http://10.8.217.27:8000/
pigz -d angr-workshop.tar.gz
docker load -i ./angr-workshop.tar

angr - the workshop*

*Not to be confused with angr, the emotion

Who we are

Alexander Druffel

Automated Program Analysis @ Fraunhofer SIT

Florian Magin

Security Research @ ERNW Research

CTF Players @ Wizards of Dos, CCC Darmstadt, Germany

@Alexeyan2

@0x464D

Agenda

- Introduction & Motivation
- Concepts
  - Theory
  - Guided Hands-on
  - Optional challenge Exercise
- Integrating angr with other tools

Introduction and Motivation

Angr: Binary Analysis Framework

Binary Analysis

- Disassembly
- CFG/Function/Variable Recovery
- Emulation

Framework

- Doesn’t solve problems on its own
- Needs scripts or tools that use it

Pro

- Open source
  - BSD 2-Clause "Simplified" License
- Python
  - # Yay, easy code
- Well documented
  - devs available via Slack

Con

- Academic focus
  - PoC/Prototype > High Performance
- Python
  - # Nay, performance, no threads
- Strong focus on automation
  - Lack of features for interactive use

angr in action (you will see this again)

```
>>> import angr

>>> project = angr.Project("./defcamp_quals_2015_r100", auto_load_libs=False) # False = Use Hooks
>>> e = project.factory.entry_state()   # State at entry point with registers set up
>>> simgr = project.factory.simulation_manager(e)

>>> simgr.explore(find=lambda state: b'Nice!' in state.posix.dumps(1)) # Explore until we win
# Takes a few moments
>>> print(simgr)
<simgr with 1 found and 24 deadended> # 25 states finished execution, 1 satisfies our find-func
>>> print(simgr.found[0].posix.dumps(0)) # For the found state, solve a string for fd0 (stdin)
b"Code_Talkers"

$ ./defcamp_quals_2015_r100
Enter the password: Code_Talkers
Nice!
```

Workshop Spirit

- angr hides a lot of theory and concepts behind simple API calls
- Our aim:
  - Explain the concepts
  - Basic API Introduction
  - Show our custom plugin to make interactive use and understanding easier
- We will focus on symbolic execution, ...
  - And all the features required for and related to this
- … not so much on static analysis
  - Value-Range/Value-Set Analysis
  - Reaching Definition Analysis

Problem #1
Guided Exploration

Following all branches doesn't scale
→ $O(2^n)$

What is "interesting" or "relevant"?

There is an API, but intuition is (often) faster

Naive use might just get your script killed by the OOM Manager, or takes until the heat death of the universe

Usage: ./challenge <password>

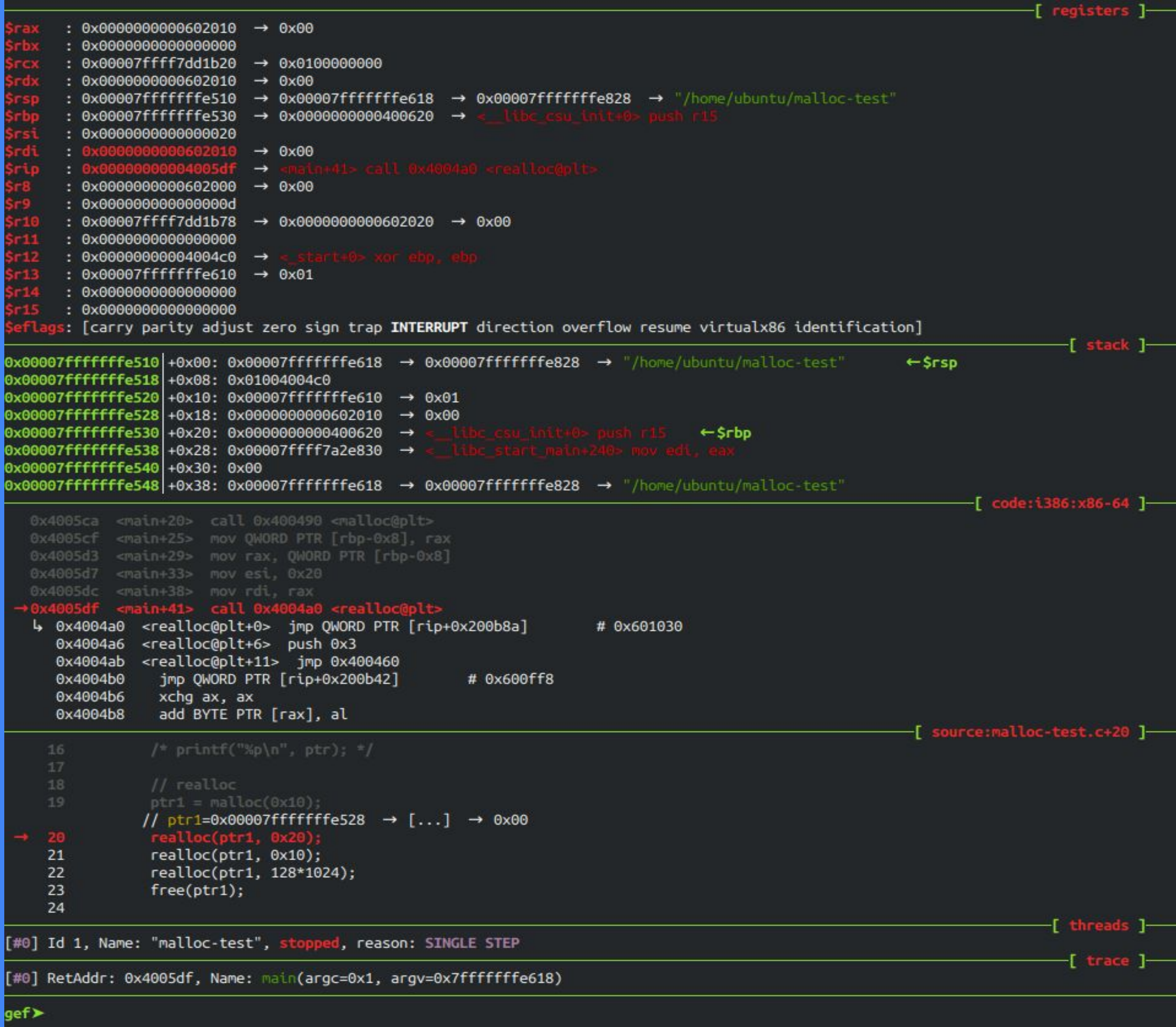
```
if( strlen(argv[1]) == 23)) {
    [...]
} else {
    [...]
}
```

Which branch is more interesting? Why?

→ Reversing experience.
Intersting paths often "more constraints"

Problem #2
Situational Awareness

- What are we looking for?
- Where are you right now?
- In what state is the program?
- API exists, but no overview
- That problem sounds familiar...
- plain gdb → Plugins



Context View Plugin

- What PEDA/pwndbg/gef is to gdb
- Render relevant information about a program state in a human digestable format

```
LEGEND: SYMBOLIC | UNINITIALIZED | STACK | HEAP | CODE R-X | DATA R*- | RWX | RODATA
[ ──────────────────────────────────────────────────────────────────── Registers ── ]
RAX:     0x0
RBX:     <BV64 reg_rbx_3_64{UNINITIALIZED}>
RCX:     0x555555554c78 <PLT.rand+0x4e0 in morph (0xc78)>
RDX:     0x2f5
RSI:     0x555555554c78 <PLT.rand+0x4e0 in morph (0xc78)>
RDI:     0x7fffffffffeffb8 ──> <BV184 argv1_0_184>
RBP:     0x7fffffffffeff40 ──> 0x0
RSP:     0x7fffffffffeff08 ──> 0x555555554b12 <PLT.rand+0x37a in morph (0xb12)>
RIP:     0x555555554987 <PLT.rand+0x1ef in morph (0x987)>
R8:      0xffffffff
R9:      0x0
R10:     <BV64 reg_r10_11_64{UNINITIALIZED}>
R11:     <BV64 reg_r11_12_64{UNINITIALIZED}>
R12:     <BV64 reg_r12_13_64{UNINITIALIZED}>
R13:     <BV64 reg_r13_14_64{UNINITIALIZED}>
R14:     <BV64 reg_r14_15_64{UNINITIALIZED}>
R15:     <BV64 reg_r15_16_64{UNINITIALIZED}>
[ ───────────────────────────────────────────────────────────────────────── Code ── ]
PLT.rand+0x370 in morph (0xb08)
0x555555554b08: mov     eax, 0
0x555555554b0d: call    0x555555554987

        |       0x1
        v
PLT.rand+0x1ef in morph (0x987)
0x555555554987: push    rbp
0x555555554988: mov     rbp, rsp
0x55555555498b: sub     rsp, 0x20
0x55555555498f: mov     edi, 0
0x555555554994: call    0x555555554778

[ ──────────────────────────────────────────────────────────────────────── Stack ── ]
00:0x00| sp 0x7fffffffffeff08  ──> 0x555555554b12 <PLT.rand+0x37a in morph (0xb12)>
01:0x08|    0x7fffffffffeff10  ──> 0x7fffffffffeff70 ──> 0x7fffffffffeffb0 ──> 0x6870726f6d2f2e
02:0x10|    0x7fffffffffeff18  ──> <BV64 Reverse(Reverse(reg_rbx_3_64{UNINITIALIZED})[63:32] .. 0x2000000)>
03:0x18|    0x7fffffffffeff20  ──> 0x0
04:0x20|    0x7fffffffffeff28  ──> 0xc0080000 ──> 0x850f333c078a5256
05:0x28|    0x7fffffffffeff30  ──> <BV64 reg_r13_14_64{UNINITIALIZED}>
06:0x30|    0x7fffffffffeff38  ──> <BV64 reg_r14_15_64{UNINITIALIZED}>
07:0x38| bp 0x7fffffffffeff40  ──> 0x0
[ ──────────────────────────────────────────────────────────────────── BackTrace ── ]
Frame 0: PLT.rand+0x370 in morph (0xb08) => PLT.rand+0x1ef in morph (0x987), sp = 0x7fffffffffeff08
Frame 1: __libc_start_main.after_init+0x0 in extern-address space (0x98) => PLT.rand+0x2de in morph (0xa76), sp = 0x7fffffffffeff48
Frame 2: PLT.rand+0x8 in morph (0x7a0) => __libc_start_main+0x0 in extern-address space (0x18), sp = 0x7fffffffffeff58
Frame 3: 0x0 => 0x0, sp = 0xffffffffffffffff
[ ────────────────────────────────────────────────────────────────────── Watches ── ]
argv[1]:        b'\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xfd\xff'

0:       IP: 0xc008000c ──> 0x9b4347000002b8e9    Cond: <Bool argv1_0_184[183:176] == 51>
         Vars: frozenset({'argv1_0_184'})

1:       IP: 0xc00802e7 ──> 0x1bf0000003cb8       Cond: <Bool !(argv1_0_184[183:176] == 51)>
         Vars: frozenset({'argv1_0_184'})
```

Example #1 - “m0rph”

Reverse Engineering Challenge from 34c3ctf

Basic Reverse Engineering:

- Usage: `$ ./challenge [flag]`
- `strings`: correct flag → probably congratulation-string in stdout
- `readelf -s`: time() and rand(), possibly for randomization in verification?

Example #1 - morph

Validation function for each character, decrypted at runtime.

Randomized verifiaction order → Side-channel resistant

`libc::rand()` in angr → return unconstrained int → state explosion

Solution: Hook rand and make it return a constant

→ Deterministic runs

DEMO TIME

Fallback: Video of example

First Handson

http://10.8.217.27:8080/helper_scripts/start_jupyter.sh

http://10.8.217.27:8080/scaffold.py

-
- Open the provided Jupyter Notebook/Script
- Open any binary you are interested in
    - Maybe something you are already familiar with
    - /bin/ls

Concept:
Intermediate Representation

Intermediate Representation: Theory

- Convert Instructions into common generalized semantics language

x86

```
mov eax, 0x1234
mov ebx, 0x4242
mov [ebx], eax
```

arm

```
mov r0, 0x1234
mov r1, 0x4242
str r0, [r1]
```

```
T0 = 0x1234
Regs[0] = T0
T1 = 0x4242
Regs[1] = T1
Mem[T1] = T0
```

Intermediate Representation: Theory

- Representation is more general, typically some “Intermediate Language” is used
  - data could be represented by an internal data structure which isn’t formally specified
- Can be optimized for various purposes
  - Human readability
  - Machine readability
  - Complexity of analysis (Single Static Assignment Form)
  - Representation of some high level source (C Code)
  - Representation of some low level source (machine Code)

Intermediate Representation: Terms

- Lifting
  - Generating some IR from a “lower” level, typically machine code
- Single Static Assignment/SSA
  - Every temporary variable only ever gets assigned once
  - Makes Data Flow Analysis less complex

Example: LLVM IR

- Typically generated from some high level source
    - C Code

```
; Function Attrs: noinline nounwind optnone
sspstrong uwtable

define dso_local i32 @add(i32, i32) #0 {
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  store i32 %0, i32* %3, align 4
  store i32 %1, i32* %4, align 4
  %5 = load i32, i32* %3, align 4
  %6 = load i32, i32* %4, align 4
  %7 = mul nsw i32 %5, %6
  ret i32 %7
}
```

Example: Binary Ninja ILs

- Generated from machine code
- Various levels
  - Lifted
  - Low
  - Medium
  - High (at some point)
- Each level also has an SSA form available
- Lifted is generated from machine code
- Every other level is generated from the level below
  - Analysis is performed to simplify

```
add:
eax = edi
eax = eax * esi
<return> jump(pop)
```

```
add:
uint64_t rax_1 = zx.q(arg1.edi)
uint64_t rax = zx.q(rax1.eax * arg2.esi)
return rax
```

Example: VEX

- **angr uses this**
- Originally developed for Valgrind
  - Not an IR optimized for RE and human readability!
- Assembly → IR
- Lifting supported for:

x86, AMD64, ARMv7, ARMv8, MIPS32, MIPS64, PPC32 & PPC64

also: brainfuck, avr (sort of), … whatever

```
IRSB {
   t0:Ity_I32 t1:Ity_I32 t2:Ity_I32 t3:Ity_I64 t4:Ity_I64 t5:Ity_I64 t6:Ity_I64 t7:Ity_I32 t8:Ity_I64
t9:Ity_I32 t10:Ity_I64 t11:Ity_I32 t12:Ity_I64 t13:Ity_I64 t14:Ity_I64 t15:Ity_I64 t16:Ity_I64
t17:Ity_I64

   00 | ------ IMark(0x401120, 2, 0) ------
   01 | t8 = GET:I64(rdi)
   02 | t7 = 64to32(t8)
   03 | t6 = 32Uto64(t7)
   04 | ------ IMark(0x401122, 3, 0) ------
   05 | t9 = 64to32(t6)
   06 | t12 = GET:I64(rsi)
   07 | t11 = 64to32(t12)
   08 | PUT(cc_op) = 0x0000000000000033
   09 | t13 = 32Uto64(t11)
   10 | PUT(cc_dep1) = t13
   11 | t14 = 32Uto64(t9)
   12 | PUT(cc_dep2) = t14
   13 | t2 = Mul32(t11,t9)
   14 | t15 = 32Uto64(t2)
   15 | PUT(rax) = t15
   16 | PUT(rip) = 0x0000000000401125
   17 | ------ IMark(0x401125, 1, 0) ------
   18 | t3 = GET:I64(rsp)
   19 | t4 = LDle:I64(t3)
   20 | t5 = Add64(t3,0x0000000000000008)
   21 | PUT(rsp) = t5
   22 | t16 = Sub64(t5,0x0000000000000080)
   23 | ====== AbiHint(0xt16, 128, t4) ======
   NEXT: PUT(rip) = t4; Ijk_Ret
```

Example: Ailment

- Also used by angr
- Fairly new
- angr decompiler is based on this

Concept: Symbolic Execution

Introduction
Symbolic Execution

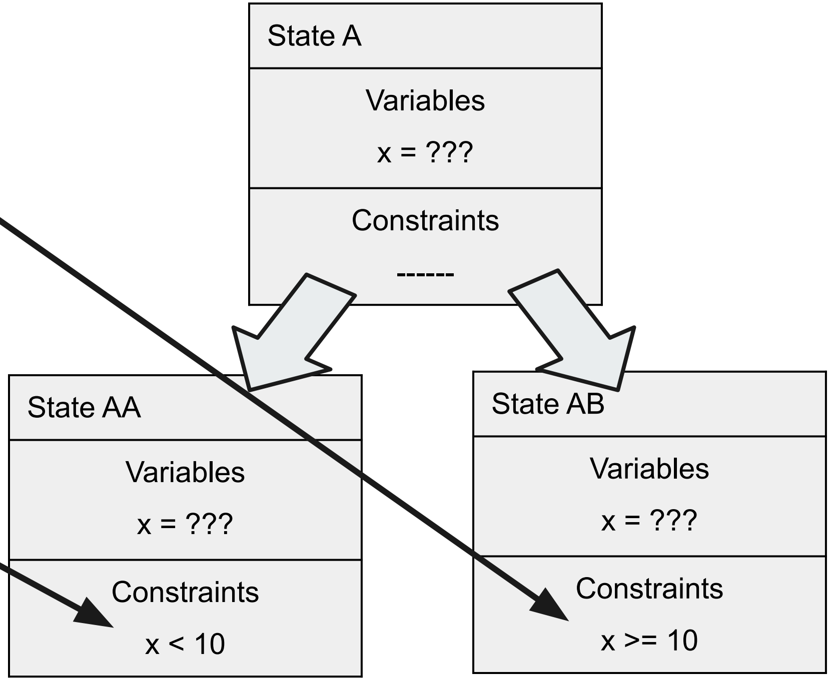
~~Execution~~ → Simulation

```python
x = int(input())
if x >= 10:
    if x < 100:
        print "You win!"
    else:
        print "You lose!"
else:
    print "You lose!"
```

State A
Variables x = ???
Constraints ------

Slide from angr Tutorial
at SecDev

```
x = int(input())
if x >= 10:
    if x < 100:
        print "You win!"
    else:
        print "You lose!"
else:
    print "You lose!"
```



Slide from angr Tutorial at SecDev

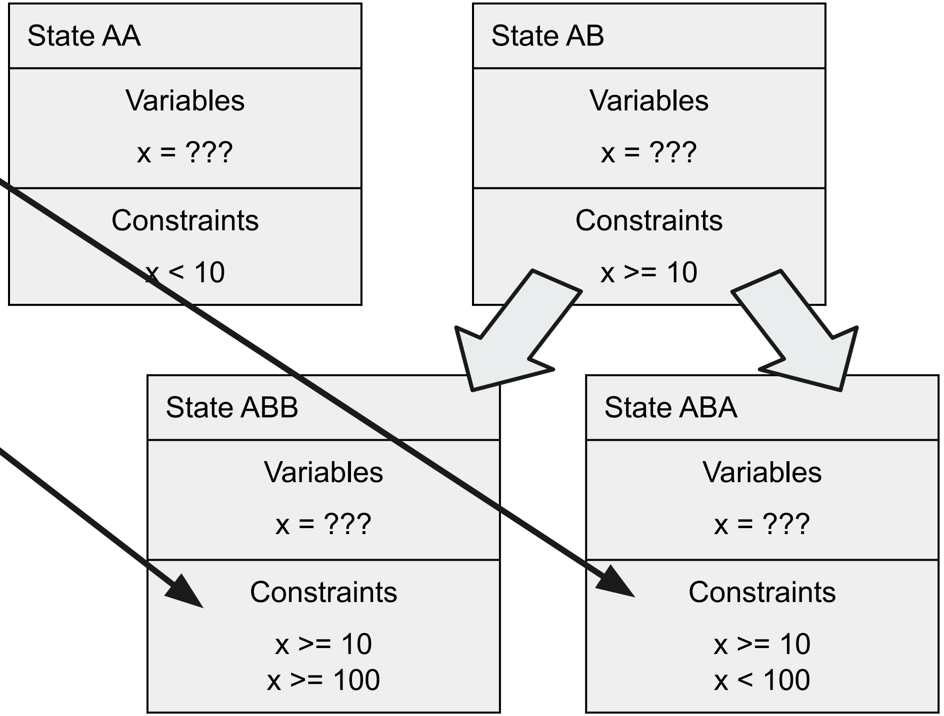
```
x = int(input())
if x >= 10:
    if x < 100:
        print "You win!"
    else:
        print "You lose!"
else:
    print "You lose!"
```

State AA
Variables x = ???
Constraints x < 10

State AB
Variables x = ???
Constraints x >= 10

Slide from angr Tutorial
at SecDev

```
x = int(input())
if x >= 10:
    if x < 100:
        print "You win!"
    else:
        print "You lose!"
else:
    print "You lose!"
```



Slide from angr Tutorial
at SecDev

```
x = int(input())
if x >= 10:
    if x < 100:
        print "You win!"
    else:
        print "You lose!"
else:
    print "You lose!"
```

State ABA

Variables

x = ???

Constraints

x >= 10
x < 100

Concretized ABA

Variables

x = 99

Slide from angr Tutorial
at SecDev

Intermediate Representation

Implement Symbolic Execution Engine over the IR

Symbolic Execution

Symbolically execute independent of architecture :)

angr in action (again)

```
>>> import angr

>>> project = angr.Project("./defcamp_quals_2015_r100", auto_load_libs=False) # False = Use Hooks
>>> e = project.factory.entry_state()   # State at entry point with registers set up
>>> simgr = project.factory.simulation_manager(e)

>>> simgr.explore(find=lambda state: b'Nice!' in state.posix.dumps(1)) # Explore until we win
# Takes a few moments
>>> print(simgr)
<simgr with 1 found and 24 deadended> # 25 states finished execution, 1 satisfies our find-func
>>> print(simgr.found[0].posix.dumps(0)) # For the found state, solve a string for fd0 (stdin)
b"Code_Talkers"

$ ./defcamp_quals_2015_r100
Enter the password: Code_Talkers
Nice!
```

Handson: Sym Exec

- Open `sym_exec.elf`

Concept: SMT Solving

~~Concept: SMT Solving~~

Blackbox!
Formulas / Constraints $\rightarrow$ Concrete Values
Following Slides oversimplify

Concept: SMT Solving

- Magic black box
  - That is abstracted away behind a nice Python API
- Like SAT Solving but smarter
- Intelligent algorithms and theorems for certain domains and problem classes
  - Depends on the solver and the available theories

Concept: SMT Solving

- Will SMT Solving work* for the following examples?
  - Bunch of equations that you could solve in your head but can’t be bothered:
    - Yes
  - System of floating point equations:
    - Usually yes, might take a minute
  - CRC:
    - Depends
  - Cryptographically secure hash functions:
    - <u>**No!**</u>
- Issue: Will it take all the years or 30 minutes to solve?
  - Both is basically “forever” in a CTF

* “work” == “terminate in a reasonably short time”

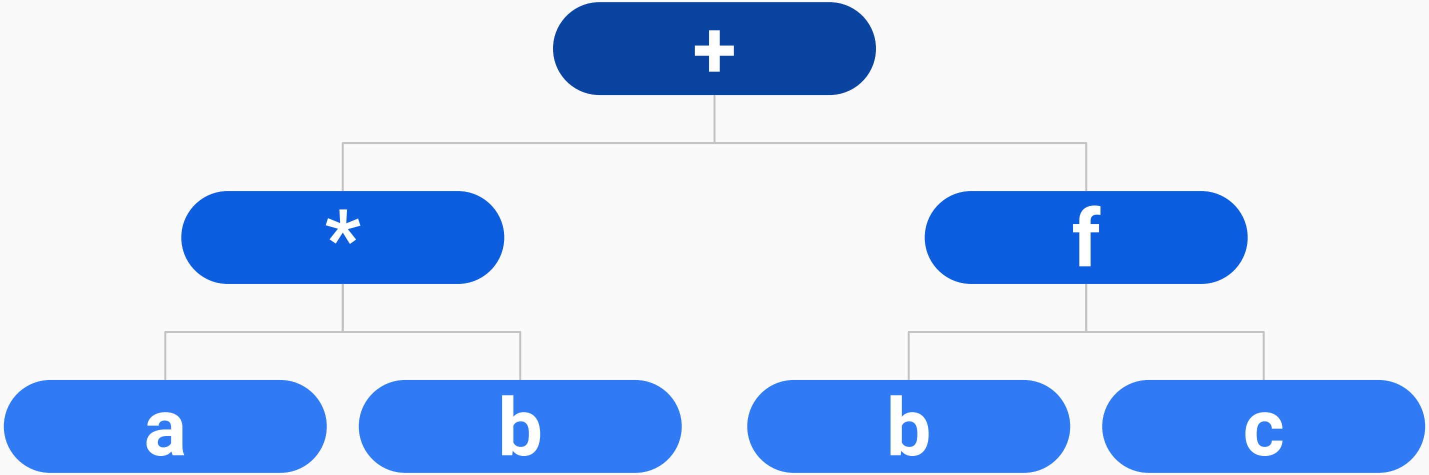
Concept: Abstract Syntax Trees

Concept: Abstract Syntax Trees

- Widely applicable concept
- Can represent:
  - Programs (a form of IR)
  - Mathematical formulas
- Similar but not the same as parse trees
  - Lacks braces, whitespace, commas, ...
- Used to represent constraints and equations in angr

A Really Simple Example: AST of "1 + 1"

Slightly More Complex: AST of “(a * b) + (f(b,c))”



ASTs in angr: Bitvector Type

- Every Leaf Node is a Bitvector (BV)
- A BV is a sequence of bits of length N
- A BV of length 8 can represent a byte
- Can be
  - Concrete (BVV)
  - Symbolic (BVS)
- Most arithmetic operations (+,-,*,/,...)
- Bitwise Operations (|, &, ^,...)
- Additional operations
  - Concat
  - Extract

Handson: ASTs

- Open the notebook
- Running it should give you some graphs
- Play around with various expressions/functions and observe the produced ASTs
- Use

Concept: Memory View

Concept: Memory View

- Powerful abstraction over memory
- Never again deal with the horror that is GDBs interface
- Data can be interpreted as a type (even custom structs) and returns Python object
  - Bitvector
  - Integer
  - String
  - Named Tuple for Structs
  - Another View (pointer deref)
- Can be arbitrarily chained

Handson: Memory View

Concept: SimProcedures

Concept: SimProcedures

- Problem: Some functions/code might be too complicated too symbolically execute
- Imagine atoi with symbolic input
- Solution: Replace that code with an abstraction written in Python

Concept: auto_load_libs

angr
@angrdothorse

Follow

angr tip: no matter what setting to which you set auto_load_libs, you're wrong!

12:34 PM - 29 Sep 2018

https://twitter.com/angrdothorse/status/1046121169068281856

angr
@angrdothorse

Follow

auto_load_libs=True? oops, your program is trying to interact with its environment in ways that you can't easily stub out and you'll get a state explosion

angr
@angrdothorse

Follow

auto_load_libs=False? oops, your program depends on its environment in ways that cant be modeled via externs and your emulation is wrong

Concept: SimProcedures

- Our recommendation for interactive use: Set it to FALSE (NOT the default)
- CFG generation is a lot faster and still mostly correct
- You will see the behaviour of each SimProc during interactive use
- Library Procedures that are not recognized get a ReturnUnconstrained
    - Enough for pure functions
    - If the procedure should return a pointer to an initialized struct things will break
        - Unconstrained memory access is bad, mmkay

Writing SimProcedures

- Extensive Doc at https://docs.angr.io/extending-angr/simprocedures
- The idea:
  - Either a class or a simple Function
  - Can access the state object directly and modify it as needed
  - Abstractions for calling conventions (arguments and returns)

Handson: SimProcedures

Concept: SimulationManager

Simulation Manager

```
>>> simgr.explore(find=0x400789, avoid=[0x400859, 0x400456])

# sym-exec all states, move every state that reaches an address in avoid into the
‘avoid’ stash and ignore them and exit if you find a state that reaches find

>>> simgr.explore(find=lambda state: b”You win!” in state.posix.dumps(1),
        avoid=lambda state: b”Wrong!” in state.posix.dumps(1))

# Sym-Exec all states, apply the function in find on every step and break if res=1
```

Simulation Manager

- Collection of States, divided into stashes
- Functions to execute states
- Symbolic Execution of state
  - One {Instruction|Basic Block} further
  - Branch → Create new states
  - Unsat? → Move to ‘unsat’ stash
  - Program exited? → Move to ‘deadended’
  - …
- Other execution forms possible
  - Concrete (unicorn accelerated)

```
>>> e = proj.factory.entry_state()
>>> simgr = proj.factory.simgr(e)

>>> print(simgr.stashes)
{‘active’: [<SimState @ 0x400234],
‘stashed’: [],
‘pruned’: [],
‘unsat’: [],
‘errored’: [],
‘deadended’: [],
‘unconstrained’: []}
```

Concept: Exploration Techniques

Concept: Exploration Techniques

- Search algorithm/heuristic for exploring the state space
    - Directed Graph
- Default is all active states are stepped on each tick: “Breadth-First-Search”
- Alternatives:
    - Limit Loop Iterations
    - Follow List of CFG-Blocks
    - Depth-First-Search
    - Order by Memory Usage
- Writing one is not fully documented :(
- But the concept is fairly simple and can be derived from existing Techniques
- Often referred to as “otie” by devs

Integration

Integration

- Angr is a framework
- Build angr into some other tool
  - Enriching that tool with information gathered by angr's analysis
- Build angr around some other tool
  - Gather information from your tool to make angr more powerful

Example 1: Enhanced CFG Analysis

- Issue: The program is obfuscated and contains something like:
  - Indirect jumps to derail the disassembler
  - Opaque Predicates
- CFGEmulated (or even CFGFast) will detect and solve both
  - UNSAT successors aren’t part of the graph
- Outline:
  - Generate a CFGEmulated of the function (or the program if you are patient)
  - Diff the graphs from $tool and angr
  - Mark every block that is not in the angr CFG (unreachable) in the $tool
    - Warning: false negatives are possible, correct indirect jumps could be found by $tool.
  - More complicated: Find $tool’s API to edit the CFG directly

Example 2: Enhanced Interactive Use

- If other tool is better in recovering symbols or you have already renamed variables in functions in a disassembler
- Use the API of $tool to disassemble a line or basic block and return that string
- mov [rax + 0x123456] , 0x41414141 => mov [symbolname.str], “AAAA”
- Our tool has an abstraction for that
    - Currently Ghidra “works” (DEMO?)

Example 3: Enhanced Fuzzing

- Use symbolic execution to find new inputs for paths if a fuzzer is stuck
- Feed the inputs back into the fuzzer
- Project: “Driller”
  - https://sites.cs.ucsb.edu/~vigna/publications/2016_NDSS_Driller.pdf
  - Based on angr and AFL and was used in the DARPA CGC

The Future:

- Daily commits, very active project
- Experimental Features:
    - Java & Android support → load an apk, sym-exec with .dex and .so
    - Decompiler
    - [Your idea here]
    - Concrete + Symbolic execution → execute concrete up to point, then switch to symbolic