Binary 自動分析的那些事

YSc
2016/07/22

- 當你拿到一個 binary ...

- 當你拿到一個 binary ...
  - file binary
  - ltrace
  - gdb
  - IDA
  - ...

- 當你拿到一個 binary ...
  - file binary
  - ltrace
  - gdb
  - IDA
  - ...

```
int __cdecl main(int argc, const char *
{
  int v3; // ebx@2

  if ( argc == 2 )
  {
    v3 = verify(argv[1]);
    if ( v3 )
    {
      v3 = 0;
      puts("Success!");
    }
    else
    {
      puts("Failure!");
    }
  }
  else
```

- 當你拿到一個 binary ...
  - file binary
  - ltrace
  - gdb
  - IDA
  - ...

```
*((_DWORD *)v1 + 2) ^= 0x55555555u;
*((_DWORD *)v1 + 3) ^= 0x33333333u;
v13 = (unsigned __int8)(v1[2] ^ v12);
v14 = (unsigned __int8)(v1[3] ^ v1[2] ^ v12);
v15 = (unsigned __int8)(v1[4] ^ v1[3] ^ v1[2] ^ v12);
v16 = (unsigned __int8)(v1[5] ^ v1[4] ^ v1[3] ^ v1[2] ^ v12);
v17 = (unsigned __int8)(v1[6] ^ v1[5] ^ v1[4] ^ v1[3] ^ v1[2] ^ v12);
v18 = v1[8] ^ v1[7] ^ v1[6] ^ v1[5] ^ v1[4] ^ v1[3] ^ v1[2] ^ v12;
v19 = (unsigned __int8)(v1[7] ^ v1[6] ^ v1[5] ^ v1[4] ^ v1[3] ^ v1[2] ^ v12);
*((_BYTE *)v1 + 8) = v18;
v20 = v1[9] ^ v18;
v21 = (unsigned __int8)(v1[10] ^ v20);
v22 = (unsigned __int8)(v1[11] ^ v1[10] ^ v20);
v23 = (unsigned __int8)(v1[12] ^ v1[11] ^ v1[10] ^ v20);
v24 = v1[14] ^ v1[13] ^ v1[12] ^ v1[11] ^ v1[10] ^ v20;
v25 = v1[13] ^ v1[12] ^ v1[11] ^ v1[10] ^ v20;
*((_BYTE *)v1 + 15) ^= v24;
*v1 ^= 0x63u;
*((_BYTE *)v1 + 8) ^= 0x30u;
*((_BYTE *)v1 + 1) = (2 * v12 | ((signed int)v12 >> 1)) ^ 0x2F;
*((_BYTE *)v1 + 2) = (4 * v13 | (v13 >> 2)) ^ 0xDC;
*((_BYTE *)v1 + 3) = (8 * v14 | (v14 >> 3)) ^ 0x20;
*((_BYTE *)v1 + 4) = (16 * v15 | (v15 >> 4)) ^ 0xCD;
*((_BYTE *)v1 + 5) = (32 * v16 | (v16 >> 5)) ^ 0xA0;
*((_BYTE *)v1 + 6) = (((_BYTE)v17 << 6) | (v17 >> 6)) ^ 0x83;
*((_BYTE *)v1 + 7) = ((_BYTE)v19 << 7) | (v19 >> 7);
*((_BYTE *)v1 + 9) = (2 * v20 | ((signed int)v20 >> 1)) ^ 0x7D;
*((_BYTE *)v1 + 10) = (4 * v21 | (v21 >> 2)) ^ 0x19;
*((_BYTE *)v1 + 11) = (8 * v22 | (v22 >> 3)) ^ 4;
*((_BYTE *)v1 + 12) = (16 * v23 | (v23 >> 4)) ^ 0xC4;
```

- 一條一條看，一條一條算
- 用工具（ z3 ）來算
- 整支程式自動跑自動算

這個議程在幹麻

- binary 自動分析的原理
- 如何用 angr 寫解 CTF reverse 的腳本

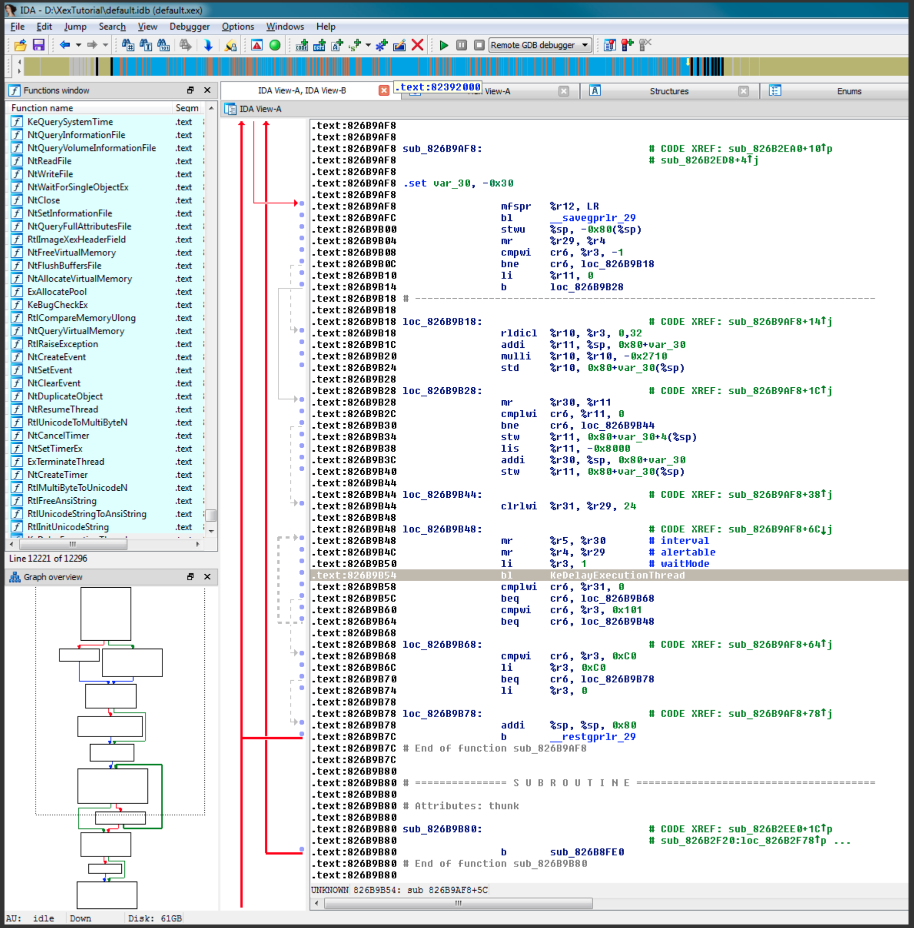
- 先來談談要怎麼自動分析，
  - 符號執行（ symbolic execution ）
  - 用 angr 來自動分析 binary
- 以及遇到的問題，要怎麼解決？
  - 符號執行的優化
  - 經驗談更多 angr 用法

先講個分類

- 靜態分析 – IDA
- 動態分析 - GDB

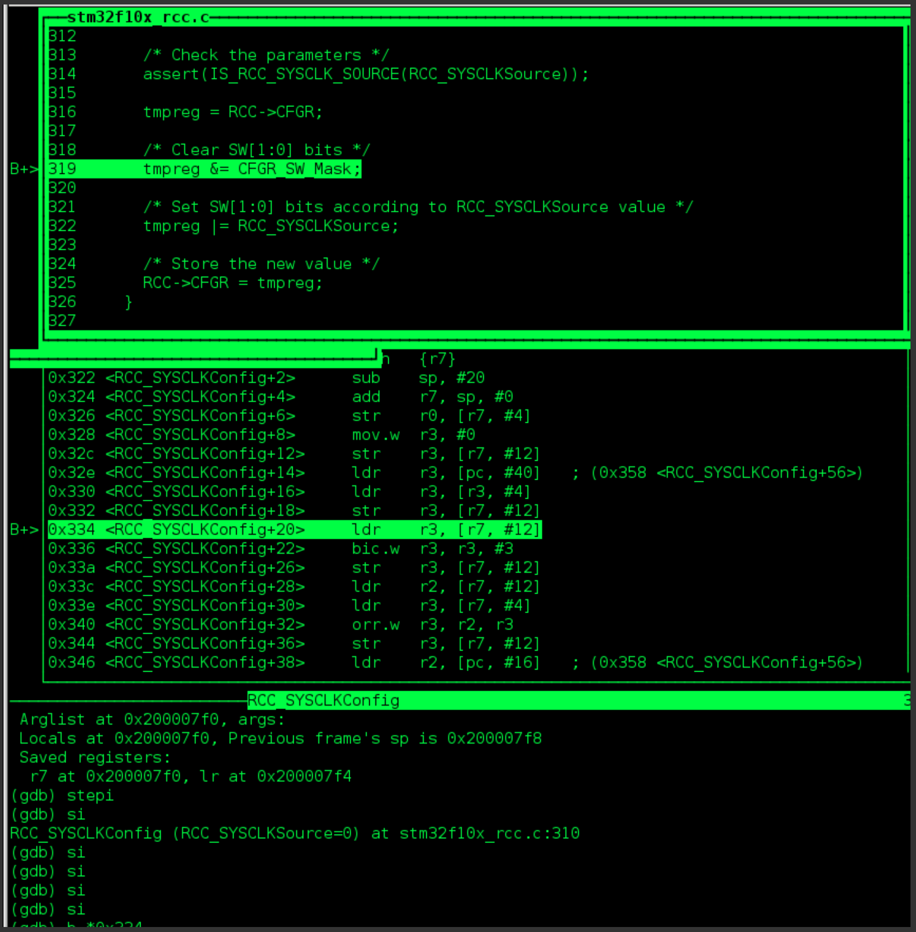
先講個分類

- 靜態分析 – IDA
  - 優點
    - 程式覆蓋
    - 找到很多條程式執行路徑
  - 缺點
    - 該從那開始分析？
    - 怎麼互動？
- 動態分析 - GDB



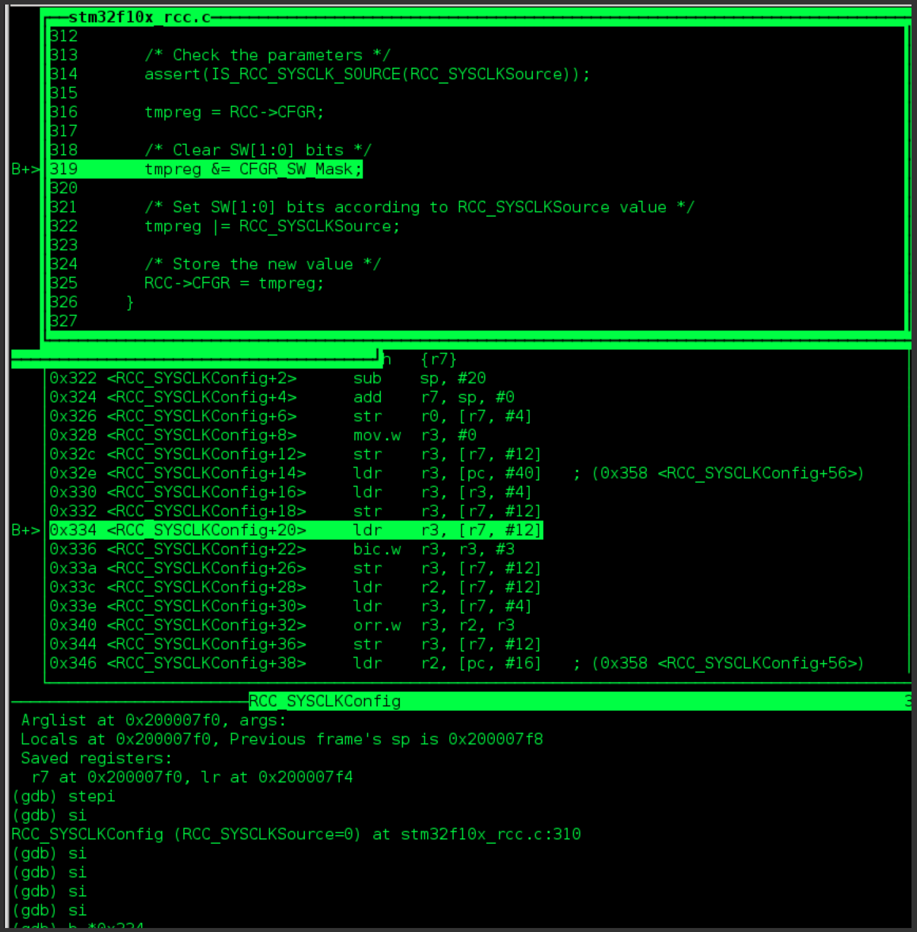
先講個分類

- 靜態分析 – IDA
- 動態分析 – GDB
  - 優點
    - 可以觀察到記憶體、暫存器的值
    - 精確的結果
  - 缺點
    - 程式覆蓋有限
    - 該怎麼模擬環境？



先講個分類

- 靜態分析 – IDA
- 動態分析 – GDB
  - 優點
    - 可以觀察到記憶體、暫存器的值
    - 精確的結果
  - 缺點
    - 程式覆蓋有限
    - 該怎麼模擬環境？



- 如何自動動態分析？自動找 bug ？

Automated Discovery

- Fuzzing
  - 隨機放一堆 input 想辦法讓程式壞掉
- Symbolic Execution
  - 用符號變數來當作 input 而非實際的數值

Symbolic Execution

Symbolic Execution

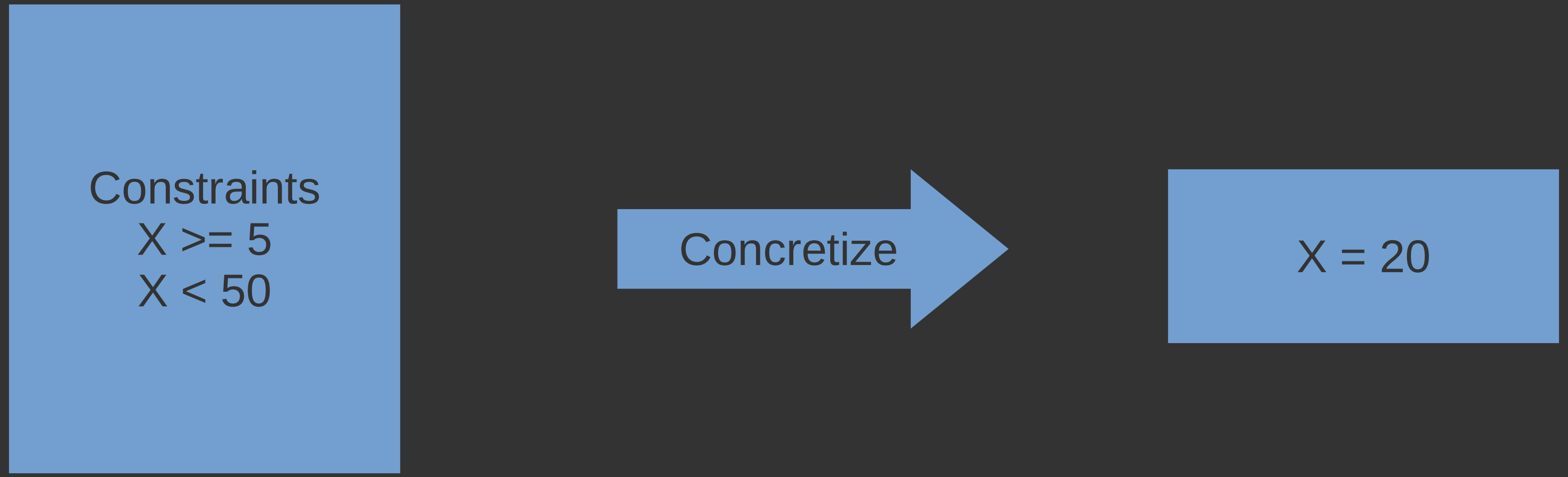
符號執行 [編輯]

符號執行（Symbolic Execution）是一種程序分析技術。其可以通過分析程序來得到讓特定代碼區域執行的輸入。使用符號執行分析一個程序時，該程序會使用符號值作為輸入，而非一般執行程序時使用的具體值。在達到目標代碼時，分析器可以得到相應的路徑約束，然後通過約束求解器來得到可以觸發目標代碼的具體值。[1]

符號模擬技術（symbolic simulation）則把類似的思想用於硬體分析。符號計算（Symbolic computation）則用於數學表達式分析。

Symbolic Execution

- Dynamic analysis
- Set symbolic values and constraints
- Concretize to obtain a possible value



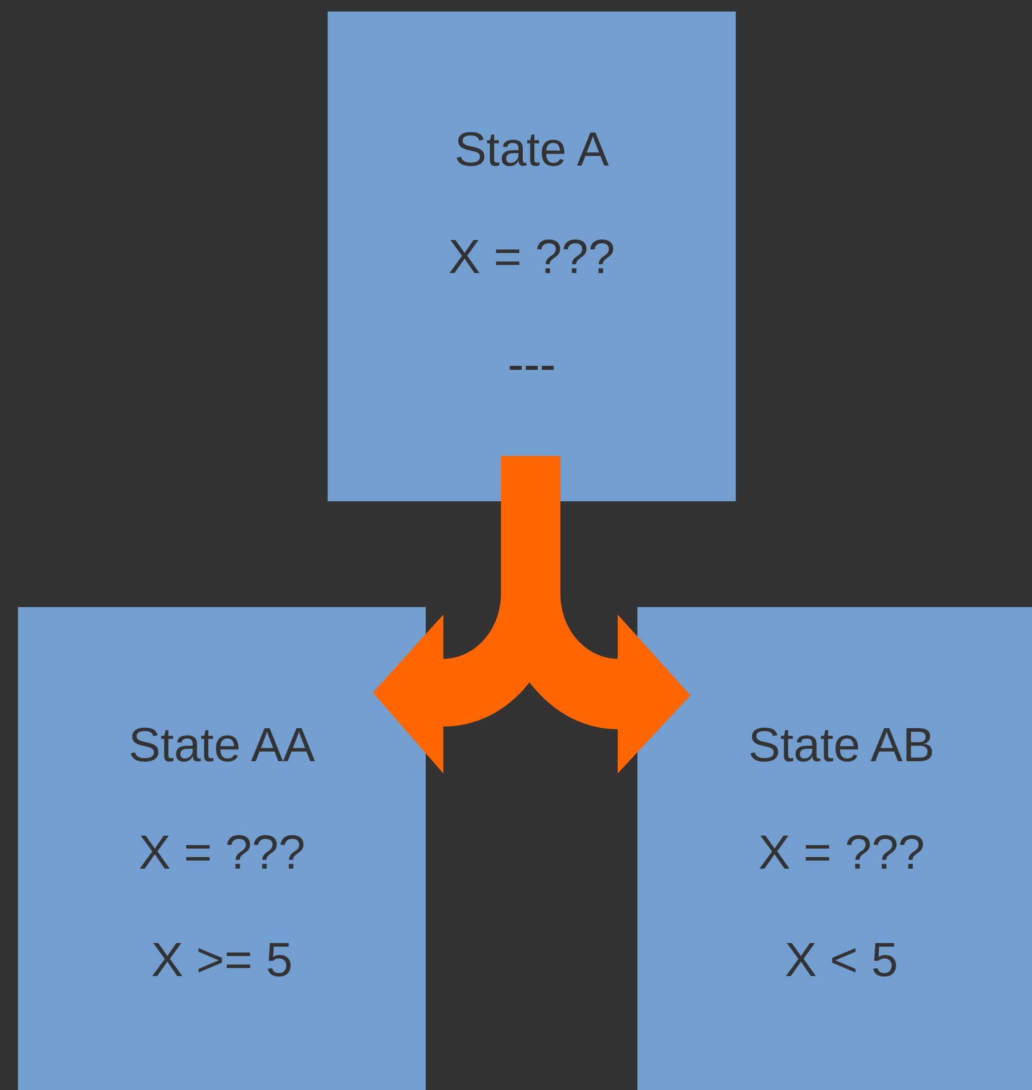
Symbolic Execution

```
x = get_intput();
if (x >= 5)
    if (x < 50)
        bug();          <-- Target
    else
        printf("??");
else
    printf("yo");
```

State A

Symbolic Execution

```
x = get_intput();
if (x >= 5)
    if (x < 50)
        bug();
    else
        printf("??");
else
    printf("yo");
```



Symbolic Execution

```
x = get_intput();
if (x >= 5)
    if (x < 50)
        bug();
    else
        printf("??");
else
    printf("yo");
```

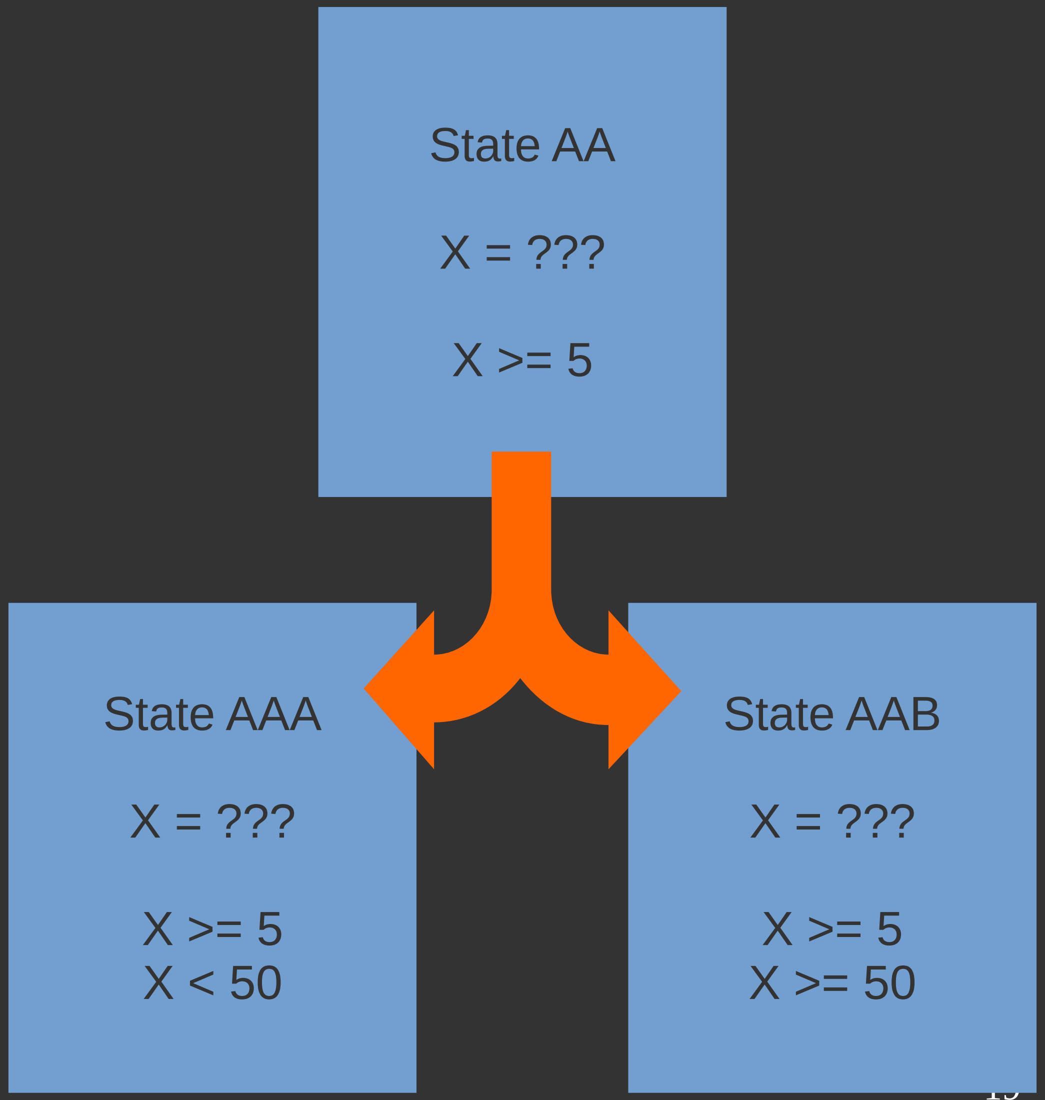
State AA

X = ???

X >= 5

Symbolic Execution

```
x = get_intput();
if (x >= 5)
    if (x < 50)
        bug();
    else
        printf("??");
else
    printf("yo");
```



Symbolic Execution

```
x = get_intput();
if (x >= 5)
    if (x < 50)
        bug();
    else
        printf("??");
else
    printf("yo");
```

State AAA

X = 20

X >= 5
X < 50

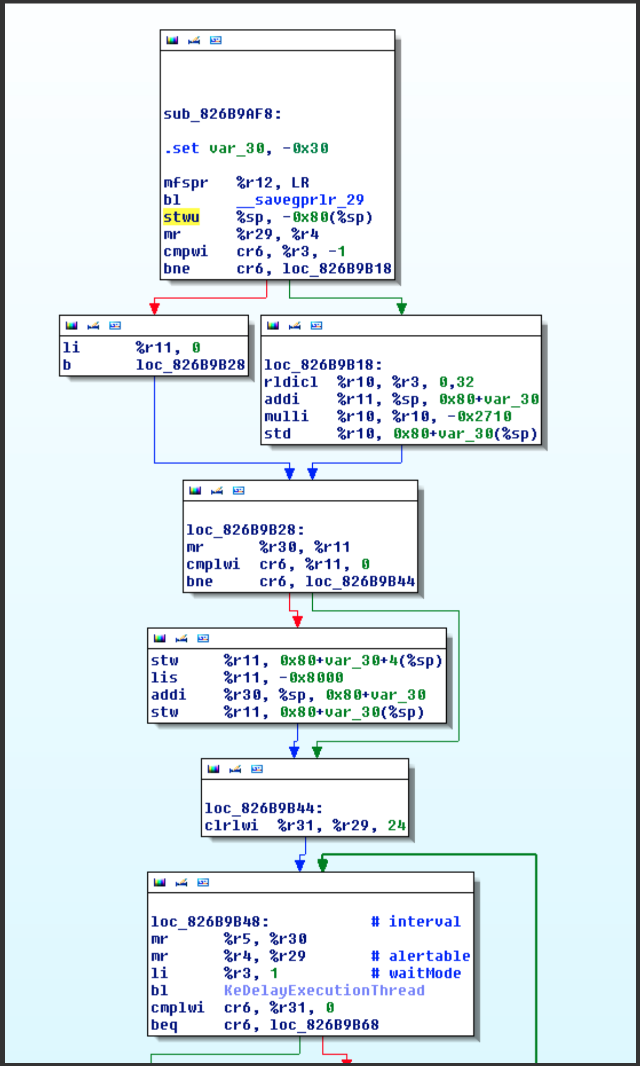
Symbolic Execution

- state 往下走一步就是往下走一個 basic block
- 在探索 path 時會不斷設置符號變數和收集限制式
- 使用 solvers 來解限制式
- 找出一組 input 使得滿足 path 上所有的限制式

Symbolic Execution

- state 往下走一步就是往下走一個 basic block
- 在探索 path 時會不斷設置符號變數和收集限制式
- 使用 solvers 來解限制式
- 找出一組 input 使得滿足 path 上所有的限制式

```
x = get_intput();
if (x >= 5)
    if (x < 50)
        bug();
    else
        printf("??");
else
    printf("yo");
```

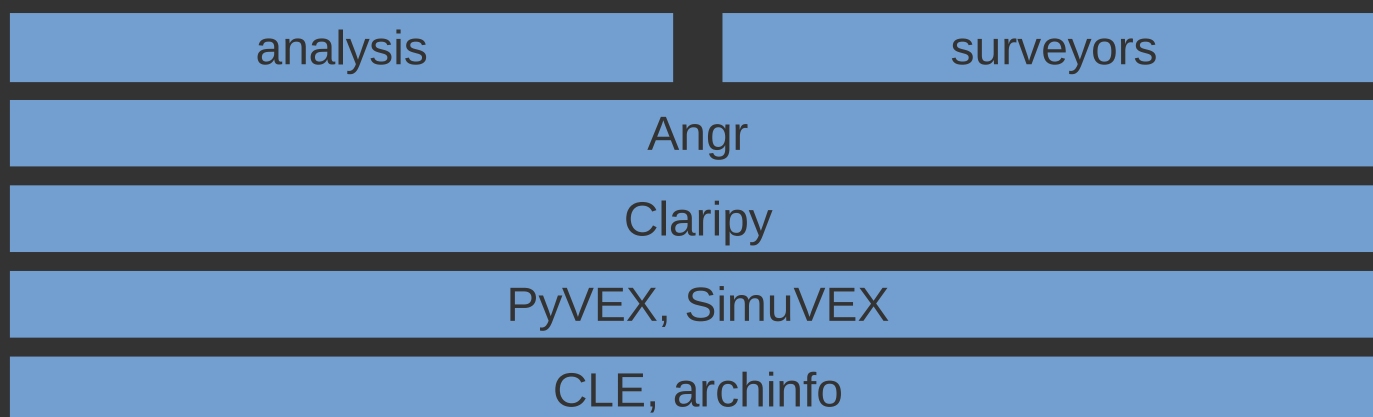


Angr

Angr

- 分析 binary 的框架（不需要 binary 的原始碼）
- 有靜態分析以及動態分析
  - CFG analysis
  - symbolic execution
- 適用於不同平台和 arch 的 binary

Angr


analysis
surveyors
Angr
Claripy
PyVEX, SimuVEX
CLE, archinfo

Angr

- 分析並讀取 binary 的資訊
  - 指令位址、 shared library 、 ...
  - arch information

Angr - CLE

```
>>> print b.loader.find_symbol_got_entry('__libc_start_main')

>>> print b.loader.main_bin.imports
{'__gmon_start__': <cle.elf.ELFRelocation at 0x7f9928941650>,
 '__libc_start_main': <cle.elf.ELFRelocation at 0x7f9928941dd0>,
 '__stack_chk_fail': <cle.elf.ELFRelocation at 0x7f9928941590>,
 'fgets': <cle.elf.ELFRelocation at 0x7f9928941550>,
 'getenv': <cle.elf.ELFRelocation at 0x7f9928406810>,
 'printf': <cle.elf.ELFRelocation at 0x7f99284062d0>,
 'ptrace': <cle.elf.ELFRelocation at 0x7f99286cca10>,
 'puts': <cle.elf.ELFRelocation at 0x7f99284068d0>}
```

Angr - archinfo

```
default_register_values = [
    ( 'esp', Arch.initial_sp, True, 'global' ), # the stack
    ( 'd', 1, False, None ),
    ( 'fpround', 0, False, None ),
    ( 'sseround', 0, False, None ),
    ( 'gdt', 0, False, None ),
    ( 'ldt', 0, False, None ),
    ( 'id', 1, False, None ),
    ( 'ac', 0, False, None )
]
entry_register_values = {
    'eax': 0x1C,
    'edx': 'ld_destructor',
    'ebp': 0
}
default_symbolic_registers = [ 'eax', 'ecx', 'edx', 'ebx', 'esp', 'ebp', 'esi', 'edi', 'eip' ]
register_names = {
    8: 'eax',
    12: 'ecx',
    16: 'edx',
    20: 'ebx',
```

Angr

- 將指令轉換成中間語言（IR)、分析 IR 並且模擬
  - i.e., 不只知道他是什麼，還知道他做了什麼
- state, symbolic memory, SimProcedure ...

Angr - IR

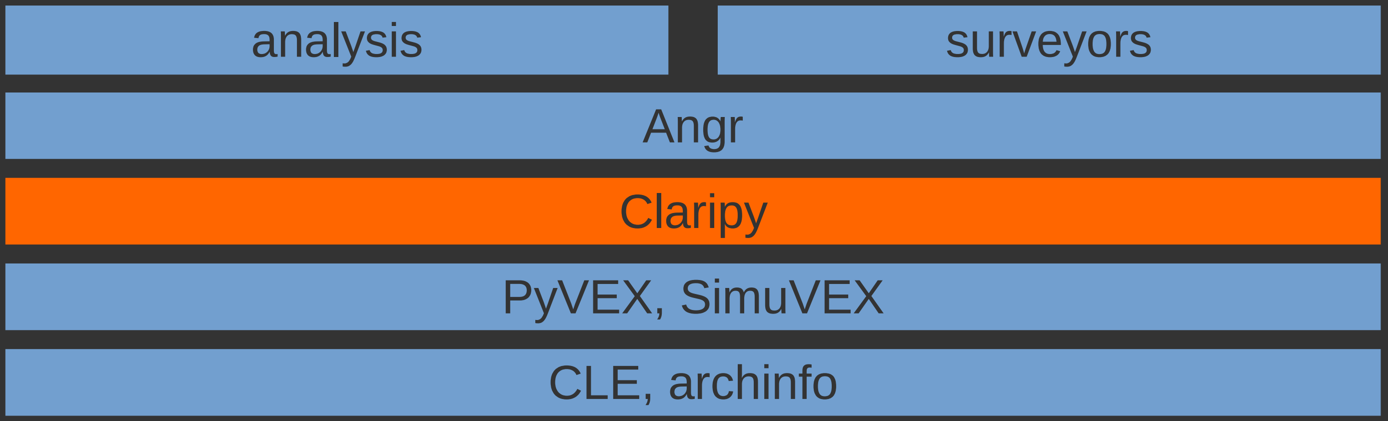
```
0x8000: dec eax
```

→

```
t0 = GET:I32(8)
t1 = Sub(t0, 1)
PUT(8) = t1
PUT(68) = 0x8001
```

Angr

- 設符號變數以及 solver、收集限制式
- 是一個前端界面，而後端可以是各種 solver 像是 z3



Z3 Solver

- 微軟的某項研究
- 有 python API
- ebx = 0x1234, eax = (ebx / ecx) ^ ecx, eax = 2, ecx=?

```
from z3 import *
x = Int('x')
y = Int('y')
s = Solver()
s.add(x > 2, y < 10, x + y == 7)
print s.check()
# sat
m = s.model()
print m
# [y = 0, x = 7]
```

Angr

- 一整個集成符號執行
- path, path_group, factory, ...

Script – Hello Angr

- 腳本初體驗

Script - Demo

```c
int __cdecl main(int argc, const char *
{
  int v3; // ebx@2

  if ( argc == 2 )
  {
    v3 = verify(argv[1]);
    if ( v3 )
    {
      v3 = 0;
      puts("Success!");
    }
    else
    {
      puts("Failure!");
    }
  }
  else
```

```c
*((_DWORD *)v1 + 2) ^= 0x55555555u;
*((_DWORD *)v1 + 3) ^= 0x33333333u;
v13 = (unsigned __int8)(v1[2] ^ v12);
v14 = (unsigned __int8)(v1[3] ^ v1[2] ^ v12);
v15 = (unsigned __int8)(v1[4] ^ v1[3] ^ v1[2] ^ v12);
v16 = (unsigned __int8)(v1[5] ^ v1[4] ^ v1[3] ^ v1[2] ^ v12);
v17 = (unsigned __int8)(v1[6] ^ v1[5] ^ v1[4] ^ v1[3] ^ v1[2] ^ v12);
v18 = v1[8] ^ v1[7] ^ v1[6] ^ v1[5] ^ v1[4] ^ v1[3] ^ v1[2] ^ v12;
v19 = (unsigned __int8)(v1[7] ^ v1[6] ^ v1[5] ^ v1[4] ^ v1[3] ^ v1[2] ^ v12);
*((_BYTE *)v1 + 8) = v18;
v20 = v1[9] ^ v18;
v21 = (unsigned __int8)(v1[10] ^ v20);
v22 = (unsigned __int8)(v1[11] ^ v1[10] ^ v20);
v23 = (unsigned __int8)(v1[12] ^ v1[11] ^ v1[10] ^ v20);
v24 = v1[14] ^ v1[13] ^ v1[12] ^ v1[11] ^ v1[10] ^ v20;
v25 = v1[13] ^ v1[12] ^ v1[11] ^ v1[10] ^ v20;
*((_BYTE *)v1 + 15) ^= v24;
*v1 ^= 0x63u;
*((_BYTE *)v1 + 8) ^= 0x30u;
*((_BYTE *)v1 + 1) = (2 * v12 | ((signed int)v12 >> 1)) ^ 0x2F;
*((_BYTE *)v1 + 2) = (4 * v13 | (v13 >> 2)) ^ 0xDC;
*((_BYTE *)v1 + 3) = (8 * v14 | (v14 >> 3)) ^ 0x20;
*((_BYTE *)v1 + 4) = (16 * v15 | (v15 >> 4)) ^ 0xCD;
*((_BYTE *)v1 + 5) = (32 * v16 | (v16 >> 5)) ^ 0xA0;
*((_BYTE *)v1 + 6) = (((_BYTE)v17 << 6) | (v17 >> 6)) ^ 0x83;
*((_BYTE *)v1 + 7) = ((_BYTE)v19 << 7) | (v19 >> 7);
*((_BYTE *)v1 + 9) = (2 * v20 | ((signed int)v20 >> 1)) ^ 0x7D;
*((_BYTE *)v1 + 10) = (4 * v21 | (v21 >> 2)) ^ 0x19;
*((_BYTE *)v1 + 11) = (8 * v22 | (v22 >> 3)) ^ 4;
*((_BYTE *)v1 + 12) = (16 * v23 | (v23 >> 4)) ^ 0xC4;
```

Script – Hello Angr

- Surveyors

```
import angr

p = angr.Project("test")
ex = p.surveyors.Explorer(find=(0x400844, ), avoid=(0x400855,))
ex.run()

print ex.found[0].state.posix.dumps(0)
```

Script – Hello Angr

- path_group

```
import angr

p = angr.Project("test")

initial_state = p.factory.entry_state()
pg = p.factory.path_group(initial_state)

pg.explore(find=(0x4005d1,))
print pg
# <PathGroup with 18 deadended, 4 active, 1 found>
print pg.found[0]
# <Path with 64 runs (at 0x4005d1)>
print pg.found[0].state.posix.dumps(0)
# input_string
```

Script – Hello Angr

- SimState
  - entry_state: a SimState initialized to the program state at the binary's entry point
  - blank_state: a SimState object with little initialization

SimState

- symbolic memory
- symbolic registers
- constraints

```
>>> import angr
>>> b = angr.Project('/bin/true')

>>> s = b.factory.blank_state(addr=0x08048591)
>>> s = b.factory.entry_state()

# The first 5 bytes of the binary
>>> print s.memory.load(b.loader.min_addr(), 5)
```

Script - ARGS

- 如何設 args ？

Script - ARGS

- 如何設 args ？

```
import angr
import claripy

p = angr.Project("test")

args = claripy.BVS('args', 8*16)
initial_state = prog.factory.entry_state(args=["./vul", args])
pg = p.factory.path_group(initial_state)

pg.explore(find=(0x4005d1,))
print pg
# <PathGroup with 18 deadended, 4 active, 1 found>
print pg.found[0]
# <Path with 64 runs (at 0x4005d1)>
print pg.found[0].state.posix.dumps(0)
# input_string
```

Script - ARGS

- Claripy frontends

```
# Create a 32-bit symbolic bitvector "x"
>>> claripy.BVS('x', 32)

# Create a 32-bit bitvectory with the value 0x12345678
>>> claripy.BVV(0x12345678, 32)
<BV32 BVV(0x12345678, 32)>
```

Script – Memory Access

- 如何在記憶體位址上放符號變數？
  - 方便我們追蹤並求解記憶體位址上的值

Script – Memory Access

- 如何在記憶體位址上放符號變數？

```
import angr

p = angr.Project('./vul')
s = p.factory.blank_state(addr=0x80485c8)

bvs = s.se.BVS('to_memory', 8*4)
s.se.add(bvs > 1000)
s.memory.store(0x08049b80, bvs, endness='Iend_LE')

pg = p.factory.path_group(s, immutable=False)

...
```

Script – Memory Access

Reverse	Reverses a bit expression.	`claripy.Reverse(x)` or `x.reversed`
And	Logical And (on boolean expressions)	`claripy.And(x == y, x > 0)`
Or	Logical Or (on boolean expressions)	`claripy.Or(x == y, y < 10)`
Not	Logical Not (on a boolean expression)	`claripy.Not(x == y)` is the same as `x != y`
If	An If-then-else	Choose the maximum of two expressions: `claripy.If(x > y, x, y)`

Script – Memory Access

- Accessing Data
- s.se is the solver engine of the state

```
# get the integer
>>> print s.se.any_int(s.regs.rax)
# get the string
>>> print s.se.any_str(s.memory.load(0x1000, 10, endness='Iend_LE'))

# storing data
>>> s.regs.rax = aaaa
>>> s.memory.store(0x1000, aaaa, endness='Iend_LE')
>>> s.memory.store(s.regs.rax, aaaa, endness='Iend_LE')
```

Script – Posix

- 如何對 stdin 的內容加上限制式？

Script – Posix

- 如何對 stdin 的內容加上限制式？

```
p = angr.Project('./vul')

st = p.factory.full_init_state(args=['./vul'])

# Constrain the first 28 bytes to be non-null and non-newline
for _ in xrange(28):
    k = st.posix.files[0].read_from(1)
    st.se.add(k != 0)
    st.se.add(k != 10)

# Constrain the last byte to be a newline
k = st.posix.files[0].read_from(1)
st.se.add(k == 10)

# Reset the symbolic stdin's properties and set its length
st.posix.files[0].seek(0)
st.posix.files[0].length = 29

...
```

Optimization

Optimization

- 實際用 angr 跑，會發現
  - 跑了幾個小時都還沒找到目標路徑
  - 跑著跑著就壞了
- 自動分析似乎很美好，但卻隱藏很多問題 ...

Optimization

- Environment
  - shared library
- Exploration Strategy
  - BFS
  - DFS
- Explosion
  - path explosion
  - path pruning

Environment

- 情境
  - 對符號執行來說， libc 裡複雜無比，一旦進入 libc function 分析可能就掛在裡面了
  - Crypto function
  - 看不懂的 syscall

Environment

- SimProcedure

```
p = angr.Project('./vul',
                  load_options={'auto_load_libs': True},
                  use_sim_procedures=True,
                  exclude_sim_procedures_func='strcmp')
```

- Hook symbol

```
class my_strcmp(simuvex.SimProcedure):
    def run(self):
        ...
        return ...

p.hook_symbol('strcmp', my_strcmp)
```

- Go into library

Environment

- Hook

```
'''
$ objdump -M intel -d ./vul | grep -A2 85d7
 80485d7:       e8 9f 00 00 00          call   804867b
 80485dc:       89 44 24 10             mov    DWORD PTR [esp+0x10],eax
 80485e0:       83 7c 24 10 ff          cmp    DWORD PTR [esp+0x10],0xffffffff
'''

def check1(state):
    state.regs.eax = 20
p.hook(0x080485d7, check1, length=5)
```

- Unknown syscall

```
initial_state = project.factory.entry_state(
    args=[project.filename, arg1],
    add_options={'BYPASS_UNSUPPORTED_SYSCALL'})
```

Exploration Strategy

- Exploration techniques

```
pg = p.factory.path_group(initial_state, immutable=False)
pg.use_technique(angr.exploration_techniques.DFS())

# pg.explore(find=(0x08041234, ))
pg.run(step_func=my_find_func)
```

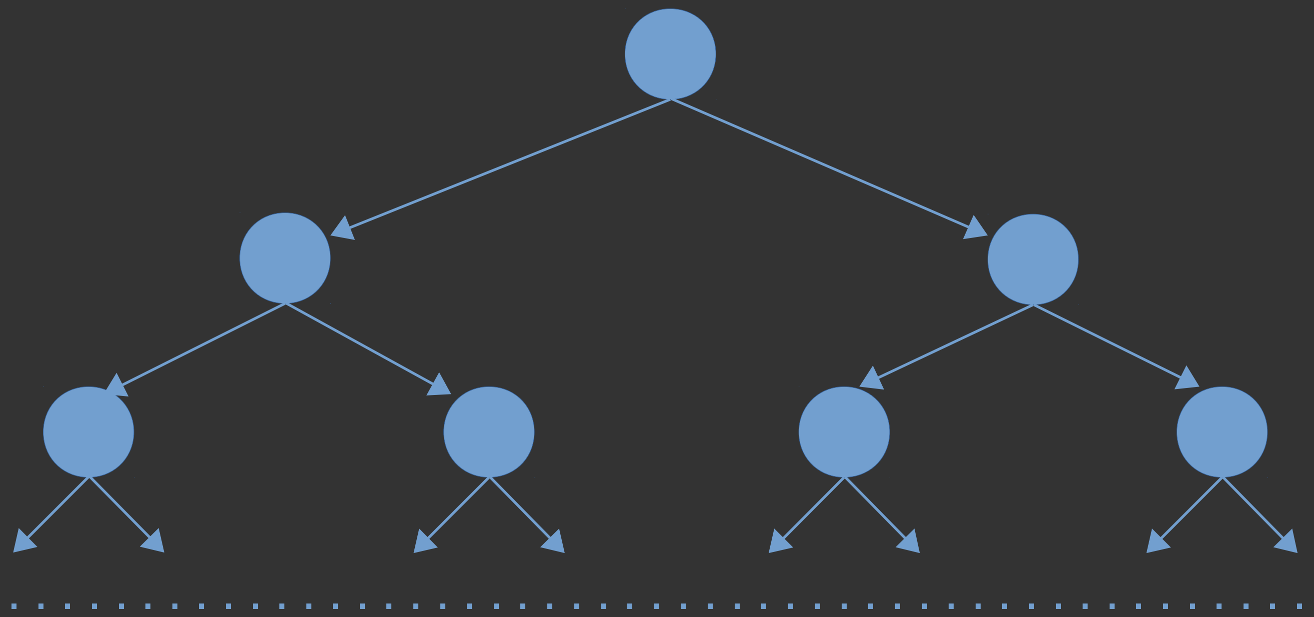
Explosion

- 情境

```
int counter = 0, values = 0;
for(i=0; i<100; i++){
    if(input[i] == 'B'){
        counter++;
        values += 2;
    }
}
if(counter == 75)
    bug();
```

Explosion

- 情境



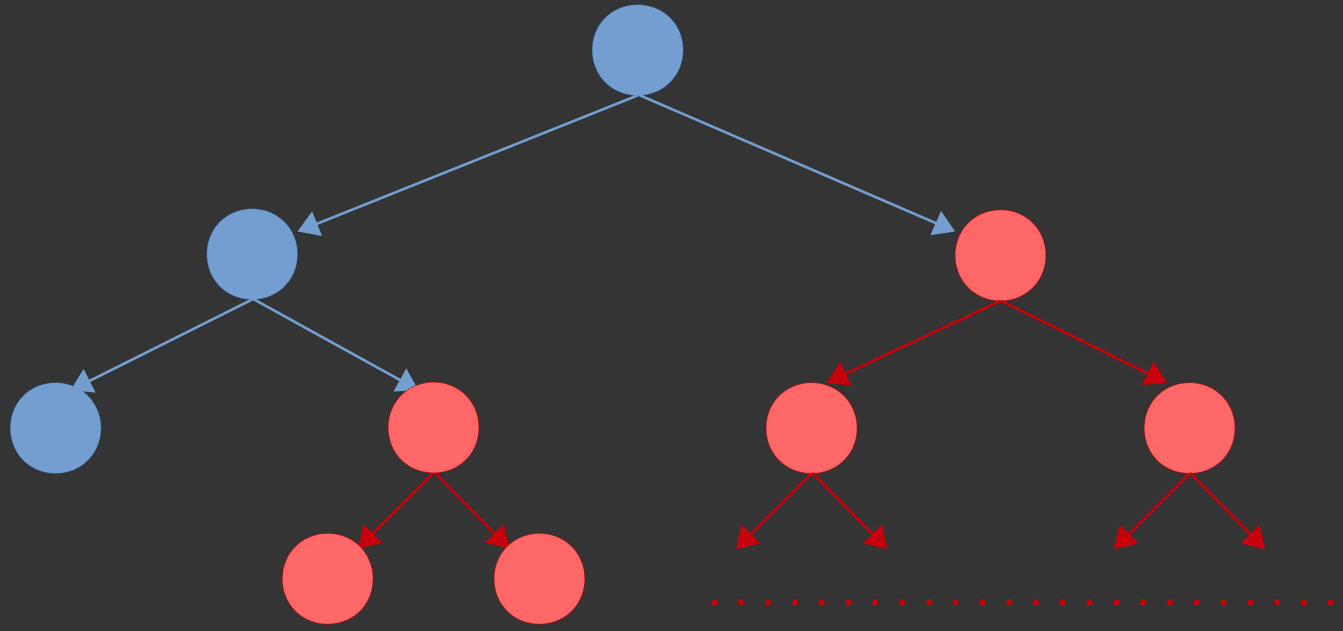
Explosion

- Veritesting
  - 結合靜態符號執行以及動態符號執行
  - 把限制式全部合併在一條路徑上
  - 減少 path explosion 的影響

```
pg = p.factory.path_group(initial_state, immutable=False, veritesting=True)
```

Explosion

- 情境
  - Unsatisfiable path 代表這條路不可能發生，即無法產生任何一組 input 使得 binary 可以照這條路執行



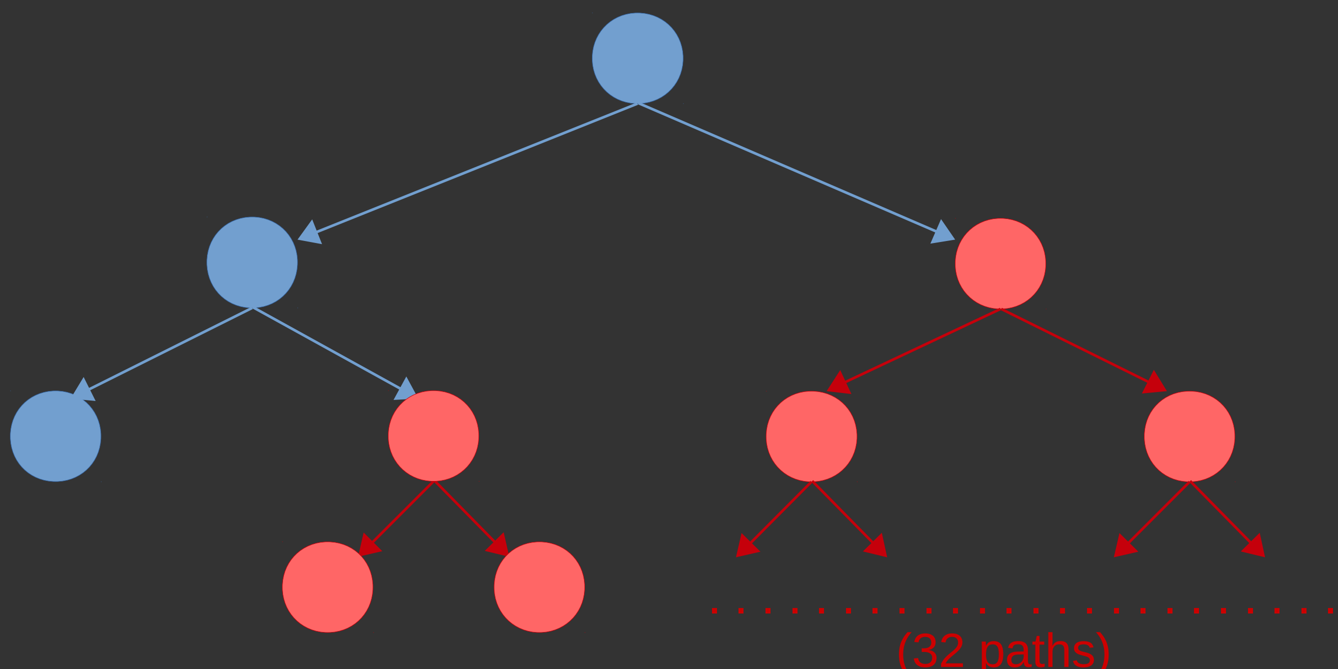
Explosion

- LAZY_SOLVES
  - 懶得檢查，意思是當路徑探索完的時候才進行檢查
  - 預設是開啟的

```
initial_state = project.factory.entry_state(args=[project.filename, arg1])
initial_state.options.discard('LAZY_SOLVES')
```

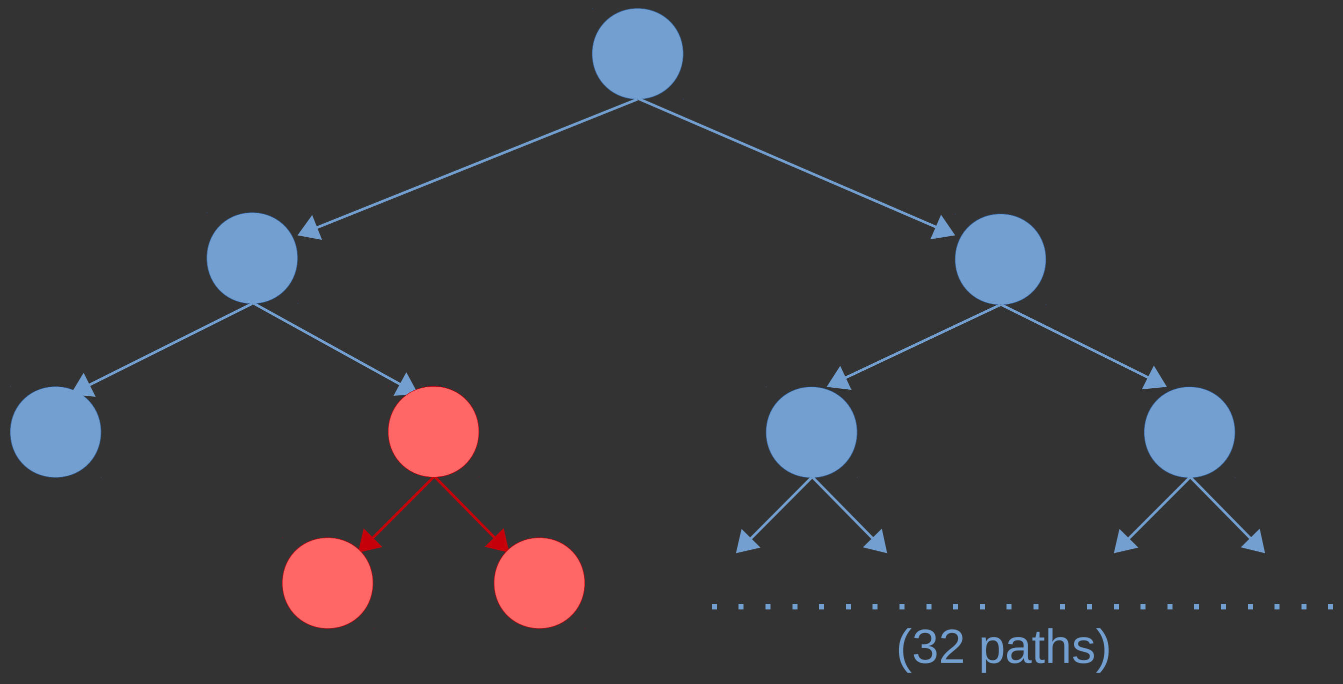
Explosion

- Without LAZY_SOLVES
  - Checked 5 paths
  - Pruned 2 paths
- LAZY_SOLVES
  - Checked 35 paths
  - Pruned 34 paths



Explosion

- Without LAZY_SOLVES
  - Checked 67 paths
  - Pruned 1 path
- LAZY_SOLVES
  - Checked 35 paths
  - Pruned 2 paths



Explosion

- Dynamic path pruning
  - 根據已經檢查的路徑們，推估現在 unsatisfiable path 的比例
  - 依照 unsatisfiable path 的比例調整之後路徑要不要進行檢查的機率

Other Debug Options

- REVERSE_MEMORY_NAME_MAP
  - 保留對記憶體位址的資訊，讓我們可以拿 BVS 的名字 （ 'file_/dev/stdin' ）來得到模擬的記憶體位址（ 0xffff1234 ）
- TRACK_ACTION_HISTORY
  - 方便查看之前所模擬執行過的狀態的 ACTION 紀錄

Demo

結論

- 現在流行自動打 CTF
- Angr 各種腳本寫法以及優化小技巧
- 單用 symbolic execution 做自動分析其實還不夠

Reference

- Symbolic Execution
  - Angr: http://angr.io/
  - KLEE: https://klee.github.io/
  - Triton: http://triton.quarkslab.com/
- My blog: http://ysc21.github.io/

Q & A