Symbolic and Concolic

Software Testing

- Software testing is the process to discover the bug inside the software
- How can we do that?
  - Manual designed testing case
  - Fuzz: run the program with mal-form input and then observe if the program crash
- Note that, vulnerability discovery is also type of software testing without source code

Can we find bug in software?

- Determine if the program has a vulnerability is undicidable
  - Assume we have a Machine M that can detect any vulnerability in the program
  - Halting Problem

```
If M(P) has no bug:
    do_some_bug()
Else:
    do_nothing()
```

Code Coverage Problem

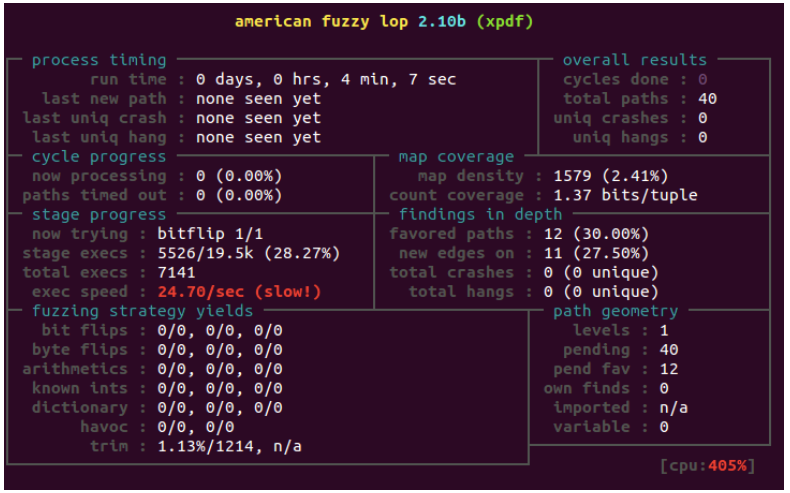
- If we have a execution trace, we can check if the bug appeared in this path
- To testing software complete, we need to traversal all the code inside the program
  - Halting problem
- But we can still do something ☺

Fuzzer

- Automatic generate the input to make the program crash
- Not inspect into program semantic
- Generate input randomly, or some heuristic
  - Coverage-based
- AFL, Peach, BFF

AFL Fuzzer

- American Fuzzy Loop
  - The easy-to-use fuzzer
  - Efficiency
    - low-level compile-time instrumentation
  - Coverage-based Fuzzer
  - Effective Mutation Strategy



At least 4 team in CGC use AFL

AFL Fuzzer

- How AFL do?
  1. Load user-supplied initial test cases into the queue
  2. Take next input file from the queue
  3. Attempt to trim the test case to the smallest size that doesn't alter the measured behavior of the program,
  4. Repeatedly mutate the file using a balanced and well-researched variety of traditional fuzzing strategies
  5. If any of the generated mutations resulted in a new state transition recorded by the instrumentation, add mutated output as a new entry in the queue.
  6. Go to 2.
- Binary fuzzer -> QEMU(emulator) support

What Fuzzer Cannot Do?

- Consider the following code

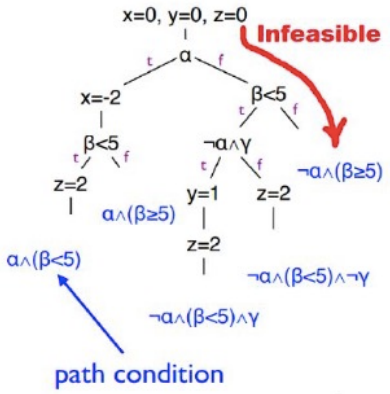
```
int main(int argc, char** argv)
{
    if(atoi(argv[1])==0x1337)
    {
        # Oh~I get a bug
        int a = /100(atoi(argv[1])-0x1337);
    }
}
```

- Without inferring program logic, fuzzer cannot find a input to satisfy branch condition
- Static analysis – the information about argv is unknown, cannot determine if input is possible to be1337.

Symbolic Execution Intro.

- A mechanism to discover the code coverage
  - Translate each instruction/code line into constraints
    - Constraints: a formula define the operation functionality
  - Collect all the constraints
  - Solve when required condition is meet
    - E.g. branch happened

```
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)
```



Common use of SE

Software testing

- C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R.Engler. EXE: Automatically generating inputs of death. 2006
- P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing 2005
- K. Sen. Concolic testing 2007
- …

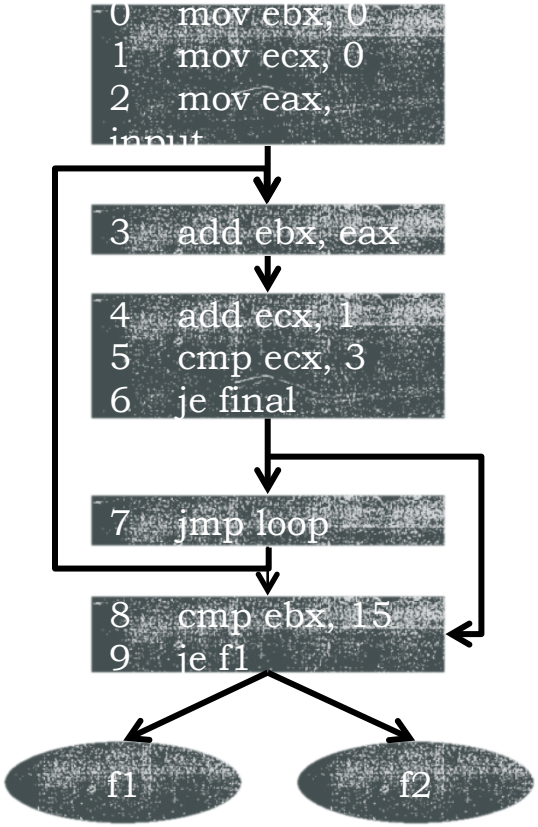
Malware analysis

- D. Brumley, C. Hartwig, M. G. Kang, Z. L. J. Newsome,P. Poosankam, D. Song, and H. Yin. BitScope: Automatically dissecting malicious binaries 2007
- A. Moser, C. Kruegel, and E. Kirda. Exploring multiple execution paths for malware analysis 2001

Available tools

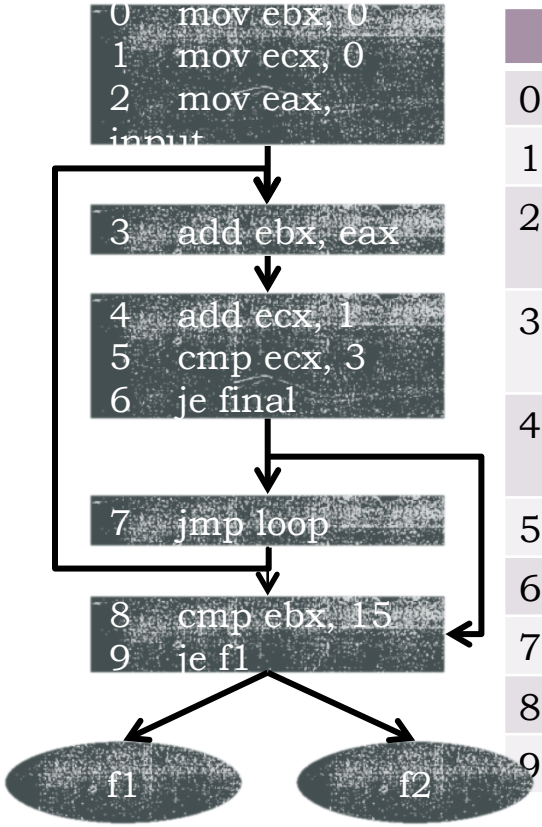
- Stanford's KLEE: http://klee.llvm.org/
- NASA's Java PathFinder: http://javapathfinder.sourceforge.net/
- Microsoft Research's SAFE
- UC Berkeley's CUTE
- EPFL's S2E http://dslab.epfl.ch/proj/s2e

Symbolic Execution



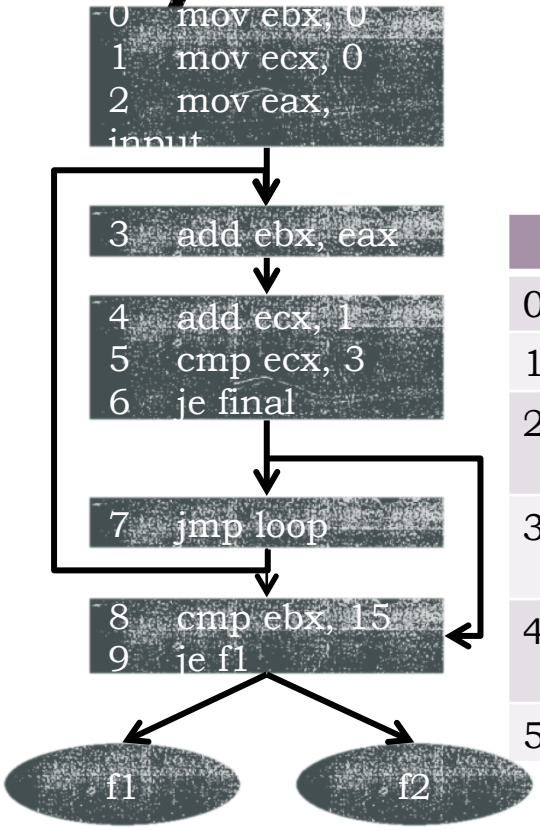
	1st	2nd	3rd
0	ebx = 0		
1	ecx = 0		
2	eax = 3		
3	ebx = 3	ebx = 6	ebx = 9
4	ecx = 1	ecx = 2	ecx = 3
5	NE	NE	E
6	N	N	Y
7	Y	Y	
8			NE
9			N

Symbolic Execution



	1st	2nd	3rd
0	ebx_0 = 0		
1	ecx_0 = 0		
2	eax_0 = sym_0		
3	ebx_1 = ebx_0+eax_0	ebx_2 = ebx_1+eax_0	ebx_3 = ebx_2+eax_0
4	ecx_1 = ecx_0+1	ecx_2 = ecx_1+1	ecx_3 = ecx_2+1
5	NE	NE	E
6	N	N	Y
7	Y	Y	
8			NE
9			N

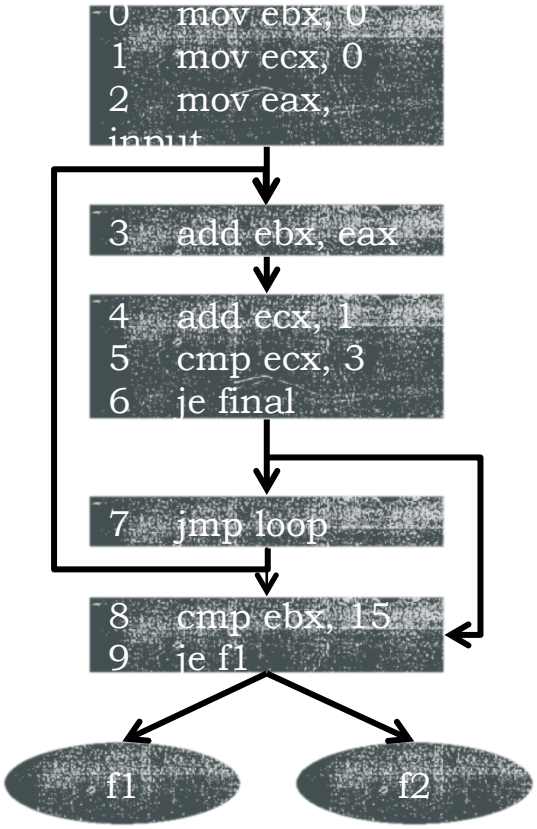
Symbolic Execution



- Can we jump to final block when loop 3 times?

	1st	2nd	3rd
0	ebx_0 = 0		
1	ecx_0 = 0		
2	eax_0 = sym_0		
3	ebx_1 = ebx_0+eax_0	ebx_2 = ebx_1+eax_0	ebx_3 = ebx_2+eax_0
4	ecx_1 = ecx_0+1	ecx_2 = ecx_1+1	ecx_3 = ecx_2+1
5	NE	NE	E

Constraint Solver



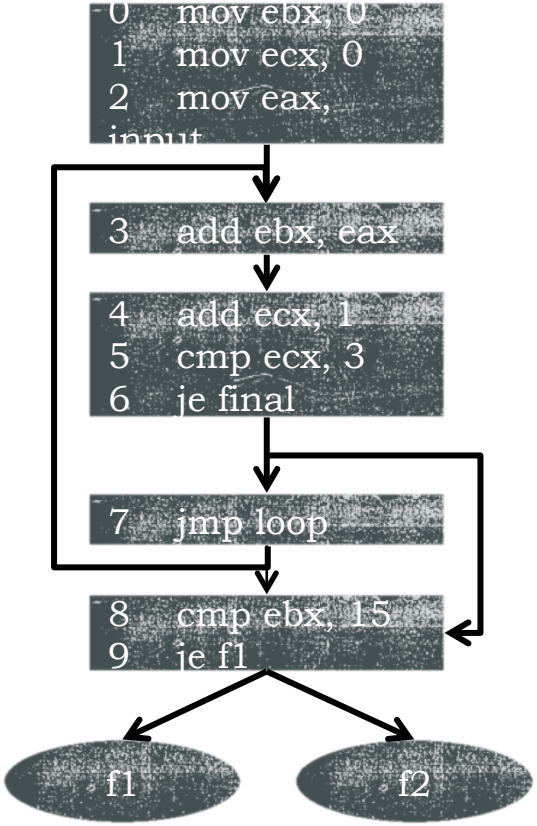
- Can we jump to final block when loop 3 times?

(= ecx_3 3)
(= (+ ecx_2 1) 3)
(= (+ (+ ecx_1 1) 1) 3)
(= (+ (+ (+ ecx_0 1) 1) 1) 3) and (= ecx_0 0)

SMT Solver

SAT! This formula is satisfiable.

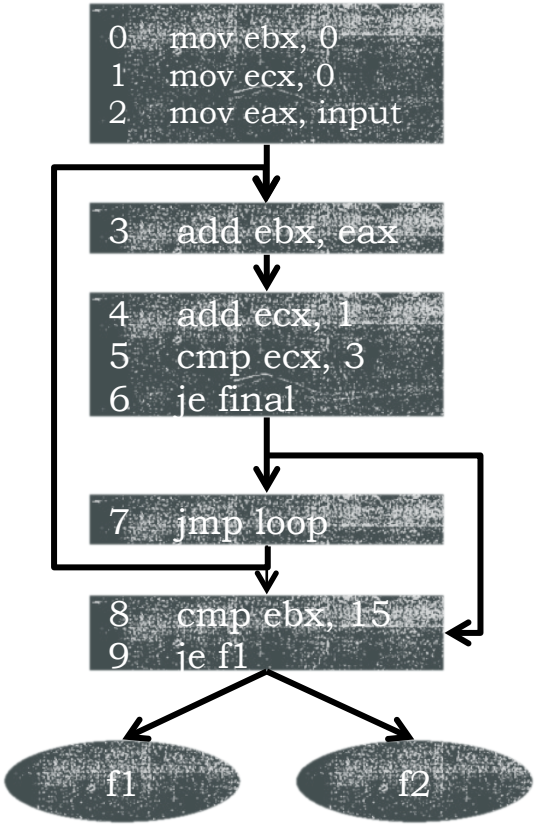
Symbolic Execution



- Can we enter f1?

	1st	2nd	3rd
0	ebx_0 = 0		
1	ecx_0 = 0		
2	eax_0 = sym_0		
3	ebx_1 = ebx_0+eax_0	ebx_2 = ebx_1+eax_0	ebx_3 = ebx_2+eax_0
4	ecx_1 = ecx_0+1	ecx_2 = ecx_1+1	ecx_3 = ecx_2+1
5	NE	NE	E
6	N	N	Y
7	Y	Y	
8			NE
9			N

Taint Track



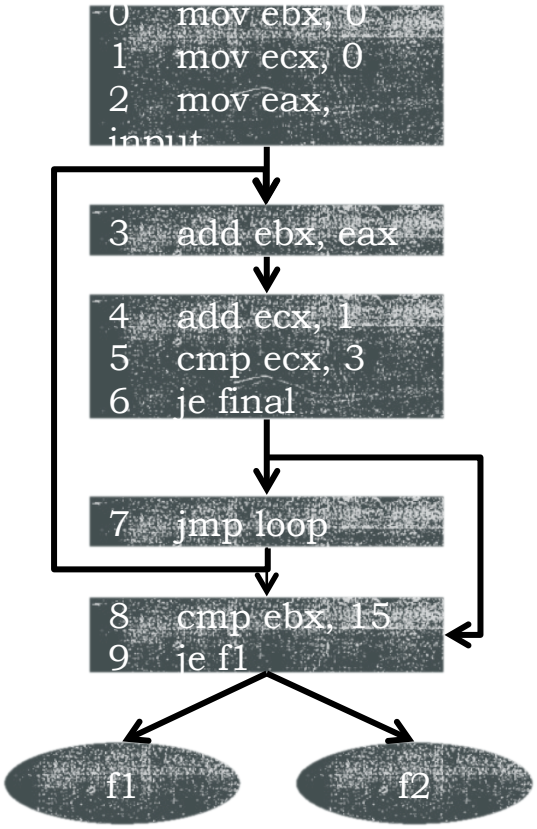
- Tracking related instructions only?

	1st	2nd	3rd
0	ebx = 0		
1	ecx = 0		
2	eax = 3		
3	ebx = 3	ebx = 6	ebx = 9
4	ecx = 1	ecx = 2	ecx = 3
5	NE	NE	E
6	N	N	Y
7	Y	Y	
8			NE
9			N

Concolic Execution

- Number of possible path increasing exponentially
  - In symbolic execution, every memory location is symbolize
  - Too many symbole to solve
- Concolic Execution
  - Only make the interesting memory symbolize
  - Concrete value

Concolic Execution

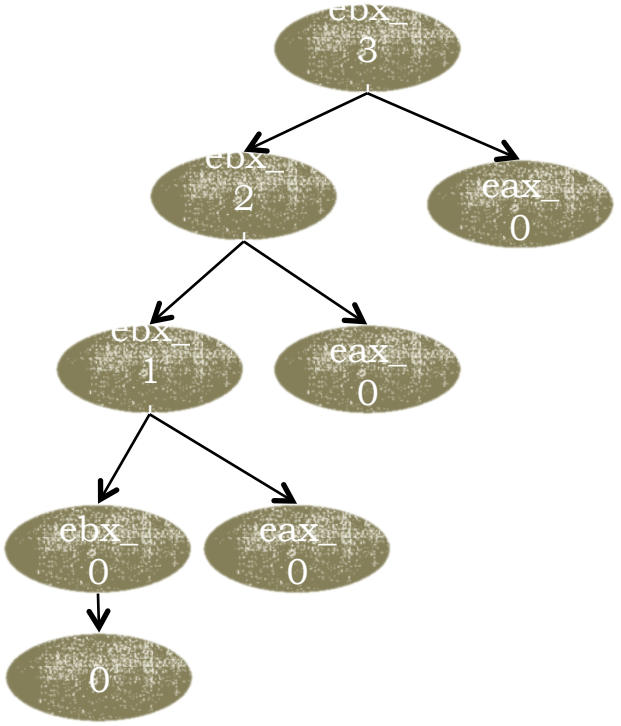


	1st	2nd	3rd
0	ebx_0 = 0		
1	ecx_0 = 0		
2	eax_0 = sym_0		
3	ebx_1 = ebx_0+eax_0	ebx_2 = ebx_1+eax_0	ebx_3 = ebx_2+eax_0
4	ecx_1 = 1	ecx_2 = 2	ecx_3 = 3
5	NE	NE	E
6	N	N	Y
7	Y	Y	
8			NE
9			N

Concolic Execution

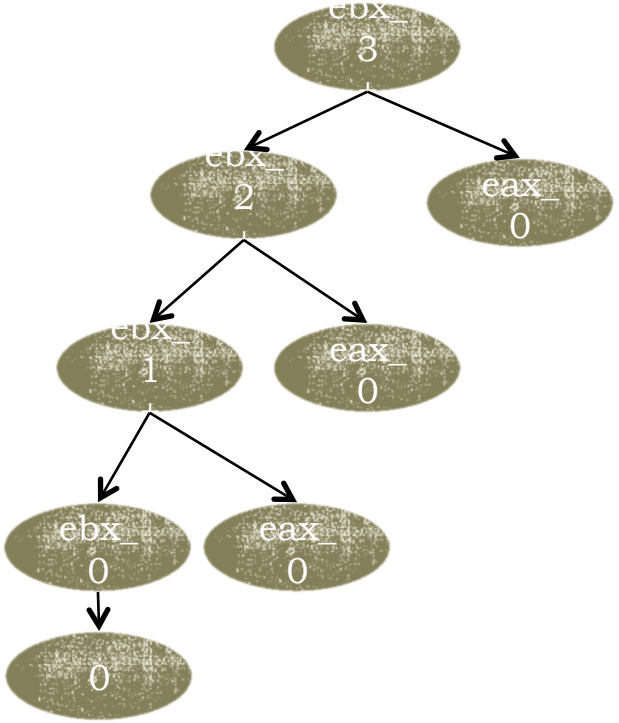
- Which input make us arrive f1?

	1st	2nd	3rd
0	ebx_0 = 0		
1	ecx_0 = 0		
2	eax_0 = sym_0		
3	ebx_1 = ebx_0+eax_0	ebx_2 = ebx_1+eax_0	ebx_3 = ebx_2+eax_0
4	ecx_1 = 1	ecx_2 = 2	ecx_3 = 3
5	NE	NE	E
6	N	N	Y
7	Y	Y	
8			NE
9			N



Concolic Execution

- Which input make us arrive f1?

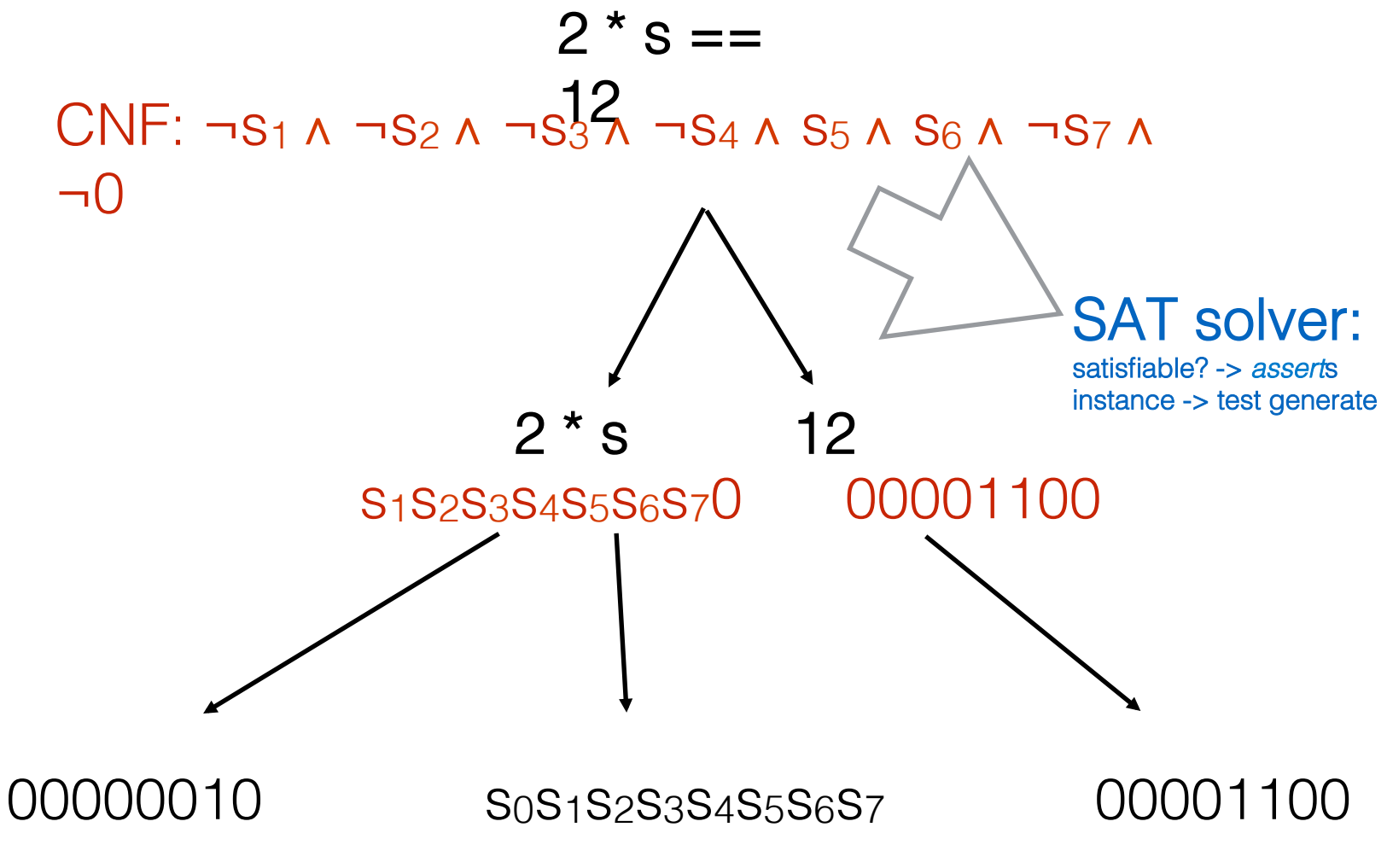


Final
(= 15 (+ in (+ in (+ in 0))))

SMT Solver

SAT! This formula is satisfiable when in = 5.

Constraint solver



Applications of Symbolic Execution

- General goal: identifying semantics of programs
  Basic applications:
  - Detecting infeasible paths
  - Generating test inputs
  - Finding bugs and vulnerabilities
  - Proving two code segments are equivalent (Code Hunt)
- Advanced applications:
  - Generating program invariants
  - Debugging
  - Repair programs

Path Explosion

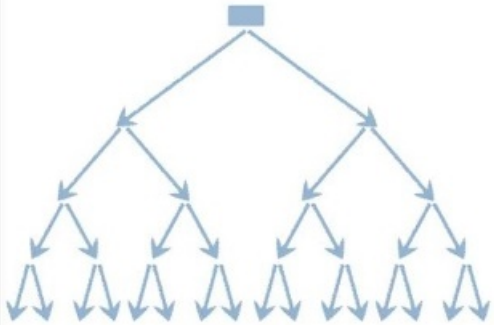
- Symbolic execution is the heavy-weight analysis
- Once the program contains too many path, symbolic execution may fails

```
int branch(int a){
    if(a >0){...}
    else{...}

    if(a == 1337){...}
    else{...}

    if(a % 10 == 0){...}
    else{...}
}
```

4 conditional nodes



16 (2^4) execution paths

Path Explosion Due to Loop

- Loop can be treated as combination of conditional branch

```
int loop()
{
    for(int i=0;i<100;i++)
    {
        do_something();
    }

}
```

About 2^{100} Paths

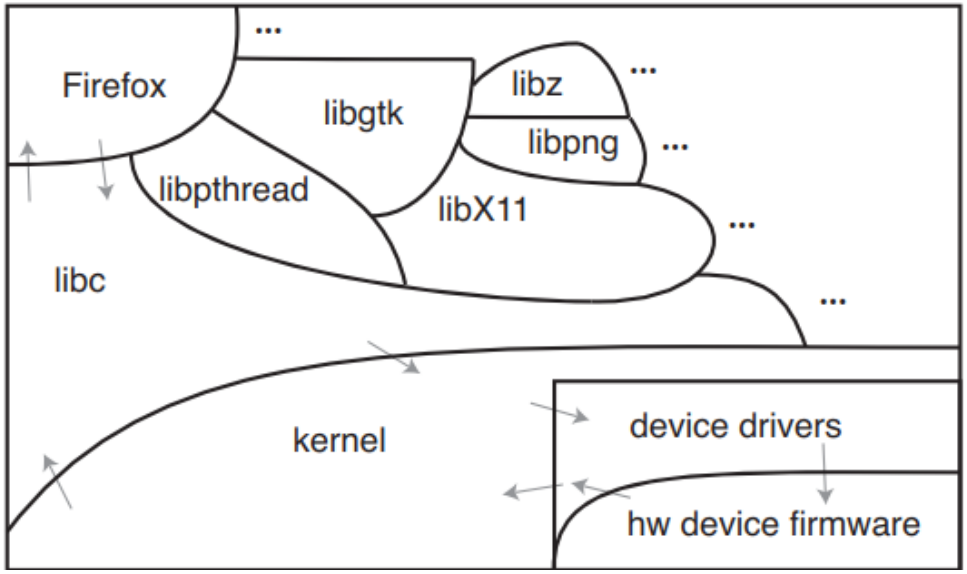
Indirect Access

- It's not easy for symbolic execution to deal with indirect access
  - Miss symbol
  - Too many symbols

```
int indirect(int addr1, int addr2)
{
    // assume char arr[1000] is global var
    arr[addr1] = addr;
    return arr[addr2];
}
```

Execution Model

- When and How to symbolize the variable?
- Application/Library/ Kernel
- Trade off between performance, soundness and completeness



Comparison of Model

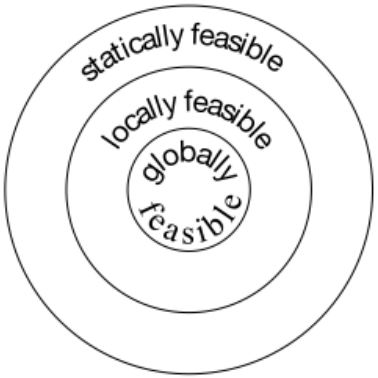
Model	Consistency	Completeness	Use Case
SC-CE	consistent	incomplete	Single-path profiling/testing of units that have a limited number of paths
SC-UE	consistent	incomplete	Analysis of units that generate hard-to-solve constraints (e.g., cryptographic code)
SC-SE	consistent	complete	Sound and complete verification without false positives or negatives; testing of tightly coupled systems with fuzzy unit boundaries.
LC	locally consistent	incomplete	Testing/profiling while avoiding false positives from the unit's perspective
RC-OC	inconsistent	complete	Reverse engineering: extract consistent path segments
RC-CC	inconsistent	complete	Dynamic disassembly of a potentially obfuscated binary

Difference Between Models

- How they switch between symbolic and concrete
- Completeness
  - Catch every path through unit globally feasible or not
  - Can affect analysis completeness
- Consistency
  - Every path through unit is globally feasible or not
  - Can affect analysis soundness

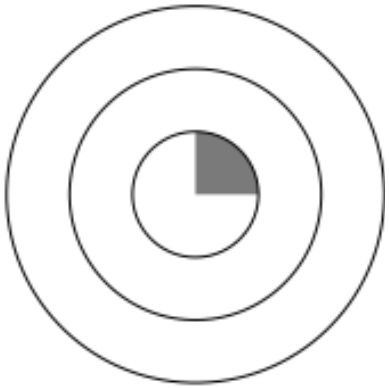
Consistency

- Statically feasible
  - If a path is Admitted by CFG in the whole system (unit & environment)
- Locally feasible
  - Additionally consistent with data restrictions in unit
- Globally feasible
  - Additionally consistent with data restrictions in environment



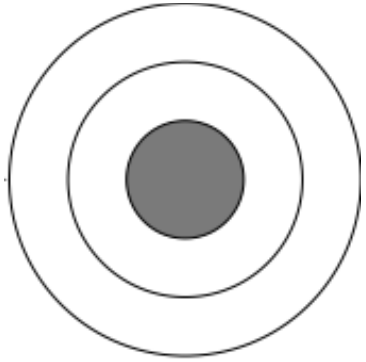
Strictly Consistent Concrete Execution

- Fuzzing
- Globally consistent but not complete
- Valgrind, Eraser use it



Strictly Consistent System-level Execution

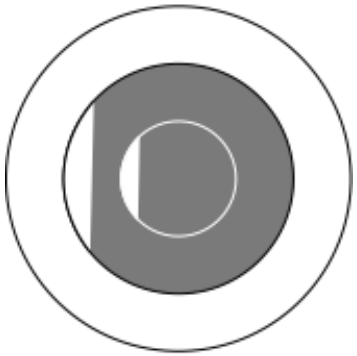
- Full symbolic execution
- Track all constraints in whole system
- Cost time and resource (Suffer from path explosion)
- Consistent & Complete
- KLEE use it



Local Consistency

- No track of constraint outside unit
- Guess the data get from outside
  - count_r = write(fd, buf, count)
  - count_r is symbolize with constraint {-1, < count}
- Fast and has advantage on soundness and completeness
  - Environment maybe inconsistent

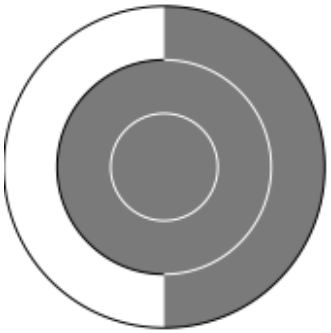
```
buf =“deadbeef”
count_r = write(fd, buf, count)
read(buf2, count)
If( count_r == 0 & buf2 == “deadbeef” )
{
    …….. // Never happened but find by LC
model
}
```



Overapproximate Consistency (RC-OC)

- No track of constraint outside unit
- Make any interaction with environment as unconstraint symbolic variable
  - count_r = write(fd, buf, count)
  - count_r is unconstraint varable

```
buf =“deadbeef”
count_r = write(fd, buf, count)
If( count_r > count)
{
    …….. // Never happened but find by LC
model
}
```



RC-OC

Concolic-assisted Fuzzer

- Driller
  - Switch between fuzzer and symbolic execution
  - Driller: Augmenting Fuzzing Through Selective Symbolic Execution
  - Network and Distributed System Security Symposium 2016

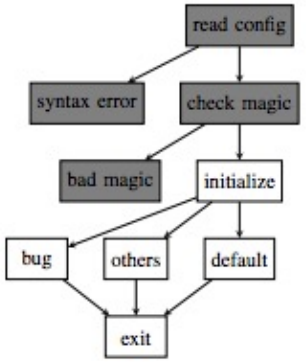


Fig. 1. The nodes initially found by the fuzzer.

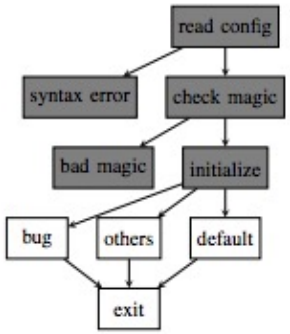


Fig. 2. The nodes found by the first invocation of concolic execution.

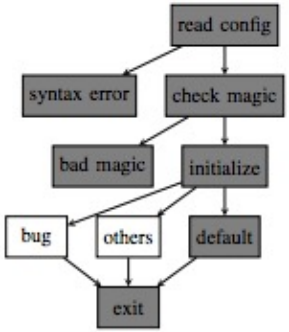


Fig. 3. The nodes found by the fuzzer, supplemented with the result of the first Driller run.

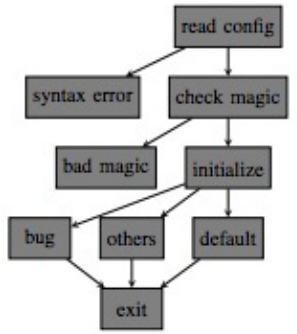


Fig. 4. The nodes found by the second invocation of concolic execution.

Q&A