IA159 Formal Verification Methods

Symbolic execution

Jan Strejček

Department of Computer Science
Faculty of Informatics
Masaryk University

Focus and sources

Focus

- symbolic execution
- automated whitebox fuzz testing

Sources

- J. C. King: *Symbolic Execution and Program Testing*, Communications of ACM, 1976.
- P. Godefroid, M. Y. Levin, and D. Molnar: *Automated whitebox fuzz testing*, NDSS 2008.

Special thanks to Marek Trtík for providing me his slides.

Motivation

```
1   procedure sum(a, b, c) {
2       x := a + b
3       y := b + c
4       z := x + y − b
5       return(z)
6   }
```

Testing checks that a program behaves correctly on selected inputs:

- $sum(1, 1, 1) =$
- $sum(1, 2, 3) =$
- ...

Motivation

```
1   procedure sum(a, b, c) {
2       x := a + b
3       y := b + c
4       z := x + y − b
5       return(z)
6   }
```

Testing checks that a program behaves correctly on selected inputs:

- $sum(1, 1, 1) = 3$
- $sum(1, 2, 3) = 6$
- ...

Motivation

```
1   procedure sum(a, b, c) {
2       x := a + b
3       y := b + c
4       z := x + y − b
5       return(z)
6   }
```

We can execute the program with symbols $\alpha_1, \alpha_2, \alpha_3$ representing arbitrary values:

- $sum(\alpha_1, \alpha_2, \alpha_3) =$

Motivation

```
1   procedure sum(a, b, c) {
2       x := a + b
3       y := b + c
4       z := x + y − b
5       return(z)
6   }
```

We can execute the program with symbols $\alpha_1, \alpha_2, \alpha_3$ representing arbitrary values:

- $sum(\alpha_1, \alpha_2, \alpha_3) = \alpha_1 + \alpha_2 + \alpha_3$

$\rightarrow$ symbolic execution

Symbolic execution semantics in general

Each programing language has an execution semantics describing:

- the data objects which program variables may represent;
- how statements manipulate data objects;
- how control flows through the statements of a program.

In symbolic execution semantics:

- real data objects can be represented by symbols
- basic operators of the language are extended to accept symbolic input and produce symbolic output.

The execution semantics is changed for symbolic execution, but neither the language syntax nor the individual programs written in the language are changed.

Simple programming language

Consider the following programming language:

- all program variables are of type unbounded signed integer
- input can be obtained by procedure parameters, global variables, or read operations
- arithmetic expressions may contain only operators $+, -, *$
- commands:
  - assignment `<var> := <expr>`
  - `GOTO <label>`
  - `IF-THEN-ELSE` with condition `<expr>` ≥ 0

Semantics of the language

Standard execution semantics

- data objects = signed integers
- ...

Symbolic execution semantics

- besides integers, we can use symbols from the list $\alpha_1, \alpha_2, \alpha_3, \ldots$ to represent some data objects
- the only oportunity to introduce symbolic data objects is as inputs to the program
- the evaluation rules for arithmetic expressions used in assignments and `IF` statements must be extended to handle symbolic values
- `GOTO`'s function exactly as in normal executions

Extending rules for expressions and `IF` statement

Values of expressions and variables are integer polynomials over the symbols $\alpha_1, \alpha_2, \ldots$.

Extending rules for expressions and `IF` statement

Values of expressions and variables are integer polynomials over the symbols $\alpha_1, \alpha_2, \ldots$.

Although `IF` statement do not change state of program variables, it plays a key role in definition of symbolic semantics.

We extend system state by path condition *pc*, which is a conjunction of inequalities of the form $R \geq 0$ or $\neg(R \geq 0)$, where R is a polynomial over $\alpha_1, \alpha_2, \ldots$.

- *pc* is initially set to *true*
- *pc* can only be modified when executing `IF` statements

Intuitively, *pc* accumulates conditions navigating the execution to the current path.

Extending path condition

Let q be an inequality resulting from substituting values of variables into condition of a `IF` statement.

Assuming $pc \not\equiv \mathit{false}$, at most one of the following implications can be valid:

(a) $pc \implies q$

(b) $pc \implies \neg q$

Extending path condition

Let q be an inequality resulting from substituting values of variables into condition of a `IF` statement.

Assuming $pc \not\equiv \mathit{false}$, at most one of the following implications can be valid:

(a) $pc \implies q$

(b) $pc \implies \neg q$

If one implication is valid, then we speak about non-forking execution and pc is not changed.

- If (a) is valid, the execution continues by `THEN` branch.
- If (b) is valid, the execution continues by `ELSE` branch.

Extending path condition

Let q be an inequality resulting from substituting values of variables into condition of a `IF` statement.

Assuming $pc \not\equiv \mathit{false}$, at most one of the following implications can be valid:

(a) $pc \implies q$

(b) $pc \implies \neg q$

When neither (a) nor (b) is valid, then we speak about forking execution. The current execution forks into two independent ones, since both branches are possible. Path conditions of resulting executions are updated as follows:

- $pc := pc \wedge q$ for `THEN` branch
- $pc := pc \wedge \neg q$ for `ELSE` branch

Example

```
1   procedure power(x, y) {
2           z := 1
3           j := 1
4   lab:    if y − j ≥ 0 then
5               z := z ∗ x
6               j := j + 1
7               goto lab
8           return(z)
9   }
```

Example: symbolic execution of *power*(α_1, α_2)

#	*j*	*x*	*y*	*z*	*pc*
1	?	α_1	α_2	?	*true*
2	-	-	-	1	-
3	1	-	-	-	-

Example: symbolic execution of *power*(α_1, α_2)

#	*j*	*x*	*y*	*z*	*pc*
1	?	α_1	α_2	?	*true*
2	-	-	-	1	-
3	1	-	-	-	-
4	neither *true* $\Longrightarrow$ $\alpha_2 - 1 \geq 0$				
	nor *true* $\Longrightarrow$ $\neg(\alpha_2 - 1 \geq 0)$				
	is valid $\rightarrow$ fork				

Example: symbolic execution of *power*(α_1, α_2)

#	*j*	*x*	*y*	*z*	*pc*
1	?	α_1	α_2	?	*true*
2	-	-	-	1	-
3	1	-	-	-	-
4	neither *true* $\Longrightarrow$ $\alpha_2 - 1 \geq 0$ nor *true* $\Longrightarrow$ $\neg(\alpha_2 - 1 \geq 0)$ is valid $\rightarrow$ fork				
4	1	α_1	α_2	1	$\alpha_2 \geq 1$
5	-	-	-	α_1	-
6	2	-	-	-	-
7	-	-	-	-	-

#	*j*	*x*	*y*	*z*	*pc*
4	1	α_1	α_2	1	$\neg(\alpha_2 \geq 1)$
8	done, returns 1 when $\alpha_2 < 1$				

Example: symbolic execution of *power*(α_1, α_2)

#	*j*	*x*	*y*	*z*	*pc*
1	?	α_1	α_2	?	*true*
2	-	-	-	1	-
3	1	-	-	-	-
4	neither *true* $\Longrightarrow$ $\alpha_2 - 1 \geq 0$ nor *true* $\Longrightarrow$ $\neg(\alpha_2 - 1 \geq 0)$ is valid $\rightarrow$ fork				
4	1	α_1	α_2	1	$\alpha_2 \geq 1$
5	-	-	-	α_1	-
6	2	-	-	-	-
7	-	-	-	-	-
4	neither $\alpha_2 \geq 1 \Longrightarrow \alpha_2 \geq 2$ nor $\alpha_2 \geq 1 \Longrightarrow \neg(\alpha_2 \geq 2)$ is valid $\rightarrow$ fork				

#	*j*	*x*	*y*	*z*	*pc*
4	1	α_1	α_2	1	$\neg(\alpha_2 \geq 1)$
8	done, returns 1 when $\alpha_2 < 1$				

Example: symbolic execution of *power*(α_1, α_2)

#	*j*	*x*	*y*	*z*	*pc*
1	?	α_1	α_2	?	*true*
2	-	-	-	1	-
3	1	-	-	-	-
4	neither *true* $\Longrightarrow$ $\alpha_2 - 1 \geq 0$ nor *true* $\Longrightarrow$ $\neg(\alpha_2 - 1 \geq 0)$ is valid $\rightarrow$ fork				
4	1	α_1	α_2	1	$\alpha_2 \geq 1$
5	-	-	-	α_1	-
6	2	-	-	-	-
7	-	-	-	-	-
4	neither $\alpha_2 \geq 1 \Longrightarrow \alpha_2 \geq 2$ nor $\alpha_2 \geq 1 \Longrightarrow \neg(\alpha_2 \geq 2)$ is valid $\rightarrow$ fork				
4	2	α_1	α_2	α_1	$\alpha_2 \geq 1 \wedge \alpha_2 \geq 2$
5			⋮		

#	*j*	*x*	*y*	*z*	*pc*
4	1	α_1	α_2	1	$\neg(\alpha_2 \geq 1)$
8	done, returns 1 when $\alpha_2 < 1$				

#	*j*	*x*	*y*	*z*	*pc*
4	2	α_1	α_2	α_1	$\alpha_2 \geq 1 \wedge \neg(\alpha_2 \geq 2)$
8	done, returns α_1 when $\alpha_2 = 1$				

Path condition is always satisfiable

Clearly, every path condition corresponds exactly to one execution path and vice versa.

Theorem

At each point of every symbolic execution $pc \not\equiv$ *false.*

Proof: Initially, *pc* is set to *true*. Further, *pc* is modified only at forking executions, using assignments of the form $pc := pc \wedge q$ and $pc := pc \wedge \neg q$.

Forking execution implies that $pc \implies \neg q$ is not valid. Hence, $\neg(pc \implies \neg q)$ is satisfiable. As $pc \wedge q \equiv \neg(pc \implies \neg q)$, $pc \wedge q$ is also satisfiable.

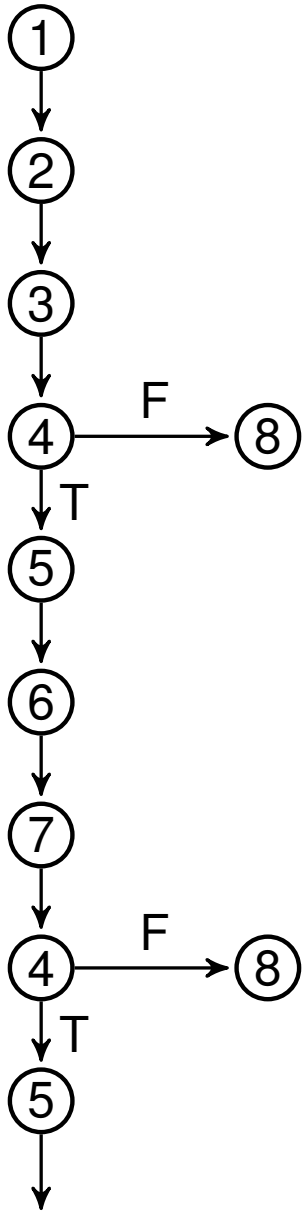
The case $pc \wedge \neg q$ is similar. □

Symbolic execution tree

The execution paths followed during the symbolic execution of a procedure can be expressed by symbolic execution tree.

- executed statement = a node labeled with the statement number
- transition between executed statements = a directed arc connecting the corresponding nodes
- for each forking `IF` statement execution there are two outgoing arcs labeled with `T` and `F` for `THEN` and `ELSE` branch, respectively

Symbolic execution tree for *power*(α_1, α_2)



Symbolic execution tree

Lemma

For each terminal leaf in the tree there does exist particular non-symbolic input, which will trace the same path.

Proof: Every input satisfying the corresponding *pc* trace the same path. As *pc* is always satisfiable, there exists such an input. □

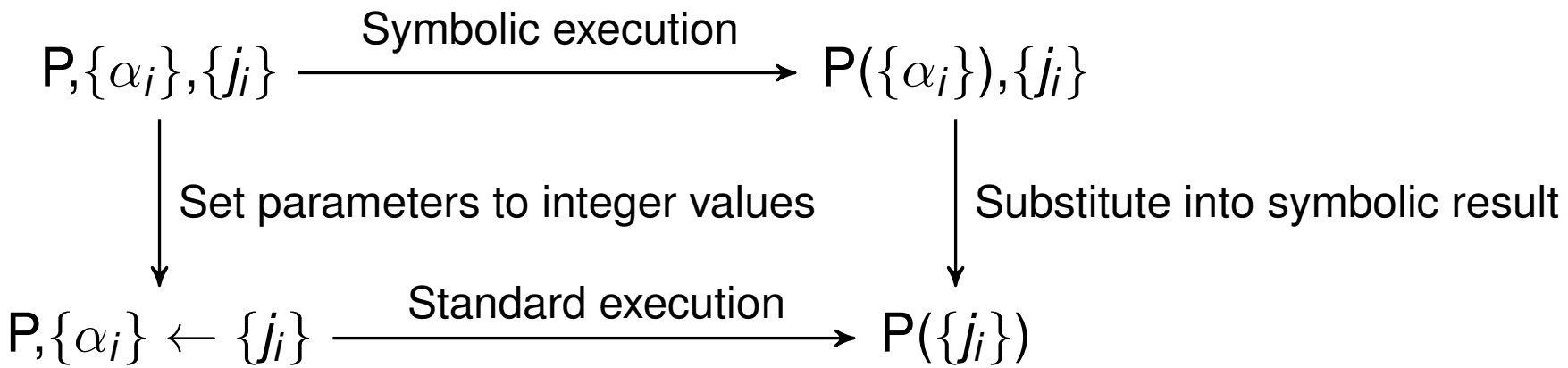
Lemma

Path conditions associated with any two terminal leaves are distinct, i.e. $pc_1 \wedge pc_2 \equiv \mathit{false}$.

Proof: The two paths leading from the root to two different terminal nodes have a unique forking node where the paths diverge. At that forking node some q was added to one while $\neg q$ to the other. Since $q \wedge \neg q \equiv \mathit{false}$, the lemma holds. □

Commutativity

If one normally executes a program with a specific set of integers $\{j_i\}$, the result will be the same as executing it symbolically (using a set of $\{\alpha_i\}$) and then instantiating the symbolic results, i.e. assigning $\{j_i\}$ to $\{\alpha_i\}$.



Applications in verification

Programs can be enriched with `ASSUME`(φ) and `ASSERT`(φ) statements. When symbolic execution passes through

- `ASSUME`(φ), it executes $pc := pc \wedge \varphi$.
- `ASSERT`(φ) and $pc \implies \varphi$ is not valid, it reports an error.

With these constructs, symbolic execution can be used with a modification of Floyd's proof method to prove program correctness.

This application is straightforward for any program whose symbolic execution tree is finite.

Practical issues

- The symbolic execution of `IF` and `ASSERT` statements requires to decide validity of some implications. Unfortunately, even for simple programming languages (including our simple language) it is impossible to build theorem prover that will decide validity of such implications.
- The conflict between discrete arithmetics of computer and continuous nature of real numbers with infinite precision is an issue.
- Variable storage referencing problem: Let expression $A(I)$ references some element in array A. When the value of I is a symbolic expression, the particular element being referenced is a function of the program input.
  - Unsatisfactory solution: Let $v(I)$ be symbolic value of I. Then we might resolve the reference $A(I)$ with $ITE(v(I) = 1, v(A(1)), ITE(v(I) = 2, v(A(2)), \ldots) \ldots)$

Practical issues (cont.)

- Poor performance of symbolic execution on cycles without fixed number of iterations (symbolic execution forks again and again) $\rightarrow$ path explosion.
- Symbolic execution cannot be precise in practice due to pointer manipulation, complex arithmetic operations (e.g. in hashing, encryption or decryption), calls to operating system and libraries
- Some of these issues and non-existence of a theorem prover can be partially solved by concolic execution.

Concolic execution

- concolic = concrete + symbolic
- program is executed on a real input and on symbolic input simultaneously
- symbolic execution does not fork, it always follows the concrete execution and computes *pc*
- if a symbolic value is not available, we can switch to a concrete one

Real applications

- Symbolic execution is more exploitable in program testing then in program verification.
- Typical applications: bug finding, test generation and analysis of abstract error traces.
- Often combined with other techniques.
- Used in many tools including KLEE, PEX, SAGE, SLAM, etc.

Bonus

Automated whitebox fuzz testing

Automated whitebox fuzz testing

- an example of modern and sophisticated testing method
- implemented in SAGE (Scalable, Automated, Guided Execution)
- discovered 30+ new bugs in large-shipped (i.e. intensively tested) file-reading Windows applications including image processors, media players, file decoders
- combines fuzz testing and symbolic execution in a better way than whitebox dynamic test generation

Fuzz testing

- a form of blackbox random testing
- randomly mutates well-formed input and test the program on resulting data
- popular since the Month of Browser Bugs (July 2006)

Whitebox dynamic test generation

- the process follows this scenario
  1. test the program with a given correct input
  2. symbolically execute the discovered execution path
  3. use path condition to generate an input that changes the evaluation of a condition on the execution path
  4. test the program with the new input
  5. go to the step 2
- the condition of step 3 can be selected in depth-first search or breath-first search manner
- the problem is that the symbolic executions are extremely expensive

Automated whitebox fuzz testing

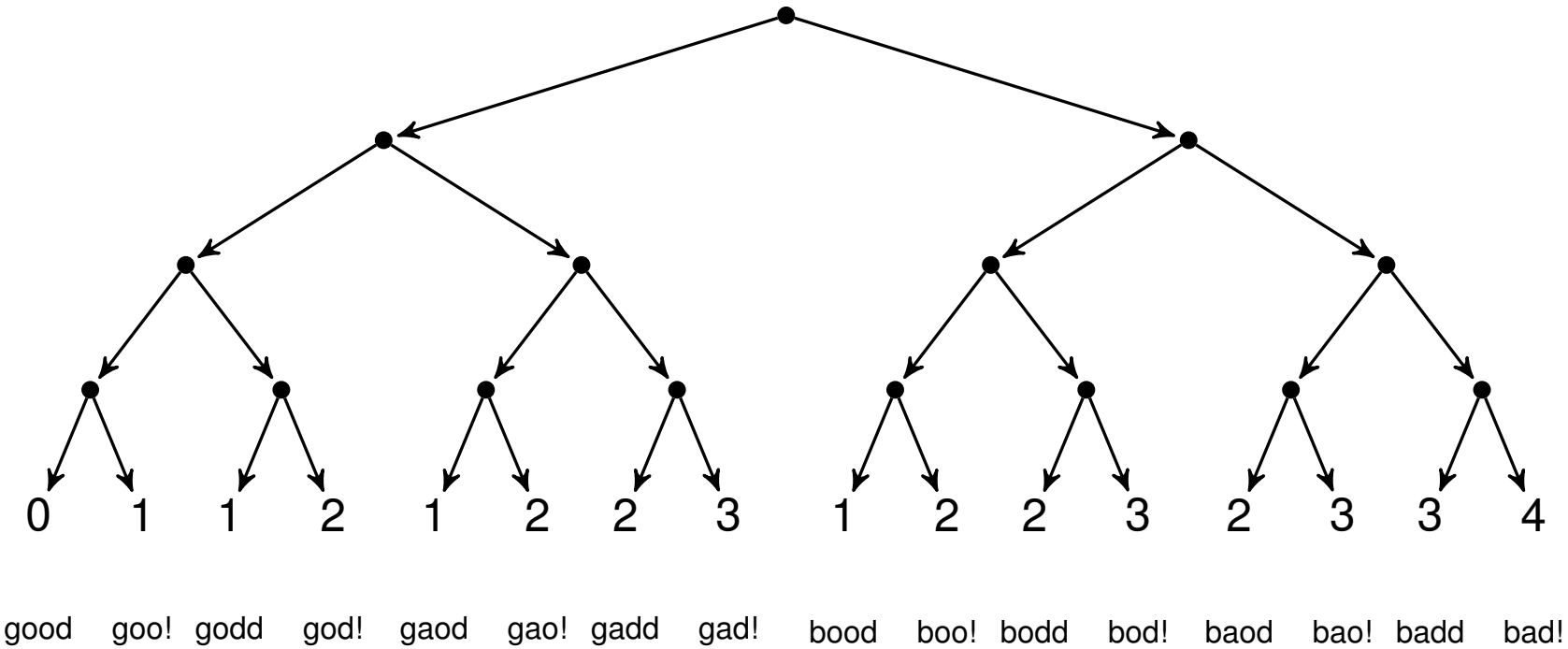
- tries to generate as many new inputs from one symbolic execution as possible
- input for the next iteration is selected by some scoring function applied to all generated inputs
- in particular, inputs exploring the biggest (so-far uncovered) pieces of code are chosen for the next symbolic execution
- in this way, SAGE can well recover from divergencies (situations when an execution deviates from the assumed execution path)

Automated whitebox fuzz testing: example

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) abort();  // error
}
```

Symbolic execution tree for *power*(α_1, α_2)



Coming next week

Abstract interpretation + static analysis

- Another standard approach.
- Applicable to large software projects, e.g. Linux kernel.
- What can one learn about a program without executing it?