Automated Test Generation via SAT/SMT Solvers

Summer School, Marktoberdorf, 2014

Lecture 1

- Automated Test Generation (ATG) and applications

- SAT solving via DPLL
- Encoding of basic (program) operations over bit vectors to SAT
- Z3: SAT/SMT Solver (Python interface)

- ATG of programs via reduction to SAT
- From symbolic execution to dynamic symbolic execution

Lecture 2

- Design and implementation of dynamic symbolic execution
  - for Python
  - in Python
- Exercises and extensions for you to work on!

Lecture 3

- On the power/limits of dynamic symbolic execution
- Satisfiability modulo theories (SMT) solvers
- Extending DSE for Python with SymbolicDict via array theory

Lecture 4

- Strings, Regular Expressions and Symbolic Automata

Lecture 1

Automatic Test Generation

via

Dynamic Symbolic Execution

Automated (White Box) Test Generation

Given a program with a set of input parameters, *automatically generate a set of input values* that will cover as many statements/branches/paths as possible (or find as many bugs as possible)

Thanks to Patrice Godefroid, Nikolai Tillman

Applications

- Security: Whitebox File Fuzzing (MSR’s SAGE)

- Software development: Parameterized Unit Testing (MSR’s Pex)

- Many others
  - Performance testing of operating systems (MIT’s Commuter)
  - Malware analysis (CMU/Berkeley’s BitScope)
  - Generate of worm filters (MSR’s Vigilante)

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:

```
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 00 00 00 00 ; ................
00000060h: 00 00 00 00                                     ; ....
```

Generation 7

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 C9 9D E4 4E ; ............É䝸N
00000060h: 00 00 00 00                                     ; ....
```

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"

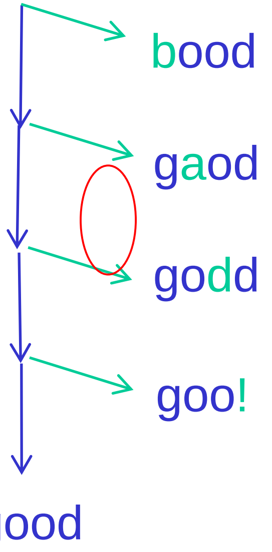
Path constraint:

I_0!='b' → I_0='b'

I_1!='a' → I_1='a'

I_2!='d' → I_2='d'

I_3!='!' → I_3='!'



Gen 1

Negate each constraint in path constraint

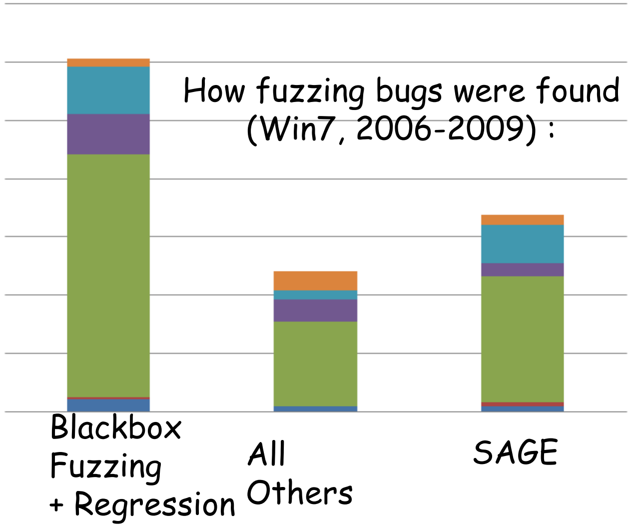
Solve new constraint → new input

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



Whitebox Testing and Satisfiability (SAT)

	Testing	SAT
Source	Program	Boolean formula
Question	Is there an input that covers some statement?	Is there a satisfying assignment?
Complexity	Undecidable	NP-complete

Propositional Formula (CNF)

$$
\begin{aligned}
\varphi \quad = \quad & (a \vee \neg b \vee d) \wedge (a \vee \neg b \vee e) \wedge \\
& (\neg b \vee \neg d \vee \neg e) \wedge \\
& (a \vee b \vee c \vee d) \wedge (a \vee b \vee c \vee \neg d) \wedge \\
& (a \vee b \vee \neg c \vee e) \wedge (a \vee b \vee \neg c \vee \neg e)
\end{aligned}
$$

SAT Solving via DPLL

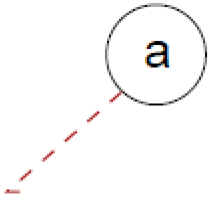
- Standard backtrack search
- DPLL(F) :
  - Apply unit propagation
  - If conflict identified, return UNSAT
  - Apply the pure literal rule
  - If F is satisfied (empty), return SAT
  - Select decision variable x
    - If DPLL($F \wedge x$)=SAT return SAT
    - return DPLL($F \wedge \neg x$)

DPLL (example)

$$\begin{aligned}\varphi \quad = \quad & (a \vee \neg b \vee d) \wedge (a \vee \neg b \vee e) \wedge \\ & (\neg b \vee \neg d \vee \neg e) \wedge \\ & (a \vee b \vee c \vee d) \wedge (a \vee b \vee c \vee \neg d) \wedge \\ & (a \vee b \vee \neg c \vee e) \wedge (a \vee b \vee \neg c \vee \neg e)\end{aligned}$$

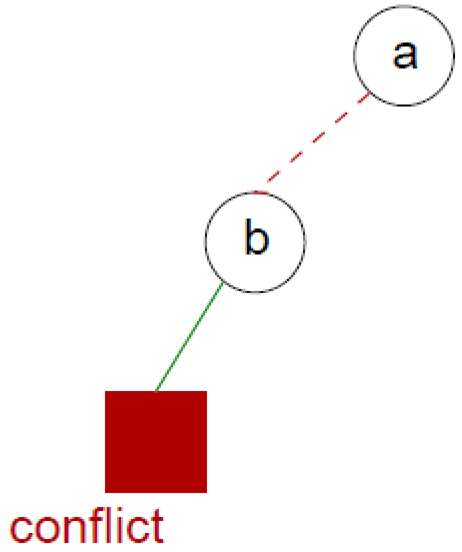
DPLL (example)

$$
\begin{aligned}
\varphi \quad = \quad & (a \vee \neg b \vee d) \wedge (a \vee \neg b \vee e) \wedge \\
& (\neg b \vee \neg d \vee \neg e) \wedge \\
& (a \vee b \vee c \vee d) \wedge (a \vee b \vee c \vee \neg d) \wedge \\
& (a \vee b \vee \neg c \vee e) \wedge (a \vee b \vee \neg c \vee \neg e)
\end{aligned}
$$



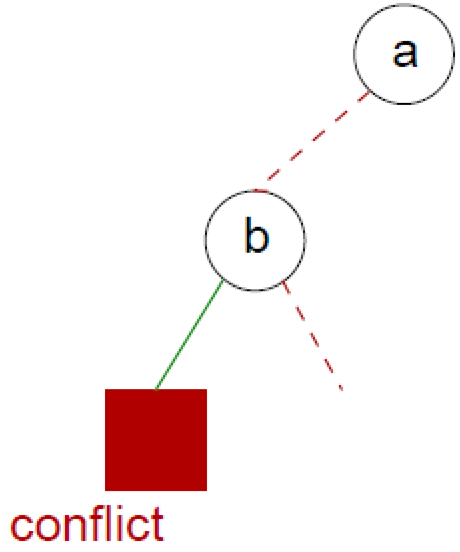
DPLL (example)

$$\begin{aligned}\varphi \;=\; & (a \vee \neg b \vee d) \wedge (a \vee \neg b \vee e) \wedge \\ & (\neg b \vee \neg d \vee \neg e) \wedge \\ & (a \vee b \vee c \vee d) \wedge (a \vee b \vee c \vee \neg d) \wedge \\ & (a \vee b \vee \neg c \vee e) \wedge (a \vee b \vee \neg c \vee \neg e)\end{aligned}$$



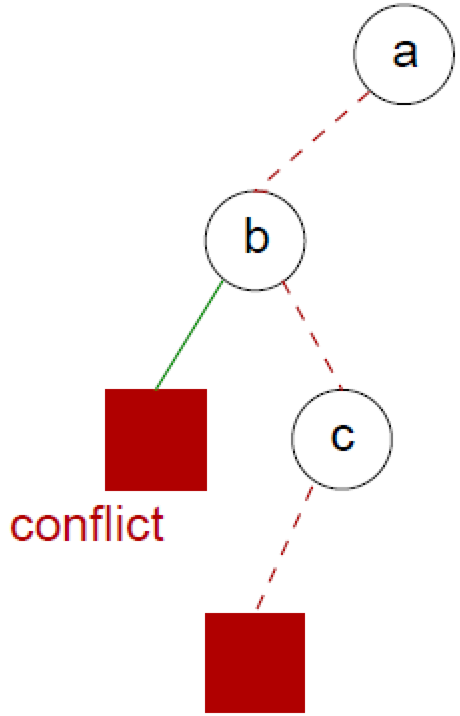
DPLL (example)

$$\begin{aligned}\varphi \quad = \quad & (a \vee \neg b \vee d) \wedge (a \vee \neg b \vee e) \wedge \\ & (\neg b \vee \neg d \vee \neg e) \wedge \\ & (a \vee b \vee c \vee d) \wedge (a \vee b \vee c \vee \neg d) \wedge \\ & (a \vee b \vee \neg c \vee e) \wedge (a \vee b \vee \neg c \vee \neg e)\end{aligned}$$



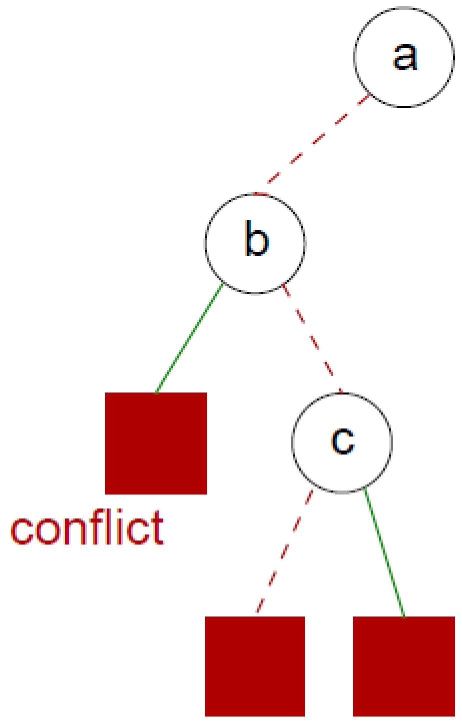
DPLL (example)

$$\begin{aligned}\varphi \quad = \quad & (a \vee \neg b \vee d) \wedge (a \vee \neg b \vee e) \wedge \\ & (\neg b \vee \neg d \vee \neg e) \wedge \\ & (a \vee b \vee c \vee d) \wedge (a \vee b \vee c \vee \neg d) \wedge \\ & (a \vee b \vee \neg c \vee e) \wedge (a \vee b \vee \neg c \vee \neg e)\end{aligned}$$



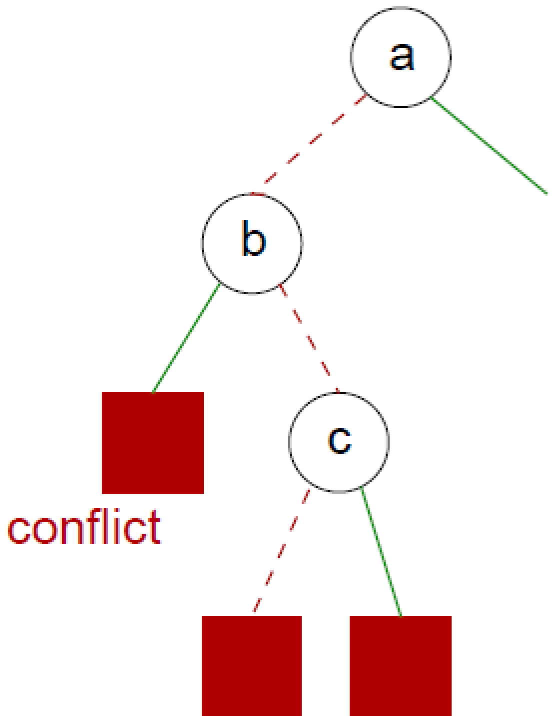
DPLL (example)

$$
\begin{aligned}
\varphi \;=\; & (a \vee \neg b \vee d) \wedge (a \vee \neg b \vee e) \wedge \\
& (\neg b \vee \neg d \vee \neg e) \wedge \\
& (a \vee b \vee c \vee d) \wedge (a \vee b \vee c \vee \neg d) \wedge \\
& (a \vee b \vee \neg c \vee e) \wedge (a \vee b \vee \neg c \vee \neg e)
\end{aligned}
$$



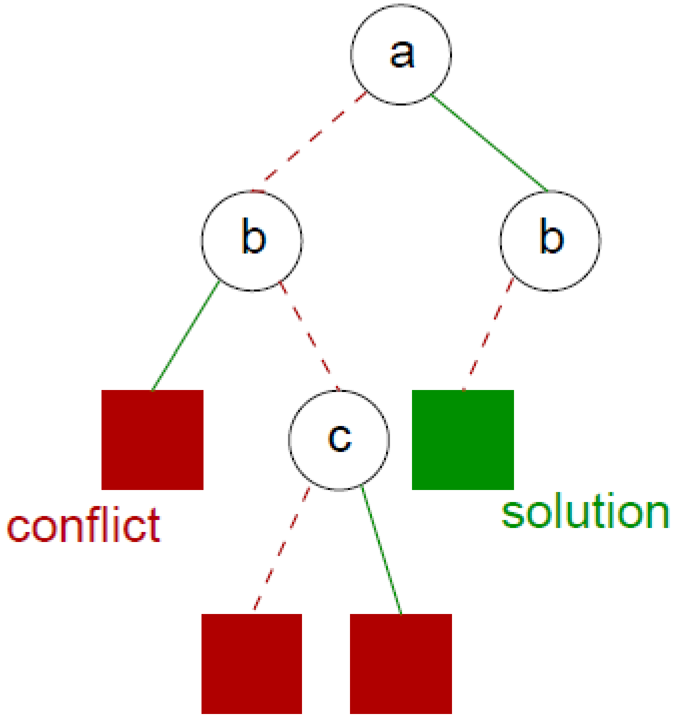
DPLL (example)

$$\begin{aligned}\varphi \quad = \quad & (a \vee \neg b \vee d) \wedge (a \vee \neg b \vee e) \wedge \\ & (\neg b \vee \neg d \vee \neg e) \wedge \\ & (a \vee b \vee c \vee d) \wedge (a \vee b \vee c \vee \neg d) \wedge \\ & (a \vee b \vee \neg c \vee e) \wedge (a \vee b \vee \neg c \vee \neg e)\end{aligned}$$



DPLL (example)

$$\begin{aligned}\varphi \;=\; & (a \vee \neg b \vee d) \wedge (a \vee \neg b \vee e) \wedge \\ & (\neg b \vee \neg d \vee \neg e) \wedge \\ & (a \vee b \vee c \vee d) \wedge (a \vee b \vee c \vee \neg d) \wedge \\ & (a \vee b \vee \neg c \vee e) \wedge (a \vee b \vee \neg c \vee \neg e)\end{aligned}$$



Bit-vector / Machine arithmetic

Let x, y and z be 8-bit (unsigned) integers.

Is $x > 0 \wedge y > 0 \wedge z = x + y \Rightarrow z > 0$ valid?

equivalently,

Is $x > 0 \wedge y > 0 \wedge z = x + y \wedge \neg(z > 0)$ unsatisfiable?

```
from z3 import *

x = BitVec("x",8)
y = BitVec("y",8)
z = BitVec("z",8)

s = Solver()
s.add(UGT(x,0),UGT(y,0),z==x+y)
s.add(Not(UGT(z,0)))

result = s.check()
if result == sat:
	print(s.model())
else:
	print(result)
```

Bit-vector / Machine arithmetic

We can encode bit-vector satisfiability problems in propositional logic.

Idea 1:

Use *n* propositional variables to encode *n*-bit integers.

$$x \rightarrow (x_1, \ldots, x_n)$$

Idea 2:

Encode arithmetic operations using hardware circuits.

Encoding equality

$p \Leftrightarrow q$ is equivalent to $(\neg p \vee q) \wedge (\neg q \vee p)$

The bit-vector equation x = y is encoded as:

$(x_1 \Leftrightarrow y_1) \wedge \ldots \wedge (x_n \Leftrightarrow y_n)$

Encoding addition

We use (r_1, …, r_n) to store the result of x + y

p xor *q* is defined as $\neg(p \Leftrightarrow q)$

xor is the 1-bit adder

p	*q*	*p* xor *q*	$p \wedge q$
0	0	0	0
1	0	1	0
0	1	1	0
1	1	0	1

carry

Encoding 1-bit full adder

1-bit full adder

Three inputs: x, y, c_{in}

Two outputs: r, c_{out}

x	y	c_{in}	$r = x \text{ xor } y \text{ xor } c_{in}$	$c_{out} = (x \wedge y) \vee (x \wedge c_{in}) \vee (y \wedge c_{in})$
0	0	0	0	0
1	0	0	1	0
0	1	0	1	0
1	1	0	0	1
0	0	1	1	0
1	0	1	0	1
0	1	1	0	1
1	1	1	1	1

Encoding n-bit adder

We use (r_1, …, r_n) to store the result of $x + y$,

$r_1 \Leftrightarrow (x_1 \text{ xor } y_1)$

$c_1 \Leftrightarrow (x_1 \wedge y_1)$

$r_2 \Leftrightarrow (x_2 \text{ xor } y_2 \text{ xor } c_1)$

$c_2 \Leftrightarrow (x_2 \wedge y_2) \vee (x_2 \wedge c_1) \vee (y_2 \wedge c_1)$

…

$r_n \Leftrightarrow (x_n \text{ xor } y_n \text{ xor } c_{n-1})$

$c_n \Leftrightarrow (x_n \wedge y_n) \vee (x_n \wedge c_{n-1}) \vee (y_n \wedge c_{n-1})$

Whitebox Testing and Satisfiability (SAT)

	Testing	SAT
Source	Program	Boolean formula
Question	Is there an input that covers some statement?	Is there a satisfying assignment?
Complexity	Undecidable	NP-complete

Reduction of Program Testing to SAT: Bounds!

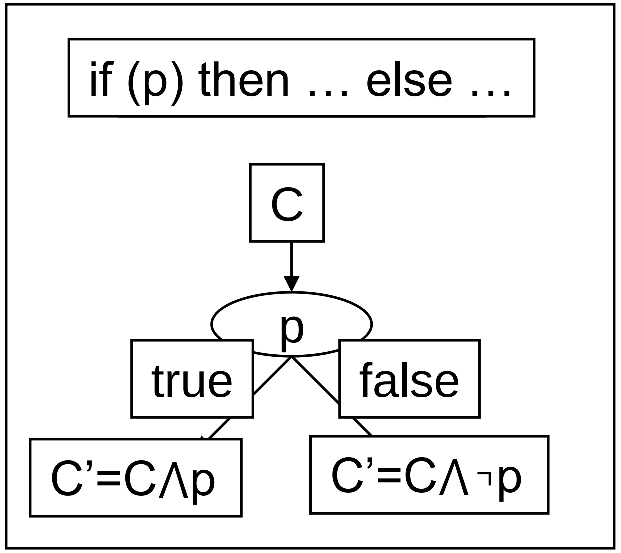
- Unbounded number of execution paths?
  - Explicit enumeration/exploration of program paths
  - <u>Bound the number of paths explored</u>

- Unbounded execution path length?
  - <u>Bound the input size and/or path length</u>

- Bounded exploration
  - enables conversion of a program path to a (finite) logic formula

Symbolic Execution

- Exploration of all feasible execution paths:
  - Start execution from initial state with symbolic values for all input
  - Program operations yield terms over symbolic values
  - At conditional branch, fork execution for each feasible evaluation of the condition
  - For each path, we get an accumulated path condition



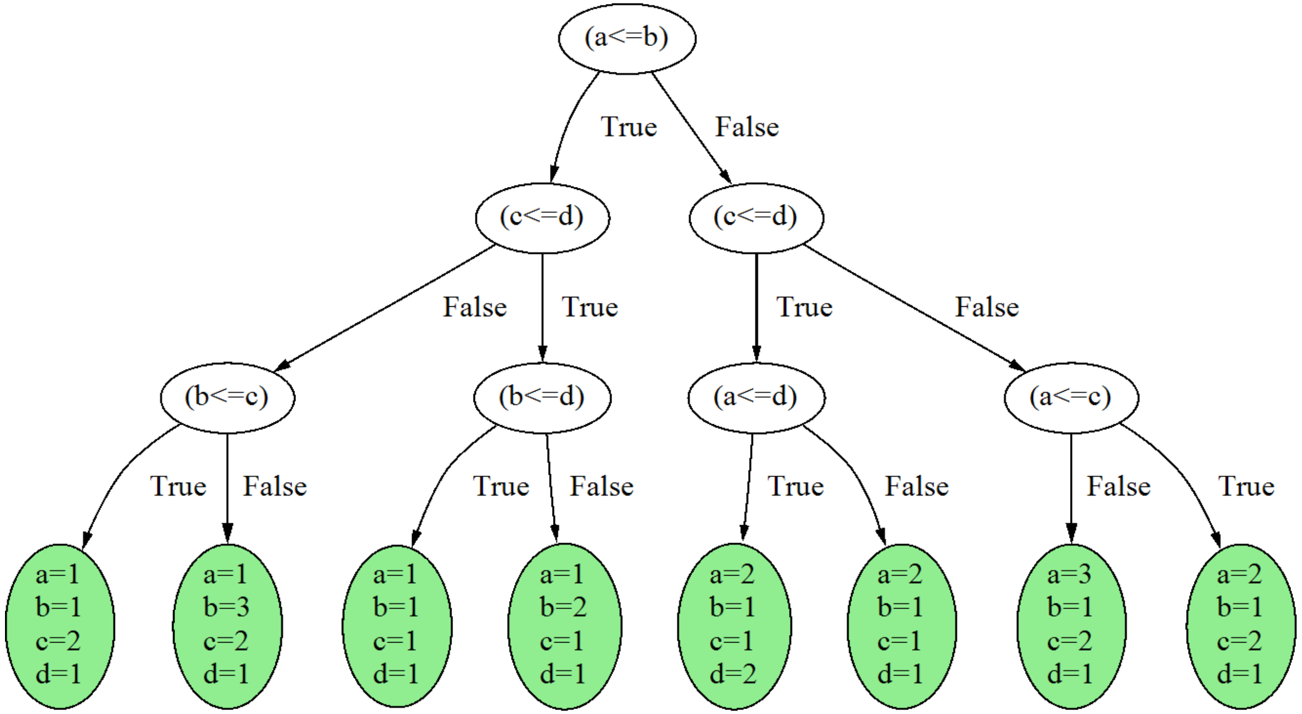
- For each path, check if path condition is satisfiable and generate input

- See: [King76]

Symbolic Execution Illustrated

```
int Max(int a, int b, int c, int d) {
    return Max(Max(a, b), Max(c, d));
}
```

```
int Max(int x, int y) {
    if (x <= y) return y;
    else return x;
}
```



Many problems remain

1. Code that is hard to analyze

2. Path explosion
    - Loops
    - Procedures

3. Environment (what are the inputs to the program under test?)
    - pointers, data structures, …
    - files, data bases, …
    - threads, thread schedules, …
    - sockets, …

1. 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 $\boxed{\text{(x != complex(y}}$))
  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 (<u>conc</u>rete + symb<u>olic</u>)
Referred to here as <u>dynamic symbolic execution</u>

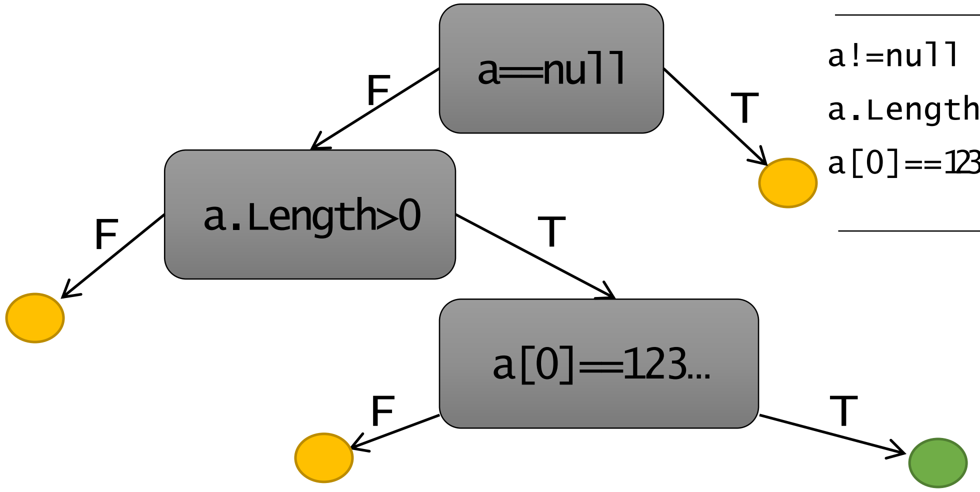
Dynamic Symbolic Execution

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]!=1234567890
a!=null && a.Length>0 && a[0]==1234567890	{123…}	a!=null && a.Length>0 && a[0]==1234567890

Done: There is no path left.



Dynamic Symbolic Execution

Formula F := False

Loop

  Find program input *i in solve(negate(F))*   // stop if no such *i* can be found

  Execute P(*i*)*;* record path condition C   // in particular, C(i) holds

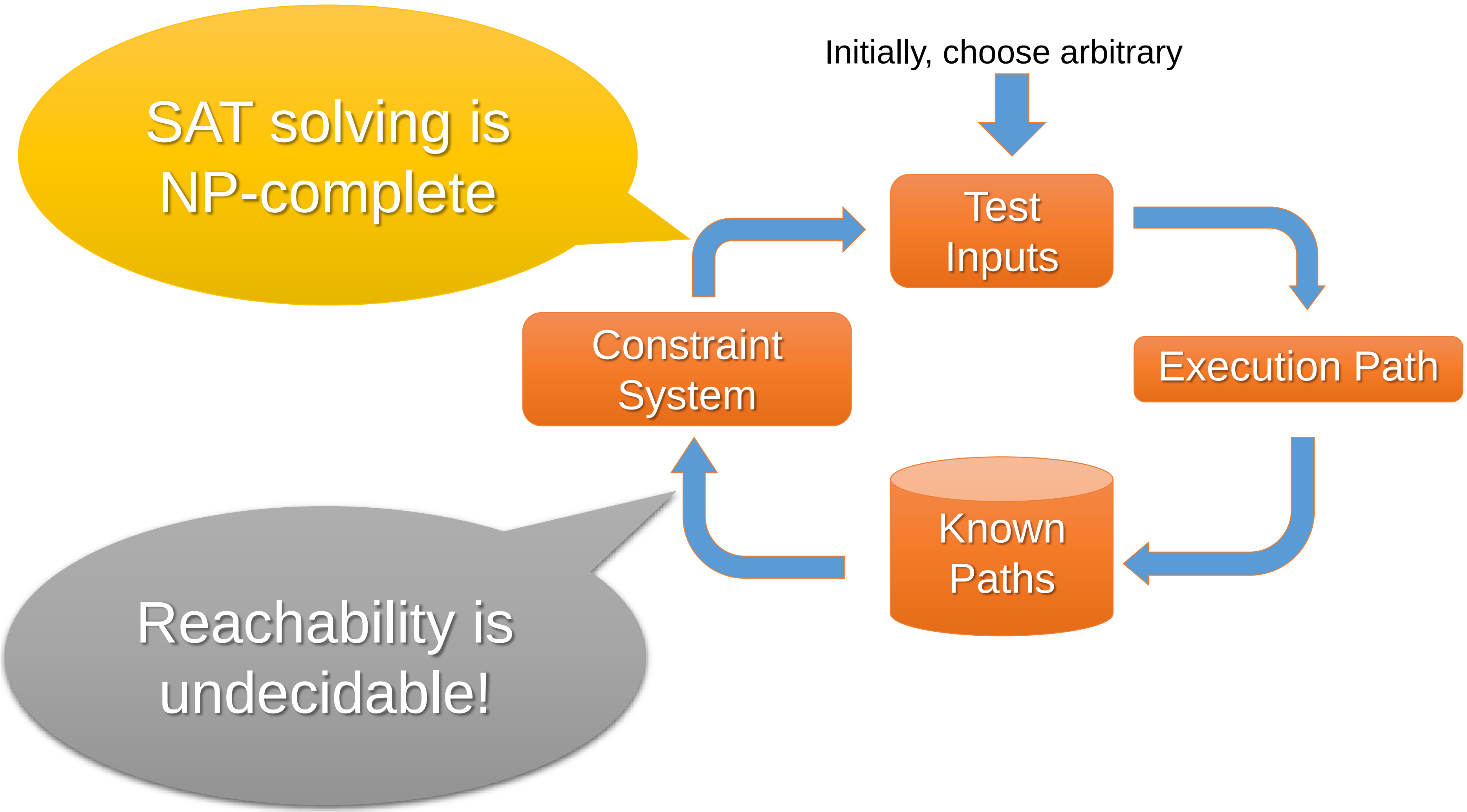
  F := F $\bigvee$ C

End

Dynamic Symbolic Execution: many implementations

- Defined by execution environment / programming language, symbolic execution precision, and constraint solving
  - Execution environment: C, Java, x86, .NET,…
  - Precision: linear vs. non-linear arithmetic, “gods integers” vs. bitvectors, concrete heap vs. symbolic heap., floating-point values, etc.
  - Solvers: lp_solve, CVCLite, STP, Disolver, Z3,…
- Examples of DSE implementations:
  - *DART* (Bell Labs), and also CUTE “concolic execution”
  - *EXE/EGT/KLEE* (Stanford) “constraint-based execution”
  - *Vigilante* (Microsoft) to generate worm filters
  - *BitScope* (CMU/Berkeley) for malware analysis
  - Sage (Microsoft) for security testing of X86 code
  - Yogi (Microsoft) to verify device drivers (integrated in SLAM)
  - Pex (Microsoft) for parameterized unit testing of .NET code
  - CREST, jCUTE, jFuzz, …

Recap: Test Generation using SAT solvers



References

- James C. King, Symbolic execution and program testing, Communications of the ACM, v.19 n.7, p.385-394, July 1976
- João P. Marques Silva, Karem A. Sakallah: GRASP: A Search Algorithm for Propositional Satisfiability. IEEE Trans. Computers 48(5): 506-521 (1999)
- Patrice Godefroid, Nils Klarlund, Koushik Sen: DART: directed automated random testing. PLDI 2005: 213-223
- Nikolai Tillmann, Wolfram Schulte: Parameterized unit tests. ESEC/SIGSOFT FSE 2005: 253-262
- Leonardo de Moura, Nikolaj Bjørner: Z3: An Efficient SMT Solver. TACAS 2008: 337-340
- Cristian Cadar, Daniel Dunbar, Dawson R. Engler: KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. OSDI 2008: 209-224
- Dries Vanoverberghe, Nikolai Tillmann, Frank Piessens: Test Input Generation for Programs with Pointers. TACAS 2009: 277-291
- Kenneth L. McMillan: Lazy Annotation for Program Testing and Verification. CAV 2010: 104-118
- Ella Bounimova, Patrice Godefroid, David A. Molnar: Billions and billions of constraints: whitebox fuzz testing in production. ICSE 2013: 122-131

Assignment 1

- Download
  - Python 3.2.3 or later (http://www.python.org/)
  - Z3 'unstable’ for your platform (http://z3.codeplex.com/)
  - Git client (http://www.github.com/)
  - Clone https://github.com/thomasjball/PyExZ3.git

- Or get the code I have on USB key for Windows (and Z3 for all platforms)

- Write a Python function to encode n-bit multiplication using Z3 Bools (you can use PyExZ3\examples\adder.py)

- Use Z3 to prove that your multiplier is equivalent to Z3’s BitVector multiplier (you can use PyExZ3\examples\check_adder.py)