Finding Bugs

Symbolic execution enumerates paths

- Runs into bugs that trigger whenever path executes
- Assertions, buffer overflows, division by zero, etc., require specific conditions

Error conditions

- Treat assertions as conditions
- Creates explicit error paths

```
assert x != NULL
```



```
if (x == NULL)
    abort();
```

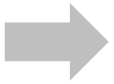
Finding Bugs

Instrument program with properties

- Translate any safety property to reachability

Division by zero

```
y = 100 / x
```



```
assert x != 0
y = 100 / x
```

Buffer overflows

```
a[x] = 10
```



```
assert x >= 0 && x < len(a)
```

Implementation is usually implicit

Dynamic Symbolic Execution

Testing, Quality Assurance, and Maintenance
Winter 2017

Prof. Arie Gurfinkel

based on slides by Prof. Johannes Kinder and others



UNIVERSITY OF
WATERLOO

Problems with Scaling Symbolic Execution

Code that is hard to analyze

Path explosion

- Complex control flow
- Loops
- Procedures

Environment (what are the inputs to the program under test?)

- pointers, data structures, …
- files, data bases, …
- threads, thread schedules, …
- sockets, …

Code that is hard to analyze

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

May be very hard to statically generate values for x and y that satisfy "x==complex(y)" !

Sources of complexity:

- Virtual functions (function pointers)
- Cryptographic functions
- Non-linear integer or floating point arithmetic
- Calls to kernel mode
- …

Directed Automated Random Testing [PLDI 2005]

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

Run 1 :

- start with (random) x=33, y=42
- execute concretely and symbolically:
  if (33 == 567) | if (x == complex(y))
  constraint too complex
  → simplify it: x = 567
- solve: x==567 → solution: x=567
- new test input: x=567, y=42

Run 2 : the other branch is executed
All program paths are now covered !

Also known as concolic execution (concrete + symbolic)
Referred to here as dynamic symbolic execution

Flavors of Symbolic Execution Algorithms

Static symbolic execution

- Simulate execution on program source code
- Computes strongest post-conditions from entry point

Dynamic symbolic execution (DSE)

- Run / interpret the program with concrete state
- Symbolic state computed in parallel (“concolic”)
- Solver generates new concrete state

DSE-Flavors

- EXE-style [Cadar et al. ‘06] vs. DART [Godefroid et al. ‘05]

Many successful tools

- EXE → KLEE (Imperial), SPF (NASA), Cloud9, S2E (EPFL)
- DART → SAGE, PEX (Microsoft), CUTE (UIUC), CREST (Berkeley)

EXE Algortihm

Program state is a tuple (ConcreteState, SymbolicState)

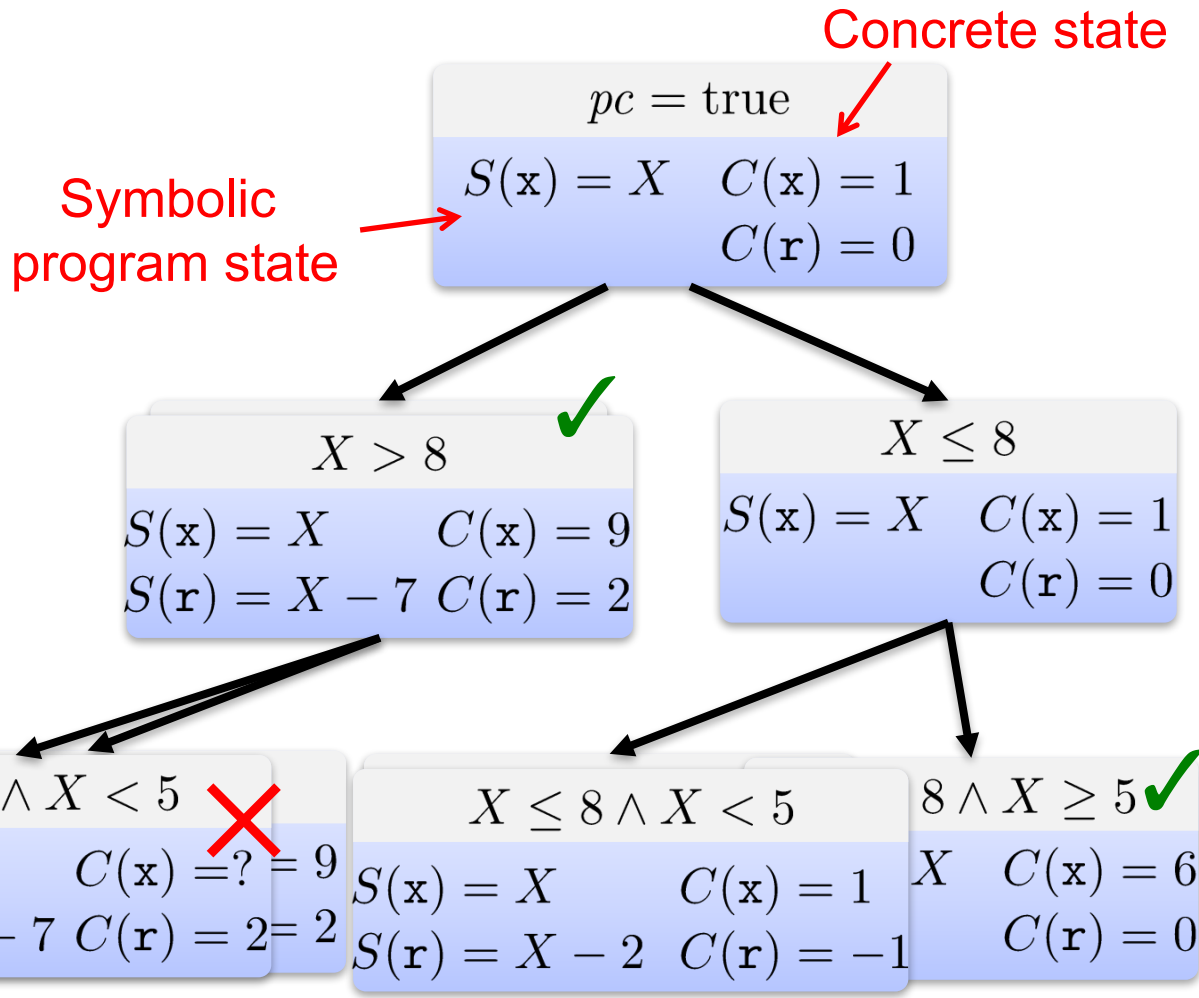
Initially all input variables are symbolic

At each execution step

- update the concrete state by executing a program instruction concretely
- update symbolic state executing symbolically
- if the last instruction was a branch
  - if PC and negation of branch condition is SAT
    - fork execution state
    - compute new concrete to match the new path condition

EXE

```
1  int proc(int x) {
2
3    int r = 0
5
6    if (x > 8) {
7      r = x - 7
8    }
9
10   if (x < 5)
11     r = x -
12   }
13
14   return r;
15 }
```



Satisfying assignments:

X = 9 X = 1 X = 6

Test cases:

proc(9) proc(1) proc(6)

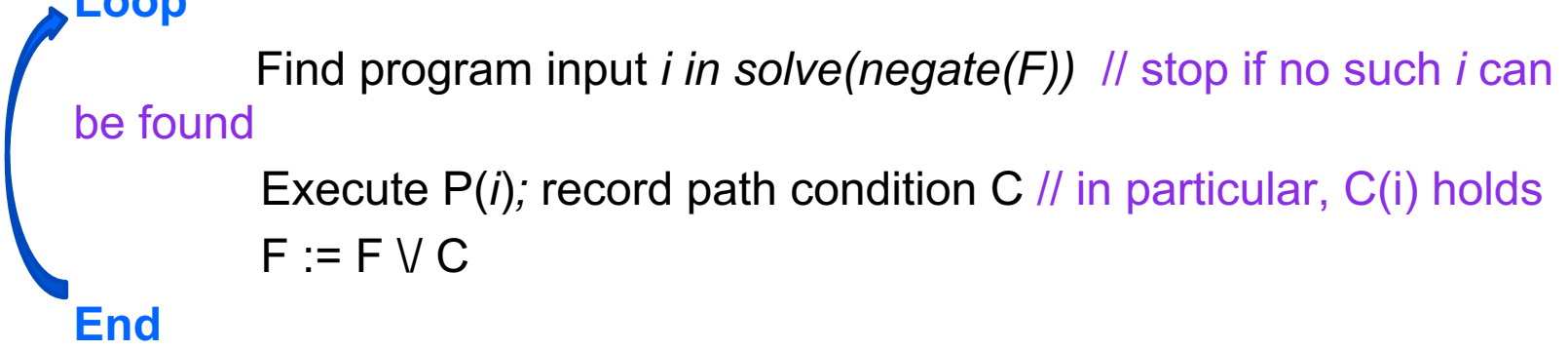
Implementing EXE

Execution states can be switched arbitrarily

Works best with virtualization or emulation

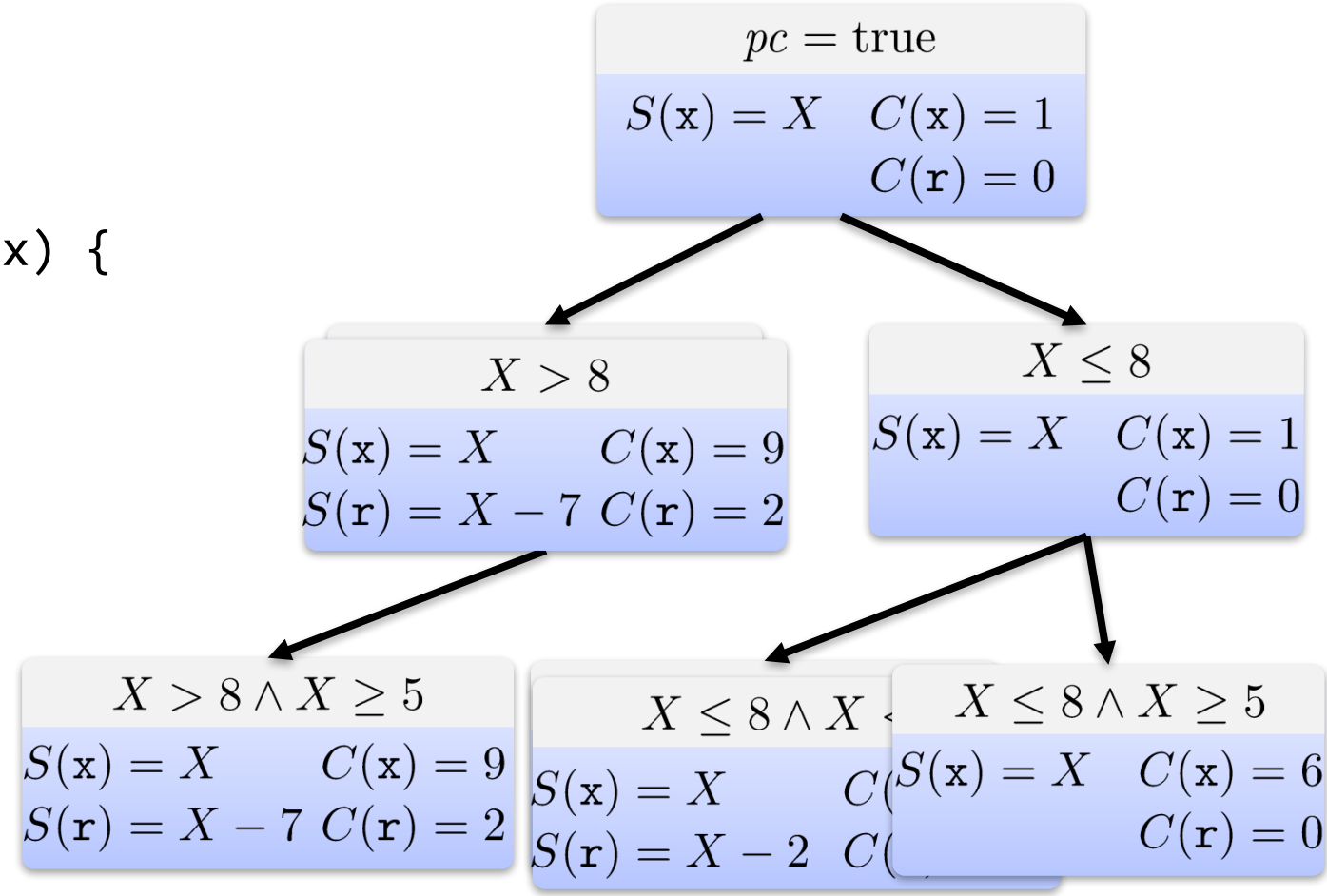
- Symbolic expressions are maintained in parallel to concrete values
- KLEE uses LLVM bitcode interpreter to maintain and update state
- S2E uses QEMU virtual machine to fork and restore the entire machine state, including OS

DART: Algorithm



DART

```
1  int proc(int x) {
2
3    int r = 0
5
6    if (x > 8) {
7      r = x - 7
8    }
9
10   if (x < 5) {
11     r = x - 2
12   }
13
14   return r;
15 }
```



New path condition:

$\neg(X > 8) \wedge \neg(X \geq 5)$

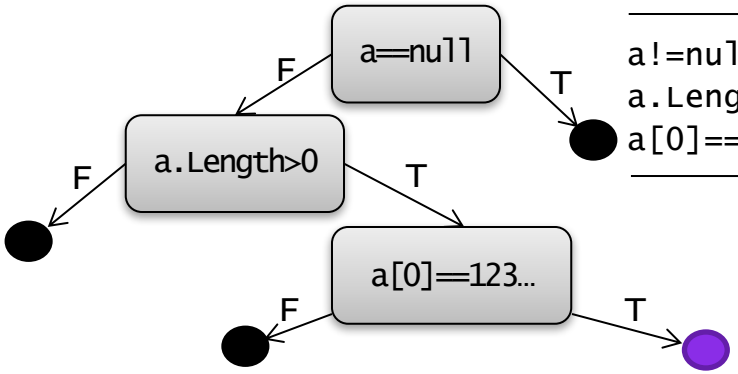
Test cases:

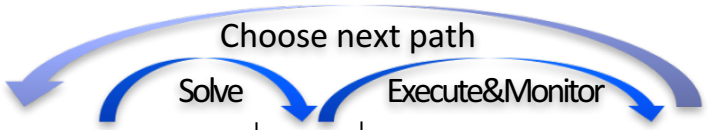
proc(9) proc(1) proc(6)

DART

Code to generate inputs for:

```
void CoverMe(int[] a)
{
  if (a == null) return;
  if (a.Length > 0)
    if (a[0] == 1234567890)
      throw new Exception("bug");
}
```





Constraints to solve	Data	Observed constraints
	null	a==null
a!=null	{}	a!=null && !(a.Length>0)
a!=null && a.Length>0	{0}	a!=null && a.Length>0 && a[0]!=123456890
a!=null && a.Length>0 && a[0]==1234567890	{123..}	a!=null && a.Length>0 && a[0]==123456890

Done: There is no path left.

Zero to Crash in 10 Generations

Starting with 100 zero bytes …

SAGE generates a crashing test for Media1 parser:

```
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

Zero to Crash in 10 Generations

Starting with 100 zero bytes …

SAGE generates a crashing test for Media1 parser:

```
00000000h: 52 49 46 46 00 00 00 00 00 00 00 00 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 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 1

Zero to Crash in 10 Generations

Starting with 100 zero bytes …

SAGE generates a crashing test for Media1 parser:

```
00000000h: 52 49 46 46 00 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 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 2

Zero to Crash in 10 Generations

Starting with 100 zero bytes …

SAGE generates a crashing test for Media1 parser:

```
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 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 3

Zero to Crash in 10 Generations

Starting with 100 zero bytes …

SAGE generates a crashing test for Media1 parser:

```
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 00 00 00 00 ; ....strh........
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 4

Zero to Crash in 10 Generations

Starting with 100 zero bytes …

SAGE generates a crashing test for Media1 parser:

```
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 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 5

Zero to Crash in 10 Generations

Starting with 100 zero bytes …

SAGE generates a crashing test for Media1 parser:

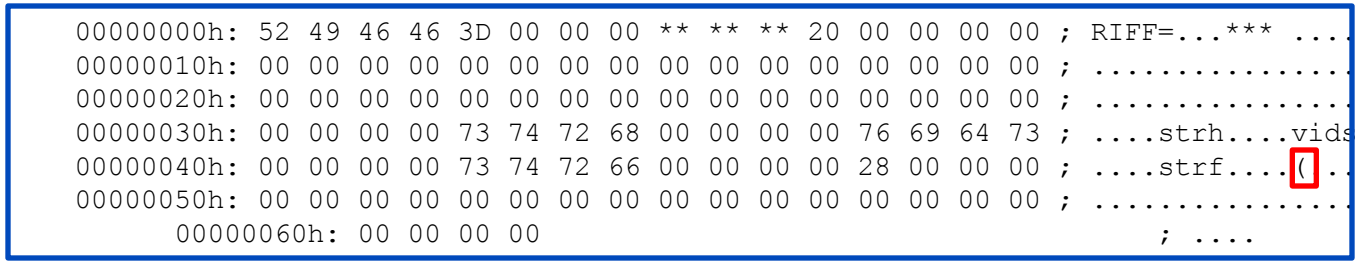
```
00000000h: 52 49 46 46 3D 00 00 00 ** ** ** 20 00 00 00 00 ; RIFF=...*** ....
00000010h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000020h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000030h: 00 00 00 00 73 74 72 68 00 00 00 00 76 69 64 73 ; ....strh....vids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

Generation 6

Zero to Crash in 10 Generations

Starting with 100 zero bytes …

SAGE generates a crashing test for Media1 parser:

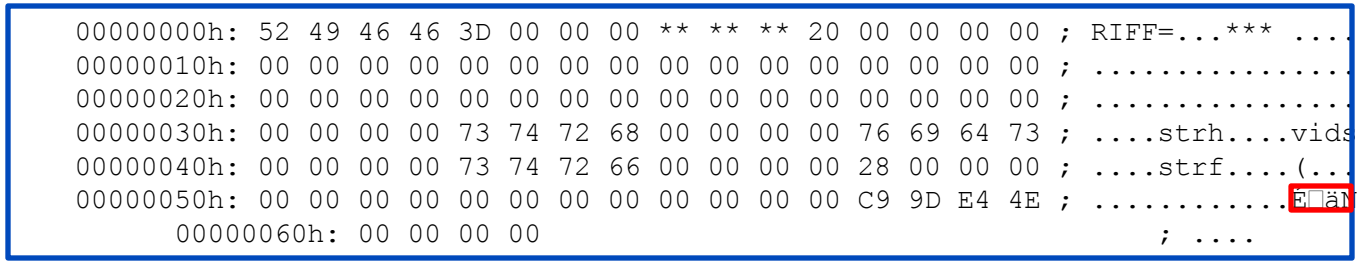


Generation 7

Zero to Crash in 10 Generations

Starting with 100 zero bytes …

SAGE generates a crashing test for Media1 parser:



Generation 8

Zero to Crash in 10 Generations

Starting with 100 zero bytes …

SAGE generates a crashing test for Media1 parser:

```
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 00 00 00 00 28 00 00 00 ; ....strf....(...
00000050h: 00 00 00 00 00 00 00 00 00 00 00 00 01 00 00 00 ; ................
      00000060h: 00 00 00 00                                ; ....
```

Generation 9

Zero to Crash in 10 Generations

Starting with 100 zero bytes …

SAGE generates a crashing test for Media1 parser:

```
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                                ; ....
```

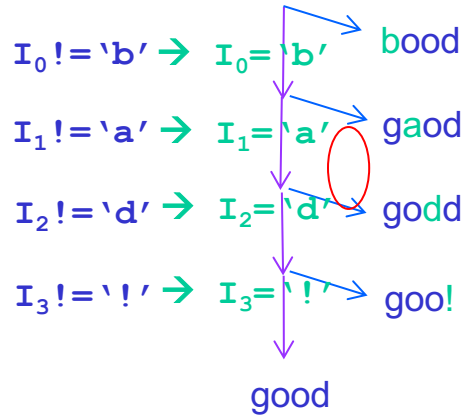
Generation 10

Example

```
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:



Gen 1

Negate each constraint in path constraint

Solve new constraint → new input

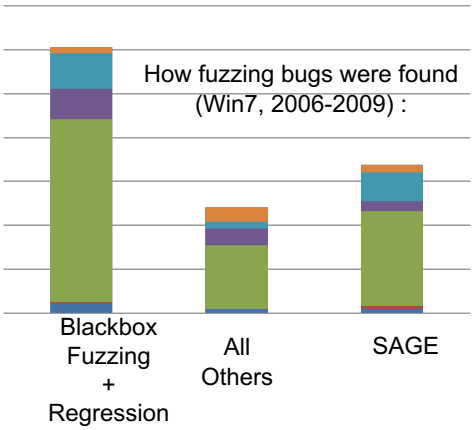
Whitebox File Fuzzing

SAGE @ Microsoft:

- 1st whitebox fuzzer for security testing
- 400+ machine years (since 2008) →
- 3.4+ Billion constraints
- 100s of apps, 100s of security bugs



- Example: Win7 file fuzzing
  - ~1/3 of all fuzzing bugs found by SAGE →
    (missed by everything else…)
- Bug fixes shipped (quietly) to 1 Billion+ PCs
- Millions of dollars saved
  - for Microsoft + time/energy for the world



Implementing DART

Instructions are instrumented or recorded

- Concrete program execution proceeds normally

Path condition and expressions computed following the concrete execution

- SAGE separates tracing and symbolic execution
- CUTE/CREST instruments the program to compute expressions on the fly

Generational search

- Generates as many test cases as possible per trace

EXE vs. DART

- Fine-grained control of execution
- Shallow exploration
  - Many queries early on
- Online SE
  - SE and interpretation in lockstep

Complete execution from first step

Deep exploration

- One query per run

Offline SE possible

- Execute along recorded trace

Loops

Symbolic execution dynamically unrolls loops

- Small-step semantics of strongest post-conditions
- No loop invariants required
- Can take a long (infinite!) time

Types of loops

- Input independent – unrolled as long as necessary
- Input dependent – different possibilities

Input-dependent Loops

Unrolling in EXE – online SE

- Every iteration of the loop forks execution
- Search algorithm decides whether to continue unrolling loop or to break out

Unrolling in DART – offline SE

- Concrete input determines iterations / unrollings
- Search algorithm can flip one of the loop branches to change the number of iterations

Naïve search algorithms can get stuck in loops

Concretization

Parts of input space can be kept concrete

- Reduces complexity
- Focuses search

Expressions can be concretized at runtime

- Avoid expressions outside of SMT solver theories (non-linear etc.)

Sound but incomplete

Concretization in EXE (Example)

$true$

$C(X) = 5$

$S(\texttt{m}) = X + 2$ $\quad C(\texttt{m}) = 7$

$S(\texttt{size}) = Y$ $\quad C(\texttt{size}) = 256$

```
if (m*m > size) {
  …
```

$(X + 2)(X + 2) \not> Y$

Concretization constraint

Solution diverges from expected path! (e.g., X = 2)

$49 > Y \wedge X = 5$

$C(X) = 5$

$S(\texttt{m}) = X + 2$ $\quad C(\texttt{m}) = 7$

$S(\texttt{size}) = Y$ $\quad C(\texttt{size}) = 48$

Implementing Concretization

Input

- concrete and symbolic states `C` and `S`
- a symbolic expression E to evaluate

Algorithm

- pick variables $x_1, \ldots, x_k$ in E to concretize
- replace x_i by $C(x_i)$ in E
- `v:=S.eval(E)`; CC := $x_1=C(x_1) \wedge \ldots \wedge x_k=C(x_k)$
- add *concretization constraint* CC to the path condition
- return v

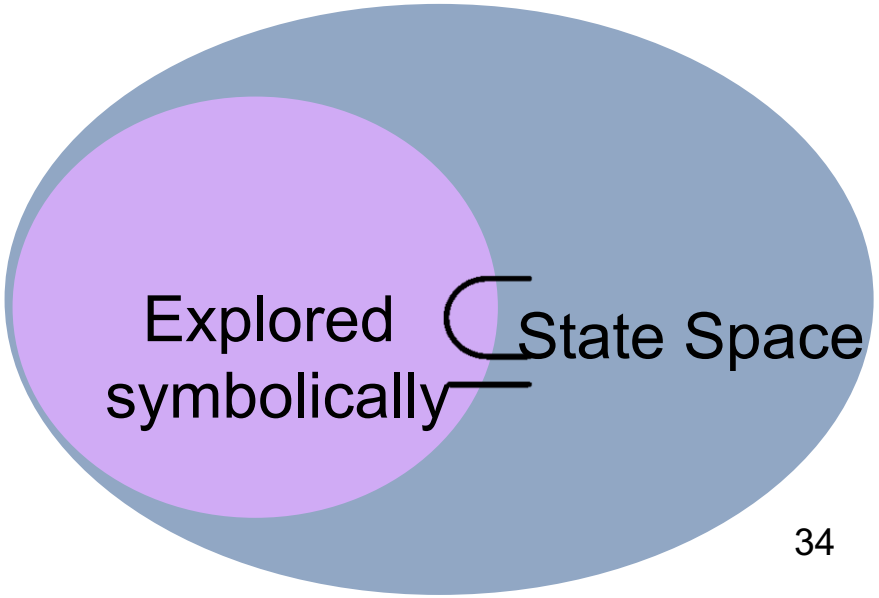
Soundness & Completeness

Conceptually, each path is exact

- Strongest postcondition in predicate transformer semantics
- No over-approximation, no under-approximation

Globally, SE under-approximates

- Explores only subset of paths in finite time
- “Eventual” completeness



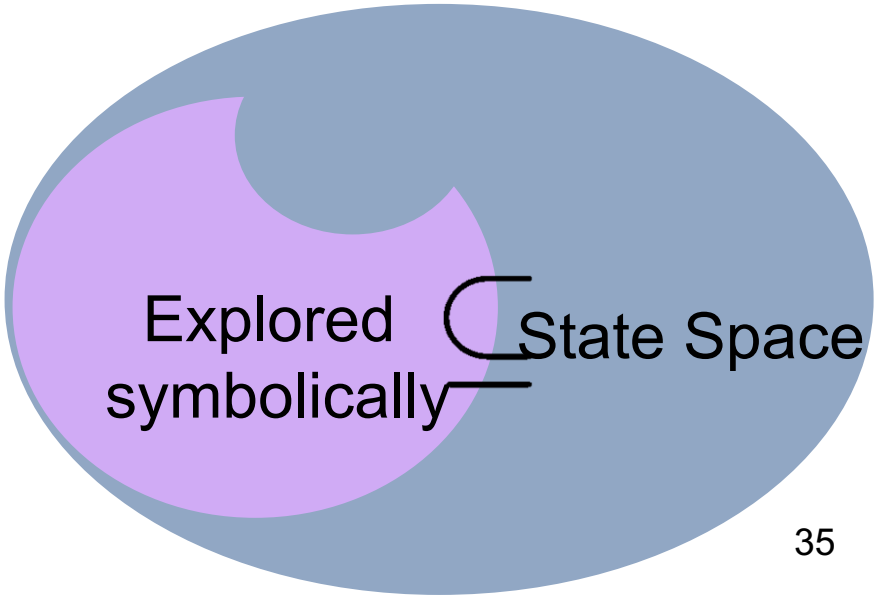
Soundness & Completeness

Symbolic Execution = Underapproximates

- Soundness = does not include infeasible behavior
- Completeness = explores all behavior

Concretization restricts state covered by path

- Remains sound
- Loses (eventual) completeness



Concretization

Key strength of dynamic symbolic execution

Enables external calls

- Concretize call arguments
- Callee executes concretely

Concretization constraints can be omitted

- Sacrifices soundness (original DART)
- Deal with divergences by random restarts

Challenges for Symbolic Execution

Expensive to create representations

Expensive to reason about expressions

- Although modern SAT/SMT solvers help!

Problems with function calls – need to keep track of calling contexts

- Called *interprocedural* analysis

Problem with handling loops

- often unroll them up to a certain depth rather than dealing with termination or loop invariants

Aliasing – leads to a massive blow-up in the number of paths