OFFSEC
www.offsec.ir

SHELLPHISH

ANGR advancing next generation research into binary analysis

Mohsen Ahmadi
pwnslinger@asu.edu

[WhoAmI]

Mohsen Ahmadi

- Ph.D. student at the SEFCOM lab at Arizona State University
  - a big fan of symbolic execution
  - angr project contributor
  - Shellphish CTF team member
  - also a sushi lover

How was angr created?

- Supported by DARPA, under the VET program
- Perpetuated by CTF needs
- After a brainstorm in DC at August 2013…

BOOM!

- Pushed by the Cyber Grand Challenge
- And now, pushed even harder by our desire to improve program analysis

Why should I be present for this tutorial?

angr has had an enormous impact since its release

- Used in 225 academic papers
- Fundamental impact on CTF
- 3rd in Cyber Grand Challenge
- Industry

angr is rather hard to get started with

- hard to prioritize time for documentation
- cutting-edge concepts

Agenda

- Binary analysis 101
- Introduction to angr
    - Design
    - Capabilities
    - Basic concepts
- Getting **angr**y
    - Installation
    - Interaction
    - Basic static analysis with angr
    - Basic symbolic execution with angr
- The future
- Q&A

Agenda

- Binary analysis 101
- Introduction to angr
  - Design
  - Capabilities
  - Basic concepts
- Getting **angry**
  - Installation
  - Interaction
  - Basic static analysis with angr
  - Basic symbolic execution with angr
- The future
- Q&A

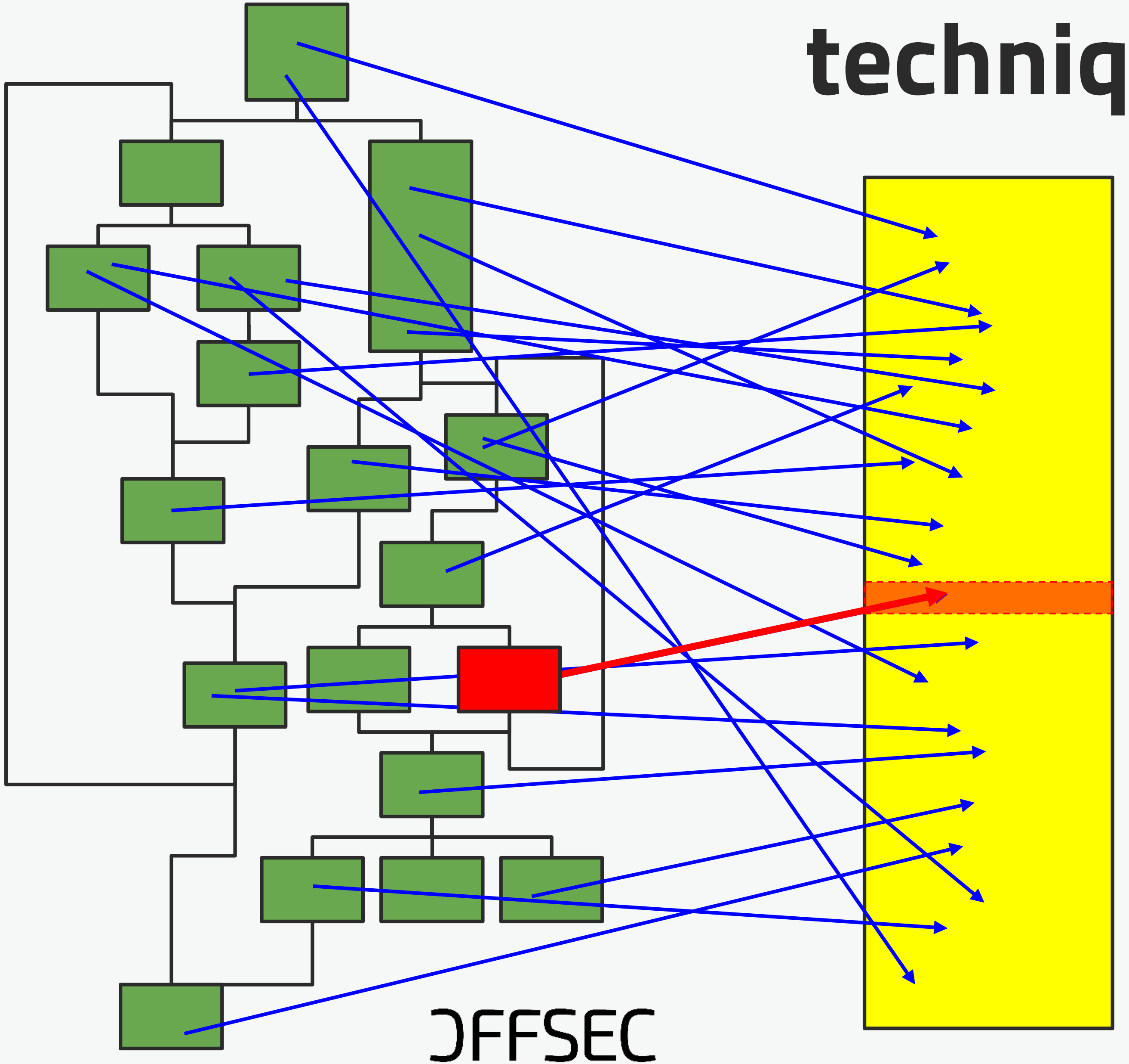
Why binaries?

- Binary code is **everywhere**.
- Binary code is **hard** to understand.
- Binary code is **hard** to analyze.
- Analyzing binary code is **important**!

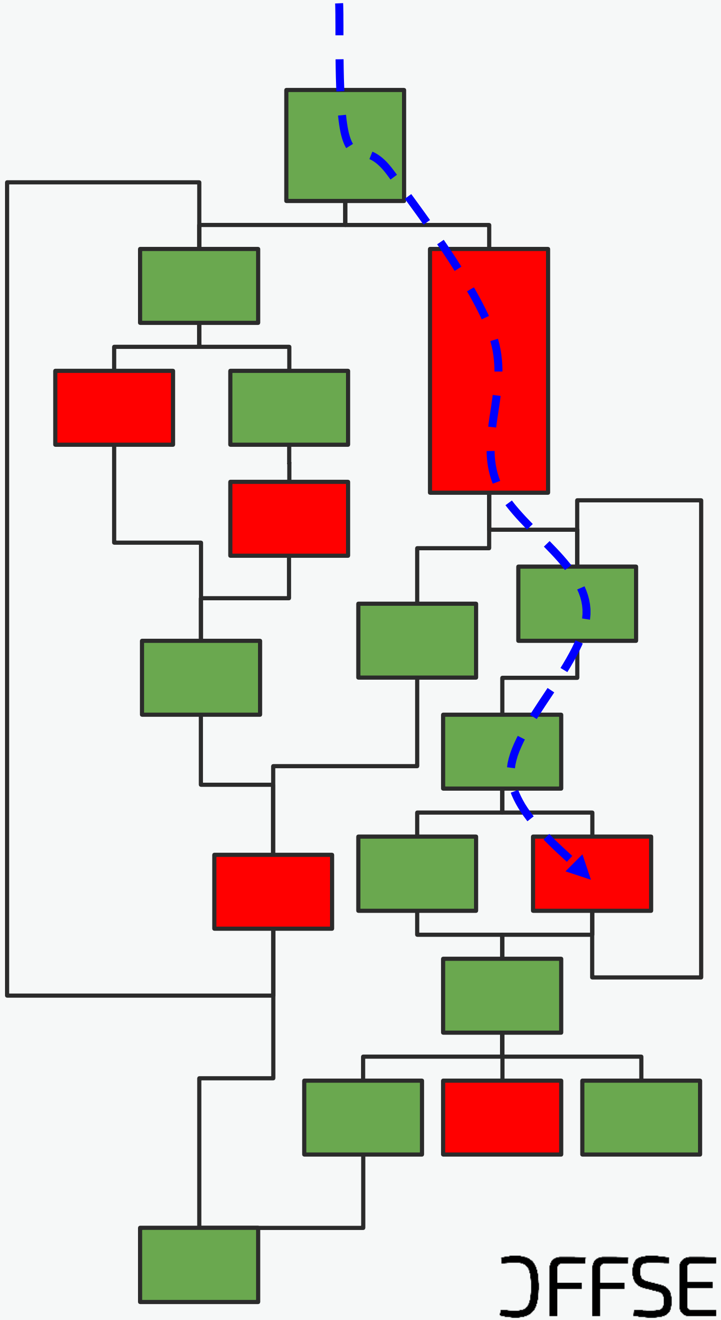
Binary analysis – General techniques

- Static
- Dynamic
- "Symbolic"

Static techniques



Problems



- False Positives
- Replayability

Static techniques...

- Scalable

- Sound

- Main problems
    - no replayability
    - false positives

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

Dynamic - Fuzzing

1 ⇒ "You lose!"

593 ⇒ "You lose!"

183 ⇒ "You lose!"

4 ⇒ "You lose!"

498 ⇒ "You lose!"

42 ⇒ CRASH

Dynamic - Fuzzing

```
x = input()
if x >= 10:
        if x*200+15 == 267415:
                bug()
        else:
                print "You lose!"
else:
        print "You lose!"
```

```
1 ⇒ "You lose!"
593 ⇒ "You lose!"
183 ⇒ "You lose!"
4 ⇒ "You lose!"
498 ⇒ "You lose!"
42 ⇒ "You lose!"
3 ⇒ "You lose!"
……….
57 ⇒ "You lose!"
```

Dynamic techniques...

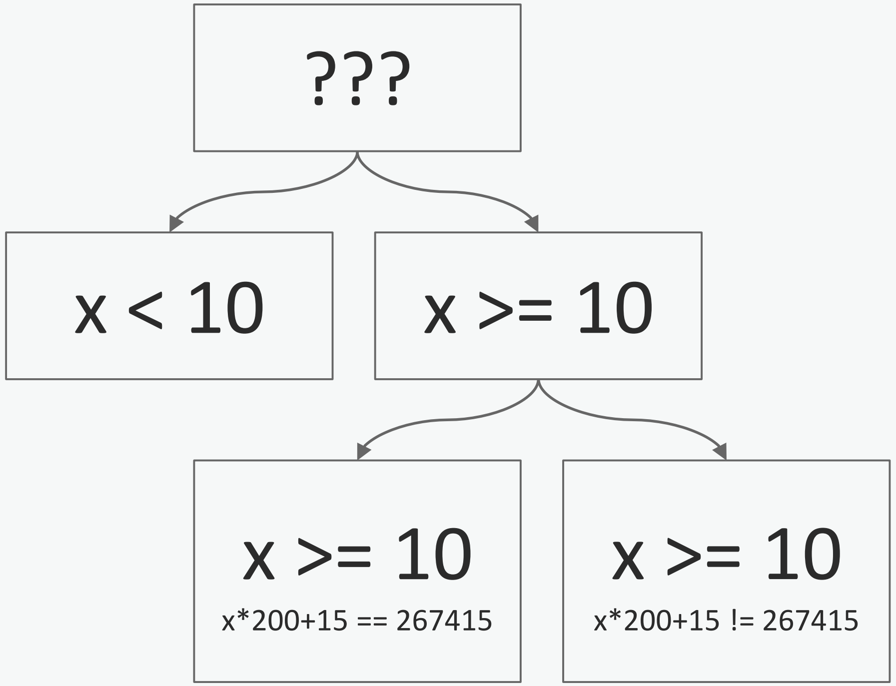
- Very fast

- Straightforward

- Main problem:
  - (lack of) dynamic coverage
  - "semantic gap"

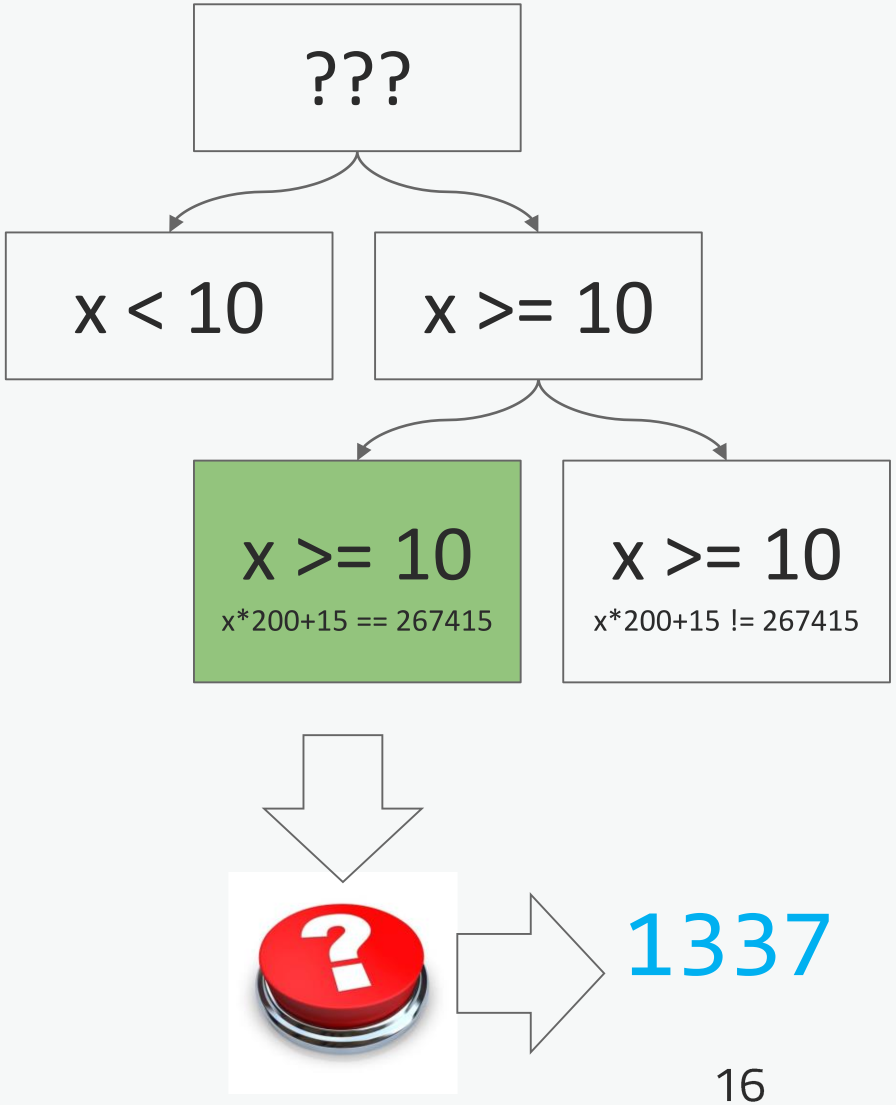
Symbolic execution

```
x = input()
if x >= 10:
      if x*200+15 == 267415:
            bug()
      else:
            print "You lose!"
else:
      print "You lose!"
```



```
x = input()
if x >= 10:
        if x*200+15 == 267415:
                bug()
        else:
                print "You lose!"
else:
        print "You lose!"
```

Symbolic execution



???

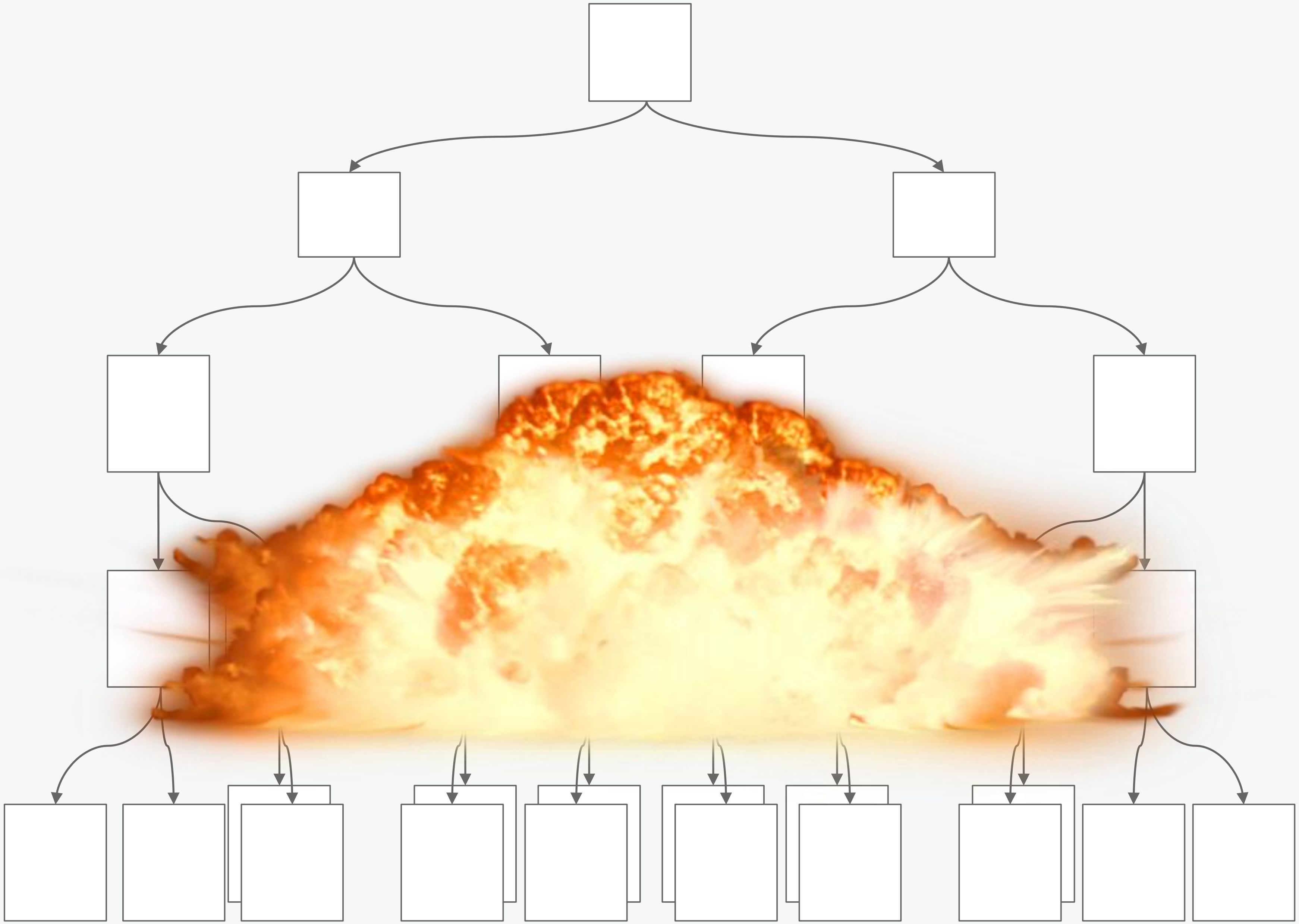
x < 10

x >= 10

x >= 10
x*200+15 == 267415

x >= 10
x*200+15 != 267415

1337



Symbolic execution...

- Semantically aware
- Targetable
- Major problems
    - path explosion
    - constraint solving

Agenda

- Binary analysis 101
- Introduction to angr
    - Design
    - Capabilities
    - Basic concepts
- Getting **angr**y
    - Installation
    - Interaction
    - Basic static analysis with angr
    - Basic symbolic execution with angr
- The future
- Q&A

Why angr?

- Broad arches/platforms support
- Easy to use
- Easy to understand
- Flexible
- Extensible
- Fast prototyping
- Everything open sourced
  … with an awesome license!

angr

Binary Loader

Intermediate Representation

Data Model Abstraction

Program Analysis Core

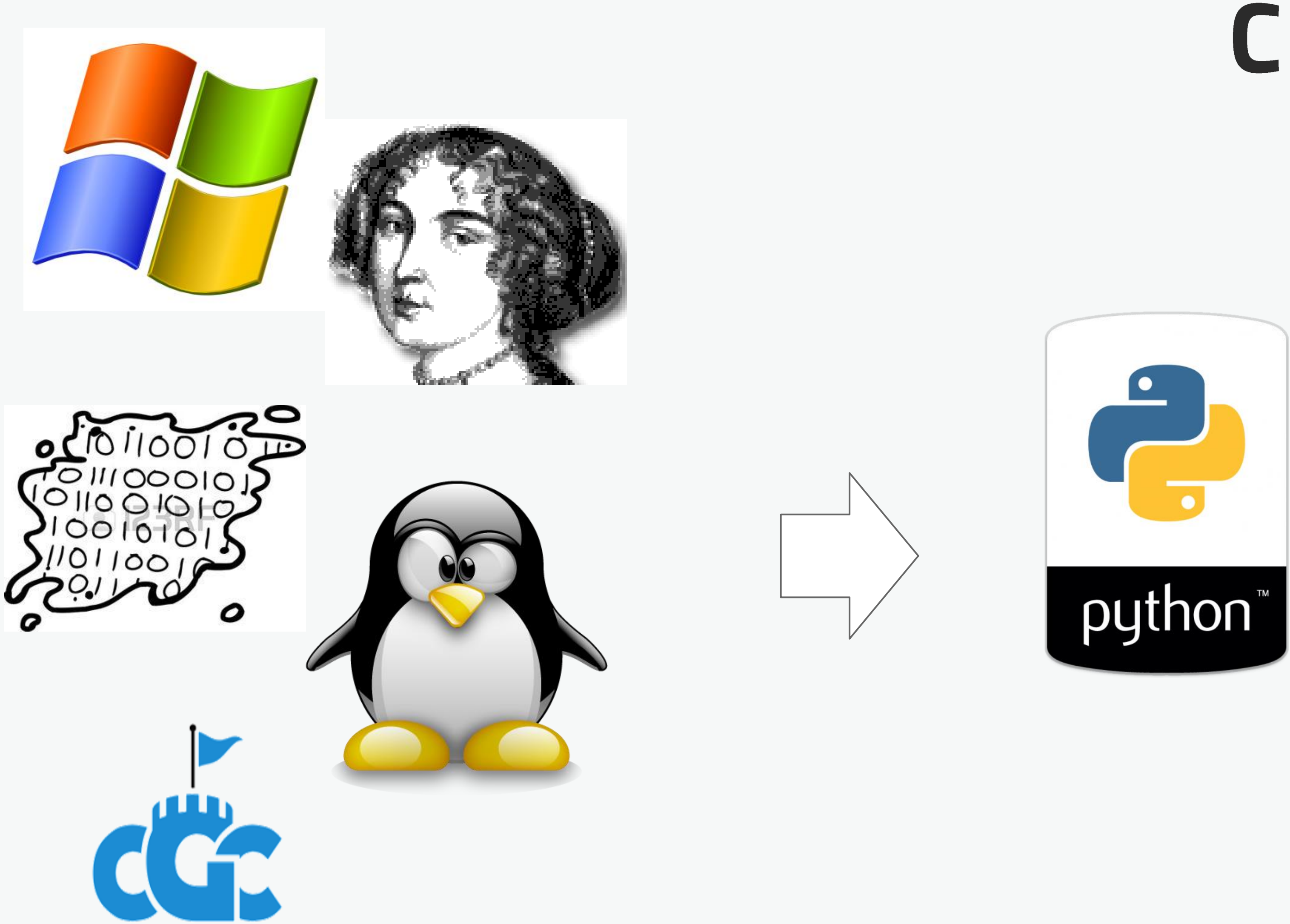
Binary Loader

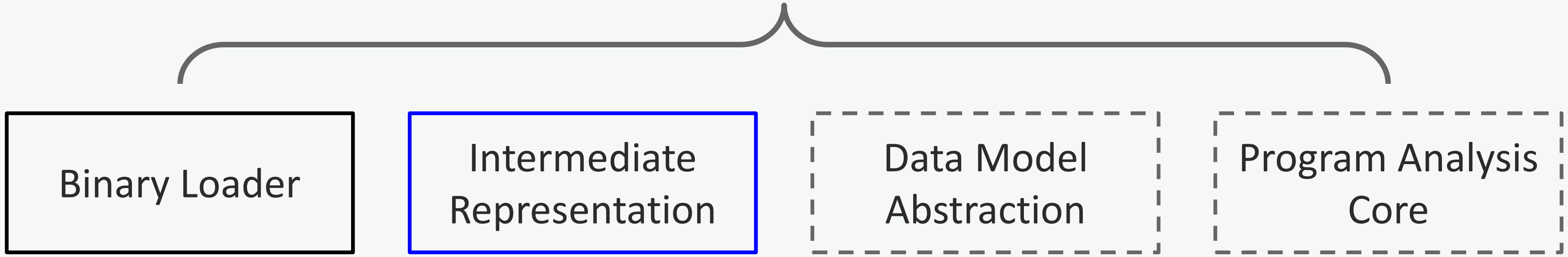
Intermediate Representation

Data Model Abstraction

Program Analysis Core

CLE





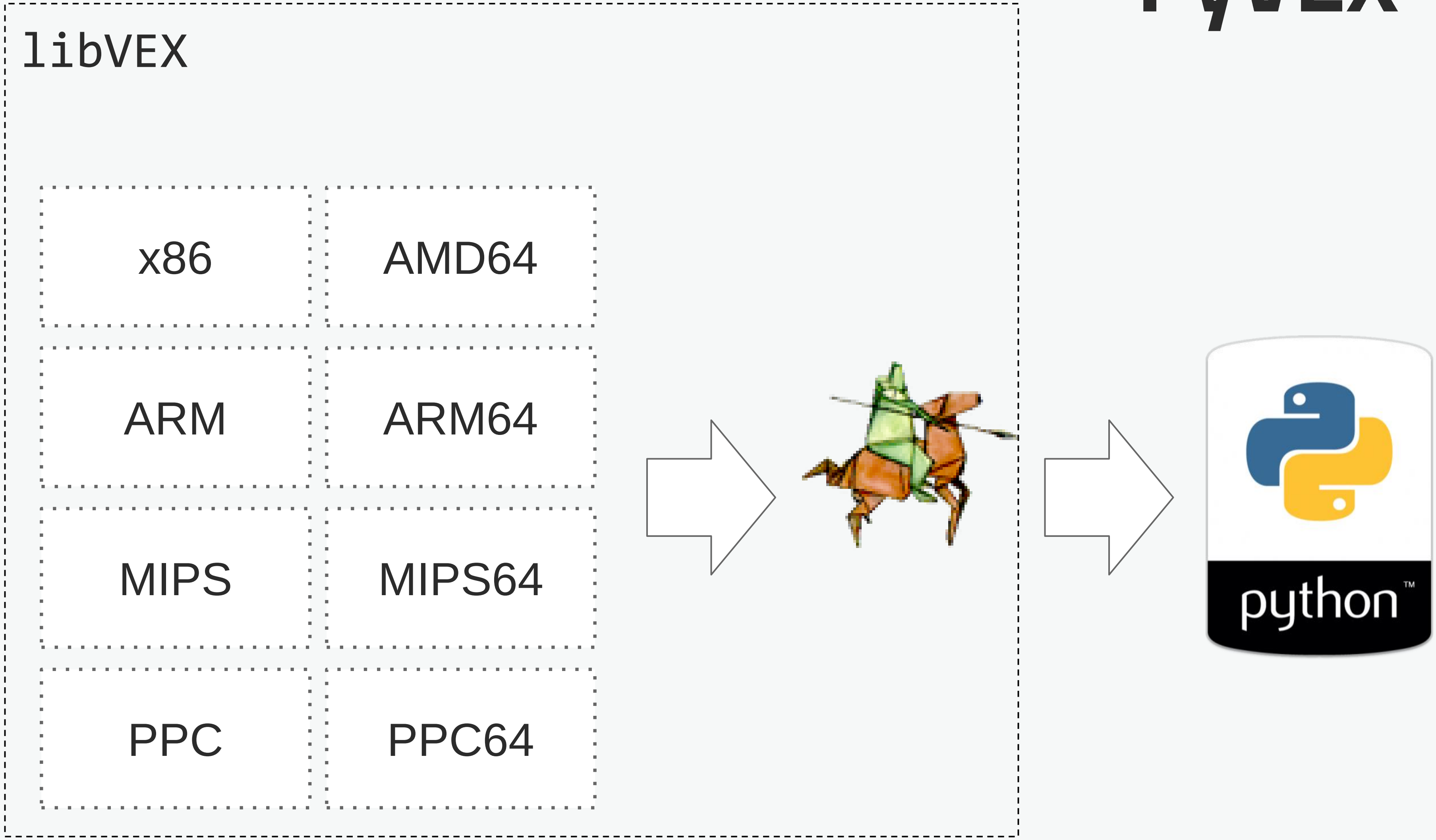
Binary Loader

Intermediate Representation

Data Model Abstraction

Program Analysis Core

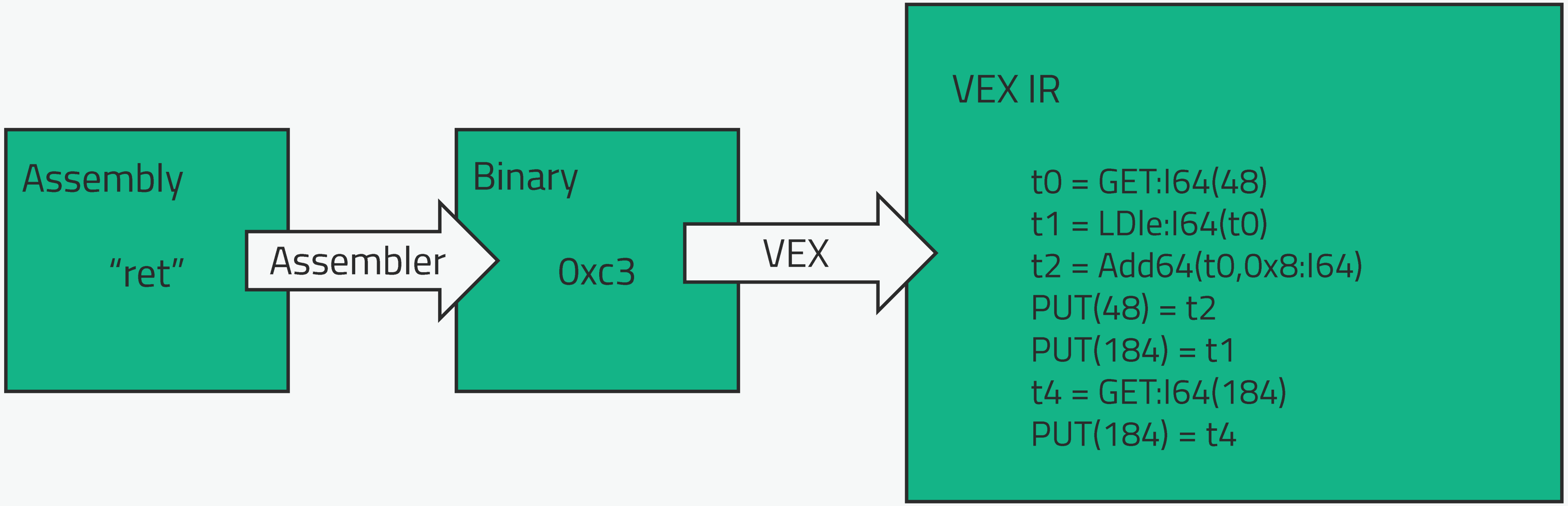
PyVEX



libVEX

- x86
- AMD64
- ARM
- ARM64
- MIPS
- MIPS64
- PPC
- PPC64

VEX example



Assembly

“ret”

Assembler

Binary

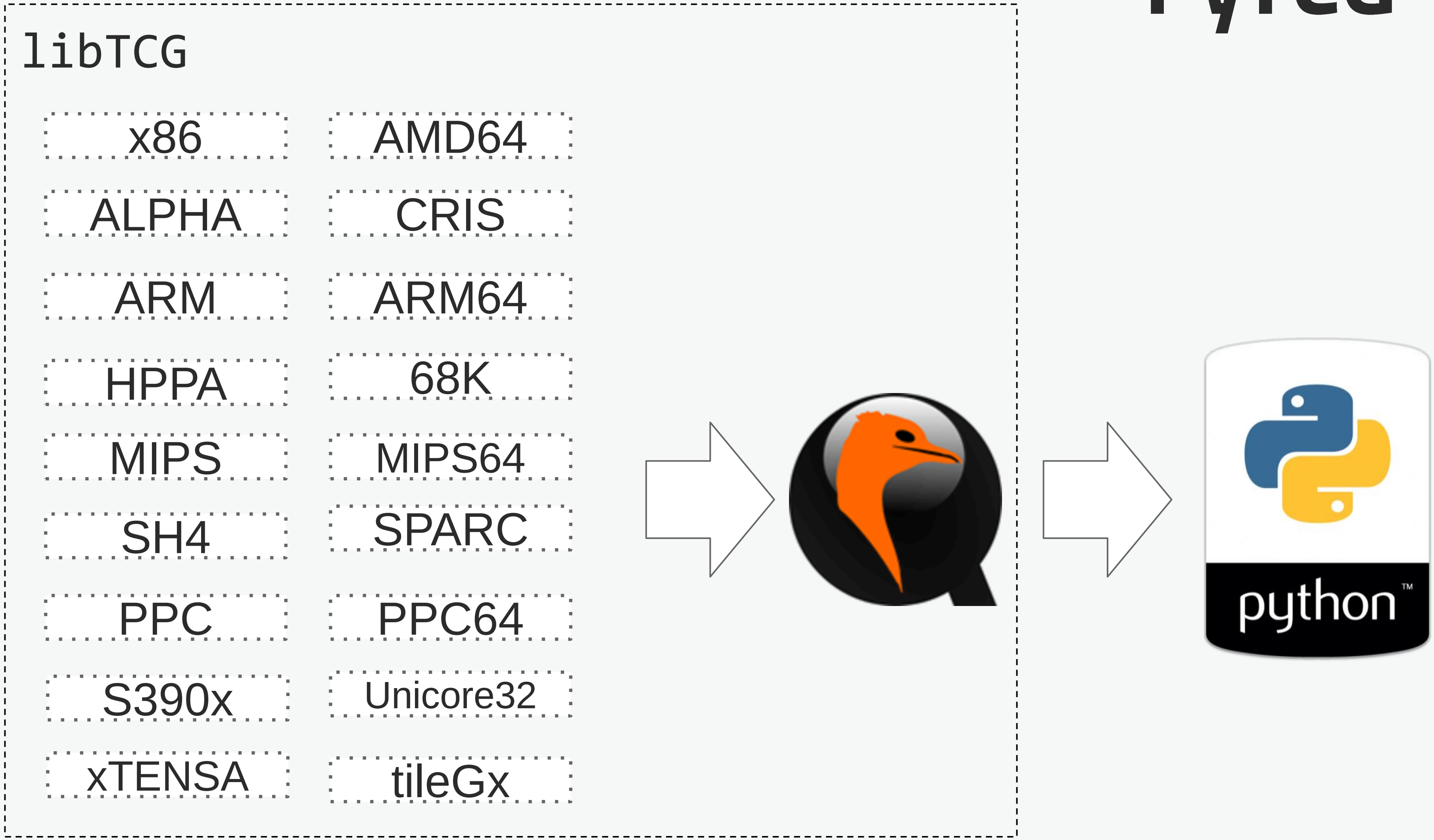
0xc3

VEX

VEX IR

```
t0 = GET:I64(48)
t1 = LDle:I64(t0)
t2 = Add64(t0,0x8:I64)
PUT(48) = t2
PUT(184) = t1
t4 = GET:I64(184)
PUT(184) = t4
```

PyTCG

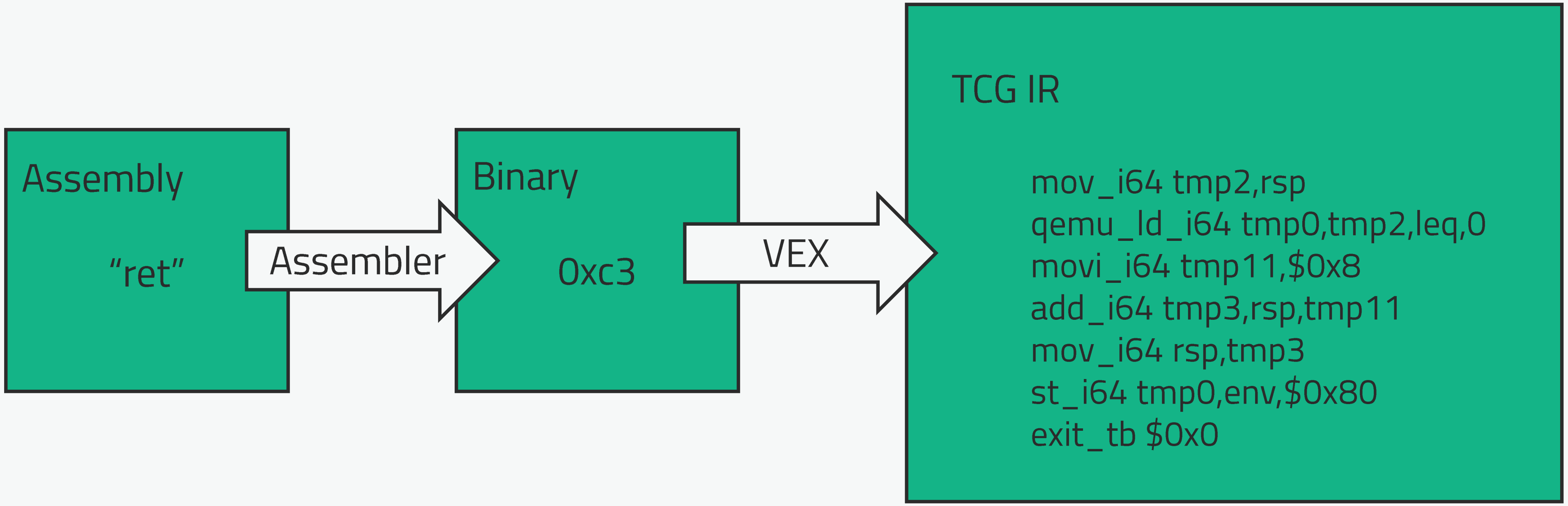


libTCG

x86	AMD64
ALPHA	CRIS
ARM	ARM64
HPPA	68K
MIPS	MIPS64
SH4	SPARC
PPC	PPC64
S390x	Unicore32
xTENSA	tileGx

https://pypi.org/project/pytcg/

TCG example



Assembly

“ret”

Assembler

Binary

0xc3

VEX

TCG IR

mov_i64 tmp2,rsp
qemu_ld_i64 tmp0,tmp2,leq,0
movi_i64 tmp11,$0x8
add_i64 tmp3,rsp,tmp11
mov_i64 rsp,tmp3
st_i64 tmp0,env,$0x80
exit_tb $0x0

Binary Loader

Intermediate Representation

Data Model Abstraction

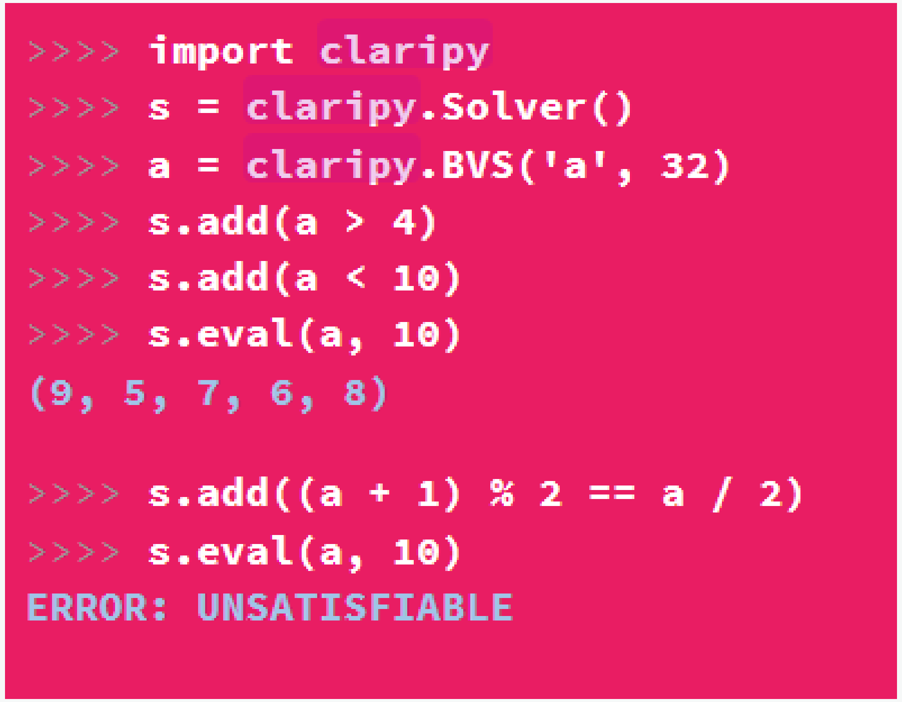
Program Analysis Core

claripy

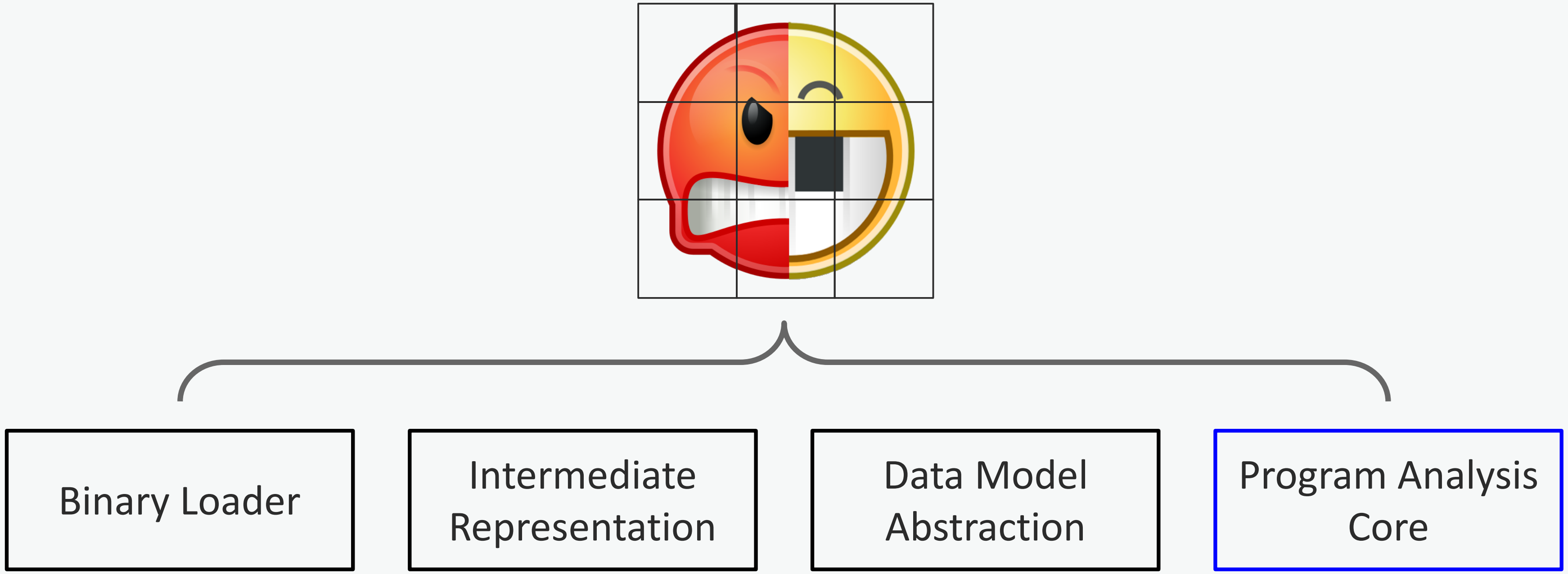
(1+2)

$(\pi+\phi)$

2[10..20]



```
>>>> import claripy
>>>> s = claripy.Solver()
>>>> a = claripy.BVS('a', 32)
>>>> s.add(a > 4)
>>>> s.add(a < 10)
>>>> s.eval(a, 10)
(9, 5, 7, 6, 8)

>>>> s.add((a + 1) % 2 == a / 2)
>>>> s.eval(a, 10)
ERROR: UNSATISFIABLE
```



Binary Loader

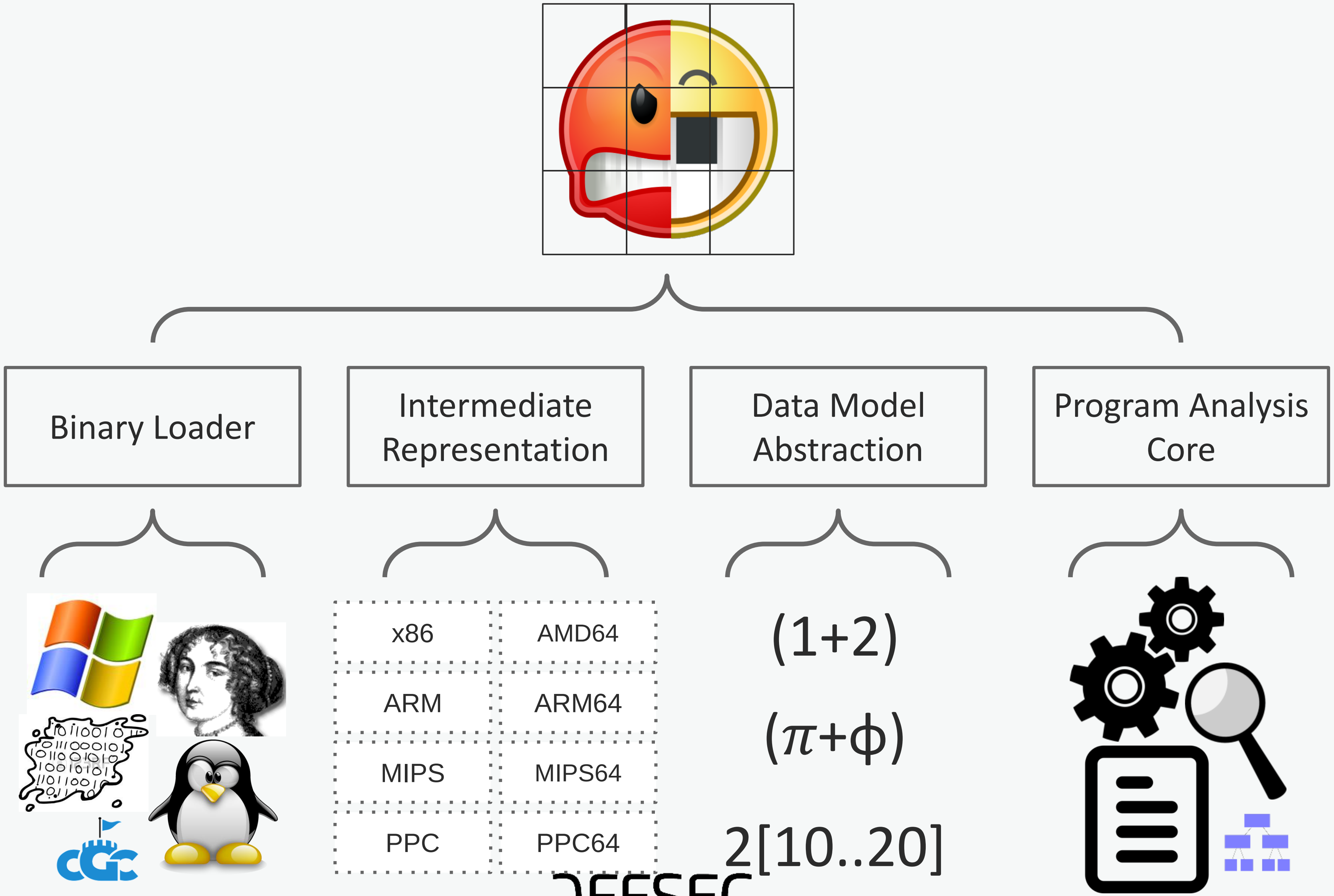
Intermediate Representation

Data Model Abstraction

Program Analysis Core

SimuVEX

- Registers
- Memory
- Files
- OS state



Binary Loader

Intermediate Representation

Data Model Abstraction

Program Analysis Core

x86	AMD64
ARM	ARM64
MIPS	MIPS64
PPC	PPC64

$(1+2)$

$(\pi+\phi)$

$2[10..20]$

Design summary

- *Unified interfaces*
- *Many *different* analysis modes*
- *Flexible and Adaptable*
- *Expandable*

Agenda

- Binary analysis 101
- Introduction to angr
    - Design
    - Capabilities
    - Basic concepts
- Getting **angr**y
    - Installation
    - Interaction
    - Basic static analysis with angr
    - Basic symbolic execution with angr
- The future
- Q&A

Comparison with other projects

Items	angr	KLEE	BAP	BitBlaze	S2E	Triton	Microsoft SAGE	Mayhem
Work on binaries w/o src	🟩	🟥	🟩	🟩	🟩	🟩	🟩	🟩
Online symbolic execution	🟩	🟩	🟥	🟩	🟩	🟩	🟩	🟩
Offline symbolic execution	🟩	?	🟥	?	🟥	🟥	?	🟥
Cross-platform analysis	🟩	🟥	🟩	🟥	🟥	🟧	🟥	🟥
Static analysis	🟩	🟥	🟩	🟥	🟥	🟥	🟥	🟥
Multi-platform/arch support	🟩	🟥	🟧	🟥	🟥	🟧	🟥	🟥
Open source	🟩	🟩	🟩	🟩	🟩	🟩	🟥	🟥
Actively maintained	🟩	🟧	🟩	🟥	?	🟩	🟩	🟩
Free license	🟩	🟩	🟩	🟩	🟩	🟩	🟥	🟥

Agenda

- Binary analysis 101
- Introduction to angr
    - Design
    - Capabilities
    - Basic concepts
- Getting **angr**y
    - Installation
    - Interaction
    - Basic static analysis with angr
    - Basic symbolic execution with angr
- The future
- Q&A

Basic concepts

Program states

- Registers
- Contents in memory
- Files
  file system, socket, etc.
- State of the environment and OS
  Number of allocated buffers in libc
  Thread-local storage

Registers	rax 0x1 rdx 0x6005bc ... rip 0x400080 ...
Memory	[0] = 0 [1] = 1 [2] = 0 ...
Files	file[0] = “Hello, world!\n” file[1] = <symbolic_read 80> ...

Basic concepts

Different modes of program states

- Symbolic mode
  - DO_CCALLS
  - TRACK_CONSTRAINTS: keep adding constraints to state
  - COMPOSITE_SOLVER: constraint set optimizations by solving smaller set of constraints along the way
    - LRU cache for caching already solved constraints
- Static mode
- Fastpath mode
- Tracing

```
state.set_mode(“symbolic”)
state.set_mode(“static”)
state.set_mode(“fastpath”)
```

Basic concepts

State options

- State modes are controlled by combinations of state options
  see `simuvex/s_options.py`
- Feel free to add or remove options to states
- You can even use options of one mode on state of another mode
  -= FLEXIBILITY MAX =-

Agenda

- Binary analysis 101
- Introduction to angr
  - Design
  - Capabilities
  - Basic concepts
- **Getting angry**
  - **Installation**
  - Interaction
  - Basic static analysis with angr
  - Basic symbolic execution with angr
- The future
- Q&A

Let’s install angr!

- OS dependencies
  - Ubuntu/Debian/Arch Linux (recommended!)
  - macOS
  - Windows (Experimental)
- VirtualEnv
- PyPI
- Docker

```
./setup.sh -i -e angr
```

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

Agenda

- Binary analysis 101
- Introduction to angr
  - Design
  - Capabilities
  - Basic concepts
- **Getting angry**
  - Installation
  - **Interaction**
  - Basic static analysis with angr
  - Basic symbolic execution with angr
- The future
- Q&A

angr	- Target program (`angr.Project`) - Analyses (`project.analyses`) - CFG, VFG, BackwardSlice - KnowledgeBase (`project.kb`) - Functions (`project.kb.functions`) - Blocks (`project.factory.block`) - PathGroups (`project.factory.path_group`) - Paths (`path_group.active[0]`)
cle	- Loaded binary (`project.loader`)
simuvex	- SimState (`path.state`) - SimState Plugins (`state.memory`, `state.registers`)
claripy	- Constraint Solver (`state.solver`) - ASTs (`state.registers.rax`)

Our guinea pig - Fauxware

- Simple "backdoor" example
- The very first binary that ran under angr.

```
$ ./angr/binaries/tests/x86_64/fauxware
Username:
blahblah
Password:
blahblah
Go away!
```

Interaction point - Loading a binary

```
# let's get started
import angr
project = angr.Project("angr/binaries/tests/i386/fauxware")
```

Interaction point - Binary information

```
# imports
print project.loader.main_bin.imports

# sections
print project.loader.main_bin.sections

# other shared objects
print project.loader.all_objects
```

Interaction point - Analyses

```
# run a Control-Flow Graph analysis
cfg = project.analyses.CFG()

# view all basic blocks
print len(cfg.nodes())

# access the CFG as a NetworkX graph
print cfg.graph.edges()
```

Interaction point - Knowledge base

```
# the prior CFG filled in a recovered callgraph
print project.kb.callgraph.edges()

# you can access function information through the function
# manager
entry_function = project.kb.functions[project.entry]
print entry_function.addr

# you can access the function's basic block NetworkX graph
print entry_function.graph.edges()

# or get at other function information
print entry_function.code_constants
print entry_function.graph
```

Interaction point - Binary lifting

```
# lift a block with angr
block = project.factory.block(project.entry)

# view the disassembly
block.pp()

# view the VEX IR
block.vex.pp()

# lift *all* blocks
all_vex = { }
for b in cfg.nodes():
    try: all_vex[b.addr] = project.factory.block(b.addr).vex
        except AngrTranslationError: pass
```

Interaction point - Symbolic execution

```
# start a new PathGroup
~~path_group = project.factory.path_group()~~
state = p.factory.entry_state()
simgr = project.factory.simulation_manager(state)

# step
simgr.step()

# step until it branches
simgr.step(until=lambda simgr: len(simgr.active) != 1)

# check the paths that are still active
print(simgr.active)

# step until everything terminates
simgr.run()
```

Interaction point - Symbolic states

```
# select one of the deadended states
state = simgr.deadended[0]

# a state has plugins, representing registers, memory, etc
print(state.regs.eax)
print(state.memory.load(state.regs.esp, 8))
print(state.memory.load(state.regs.esp, 8, endness="Iend_LE"))

# one of the plugins represents the system state
print(state.posix.files)

# files are backed by a memory region
print(state.posix.files[1].content.load(0, 8))
```

Interaction point - Symbolic solver

```
# each state has a solver plugin
state.se

# you can use this solver to retrieve concrete values.
# for example, you can get 2 potential solutions for RAX
print(state.se.eval(state.regs.eax, 2))

# or you can get ranges for values
print(state.se.min(state.regs.eax))
print(state.se.max(state.regs.eax))

# you can also add constraints
state.add_constraints(state.regs.eax == 0x100)
```

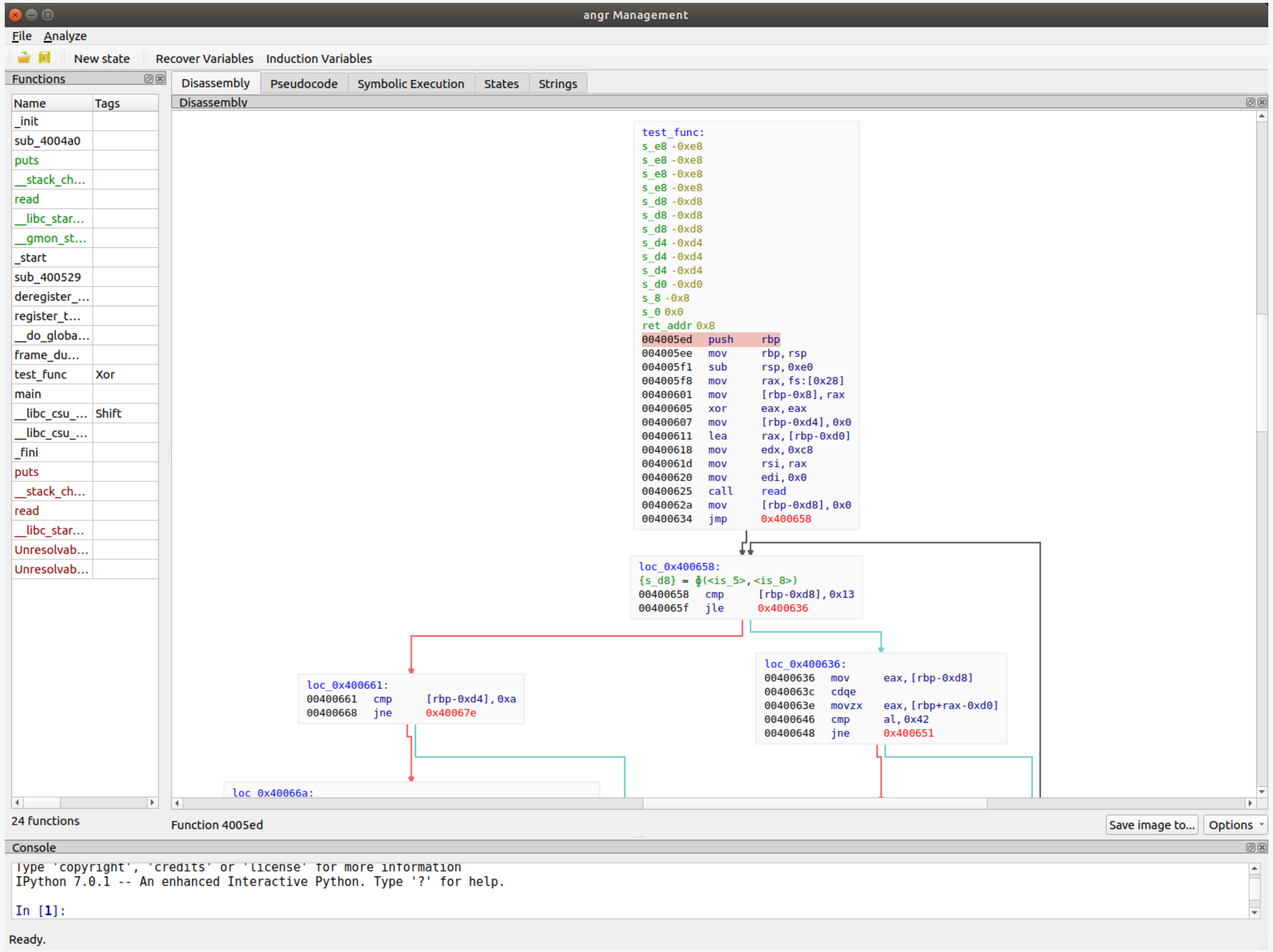
Interaction point - Symbolic expressions

```
# each value in angr is represented as an expression tree
print(state.regs.eax)
complex_expression = state.regs.eax + 1
print(complex_expression.op)
print(complex_expression.args)

# you can slice expressions, but the bit addressing is weird
whole_expression = complex_expression[-31:0]
lsb_byte = complex_expression[-7:0]

# simple optimizations are automatically performed
assert complex_expression is complex_expression + 0
```

Say hi to angr-management

- Based on Qt and Enaml
- Visualize important information
  - Disassembly
  - Control flow graph
  - Data dependencies
  - Program states
- Allow programming and easy scripting!

Disassembly view



Name	Tags
_init	
sub_4004a0	
puts	
__stack_ch...	
read	
__libc_star...	
__gmon_st...	
_start	
sub_400529	
deregister_...	
register_t...	
__do_globa...	
frame_du...	
test_func	Xor
main	
__libc_csu_...	Shift
__libc_csu_...	
_fini	
puts	
__stack_ch...	
read	
__libc_star...	
Unresolvab...	
Unresolvab...	

24 functions

```
test_func:
s_e8 -0xe8
s_e8 -0xe8
s_e8 -0xe8
s_e8 -0xe8
s_d8 -0xd8
s_d8 -0xd8
s_d8 -0xd8
s_d4 -0xd4
s_d4 -0xd4
s_d4 -0xd4
s_d0 -0xd0
s_8 -0x8
s_0 0x0
ret_addr 0x8
004005ed  push    rbp
004005ee  mov     rbp, rsp
004005f1  sub     rsp, 0xe0
004005f8  mov     rax, fs:[0x28]
00400601  mov     [rbp-0x8], rax
00400605  xor     eax, eax
00400607  mov     [rbp-0xd4], 0x0
00400611  lea     rax, [rbp-0xd0]
00400618  mov     edx, 0xc8
0040061d  mov     rsi, rax
00400620  mov     edi, 0x0
00400625  call    read
0040062a  mov     [rbp-0xd8], 0x0
00400634  jmp     0x400658
```

```
loc_0x400658:
{s_d8} = ϕ(<is_5>, <is_8>)
00400658  cmp     [rbp-0xd8], 0x13
0040065f  jle     0x400636
```

```
loc_0x400661:
00400661  cmp     [rbp-0xd4], 0xa
00400668  jne     0x40067e
```

```
loc_0x400636:
00400636  mov     eax, [rbp-0xd8]
0040063c  cdqe
0040063e  movzx   eax, [rbp+rax-0xd0]
00400646  cmp     al, 0x42
00400648  jne     0x400651
```

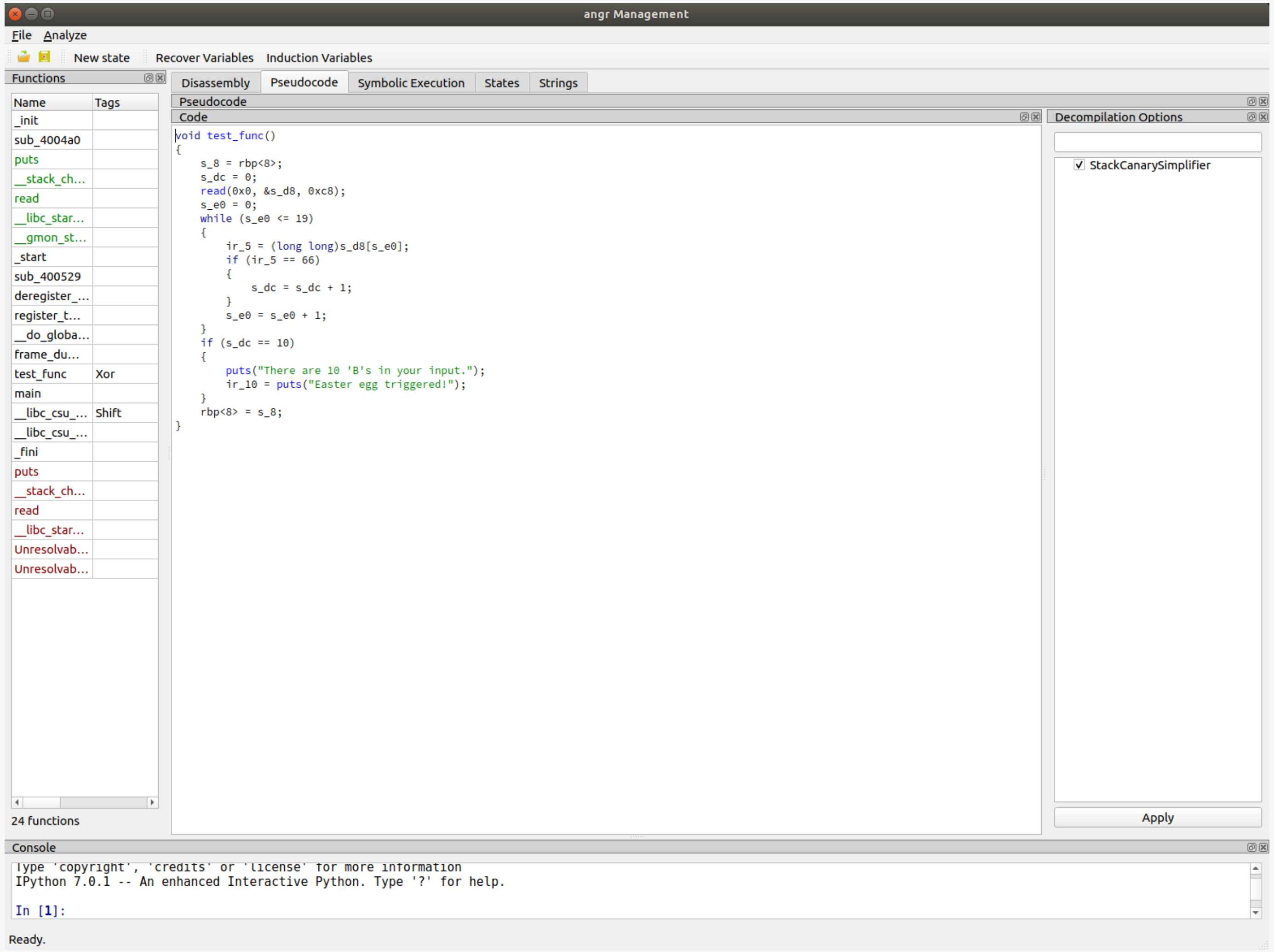
```
loc_0x40066a:
```

Function 4005ed

```
Type 'copyright', 'credits' or 'license' for more information
IPython 7.0.1 -- An enhanced Interactive Python. Type '?' for help.

In [1]:
```

Ready.

Decompiler view



Name	Tags
_init	
sub_4004a0	
puts	
__stack_ch...	
read	
__libc_star...	
__gmon_st...	
_start	
sub_400529	
deregister_...	
register_t...	
__do_globa...	
frame_du...	
test_func	Xor
main	
__libc_csu_...	Shift
__libc_csu_...	
_fini	
puts	
__stack_ch...	
read	
__libc_star...	
Unresolvab...	
Unresolvab...	

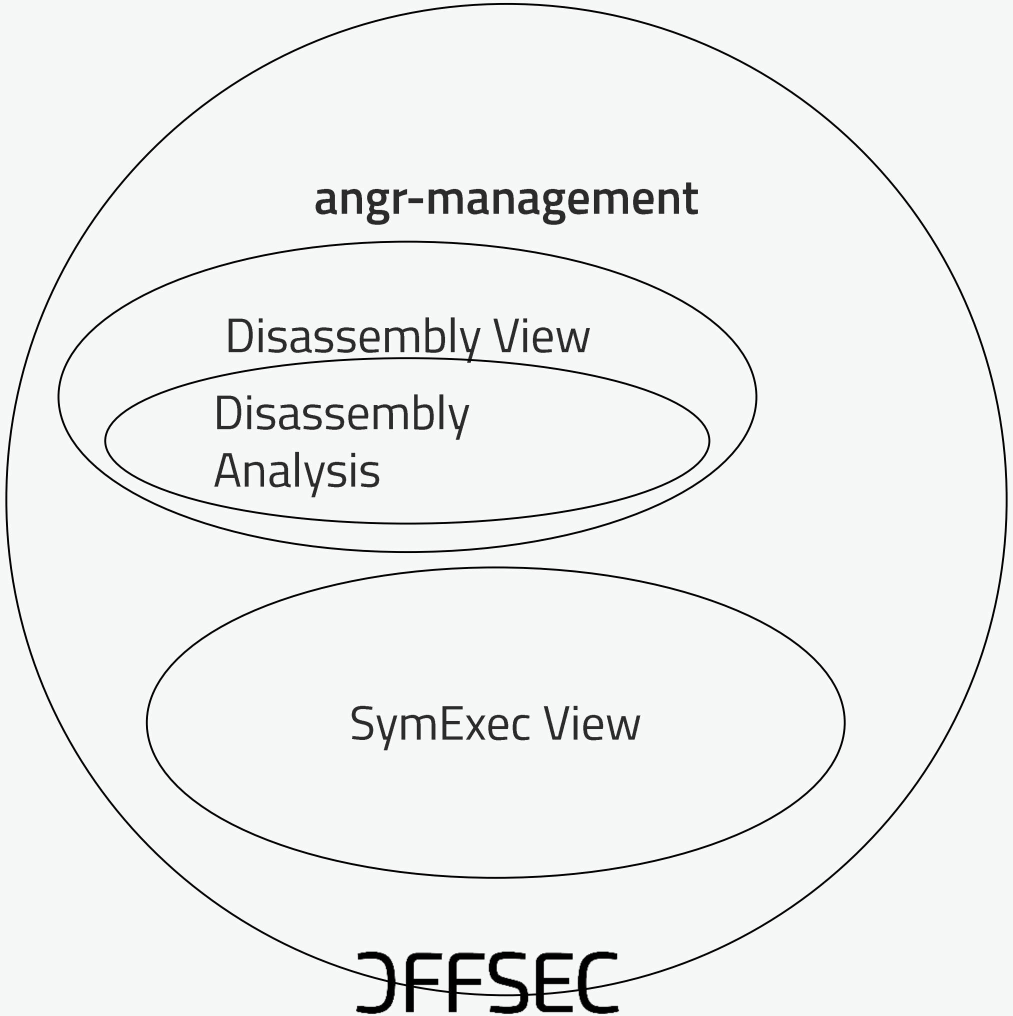
24 functions

```
void test_func()
{
    s_8 = rbp<8>;
    s_dc = 0;
    read(0x0, &s_d8, 0xc8);
    s_e0 = 0;
    while (s_e0 <= 19)
    {
        ir_5 = (long long)s_d8[s_e0];
        if (ir_5 == 66)
        {
            s_dc = s_dc + 1;
        }
        s_e0 = s_e0 + 1;
    }
    if (s_dc == 10)
    {
        puts("There are 10 'B's in your input.");
        ir_10 = puts("Easter egg triggered!");
    }
    rbp<8> = s_8;
}
```

☑ StackCanarySimplifier

```
Type 'copyright', 'credits' or 'license' for more information
IPython 7.0.1 -- An enhanced Interactive Python. Type '?' for help.

In [1]:
```

angr-management overview



angr-management

Disassembly View

Disassembly Analysis

SymExec View

Agenda

- Binary analysis 101
- Introduction to angr
  - Design
  - Capabilities
  - Basic concepts
- **Getting angry**
  - Installation
  - Interaction
  - **Basic static analysis with angr**
  - Basic symbolic execution with angr
- The future
- Q&A

Analyses in angr… what are they?

- analyses in angr are analogous to analyses in LLVM

  angr's analyses are more ad-hoc and more flexible
- angr.Analysis provides some references…

  `.project`

  `.kb`

  And some infrastructure support…

  `Resiliency`

  `Progress reporting`

  `Progress bar`

Analysis overview

Analysis	Description
CFGFast/CFGAccurate	Control flow graph recovery
Disassembly	Linear disassembly rendering routine
DDG	Data dependence analysis
VFG	Value-flow graph recovery, performs value-set analysis on programs
BackwardSlicing	Backward program slicing based on control dependence and data dependence
BoyScout*	Determines architecture of binary blobs
GirlScout*	Determines base addresses of binary blobs

Control flow graph…

- Generating a CFG in angr is easy
  as easy as a one-liner

```
cfg = proj.analyses.CFG()
```

Control flow graph…

But it can also be hard…

- Resolving indirect jumps
  - Jump tables
  - Non-jump-table indirect jumps
  - Indirect calls

- Guessing data types
  - Strings
  - Integers
  - Floats

Control flow graph… the graph?

```
CFG.graph  # a networkx.DiGraph()
```

- Traverse the graph like traversing any other directed graph
- `.get_successors()` and `.get_predecessors()` … they take arguments!
- Get a node from the graph, and then access `.successors` and `.predecessors`

Using the CFG

```
# this grabs *any* node at a given location
entry_node = cfg.get_any_node(p.entry)

# we can also look up predecessors and successors
Print(list(entry_node.predecessors))
Print(list(entry_node.successors))
```

Control flow graph… the graph?

```
CFG.graph  # a networkx.DiGraph()
```

- Each edge is a transition between nodes on the control flow graph
- There are labels on edges!

Label name	Description
jumpkind	Type of the exit that the edge represents. Can be ‘Ijk_Boring’, ‘Ijk_Call’, ‘Ijk_Ret’ etc.
stmt_idx	Index of the statement that creates this transition
ins_addr	Address of the instruction that creates this transition

Control flow graph… the graph?

How can I uniquely mark an instruction in CFG?

- **CFGNode**
  Stores valuable information for each basic block
  - Address
  - Size
  - Instruction addresses
  - Function it belongs to
  - …
- **CFGNode identifier**: a unique ID for nodes in a control flow graph
  … nicknamed *SimRunKey* as of now

Control flow graph… and beyond

Why do we have two CFG analyses?

- **CFGFast** and **CFGAccurate**

Control flow graph… and beyond

- **CFGFast** is way faster than **CFGAccurate**
  Tens of seconds **vs** several minutes or hours

- **CFGAccurate** is as easy to use as **CFGFast**
  They share most arguments
  … if they haven't yet, we'll make them do!

Control flow graph… and beyond

*When do I want to be **fast**?*

– An IDA disassembly style CFG recovery
  …with more indirect jump resolution techniques
  …with basic type inference and "guessing"
  …with function prototype recovery

Control flow graph… and beyond

When do I want to be ***accurate****?*

- Concrete forced execution with states as input
  …it takes a symbolic state!
  …can be viewed as an *emulation*
- Resolves indirect jumps better
  …better (and slower) simulation
- Sound results and complete results

Control flow graph… and beyond

When do I want to be ***accurate****?*

- Context sensitivity
  …you can implement other types of sensitivity!
- Can be used as inputs for other static analyses
- Slow…

Beneath CFG analyses

- Normalization
  Convert all loops to natural loops
  Specify `normalize=True` or call `cfg.normalize()` later

- Loop unrolling
  Unroll a loop into multiple iterations

On top of CFG analysis

Analysis techniques that take CFG as input

- **CFGAccurate** takes a base graph
  …allows you to refine a fast CFG to an accurate CFG
- **Disassembly** takes a CFG
- **Data dependence analysis** (DDG) takes a **CFGAccurate** result
- **Value-set analysis** (VFG) takes a CFG as input
- **BackwardSlice** takes a **CFGAccurate** result

Function

- CFG analyses will fill in `kb.functions` with Function instances
- FakeRets and function returns

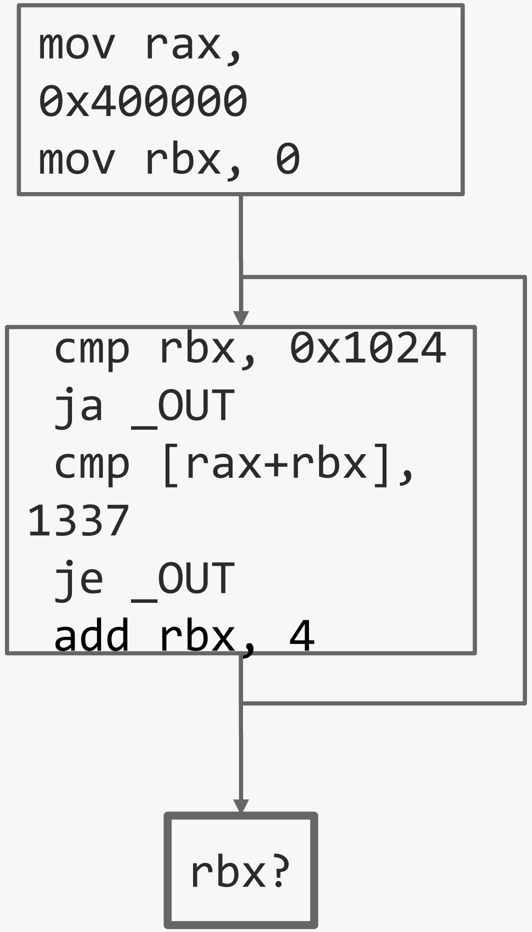
Disassembly analysis

- Generating textual disassembly output
- No overlapping blocks, guaranteed
- Controllable by a number of switches and format strings

Future: a text-based disassembly utility

Value-set analysis

The incentive for having a static analysis



```
mov rax,
0x400000
mov rbx, 0
```

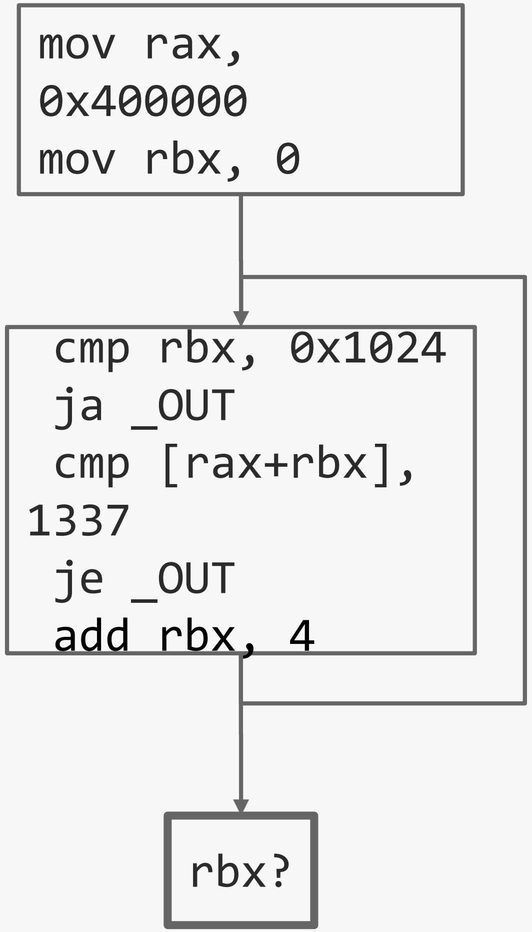
```
 cmp rbx, 0x1024
 ja _OUT
 cmp [rax+rbx],
1337
 je _OUT
 add rbx, 4
```

```
rbx?
```

- What is the value of `rbx` in the end?
- How many times does this loop iterate?
- What are the possible values of `[rax+rbx]` during the loop?

Value-set analysis

The incentive for having a static analysis



```
mov rax,
0x400000
mov rbx, 0
```

```
 cmp rbx, 0x1024
 ja _OUT
 cmp [rax+rbx],
1337
 je _OUT
 add rbx, 4
```

```
rbx?
```

- What is the value of `rbx` in the end?
- How many times does this loop iterate?
- What are the possible values of `[rax+rbx]` during the loop?

Symbolic execution can be used to answer them, but that's not ideal!

Value-set analysis

One solution…

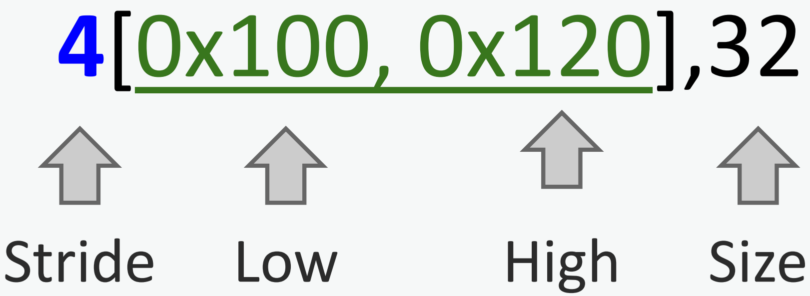
Range analysis

- Instead of concrete values, use **ranges**

Concrete Value Domain	Range Domain
`rax = 0`	`rax = [0, 0]`
`rax_0 = rax + 1 = 1`	`rax_0 = rax + 1 = [1, 1]`
`rax_0 union rax = ???` *Multiple states are required*	`rax_0 union rax = [0, 1]` *One range for many values!*

Value-set analysis

*What is a **value-set**?*



4[0x100, 0x120],32

Stride

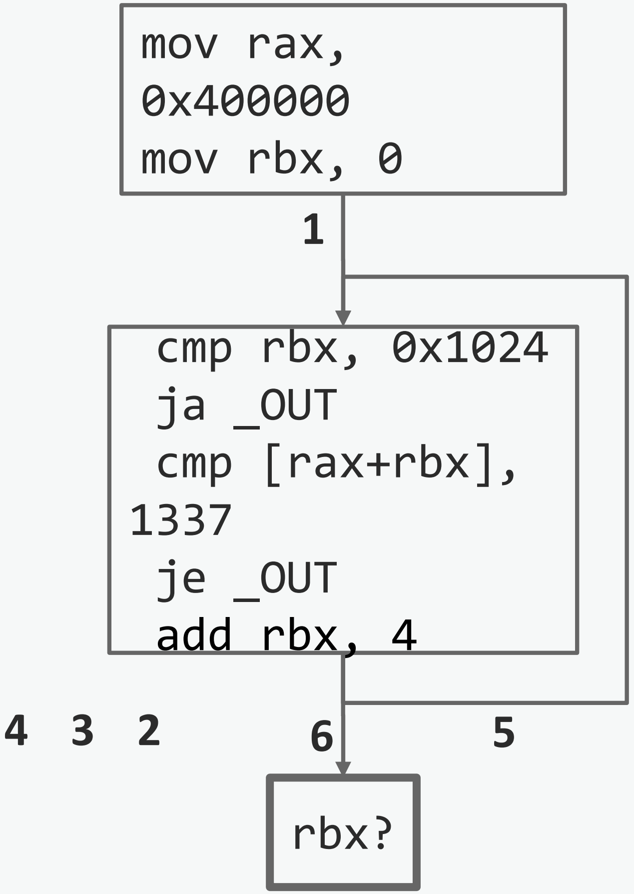
Low

High

Size

0x100	0x10c	0x118
0x104	0x110	0x11c
0x108	0x114	0x120

Value-set analysis



```
mov rax,
0x400000
mov rbx, 0
```

```
 cmp rbx, 0x1024
 ja _OUT
 cmp [rax+rbx],
1337
 je _OUT
 add rbx, 4
```

```
rbx?
```

What is the value of `rbx` eventually?

1.	1[0x0, 0x0],64
2.	4[0x0, 0x4],64
3.	4[0x0, 0x8],64
4.	4[0x0, 0xc],64
5.	4[0x0, ∞],64
6.	4[0x0, 0x1024],64

Widen ⇨

Narrow ⇨

Value-set analysis

How to do it in angr?

Value-flow Graph

- Value-set information is stored in individual program states
- Program states are put in each node on a control flow graph

```
vfg = proj.analyses.VFG(start=...)
```

Value-set analysis

Extract information from a VFG

- Program states are stored in **VFGNode**s
  `Node.states   # a list of states`

- VFGNodes are stored in a connected graph
  `vfg.graph   # a networkx.DiGraph`

Integrate VFG with other analyses

- **Data dependence analysis** (DDG) may take a VFG as an input
  … which is VSA_DDG as of now

Data dependence analysis

What? Why?

- Data dependency information is useful for
  program understanding
  bug finding
  optimizing symbolic execution
  binary rewriting

Agenda

- Binary analysis 101
- Introduction to angr
    - Design
    - Capabilities
    - Basic concepts
- **Getting angry**
    - Installation
    - Interaction
    - Basic static analysis with angr
    - **Basic symbolic execution with angr**
- The future
- Q&A

Symbolic execution

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

State A

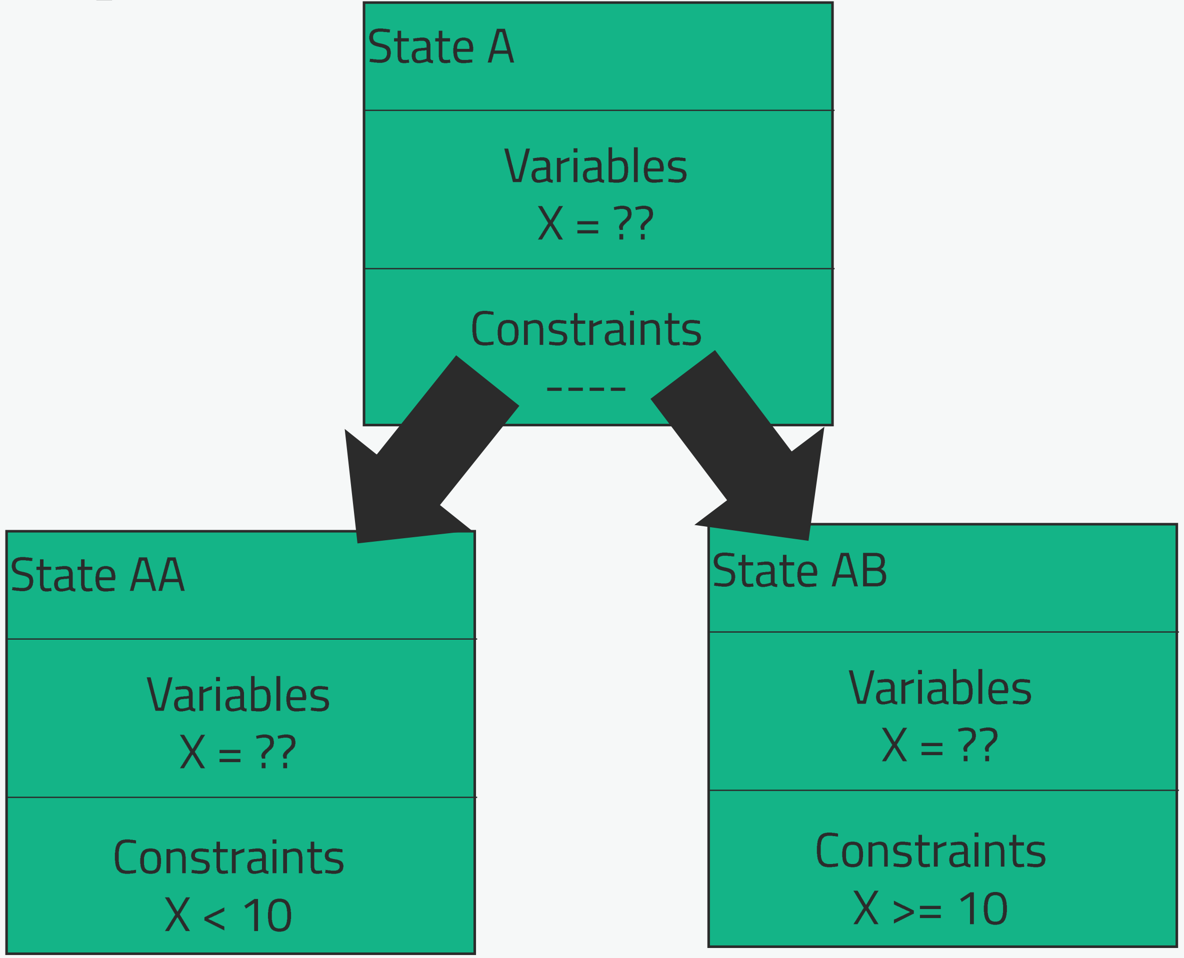
Variables

X = ??

Constraints

Symbolic execution

```
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

State AA

Variables
X = ??

Constraints
X < 10

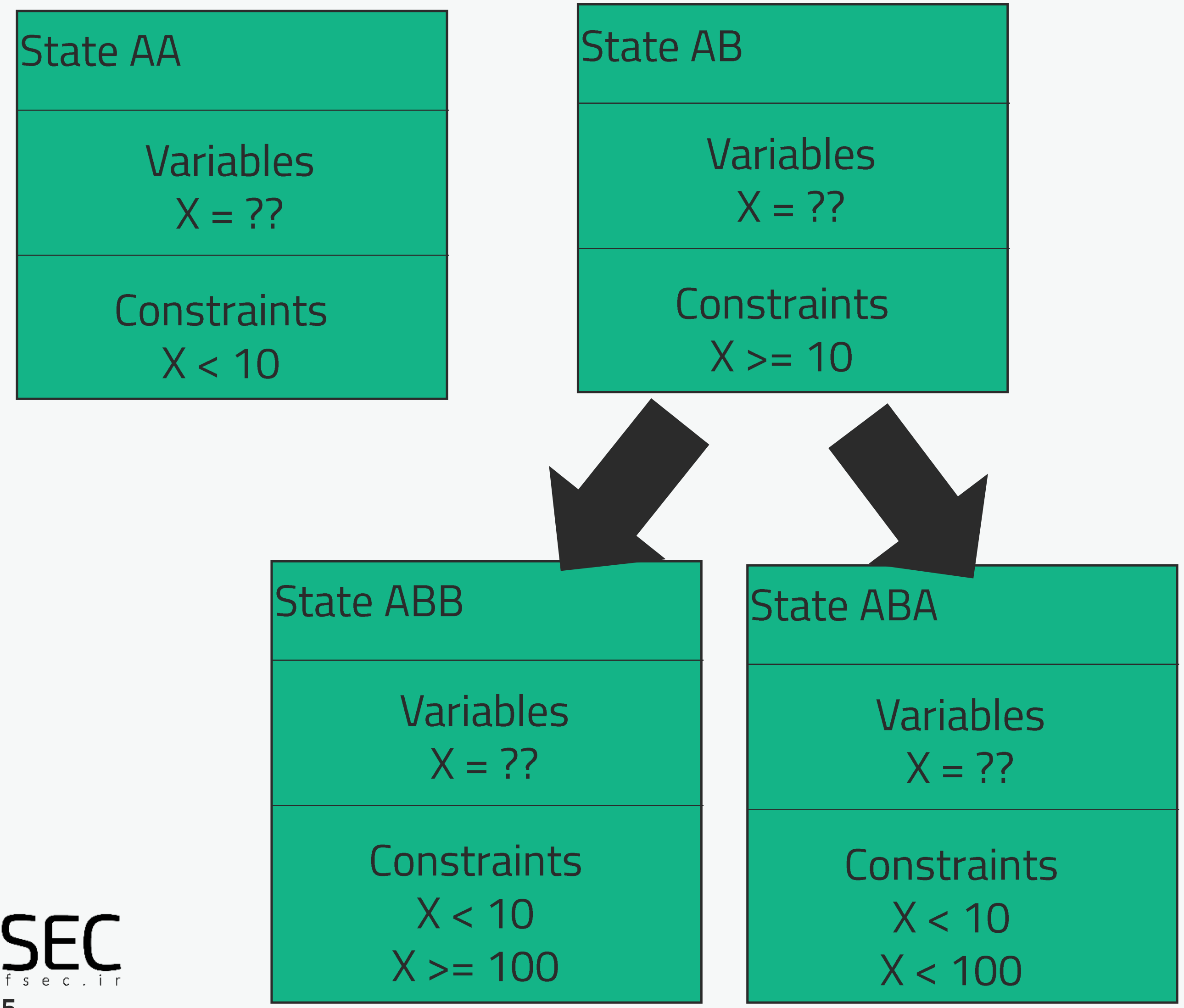
State AB

Variables
X = ??

Constraints
X >= 10

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

Symbolic execution



State AA

Variables
X = ??

Constraints
X < 10

State AB

Variables
X = ??

Constraints
X >= 10

State ABB

Variables
X = ??

Constraints
X < 10
X >= 100

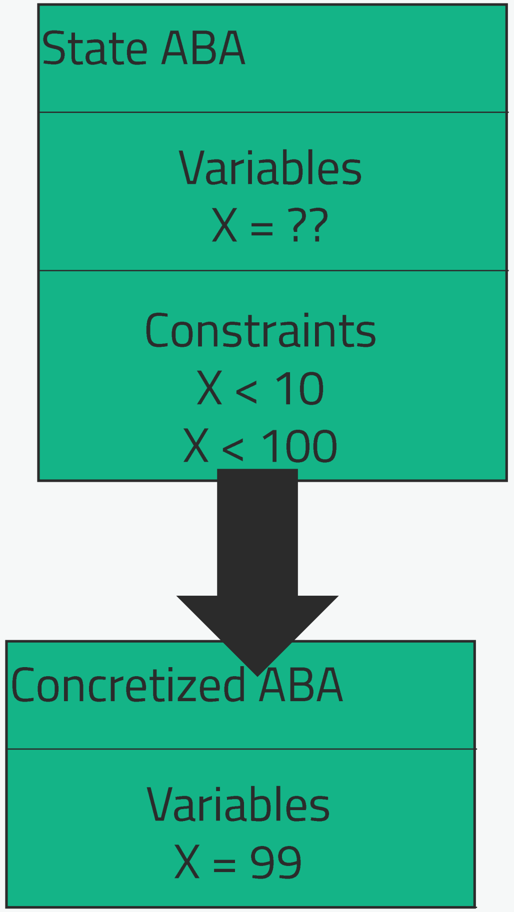
State ABA

Variables
X = ??

Constraints
X < 10
X < 100

Symbolic execution

```
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

Simulation manager

```
>> simgr = project.factory.simulation_manager()
```

Standard symbolic execution interface.

```
>> simgr.step()
>> simgr.active
>> simgr.explore(find=X)
>> simgr.found
```

https://docs.angr.io/docs/pathgroups.html

Simulation manager- Stashes

Simulation managers are containers for “state stashes":

```
Print(simgr.stashes)
```

Standard stashes:

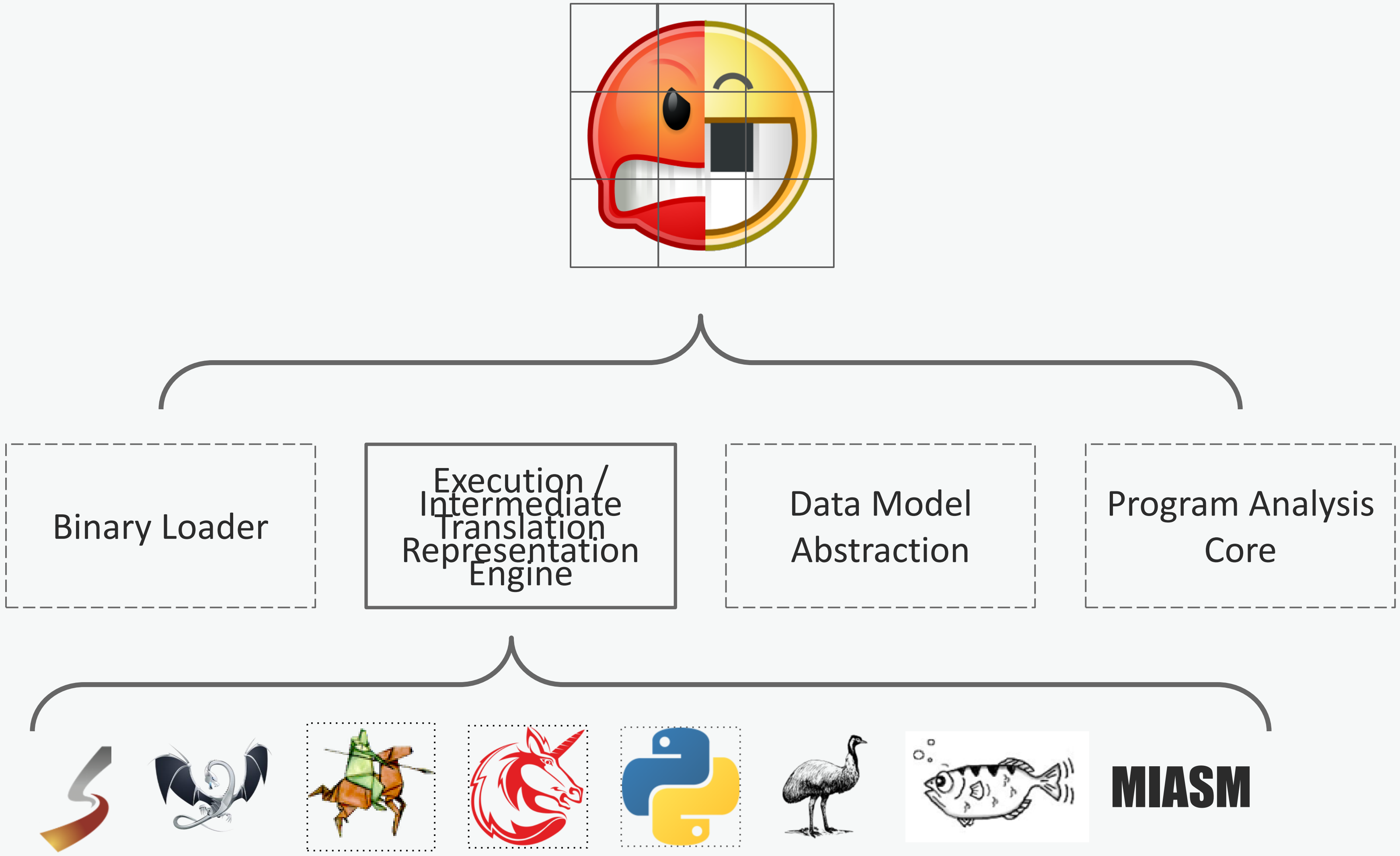
- active - paths that are "live"
- errored - paths that errored out (actual angr bugs)
- found - paths that reached the "find" condition
- avoided - paths that hit the "avoid" condition
- deadended - paths that produced no successors

Path groups - Interaction

```python
# the core simulation manager interaction is stepping states
simgr.step(
    stash, # name of the stash to step ('active' by default)
    n=???, # number of times to step (1 by default)
    until=???, # alternative to "n": step until this condition
    selector_func=???, # filter function for paths to step
    step_func=???, # callback function to call after every step
)

# to augment this, SimulationManager supports messing with stashes
simgr.move('active', 'deadended', lambda state: True)
simgr.merge('active') # merge all paths in a stash
```

Agenda

- Binary analysis 101
- Introduction to angr
    - Design
    - Capabilities
    - Basic concepts
- Getting **angry**
    - Installation
    - Interaction
    - Basic static analysis with angr
    - Basic symbolic execution with angr
- **The future**
- Q&A



Binary Loader

Execution /
Intermediate
Translation
Representation
Engine

Data Model
Abstraction

Program Analysis
Core

MIASM

More?

- We will keep developing angr-management, and make it a data-flow-centric binary analysis tool
- Multiple important analyses, like variable liveness analysis, loop analysis, etc. are being refactored and will be merged in
- Speed improvement: rewrite some angr core components in C++
  - hot spots: symbolic memory, symbolic expressions
- More and better documentation!
  - https://docs.angr.io/

http://angr.slack.com

angr@lists.cs.ucsb.edu

http://github.com/angr/

pwnslinger@asu.edu

https://git.seclab.cs.ucsb.edu/pwnslinger.keys

Agenda

- Binary analysis 101
- Introduction to angr
    - Design
    - Capabilities
    - Basic concepts
- Getting **angry**
    - Installation
    - Interaction
    - Basic static analysis with angr
    - Basic symbolic execution with angr
- The future
- Q&A