Klee and Angr

bananaappletw @ UCCU Hacker

2017/10/29

$whoami

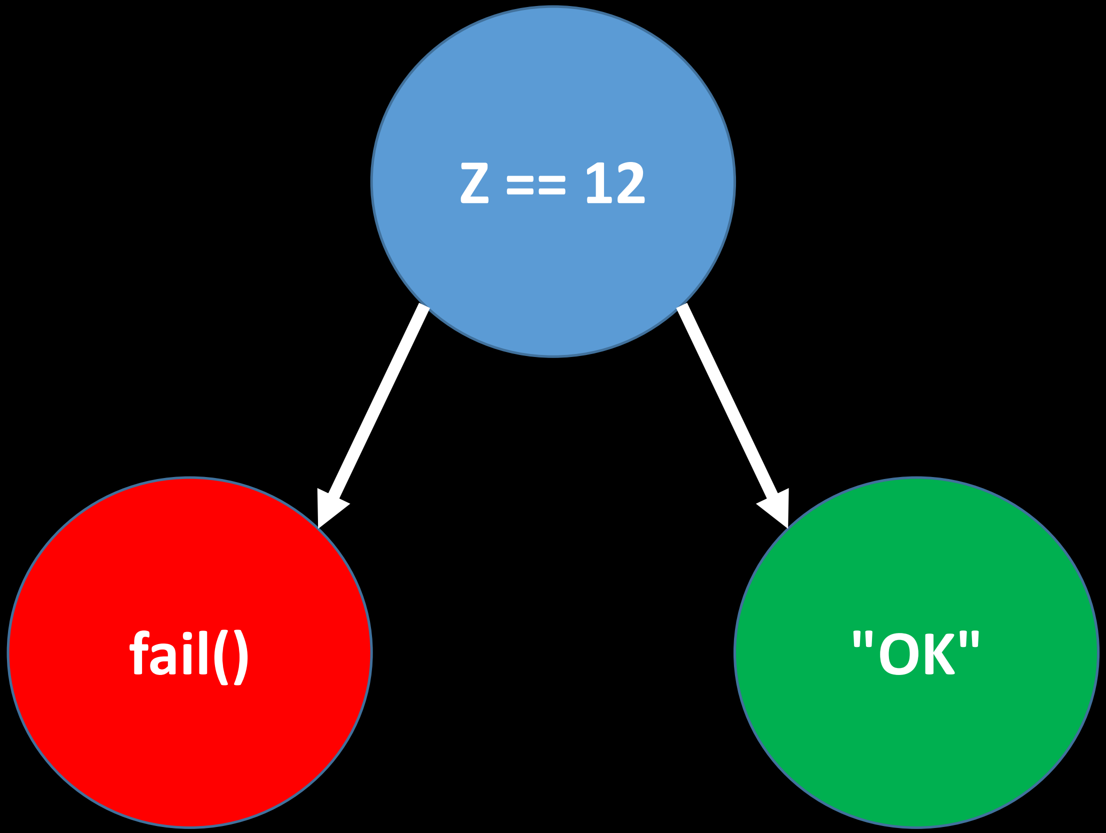
- 陳威伯(bananaappletw)
- Master of National Chiao Tung University
- Organizations:
  - Software Quality Laboratory
  - Bamboofox member
  - Vice president of NCTUCSC
- Specialize in:
  - symbolic execution
  - binary exploit
- Talks:
  - HITCON CMT 2015
  - HITCON CMT 2017

What is symbolic execution?

- Symbolic execution is a means of analyzing a program to determine what inputs cause each part of a program to execute
- System-level
  - S2e(https://github.com/dslab-epfl/s2e)
- User-level
  - Angr(http://angr.io/)
  - Triton(https://triton.quarkslab.com/)
- Code-based
  - klee(http://klee.github.io/)

Symbolic execution

```
int f() {
  ...
  y = read();
  z = y * 2;
  if (z == 12) {
    fail();
  } else {
    printf("OK");
  }
}
```



Klee

- Symbolic virtual machine built on top of the LLVM compiler infrastructure
- Website: http://klee.github.io/
- Github: https://github.com/klee/klee
- Klee paper: http://llvm.org/pubs/2008-12-OSDI-KLEE.pdf (Worth reading)
- Main goal of Klee:
    1. Hit every line of executable code in the program
    2. Detect at each dangerous operation

Installation

- Install docker
  - sudo apt-get install docker.io
- Get klee image
  - sudo docker pull klee/klee
- Run klee docker image
  - sudo docker run --rm -ti --ulimit='stack=-1:-1' klee/klee
- Test environment
  - Ubuntu 17.04 desktop amd64

Introduction

- Klee is a symbolic machine to generate test cases
- Need source code to compile to LLVM bitcode
- Steps:
  - Replace input with Klee function to make memory region symbolic
  - Compile source code to LLVM bitcode
  - Run Klee
  - Get the test cases and path's information

Steps

- Source code at ~/klee_src/examples/get_sign.c
- Include klee library

```
#include <klee/klee.h>
```

- Make input symbolic

```
klee_make_symbolic(&a, sizeof(a), "a");
```

- Compile to LLVM bitcode

```
clang -I ../../include -emit-llvm -c -g get_sign.c
```

Steps

- Examine LLVM bitcode

llvm-dis get_sign.bc

- Running KLEE

klee get_sign.bc

- Show information about test case

ktest-tool ./klee-last/*.ktest

get_sign.c

```c
#include <klee/klee.h>

int get_sign(int x) {
  if (x == 0)
     return 0;

  if (x < 0)
     return -1;
  else
     return 1;
}

int main() {
  int a;
  klee_make_symbolic(&a, sizeof(a), "a");
  return get_sign(a);
}
```

get_sign.ll

```
define i32 @main() #0 {
  %1 = alloca i32, align 4
  %a = alloca i32, align 4
  store i32 0, i32* %1
  call void @llvm.dbg.declare(metadata !{i32*
%a}, metadata !25), !dbg !26
  %2 = bitcast i32* %a to i8*, !dbg !27
  call void @klee_make_symbolic(i8* %2, i64 4,
i8* getelementptr inbounds ([2 x i8]* @.str,
i32 0, i32 0)), !dbg !27
  %3 = load i32* %a, align 4, !dbg !28
  %4 = call i32 @get_sign(i32 %3), !dbg !28
  ret i32 %4, !dbg !28
}
```

Result

```
klee@561b436ff126:~/klee_src/examples/get_sign$ klee get_sign.bc
KLEE: output directory is "/home/klee/klee_src/examples/get_sign/klee-out-3"
KLEE: Using STP solver backend

KLEE: done: total instructions = 31
KLEE: done: completed paths = 3
KLEE: done: generated tests = 3
klee@561b436ff126:~/klee_src/examples/get_sign$ ktest-tool ./klee-last/*.ktest
ktest file : './klee-last/test000001.ktest'
args       : ['get_sign.bc']
num objects: 1
object    0: name: b'a'
object    0: size: 4
object    0: data: b'\x00\x00\x00\x00'

ktest file : './klee-last/test000002.ktest'
args       : ['get_sign.bc']
num objects: 1
object    0: name: b'a'
object    0: size: 4
object    0: data: b'\x01\x01\x01\x01'

ktest file : './klee-last/test000003.ktest'
args       : ['get_sign.bc']
num objects: 1
object    0: name: b'a'
object    0: size: 4
object    0: data: b'\x00\x00\x00\x80'
```

Functions

- klee_make_symbolic(address, size, name)

Make memory address symbolic

klee_make_symbolic(&c, sizeof(c), "c");

Notice: Don't overlap symbolic memory regions

- klee_assume(condition)

Add condition constraint on input

klee_assume((c==2)||(d==3));

- klee_prefer_cex(object, condition)

When generating test cases, prefer certain values between equivalent test cases

klee_prefer_cex(input, 32 <= input[i] && input[i] <= 126);

Functions

- Insert assert on target path

klee_assert(0)

- Find assert test case

ls ./klee-last/ | grep .assert

- Show information about test case

ktest-tool ./klee-last/*.ktest

- Using GNU c library functions

Run with --libc=uclibc --posix-runtime parameters

Environment Modeling

- For example:
  - command-line arguments
  - environment variables
  - file
  - data and metadata
  - network packets
  - ……
- Klee redirects library call to models
- If fd refers to a concrete file

Klee use pread to multiplex access from KLEE's many states onto the one actual underlying file descriptor

- If fd refers to a symbolic file

read() copies from the underlying symbolic buffer

Environment Modeling

- -sym-arg <N>

Replace by a symbolic argument with length N

- -sym-args <MIN> <MAX> <N>

Replace by at least MIN arguments and at most MAX arguments, each with maximum length N

- -sym-files <NUM> <N>

Make NUM symbolic files (‘A’, ‘B’, ‘C’, etc.), each with size N (excluding stdin)

- -sym-stdin <N>

Make stdin symbolic with size N

Solver

- MetaSMT
- STP(Default option for Klee)
- Z3

Architecture

- Represent Klee as an interpreter loop which select a state to run and then symbolically executes a single instruction in the context of that state
- When Klee meets the conditional branch instruction, it clones the state so that it could explore both paths
- By implementing heap as immutable map, portions of the heap structure itself can also be shared amongst multiple states(copy on write)

Architecture

- Potentially dangerous operations implicitly generate branches that check if any input value exists that could cause an error(e.g., zero divisor)
- When processing instruction, if all given operands are concrete, performs the operation natively, returning a constant expression
- When KLEE detects an error or when a path reaches an exit call, Klee solves the current path's constraints

Diagram

1. Step the program until it meets the branch

```c
#include <klee/klee.h>

int get_sign(int x) {
  if (x == 0)
     return 0;

  if (x < 0)
     return -1;
  else
     return 1;
}

int main() {
  int a;
  klee_make_symbolic(&a, sizeof(a), "a");
  return get_sign(a);
}
```

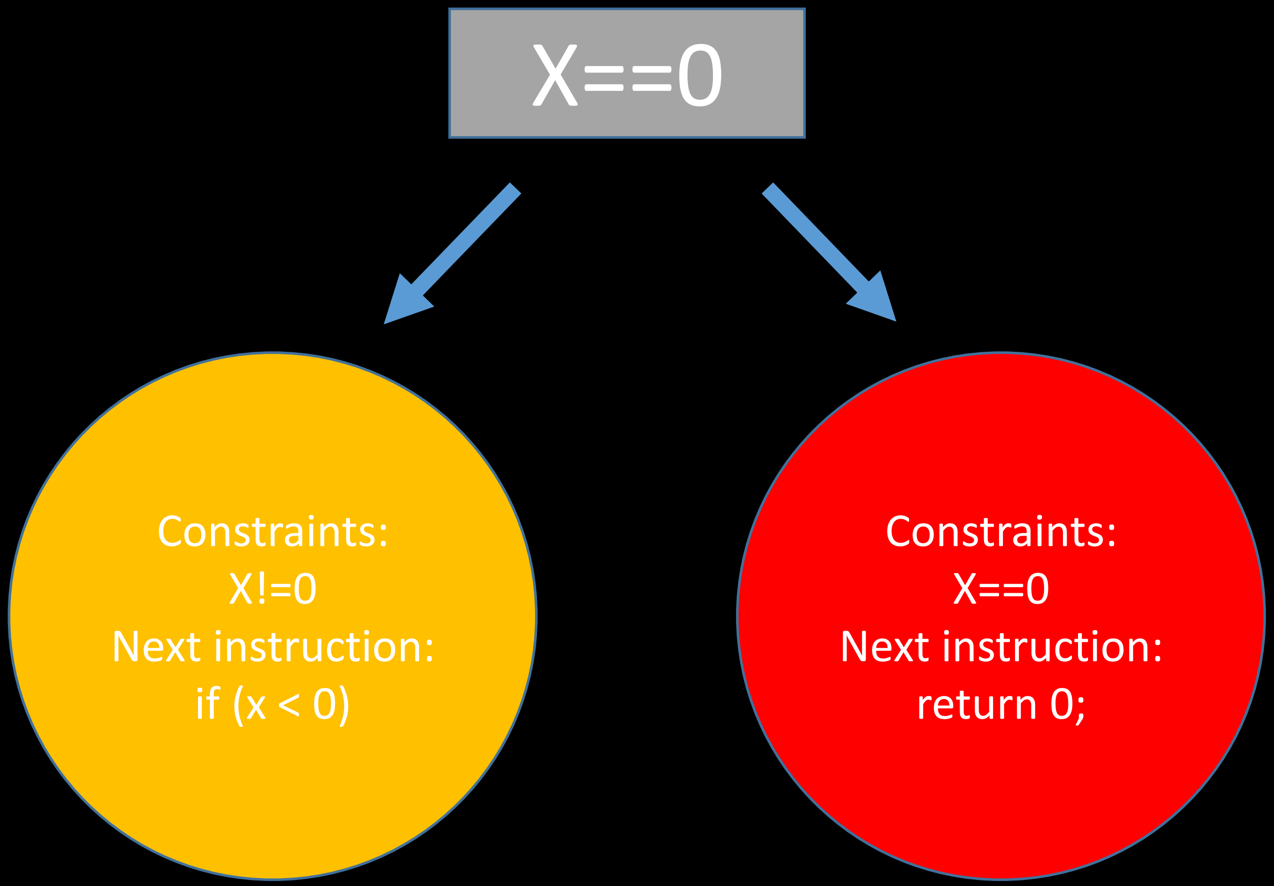
Diagram

1. Step the program until it meets the branch
2. If all given operands are concrete, return constant expression. If not, record current condition constraints and clone the state.

```
#include <klee/klee.h>

int get_sign(int x) {
  if (x == 0)
     return 0;

  if (x < 0)
     return -1;
  else
     return 1;
}

int main() {
  int a;
  klee_make_symbolic(&a, sizeof(a), "a");
  return get_sign(a);
}
```

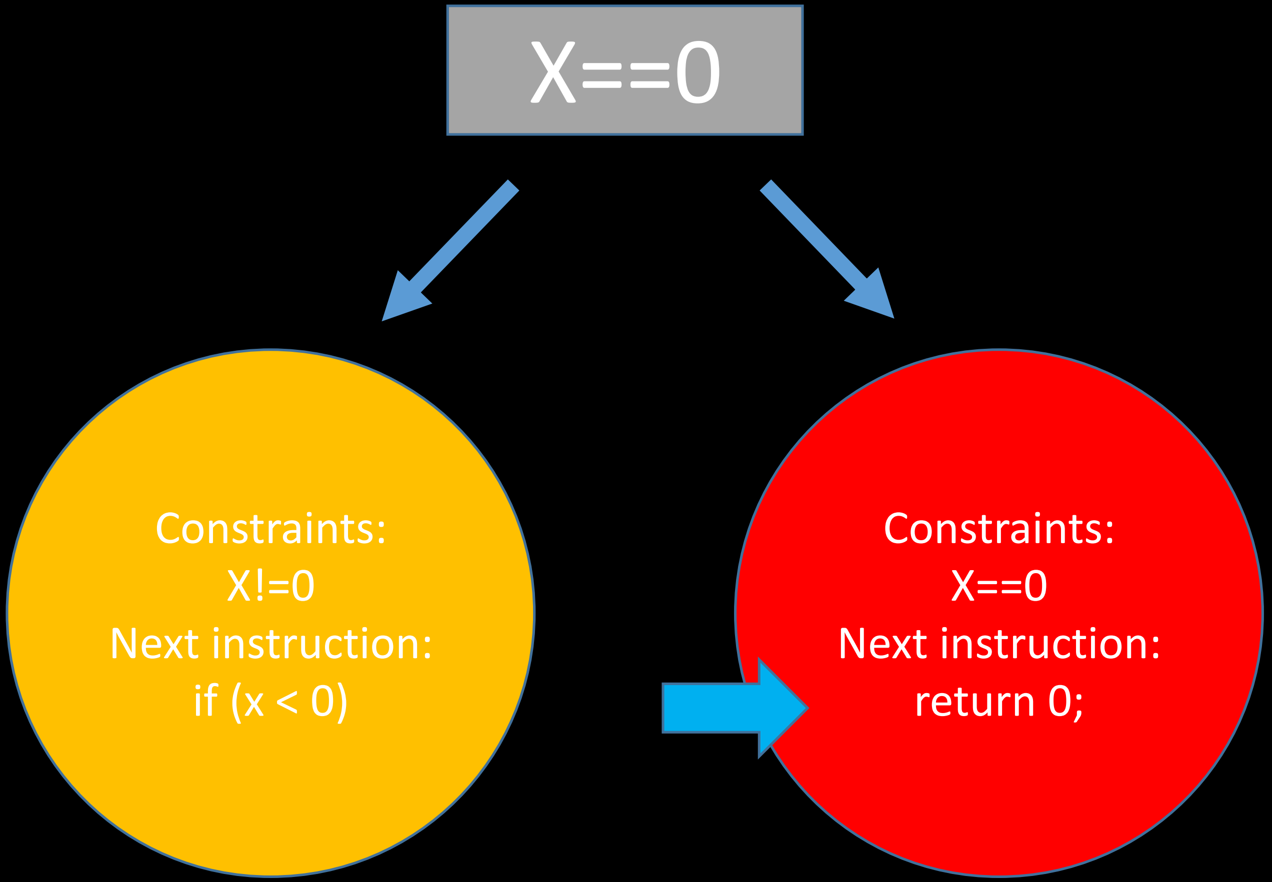
Diagram

1. Step the program until it meets the branch
2. If all given operands are concrete, return constant expression. If not, record current condition constraints and clone the state
3. Step the states until they hit exit call or error



Diagram

1. Step the program until it meets the branch
2. If all given operands are concrete, return constant expression. If not, record current condition constraints and clone the state
3. Step the states until they hit exit call or error
4. Solve the conditional constraint



Diagram

1. Step the program until it meets the branch
2. If all given operands are concrete, return constant expression. If not, record current condition constraints and clone the state
3. Step the states until they hit exit call or error
4. Solve the conditional constraint
5. Loop until no remaining states or user-defined timeout is reached

Exercises

- ccr

Sum of continuous number

https://bamboofox.cs.nctu.edu.tw/courses/1/challenges/67

- Suicide

Product of continuous number

https://bamboofox.cs.nctu.edu.tw/courses/1/challenges/68

Angr

- Website: http://angr.io/
- Angr is a python framework for analyzing binaries. It combines both static and dynamic symbolic ("concolic") analysis, making it applicable to a variety of tasks.
- Flow
  - Loading a binary into the analysis program.
  - Translating a binary into an intermediate representation (IR).
  - Performing the actual analysis

Installation

- sudo apt-get install -y python-pip python-dev libffi-dev build-essential virtualenvwrapper
- Virtualenv
  - mkvirtualenv angr && pip install angr
- Direct install in system
  - sudo pip install angr
- Test environment
  - Ubuntu 17.04 desktop amd64

Flow

- Import angr

```
import angr
```

- Load the binary and initialize angr project

```
project = angr.Project('./ais3_crackme')
```

- Define argv1 as 100 bytes bitvectors

```
argv1 = claripy.BVS("argv1",100*8)
```

- Initialize the state with argv1

```
state = project.factory.entry_state(args=["./crackme1",argv1])
```

Flow

- Initialize the simulation manager

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

- Explore the states that matches the condition

```
simgr.explore(find= 0x400602)
```

- Extract one state from found states

```
found = simgr.found[0]
```

- Solve the expression with solver

```
solution = found.solver.eval(argv1, cast_to=str)
```

ais3 crackme

- Binary could be found in: https://github.com/angr/angr-doc/blob/master/examples/ais3_crackme/
- Run binary with argument
- If argument is correct

print "Correct! that is the secret key!"

- else

print "I'm sorry, that's the wrong secret key!"

Target address

```
.text:00000000004005EB loc_4005EB:                             ; CODE XREF: main+13↑j
.text:00000000004005EB                 mov     rax, [rbp+var_10]
.text:00000000004005EF                 add     rax, 8
.text:00000000004005F3                 mov     rax, [rax]
.text:00000000004005F6                 mov     rdi, rax
.text:00000000004005F9                 call    verify
.text:00000000004005FE                 test    eax, eax
.text:0000000000400600                 jz      short loc_40060E
.text:0000000000400602                 mov     edi, offset aCorrectThatIsT ; "Correct! that is the secret key!"
.text:0000000000400607                 call    _puts
.text:000000000040060C                 jmp     short loc_400618
.text:000000000040060E ; ---------------------------------------------------------------------------
.text:000000000040060E
.text:000000000040060E loc_40060E:                             ; CODE XREF: main+3B↑j
.text:000000000040060E                 mov     edi, offset aIMSorryThatSTh ; "I'm sorry, that's the wrong secret key!"
```

Solution

```
import angr
import claripy
project = angr.Project("./ais3_crackme")
argv1 = claripy.BVS("argv1",100*8)
state = project.factory.entry_state(args=["./crackme1",argv1])
simgr = project.factory.simgr(state)
simgr.explore(find=0x400602)
found = simgr.found[0]
solution = found.solver.eval(argv1, cast_to=str)
print(repr(solution))
```

Result

```
apple  ~  angr-doc  examples  ais3_crackme  python solve.py
'ais3{I_tak3_g00d_n0t3s}\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00'
```

Loader

- Load the binary through angr.Project
- Loading options
  - auto_load_libs
    - When there is no SimProcedure
    - True(Default, real library function is executed)
    - False(return a generic "stub" SimProcedure called ReturnUnconstrained)

Intermediate Representation

- In order to be able to analyze and execute machine code from different CPU architectures, Angr performs most of its analysis on an intermediate representation
- Angr's intermediate representation is **VEX**(Valgrind), since the uplifting of binary code into VEX is quite well supported

Intermediate Representation

- IR abstracts away several architecture differences when dealing with different architectures
  - Register names: VEX models the registers as a separate memory space, with integer offsets
  - Memory access: The IR abstracts difference between architectures access memory in different ways
  - Memory segmentation: Some architectures support memory segmentation through the use of special segment registers
  - Instruction side-effects: Most instructions have side-effects

Intermediate Representation

- addl %eax, %ebx

- t3 = GET:I32(0)
- # get %eax, a 32-bit integer
- t2 = GET:I32(12)
- # get %ebx, a 32-bit integer
- t1 = Add32(t3,t2)
- # addl
- PUT(0) = t1
- # put %eax

Simulation Managers

- simgr.step()

Step forward all states in a given stash by one basic block

- simgr.run()

Step until everything terminates

- simgr.explore(conditions)
  - find, avoid:
    - An address to find or avoid
    - A set or list of addresses to find or avoid
    - A function that takes a state and returns whether or not it matches.

Bit-Vector Symbol

- claripy.BVS(name, size)
  - name: The name of the symbol.
  - size: The size (in bits) of the bit-vector.
- chop(bits)
  - bits: A list of smaller bitvectors, each bits in length.

State

- Property
  - registers: The state's register file as a flat memory region
  - memory: The state's memory as a flat memory region
  - solver(e.g., se): The solver engine for this state
  - posix: information about the operating system or environment model(e.g., posix.files[fd])
- add_constraints(BVS condition)
- inspect.b(event, when=angr.BP_AFTER, action=debug_func)
  - event: mem_read, reg_write
  - when: BP_BEFORE or BP_AFTER
  - action: debug_func

SimMemory

- load(addr, size=None)
  - addr: memory address
  - size: The sizein bytes
- find(addr, what)
  - addr: The start address
  - what: what to search for
- store(addr, data, size=None)
  - addr: address to store at
  - data: The data(claripy expression or something convertable to a claripy expression)
  - size: The data size(claripy expression or something convertable to a claripy expression)

Stash types

active	**This stash contains the states that will be stepped by default, unless an alternate stash is specified.**
deadended	A state goes to the deadended stash when it cannot continue the execution for some reason, including no more valid instructions, unsat state of all of its successors, or an invalid instruction pointer.
pruned	When using LAZY_SOLVES, states are not checked for satisfiability unless absolutely necessary. When a state is found to be unsat in the presence of LAZY_SOLVES, the state hierarchy is traversed to identify when, in its history, it initially became unsat. All states that are descendants of that point (which will also be unsat, since a state cannot become un-unsat) are pruned and put in this stash.
unconstrained	If the save_unconstrained option is provided to the SimulationManager constructor, states that are determined to be unconstrained (i.e., with the instruction pointer controlled by user data or some other source of symbolic data) are placed here.
unsat	If the save_unsat option is provided to the SimulationManager constructor, states that are determined to be unsatisfiable (i.e., they have constraints that are contradictory, like the input having to be both "AAAA" and "BBBB" at the same time) are placed here.

Hooking

- project.hook(address, hook)
- hook is a SimProcedure instance
- At every step, angr checks if the current address has been hooked, and if so, runs the hook instead of the binary code at that address
- You could also use symbol to locate the address
- proj.hook_symbol(name, hook)
- You could use hook as function decorator, length is to jump after finishing the hook

```
@proj.hook(0x20000, length=5)
def my_hook(state):
       state.regs.rax = 1
```

Hooking

- proj.is_hooked(address)

True or False

- proj.unhook(address)

Unhook hook on address

- proj.hooked_by(address)

Return SimProcedure instance

Symbolic Function

- Project tries to replace external calls to library functions by using symbolic summaries termed **SimProcedures**
- Because SimProcedures are library hooks written in Python, it has inaccuracy
- If you encounter path explosion or inaccuracy, you can do:
  1. Disable the SimProcedure
  2. Replace the SimProcedure with something written directly to the situation in question
  3. Fix the SimProcedure

Symbolic Function(scanf)

- Source code: https://github.com/angr/angr/blob/master/angr/procedures/libc/scanf.py

1. Get first argument(pointer to format string)
2. Define function return type by the architecture
3. Parse format string
4. According format string, read input from file descriptor 0(i.e., standard input)
5. Do the read operation

Symbolic Function(scanf)

```
from angr.procedures.stubs.format_parser import FormatParser
from angr.sim_type import SimTypeInt, SimTypeString
class scanf(FormatParser):
    def run(self, fmt):
        self.argument_types = {0: self.ty_ptr(SimTypeString())}
        self.return_type = SimTypeInt(self.state.arch.bits, True)
        fmt_str = self._parse(0)
        f = self.state.posix.get_file(0)
        region = f.content
        start = f.pos
        (end, items) = fmt_str.interpret(start, 1, self.arg, region=region)
        # do the read, correcting the internal file position and logging the action
        self.state.posix.read_from(0, end - start)
        return items
```

Symbolic Function(scanf)

```
class SimProcedure(object):
        @staticmethod
        def ty_ptr(self, ty):
         return SimTypePointer(self.arch, ty)

class FormatParser(SimProcedure):
        def _parse(self, fmt_idx):
                """
                fmt_idx: The index of the (pointer to the) format string in the
arguments list.
                """


def interpret(self, addr, startpos, args, region=None):
    """
    Interpret a format string, reading the data at `addr` in `region` into `args`
starting at `startpos`.
    """
```

def _parse(self, fmt_idx):

- int scanf (const char * format, ...);
- scanf ("%d",&i);

fmt_str = self._parse(0)

- int sscanf (const char * s, const char * format, ...);
- sscanf (sentence,"%s %*s %d",str,&i);

fmt_str = self._parse(1)

def interpret(self, addr, startpos, args, region=None):

- int scanf (const char * format, ...);
- scanf ("%d",&i);

f = self.state.posix.get_file(0)

region = f.content

start = f.pos

(end, items) = fmt_str.interpret(start, 1, self.arg, region=region)

- int sscanf (const char * s, const char * format, ...);
- sscanf (sentence,"%s %*s %d",str,&i);

_, items = fmt_str.interpret(self.arg(0), 2, self.arg, region=self.state.memory)

Exercises

- angrman

https://bamboofox.cs.nctu.edu.tw/courses/1/challenges/69

Local binary

- magicpuzzle

Server response the base64-encoded binary, try to find out the differences, and automatically generate the exploit

https://bamboofox.cs.nctu.edu.tw/courses/1/challenges/70

- Sushi

Server response the base64-encoded binary, try to find out the differences, and automatically generate the exploit

https://bamboofox.cs.nctu.edu.tw/courses/1/challenges/72

Q & A