IA169 System Verification and Assurance

Symbolic Execution

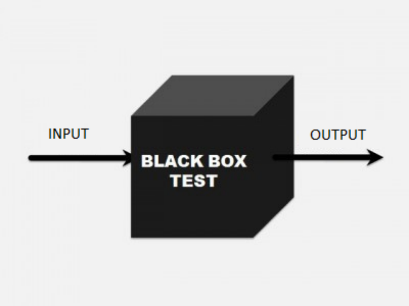
Jiří Barnat

Testing Strategies

Black-box Testing

Black-box

- A product under test is viewed as a **black box**.
- It is analysed through the input-output behaviour.
- Inner details (such as source code) are hidden or not taken into account.



White-box Testing (Glass-box)

- Inner details are taken into account.
- Tests are selected and executed with respect to the inner details of the product, e.g. code coverage.
- Error insertion, modification of the product for the purpose of testing.
- **Basically only extends any Black-box approach.**

Gray-box Testing

- In between of Black-box and White-box.
- Sometimes the same as White-box, inconsistent terminology.

Testing Techniques

Primary Black-box Strategies

- Domain Testing
- Combinatory Testing
- Scenario Testing
- Risk-based Testing
- Functional Testing
- Fuzz Testing (Mutation Testing)

Primary White-box Extensions

- Model-based Testing
- Unit Testing

Support for Developers

- Regression Testing

Symbolic Execution

Motivation

Problem

- To detect errors that systematically exhibit only for specific input values is difficult.
- Relates to incompleteness of testing.

Still we would like to ...

- test the program on inputs that make program execute differently from what has already been tested.
- test the program for all inputs.

Symbolic Execution

Idea

- Execute a program so that values of input variables are referred to as to symbols instead of concrete values.

Demo

Program	Selected concrete values	Symbolic representation
`read(A)`		
	$A = 3$	$A = \alpha$
`A = A * 2`		
	$A = 6$	$A = \alpha * 2$
`A = A + 1`		
	$A = 7$	$A = (\alpha * 2) + 1$
`output(A)`		

Branching and Path Condition

Observation

- Branching in the code put some restrictions on the data depending on the condition of a branching point.

Example

```
1 if (A == 2)
2   then ...
3   else ...
```

$A = (\alpha * 2) + 1$

$(\alpha * 2) + 1 = 2$

$(\alpha * 2) + 1 \neq 2$

Path Condition

- Formula over symbols referring to input values.
- Encodes history of computation, i.e. cumulative restrictions implied from all the branching points walked-through up to the curent point of execution.
- Initially set to true.

Unfeasible Paths

Observation

- The path condition may become unsatisfiable.
- If it is so, it means there are no input values that would make the program execute this way.

Example 1

`if (A == B)`	$A = \alpha, B = \beta$	
`  then`	$\alpha = \beta$	
`    if (A == B)`		
`      then ...`	$\alpha = \beta \wedge \alpha = \beta$	
~~`      else ...`~~	$\alpha = \beta \wedge \alpha \neq \beta$	*is UNSAT*
`  else ...`	$\alpha \neq \beta$	

Example 2

% – operation modulo

`A=A%2`	$A = \alpha\%2$	
`if (A == 3)` ~~`then ...`~~	$\alpha\%2 = 3$	*is UNSAT*
`            else ...`	$\alpha\%2 \neq 3$	

Tree of Symbolic Execution

Observation

- All possible executions of program may be represented by a tree structure – **Symbolic Execution Tree**.
- The tree is obtained by unfolding/unwinding the control flow graph of the program.

Symbolic Execution Tree

- Node of the tree encodes program location, symbolic representation of variables, and a concrete path condition.

location	symbolic valuation	path condition
#12	$A = \alpha + 2, B = \alpha + \beta - 2$	$\alpha = 2 * \beta - 1$

- An edge in the tree corresponds to a symbolic execution of a program instruction on a given location.
- Branching point is reflected as branching in the tree and causes updates of path conditions in individual branches.

Program

```
input A,B
if (B<0) then
  return 0
else
  while (B > 0)
  { B=B-1
    A=A+B
  }
return A
```

Draw Yourself.

Path Explosion

Properties of Symbolic Tree Execution

- No nodes are merged, even if they are the same (the structure is a tree).
- A single program location may be contained in (infinitely) many nodes of the tree.
- Tree may contain infinite paths.

Path Explosion Problem

- The number of branches in the symbolic execution tree may be large for non-trivial programs.
- The number of paths may grow exponentially with the number of branching points visited.

Employing Symbolic Execution Tree for Verification

Analysis of the Tree

- Breadth-first strategy, the tree may be infinite.

Deduced Program Properties

- Identification of feasible and unfeasible paths.
- Proof of reachability of a given program location.
- Error detection (division by zero, out-of-array access, assertion violation, etc.).

Synthesis of Test Input Data

- If the formula encoded as a path condition is satisfiable for a symbolic run, the model of the formula gives concrete input values that make the program to follow the symbolic run.
- Excellent for synthesis of tests that increase code coverage.

Automated Test Generation

Principle

1. Generate random input values (encode some random path).
2. Perform a walk through the Symbolic Execution Tree with the random input values and record the path condition.
3. Generate a new path condition from the recorded one by negating one of the restrictions related to a single branching point.
4. Find input values satisfying the new path condition.
5. Repeat from number 2 until desired coverage is reached.

Practical Notes

- Heuristics for selection of branching point to be negated.
- Augmentation of the code to enable path condition recording.

Limits of Symbolic Execution

Undecidability

- Using complex arithmetic operations on unbounded domains implies general undecidability of the formula satisfaction problem.
- Symbolic Execution Tree is infinite (due to unwinding of cycles with unbound number of repetition).

Computational Complexity

- Path explosion problem.
- Efficiency of algorithms for formula satisfiability on finite domains.

Known Limits

- Symbolic operations on non-numerical variables.
- Not clear how to deal with dynamic data structures.
- Symbolic evaluation of calls to external functions.

Tools for SAT Solving

SAT Problem

Satisfiability Problem – SAT

- Is to decide if there exists a valuation of Boolean variables of propositional logic formula that makes the formula hold true (be valid).

SAT Problem Properties

- Famous NP-complete problem.
- Polynomial algorithm is unlikely to exist.
- Still there are existing SAT solvers that are very efficient and due to a plethora of heuristics can solve surprisingly large instances of the problem.

Tool Z3

ZZZ aka **Z3**

- Developed by Microsoft Research.
- SAT and SMT Solver.
- WWW interface — `http://www.rise4fun.com/Z3`
- Standardised binary API for use within other verification tools.

Decide using Z3

- Is formula $(a \vee \neg b) \wedge (\neg a \vee b)$ satisfiable?

Usage of Z3 – SAT

Reformulate into language of Z3 $(a \vee \neg b) \wedge (\neg a \vee b)$

```
(declare-const a Bool)
(declare-const b Bool)
(assert (and (or a (not b)) (or (not a) b)))
(check-sat)
(get-model)
```

Answer of Z3

```
sat
(model
  (define-fun b () Bool
    false)
  (define-fun a () Bool
    false)
)
```

Satisfiability Modulo Theory – SMT

Satisfiability Modulo Theory – SMT

- Is to decide satisfiability of first order logic with predicates and function symbols that encode one or more selected theories.
- Typically used theories
  - Arithmetic of integral and floating point numbers.
  - Theories of data structures (lists, arrays, bit-vectors, . . .).

Other view (Wikipedia)

- SMT can be thought of as a form of the constraint satisfaction problem and thus a certain formalised approach to constraint programming.

Solve using Z3 http://rise4fun.com/Z3/tutorial/guide

- Are there two integral non-zero numbers x and y such that y=x*(x-y)?

```
(declare-const y Int)
(declare-const x Int)
(assert (= y (* x (- x y))))
(assert (not (= y 0)))
(check-sat)
(get-model)
```

- Are there two integral non-zero numbers x and y such that y=x*(x-(y*y))?

```
(declare-const y Int)
(declare-const x Int)
(assert (= y (* x (- x (* y y)))))
(assert (not (= x 0)))
(check-sat)
```

Satisfiability and Validity

Observation

- A formula is valid if and only if its negation is not satisfiable.

Consequence

- SAT and SMT solvers can be used as theorem provers to show validity of some theorems.

Model Synthesis

- SAT solvers not only decide satisfiability of formulae but in positive case also give concrete valuation of variables for which the formula is valid.
- Unlike general theorem provers they provide a counterexample in case the theorem to be proved is invalid (negation is satisfiable).

Concolic Testing

Motivation

Problem

- Efficient undecidability of path feasibility.
- In practice, unknown result often means unsatisfiability (no witness found).
- However, skipping paths that we only think are unfeasible, may result in undetected errors.
- On the other hand, executing unfeasible path may report unreal errors.

Partial Solution

- Let us use concrete and symbolic values at the same time in order to support decisions that are practically undecidable by a SAT or SMT solver.
- Heuristics.
- An interesting case (correct): UNKNOWN $\Longrightarrow$ SAT
- **Con**crete and Symb**olic Testing** = **Concolic Testing**

Hypothetical demo of concolic testing

Program

```
1 input A,B
2 if (A==(B*B)%30) then
3   ERROR
4 else
5   return A
```

Concolic Testing

1. A=22, B=7 (random values), test executed, no errors found.
2. (22==(7*7)%30) is *False*, path condition: $\alpha \neq (\beta * \beta)\%30$
3. Synthesis of input data from negation of path condition: $\alpha = (\beta * \beta)\%30$ – **UNKNOWN**
4. Employ concrete values: $\alpha = (7 * 7)\%30$ – **SAT**, $\alpha = 19$
5. A=19, B=7
6. Test detected error location on program line 3.

Section

SAGE Tool

Story of SAGE

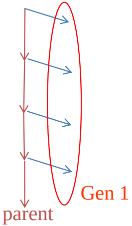
Systematic Testing for Security: Whitebox Fuzzing

Patrice Godefroid
Michael Y. Levin and David Molnar

http://research.microsoft.com/projects/atg/
Microsoft Research

Whitebox Fuzzing (SAGE tool)

- Start with a well-formed input (not random)
- Combine with a generational search (not DFS)
  - Negate 1-by-1 each constraint in a path constraint
  - Generate many children for each parent run
  - Challenge all the layers of the application sooner
  - Leverage expensive symbolic execution



- Search spaces are huge, the search is partial… yet effective at finding bugs !

Example: Dynamic Test Generation

```
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"

Dynamic Test Generation

```
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();
}
```

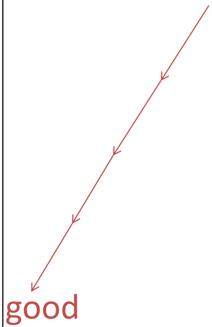
input = "good"

Path constraint:

I_0 != 'b'
I_1 != 'a'
I_2 != 'd'
I_3 != '!'

Negate a condition in path constraint
Solve new constraint → new input

Depth-First Search

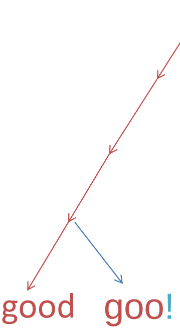


input = "good"

```
void top(char input[4])
{
    int cnt = 0;
    if (input[0] == 'b') cnt++;   I0 != 'b'
    if (input[1] == 'a') cnt++;   I1 != 'a'
    if (input[2] == 'd') cnt++;   I2 != 'd'
    if (input[3] == '!') cnt++;   I3 != '!'
    if (cnt > 3) crash();
}
```

Story of SAGE

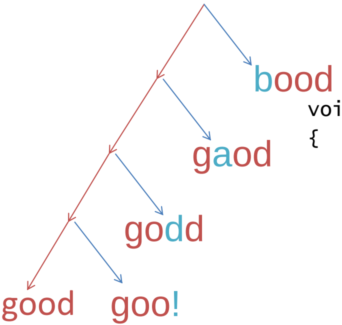
Depth-First Search




```
void top(char input[4])
{
    int cnt = 0;
    if (input[0] == 'b') cnt++;   I0 != 'b'
    if (input[1] == 'a') cnt++;   I1 != 'a'
    if (input[2] == 'd') cnt++;   I2 != 'd'
    if (input[3] == '!') cnt++;   I3 == '!'
    if (cnt > 3) crash();
}
```

Story of SAGE

Generational Search



Four "Generation 1" test cases !

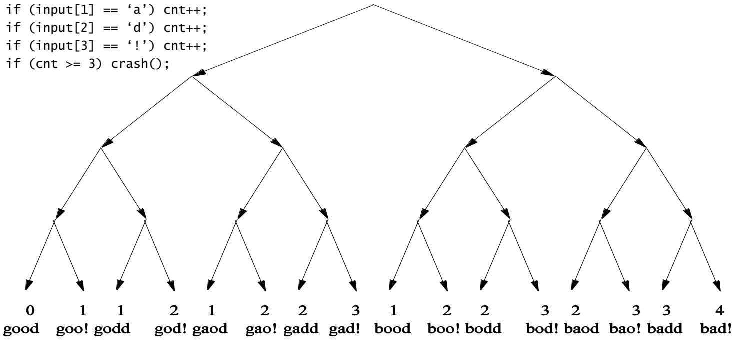
```
void top(char input[4])
{
    int cnt = 0;
    if (input[0] == 'b') cnt++;
    if (input[1] == 'a') cnt++;
    if (input[2] == 'd') cnt++;
    if (input[3] == '!') cnt++;
    if (cnt > 3) crash();
}
```

I_0 == 'b'
I_1 == 'a'
I_2 == 'd'
I_3 == '!'

Story of SAGE

The Search Space

```
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();
}
```



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 ; ............EäN
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 – crash bucket 1212954973!

Initial Experiences with SAGE

- Since 1st internal release in April'07: tens of new security bugs found
- Apps: image processors, media players, file decoders,… Confidential !
- Bugs: Write A/Vs, Read A/Vs, Crashes,… Confidential !
- Many bugs found triaged as "security critical, severity 1, priority 1"

Practicals and Homework

Practicals

- Follow Klee tutorials 1 and 2 (`http://klee.github.io/klee/Tutorials.html`)

Homework

- Solve The wolf, goat and cabbage problem with Klee
- Solve `http://pex4fun.com/`