Fuzzing
and
Symbolic Execution

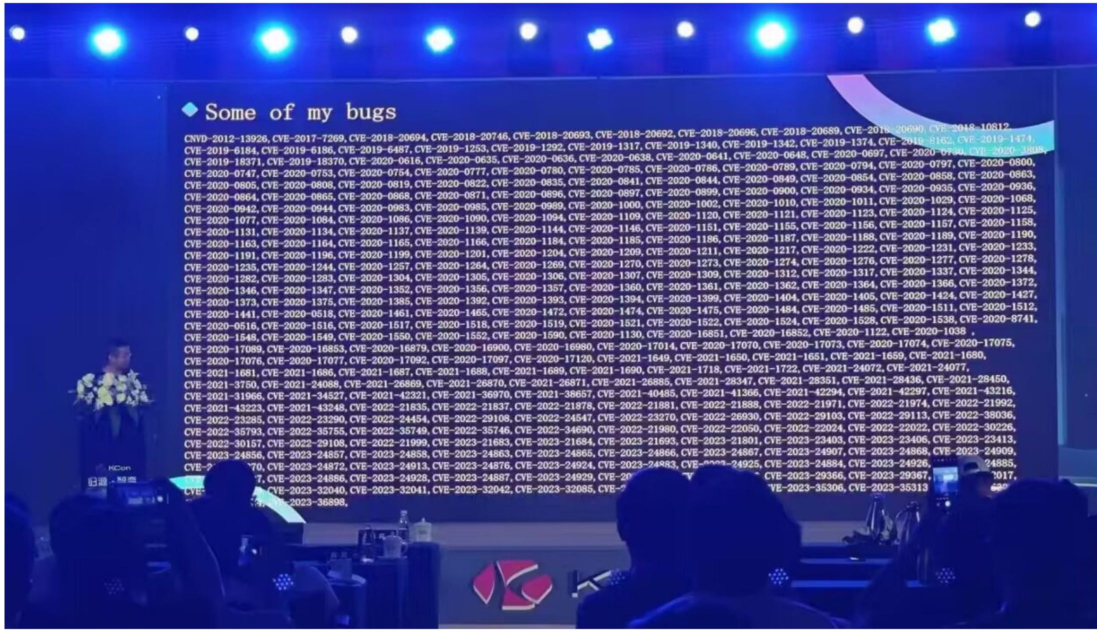
SUSTech CS 315 Computer Security 2023

Outline

- Program Analysis
- Exploiting Real World Programs
- Fuzz Testing
- Symbolic Execution
- Mixed Fuzz and Symbolic Execution (Concolic Execution)
- Summary

Program Analysis

- Bug hunters reported 1000+ bugs
    - They want to find bug efficiently in countless real-world applications
- Large companies like google
    - They need to ensure the quality of the large amounts of software they release
    - They also need to ensure that the open-source components they use are safe and stable



◆ Some of my bugs

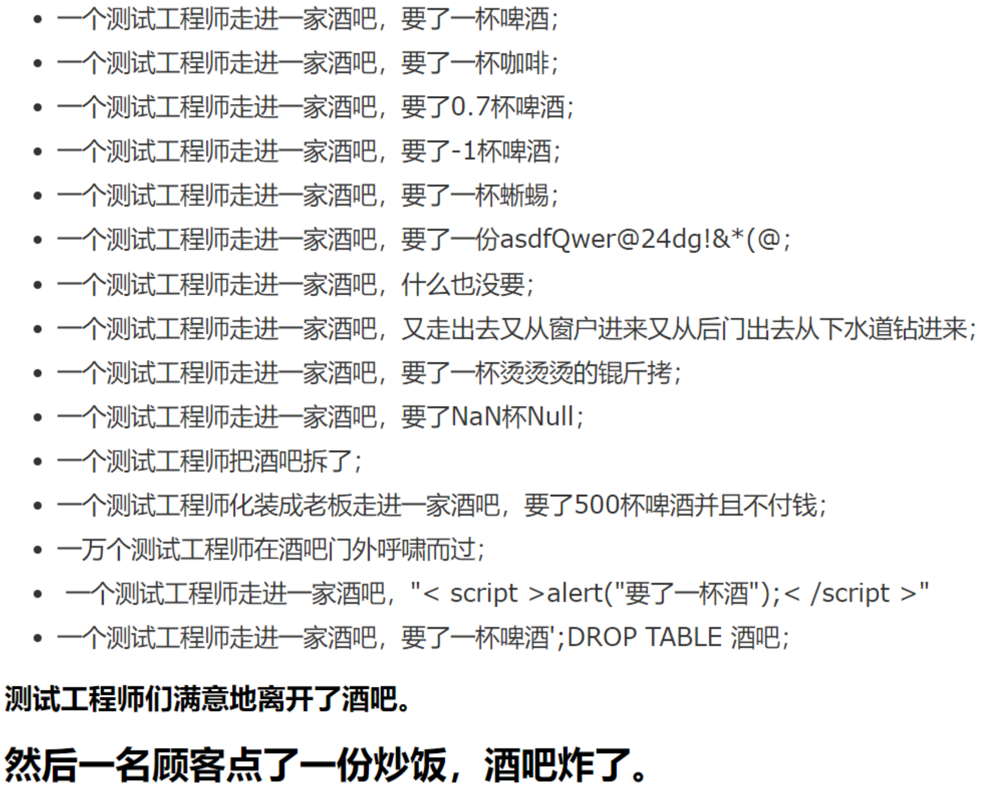
CNVD-2012-13926, CVE-2017-7269, CVE-2018-20694, CVE-2018-20746, CVE-2018-20693, CVE-2018-20692, CVE-2018-20696, CVE-2018-20689, CVE-2018-20690, CVE-2018-10812,
CVE-2019-6184, CVE-2019-6186, CVE-2019-6487, CVE-2019-1253, CVE-2019-1292, CVE-2019-1317, CVE-2019-1340, CVE-2019-1342, CVE-2019-1374, CVE-2019-8162, CVE-2019-1474,
CVE-2019-18371, CVE-2019-18370, CVE-2020-0616, CVE-2020-0635, CVE-2020-0636, CVE-2020-0638, CVE-2020-0641, CVE-2020-0648, CVE-2020-0697, CVE-2020-0730, CVE-2020-3898,
CVE-2020-0747, CVE-2020-0753, CVE-2020-0754, CVE-2020-0777, CVE-2020-0780, CVE-2020-0785, CVE-2020-0786, CVE-2020-0789, CVE-2020-0794, CVE-2020-0797, CVE-2020-0800,
CVE-2020-0805, CVE-2020-0808, CVE-2020-0819, CVE-2020-0822, CVE-2020-0835, CVE-2020-0841, CVE-2020-0844, CVE-2020-0849, CVE-2020-0854, CVE-2020-0858, CVE-2020-0863,
CVE-2020-0864, CVE-2020-0865, CVE-2020-0868, CVE-2020-0871, CVE-2020-0896, CVE-2020-0897, CVE-2020-0899, CVE-2020-0900, CVE-2020-0934, CVE-2020-0935, CVE-2020-0936,
CVE-2020-0942, CVE-2020-0944, CVE-2020-0983, CVE-2020-0985, CVE-2020-0989, CVE-2020-1000, CVE-2020-1002, CVE-2020-1010, CVE-2020-1011, CVE-2020-1029, CVE-2020-1068,
CVE-2020-1077, CVE-2020-1084, CVE-2020-1086, CVE-2020-1090, CVE-2020-1094, CVE-2020-1109, CVE-2020-1120, CVE-2020-1121, CVE-2020-1123, CVE-2020-1124, CVE-2020-1125,
CVE-2020-1131, CVE-2020-1134, CVE-2020-1137, CVE-2020-1139, CVE-2020-1144, CVE-2020-1146, CVE-2020-1151, CVE-2020-1155, CVE-2020-1156, CVE-2020-1157, CVE-2020-1158,
CVE-2020-1163, CVE-2020-1164, CVE-2020-1165, CVE-2020-1166, CVE-2020-1184, CVE-2020-1185, CVE-2020-1186, CVE-2020-1187, CVE-2020-1188, CVE-2020-1189, CVE-2020-1190,
CVE-2020-1191, CVE-2020-1196, CVE-2020-1199, CVE-2020-1201, CVE-2020-1204, CVE-2020-1209, CVE-2020-1211, CVE-2020-1217, CVE-2020-1222, CVE-2020-1231, CVE-2020-1233,
CVE-2020-1235, CVE-2020-1244, CVE-2020-1257, CVE-2020-1264, CVE-2020-1269, CVE-2020-1270, CVE-2020-1273, CVE-2020-1274, CVE-2020-1276, CVE-2020-1277, CVE-2020-1278,
CVE-2020-1282, CVE-2020-1283, CVE-2020-1304, CVE-2020-1305, CVE-2020-1306, CVE-2020-1307, CVE-2020-1309, CVE-2020-1312, CVE-2020-1317, CVE-2020-1337, CVE-2020-1344,
CVE-2020-1346, CVE-2020-1347, CVE-2020-1352, CVE-2020-1356, CVE-2020-1357, CVE-2020-1360, CVE-2020-1361, CVE-2020-1362, CVE-2020-1364, CVE-2020-1366, CVE-2020-1372,
CVE-2020-1373, CVE-2020-1375, CVE-2020-1385, CVE-2020-1392, CVE-2020-1393, CVE-2020-1394, CVE-2020-1399, CVE-2020-1404, CVE-2020-1405, CVE-2020-1424, CVE-2020-1427,
CVE-2020-1441, CVE-2020-0518, CVE-2020-1461, CVE-2020-1465, CVE-2020-1472, CVE-2020-1474, CVE-2020-1475, CVE-2020-1484, CVE-2020-1485, CVE-2020-1511, CVE-2020-1512,
CVE-2020-0516, CVE-2020-1516, CVE-2020-1517, CVE-2020-1518, CVE-2020-1519, CVE-2020-1521, CVE-2020-1522, CVE-2020-1524, CVE-2020-1528, CVE-2020-1538, CVE-2020-8741,
CVE-2020-1548, CVE-2020-1549, CVE-2020-1550, CVE-2020-1552, CVE-2020-1590, CVE-2020-1130, CVE-2020-16851, CVE-2020-16852, CVE-2020-1122, CVE-2020-1038 ,
CVE-2020-17089, CVE-2020-16853, CVE-2020-16879, CVE-2020-16900, CVE-2020-16980, CVE-2020-17014, CVE-2020-17070, CVE-2020-17073, CVE-2020-17074, CVE-2020-17075,
CVE-2020-17076, CVE-2020-17077, CVE-2020-17092, CVE-2020-17097, CVE-2020-17120, CVE-2021-1649, CVE-2021-1650, CVE-2021-1651, CVE-2021-1659, CVE-2021-1680,
CVE-2021-1681, CVE-2021-1686, CVE-2021-1687, CVE-2021-1688, CVE-2021-1689, CVE-2021-1690, CVE-2021-1718, CVE-2021-1722, CVE-2021-24072, CVE-2021-24077,
CVE-2021-3750, CVE-2021-24088, CVE-2021-26869, CVE-2021-26870, CVE-2021-26871, CVE-2021-26885, CVE-2021-28347, CVE-2021-28351, CVE-2021-28436, CVE-2021-28450,
CVE-2021-31966, CVE-2021-34527, CVE-2021-42321, CVE-2021-36970, CVE-2021-38657, CVE-2021-40485, CVE-2021-41366, CVE-2021-42294, CVE-2021-42297, CVE-2021-43216,
CVE-2021-43223, CVE-2021-43248, CVE-2022-21835, CVE-2022-21837, CVE-2022-21878, CVE-2022-21881, CVE-2022-21888, CVE-2022-21971, CVE-2022-21974, CVE-2022-21992,
CVE-2022-23285, CVE-2022-23290, CVE-2022-24454, CVE-2022-29108, CVE-2022-24547, CVE-2022-23270, CVE-2022-26930, CVE-2022-29103, CVE-2022-29113, CVE-2022-38036,
CVE-2022-35793, CVE-2022-35755, CVE-2022-35749, CVE-2022-35746, CVE-2022-34690, CVE-2022-21980, CVE-2022-22050, CVE-2022-22024, CVE-2022-22022, CVE-2022-30226,
CVE-2022-30157, CVE-2022-29108, CVE-2022-21999, CVE-2023-21683, CVE-2023-21684, CVE-2023-21693, CVE-2023-21801, CVE-2023-23403, CVE-2023-23406, CVE-2023-23413,
CVE-2023-24856, CVE-2023-24857, CVE-2023-24858, CVE-2023-24863, CVE-2023-24865, CVE-2023-24866, CVE-2023-24867, CVE-2023-24907, CVE-2023-24868, CVE-2023-24909,
CVE-2[illegible]70, CVE-2023-24872, CVE-2023-24913, CVE-2023-24876, CVE-2023-24924, CVE-2023-24883, CVE-[illegible]-24925, CVE-2023-24884, CVE-2023-24926, [illegible]24885,
[illegible]7, CVE-2023-24886, CVE-2023-24928, CVE-2023-24887, CVE-2023-24929, CVE-20[illegible] CVE-2023-29366, CVE-2023-29367, [illegible]017,
CVE-[illegible] CVE-2023-32040, CVE-2023-32041, CVE-2023-32042, CVE-2023-32085, CVE-[illegible] VE-2023-35306, CVE-2023-35313 [illegible]
[illegible]-2023-36898,

KCON 2023

Exploiting Real World Programs

- Real program can be complicated
  - linux kernel has 66000+ files and 270000000+ lines of code
  - what about your Java class project? 1000~5000 lines?
- How to find bugs and vulnerabilities in large and complicated programs?

当你帮酒吧老板写了一个程序:



- 一个测试工程师走进一家酒吧，要了一杯啤酒;
- 一个测试工程师走进一家酒吧，要了一杯咖啡;
- 一个测试工程师走进一家酒吧，要了0.7杯啤酒;
- 一个测试工程师走进一家酒吧，要了-1杯啤酒;
- 一个测试工程师走进一家酒吧，要了一杯蜥蜴;
- 一个测试工程师走进一家酒吧，要了一份asdfQwer@24dg!&*(@;
- 一个测试工程师走进一家酒吧，什么也没要;
- 一个测试工程师走进一家酒吧，又走出去又从窗户进来又从后门出去从下水道钻进来;
- 一个测试工程师走进一家酒吧，要了一杯烫烫烫的锟斤拷;
- 一个测试工程师走进一家酒吧，要了NaN杯Null;
- 一个测试工程师把酒吧拆了;
- 一个测试工程师化装成老板走进一家酒吧，要了500杯啤酒并且不付钱;
- 一万个测试工程师在酒吧门外呼啸而过;
- 一个测试工程师走进一家酒吧，"< script >alert("要了一杯酒");< /script >"
- 一个测试工程师走进一家酒吧，要了一杯啤酒';DROP TABLE 酒吧;

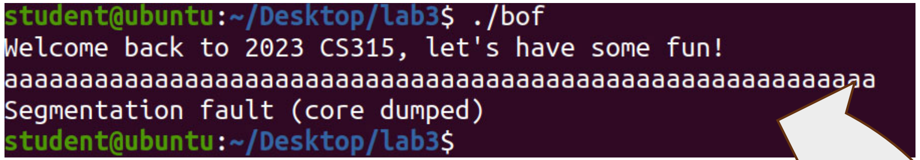
测试工程师们满意地离开了酒吧。

然后一名顾客点了一份炒饭，酒吧炸了。

Manually constructing test cases is not enough

Fuzz Testing

- Recall vulnerable program in last lab
  - Easily crashed by some long input

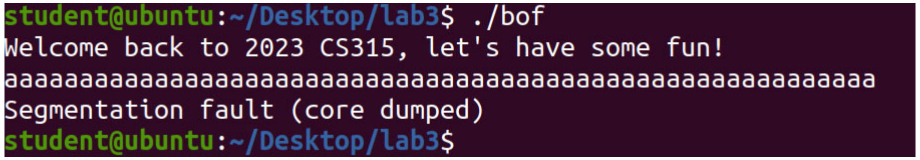


```
student@ubuntu:~/Desktop/lab3$ ./bof
Welcome back to 2023 CS315, let's have some fun!
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
Segmentation fault (core dumped)
student@ubuntu:~/Desktop/lab3$
```

- How can we find bug in thousands of real world programs?
  - Hire some monkeys
  - “randomly” generate some inputs

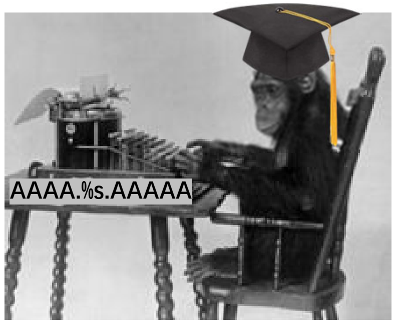
Fuzz Testing

- Recall vulnerable program in last lab
  - Easily crashed by some long input



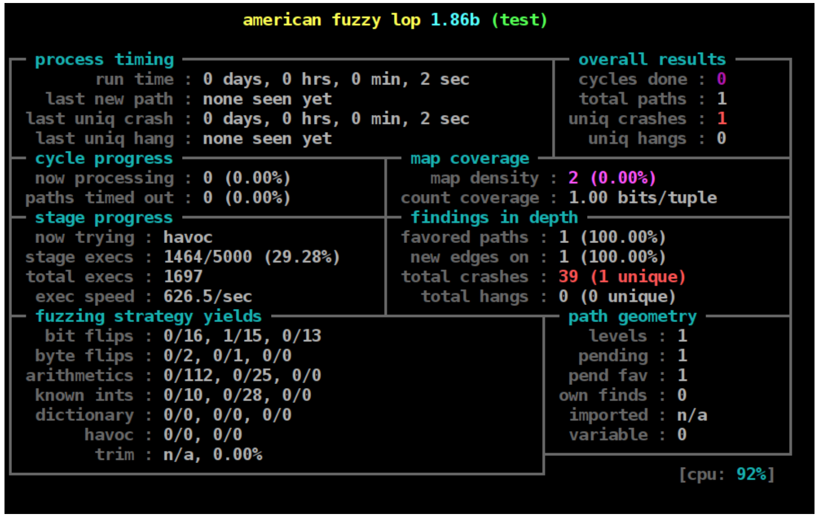
```
student@ubuntu:~/Desktop/lab3$ ./bof
Welcome back to 2023 CS315, let's have some fun!
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
Segmentation fault (core dumped)
student@ubuntu:~/Desktop/lab3$
```

- How can we find bug in thousands of real-world programs?
  - Hire some monkeys
  - “randomly” generate some inputs
- Fuzzer: hire some clever monkeys
  - automatically generate and mutate inputs



Fuzz Testing

- AFL & AFL++ (American fuzzy lop)
  - A modern fuzzing tool
  - employs genetic algorithms to efficiently increase code coverage
  - https://www.usenix.org/system/files/woot20-paper-fioraldi.pdf
  - Best used with address sanitization (ASAN)
    - Flag *all* invalid memory accesses
    - Shadow memory tracking which memory areas are valid
    - Finds out of bounds access and use after free bugs
    - AFL+ASAN combination is gold standard of fuzzing
    - we will try AFL+ASAN in our lab



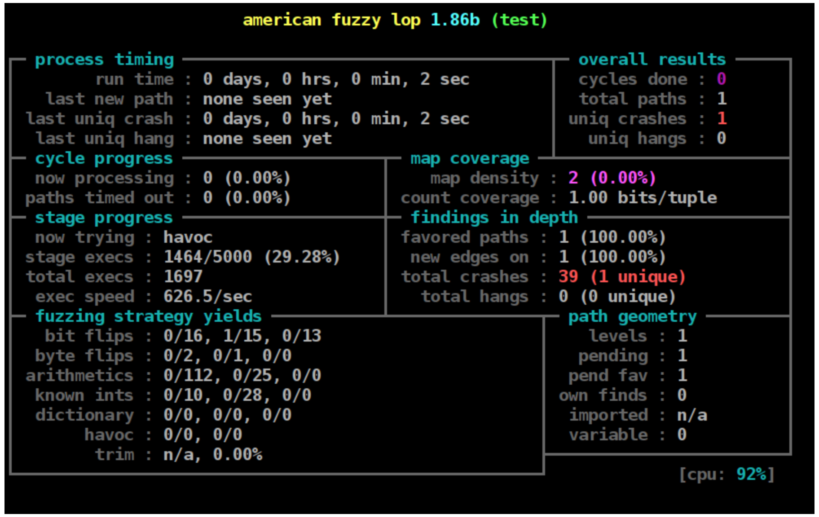
```
                    american fuzzy lop 1.86b (test)

┌─ process timing ─────────────────────────────────────┬─ overall results ─────┐
│        run time : 0 days, 0 hrs, 0 min, 2 sec        │  cycles done : 0      │
│   last new path : none seen yet                      │  total paths : 1      │
│ last uniq crash : 0 days, 0 hrs, 0 min, 2 sec        │ uniq crashes : 1      │
│  last uniq hang : none seen yet                      │   uniq hangs : 0      │
├─ cycle progress ────────────────────┬─ map coverage ─┴───────────────────────┤
│  now processing : 0 (0.00%)         │    map density : 2 (0.00%)             │
│ paths timed out : 0 (0.00%)         │ count coverage : 1.00 bits/tuple       │
├─ stage progress ────────────────────┼─ findings in depth ────────────────────┤
│  now trying : havoc                 │ favored paths : 1 (100.00%)            │
│ stage execs : 1464/5000 (29.28%)    │  new edges on : 1 (100.00%)            │
│ total execs : 1697                  │ total crashes : 39 (1 unique)          │
│  exec speed : 626.5/sec             │   total hangs : 0 (0 unique)           │
├─ fuzzing strategy yields ───────────┴───────────────┬─ path geometry ────────┤
│   bit flips : 0/16, 1/15, 0/13                      │    levels : 1          │
│  byte flips : 0/2, 0/1, 0/0                         │   pending : 1          │
│ arithmetics : 0/112, 0/25, 0/0                      │  pend fav : 1          │
│  known ints : 0/10, 0/28, 0/0                       │ own finds : 0          │
│  dictionary : 0/0, 0/0, 0/0                         │  imported : n/a        │
│       havoc : 0/0, 0/0                              │  variable : 0          │
│        trim : n/a, 0.00%                            ├────────────────────────┘
└─────────────────────────────────────────────────────┘             [cpu: 92%]
```

American fuzzy lop's afl-fuzz running on a test program

Fuzz Testing

- AFL & AFL++ (American fuzzy lop)
  - A modern fuzzing tool
  - employs genetic algorithms to efficiently increase code coverage
  - https://www.usenix.org/system/files/woot20-paper-fioraldi.pdf
- VUzzer，SYMcc，etc…

CCS Hawkeye: Towards a Desired Directed Grey-box Fuzzer
CCS Revery: from Proof-of-Concept to Exploitable (One Step towards Automatic Exploit Generation)
S&P T-Fuzz: fuzzing by program transformation
(and many many other interesting recent works)



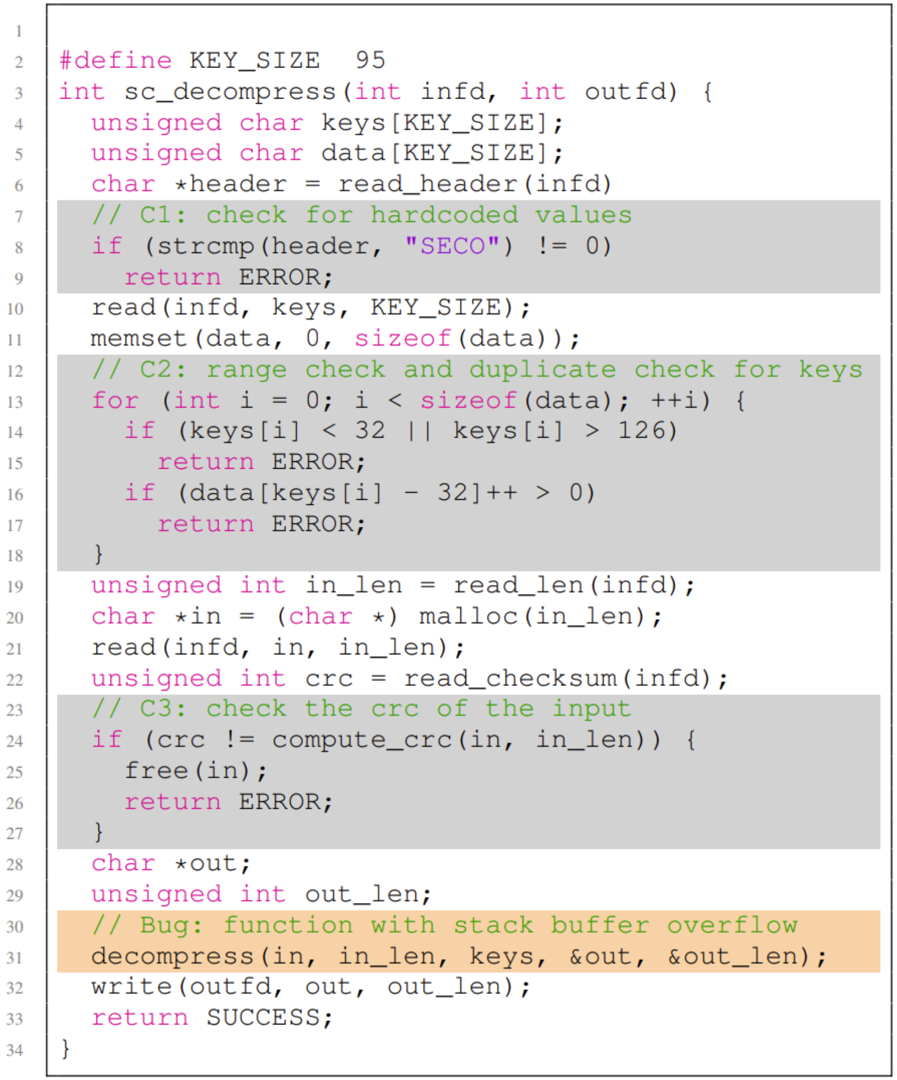
```
                    american fuzzy lop 1.86b (test)

┌─ process timing ─────────────────────────────────────┬─ overall results ─┐
│        run time : 0 days, 0 hrs, 0 min, 2 sec         │  cycles done : 0  │
│   last new path : none seen yet                       │  total paths : 1  │
│ last uniq crash : 0 days, 0 hrs, 0 min, 2 sec         │ uniq crashes : 1  │
│  last uniq hang : none seen yet                       │   uniq hangs : 0  │
├─ cycle progress ────────────────────┬─ map coverage ─┴───────────────────┤
│  now processing : 0 (0.00%)          │    map density : 2 (0.00%)         │
│ paths timed out : 0 (0.00%)          │ count coverage : 1.00 bits/tuple   │
├─ stage progress ────────────────────┼─ findings in depth ────────────────┤
│  now trying : havoc                  │ favored paths : 1 (100.00%)        │
│ stage execs : 1464/5000 (29.28%)     │  new edges on : 1 (100.00%)        │
│ total execs : 1697                   │ total crashes : 39 (1 unique)      │
│  exec speed : 626.5/sec              │   total hangs : 0 (0 unique)       │
├─ fuzzing strategy yields ───────────┴───────────┬─ path geometry ────────┤
│   bit flips : 0/16, 1/15, 0/13                   │    levels : 1          │
│  byte flips : 0/2, 0/1, 0/0                      │   pending : 1          │
│ arithmetics : 0/112, 0/25, 0/0                   │  pend fav : 1          │
│  known ints : 0/10, 0/28, 0/0                    │ own finds : 0          │
│  dictionary : 0/0, 0/0, 0/0                      │  imported : n/a        │
│       havoc : 0/0, 0/0                           │  variable : 0          │
│        trim : n/a, 0.00%                         ├────────────────────────┘
└──────────────────────────────────────────────────┘             [cpu: 92%]
```

American fuzzy lop's afl-fuzz running on a test program

Fuzz Testing

- Problem:
  - Still hard to reach some branches

hard to pass this check by fuzz



```c

#define KEY_SIZE  95
int sc_decompress(int infd, int outfd) {
  unsigned char keys[KEY_SIZE];
  unsigned char data[KEY_SIZE];
  char *header = read_header(infd)
  // C1: check for hardcoded values
  if (strcmp(header, "SECO") != 0)
    return ERROR;
  read(infd, keys, KEY_SIZE);
  memset(data, 0, sizeof(data));
  // C2: range check and duplicate check for keys
  for (int i = 0; i < sizeof(data); ++i) {
    if (keys[i] < 32 || keys[i] > 126)
      return ERROR;
    if (data[keys[i] - 32]++ > 0)
      return ERROR;
  }
  unsigned int in_len = read_len(infd);
  char *in = (char *) malloc(in_len);
  read(infd, in, in_len);
  unsigned int crc = read_checksum(infd);
  // C3: check the crc of the input
  if (crc != compute_crc(in, in_len)) {
    free(in);
    return ERROR;
  }
  char *out;
  unsigned int out_len;
  // Bug: function with stack buffer overflow
  decompress(in, in_len, keys, &out, &out_len);
  write(outfd, out, out_len);
  return SUCCESS;
}
```

Listing 1: An example containing various sanity checks

S&P'18 T-Fuzz: fuzzing by program transformation

Fuzz Testing

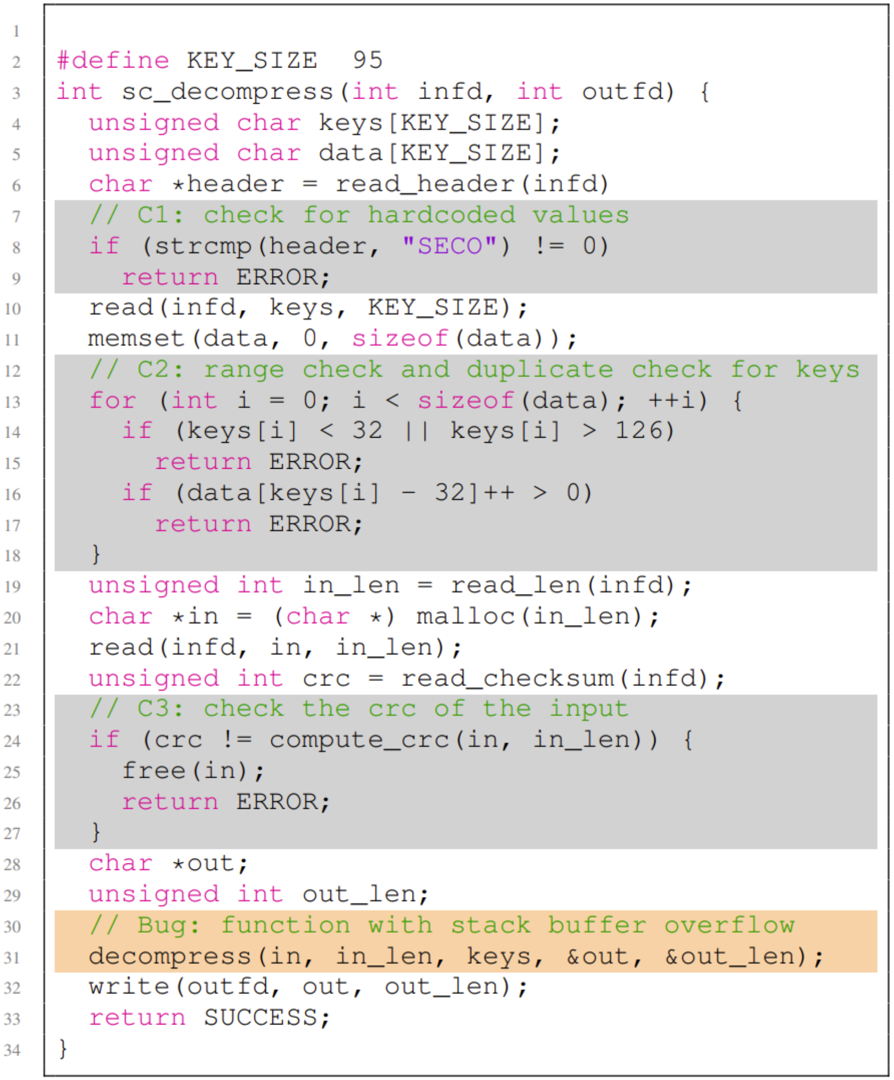
- Problem:
  - Still hard to reach some branches

hard to pass this check by fuzz

if fuzz is “clever” enough and applied some genetic algorithms, it may pass the magic number check after a while

hard to pass crc check even after mutations

but the fuzz is nearly impossible to pass CRC check
CRC is a kind of hash algorithm, it often used to check the integrity of data (e.g. a zip archive can store CRC to check weather uncompressed file is correct)



```

#define KEY_SIZE  95
int sc_decompress(int infd, int outfd) {
  unsigned char keys[KEY_SIZE];
  unsigned char data[KEY_SIZE];
  char *header = read_header(infd)
  // C1: check for hardcoded values
  if (strcmp(header, "SECO") != 0)
    return ERROR;
  read(infd, keys, KEY_SIZE);
  memset(data, 0, sizeof(data));
  // C2: range check and duplicate check for keys
  for (int i = 0; i < sizeof(data); ++i) {
    if (keys[i] < 32 || keys[i] > 126)
      return ERROR;
    if (data[keys[i] - 32]++ > 0)
      return ERROR;
  }
  unsigned int in_len = read_len(infd);
  char *in = (char *) malloc(in_len);
  read(infd, in, in_len);
  unsigned int crc = read_checksum(infd);
  // C3: check the crc of the input
  if (crc != compute_crc(in, in_len)) {
    free(in);
    return ERROR;
  }
  char *out;
  unsigned int out_len;
  // Bug: function with stack buffer overflow
  decompress(in, in_len, keys, &out, &out_len);
  write(outfd, out, out_len);
  return SUCCESS;
}
```

Listing 1: An example containing various sanity checks

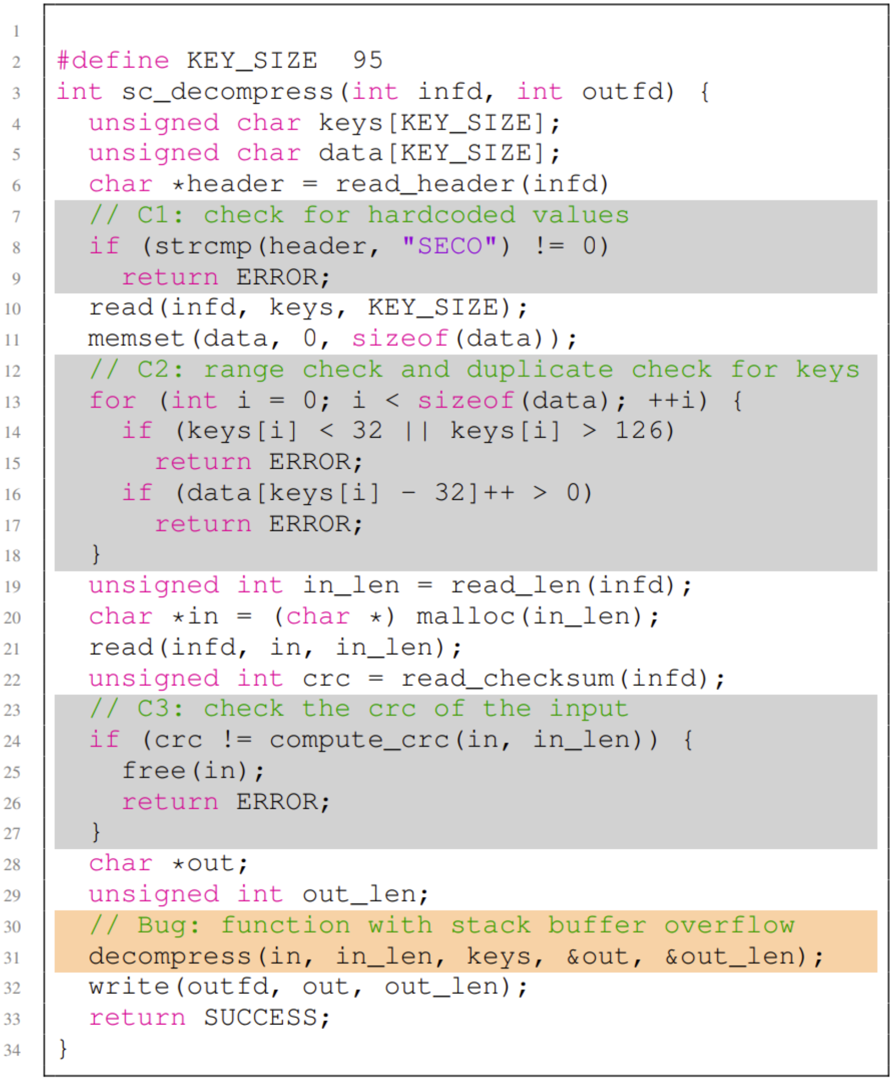
S&P‘18 T-Fuzz: fuzzing by program transformation

Fuzz Testing

hard to pass this check by fuzz

- Problem:
    - Still hard to reach some branches
- Another technique: symbolic execution

hard to pass crc check even after mutations



```c

#define KEY_SIZE  95
int sc_decompress(int infd, int outfd) {
  unsigned char keys[KEY_SIZE];
  unsigned char data[KEY_SIZE];
  char *header = read_header(infd)
  // C1: check for hardcoded values
  if (strcmp(header, "SECO") != 0)
    return ERROR;
  read(infd, keys, KEY_SIZE);
  memset(data, 0, sizeof(data));
  // C2: range check and duplicate check for keys
  for (int i = 0; i < sizeof(data); ++i) {
    if (keys[i] < 32 || keys[i] > 126)
      return ERROR;
    if (data[keys[i] - 32]++ > 0)
      return ERROR;
  }
  unsigned int in_len = read_len(infd);
  char *in = (char *) malloc(in_len);
  read(infd, in, in_len);
  unsigned int crc = read_checksum(infd);
  // C3: check the crc of the input
  if (crc != compute_crc(in, in_len)) {
    free(in);
    return ERROR;
  }
  char *out;
  unsigned int out_len;
  // Bug: function with stack buffer overflow
  decompress(in, in_len, keys, &out, &out_len);
  write(outfd, out, out_len);
  return SUCCESS;
}
```

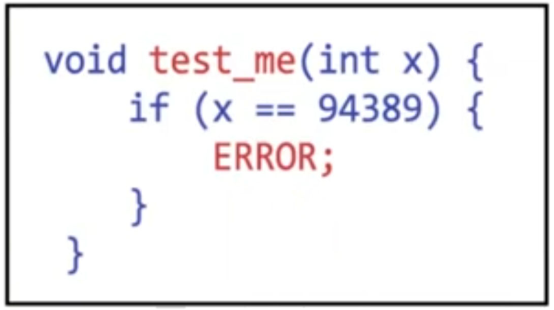
Listing 1: An example containing various sanity checks

S&P'18 T-Fuzz: fuzzing by program transformation

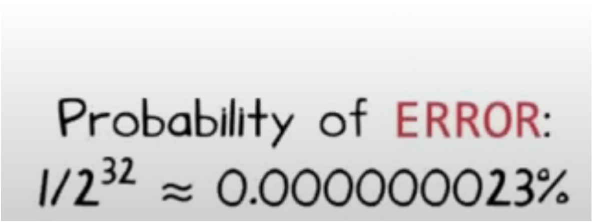
Symbolic Execution

- ①Use symbol to represent variables
- ②Simulate program execution
- ③Extract and solve constraints in execution path

- generate constraints in every path
- constraint solver: z3, cvc5



```
void test_me(int x) {
    if (x == 94389) {
        ERROR;
    }
}
```



Probability of ERROR:

$1/2^{32} \approx 0.000000023\%$

some path is hard to reach by randomly fuzzing
If we can extract the constraint (x==94389), we can easily find the input to reach those path

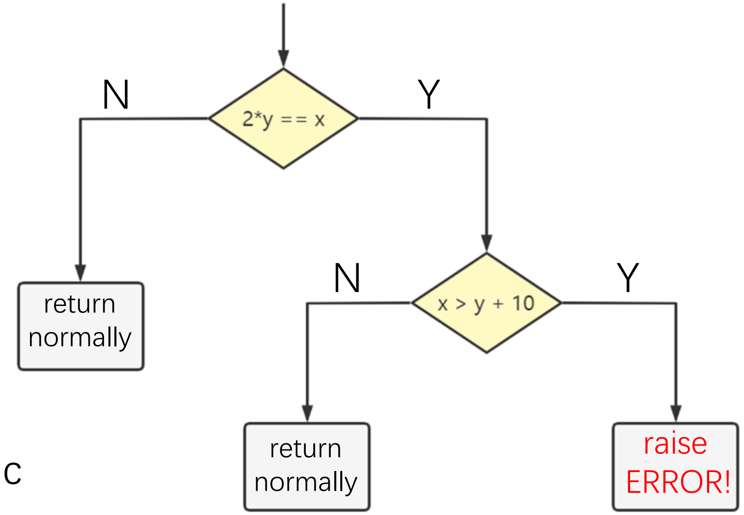
Symbolic Execution

```
int main(){
    x = read_int();
    y = read_int();
    z = 2 * x;
    if (z == x){
        if (x > y+10)
            ERROR;
    }
}
```

- ①Use symbol to represent variables
- ②Simulate program execution
- ③Extract and solve constraints in execution path

- generate constraints in every path
- constraint solver: z3, cvc5

let’s see how symbolic execution works ->



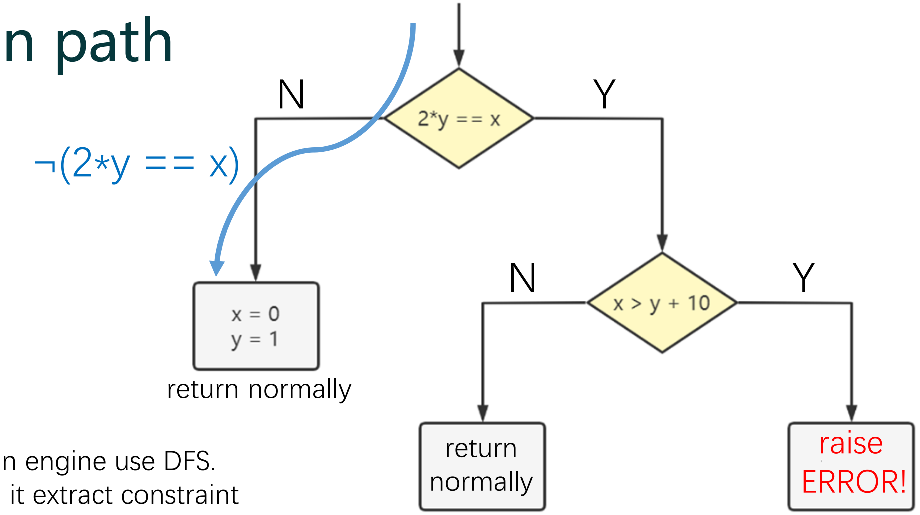
Frqwuro#Iorz#Judsk

Symbolic Execution

```
int main(){
    x = read_int();
    y = read_int();
    z = 2 * x;
    if (z == x){
        if (x > y+10)
            ERROR;
    }
}
```

- ①Use symbol to represent variables
- ②Simulate program execution
- ③Extract and solve constraints in execution path

- generate constraints in every path
- constraint solver: z3, cvc5



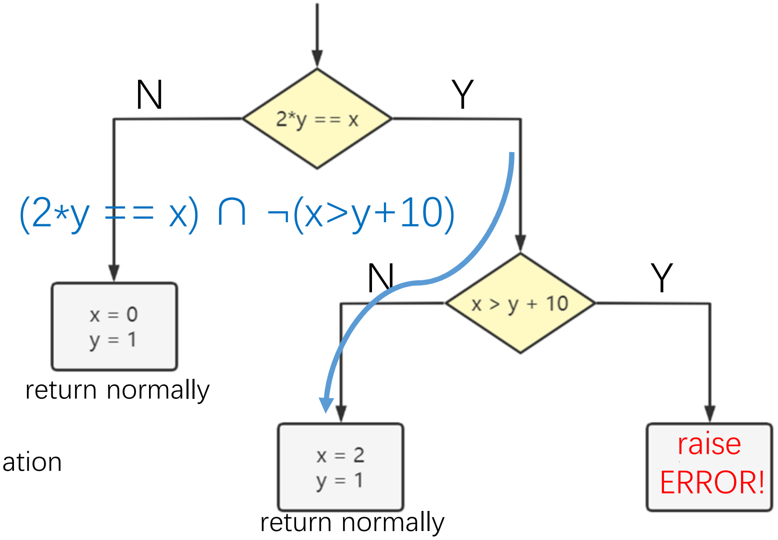
assume simulation engine use DFS.
For the first path, it extract constraint
and get a satisfied input

Symbolic Execution

```
int main(){
    x = read_int();
    y = read_int();
    z = 2 * x;
    if (z == x){
        if (x > y+10)
            ERROR;
    }
}
```

- Use symbol to represent variables
- Simulate program execution
- Extract and solve constraints in execution path

- generate constraints in every path
- constraint solver: z3, cvc5



(2*y == x) ∩ ¬(x>y+10)

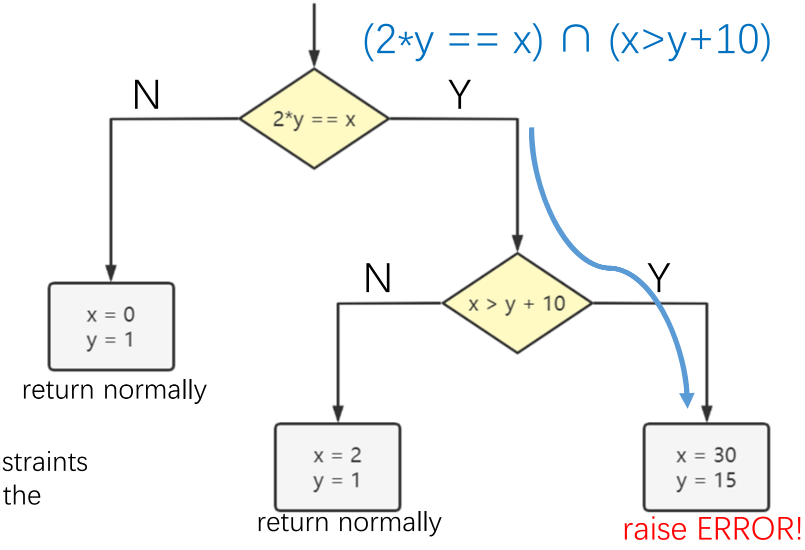
for second one it is a combination
of two constraints

Symbolic Execution

```
int main(){
    x = read_int();
    y = read_int();
    z = 2 * x;
    if (z == x){
        if (x > y+10)
            ERROR;
    }
}
```

- Use symbol to represent variables
- Simulate program execution
- Extract and solve constraints in execution path

- generate constraints in every path
- constraint solver: z3, cvc5

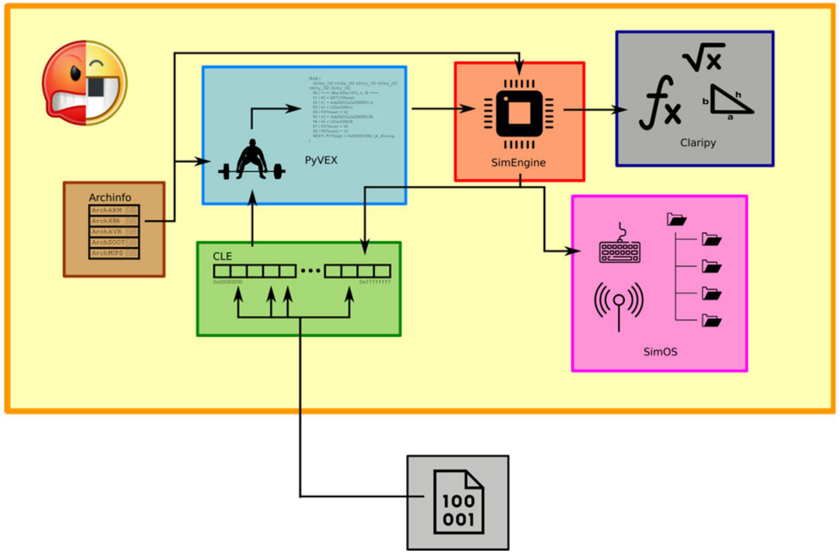


similarly, it extract simple constraints
and can get a result to reach the
path that contains bug
linear problems can be solved by
constraint solver easily

Symbolic Execution

- angr: platform-agnostic binary analysis framework
- convert input to bit vector, simulate program instructions

Shoshitaishvili, Yan, et al. "Sok:(state of) the art of war: Offensive techniques in binary analysis." *2016 IEEE symposium on security and privacy (SP)*. IEEE, 2016.



angr life cycle

- KLEE: source code analysis framework
- compile from source code and make instrumentation
- http://klee.doc.ic.ac.uk/ (try it online)

Cadar, Cristian, Daniel Dunbar, and Dawson R. Engler. "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs." *OSDI*. Vol. 8. 2008.

- Mayhem， Triton, etc…

Symbolic Execution

- Problem:
  - state exploitation when handling many branches
    - With each if statement, the number of possible branches might double. The growth of the problem is exponential with respect to the size of the program.
  - cost many time solving large constraint (often a constraint will be solved many times)

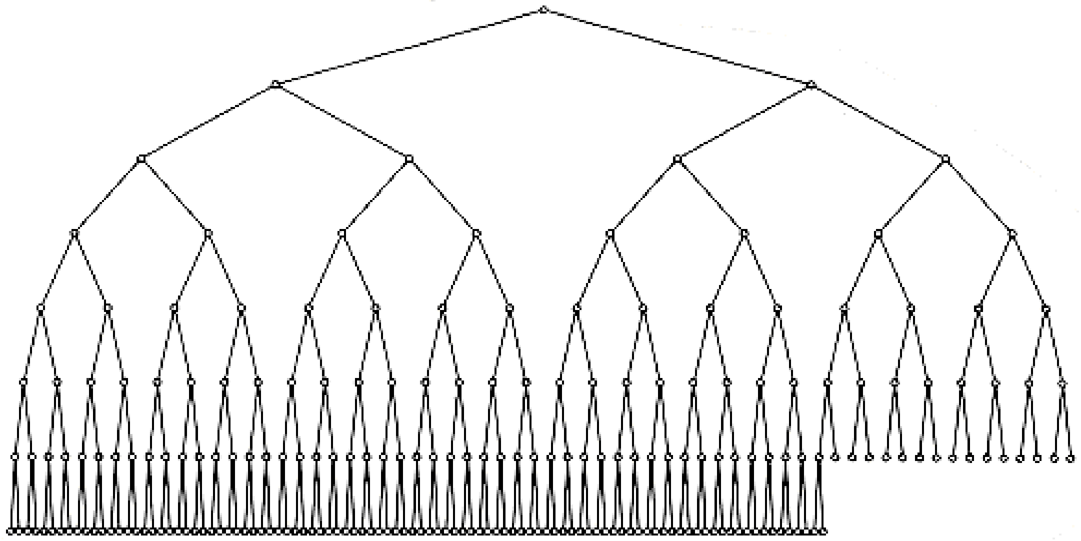


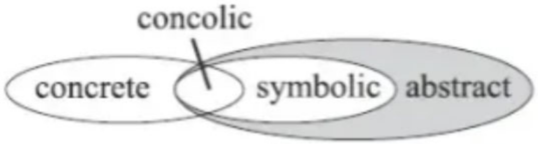
Image source: http://www.icodeguru.com/vc/10book/books/book3/chap6.htm

Symbolic Execution

- Problem:
  - state exploitation when handling many branches
  - cost many time solving large constraint (often a constraint will be solved many times)
- Combination of two techniques: Concolic Execution

Concolic Execution

- Skip solving unnecessary constraints
    - use fuzzing to accelerate
    - use symbolic execution to reach deeper branches



- An example：Diller

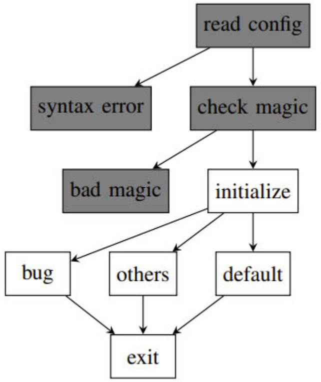


Fig. 1. The nodes initially found by the fuzzer.

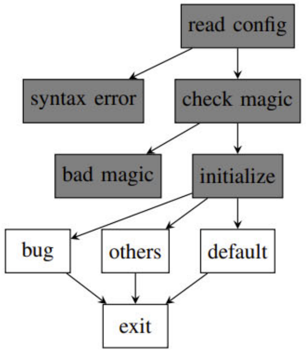


Fig. 2. The nodes found by the first invocation of concolic execution.

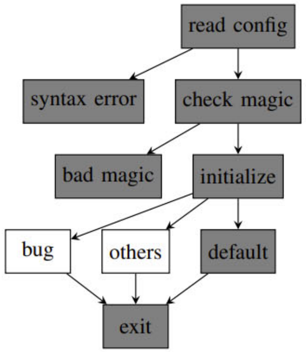


Fig. 3. The nodes found by the fuzzer, supplemented with the result of the first Driller run.

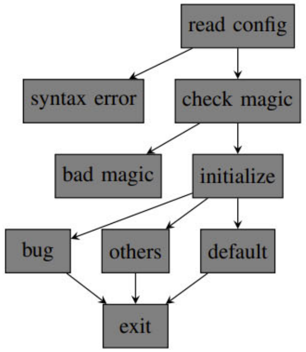
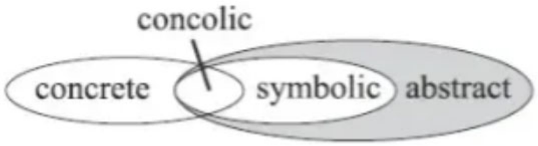


Fig. 4. The nodes found by the second invocation of concolic execution.

Stephens, Nick, et al. "Driller: Augmenting fuzzing through selective symbolic execution." *NDSS*. 2016.

Concolic Execution

- Skip solving unnecessary constraints
    - use fuzzing to accelerate
    - use symbolic execution to reach deeper branches



- An example：Diller

after fuzz is hard to continue, only do symbolic execution on this node

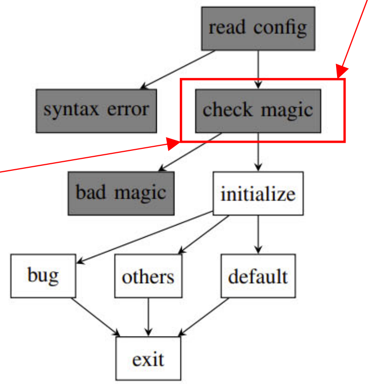


Fig. 1. The nodes initially found by the fuzzer.

```
// C1: check for hardcoded values
if (strcmp(header, "SECO") != 0)
  return ERROR;
```

check magic (fuzzer is hard to pass this check)

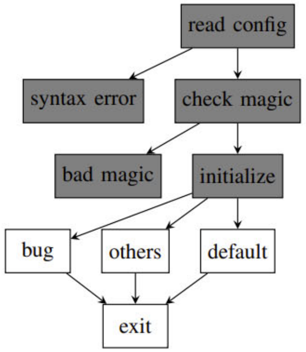


Fig. 2. The nodes found by the first invocation of concolic execution.

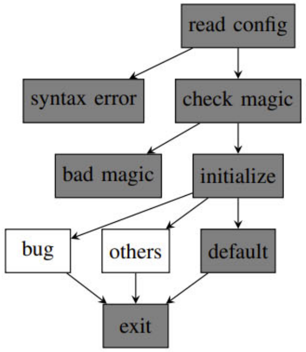


Fig. 3. The nodes found by the fuzzer, supplemented with the result of the first Driller run.

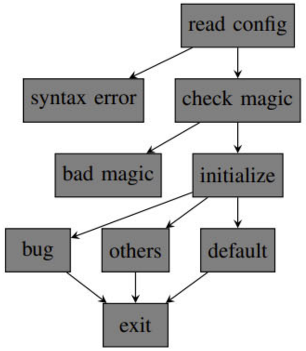
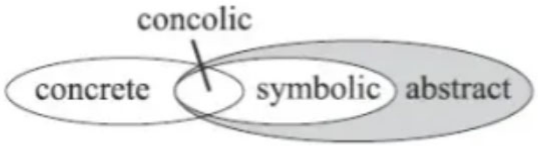


Fig. 4. The nodes found by the second invocation of concolic execution.

Stephens, Nick, et al. "Driller: Augmenting fuzzing through selective symbolic execution." *NDSS*. 2016.

Concolic Execution

- Skip solving unnecessary constraints
  - use fuzzing to accelerate
  - use symbolic execution to reach deeper branches
- An example： Diller



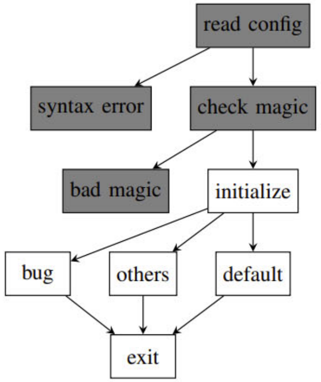


Fig. 1. The nodes initially found by the fuzzer.

find input that can reach new path,
use this input to continue fuzzing

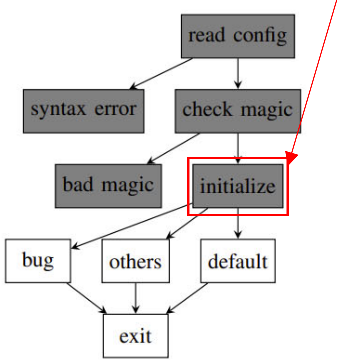


Fig. 2. The nodes found by the first invocation of concolic execution.

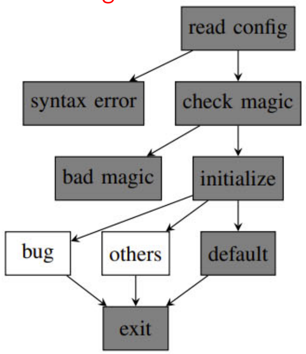


Fig. 3. The nodes found by the fuzzer, supplemented with the result of the first Driller run.

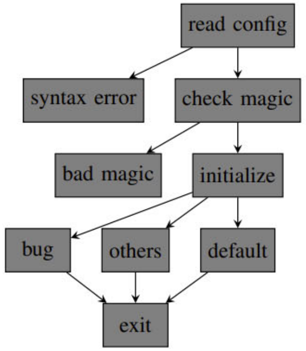
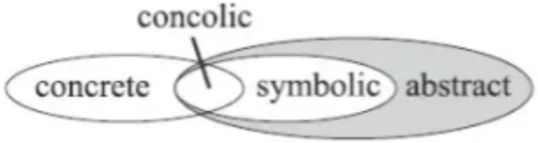


Fig. 4. The nodes found by the second invocation of concolic execution.

Stephens, Nick, et al. "Driller: Augmenting fuzzing through selective symbolic execution." *NDSS*. 2016.

Concolic Execution

- Skip solving unnecessary constraints
  - use fuzzing to accelerate
  - use symbolic execution to reach deeper branches



- An example：Diller

after fuzz for a while, do symbolic execution on this edge and try find input that can reach another path

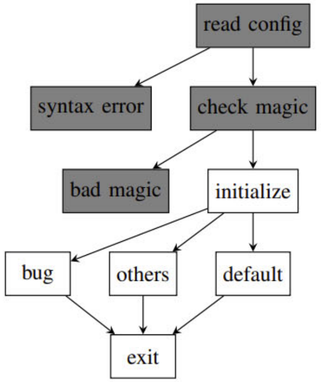


Fig. 1. The nodes initially found by the fuzzer.

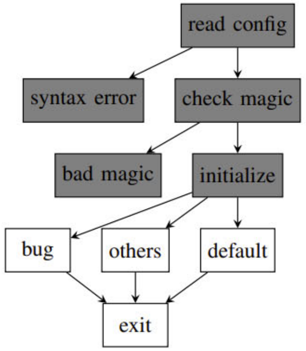


Fig. 2. The nodes found by the first invocation of concolic execution.

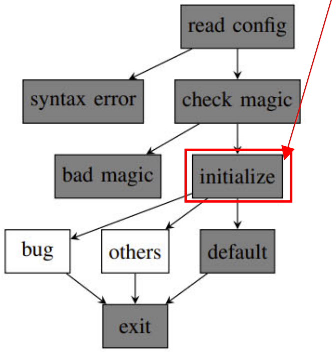


Fig. 3. The nodes found by the fuzzer, supplemented with the result of the first Driller run.

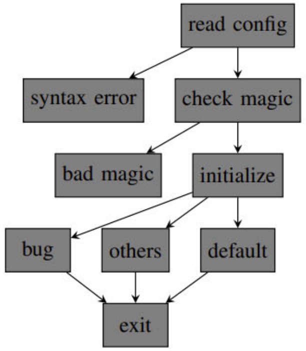
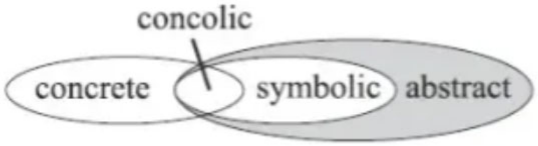


Fig. 4. The nodes found by the second invocation of concolic execution.

Stephens, Nick, et al. "Driller: Augmenting fuzzing through selective symbolic execution." *NDSS*. 2016.

Concolic Execution

- Skip solving unnecessary constraints
  - use fuzzing to accelerate
  - use symbolic execution to reach deeper branches
- An example：Diller



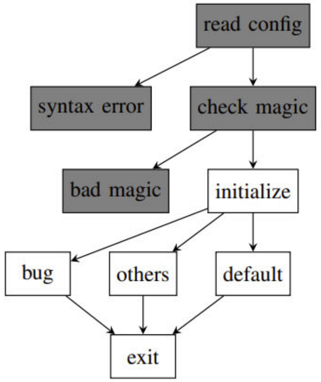


Fig. 1. The nodes initially found by the fuzzer.

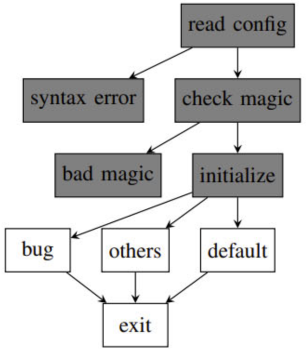


Fig. 2. The nodes found by the first invocation of concolic execution.

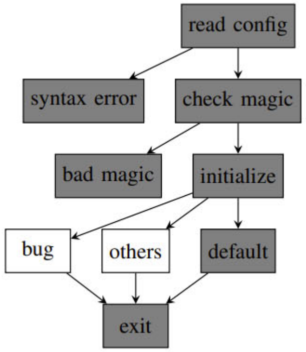


Fig. 3. The nodes found by the fuzzer, supplemented with the result of the first Driller run.

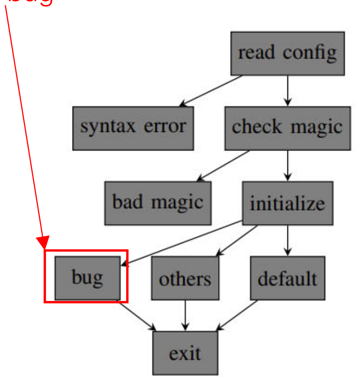


Fig. 4. The nodes found by the second invocation of concolic execution.

symbolic execution find input
to reach a path with bug

Stephens, Nick, et al. "Driller: Augmenting fuzzing through selective symbolic execution." *NDSS*. 2016.

Summary:

- Fuzz can quickly find some bug of large real word program, but fuzz is hard to reach some complicated path
- We use symbolic execution to find more complicated bugs, but it may produce too many unnecessary states
- We can combine fuzzing and symbolic execution

Happy exploiting!

Extra Notes:

- Fuzz target not only contains source code and binary
- browser，blockchain，compiler，kernel... everything can be tested!

- There are some other program analyzing techniques like module checking