Whitebox Fuzzing

David Molnar

Microsoft Research

Problem: Security Bugs in File Parsers

Ongoing challenge for Microsoft ecosystem

Hundreds of file formats are supported in Windows, Office, et al.

Many written in C/C++

Programming errors → security bugs!

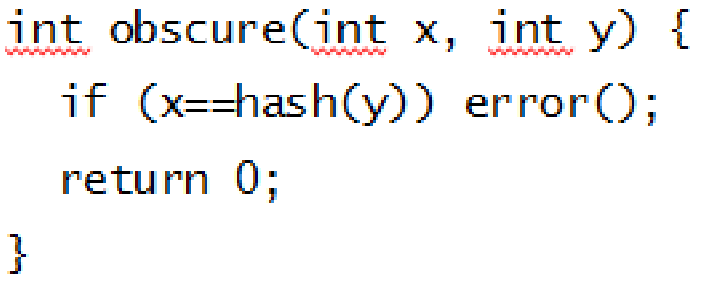
Every security patch costs Microsoft alone **one million dollars.**

To catch “million dollar bugs,”

every team at Microsoft employs random “fuzz testing”

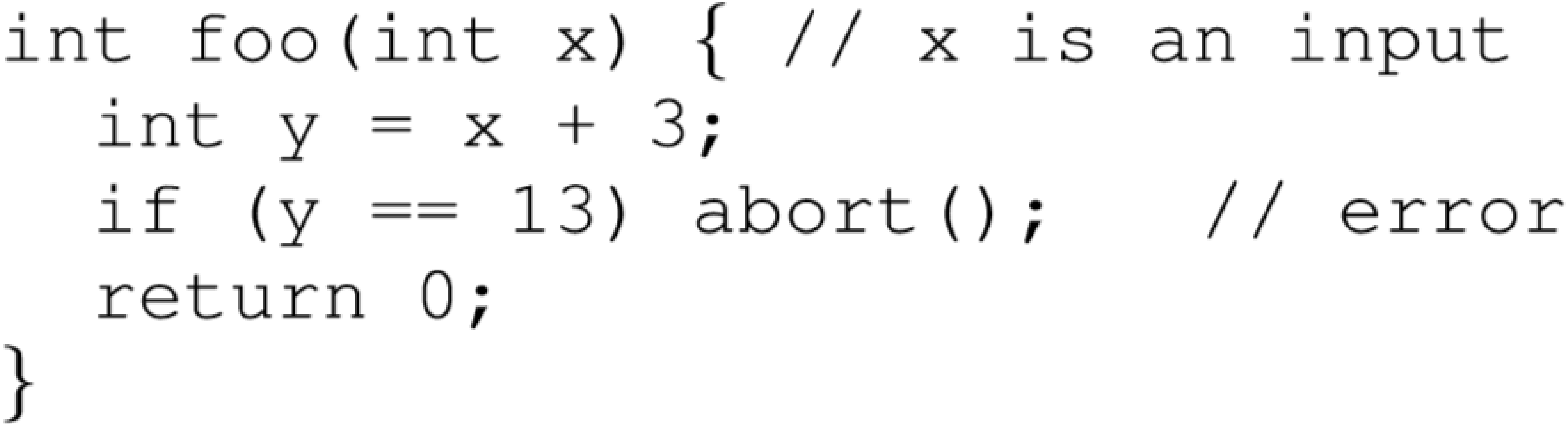
Fuzzing finds 1000s of bugs!

Traditional random fuzz testing **can’t catch this bug**:



```
int obscure(int x, int y) {
  if (x==hash(y)) error();
  return 0;
}
```

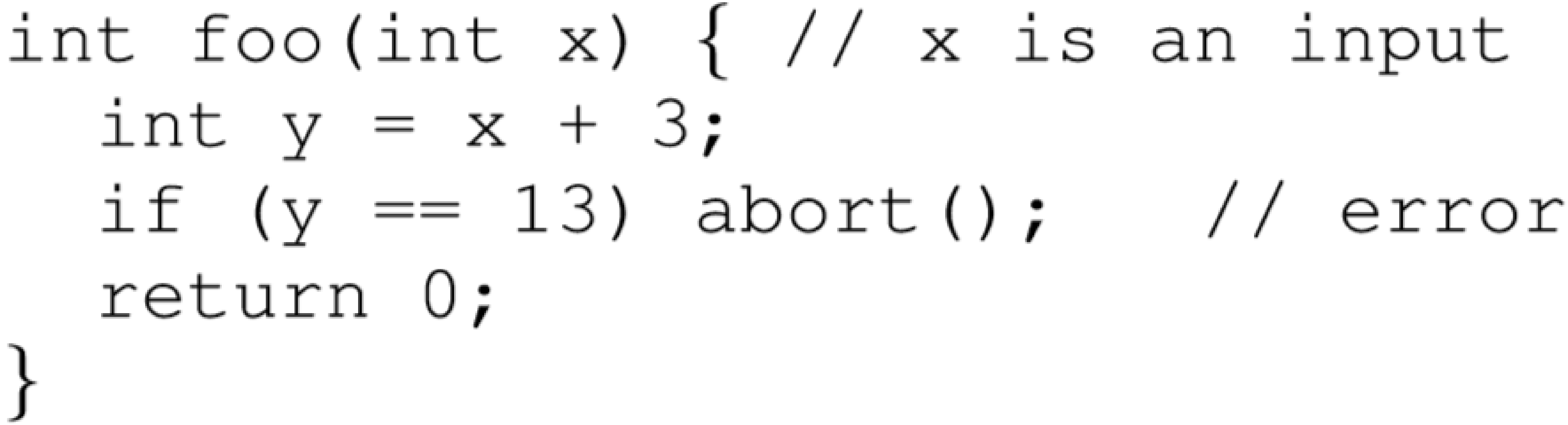
```
int foo(int x) { // x is an input
  int y = x + 3;
  if (y == 13) abort();    // error
  return 0;
}
```



```
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

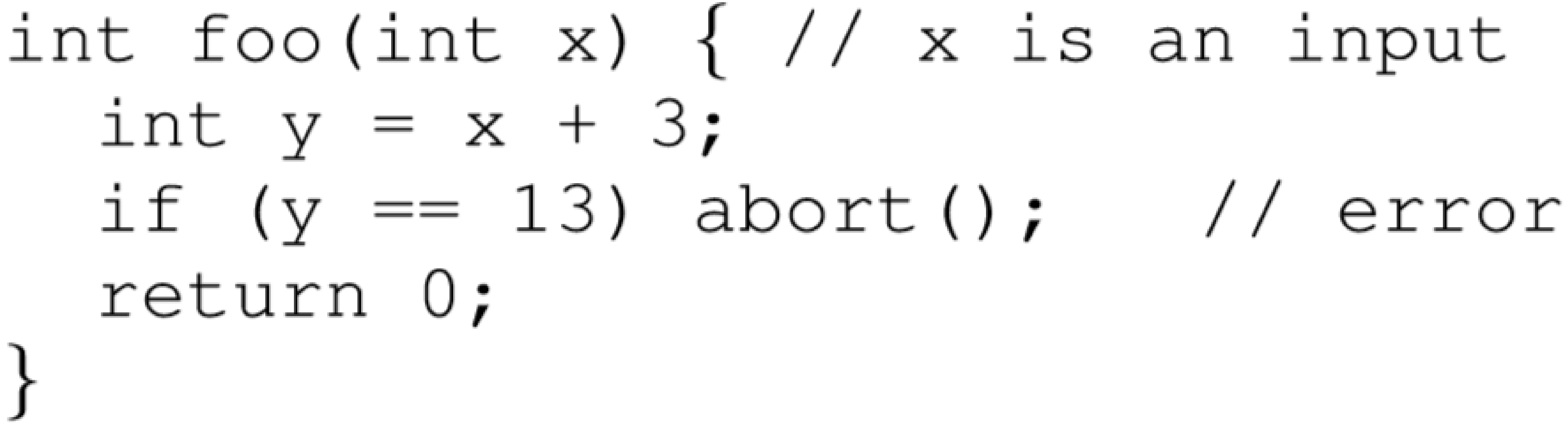
“Fuzz testing” Widely used, remarkably effective!



```
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



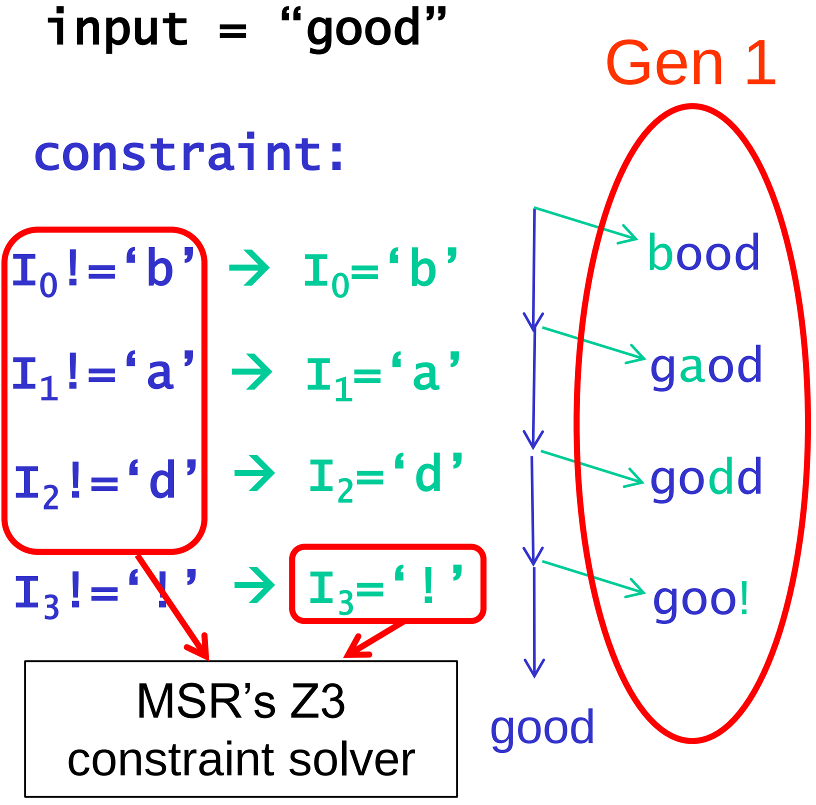
```
int foo(int x) { // x is an input
   int y = x + 3;
   if (y == 13) abort();    // error
   return 0;
}
```

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”

How SAGE Works

```
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 >= 4) crash();
}
```



input = "good"

Path constraint:

I_0!='b' → I_0='b'

I_1!='a' → I_1='a'

I_2!='d' → I_2='d'

I_3!='!' → I_3='!'

MSR's Z3
constraint solver

Gen 1

bood
gaod
godd
goo!

good

Create new constraints to cover new paths

Solve new constraints → new inputs

How SAGE Works

```
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 >= 4) crash();
}
```

input = “bad!”

Path constraint:

$I_0 != 'b'$ → $I_0 = 'b'$

$I_1 != 'a'$ → $I_1 = 'a'$

$I_2 != 'd'$ → $I_2 = 'd'$

$I_3 != '!'$ → $I_3 = '!'$

Gen 1 Gen 2 Gen 3 Gen 4

…

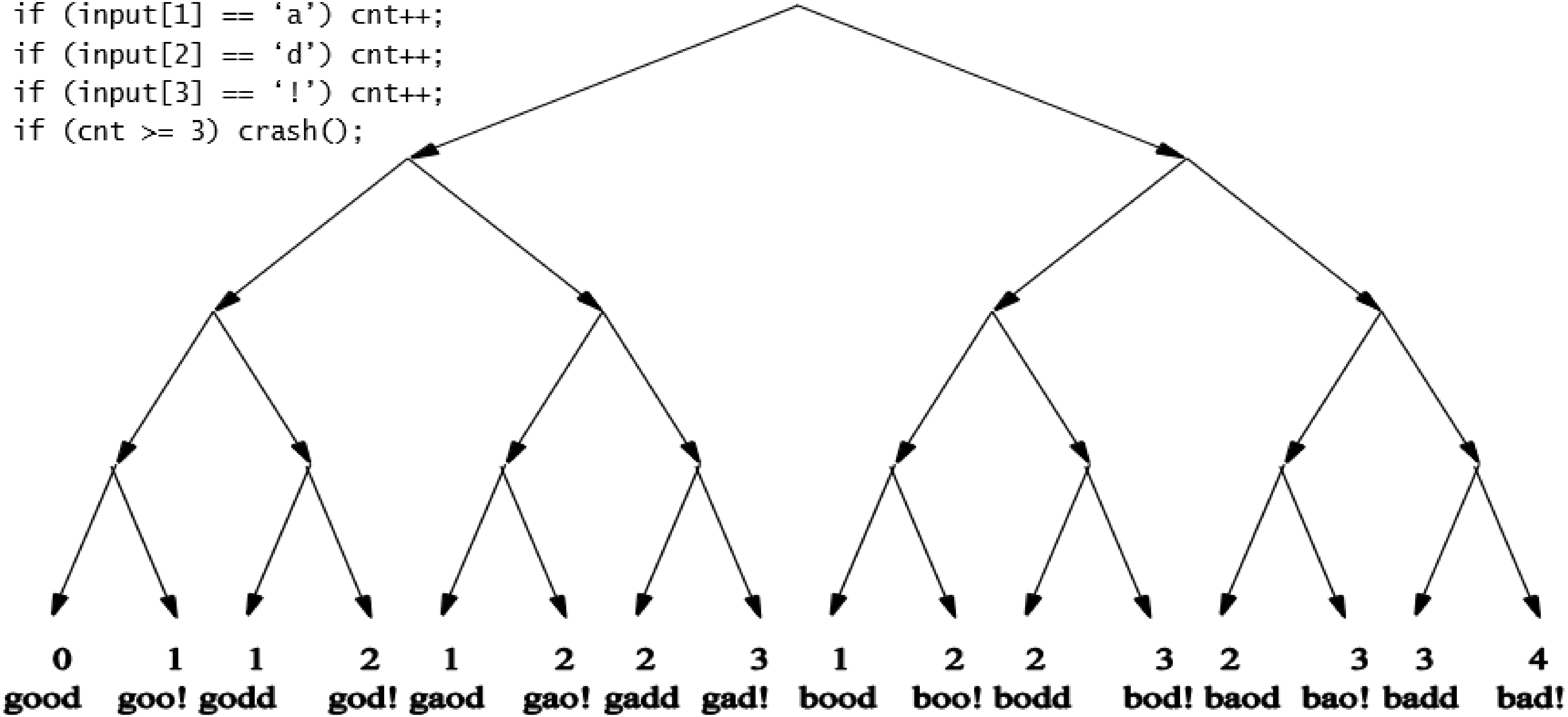
bad!

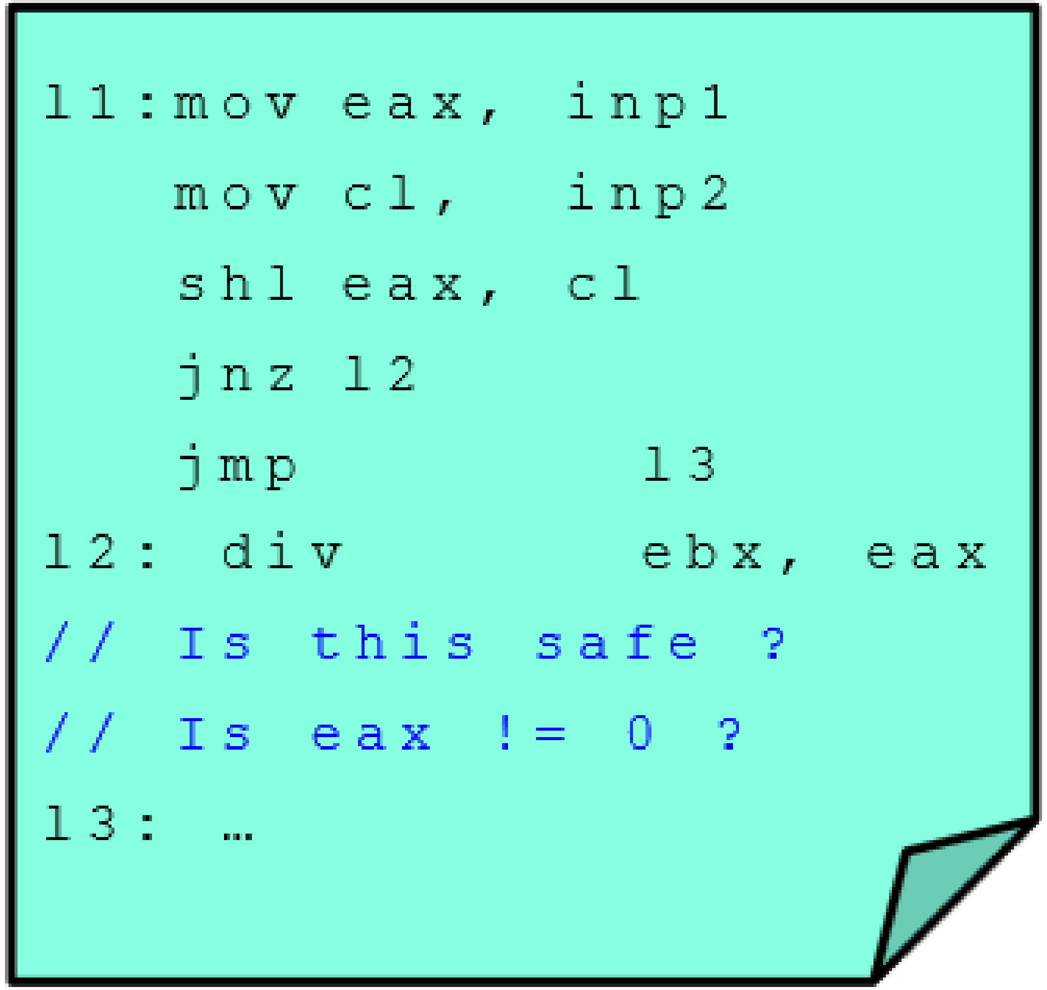
SAGE finds the crash!

Create new constraints to cover new paths

Solve new constraints → new inputs

```
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();
}
```





```
l1:mov eax, inp1
   mov cl,  inp2
   shl eax, cl
   jnz l2
   jmp       l3
l2: div      ebx, eax
// Is this safe ?
// Is eax != 0 ?
l3: …
```

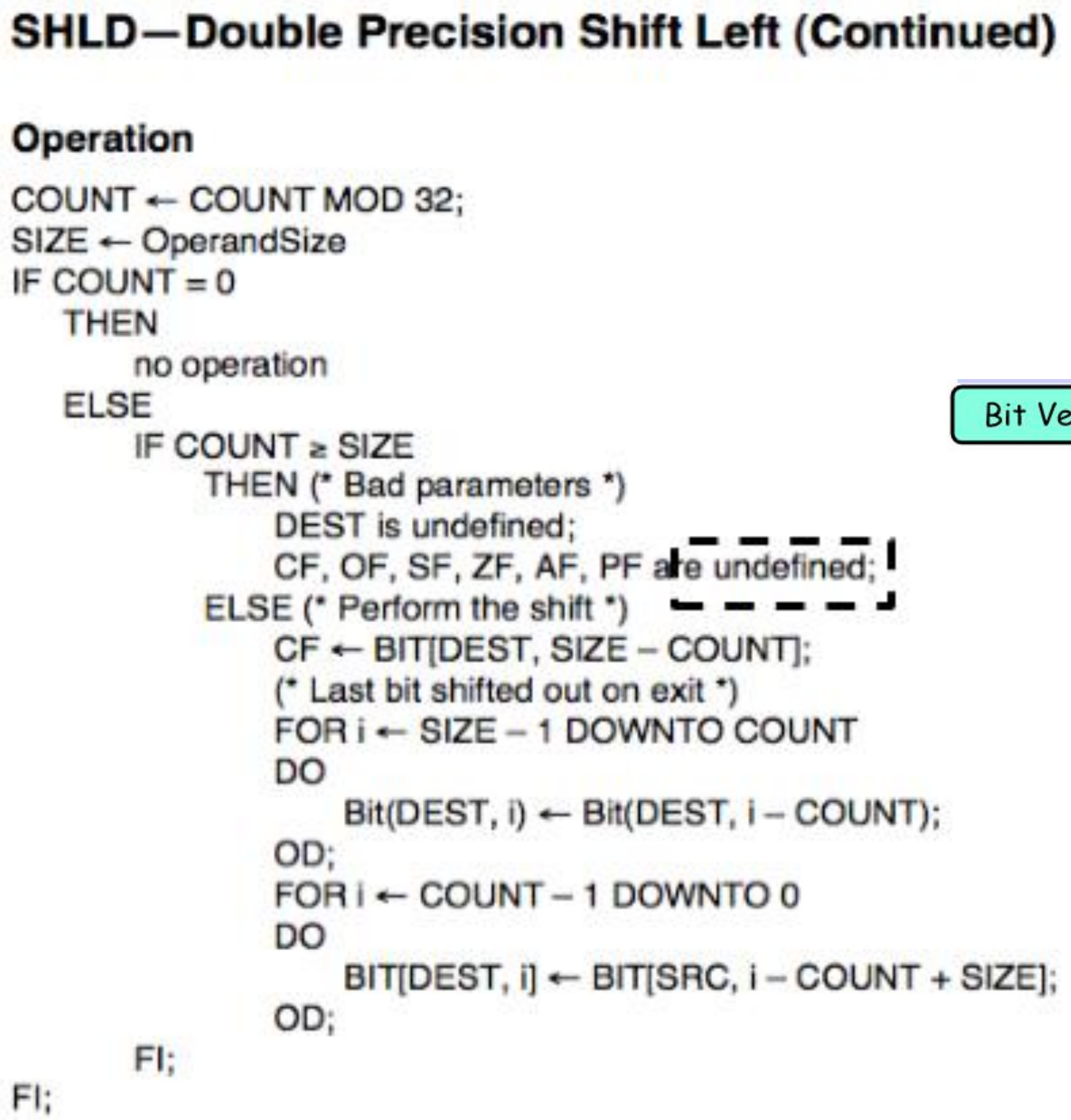
Work with x86 **binary code** on Windows
Leverage full-instruction-trace recording

Pros:

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

Cons:

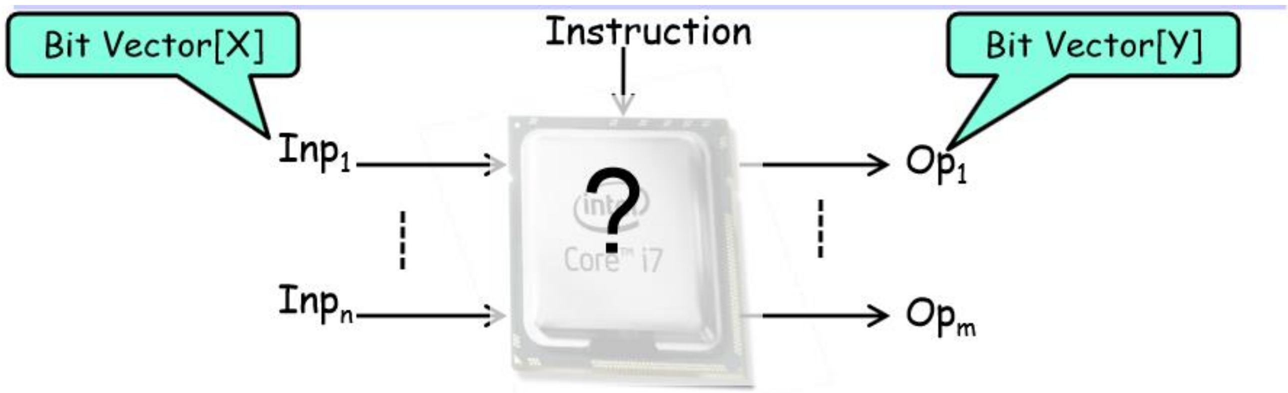
- Lose programmer’s intent (e.g. types)
- Hard to “see” string manipulation, memory object graph manipulation, etc.



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;
```

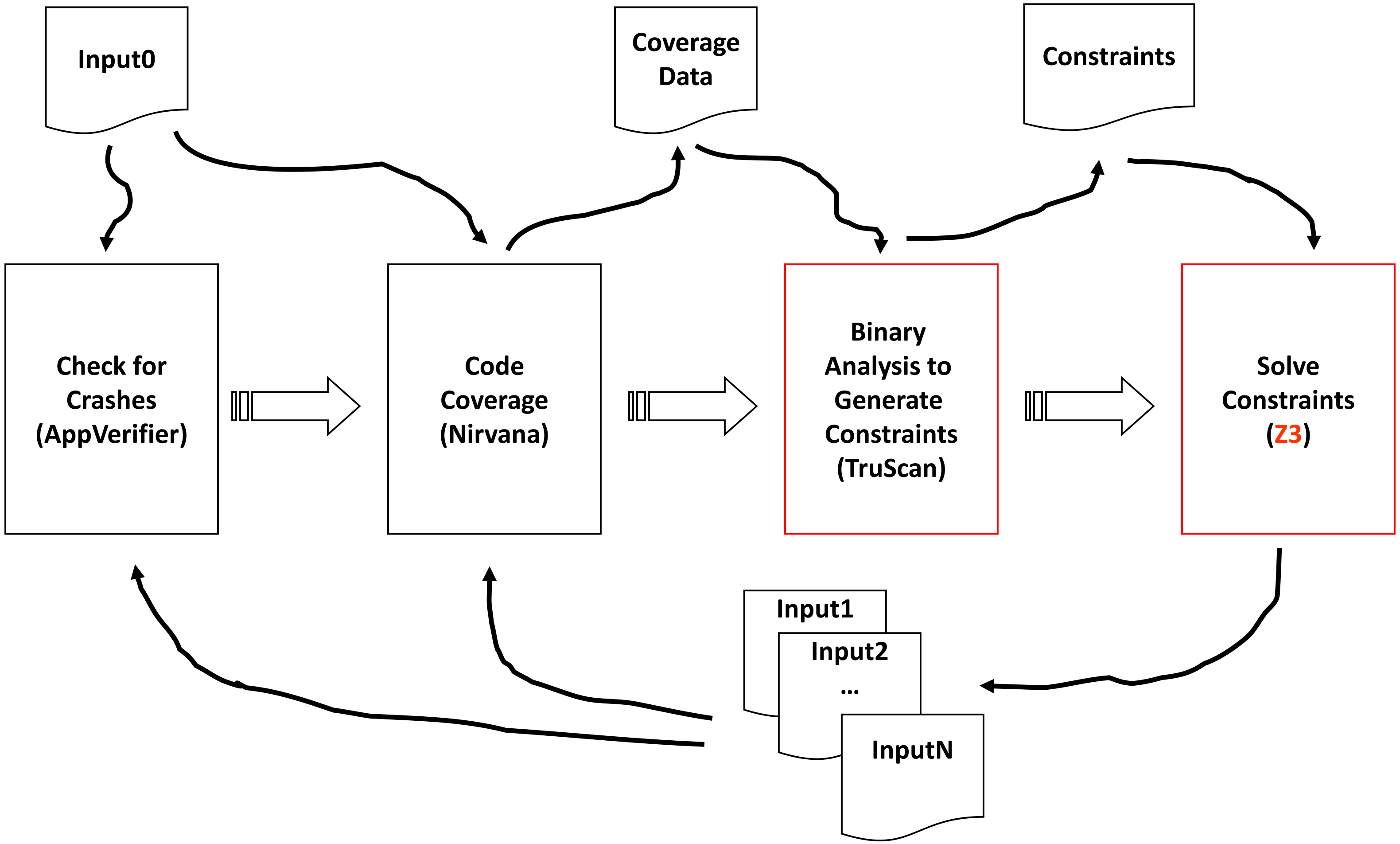


Hand-written models (so far)
Uses Z3 support for non-linear operations

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

SAGE: A Whitebox Fuzzing Tool



Input0

Coverage Data

Constraints

Check for Crashes (AppVerifier)

Code Coverage (Nirvana)

Binary Analysis to Generate Constraints (TruScan)

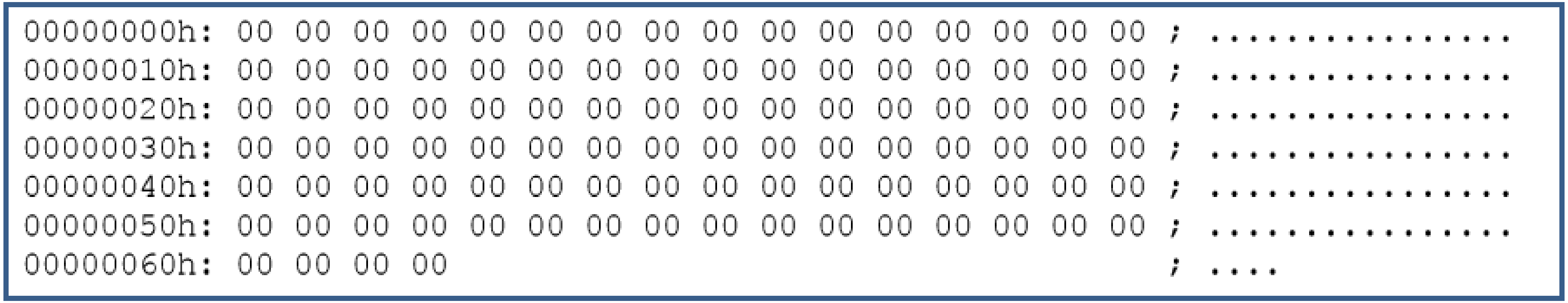
Solve Constraints (Z3)

Input1

Input2

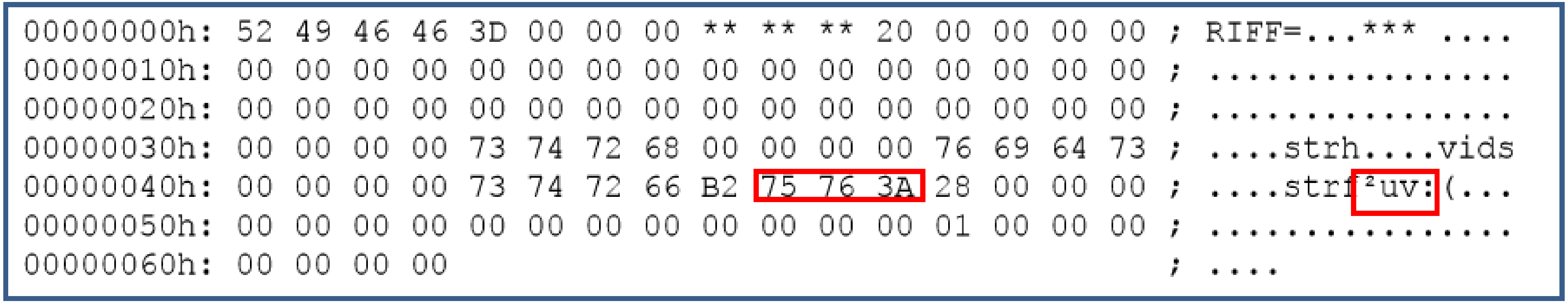
…

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!

Research Behind SAGE

- Precision in symbolic execution: PLDI’05, PLDI’11
- Scaling to billions of instructions: NDSS’08
- Checking many properties together: EMSOFT’08
- Grammars for complex input formats: PLDI’08
- Strategies for dealing with path explosion: POPL’07, TACAS’08, POPL’10, SAS’11
- Reasoning precisely about pointers: ISSTA’09
- Floating-point instructions: ISSTA’10
- Input-dependent loops: ISSTA’11

\+ research on constraint solvers (Z3)

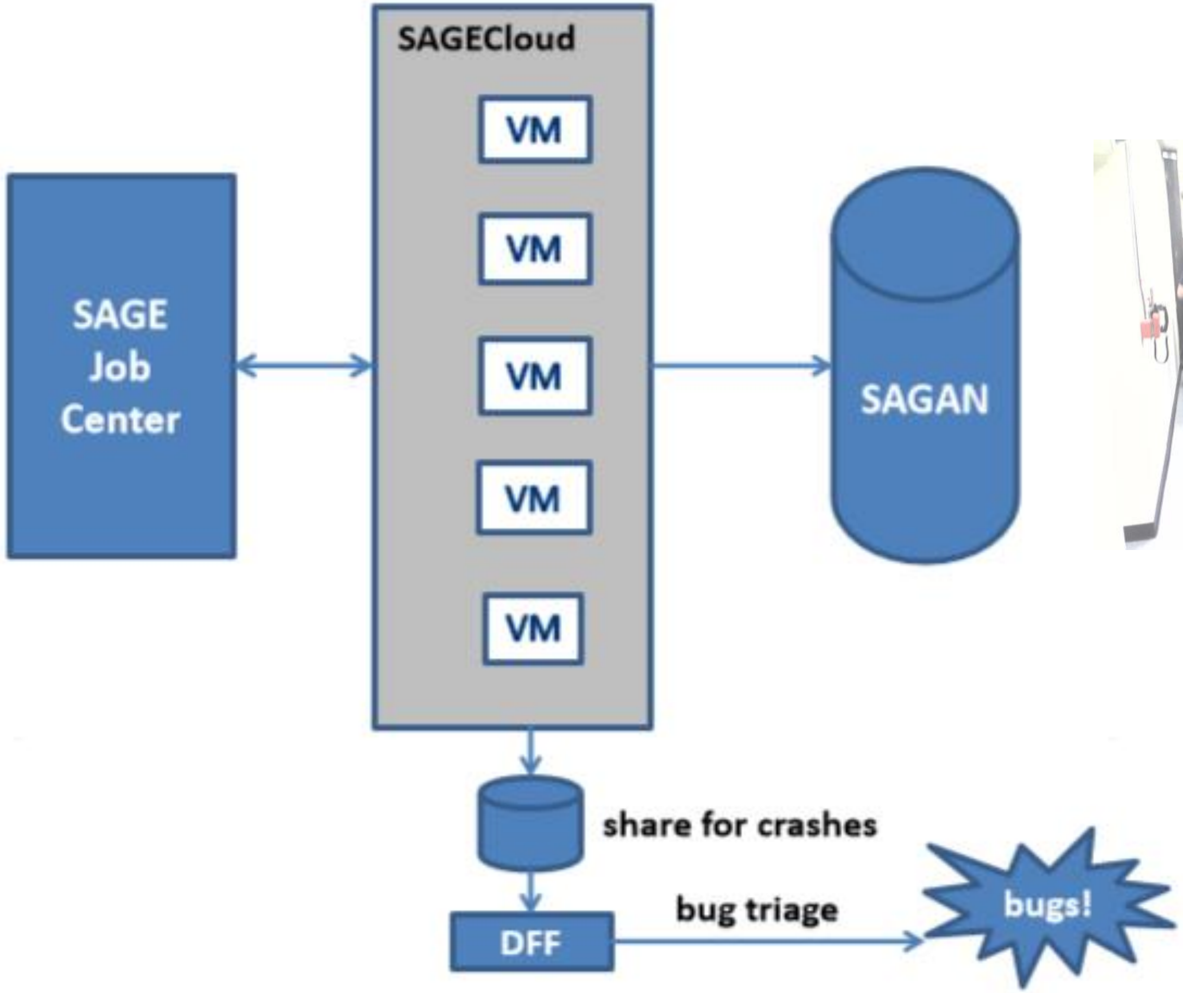
Challenges: from Research to Production

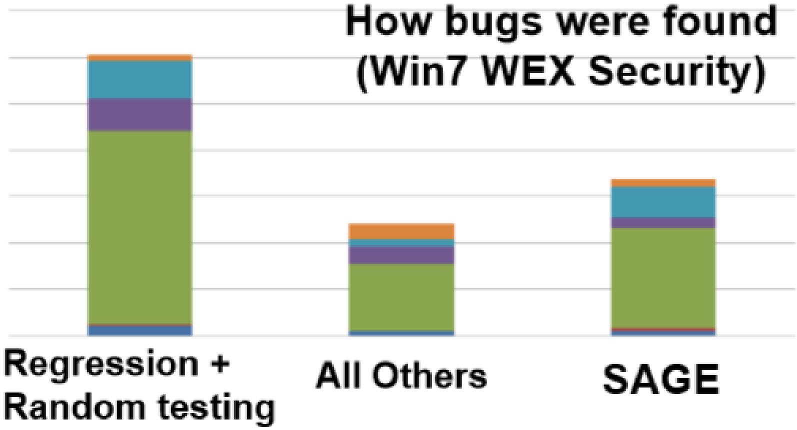
1) Symbolic execution on long traces
2) Fast constraint generation and solving
3) Months-long searches
4) Hundreds of test drivers & file formats
5) Fault-tolerance

A Single Symbolic Execution of an Office App

# of instructions executed	1.45 billion
# instructions after reading from file	928 million
# constraints in path constraint	25,958
# constraints dropped due to optimizations	438,123
# of satisfiable constraints → new tests	2,980
# of unsatisfiable constraints	22,978
# of constraint solver timeouts (> 5 seconds)	0
Symbolic execution time	**45 minutes 45 seconds**
Constraint solving time	**15 minutes 53 seconds**

SAGAN and SAGECloud for Telemetry and Management





Hundreds of machines / VMs on average
Hundreds of applications on thousands of "seed files"
Over **500 machine-years** of whitebox fuzzing!

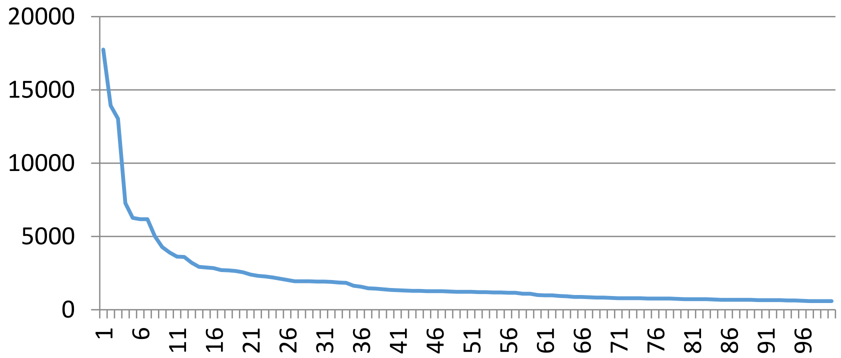
Challenges: From Research to Production

1) Symbolic execution on long traces
   SAGAN telemetry points out imprecision
2) Fast constraint generation and solving
   SAGAN sends back long-running constraints
3) Months-long searches
   JobCenter monitors progress of search
4) Hundreds of test drivers & file formats
   JobCenter provisions apps and configurations in SAGECloud
5) Fault-tolerance
   SAGAN telemetry enables quick response

Feedback From Telemetry At Scale

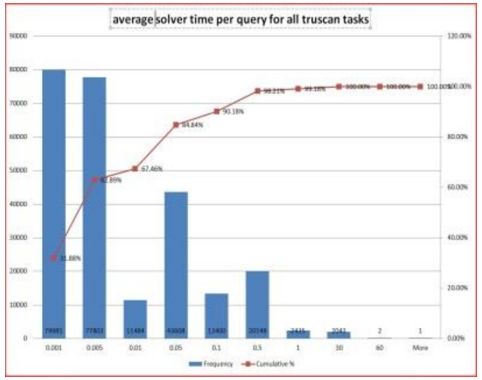
Any test anywhere helps every test everywhere!

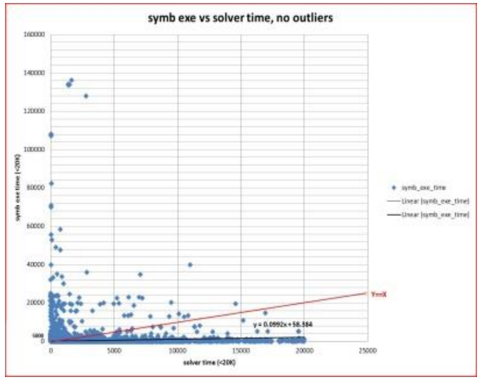
How much **sharing** between symbolic execution of **different** programs run on Windows?



Most common branch appears **17761** times out of **290430** symbolic executions. Motivates **symbolic summaries built up over time**.

How does the Z3 solver perform on **constraints** arising from **real code**?





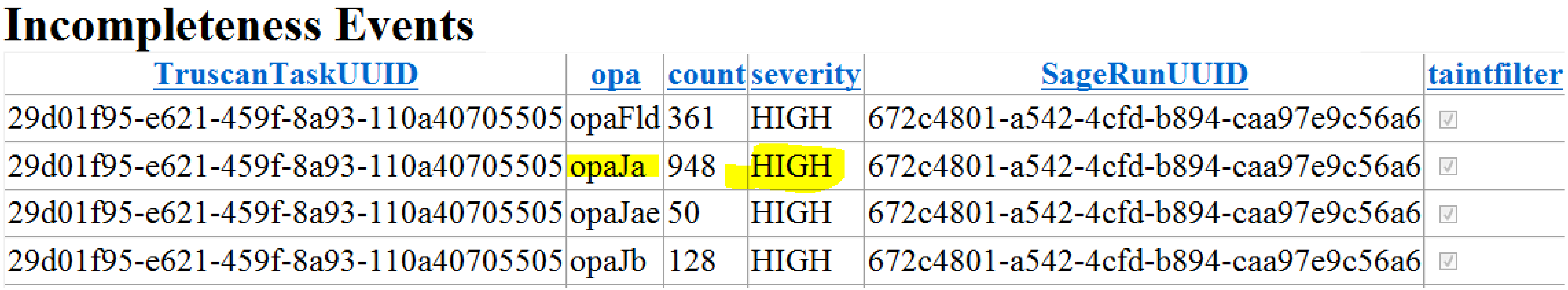
90.18% of Z3 queries solved in **0.1 seconds or less**. Solving time still dominates! Tells us where to focus Z3.

Leverage data collection to create **virtuous cycle** of improvement!

Answer questions and pursue directions **impractical without scale**.

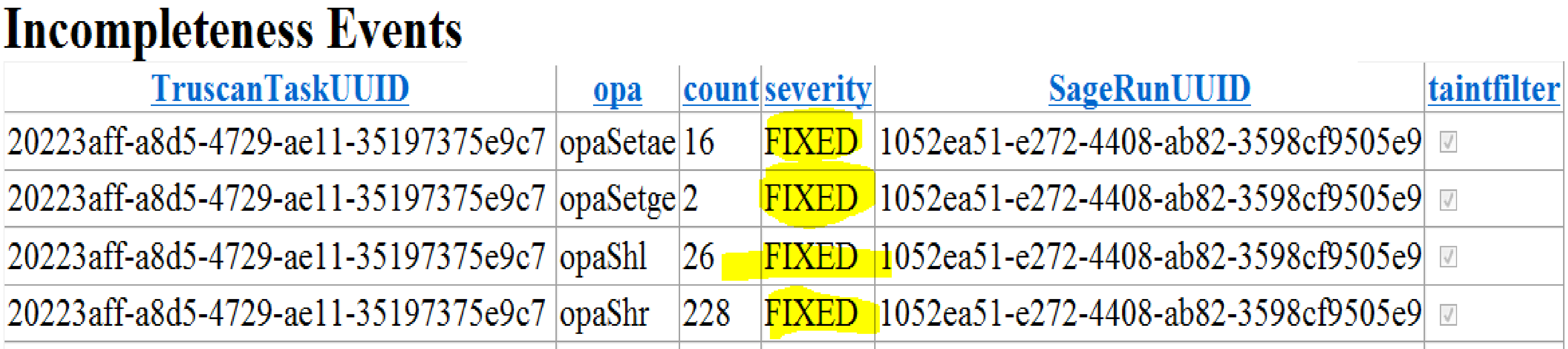
Key Analyses Enabled by Data

Imprecision in Symbolic Execution



Incompleteness Events

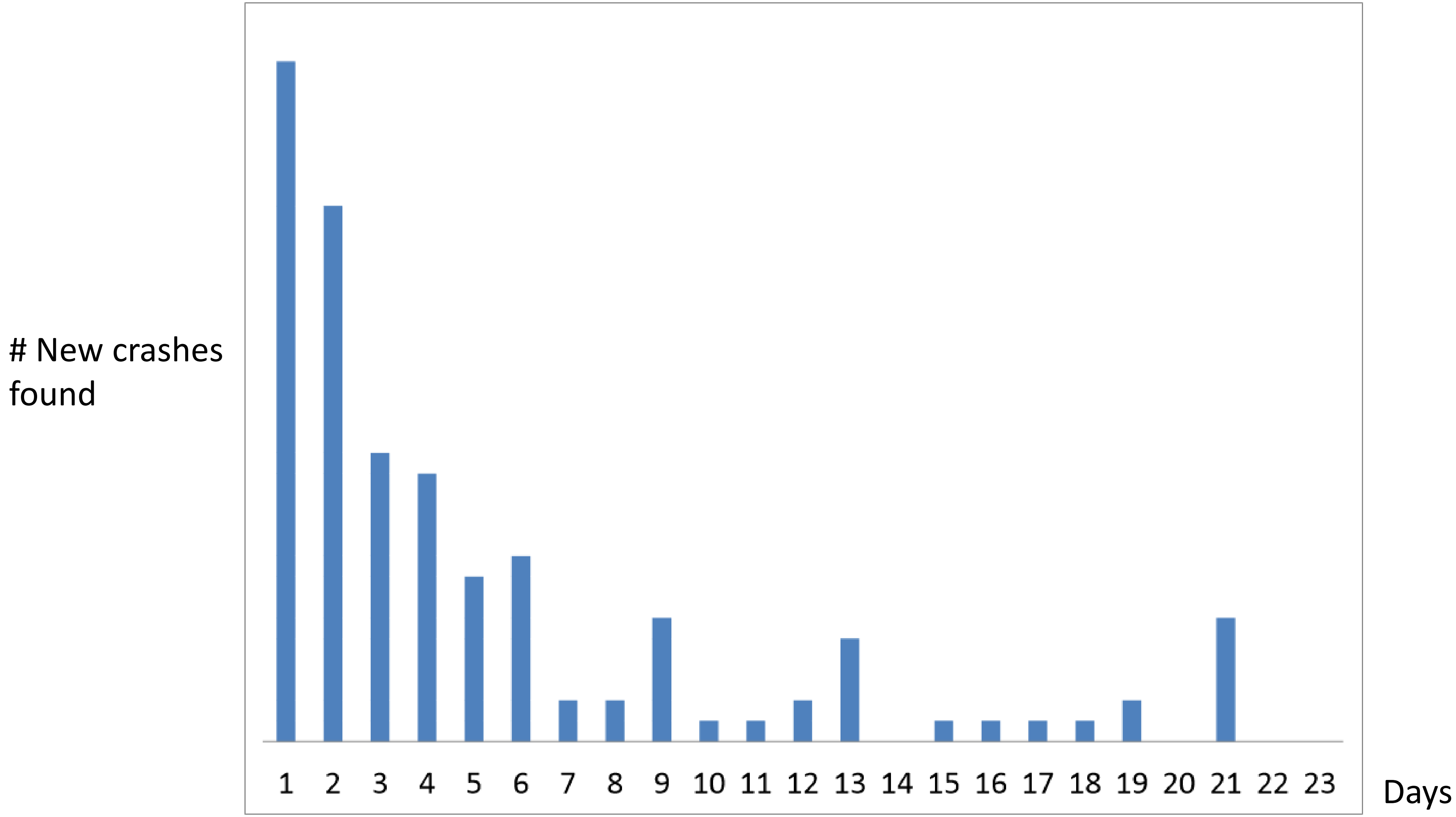
TruscanTaskUUID	opa	count	severity	SageRunUUID	taintfilter
29d01f95-e621-459f-8a93-110a40705505	opaFld	361	HIGH	672c4801-a542-4cfd-b894-caa97e9c56a6	☑
29d01f95-e621-459f-8a93-110a40705505	opaJa	948	HIGH	672c4801-a542-4cfd-b894-caa97e9c56a6	☑
29d01f95-e621-459f-8a93-110a40705505	opaJae	50	HIGH	672c4801-a542-4cfd-b894-caa97e9c56a6	☑
29d01f95-e621-459f-8a93-110a40705505	opaJb	128	HIGH	672c4801-a542-4cfd-b894-caa97e9c56a6	☑



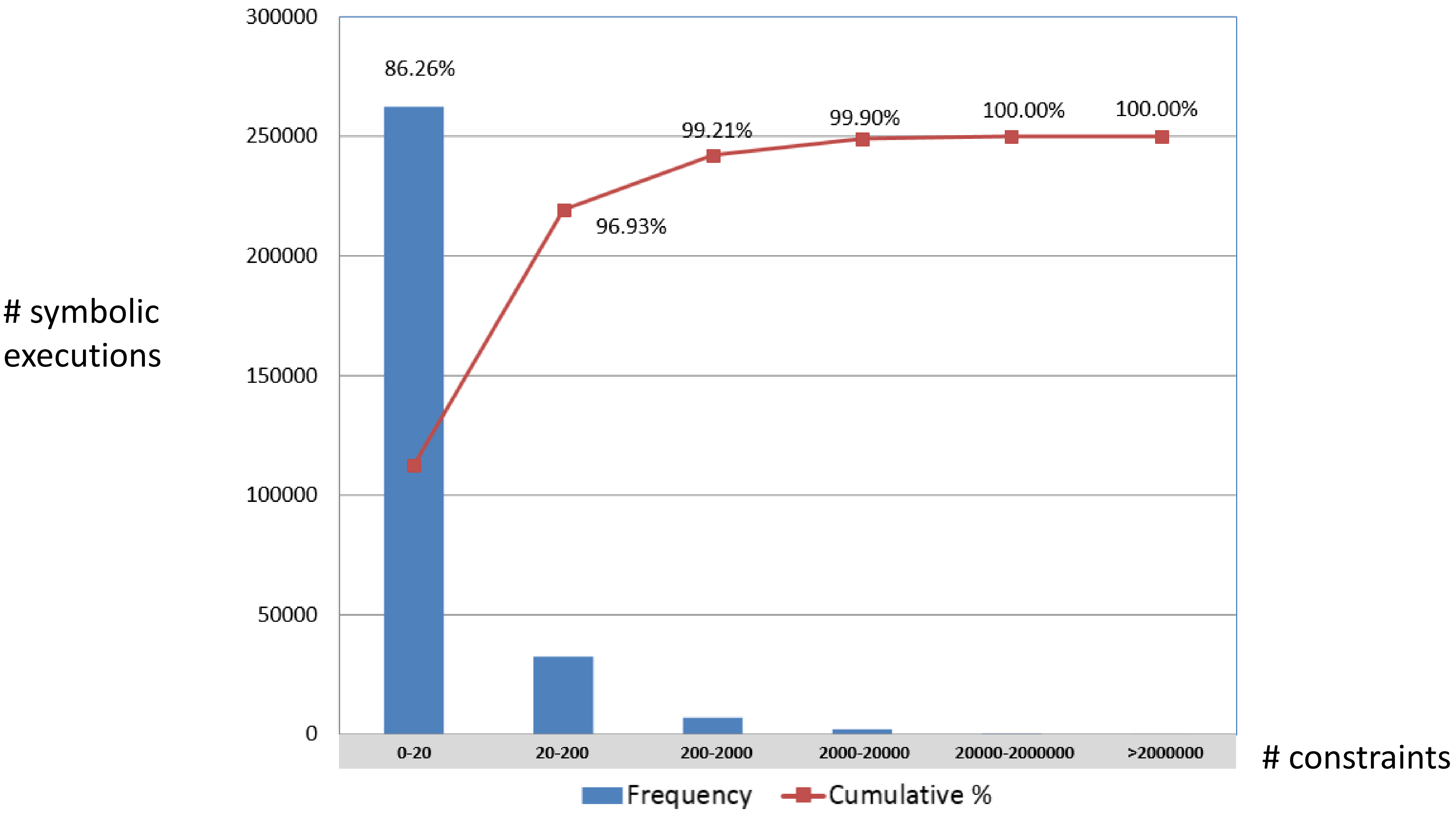
Incompleteness Events

TruscanTaskUUID	opa	count	severity	SageRunUUID	taintfilter
20223aff-a8d5-4729-ae11-35197375e9c7	opaSetae	16	FIXED	1052ea51-e272-4408-ab82-3598cf9505e9	☑
20223aff-a8d5-4729-ae11-35197375e9c7	opaSetge	2	FIXED	1052ea51-e272-4408-ab82-3598cf9505e9	☑
20223aff-a8d5-4729-ae11-35197375e9c7	opaShl	26	FIXED	1052ea51-e272-4408-ab82-3598cf9505e9	☑
20223aff-a8d5-4729-ae11-35197375e9c7	opaShr	228	FIXED	1052ea51-e272-4408-ab82-3598cf9505e9	☑

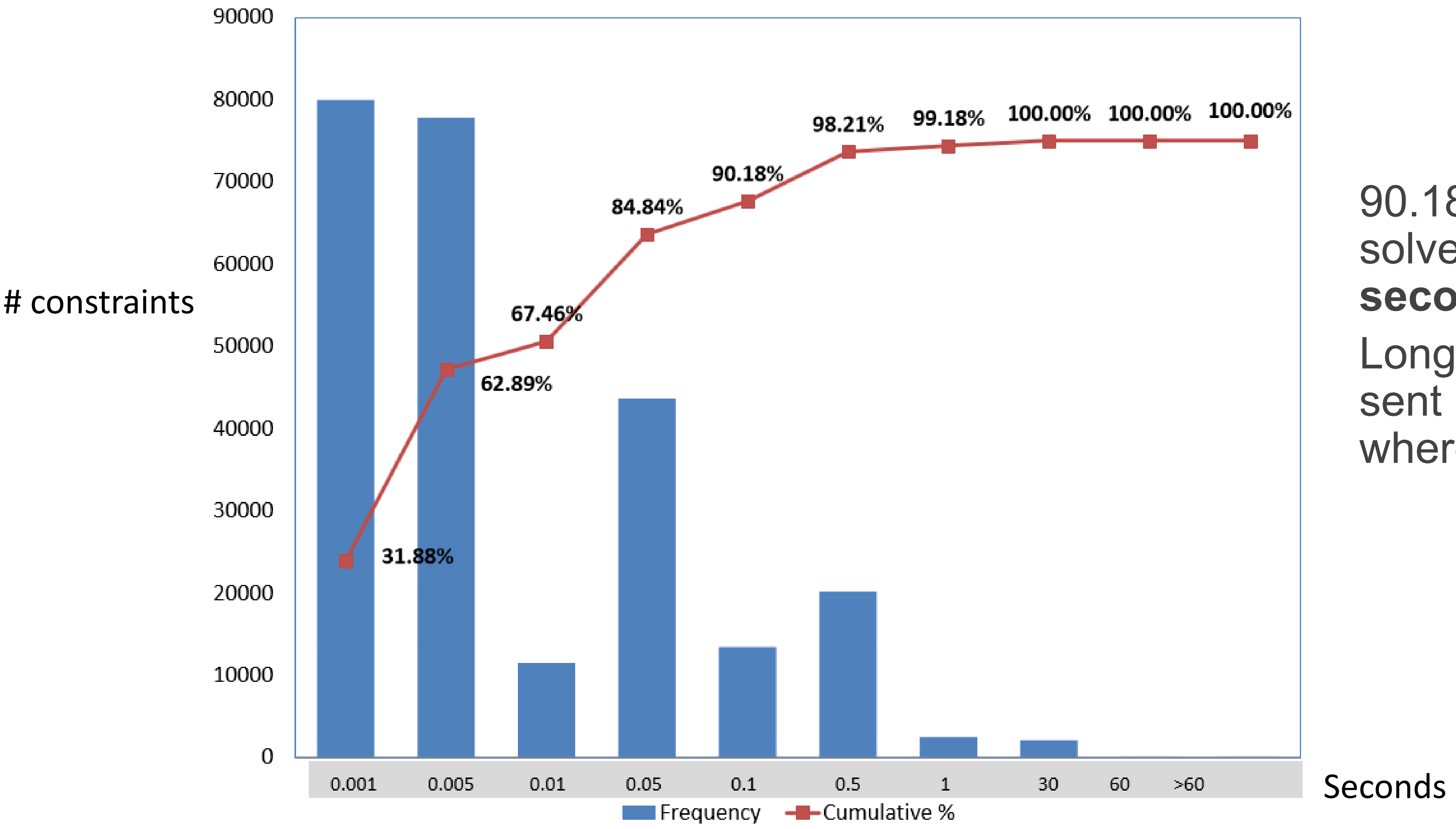
Distribution of crashes in the search



Constraints generated by symbolic execution



Time to solve constraints



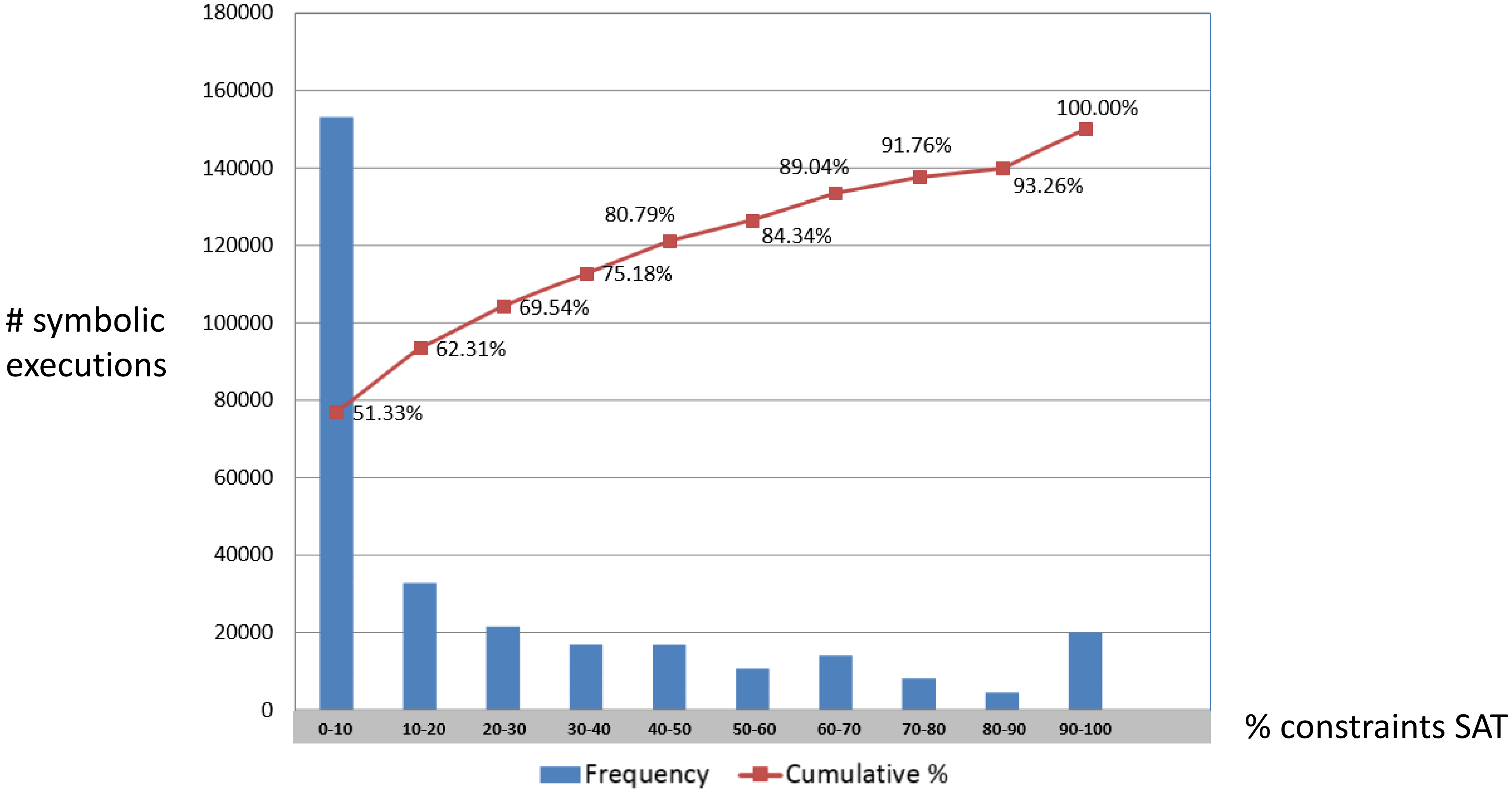
90.18% of Z3 queries solved in **0.1 seconds or less**.

Long-running queries sent back, tell us where to focus Z3.

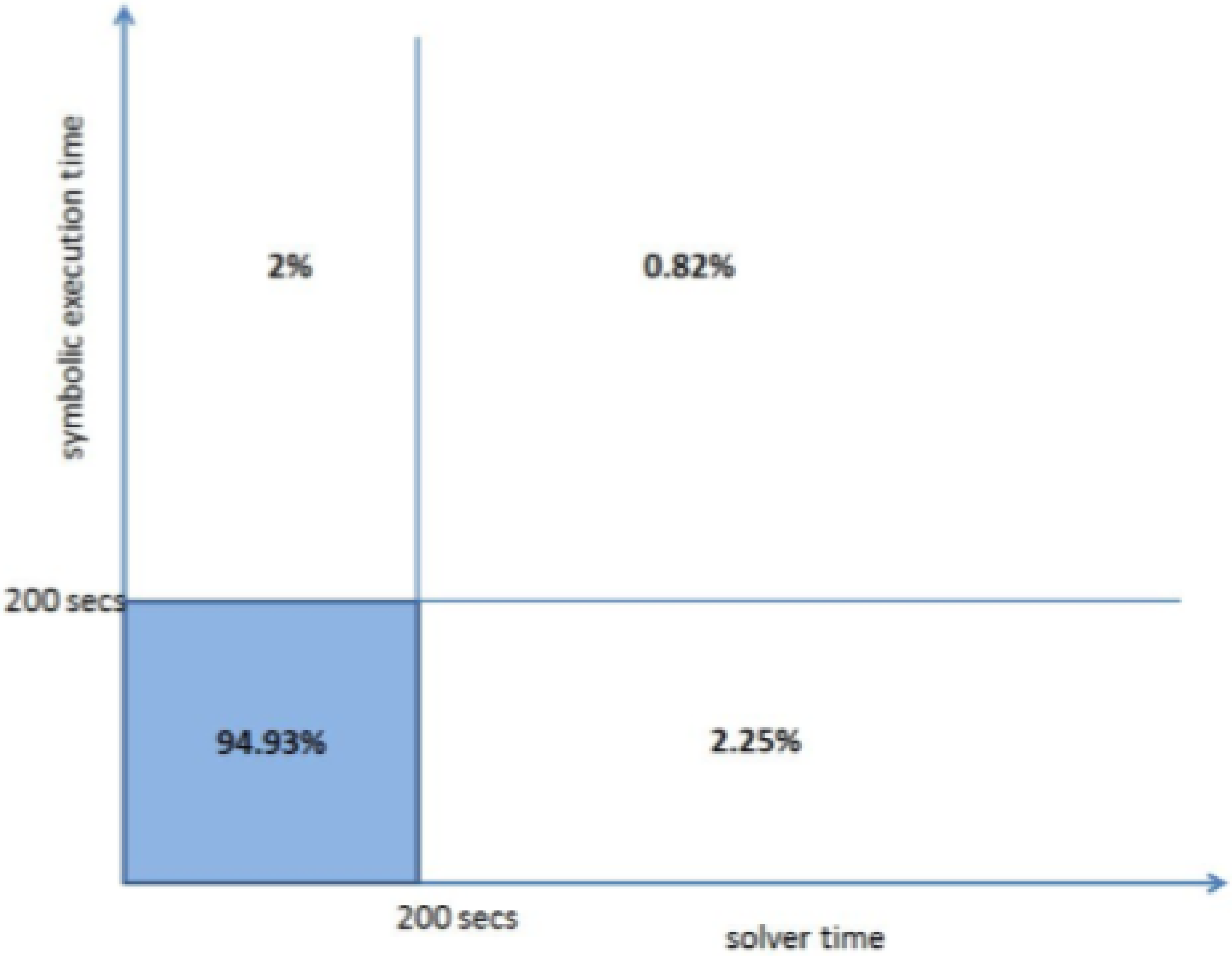
Optimizations In Constraint Generation

- Sound
  - Common subexpression elimination on every new constraint
    - Crucial for memory usage
  - “Related Constraint Optimization”
- Unsound
  - Constraint subsumption
    - Syntactic check for implication, take strongest constraint
  - Drop constraints at same instruction pointer after threshold

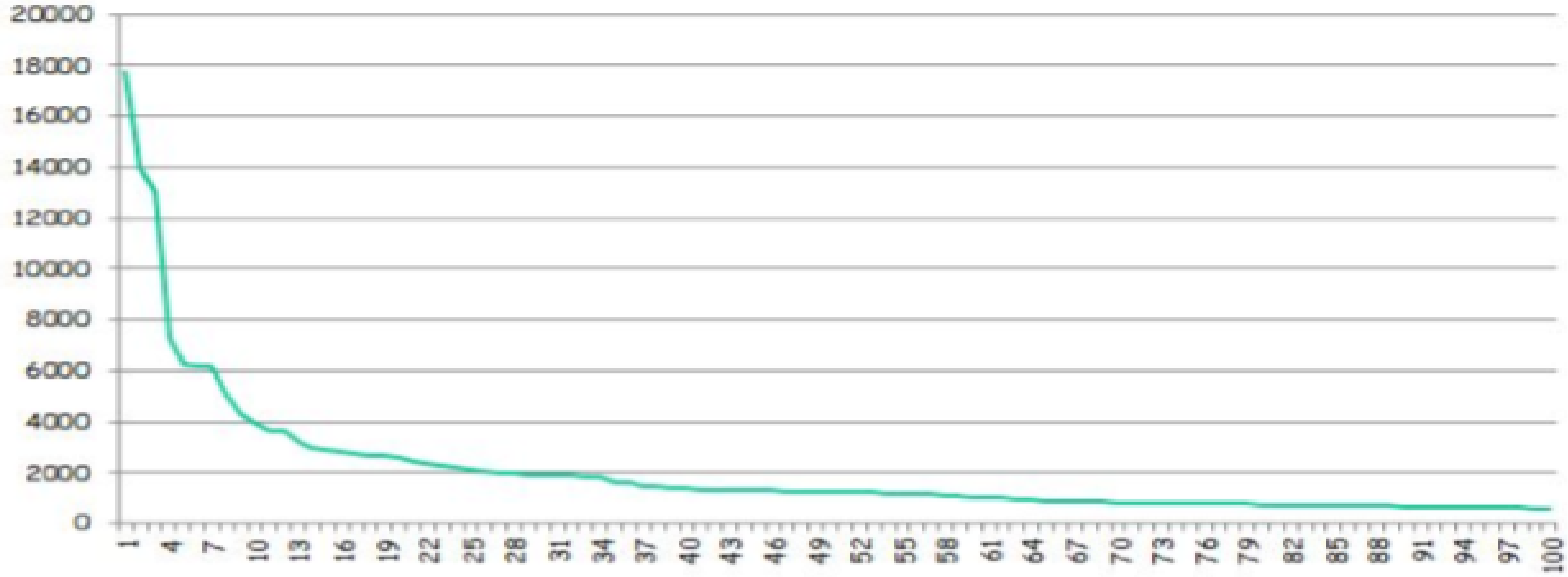
Ratio between SAT and UNSAT constraints



Long-running tasks can be pruned!



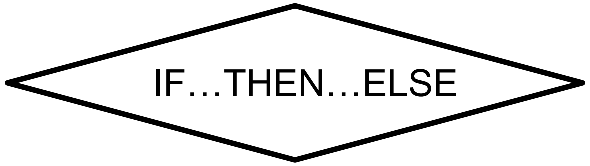
Sharing Between Symbolic Executions



Sampled runs on Windows, many different file-reading applications
Max frequency **17761**, min frequency **592**
Total of **290430** branches flipped, **3360** distinct branches

Summaries Leverage Sharing

- Redundancy in searches
  - Redundancy in paths



  - Redundancy in different versions of same application
  - Redundancy across applications
    - How many times does Excel/Word/PPT/… call mso.dll ?
- Summaries (POPL 2007): avoid re-doing this unnecessary work
- SAGAN data shows redundancy exists in practice

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 to 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!

Impact In Numbers

- 100s of apps, 100s of bugs fixed
- 3.5+ billion constraints
  - Largest computational usage ever for any SMT solver
- 500+ machine-years

SAGE-like tools outside Microsoft

- KLEE http://klee.github.io/klee/
- FuzzGrind http://esec-lab.sogeti.com/pages/Fuzzgrind
- SmartFuzz

Thanks to all SAGE contributors!

- MSR: Ella Bounimova, Patrice Godefroid, David Molnar
  (+ our managers for their support! ☺)
- CSE: Michael Levin, Chris Marsh, Lei Fang, Stuart de Jong,…
- Interns : Dennis Jeffries (06), David Molnar (07), Adam Kiezun (07), Bassem Elkarablieh (08), Marius Nita (08) , Cindy Rubio-Gonzalez (08,09), Johannes Kinder (09), Daniel Luchaup (10), Mehdi Bouaziz (11), Ankur Taly (11), Gena Pekhimenko (12), Maria Christakis (13),…
- Z3 (MSR): Nikolaj Bjorner, Leonardo de Moura,…
- Windows: Nick Bartmon, Eric Douglas, Dustin Duran, Elmar Langholz , Isaac Sheldon, Dave Weston,…
  - Win8 TruScan support: Evan Tice, David Grant,…
- Office: Tom Gallagher, Eric Jarvi, Octavian Timofte, Mike Caldwell…
- MSEC: Dan Margolis, Matt Miller, Lars Opstad, Jason Shirk, Dazhi Zhang…
- SAGE users all across Microsoft!

Questions? dmolnar@microsoft.com