Billions and Billions of Constraints: Whitebox Fuzz Testing in Production

Ella Bounimova, Patrice Godefroid, **David Molnar**
Microsoft Research

Microsoft Fixes 21 Security Bugs in Windows, IE, Office

Each bug like this costs Microsoft ~USD 1 million
If you're unlucky, it could cost you too…
Many such bugs are "corner cases" in C/C++ code
File parsers: video, audio, pictures…

```
int foo(int x) { // x is an input
  int y = x + 3;
  if (y == 13) abort();   // error
  return 0;
}
```

```
int foo(int x) { // x is an input
   int y = x + 3;
   if (y == 13) abort();    // error
   return 0;
}
```

Random choice of x: one chance in 2^32 to find error

"Fuzz testing" Widely used, remarkably effective!

```
int foo(int x) { // x is an input
  int y = x + 3;
  if (y == 13) abort();   // error
  return 0;
}
```

Core idea:

1) Pick an arbitrary “seed” input
2) Record path taken by program executing on “seed”
3) Create symbolic abstraction of path and generate tests

```
int foo(int x) { // x is an input
  int y = x + 3;
  if (y == 13) abort();   // error
  return 0;
}
```

Example:

1) Pick x to be 5
2) Record y = 5+3 = 8, record program tests “8 ?= 13”
3) Symbolic *path condition*: “x + 3 != 13”

```
void top(char input[4])
{
    int cnt = 0;
    if (input[0] == 'b') cnt++;
    if (input[1] == 'a') cnt++;
    if (input[2] == 'd') cnt++;
    if (input[3] == '!') cnt++;
    if (cnt >= 3) crash();
}
```

input = "good"

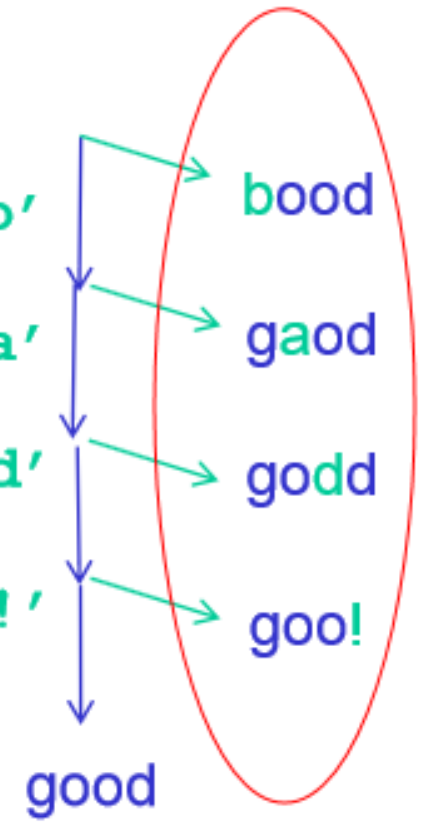
Path constraint:

$I_0 != 'b' \rightarrow I_0 = 'b'$

$I_1 != 'a' \rightarrow I_1 = 'a'$

$I_2 != 'd' \rightarrow I_2 = 'd'$

$I_3 != '!' \rightarrow I_3 = '!'$

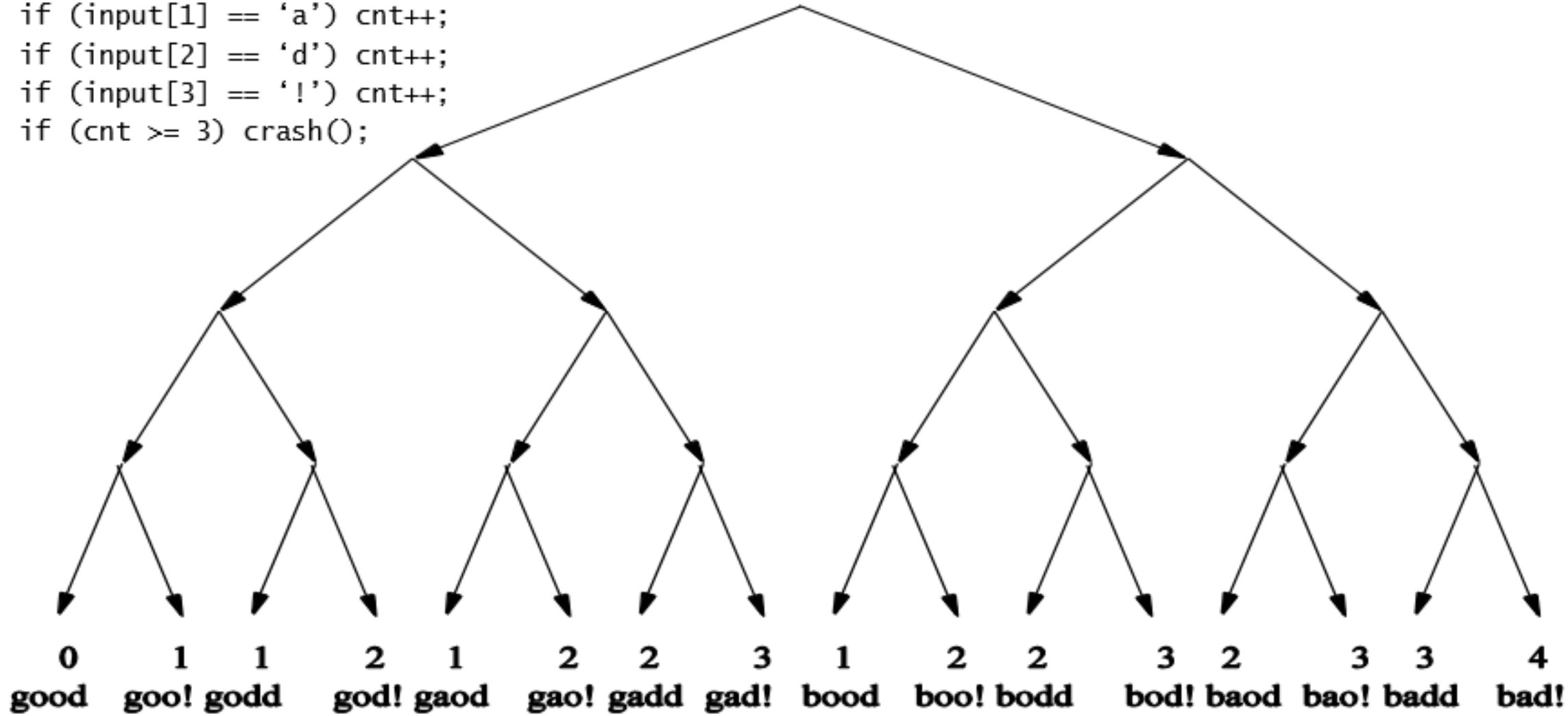


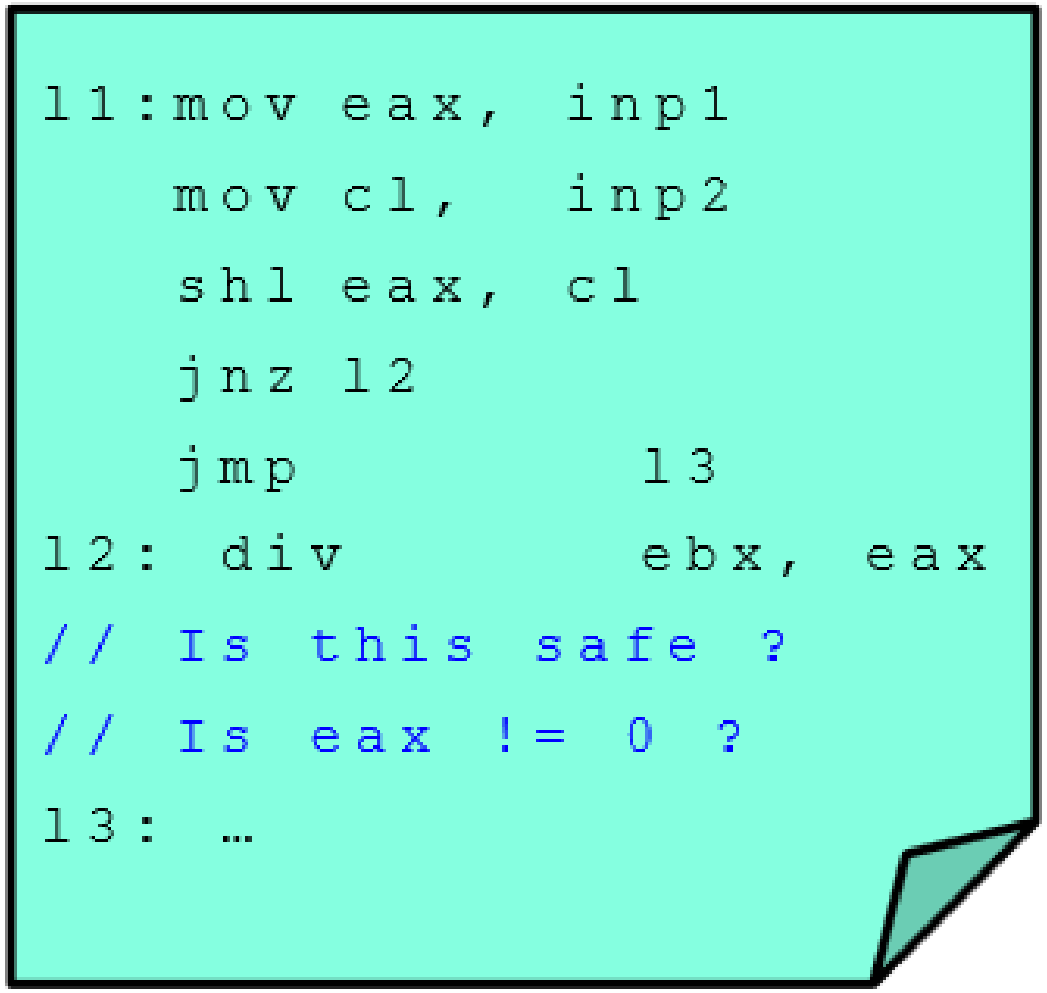
Gen 1

Negate each constraint in path constraint

Solve new constraint → new input

```
void top(char input[4])
{
   int cnt = 0;
   if (input[0] == 'b') cnt++;
   if (input[1] == 'a') cnt++;
   if (input[2] == 'd') cnt++;
   if (input[3] == '!') cnt++;
   if (cnt >= 3) crash();
}
```





Work with x86 **binary code** on Windows
Leverage full-instruction-trace recording

Pros:

- If you can run it, you can analyze it
- Don't care about build processes
- Don't care if source code available

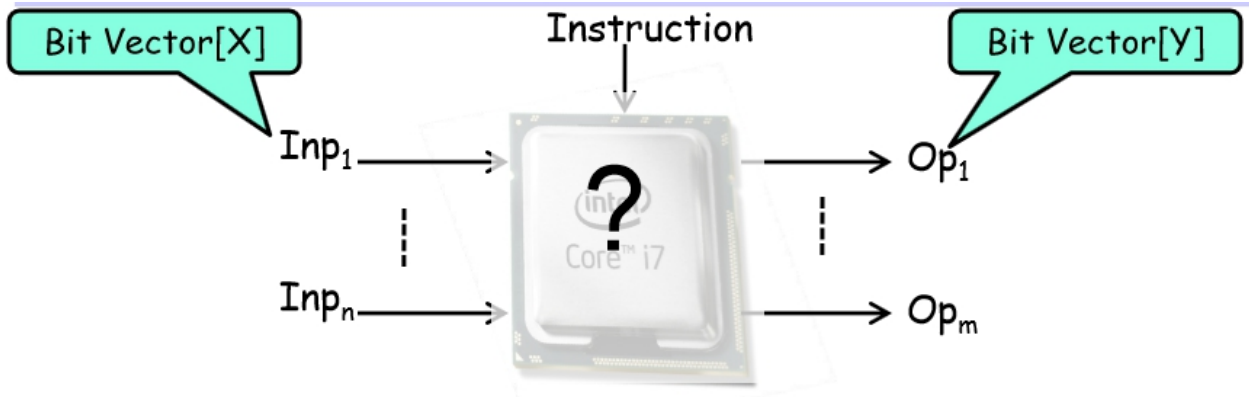
Cons:

- Lose programmer's intent (e.g. types)
- Hard to "see" string manipulation, memory object graph manipulation, etc.

SHLD—Double Precision Shift Left (Continued)

Operation

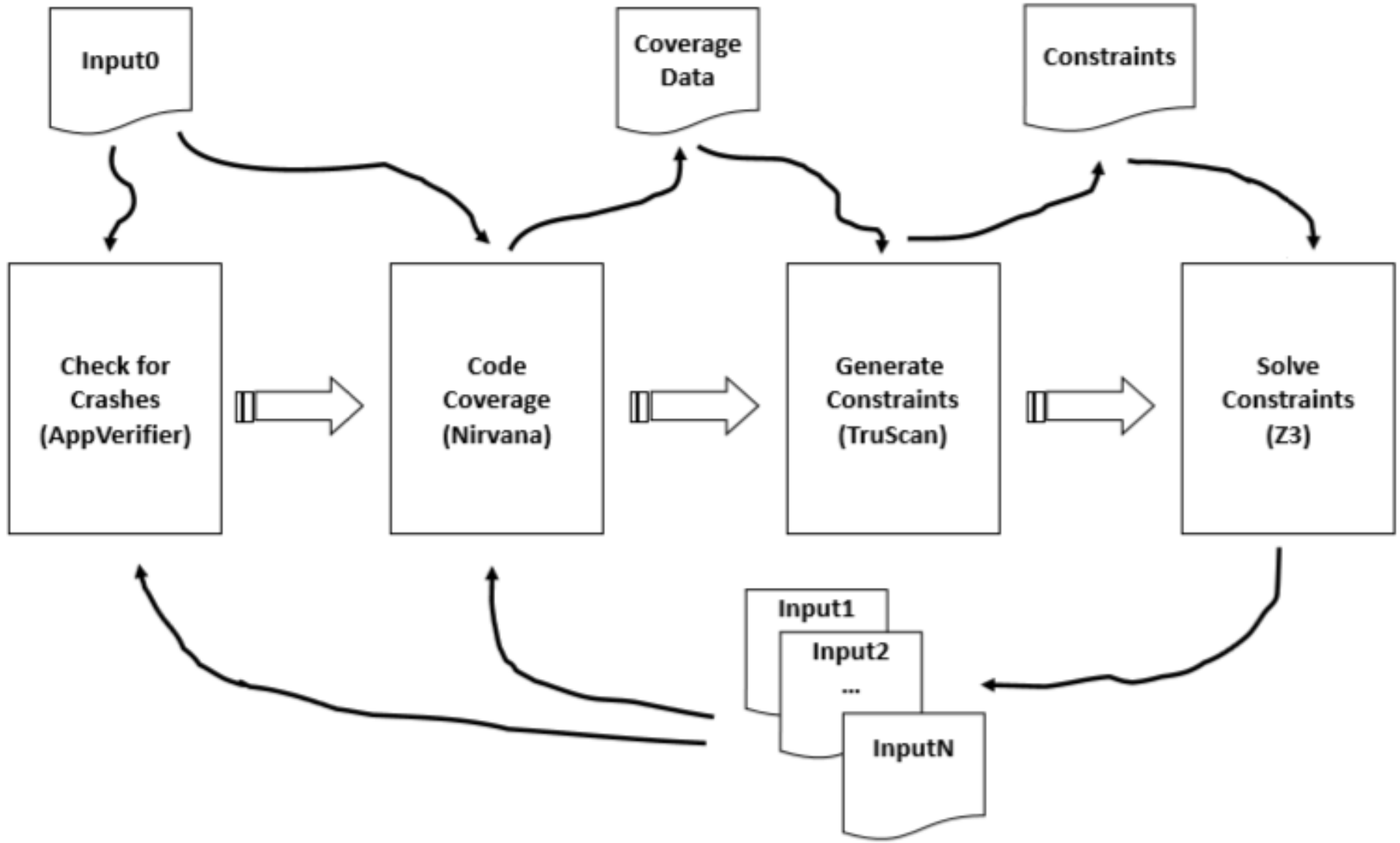
```
COUNT ← COUNT MOD 32;
SIZE ← OperandSize
IF COUNT = 0
    THEN
        no operation
    ELSE
        IF COUNT ≥ SIZE
            THEN (* Bad parameters *)
                DEST is undefined;
                CF, OF, SF, ZF, AF, PF are undefined;
            ELSE (* Perform the shift *)
                CF ← BIT[DEST, SIZE – COUNT];
                (* Last bit shifted out on exit *)
                FOR i ← SIZE – 1 DOWNTO COUNT
                DO
                    Bit(DEST, i) ← Bit(DEST, i – COUNT);
                OD;
                FOR i ← COUNT – 1 DOWNTO 0
                DO
                    BIT[DEST, i] ← BIT[SRC, i – COUNT + SIZE];
                OD;
        FI;
FI;
```



Hand-written models (so far)
Uses Z3 support for non-linear operations

Normally “concretize” memory accesses where address is symbolic

# instructions executed	1,455,506,956
# instr. executed after 1st read from file	928,718,575
# constraints generated (full path constraint)	25,958
# constraints dropped due to cache hits	244,170
# constraints dropped due to limit exceeded	193,953
# constraints satisfiable (= # new tests)	2,980
# constraints unsatisfiable	22,978
# constraint solver timeouts (>5 secs)	0
symbolic execution time (secs)	2,745
constraint solving time (secs)	953

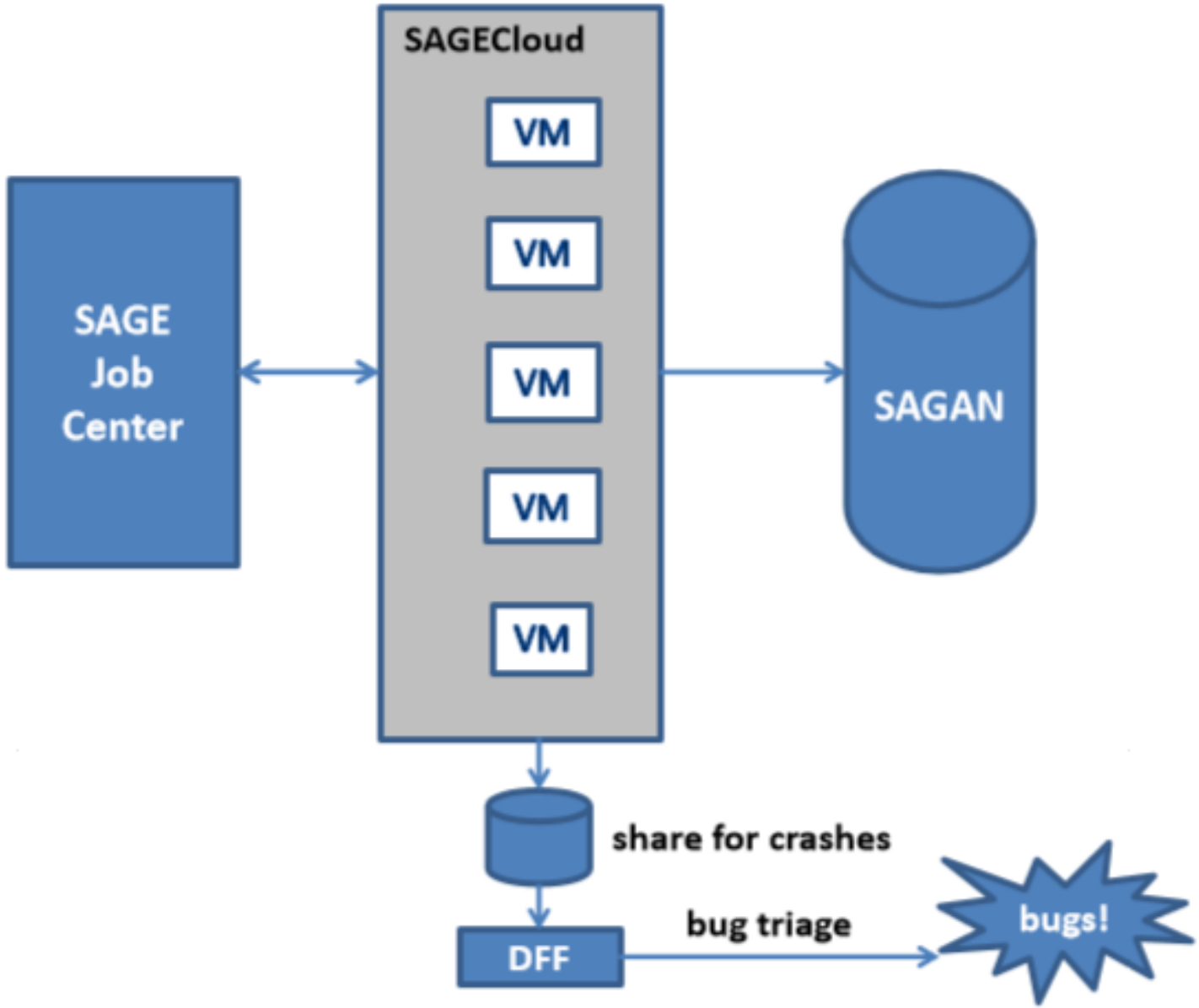

Input0
Coverage Data
Constraints
Check for Crashes (AppVerifier)
Code Coverage (Nirvana)
Generate Constraints (TruScan)
Solve Constraints (Z3)
Input1
Input2
...
InputN

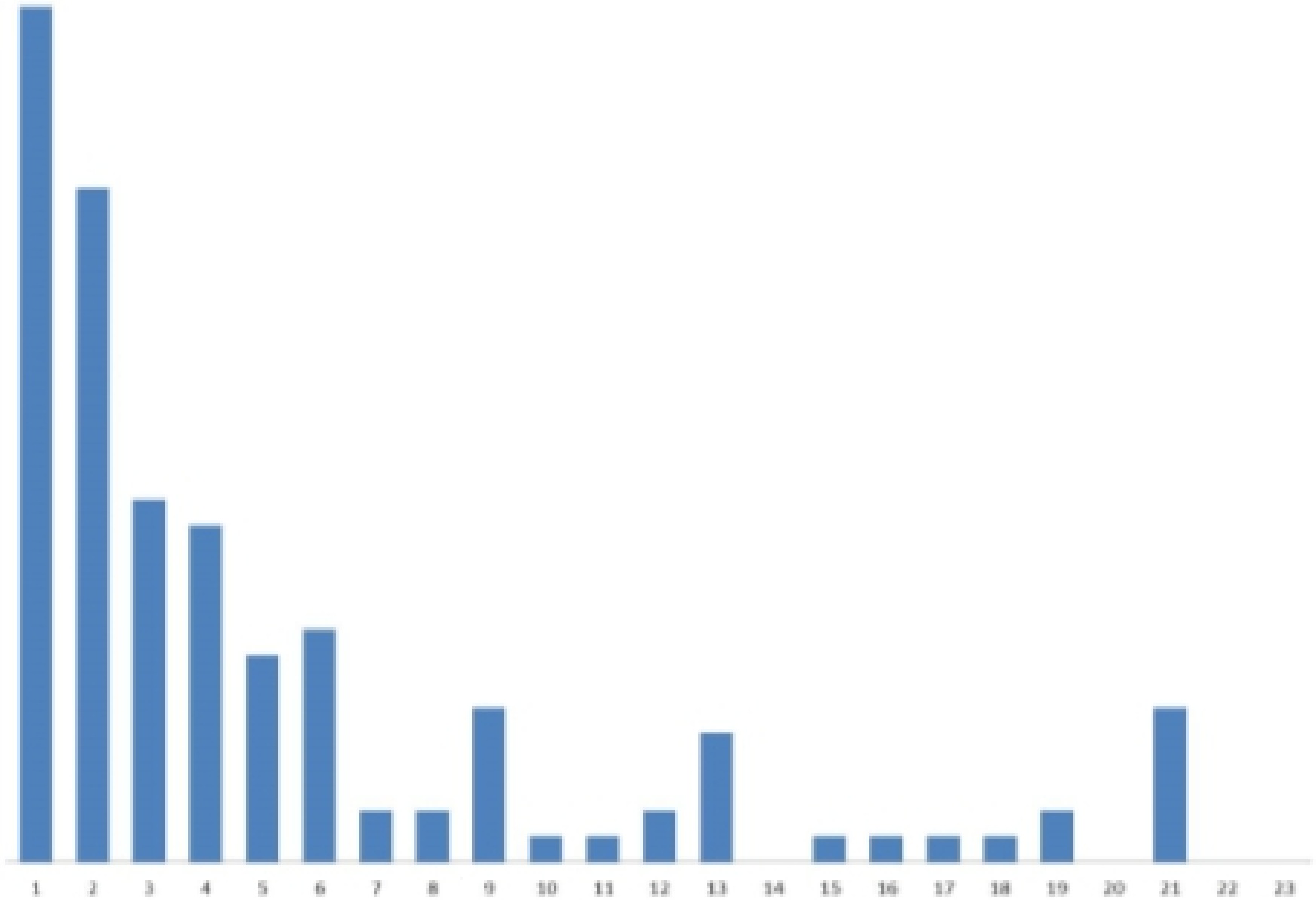
```
00000000h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000010h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000020h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000030h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000040h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000050h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000060h: 00 00 00 00                                     ; ....
```

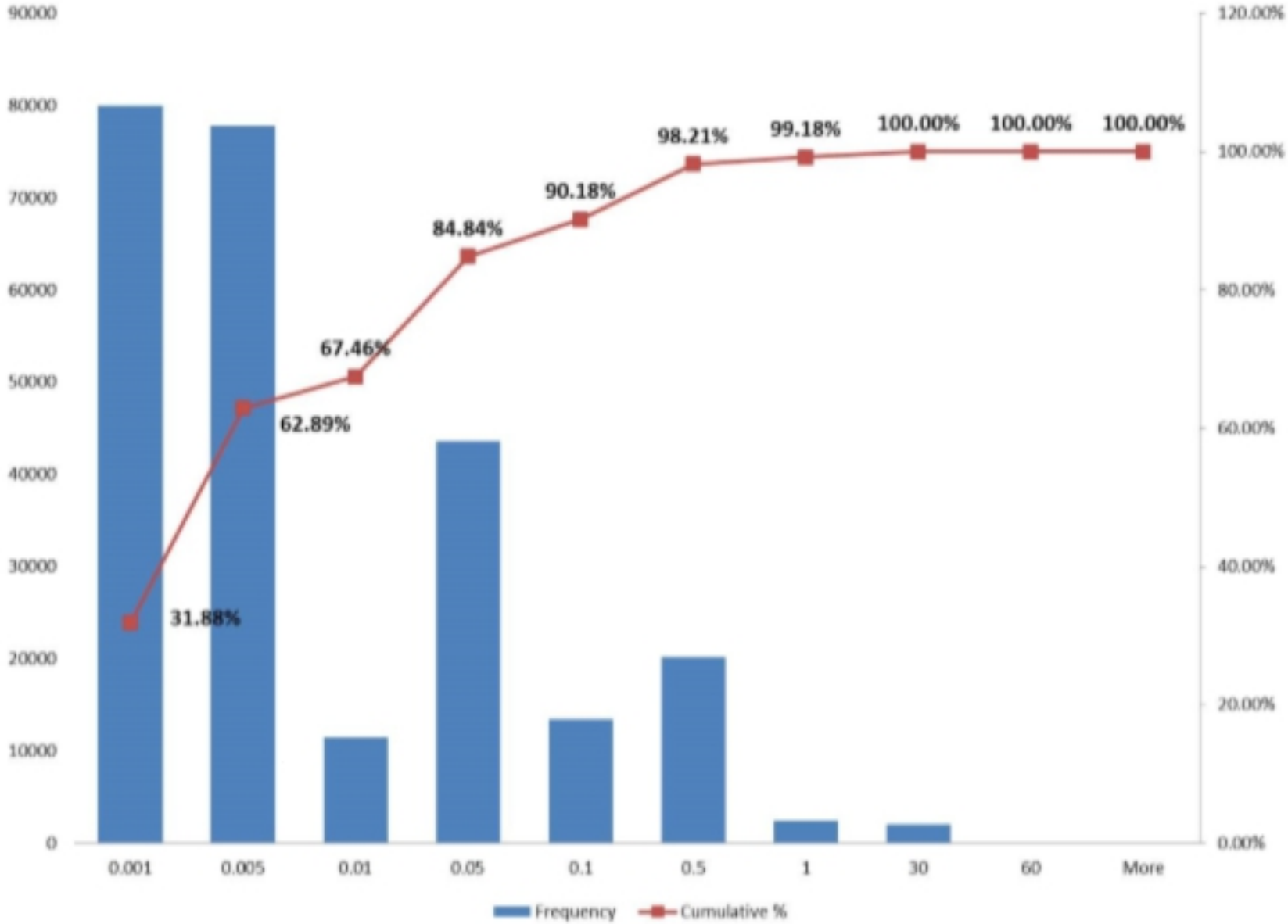
Generation 0 – seed file

```
00000000h: 52 49 46 46 3D 00 00 00 ** ** ** 20 00 00 00 00 ; RIFF=...*** ....
00000010h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000020h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000030h: 00 00 00 00 73 74 72 68 00 00 00 00 76 69 64 73 ; ....strh....vids
00000040h: 00 00 00 00 73 74 72 66 B2 75 76 3A 28 00 00 00 ; ....strf²uv:(...
00000050h: 00 00 00 00 00 00 00 00 00 00 00 00 01 00 00 00 ; ................
00000060h: 00 00 00 00                                     ; ....
```

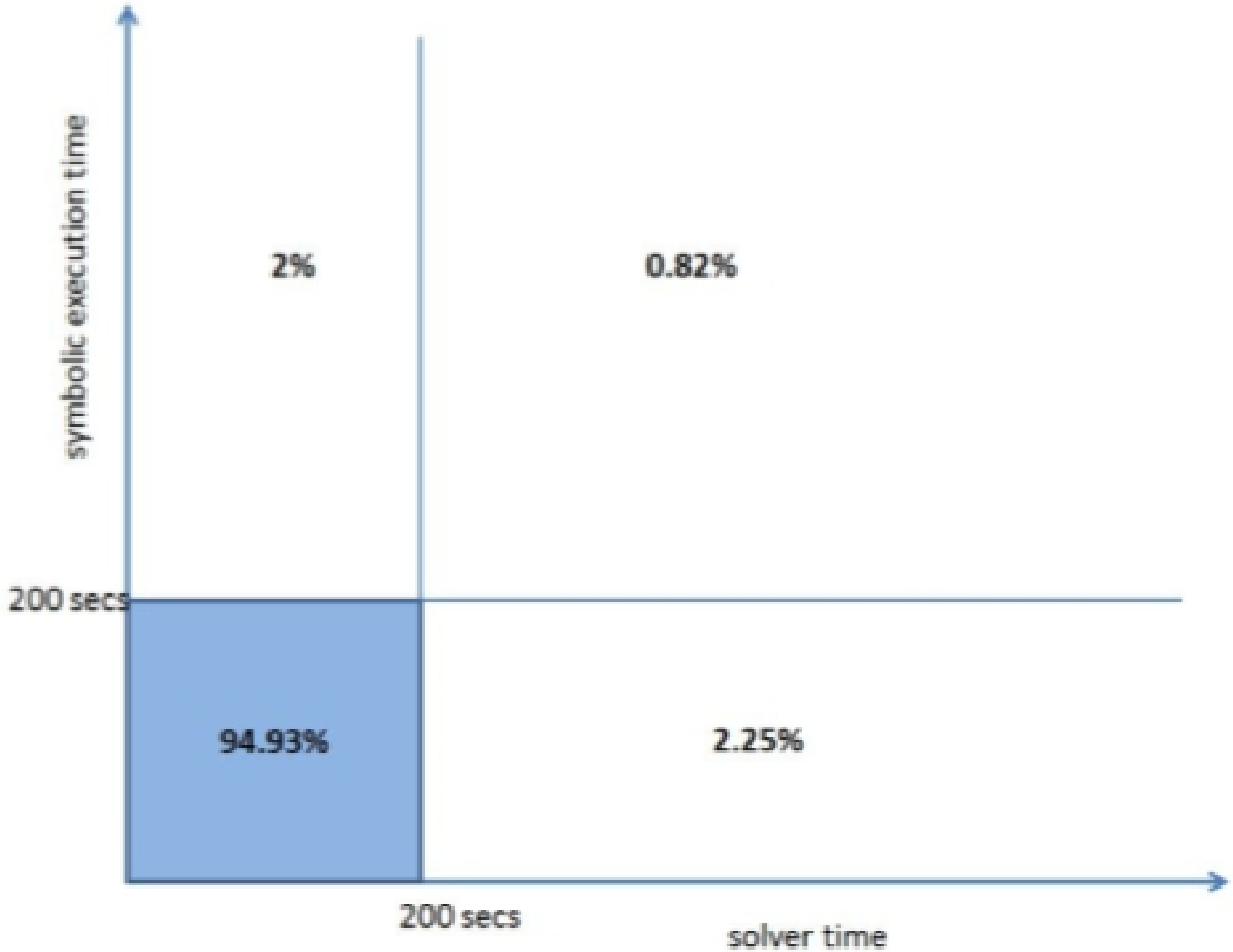
Generation 10 – crash bucket 1212954973!

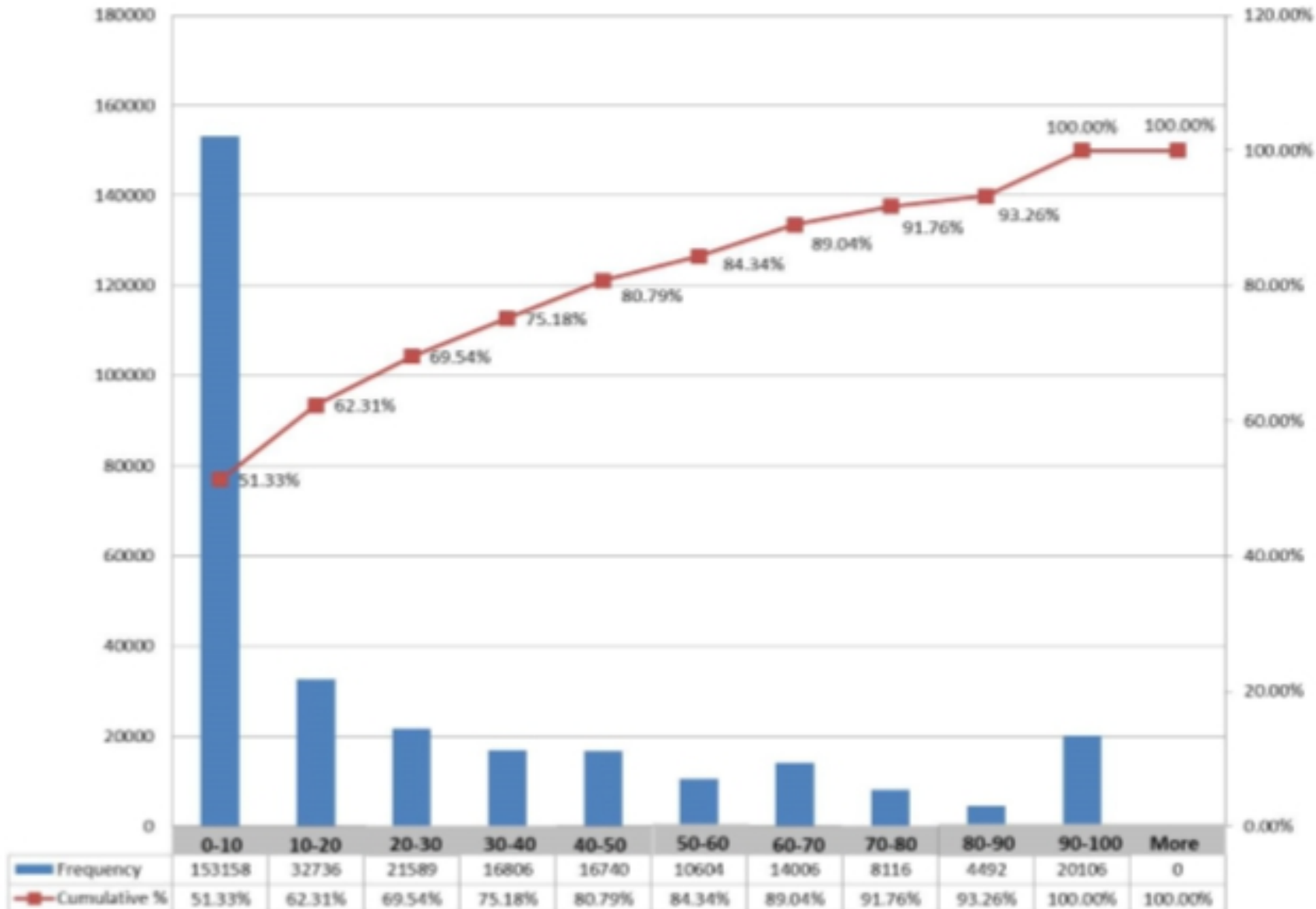

SAGECloud
VM
VM
VM
VM
VM
SAGE
Job
Center
SAGAN
share for crashes
bug triage
DFF
bugs!


1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23

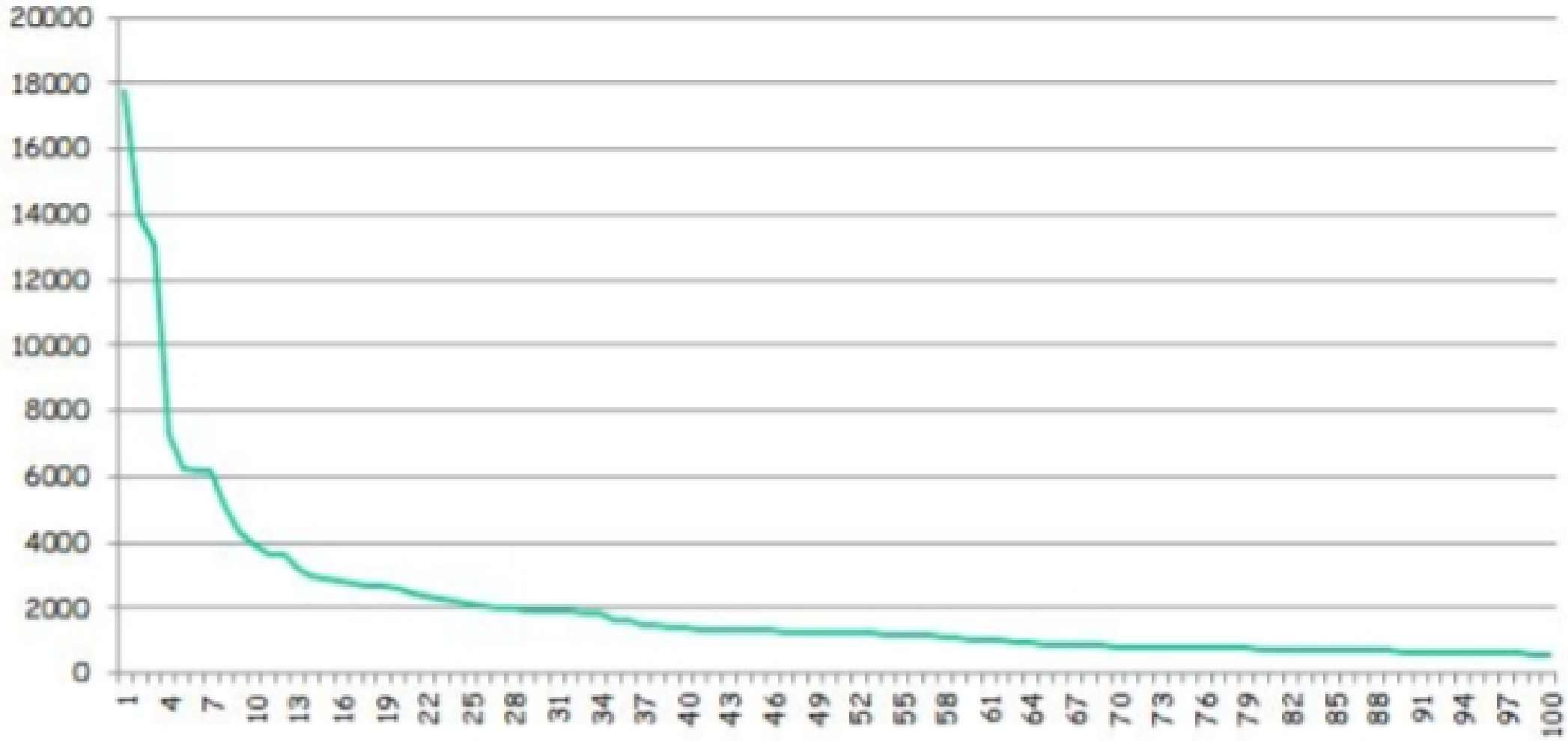

90000
80000
70000
60000
50000
40000
30000
20000
10000
0
120.00%
100.00%
80.00%
60.00%
40.00%
20.00%
0.00%
31.88%
62.89%
67.46%
84.84%
90.18%
98.21%
99.18%
100.00%
100.00%
100.00%
0.001
0.005
0.01
0.05
0.1
0.5
1
30
60
More
Frequency
Cumulative %

	0-20	20-200	200-2000	2000-20000	20000-200000	More
Frequency	70191	225407	7926	556	102	7
Cumulative %	23.07%	97.18%	99.78%	99.96%	100.00%	100.00%


symbolic execution time
2%
0.82%
200 secs
94.93%
2.25%
200 secs
solver time



	0-10	10-20	20-30	30-40	40-50	50-60	60-70	70-80	80-90	90-100	More
Frequency	153158	32736	21589	16806	16740	10604	14006	8116	4492	20106	0
Cumulative %	51.33%	62.31%	69.54%	75.18%	80.79%	84.34%	89.04%	91.76%	93.26%	100.00%	100.00%


20000
18000
16000
14000
12000
10000
8000
6000
4000
2000
0
1
4
7
10
13
16
19
22
25
28
31
34
37
40
43
46
49
52
55
58
61
64
67
70
73
76
79
82
85
88
91
94
97
100

Reflections

Data invaluable for driving investment priorities

Can’t cover all x86 instructions by hand – look at which ones are used!

Recent: synthesizing circuits from templates (Godefroid & Taly PLDI 2012)

Plus finds configuration errors, compiler changes, etc. impossible otherwise

Data can reveal test programs have special structure

Scaling too long traces needs careful attention to representation

Sometimes run out of memory on 4 GB machine with large programs

Even incomplete, unsound analysis useful because whole-program

SAGE finds bugs missed by all other methods

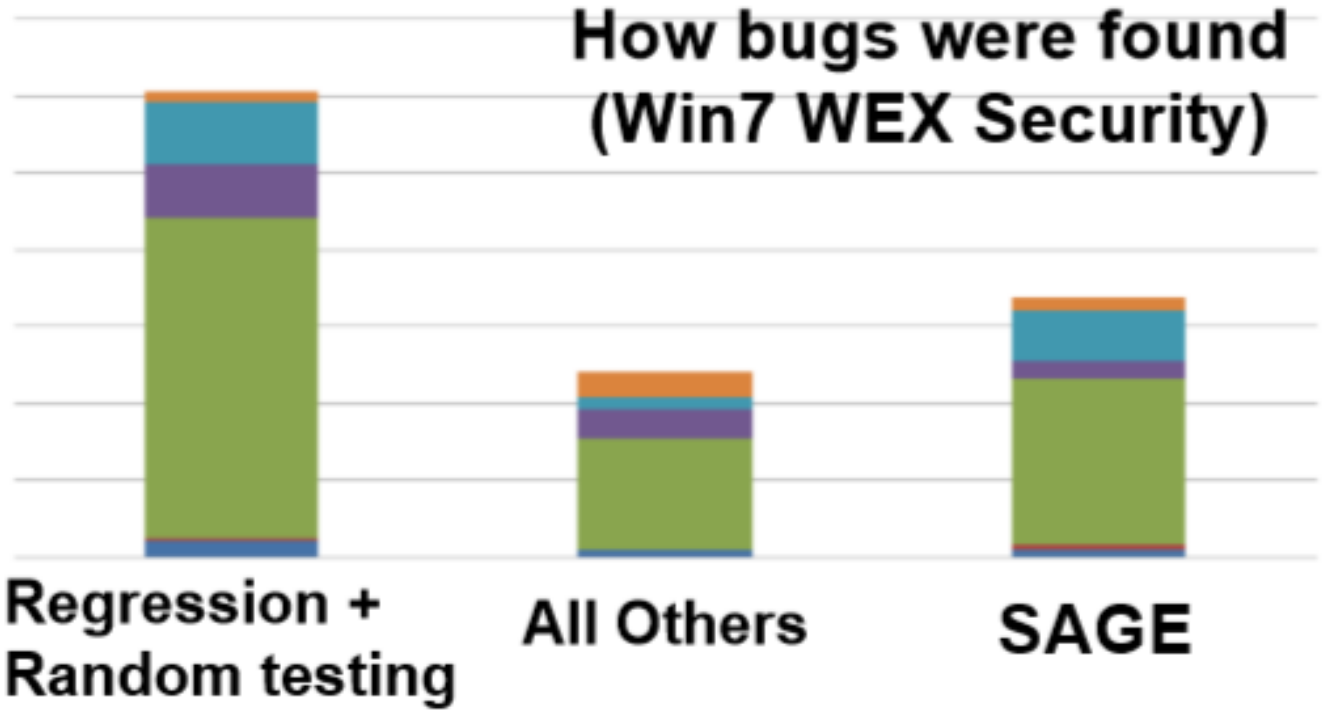
Supporting users & partners super important, a lot of work!

Payoff

3.4 billion constraints queried June 2010 – November 2012

Millions of test cases generated

Run daily on Office, Windows



Thank you!

Questions? dmolnar@icrosoft.com