

```
start_addr = 0x000013cc

filename = './inputfile.txt'
sym_file_size = 64

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

Symbolic Filesystem

Set start address before input

Define symbolic bitvector

Define symbolic file with bitvector as content

```
start_addr = 0x000013cc

filename = './inputfile.txt'
sym_file_size = 64

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

password_file = angr.SimFile(filename,
          content = password,
          size = sym_file_size
          )
```

Symbolic Filesystem

Set start address before input

Define symbolic bitvector

Define symbolic file with bitvector as content

Define initial state with start address and filesystem

```
start_addr = 0x000013cc

filename = './inputfile.txt'
sym_file_size = 64

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

password_file = angr.SimFile(filename,
          content = password,
          size = sym_file_size
          )

initial_state = proj.factory.blank_state(
          addr = start_addr,
          fs = {filename: password_file}
)
```

Symbolic Filesystem

Set start address before input

Define symbolic bitvector

Define symbolic file with bitvector as content

Define initial state with start address and filesystem

Solve symbolic memory

```
start_addr = 0x000013cc

filename = './inputfile.txt'
sym_file_size = 64

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

password_file = angr.SimFile(filename,
          content = password,
          size = sym_file_size
          )

initial_state = proj.factory.blank_state(
          addr = start_addr,
          fs = {filename: password_file}
)

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

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

Symbolic Filesystem

Set start address before input

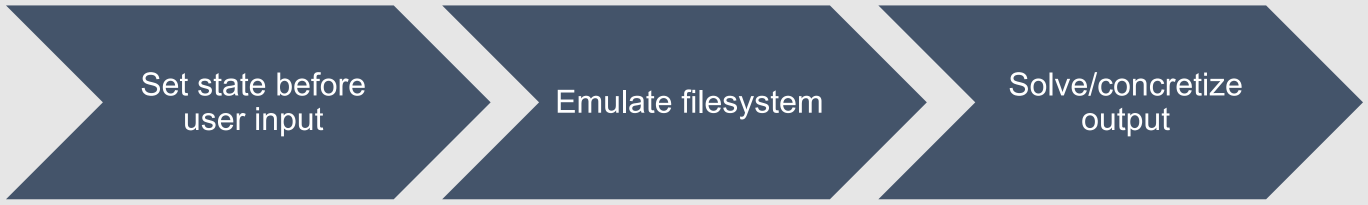
Define symbolic bitvector

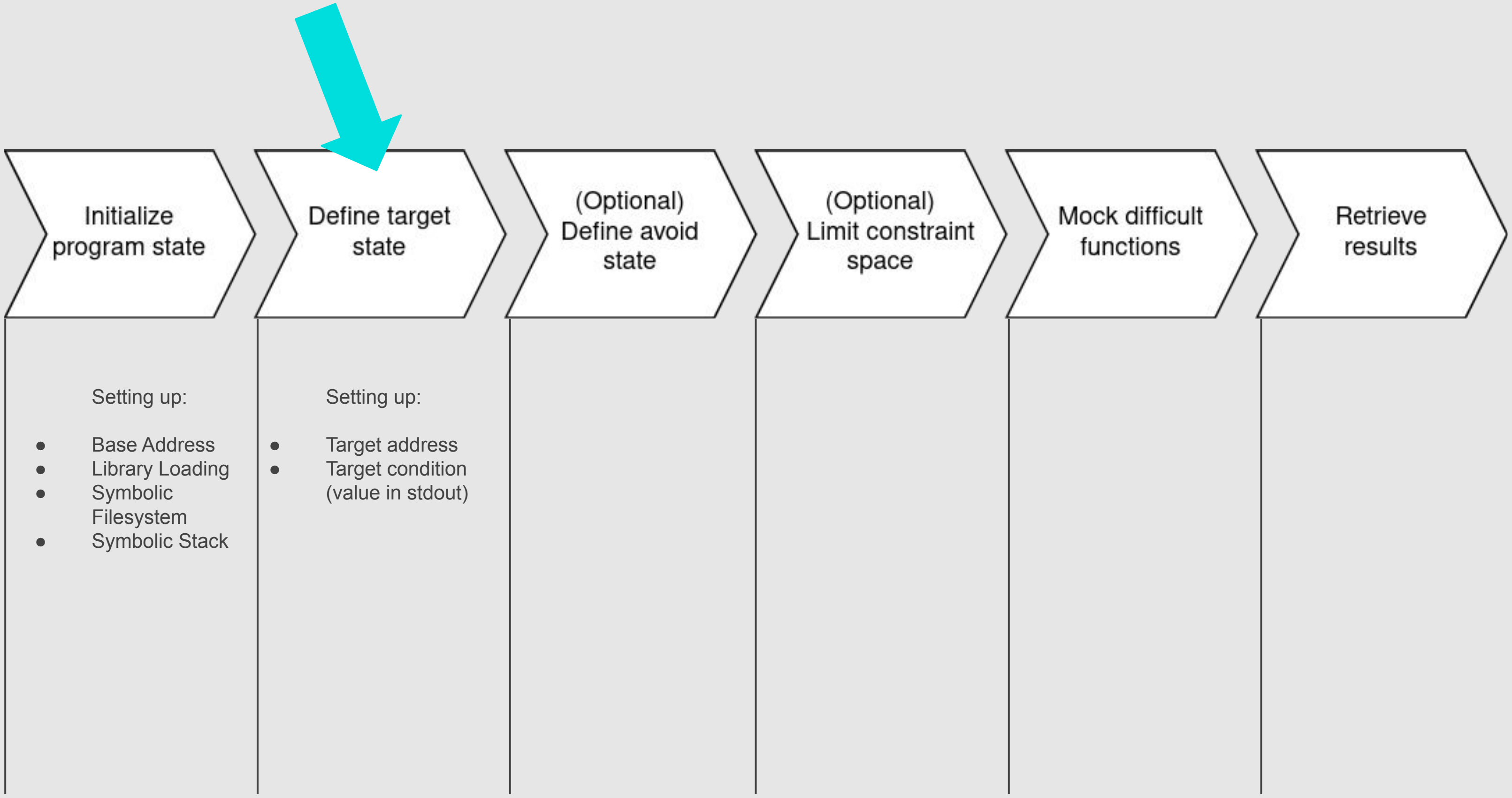
Define symbolic file with bitvector as content

Define initial state with start address and filesystem

Solve symbolic memory

```
import angr, claripy

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

start_addr = 0x000013cc

filename = './inputfile.txt'
sym_file_size = 64

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

password_file = angr.SimFile(filename,
        content = password,
        size = sym_file_size
        )

initial_state = proj.factory.blank_state(
        addr = start_addr,
        fs = {filename: password_file}
)

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

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


Set state before user input
Emulate filesystem
Solve/concretize output


Initialize program state
Define target state
(Optional) Define avoid state
(Optional) Limit constraint space
Mock difficult functions
Retrieve results
Setting up:
Base Address
Library Loading
Symbolic Filesystem
Symbolic Stack
Setting up:
Target address
Target condition (value in stdout)

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

```python
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
)
```

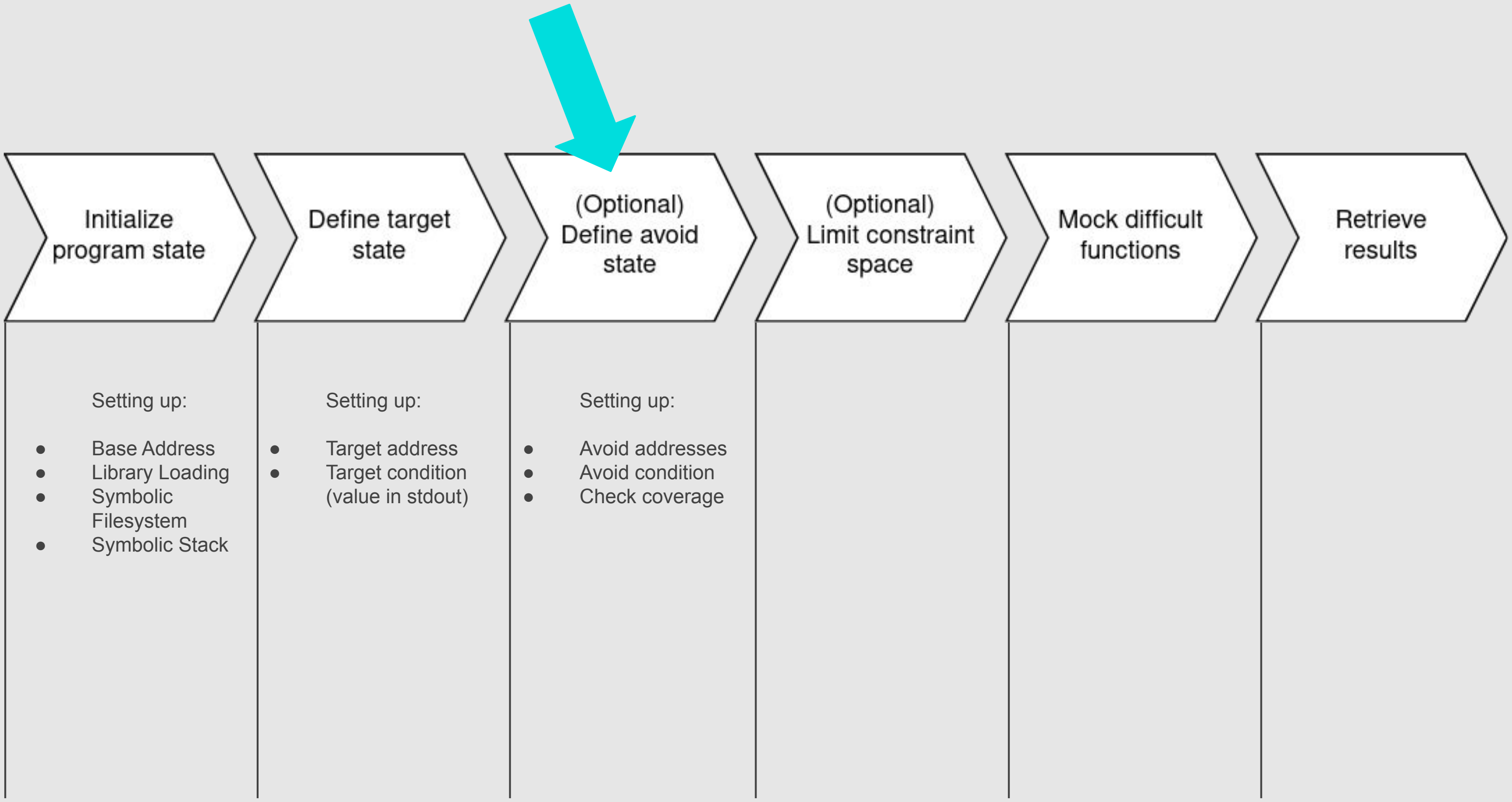
Can also be value

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()))
```


Initialize program state
Define target state
(Optional) Define avoid state
(Optional) Limit constraint space
Mock difficult functions
Retrieve results
Setting up:
• Base Address
• Library Loading
• Symbolic Filesystem
• Symbolic Stack
Setting up:
• Target address
• Target condition (value in stdout)
Setting up:
• Avoid addresses
• Avoid condition
• Check coverage

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

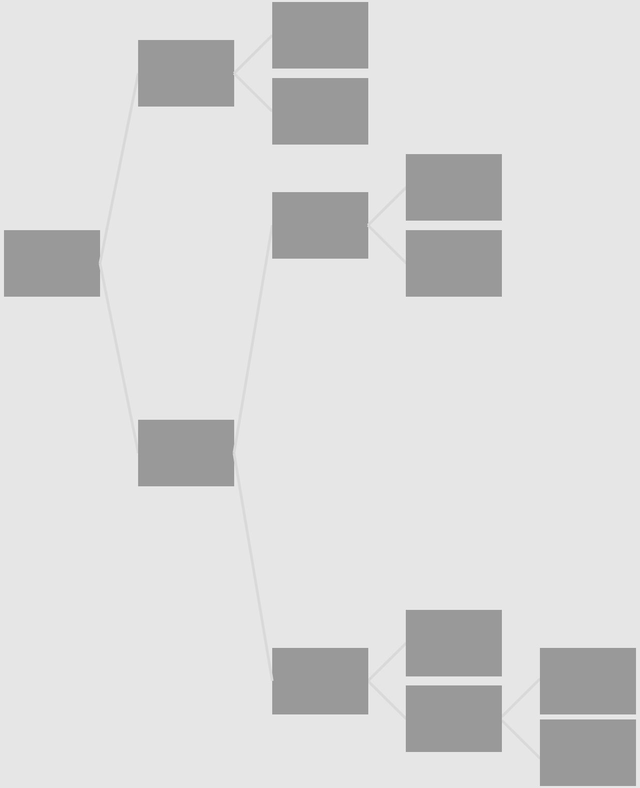
```
simgr.explore(find=0x00001407, avoid=[0x0000142d])
```

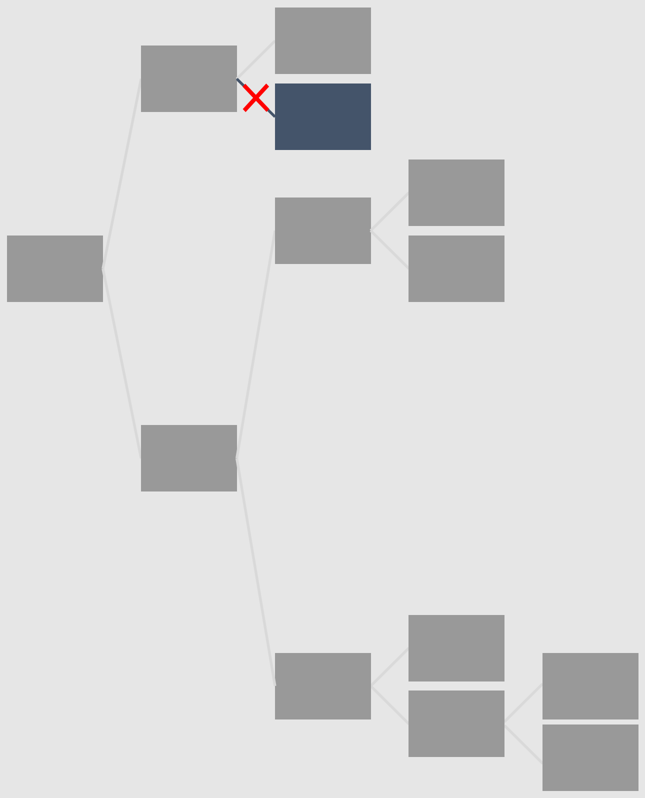
Just exclude, it's easy

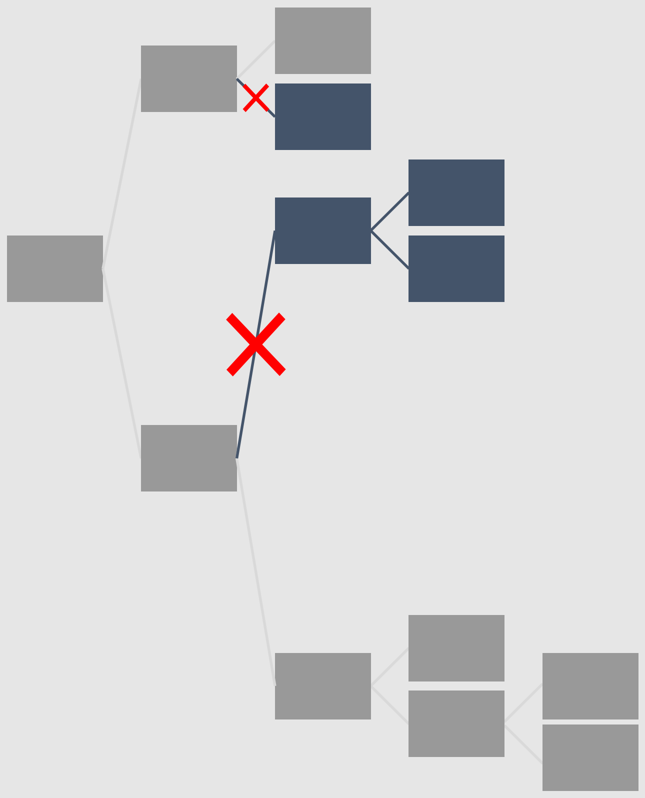
A great way to reduce complexity is by entirely avoiding unneeded paths

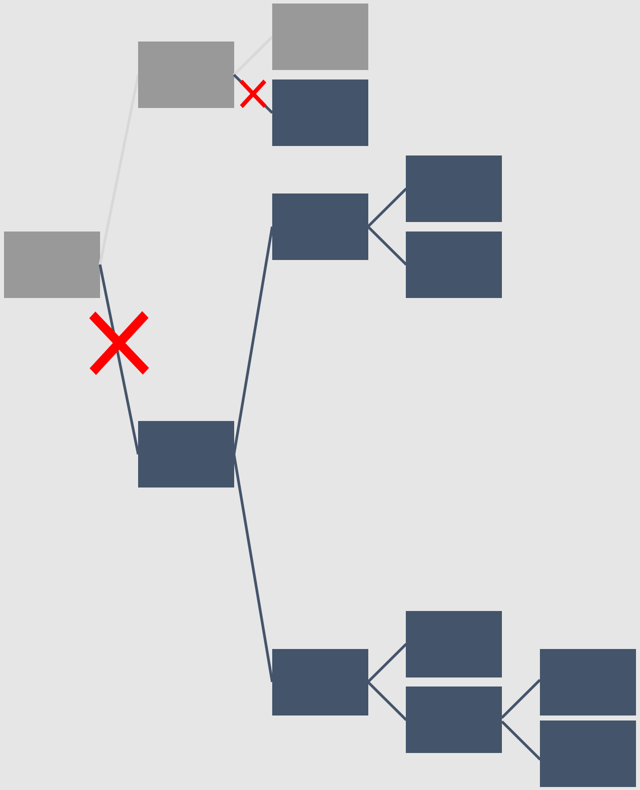
Selecting those paths works best with reverse engineering & human intuition

```
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))
```

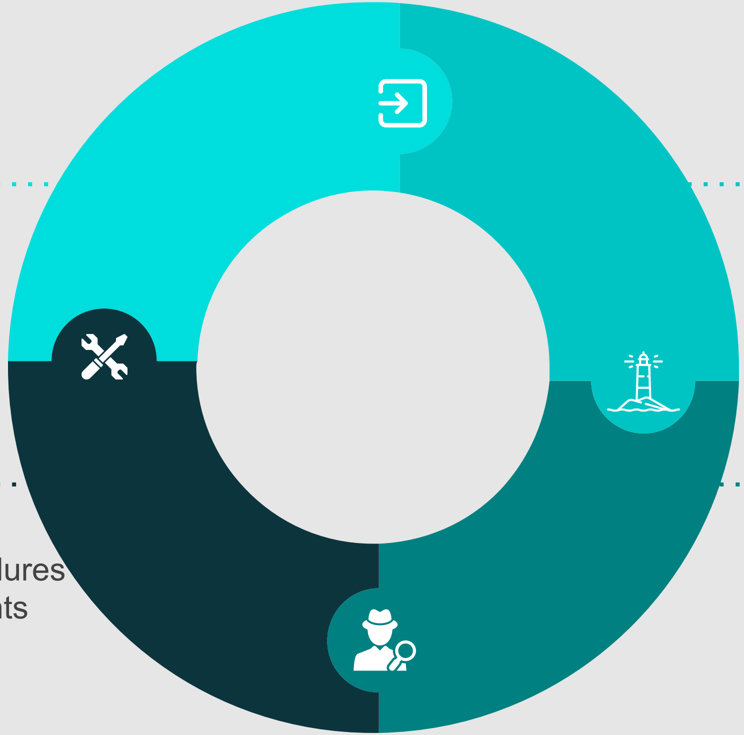








Code Coverage Collection Process



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

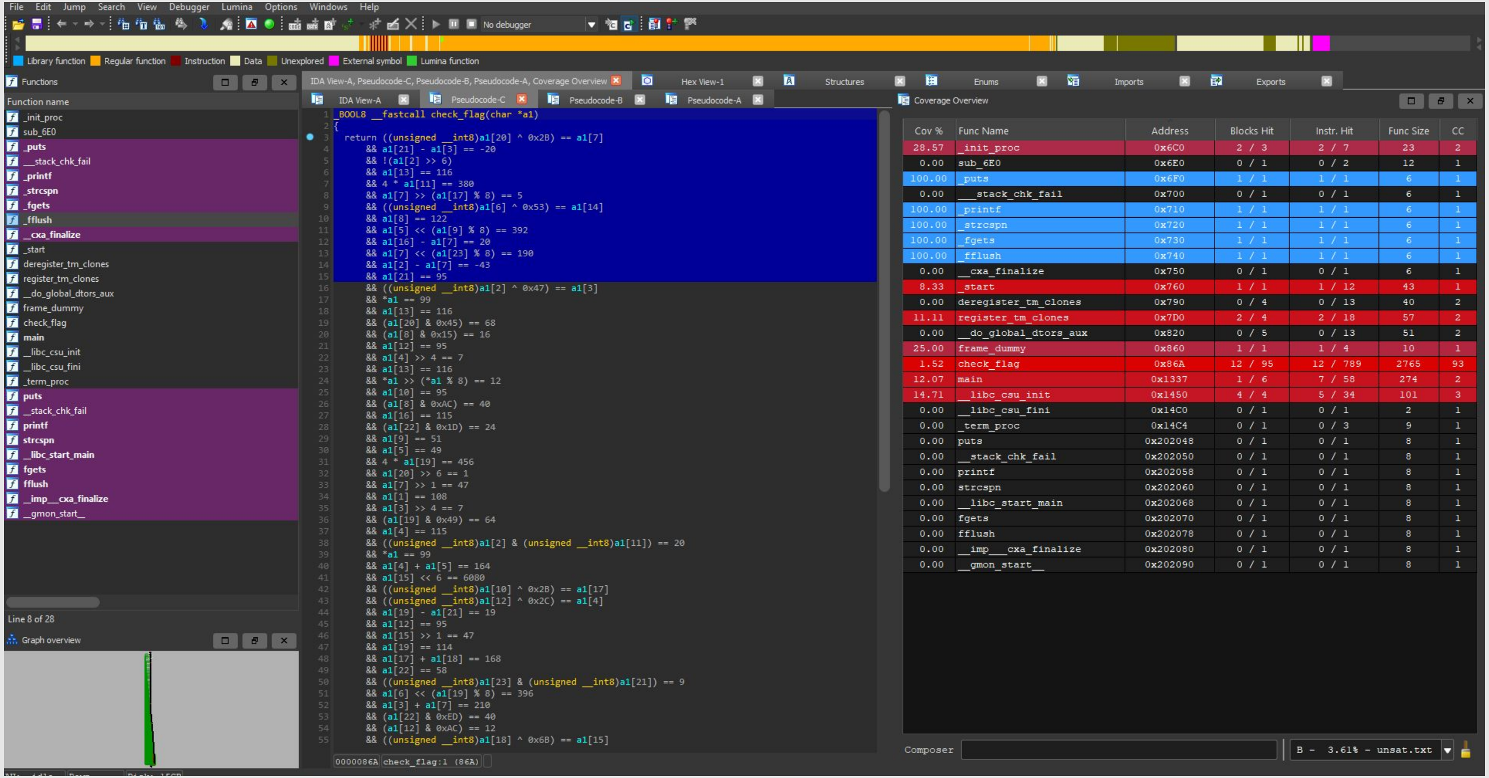
Fix Bottlenecks

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

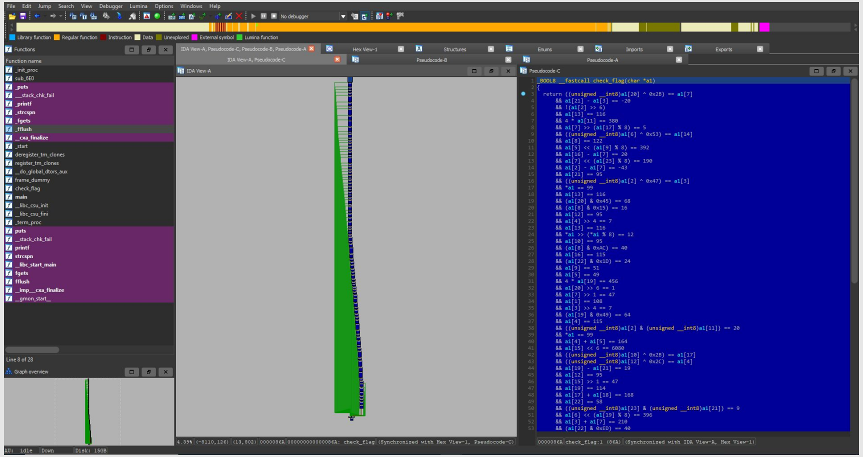
Code Coverage

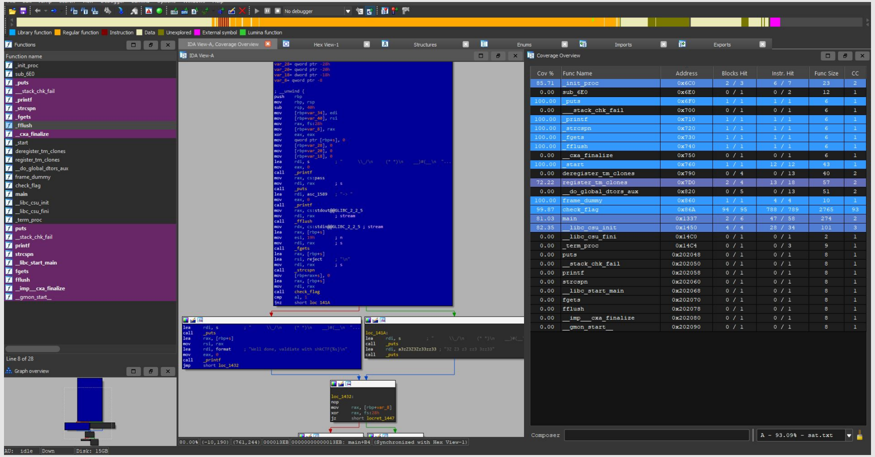
```
def get_coverage(*args, **kwargs):
    sm = args[0]
    stashes = sm.stashes
    i = 0
    for simstate in stashes['active']:
        state_history = ''

        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_coverage)
```

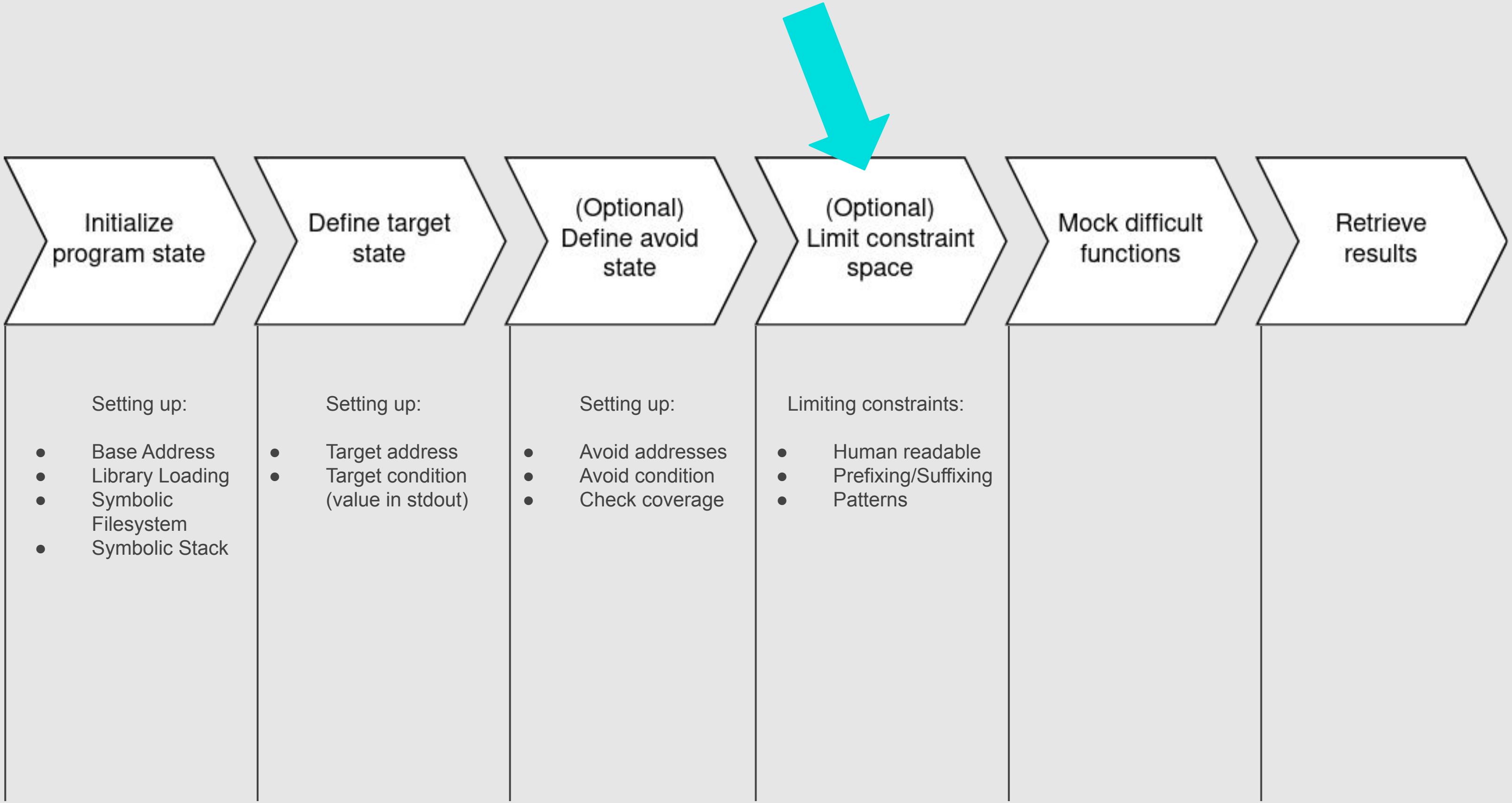
Load into lighthouse to find bottlenecks…



…and guide angr into resolving them







Initialize program state
Define target state
(Optional) Define avoid state
(Optional) Limit constraint space
Mock difficult functions
Retrieve results
Setting up:
Base Address
Library Loading
Symbolic Filesystem
Symbolic Stack
Setting up:
Target address
Target condition (value in stdout)
Setting up:
Avoid addresses
Avoid condition
Check coverage
Limiting constraints:
Human readable
Prefixing/Suffixing
Patterns

Limiting Constraints

Import the constraint solver engine

```python
import angr, claripy
```

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{“

```python
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

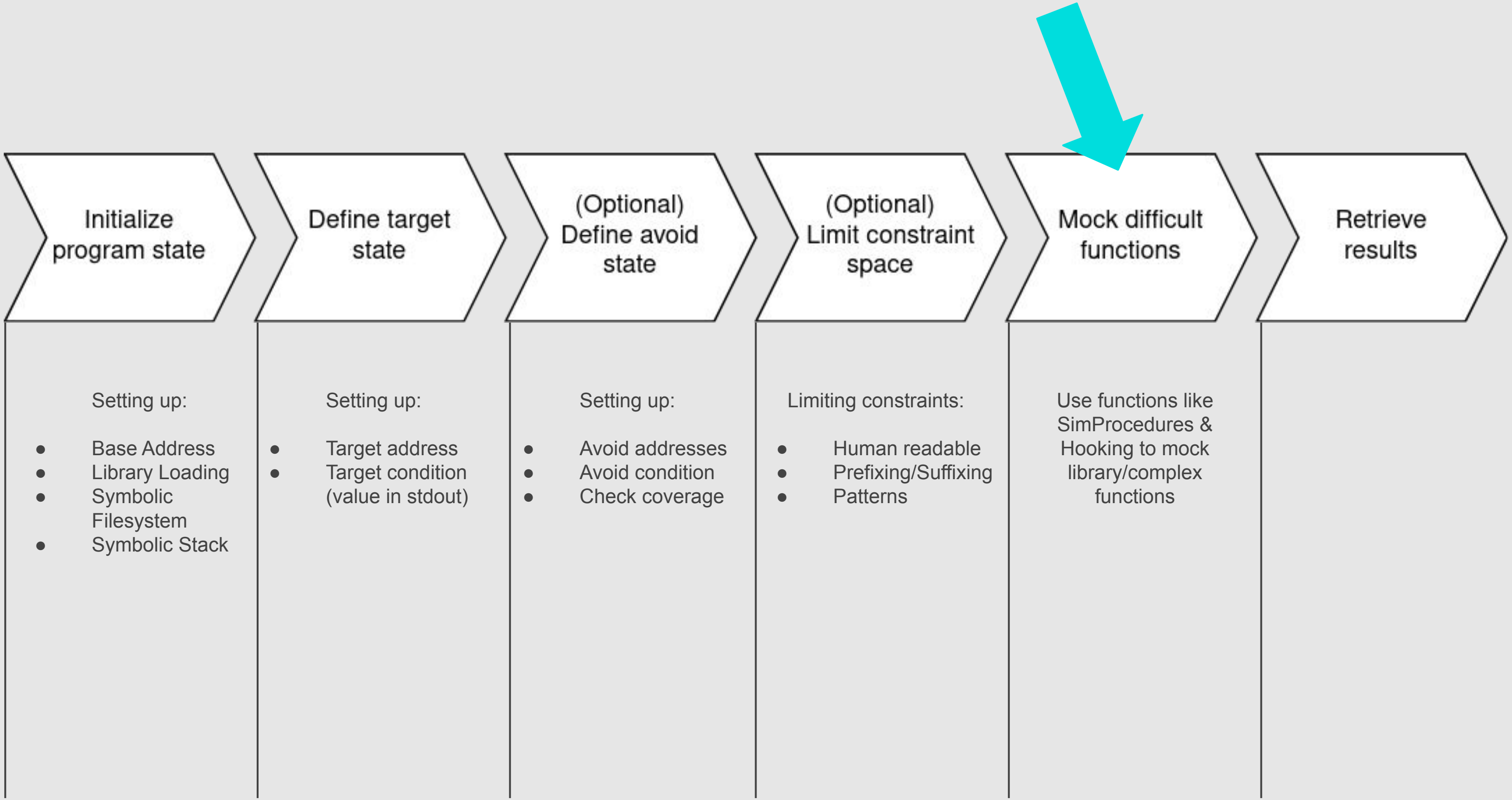
Create new symbolic password bitvector

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))
```


Initialize program state
Define target state
(Optional) Define avoid state
(Optional) Limit constraint space
Mock difficult functions
Retrieve results
Setting up:
Base Address
Library Loading
Symbolic Filesystem
Symbolic Stack
Setting up:
Target address
Target condition (value in stdout)
Setting up:
Avoid addresses
Avoid condition
Check coverage
Limiting constraints:
Human readable
Prefixing/Suffixing
Patterns
Use functions like SimProcedures & Hooking to mock library/complex functions

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

```python
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

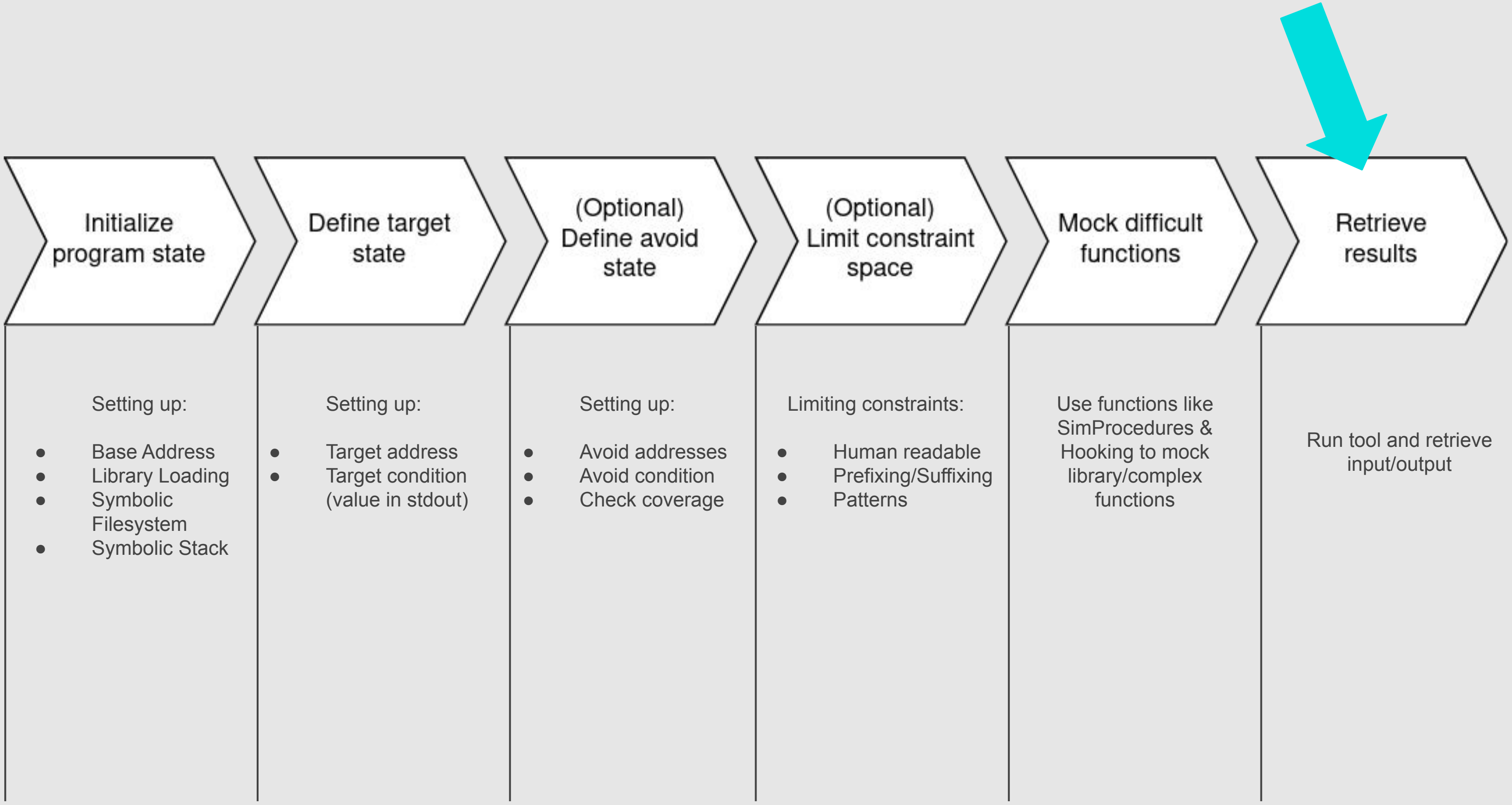
```
# length determines how many bytes get skipped/overwritten
@proj.hook(0x1337, length=5)
def set_rax(state):
  state.regs.rax = 1
```

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

```python
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))
```



Initialize program state
Define target state
(Optional) Define avoid state
(Optional) Limit constraint space
Mock difficult functions
Retrieve results
Setting up:
Base Address
Library Loading
Symbolic Filesystem
Symbolic Stack
Setting up:
Target address
Target condition (value in stdout)
Setting up:
Avoid addresses
Avoid condition
Check coverage
Limiting constraints:
Human readable
Prefixing/Suffixing
Patterns
Use functions like SimProcedures & Hooking to mock library/complex functions
Run tool and retrieve input/output

Concretizing results

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

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

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

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


IM IN UR FOLDUR
KERUPTIN YR FYLEZ

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

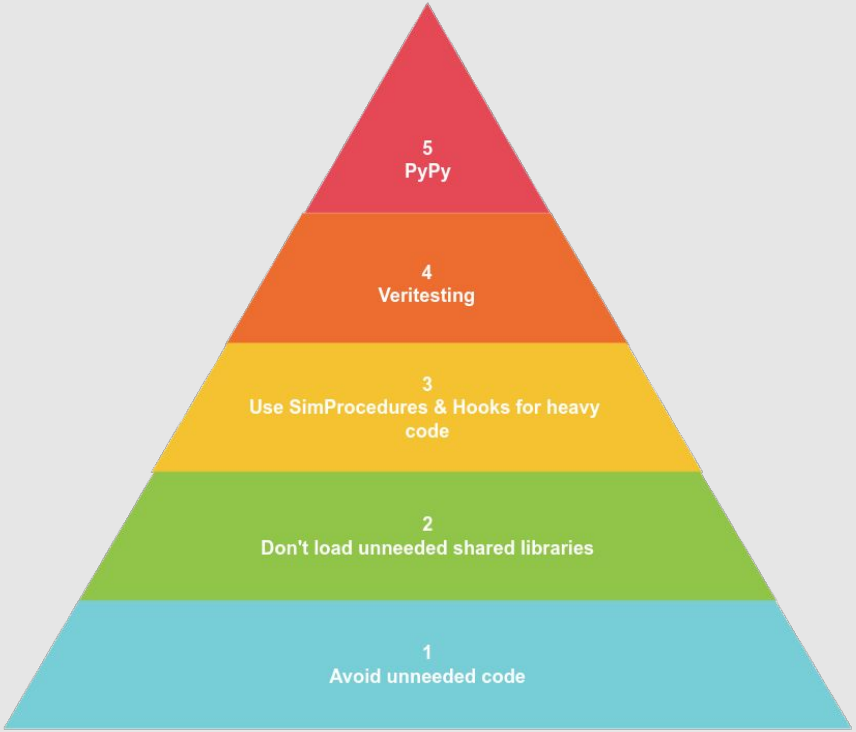
Relies on heuristics to merge states

```python
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

Profiling results
45 sec.
40 sec.
35 sec.
30 sec.
25 sec.
20 sec.
15 sec.
10 sec.
5 sec.
0 sec.
Time to solve
real 0m4.500s
user 0m3.781s
sys 0m0.453s
real 0m7.803s
user 0m6.844s
sys 0m0.750s
real 0m5.277s
user 0m4.531s
sys 0m0.625s
real 0m22.509s
user 0m9.734s
sys 0m1.344s
real 0m5.743s
user 0m5.141s
sys 0m0.359s
real 0m11.643s
user 0m10.656s
sys 0m0.688s
Easy (pathfinder)
Medium (10_angr_simprocedures)
Complex (01_angr_avoid)
Sample complexity
cpython
pypy

Maslow’s Hierarchy of Symbolic Execution



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

Now is your turn!

Solve the challenges and decipher the mystery of a strange distress signal!

github.com/JannisKirschner/REcon-Workshop-Public

@xorkiwi

/in/janniskirschner