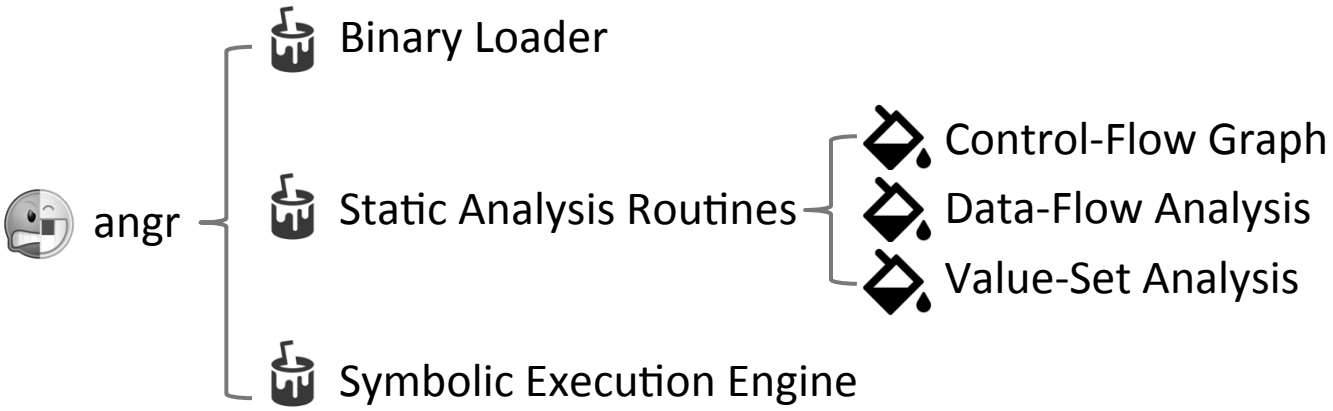
# ANGR

# ANGR

- **Angr, an binary analysis and symbolic framework**




Shellphish, "How angr pwned CTFs and the CGC," DEFCON 2015, https://docs.google.com/presentation/d/1t7KaCMc73z7WdV7EcL0z9TSHlT_kjdMdSrPHtpA6ezc/edit#slide=id.g5b8e5606f_0_0.

# PROJECT

- The first step to use angr is creating the project

```
In [1]: import angr
In [2]: b = angr.Project("./a.out")
```

# CODE BLOCK

- Angr takes code block as basic analysis unit

- Definition of Code Block
  - A linear sequence of code, with one entry point, one exit point and no jump instructions contained within it
- Property
  - Without considering interrupt, the instructions in the same code block will executing in order

# CODE BLOCK AND ASSEMBLY

```
In [133]: block = b.factory.block(b.entry)
In [134]: block
Out[134]: <Block for 0x400430, 41 bytes>

In [141]: block = b.factory.block(0x400526)

In [142]: block
Out[142]: <Block for 0x400526, 21 bytes>
```

```
400430:       31 ed                   xor    ebp,ebp
400432:       49 89 d1                mov    r9,rdx
400435:       5e                      pop    rsi
400436:       48 89 e2                mov    rdx,rsp
400439:       48 83 e4 f0             and    rsp,0xfffffffffffffff0
40043d:       50                      push   rax
40043e:       54                      push   rsp
40043f:       49 c7 c0 c0 05 40 00    mov    r8,0x4005c0
400446:       48 c7 c1 50 05 40 00    mov    rcx,0x400550
40044d:       48 c7 c7 26 05 40 00    mov    rdi,0x400526
400454:       e8 b7 ff ff ff          call   400410 <__libc_start_main@plt>
```

# ANGR DISASSEMBLE

- With the help of capstone, Angr can disassemble more accessibility

```
In [144]: b.factory.block(b.entry).pp()
0x400430:       xor     ebp, ebp
0x400432:       mov     r9, rdx
0x400435:       pop     rsi
0x400436:       mov     rdx, rsp
0x400439:       and     rsp, 0xfffffffffffffff0
0x40043d:       push    rax
0x40043e:       push    rsp
0x40043f:       mov     r8, 0x4005c0
0x400446:       mov     rcx, 0x400550
0x40044d:       mov     rdi, 0x400526
0x400454:       call    0x400410
```

# CAPSTONE INSTRUCTION

- Capstone is the modern disassembler
- Get every instruction through insns

```
In [155]: insns = b.factory.block(b.entry).capstone.insns

In [156]: insns
Out[156]:
[<CapstoneInsn "xor" for 0x400430>,
 <CapstoneInsn "mov" for 0x400432>,
 <CapstoneInsn "pop" for 0x400435>,
 <CapstoneInsn "mov" for 0x400436>,
 <CapstoneInsn "and" for 0x400439>,
 <CapstoneInsn "push" for 0x40043d>,
 <CapstoneInsn "push" for 0x40043e>,
 <CapstoneInsn "mov" for 0x40043f>,
 <CapstoneInsn "mov" for 0x400446>,
 <CapstoneInsn "mov" for 0x40044d>,
 <CapstoneInsn "call" for 0x400454>]
```

# VEX

- VEX IR
  - First used by Valgrind
  - Machine independent intermediate representation

- In angr, different ISA instruction is translated to VEX IR
  - Instruction's side effects
  - Cross platform
  - Instruction semantics

```
In [164]: b.factory.block(b.entry)
Out[164]: <Block for 0x400430, 41 bytes>

In [165]: b.factory.block(b.entry).vex
Out[165]: <pyvex.block.IRSB at 0x7efd560462c0>
```

# INTERMEDIATE REPRESENTATION

- IR code is fully typed
  - All variables and results of calculations have type
  - No implicit type conversion
  - VEX performs sanity checks on these types all of the time
- IR code is in a Single Static Assignment form
  - Each variable is assigned only once
  - Simplifies the instrumentation of the code
  - Also Simplifies the optimization of the code when running it
- IR code is presented to the tools in semi-parsed form
  - Easy to manipulate lists of instructions
  - Instructions are presented in a convenient data-structure
  - Useful functions for manipulating IR code (add/remove instructions, etc.)

```
In [166]: irsb = b.factory.block(b.entry).vex

In [167]: irsb.pp()
IRSB {
   t0:Ity_I32 t1:Ity_I32 t2:Ity_I32 t3:Ity_I64 t4:Ity_I64 t5:Ity_I64 t6:Ity_I64 t7:Ity_I64

   00 | ------ IMark(0x400430, 2, 0) ------
   01 | PUT(bp) = 0x0000000000000000
   02 | ------ IMark(0x400432, 3, 0) ------
   03 | t21 = GET:I64(rdx)
   04 | PUT(r9) = t21
   05 | PUT(pc) = 0x0000000000400435
   06 | ------ IMark(0x400435, 1, 0) ------
   07 | t4 = GET:I64(rsp)
   08 | t3 = LDle:I64(t4)
   09 | t22 = Add64(t4,0x0000000000000008)
   10 | PUT(rsi) = t3
   11 | ------ IMark(0x400436, 3, 0) ------
   12 | PUT(rdx) = t22
   13 | ------ IMark(0x400439, 4, 0) ------
   14 | t5 = And64(t22,0xfffffffffffffff0)
   15 | PUT(cc_op) = 0x0000000000000014
   16 | PUT(cc_dep1) = t5
   17 | PUT(cc_dep2) = 0x0000000000000000
   18 | PUT(pc) = 0x000000000040043d
   19 | ------ IMark(0x40043d, 1, 0) ------
   20 | t8 = GET:I64(rax)
   21 | t24 = Sub64(t5,0x0000000000000008)
   22 | PUT(rsp) = t24
   23 | STle(t24) = t8
   24 | PUT(pc) = 0x000000000040043e
```

# PRACTICE: IR

Compare IRSB and asm

```
0x400430:       xor     ebp, ebp
   00 | ------ IMark(0x400430, 2, 0) ------
   01 | PUT(bp) = 0x0000000000000000

0x400432:       mov     r9, rdx
   02 | ------ IMark(0x400432, 3, 0) ------
   03 | t21 = GET:I64(rdx)
   04 | PUT(r9) = t21
   05 | PUT(pc) = 0x0000000000400435
```

# CLARIPY — SOLVER ENGINE

- Angr uses Claripy as solver engine
- Claripy ASTs (the subclasses of claripy.ast.Base) provide a unified way to interact with concrete and symbolic expressions
- Claripy frontends provide a unified interface to expression resolution (including constraint solving) over different backends
  - Most widely used one – z3

# SOLVER BACKEND

| Name | Description |
|---|---|
| Solver | This is analogous to a z3.Solver(). It is a solver that tracks constraints on symbolic variables and uses a constraint solver (currently, Z3) to evaluate symbolic expressions. |
| SolverVSA | This solver uses VSA to reason about values. It is an approximating solver, but produces values without performing actual constraint solves. |
| SolverReplacement | This solver acts as a pass-through to a child solver, allowing the replacement of expressions on-the-fly. It is used as a helper by other solvers and can be used directly to implement exotic analyses. |
| SolverHybrid | This solver combines the SolverReplacement and the Solver (VSA and Z3) to allow for approximating values. You can specify whether or not you want an exact result from your evaluations, and this solver does the rest. |
| SolverComposite | This solver implements optimizations that solve smaller sets of constraints to speed up constraint solving. |

# CLASIPY – SAMPLE CODE

```
In [179]: s = claripy.Solver()

In [180]: x = claripy.BVS('x', 8)
     ...:

In [181]: s.add(claripy.ULT(x, 5))
Out[181]: (<Bool x_41_8 < 5>,)

In [182]: s.eval(x, 10)
Out[182]: (2L, 0L, 3L, 1L, 4L)

In [183]: s.eval(x, 1)
Out[183]: (0L,)

In [184]: s.eval(x, 3)
Out[184]: (2, 0L, 3L)
```

# CLARIPY DATA TYPE

- Claripy supports different data types
  - BV - bit vector
    - BVV - 32-bit bitvector with the value `0xc001b3475`: `claripy.BVV(0xc001b3a75, 32)`
    - BVS - 32-bit symbolic bitvector "x": `claripy.BVS('x', 32)`
  - FP - floating-point number
    - FPV
    - FPS
  - Bool - boolean
    - BoolV - claripy.BoolV(True)
    - true
    - False
    - BoolS - b2 = claripy.BoolS( 'b' )

# SYMBOLIC EXPRESSION

- Angr use the class AST to represent a symbolic expresseion
  - https://docs.angr.io/docs/claripy.html
- API and members
  - op
  - args
  - variables
  - symbolic

| Name | Description | Example |
|---|---|---|
| LShR | Logically shifts a bit expression (BVV, BV, SI) to the right. | claripy.LShR(x, 10) |
| SignExt | Sign-extends a bit expression. | claripy.SignExt(32, x) or x.sign_extend(32) |
| ZeroExt | Zero-extends a bit expression. | claripy.ZeroExt(32, x) or x.zero_extend(32) |
| Extract | Extracts the given bits (zero-indexed from the right, inclusive) from a bit expression. | Extract the rightmost byte of x: claripy.Extract(7, 0, x) or x[7:0] |
| Concat | Concatenates several bit expressions together into a new bit expression. | claripy.Concat(x, y, z) |

# SYMBOLIC EXPRESSION SAMPLE CODE

```
In [228]: bv = claripy.BVV(0x41424344, 32)

In [229]: x = claripy.BVS('x', 32)

In [230]: q = claripy.And(claripy.Or(bv == x, bv * 2 == x, bv * 3 == x), x == 0)

In [231]: print q
<Bool Or((0x41424344 == x_49_32), (0x82848688 == x_49_32), (0xc3c6c9cc == x_49_32)) && (x_4

In [232]: q.op
Out[232]: 'And'

In [233]: q.args
Out[233]:
(<Bool Or((0x41424344 == x_49_32), (0x82848688 == x_49_32), (0xc3c6c9cc == x_49_32))>,
 <Bool x_49_32 == 0x0>)

In [234]: q.variables
Out[234]: frozenset({'x_49_32'})

In [235]: q.symbolic
Out[235]: True
```

# PRACTICE: CLASIPY

Practice with claripy
Simple script to traverse all operands

```
In [286]: s = claripy.Solver()
In [287]: s.add( y < 100)
Out[287]: (<Bool y_55_32 < 0x64>,)
In [288]: s.add( x < 100)
In [289]: s.add( y > 0)
In [290]: s.add( x > 0)
In [291]: s.add( x*y == 256)
In [292]: s.satisfiable()
Out[292]: True

In [293]: s.eval( x, 3)
Out[293]: (8, 64L, 4L)

In [294]: s.eval( x, 10)
Out[294]: (32L, 16L, 8, 64, 4)

In [295]: s.eval( y, 10)
Out[295]: (32, 16, 8, 64, 4)

In [307]: z = claripy.BVS('z', 32)
In [309]: s.add(z==30)

In [310]: s.independent_constraints()
Out[310]:
[({'x_54_32', 'y_55_32'},
  [<Bool !(0x64 <= y_55_32)>,
   <Bool !(0x64 <= x_54_32)>,
   <Bool (x_54_32 * y_55_32) == 0x100>]),
 ({'z_56_32'}, [<Bool z_56_32 == 0x1e>])]
```

# SIMUVEX: EMULATOR FOR VEX IR

- Program Sementic
  - Not only what instructions are executed
  - But how they change the program state
- Angr use SimuVEX to inference program sementic
  - The program will be first translated to VEX IR
  - SimuVEX is the symbolic VEX emulator
- Given a program state and a VEX IR block(IRSB), SimuVEX can produces result machine states
  - SimEngines
  - The number of result machine state may more than 1
- Class Path is high level abstract for controlling execution
  - A sequence of executed IRSB
  - Repeat/infinite

# ANGR PATH

- Make a path at program's entry

```
In [329]: p = angr.Project("../a.out")

In [330]: s = p.factory.path()

In [331]: hex(p.entry)
Out[331]: '0x400430'

In [333]: hex(s.addr)
Out[333]: '0x400430L'
```

# PATH STEP

- Take one step forward
  - 1 block each step()

```
In [355]: hex(s.addr)
Out[355]: '0x4003c8L'

#step return a list of path
In [356]: s= s.step()[0]

In [357]: hex(s.addr)
Out[357]: '0x4003ddL'

In [360]: for addr in s.addr_trace:
     ...:     print hex(addr)
     ...:
0x400430L
0x400410L
0x4000010L
0x400550L
0x4003c8L

In [361]: s.length
Out[361]: 5
```

# MORE SAMPLE CODE

```
In [356]: s= s.step()[0]

In [365]: s.step()
Out[365]: [<Path with 6 runs (at 0x400581 : /home/angr/a.out)>]

In [366]: s.successors
Out[366]: [<Path with 6 runs (at 0x400581 : /home/angr/a.out)>]
```

```
In [372]: for q in s.trace:
     ...:     print q
     ...:
<IRSB from 0x400430: 1 sat>
<IRSB from 0x400410: 1 sat>
<SimProcedure __libc_start_main from 0x4000010: 1 sat>
<IRSB from 0x400550: 1 sat>
<IRSB from 0x4003c8: 1 sat 1 unsat>
```

# JUMP KIND

```
In [369]: for j in s.jumpkinds.hardcopy:
     ...:     print j
     ...:
Ijk_Boring
Ijk_Call
Ijk_Boring
Ijk_Call
Ijk_Call
Ijk_Boring
```

| Type | Description |
|---|---|
| Ijk_Boring | A normal jump to an address. |
| Ijk_Call | A call to an address. |
| Ijk_Ret | A return. |
| Ijk_Sig* | Various signals. |
| Ijk_Sys* | System calls. |
| Ijk_NoHook | A jump out of an angr hook. |

# PRACTICE: PATH

Practice with path

# PATH MERGE

- The path reach to the same program point can be merge…

```
# step until branch
p = b.factory.path()
p.step()
while len(p.successors) == 1:
    print 'step'
    p = p.successors[0]
    p.step()

print p
branched_left = p.successors[0]
branched_right = p.successors[1]
assert branched_left.addr != branched_right.addr

# Step the branches until they converge again
after_branched_left = branched_left.step()[0]
after_branched_right = branched_right.step()[0]
assert after_branched_left.addr == after_branched_right.addr

# this will merge both branches into a single path. Values in memory and registers
# will hold any possible values they could have held in either path.
merged = after_branched_left.merge(after_branched_right)
assert merged.addr == after_branched_left.addr and merged.addr == after_branched_right.addr
```

# PATH UNMERGE

```
merged_successor = merged.step()[0].step()[0]
unmerged_paths = merged_successor.unmerge()

assert len(unmerged_paths) == 2
assert unmerged_paths[0].addr == unmerged_paths[1].addr
```

# PATH GROUP

- As the path can be merged together, the path group can help to manage multiple path.
  - The path group separates paths into different stashes
- API
  - step
  - run
  - deadended
  - active
  - found
  - unsat

# FAIRLIGHT

- Symbolize Input
- Run the program
- Resolve the symbol

```
In [3]: proj = angr.Project('angr-doc/examples/securityfest_fairlight/fairlight', load_opti
In [5]: argv1 = angr.claripy.BVS("argv1", 0xE * 8)
path_group.deadended[-1].state.se.any_str(argv1)
In [6]: initial_state = proj.factory.entry_state(args=["./fairlight", argv1])

In [8]: path_group = proj.factory.path_group(initial_state)

In [9]: path_group.run()

In [10]: path_group.deadended[0].state.se.any_str(argv1)

In [10]: path_group.deadended[1].state.se.any_str(argv1)

In [10]: path_group.deadended[-1].state.se.any_str(argv1)
```

# PRACTICE: PATH GROUP

Practice PG
Use run and check every symbolic input

- Create path group
- And run…
- Check output

# BINARY LOADER

- Angr includes it's own loader component CLE loader.
  - Parse the binary and depended libraries
  - Correctly setup memory layout for angr
  - Support wide range of machine types
  - Once the Project is used, the loader is automatic used

# LOADER BASIC USAGE

```
In [31]: b = angr.Project("/bin/true")

In [32]: print hex(b.entry)
0x4013d0

In [33]: print hex(b.loader.min_addr()),
                hex(b.loader.max_addr())
0x400000 0x5004000
```

```
In [34]: # this is the CLE Loader object

In [35]: print b.loader
<Loaded true, maps [0x400000:0x5004000]>

In [36]: # this is a dictionary of the objects that are loaded
         # as part of loading the binary

In [37]: print b.loader.shared_objects
    ...:
OrderedDict([('true', <ELF Object true, maps [0x400000:0x6063bf]>),
(u'libc.so.6', <ELF Object libc-2.23.so, maps [0x1000000:0x13c899f]>),
(u'ld-linux-x86-64.so.2', <ELF Object ld-2.23.so, maps [0x2000000:0x2227167]>), .
```

```
In [38]: # this is the memory space of the process after being loaded.

In [39]: print b.loader.memory[b.loader.min_addr()]


In [40]: # this is the object for the main binary

In [41]: print b.loader.main_bin
    ...:
<ELF Object true, maps [0x400000:0x6063bf]>
```

```
In [42]: # this retrieves the binary object which maps
         # memory at the specified address
In [44]: print b.loader.addr_belongs_to_object(0x400000)
<ELF Object true, maps [0x400000:0x6063bf]>

# Get the address of the GOT slot for a symbol (in the main binary)
In [46]: print hex(b.loader.find_symbol_got_entry
                        ('__libc_start_main'))
0x6060c0

In [47]: # this is a dict (name->ELFRelocation) of imports

In [48]: b.loader.shared_objects['libc.so.6'].imports
    ...:
Out[48]:
{u'__libc_enable_secure': <cle.backends.relocations.generic.GenericJumpslotReloc
 u'__tls_get_addr': <cle.backends.relocations.generic.GenericJumpslotReloc at 0x7
 u'_dl_argv': <cle.backends.relocations.generic.GenericJumpslotReloc at 0x7f7d0b5
 u'_dl_find_dso_for_object': <cle.backends.relocations.generic.GenericJumpslotRel
 u'_dl_starting_up': <cle.backends.relocations.generic.GenericJumpslotReloc at 0x
 u'_rtld_global': <cle.backends.relocations.generic.GenericJumpslotReloc at 0x7f7
 u'_rtld_global_ro': <cle.backends.relocations.generic.GenericJumpslotReloc at 0x
```

# LOADER OPTIONS

- By default, Angr loads every needed libraries.
- The `auto_load_libs` options can turn it off.

```
b = angr.Project("/bin/true", load_options=dict(auto_load_libs=False))
```

- Options can be set by follow

```
load_options = {'main_opts':{options0},
      'lib_opts': {libname1:{options1},
                   path2:{options2}, ...}}
```

# LOADER BACKEND

| backend key | description | requires custom_arch? |
|---|---|---|
| elf | Static loader for ELF files based on PyELFTools | no |
| pe | Static loader for PE files based on PEFile | no |
| cgc | Static loader for Cyber Grand Challenge binaries | no |
| backedcgc | Static loader for CGC binaries that allows specifying memory and register backers | no |
| elfcore | Static loader for ELF core dumps | no |
| ida | Launches an instance of IDA to parse the file | yes |
| blob | Loads the file into memory as a flat image | yes |

# SYMBOLIC PROGRAM STATE

- Loader can help us to setup initial program state
- Symbolize program argument

```
argv1 = angr.claripy.BVS("argv1", 0xE * 8)
initial_state = proj.factory.entry_state(args=["./fairlight", argv1])
```

# PRACTICE: AIS3 CRACKME

ais3_crackme

# CALL STACK

```
# s is the Path
In [362]: print s.callstack
Backtrace:
Func 0x4003c8, sp=0x7fffffffffeff40, ret=0x400581
Func 0x400550, sp=0x7fffffffffeff80, ret=0x4000020
Func 0x400410, sp=0x7fffffffffeff88, ret=0x400459
Func 0x400430, sp=0x7fffffffffeff98, ret=-0x1
```

# SYMBOLIC LIBRARY MODEL

- Loading all the libraries
  - Sometimes, libraries is unavailable
  - Library may not be transfer to IRs correctly
  - Performance overhead & Symbolic expression explosion

# SYMBOLIC SUMMARY

- Summaries the state change made by the function
  - if auto_load_libs is True -> Origin function
  - if auto_load_libs is False -> Use symbolic summary instead
- Angr provides many symbolic summaries of glibc functions
  - String Manipulation
  - Input/Output
- May not sementic equivelent to the original functions

# POSIX MODEL

- Angr implements many modesl for posix interface.
- Symbolize Stdin
  - state.posix.dumps(0)
  - state.posix.dumps(1)
- Symbolize File

# PROGRAM INPUT

- With the symbolic posix model and environment, we can cover most of inputs.
  - STDIN
  - FILE IO
  - Program Argument
  - Environment

# EXPLORER

- The amazing function provided by path group is PathGroup.Explorer().
- Given the target to find, PathGroup.Explorer() can automatic reach it.
  - For every path in active stash, Explorer steps until target found
  - Once the target found, the path will move from active stash to found stash
- To make it more efficient, the avoid argument can also given to Explorer
  - The path which meets avoid is moved to avoid stash

# PLAY WITH EXPLORER

fairlight

- Use Explorer() instead of run() multiple times
- path_group.explore(find=0x4018f7, avoid=0x4018f9)
- Why we should reach 0x4018f7?

# EXPLORER - CONT

- Explorer() find argument
  - An address to find
  - A set or list of addresses to find
  - A function that takes a path and returns whether or not it matches.

# PLAY WITH EXPLORER(2)

crackme0x00a

- Crackme0x00a
  - Define a function to check if STDOUT contains string “Congrats”
  - Explorer !!

# CRACKME0X00A

```
In [54]: p = angr.Project('angr-doc/examples/'
 'CSCI-4968-MBE/challenges/crackme0x00a/crackme0x00a')

In [55]: pg = p.factory.path_group()

In [56]: pg.explore(find=lambda p: "Congrats" in p.state.posix.dumps(1))
Out[56]: <PathGroup with 1 active, 1 found>

In [57]: s = pg.found[0].state

In [58]: print s.posix.dumps(1)
Enter password: Congrats!


In [59]: print(s.posix.dumps(0))
```

# HOOK

- As aforementioned, angr implement several SimProcedures to model POSIX interface

- User can also define their own hook and SimProcedures

# SIMPROCEDURE

- Define your callback as a Class inherited from SimProcedure
- Include the run() function for your hook Logic

```
In [1]: from angr import Hook, Project

In [2]: from simuvex import SimProcedure

In [3]: class NotVeryRand(SimProcedure):
   ...:     def run(self, return_values=None):
   ...:         return 1337
```

# SETUP HOOK

- Hook Function with Symbol
  - Use project.hook_symbol()
- Hook on address
  - Use project.hook()

```
In [7]: p.hook_symbol('rand', Hook(NotVeryRand))
Out[7]: 67109024
```

# HOOK THE RANDOM

Random

- Hook random and return 1337

- Check output by
  - pg.state.posix.dumps(1)

# CFG

- Angr can generates the control flow graphCFG: connect each block according to control transfer instructions
  - CFGFast
  - CFGAccurate

# CONTEXT SENSITIVE

- The function's behavior may depend on it's context

- What property can be used to determined function's behaviors ?
  - program state: memory + register
  - instruction trace
  - call trace
  - variables state
  - function's address

# CFGACCURATE: CONTEXT SENSITIVE

- Angr provide that contect-sensitive control flow analysis
- Used call trace as context
- CFGAccurate
  - cfg.graph
  - cfg.get_all_nodes(address)
  - cfg.nodes()
  - node.predecessors()
  - node.successors()

```
In [101]: cfg0 = p.analyses.CFGAccurate(keep_state=True, context_sensitivity_level=0)

In [102]: cfg1 = p.analyses.CFGAccurate(keep_state=True, context_sensitivity_level=1)

In [103]: cfg2 = p.analyses.CFGAccurate(keep_state=True, context_sensitivity_level=2)

In [104]: cfg2.nodes()
Out[104]:
[<CFGNode rec 0x400541L[14]>,...]

In [106]: n2 = cfg2.get_all_nodes(0x400541L)

In [107]: n1 = cfg1.get_all_nodes(0x400541L)

In [110]: n0 = cfg0.get_all_nodes(0x400541L)

In [111]: print n0, n1, n2
[<CFGNode rec 0x400541L[14]>]
[<CFGNode rec 0x400541L[14]>, <CFGNode rec 0x400541L[14]>]
[<CFGNode rec 0x400541L[14]>, <CFGNode rec 0
x400541L[14]>,
<CFGNode rec 0x400541L[14]>]
```

# PRACTICE: SENSITIVE LEVEL

What's different if different sensitive-levels are given ?

# FUNCTION MANAGER

- During CFG generating, the Function Manager is also produced
  - cfg.kb.functions
- cfg.kb.functions
  - entry_func = cfg.kb.functions[b.entry]
  - entry_func.block_addrs
  - entry_func.blocks
  - entry_func.name
  - entry_func.get_call_sites()
  - …

# PRACTICE: FIND CALLSITE ADDRESS

Find all the instructions call to printf

# BACKWARD SLICING

- Collect all the instruction that has data flow to the target
- Def-Use Chain

# BACKWARD SLICING

```
1. c = input()
2. b = input()
3. d = 1
4. f = c+1
5. e = c+d
6. send(e)
```

- Backward slice =>
  6.send(e)
  5.e = c+d
  3.d = 1
  1.c = input()

# BACKWARD SLICING

```
cfg = b.analyses.CFGAccurate(context_sensitivity_level=2, keep_state=True)
cdg = b.analyses.CDG(cfg)
ddg = b.analyses.DDG(cfg)
...
bs = b.analyses.BackwardSlice(cfg, cdg=cdg, ddg=ddg, targets=[ (target_node, -1) ])
```

# PRACTICE

- Define the security model check for certain functions
  - printf, memcpt, strcpy, scanf,…
- Find all instructions calling to these functions
- Symbolize inputs
- Make angr automatic finds these instructions
- Check if the function vulnerable

# Q&A