Symbolic Execution

yuan

Symbolic execution

- 符號執行
- 透過分析程式，讓輸入可以到達特定的 basic block。
- 通常會使用一個符號值 (λ) 作為輸入，而非具體值。
- 在程式執行時可以得到相應的 path constraint，然後通過 constraint solver 來取得可以到達目標的具體值。

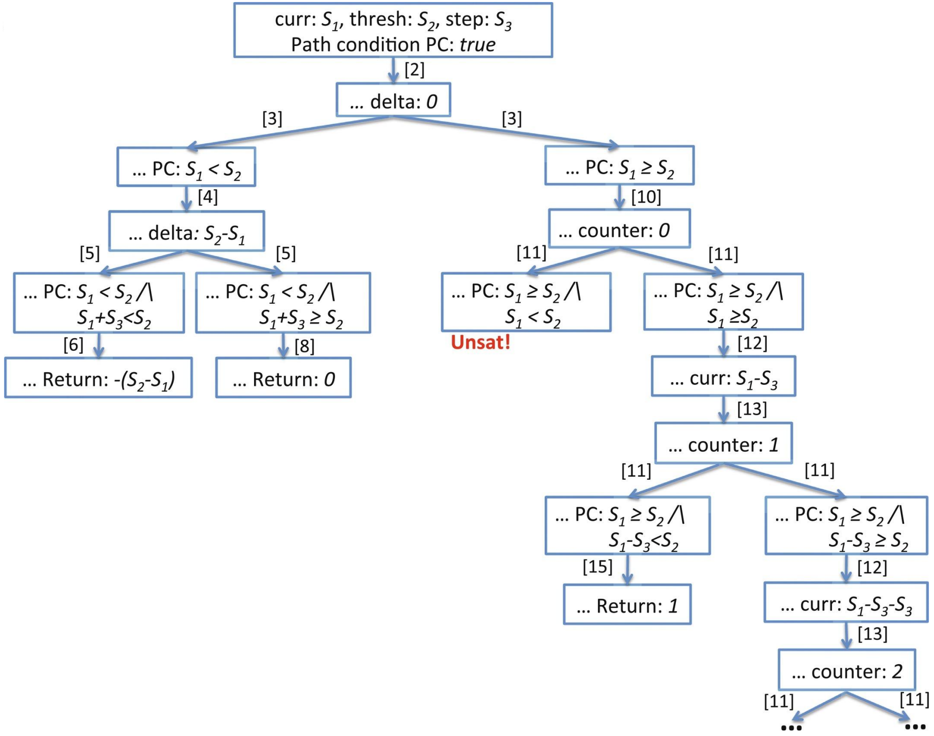
Example

```
1  int f() {
2    y = read();
3    z = y * 2;
4    if (z == 12) {
5      fail();
6    } else {
7      printf("OK");
8    }
9  }
```

2	y = read()	y = λ
3	z = y * 2	z = λ * 2
4	z == 12	λ * 2 == 12
5	fail()	
6	z != 12	λ * 2 != 12
7	printf()	

More Example

```
int compute(int curr, int thresh, int step)
  int delta = 0;
  if (curr < thresh){
    delta = thresh - curr;
    if ((curr + step) < thresh)
      return -delta;
    else
      return 0;
  } else {
      int counter = 0;
      while (curr >= thresh) {
        curr = curr - step;
        counter++;
      }
      return counter;
  }
}
```



常見的工具

- KLEE
- S2E
- Angr

Angr

- Disassembly and intermediate-representation lifting
- Program instrumentation
- Symbolic execution
- Control-flow analysis
- Data-dependency analysis
- Value-set analysis (VSA)
- Decompilation
- Github: https://github.com/angr/angr

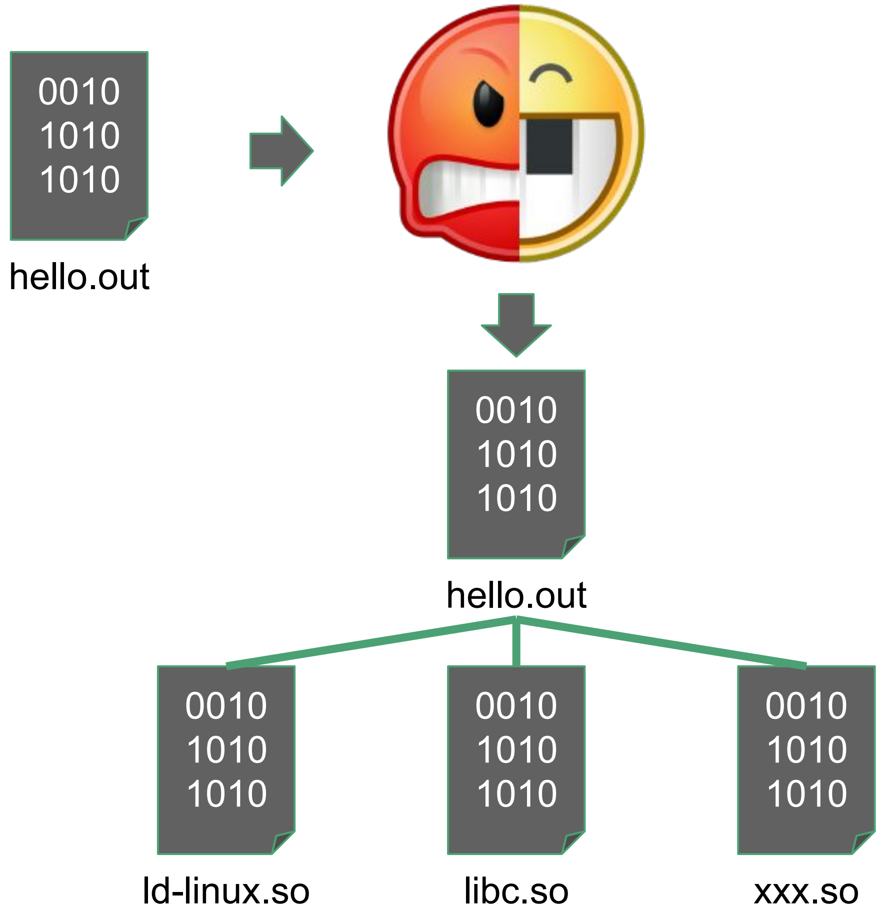
Angr install

- 基本只要
  - $ pip3 install angr
  - $ pip3 install angr-utils
  - $ pip3 install bingraphvis
- angr 會使用一些特殊版本 libraries
  - 推薦使用 python 虛擬環境
  - $ sudo apt-get install python3-dev libffi-dev build-essential virtualenvwrapper
  - $ mkvirtualenv --python=$(which python3) angr && pip install angr angr-utils bingraphvis
- 以上為 ubuntu 如果是別的系統或安裝過程有問題，可以參考：
  - https://docs.angr.io/introductory-errata/install

Angr 內部執行流程

Loading a binary → Translating IR → Analysis

The loader


0010
1010
1010
hello.out
0010
1010
1010
hello.out
0010
1010
1010
ld-linux.so
0010
1010
1010
libc.so
0010
1010
1010
xxx.so

use

```
In [1]: import angr

In [2]: proj = angr.Project('/bin/true')
WARNING | 2020-06-03 13:35:59,976 | cle.loader | The main binary is a position-independent executable. It is being loaded with a base address of 0x400000.

In [3]: hex(proj.entry)
Out[3]: '0x4017b0'
```

The factory

- block
- states
- simulation managers

block

```
In [4]: block = proj.factory.block(proj.entry)

In [5]: block.pp()
0x4017b0:       xor     ebp, ebp
0x4017b2:       mov     r9, rdx
0x4017b5:       pop     rsi
0x4017b6:       mov     rdx, rsp
0x4017b9:       and     rsp, 0xfffffffffffffff0
0x4017bd:       push    rax
0x4017be:       push    rsp
0x4017bf:       lea     r8, [rip + 0x322a]
0x4017c6:       lea     rcx, [rip + 0x31b3]
0x4017cd:       lea     rdi, [rip - 0xe4]
0x4017d4:       call    qword ptr [rip + 0x2057fe]
```

states

- state => a program at a given point in time
- project object 只有存 program 初始狀態

```
In [5]: state = proj.factory.entry_state()

In [6]: state.regs.rip
Out[6]: <BV64 0x4004e0>
```

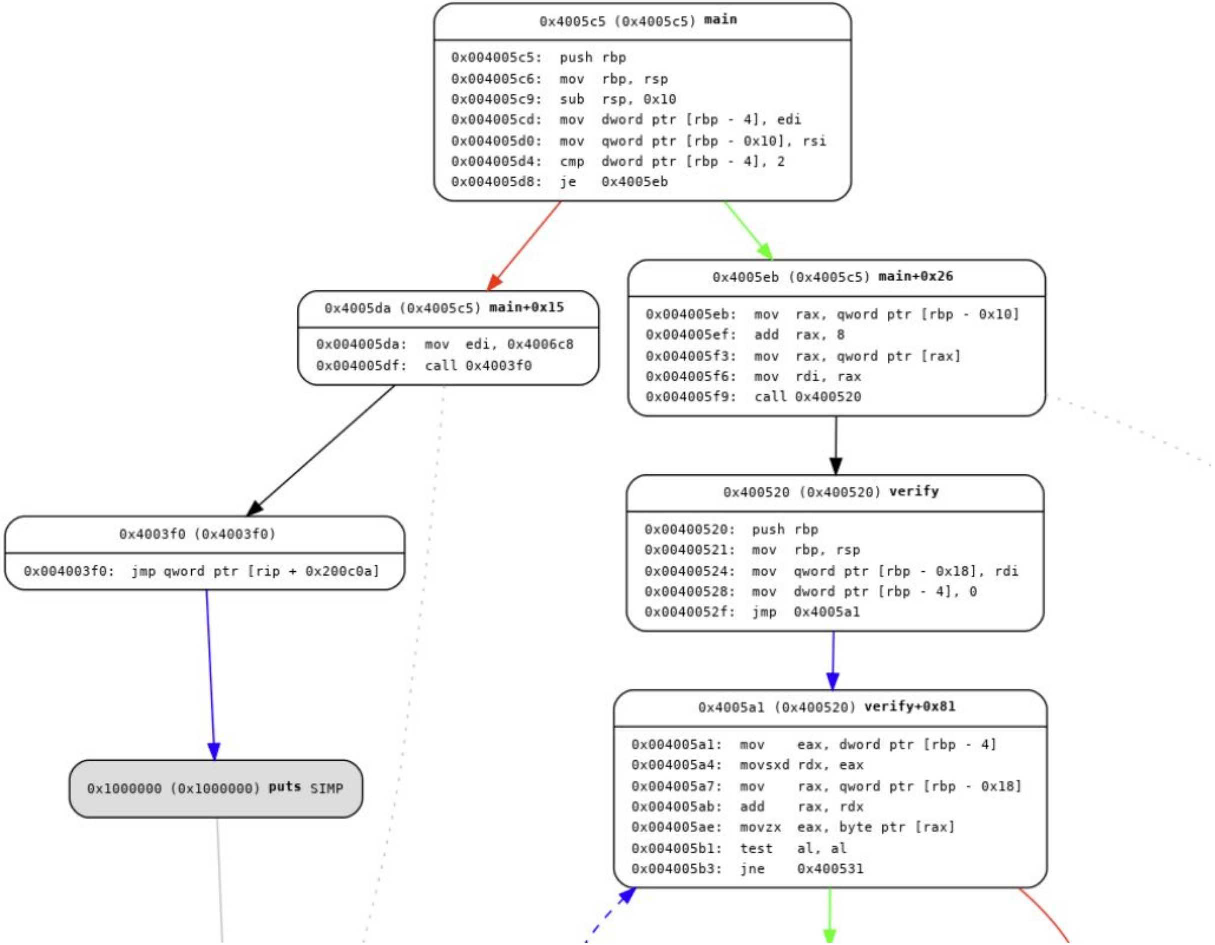
simulation managers

- primary interface in angr for performing execution
- contain several stashes of states
    - active is initialized with the state we passed in

```
In [7]: simgr = proj.factory.simulation_manager(state)

In [8]: simgr.active
Out[8]: [<SimState @ 0x4004e0>]

In [9]: simgr.step()
Out[9]: <SimulationManager with 1 active>

In [10]: simgr.active
Out[10]: [<SimState @ 0x1021ab0>]

In [11]: simgr.active[0].regs.rip
Out[11]: <BV64 0x1021ab0>

In [12]: state.regs.rip
Out[12]: <BV64 0x4004e0>
```

CFG

- angr 可以產生 binary 的 cfg
- https://github.com/axt/angr-utils
- 為了產生 cfg 不受 glibc 影響
    - angr.Project("./a.out", load_options={'auto_load_libs':False})


0x4005c5 (0x4005c5) main
0x004005c5: push rbp
0x004005c6: mov rbp, rsp
0x004005c9: sub rsp, 0x10
0x004005cd: mov dword ptr [rbp - 4], edi
0x004005d0: mov qword ptr [rbp - 0x10], rsi
0x004005d4: cmp dword ptr [rbp - 4], 2
0x004005d8: je 0x4005eb
0x4005da (0x4005c5) main+0x15
0x004005da: mov edi, 0x4006c8
0x004005df: call 0x4003f0
0x4005eb (0x4005c5) main+0x26
0x004005eb: mov rax, qword ptr [rbp - 0x10]
0x004005ef: add rax, 8
0x004005f3: mov rax, qword ptr [rax]
0x004005f6: mov rdi, rax
0x004005f9: call 0x400520
0x4003f0 (0x4003f0)
0x004003f0: jmp qword ptr [rip + 0x200c0a]
0x400520 (0x400520) verify
0x00400520: push rbp
0x00400521: mov rbp, rsp
0x00400524: mov qword ptr [rbp - 0x18], rdi
0x00400528: mov dword ptr [rbp - 4], 0
0x0040052f: jmp 0x4005a1
0x1000000 (0x1000000) puts SIMP
0x4005a1 (0x400520) verify+0x81
0x004005a1: mov eax, dword ptr [rbp - 4]
0x004005a4: movsxd rdx, eax
0x004005a7: mov rax, qword ptr [rbp - 0x18]
0x004005ab: add rax, rdx
0x004005ae: movzx eax, byte ptr [rax]
0x004005b1: test al, al
0x004005b3: jne 0x400531

Create symbolic object

- angr's solver engine is called Claripy
  - import claripy
- BV : bitvector
  - claripy.BVS('x', 32)
- FP : floating-point
  - claripy.FPS('y', claripy.fp.FSORT_DOUBLE)
- Bool : boolean
  - claripy.BoolV(True)

explore

- 在 simulation managers 執行 explore，條件匹配的狀態
    - find => 符合條件
    - avoid => 不符合
- simgr.explore(find=find_addr, avoid=avoid_addr)
    - avoid_addr = [0x400c06, 0x400bc7]
    - find_addr = 0x400c10d

Get solution

- 可以去查看 found 是否有產生能執行到的 constraint
    - found = simgr.found[0]
- 透過 solver.eval 去拿到符合該解的值
    - found.solver.eval(sym_arg, cast_to=str)

LAB

- 我們提供一個 linux binary，請透過 angr 產生出他的 cfg
- 請找到能夠讓該程式印出 correct 的輸入
- 繳交方式 : 學號.zip
  - cfg : 這支程式的 cfg 圖片
  - solve.py : 執行 angr 的 python (please use python3)
  - flag : 正確的輸入
- Note :
  - 此 Lab 把 pie 關了，程式每次執行位置都是相同的
    - 如果有開 angr 會自動把 pie base 放到 0x40000
    - 想說簡單點，就關了 ><

Reference

- https://docs.angr.io/
- https://github.com/angr/angr
- https://github.com/angr/angr-doc
- https://github.com/angr/angr-doc/blob/master/CHEATSHEET.md
- https://github.com/axt/angr-utils