Symbolic Execution Concepts

CS6262

Symbolic Execution Concepts

- The examples and concepts are prepared base on
  - MITOPENCOURSEWARE
  - Computer Systems Security
  - Lecture 10 : Symbolic Execution
  - Instructor : Armando Solar-Lezama
    - https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-858-computer-systems-security-fall-2014/video-lectures/lecture-10-symbolic-execution/

Sample Program

- Is // target activity could ever be triggered by any input?
- Is there any (x,y) input tupple make // target activity point reachable?

```
void foo(int x, int y) {
    int t = 0;
    if(x > y) {
        t = x;
    } else {
        t = y;
    }
    if(t < x){
        // target activity
    }
}
```

Concrete Execution

- foo(4,4);

Line	X	Y	T
2	4	4	0

```
1 void foo(int x, int y) {
2     int t = 0;
3     if(x > y) {
4         t = x;
5     } else {
6         t = y;
7     }
8     if(t < x){
9         // target activity
10    }
11 }
```

Concrete Execution

- foo(4,4);

Line	X	Y	T
2	4	4	0
3	x > y = FALSE		

```
1  void foo(int x, int y) {
2      int t = 0;
3      if(x > y) {
4          t = x;
5      } else {
6          t = y;
7      }
8      if(t < x){
9          // target activity
10     }
11 }
```

Concrete Execution

- foo(4,4);

Line	X	Y	T
2	4	4	0
3	x > y = FALSE		
6	4	4	**4**

```
1  void foo(int x, int y) {
2      int t = 0;
3      if(x > y) {
4          t = x;
5      } else {
6          t = y;
7      }
8      if(t < x){
9          // target activity
10     }
11 }
```

Concrete Execution

- foo(4,4);

Line	X	Y	T
2	4	4	0
3	x > y = FALSE		
6	4	4	**4**
8	t < x = FALSE		

```
1 void foo(int x, int y) {
2     int t = 0;
3     if(x > y) {
4         t = x;
5     } else {
6         t = y;
7     }
8     if(t < x){
9         // target activity
10    }
11 }
```

Concrete Execution

- foo(4,4);
- What does it tell us?
    - No activity occurred for x=4 and y=4
- What else?

```
void foo(int x, int y) {
    int t = 0;
    if(x > y) {
        t = x;
    } else {
        t = y;
    }
    if(t < x){
        // target activity
    }
}
```

Concrete Execution

- foo(4,4);
- We don't know anything about other executions
- All we learn is // target activity is not reachable when x=4 and y=4

```c
void foo(int x, int y) {
    int t = 0;
    if(x > y) {
        t = x;
    } else {
        t = y;
    }
    if(t < x){
        // target activity
    }
}
```

Concrete Execution

- foo(2,1);

Line	X	Y	T
2	2	1	0
3	x > y = TRUE		
4	2	1	**2**
8	t < x = FALSE		

```
1  void foo(int x, int y) {
2      int t = 0;
3      if(x > y) {
4          t = x;
5      } else {
6          t = y;
7      }
8      if(t < x){
9          // target activity
10     }
11 }
```

Concrete Execution

- foo(2,1);
- What we learn about the program?
    - foo(2,1) will not reach // target activity
    - No information about other executions
- How many times to perform concrete execution to learn about the behavior of the program?
- How to learn the program behavior?
    - Generalize the input space to understand the program

```
void foo(int x, int y) {
    int t = 0;
    if(x > y) {
        t = x;
    } else {
        t = y;
    }
    if(t < x){
        // target activity
    }
}
```

Symbolic Execution

- Use symbolic values instead of concrete values

Line	X	Y	T
2	x	y	0

```
1 void foo(int x, int y) {
2     int t = 0;
3     if(x > y) {
4         t = x;
5     } else {
6         t = y;
7     }
8     if(t < x){
9         // target activity
10    }
11 }
```

Symbolic Execution

- What could the variable **t** hold at the 8th line?

Line	X	Y	T
2	x	y	0
8	x	y	t_1

```
1  void foo(int x, int y) {
2      int t = 0;
3      if(x > y) {
4          t = x;
5      } else {
6          t = y;
7      }
8      if(t < x){
9          // target activity
10     }
11 }
```

Symbolic Execution

- Depends on which branch is taken

Line	X	Y	T
2	x	y	0
8	x	y	t_1

$$t_1 = \begin{cases} x: \ x > y \ (line\ 2) \\ y: x \leq y \end{cases}$$

```
1  void foo(int x, int y) {
2      int t = 0;
3      if(x > y) {
4          t = x;
5      } else {
6          t = y;
7      }
8      if(t < x){
9          // target activity
10     }
11 }
```

Symbolic Execution

- Use symbolic values instead of concrete values

Line	X	Y	T
2	x	y	0
8	x	y	t_1

$$t_1 = \begin{cases} x: & x > y \ (line\ 2) \\ y: & x \le y \end{cases}$$

- Does the check on the 8th line hold?

```
1  void foo(int x, int y) {
2      int t = 0;
3      if(x > y) {
4          t = x;
5      } else {
6          t = y;
7      }
8      if(t < x){
9          // target activity
10     }
11 }
```

Symbolic Execution

- Use symbolic values instead of concrete values

Line	X	Y	T
2	x	y	0
8	x	y	t_1

- Current formula

$$\frac{\boldsymbol{t1 < x}}{\begin{array}{c} \boldsymbol{x > y => t_1 = x} \\ \boldsymbol{x \le y => t_1 = y} \end{array}} = ?$$

```
1 void foo(int x, int y) {
2       int t = 0;
3       if(x > y) {
4           t = x;
5       } else {
6           t = y;
7       }
8       if(t < x){
9           // target activity
10      }
11 }
```

Symbolic Execution

- Use symbolic values instead of concrete values

Line	X	Y	T
2	x	y	0
8	x	y	t_1

- Current formula

$$\frac{\boldsymbol{t1 < x}}{\begin{array}{c}\boldsymbol{x > y => t_1 = x}\\ \boldsymbol{x \leq y => t_1 = y}\end{array}} = \emptyset$$

```
1  void foo(int x, int y) {
2        int t = 0;
3        if(x > y) {
4            t = x;
5        } else {
6            t = y;
7        }
8        if(t < x){
9            // target activity
10       }
11 }
```

Symbolic Execution

- // target activity is not reachable

$$\frac{\boldsymbol{t1 < x}}{\boldsymbol{x > y => t_1 = x}} = \emptyset$$
$$\boldsymbol{x \leq y => \ t_1 \ = y}$$

- If x > y holds

$$x > y \rightarrow \ t_1 = x \ \vdash t < x = FALSE$$

- Else

$$x \leq y \rightarrow \ t_1 = y \rightarrow x \leq \ t_1 \ \vdash t < x = FALSE$$

```
void foo(int x, int y) {
    int t = 0;
    if(x > y) {
        t = x;
    } else {
        t = y;
    }
    if(t < x){
        // target activity
    }
}
```

Symbolic Execution

- No matter what inputs are provided to the function **foo**, the program will not go through // target activity

```
void foo(int x, int y) {
    int t = 0;
    if(x > y) {
        t = x;
    } else {
        t = y;
    }
    if(t < x){
        // target activity
    }
}
```

Symbolic Execution

- What about this program?

```c
void foo(int x, int y) {
    int t = 0;
    if(x > y) {
        t = x - 1;
    } else {
        t = y;
    }
    if(t < x){
        // target activity
    }
}
```

Symbolic Execution

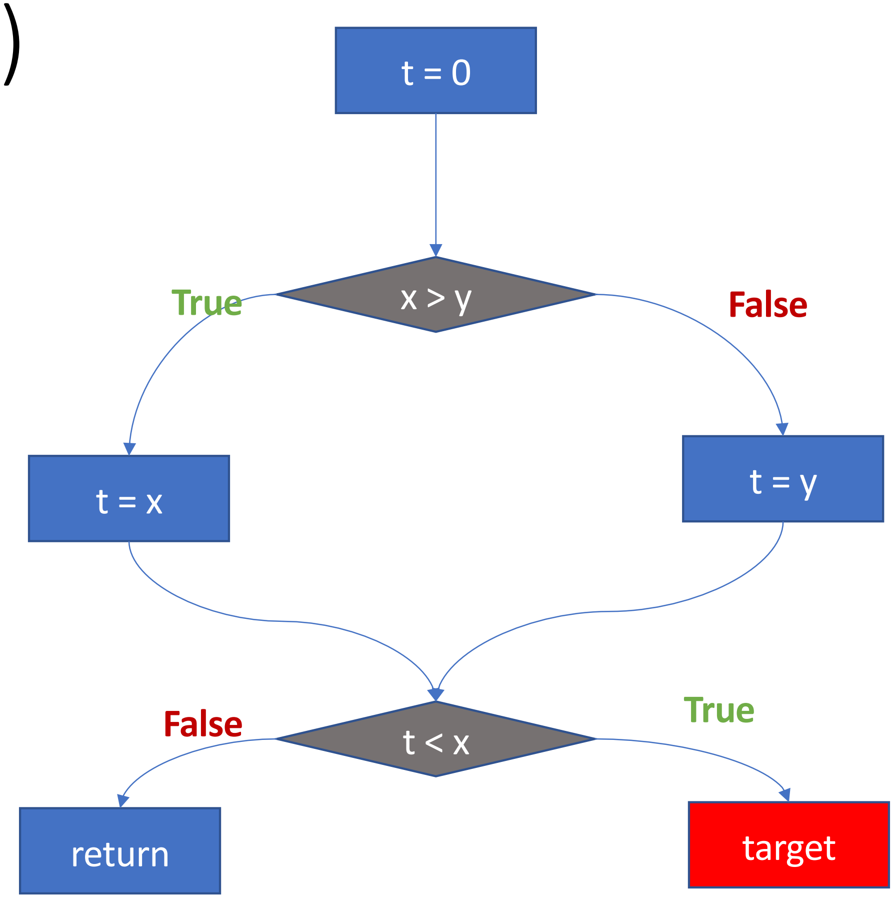
- This function can reach the target whenever **x > y**

$$\frac{t1 < x}{x > y => t_1 = x - 1} = \{x > y\}$$
$$x \leq y => t_1 = y$$

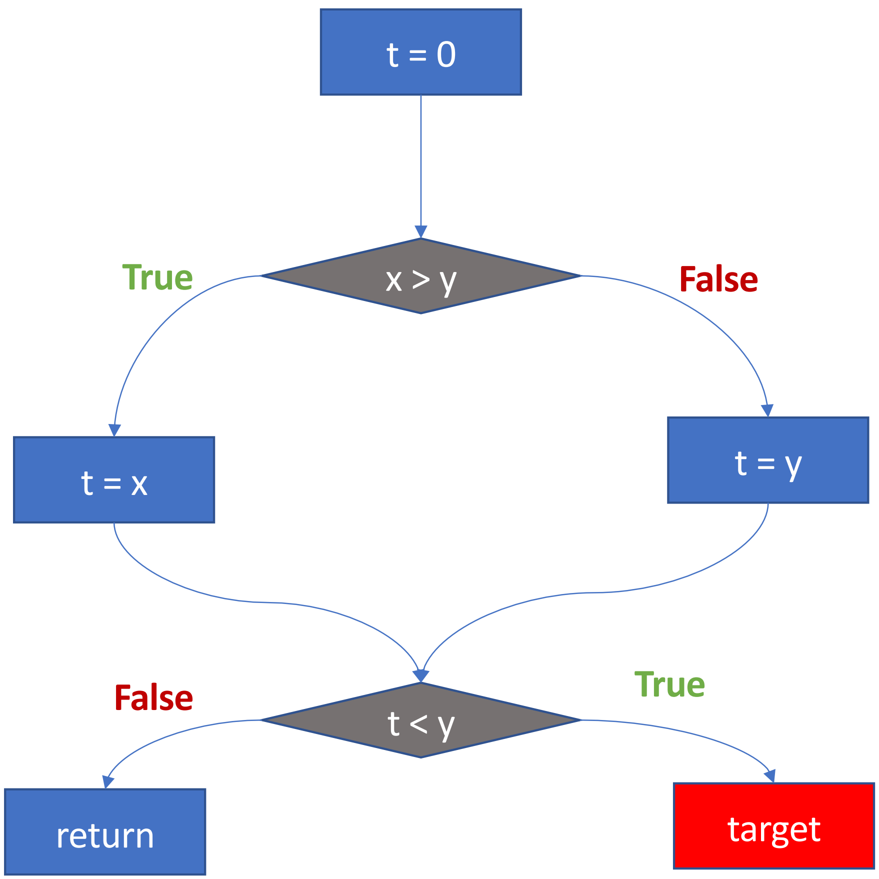
```
void foo(int x, int y) {
    int t = 0;
    if(x > y) {
        t = x - 1;
    } else {
        t = y;
    }
    if(t < x){
        // target activity
    }
}
```

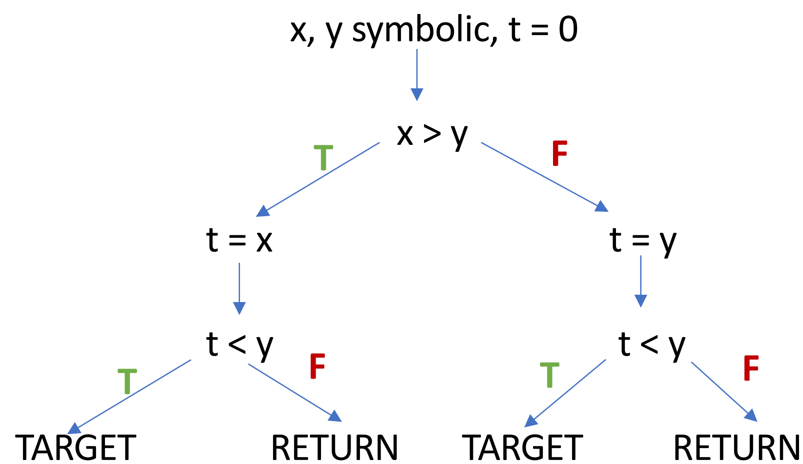
Control Flow Graph (CFG)

```
void foo(int x, int y) {
    int t = 0;
    if(x > y) {
        t = x;
    } else {
        t = y;
    }
    if(t < x){
        // target activity
    }
}
```



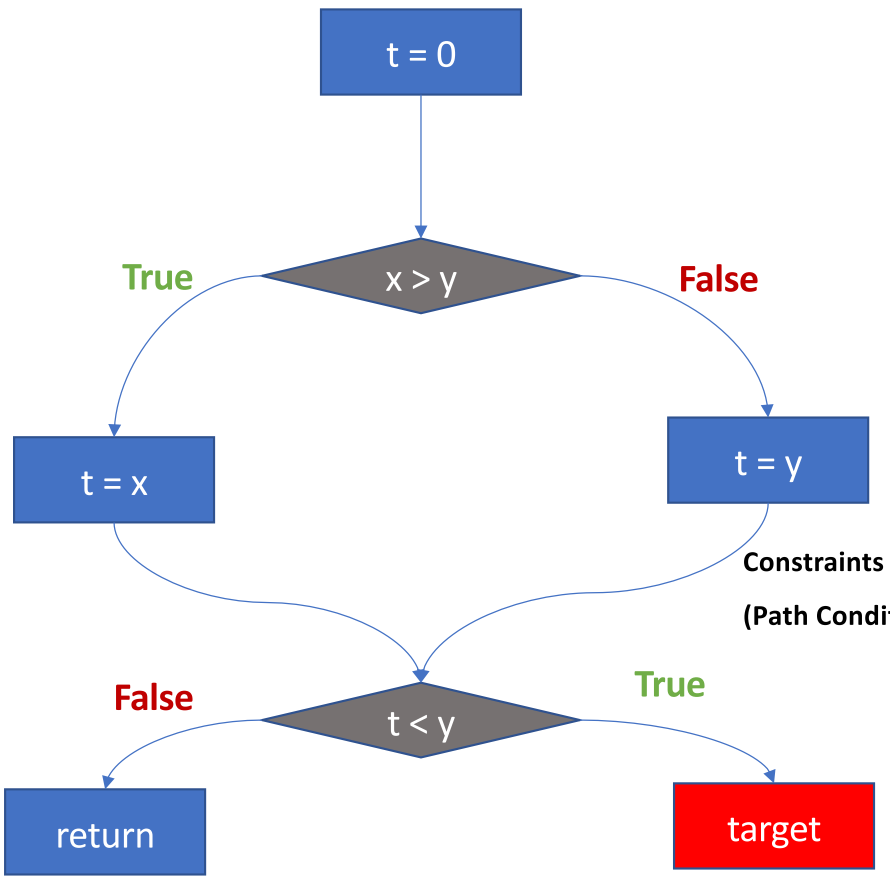
Symbolic Execution

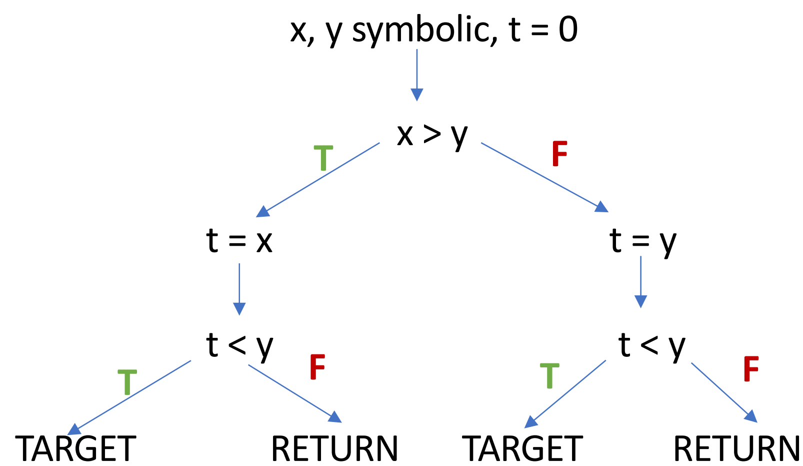




No more path left to explore in CFG!

Control Flow Graph (CFG)



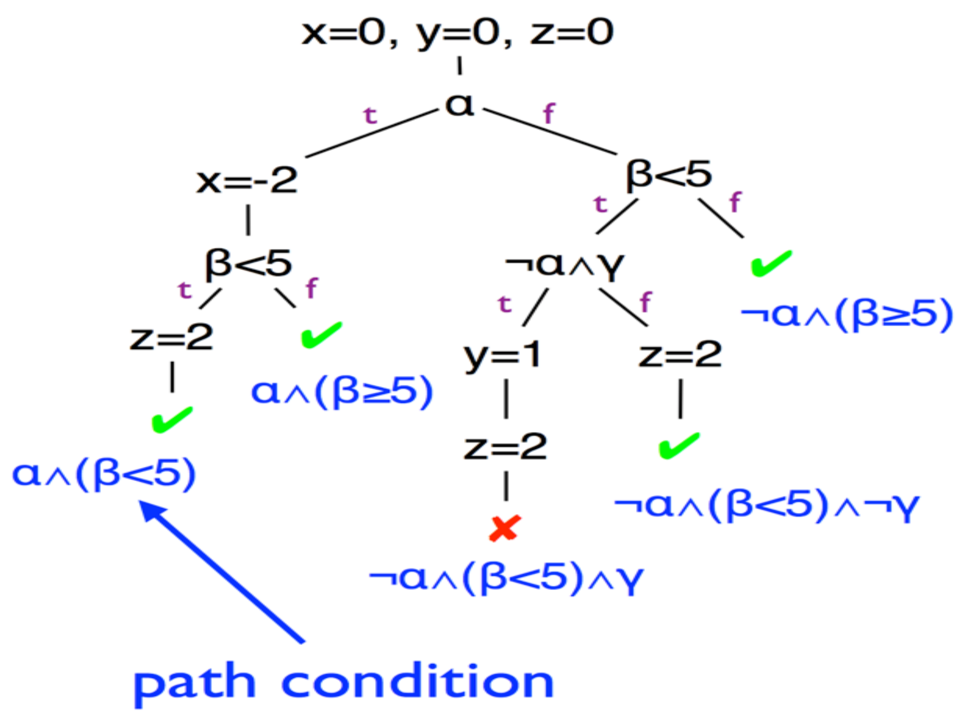


Constraints	$x > y \wedge t < y$ where $t = x$	$\neg (x > y) \wedge t < y$ where $t = y$
(Path Conditions)	$= x > y \wedge x < y$	$= \neg (x > y) \wedge y < y$
	= **FALSE**	$= \neg (x > y) \wedge$ **FALSE** = **FALSE**

The above conditions are the path constrains for TARGET. These path constraints show that both paths down to TARGET is not satisfiable. Hence, the program will never execute TARGET state.

Another Example

```
int a = α, b = β, c = γ;
                    // symbolic
int x = 0, y = 0, z = 0;
if (a) {
  x = -2;
}
if (b < 5) {
  if (!a && c)  { y = 1; }
  z = 2;
}
assert(x+y+z!=3)
```



Binary Program

- How to perform symbolic execution on binaries?
  - Symbolic Execution on 32-bit or 64-bit integers
- More complex arithmetic operations required

More Information

- angr: a concolic execution engine for binaries written in python
  - http://angr.io/
- z3: a theorem prover supports Java, C++, python etc.
  - https://rise4fun.com/Z3
  - https://github.com/Z3Prover/z3

angr Tutorial

angr – SimState

- While angr perform symbolic execution, it stores the current state of the program in the **SimState** objects.
- SimState is a structure that contains the program's memory, register and other information.
- SimState provides interaction with memory and registers. For example, **state.regs** offers read, write accesses with the name of each registers such as state.regs.eip, state.regs.rbx, state.regs.ebx, state.regs.ebh

angr - SimState

- Creating an empty 64 bit SimState

```
(angr) analysis@analysis-VirtualBox:~/report$ python
Python 2.7.12 (default, Jul  1 2016, 15:12:24)
[GCC 5.4.0 20160609] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import angr
WARNING | 2019-02-22 16:23:44,860 | angr.analyses.disassembly_utils | Your
>>> state = angr.SimState(arch='AMD64')
WARNING | 2019-02-22 16:24:02,100 | angr.sim_state | SimState defaulting to
>>> state
<SimState @ <BV64 reg_rip_0_64{UNINITIALIZED}>>
>>> 
```

angr - Bitvectors

- Since, we are dealing with binary files, we don’t deal with regular integers.
- In binary program, everything becomes bits and sequence of bits.
- A bitvector is a sequence of bits used to perform integer arithmetic for symbolic execution.

angr - Bitvectors

- Creating some 32 bit bitvector values
- state.solver.BVV(4,32) will create 32 bit length bitvector with value 4
- We can perform arithmetic operations or comparisons using the bitvectors

```
>>> four = state.solver.BVV(4,32)
>>> four
<BV32 0x4>
>>> seven = state.solver.BVV(7, 32)
>>> seven
<BV32 0x7>
>>> total = seven + four
>>> total
<BV32 0xb>
>>> state.solver.eval(four > seven)
False
>>> state.solver.eval(four < seven)
True
>>> state.solver.eval(four == seven)
False
>>>
```

angr – Symbolic Bitvectors

- **state.solver.BVS('x', 32)** will create a symbolic variable named x with 32 bit length
- Angr allows us to perform arithmetic operation or comparisons using them.

```
>>> x = state.solver.BVS('x', 32)
>>> x
<BV32 x_5_32>
>>> y = state.solver.BVS('y', 32)
>>> y
```

```
>>> x + y
<BV32 x_5_32 + y_6_32>
>>> x + four
<BV32 x_5_32 + 0x4>
>>> x + seven
<BV32 x_5_32 + 0x7>
>>> x + y
<BV32 x_5_32 + y_6_32>
>>>
```

angr - Registers

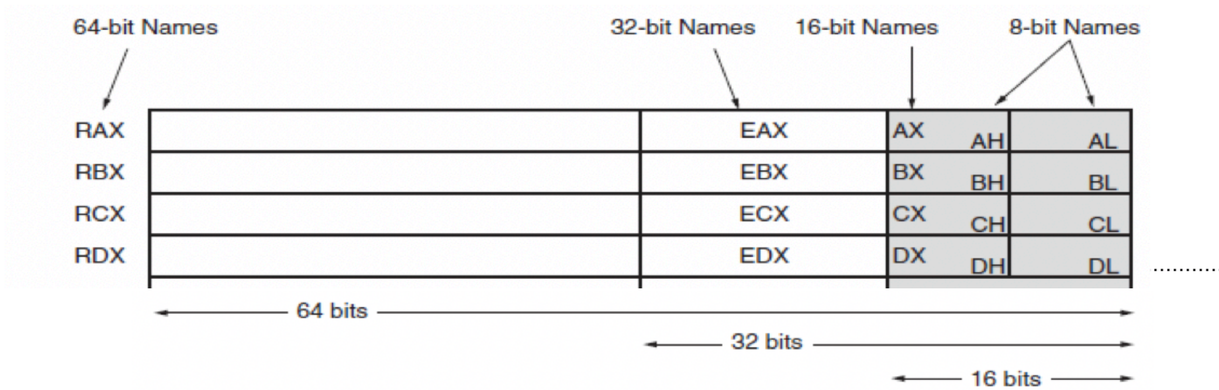
- State provides accesing the registers through **state.regs.register_name** where register_name could be rcx, ecx, cx, ch and cl. Same applies to the other registers.
- Look at the types of registers -- they are bitvectors

```
>>> state.regs.rcx
<BV64 reg_rcx_4_64{UNINITIALIZED}>
>>> state.regs.ecx
<BV32 Reverse(Reverse(reg_rcx_4_64)[63:32])>
>>> state.regs.cx
<BV16 Reverse(Reverse(reg_rcx_4_64)[63:48])>
>>> state.regs.ch
<BV8 reg_rcx_4_64[15:8]>
>>> state.regs.cl
<BV8 reg_rcx_4_64[7:0]>
>>>
```

angr - Registers

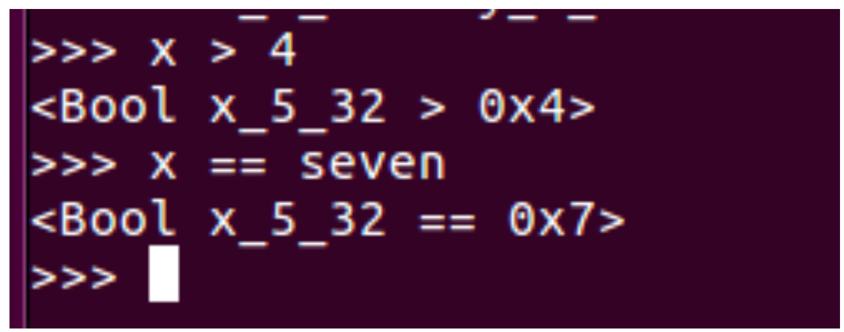
- Look at the length of registers examined below.
- They are all symbolic bitvector because they are not initizlized yet.
- For **cl, ch, cx** and **ecx** they are all part of **rcx**.
- You can compare the length and the location of cl, ch, cx, ecx and rcx in angr with the actual architecture depicted below.

```
>>> state.regs.rcx
<BV64 reg_rcx_4_64{UNINITIALIZED}>
>>> state.regs.ecx
<BV32 Reverse(Reverse(reg_rcx_4_64)[63:32])>
>>> state.regs.cx
<BV16 Reverse(Reverse(reg_rcx_4_64)[63:48])>
>>> state.regs.ch
<BV8 reg_rcx_4_64[15:8]>
>>> state.regs.cl
<BV8 reg_rcx_4_64[7:0]>
>>>
```



angr - Constraints

- In a CFG, a line like **if (x > 10)** creates a branch. Please look at the Symbolic Execution Concepts tutorial.
- Assuming **x** is a symbolic variable, this will create a **<Bool x_5_32 > 4>** when the True branch is taken for the successor state
- For the false branch, negation of a **<Bool x_5_32 > 4>** will be created.



angr - Constraints

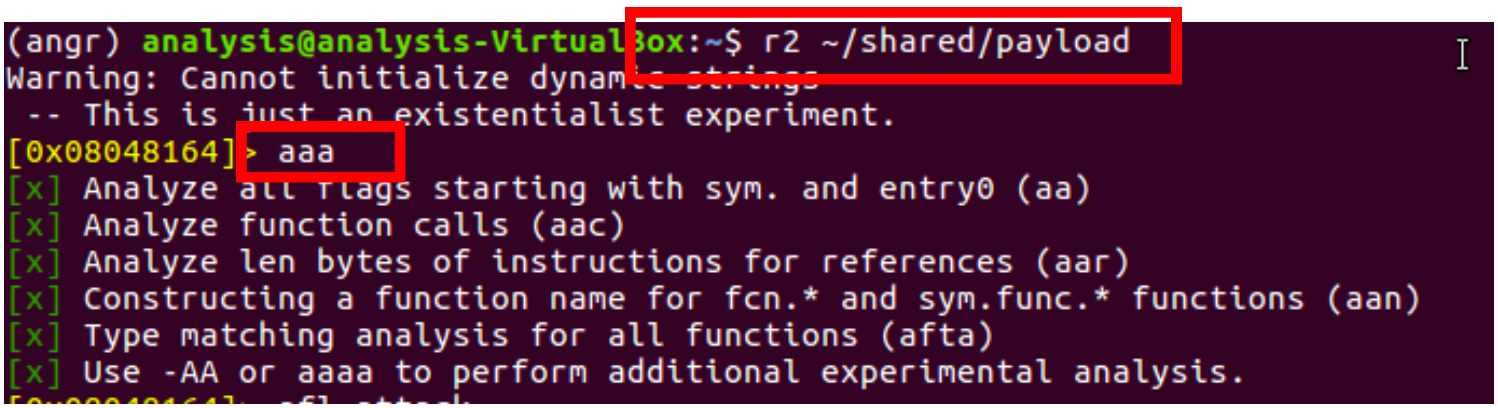
- Adding a constraint to a SimState
  - Cl register equals to 11
  - **state.add_constraints(state.regs.cl == 11)**
  - **state.add_constraints(state.regs.cl == state.solver.BVV(0xb, 8)**
  - Both constraints are same, since state.solver.BVV(0xb, 8) equals to 11
  - You can see their affect is same for SimState in the example below.

```
>>> state.add_constraints(state.regs.cl == state.solver.BVV(0xb, 8))
>>> state.se.constraints
[<Bool reg_rcx_7_8 == 11>]
>>> state.add_constraints(state.regs.cl == 11)
>>> state.se.constraints
[<Bool reg_rcx_7_8 == 11>]
>>> state.add_constraints(state.regs.cl >= 13)
>>> state.se.constraints
[<Bool reg_rcx_7_8 == 11>, <Bool reg_rcx_7_8 >= 13>]
>>> 
```

Radare2 Tutorial

Radare2

- Launch radare2 with **$ r2 ~/shared/payload.exe**
- Then type **aaa** which will analze all (functions + bbs)



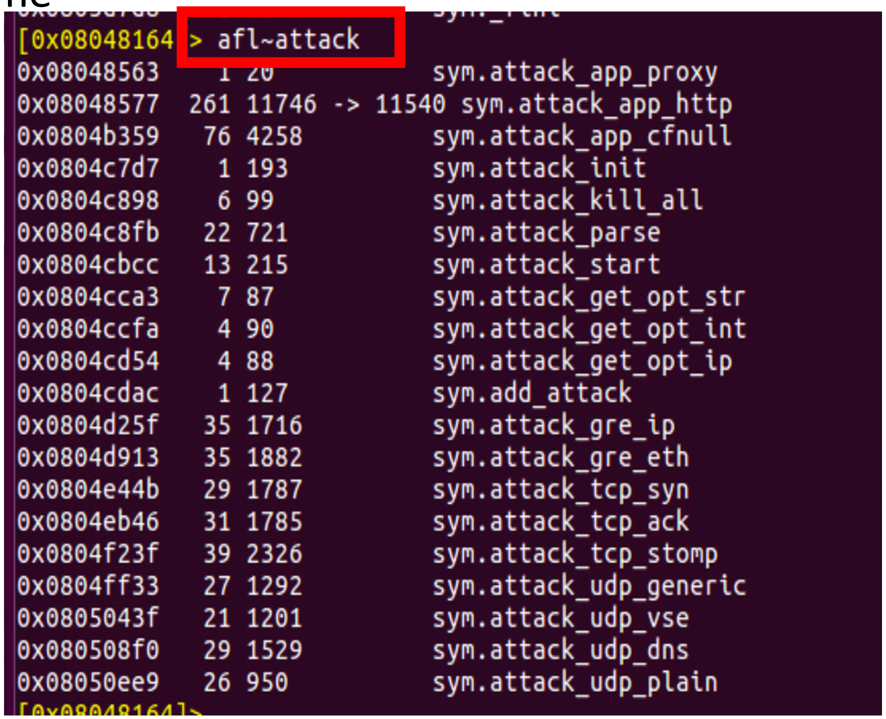
Radare2

- **afl** list all functions

```
[x] Use -AA or aaaa to perform additional experimental analysis.
[0x08048164]> afl
0x08048094    1 28           sym._init
0x080480b0    1 4            sym.__get_pc_thunk_bx
0x080480c0    8 72           sym.__do_global_dtors_aux
0x08048110    6 81           sym.frame_dummy
0x08048164    1 33           entry0
0x08048188    4 60           sym.xputc
0x080481c4    4 42           sym.xputs
0x080481ee   56 853  -> 704  sym.xvprintf
0x08048543    1 32           sym.xprintf
0x08048563    1 20           sym.attack_app_proxy
0x08048577  261 11746 -> 11540 sym.attack_app_http
0x0804b359   76 4258         sym.attack_app_cfnull
0x0804c3fc    4 60           sym.xputc_1
0x0804c438    4 42           sym.xputs_1
0x0804c462   56 853  -> 704  sym.xvprintf_1
0x0804c7b7    1 32           sym.xprintf_1
0x0804c7d7    1 193          sym.attack_init
0x0804c898    6 99           sym.attack_kill_all
0x0804c8fb   22 721          sym.attack_parse
0x0804cbcc   13 215          sym.attack_start
0x0804cca3    7 87           sym.attack_get_opt_str
0x0804ccfa    4 90           sym.attack_get_opt_int
0x0804cd54    4 88           sym.attack_get_opt_ip
0x0804cdac    1 127          sym.add_attack
0x0804ce2b    8 86           sym.free_opts
0x0804ce84    4 60           sym.xputc_2
0x0804cec0    4 42           sym.xputs_2
0x0804ceea   56 853  -> 704  sym.xvprintf_2
0x0804d23f    1 32           sym.xprintf_2
0x0804d25f   35 1716         sym.attack_gre_ip
0x0804d913   35 1882         sym.attack_gre_eth
0x0804e070    4 60           sym.xputc_3
0x0804e0ac    4 42           sym.xputs_3
0x0804e0d6   56 853  -> 704  sym.xvprintf_3
```

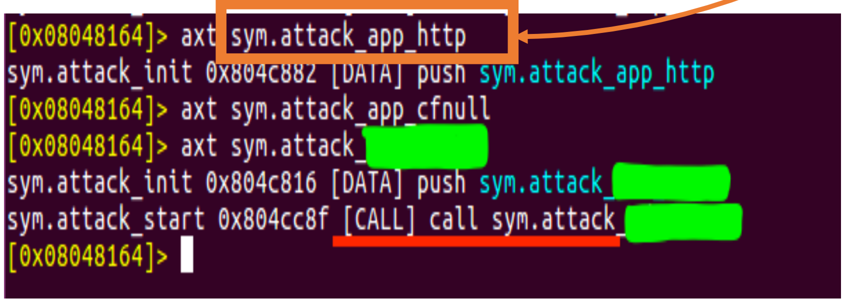
Radare2

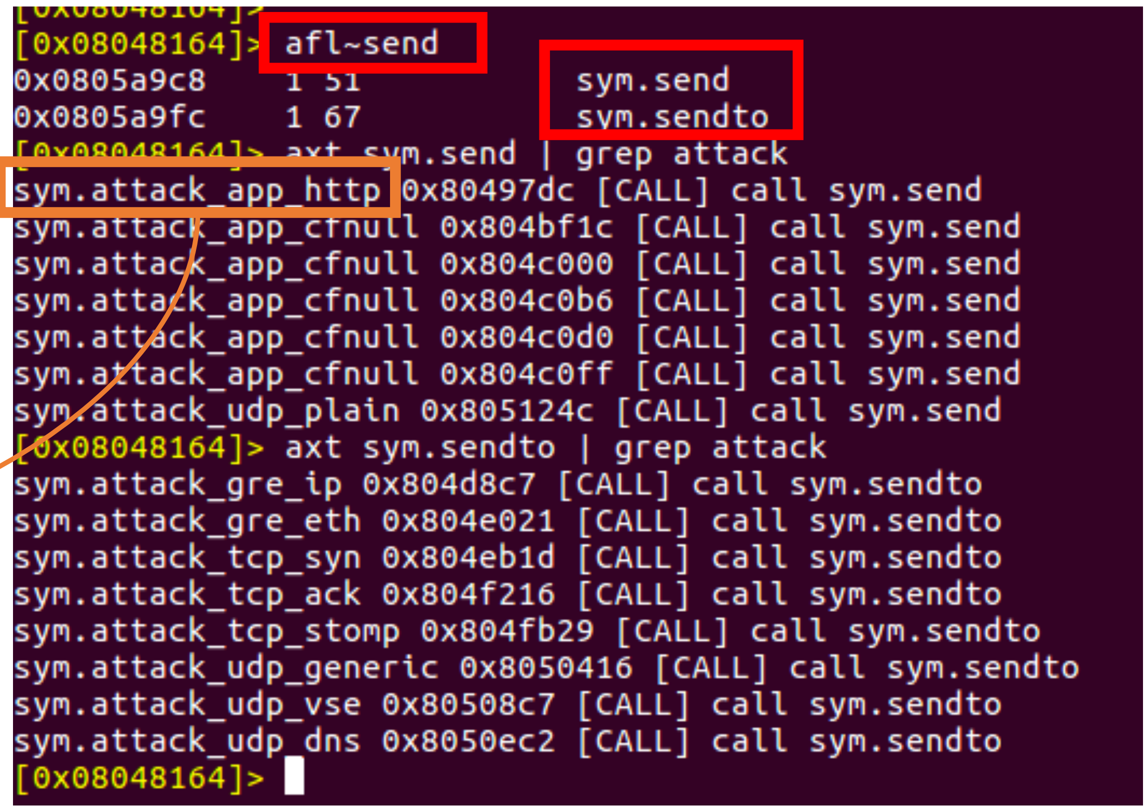
- **afl** lists all the functions which is hard to analyze.
- **afl~name** grep the list of functions with given name
- **afl~attack** will list all the functions having **attack**



Radare2

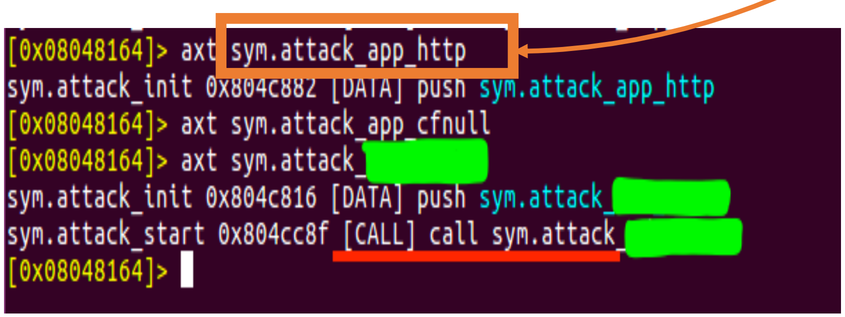
- You can use linux commands while inside the r2 console such as **grep**.
- On the right side, you can see all the functions having the attack vector **(afl~send)**
- Using those api calls, this linux malware performs DDoS attacks based on the commands they receive from C&C server.
- The example on the right side shows how to find all the attack vectors calling sym.send/sym.sendto
- Now, we have to iterate all the attack functions on the right. For example, the example below shows three attack functions, and only one of them is called. Our focus is the call sym.attack_????? functions.

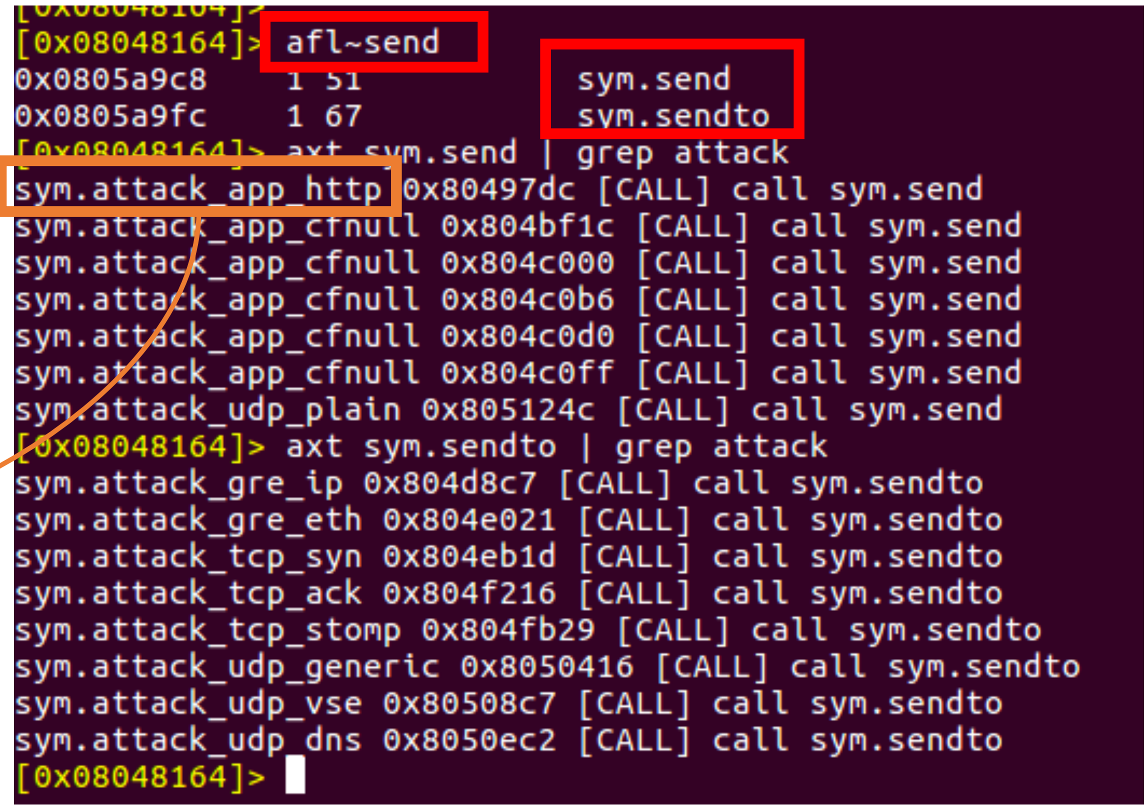




Radare2

- Let's analyze the example below.
- **axt sym.attack_app_http** has only one reference which is a push instruction. This is not the attack function we are interested in.
- **axt sym_attack_app_cfnull** has no reference at all. This is not the attack function we need to explore.
- **axt sym_attack_????** Is one of the functions listed on the right example, and have **call sym.attack_?????** Instruction. That is the function we need to explore more to determine the target address for the symbolic execution.
- **You need to find 2 attack functions.**





Radare2

- After finding the attack function, we can determine the target address.
- First, step into the function using **s sym.attack_????**.
- Second, **pdf | grep sym.send** or **pdf | grep sym.sendto** to determine the instruction address
- Third, **s address_for_call_sym.send(to)** to point to the instruction which is call sym.send or sym.sendto
- Lastly, print 2 instructions starting with the call sym.send/sym.sendto instruction
- The address of the instruction which is the successor of call sym.send(to) is the target address for the symbolic execution.

Radare2

- For more information :
  - https://github.com/radare/radare2
  - https://www.radare.org/get/THC2018.pdf

Other tools

- You don't have to use Radare2
- Here some of the tools you may want to use
  - objdump
  - IDA-Pro (Dissambly tool with GUI) (Free version)
    - https://www.hex-rays.com/products/ida/support/download_freeware.shtml
  - Cutter (GUI for the radare2)
    - https://github.com/radareorg/cutter