Higher-Order Test Generation

Patrice Godefroid

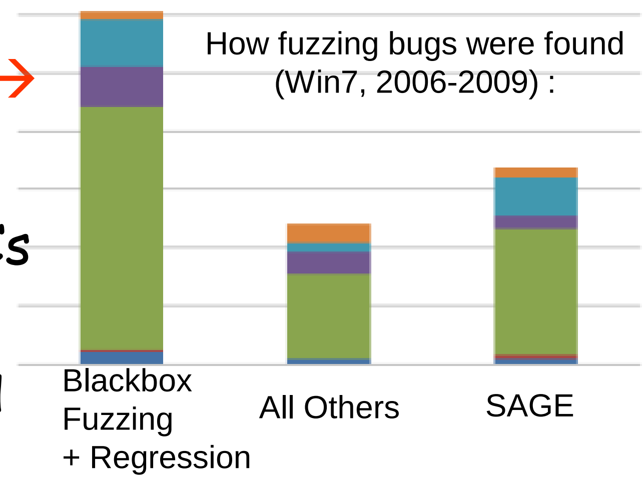
Microsoft Research

Test Generation is Big Business

- #1 application for SMT solvers today (CPU usage)
- SAGE @ Microsoft:
  - 1st whitebox fuzzer for security testing
  - 200+ machine years (since 2008) →
  - 200+ million 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



Test Generation: How?

- Most precise: dynamic test generation
  - Dynamic symbolic execution to collect constraints on inputs
  - Negate those, solve new constraints to get new tests
  - Repeat → systematic state-space exploration (= DART)

```
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 = '!'

SMT solver → SAT

bood

gaod

godd

goo!

Implemented in SAGE

Optimized for large x86 trace analysis (ex: Excel)

Problem: Symbolic Reasoning is Imprecise

- For large complex programs (pointer manipulations, complex arithmetic, calls to OS/library functions, etc.)
- Imprecision forces approximation
- How? (1) Static test generation ([King76,…])
  - Static analysis to partition the program's input space
  - Ineffective whenever precise symbolic reasoning is not possible

Example:

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

Can't statically generate values for x and y that satisfy "x==hash(y)" !

How? (2) Dynamic Test Generation

Example:

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

Run 1 : - start with (random) x=33, y=42

- execute concretely and symbolically:

  if (33 != 567) | if $\boxed{\text{(x != hash(y))}}$

  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 !

Observations:

- "Unknown/complex symbolic expressions can be simplified using concrete runtime values" [DART, PLDI'05]
- Let's call this step "concretization" (ex: hash(y) → 567)
- Dynamic test generation extends static test generation with additional runtime information: it is more powerful

How often? When exactly? Why? → this work!

Unsound and Sound Concretization

- Concretization is not always sound

```
int foo(int x, int y) {
  if (x==hash(y)) {
     if (y==10) error();
  } …
}
```

Run: x=567, y=42
pc: x==567 and y!=10
New pc: x==567 and y==10
New inputs: x=567, y=10
Divergence!

pc and new pc are unsound !

- Definition: A path constraint pc for a path w is sound if every input satisfying pc defines an execution following w

- Sound concretization: add concretization constraints

  pc: y==42 and x==567 and y!=10 (sound)
  New pc: y==42 and x==567 and y==10 (sound)

- Theorem: path constraint is now always sound. Is this better? No
  - Forces us to detect all sources of imprecision (expensive/impossible…)
  - Can prevent test generation and “good” divergences

Idea: Using Uninterpreted Functions

- Modeling imprecision with uninterpreted functions

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

Run: x=33, y=42

pc: x != h(y)

New pc: x == h(y)

- How to generate tests?
  - Is $(\exists x,y,h:)\ x=h(y)$ SAT? Yes, but so what? (ex: x=y=0, h(0)=0)
  - Need universal quantification !

  $(\forall h:)\ \exists x,y: x=h(y)$ is this first-order logic formula valid?

  Yes. Solution (strategy): "fix y, set x to the value of h(y)"

- Test generation from validity proofs ! (not SAT models)
  - Necessary but not sufficient: what "value of h(y)"?

Need for Uninterpreted Function Samples

- Record I/O UF samples

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

Run: x=33, y=42

Record: 567 == h(42)

pc: x!=h(y)

- Use UF samples to interpret a validity proof/strategy
  - "fix y, set x to the value of h(y)" → set y=42, x=567
- Or new pc: $(\forall h:)\ \exists x,y: (567=h(42)) \Rightarrow (x=h(y))$ is valid?
- Higher-order test generation =
  - models imprecision using Uninterpreted Functions
  - records UF samples as concrete input/output value pairs
  - generates tests from validity proofs of FOL formulas

Key: a "higher-order" logic representation of path constraints

Higher-Order Test Generation is Powerful

- Theorem: HOTG is as powerful as sound concretization
  - Can simulate it (both UFs and UF samples are needed for this)

- Higher-Order Test Generation is more powerful

Ex 1: $(\forall h:)\ \exists x,y: h(x)=h(y)$ is valid (solution: set x=y)

Ex 2: $(\forall h:)\ \exists x,y: h(x)=h(y)+1$ is invalid

But $(\forall h:)\ \exists x,y: (h(0)=0 \wedge h(1)=1) \Rightarrow h(x)=h(y)+1$ is valid

(solution: set x=1, y=0)

Ex 3:

```
int foo(int x, int y) {
  if (x==hash(y)) {
     if (y==10) error();
  } …
}
```

Run: x=567, y=42

pc: x==h(y) and y!=10

New pc: $(\forall h:)\exists x,y: (h(42)=567) \Rightarrow x=h(y) \wedge y=10$

is valid. Solution: set y=10, set x=h(10)

2-step test generation:

- run1 with y=10, x=567 to learn h(10) =66
- run2 with y=10, x=66 !

Implementability Issues

- Tracking all sources of imprecision is problematic
  - Excel on a 45K input bytes executes 1 billion x86 instructions
- Imprecision cannot always be represented by UFs
  - Unknown input/output signatures, nondeterminism,…
- Capturing all input/output pairs can be very expensive
- Limited support from current SMT solvers
  - $\exists X{:}\Phi(F,X)$ is valid iff $\forall X{:}\neg\Phi(F,X)$ is UNSAT
  - little support for generating+parsing UNSAT 'saturation' proofs
- In practice, HOTG can be used for targeted reasoning about specific user-defined complex/unknown functions

Application: Lexers with Hash Functions

- Parsers with input lexers using hash functions for fast keyword recognition

```
Initially, forall language keywords: addsym(keyword, hashtable)
When parsing the input:
X=findsym(inputChunk, hashtable); // is inputChunk in hashtable?
if (x==52)  … // how to get here?
```

- With higher-order test generation:
  - Represent hashfunct by one UF h
  - Capture all pairs (hashvalue,h(keyword))
  - If "h(inputChunk)==52" and "(52,h('while'))" -> inputChunk='while'
  - This effectively inverses hashfunct only for all keywords
  - Sufficient to drive executions through the lexer !

Other Related Work

- Modeling imprecision with UFs is well-known in program verification of universal properties
  - "may" abstractions, universal quantifiers only, validity checks
  - Here, novelty is for existential properties
- Test generation is only one way to verify existential properties of programs
  - More generally, one can build "must" abstractions
  - Alternation $\forall\exists$ is also used then
- Test generation as a game is not new
  - in model-based testing, testing for reactive systems, etc.
  - but from validity proofs of FOL formulas with UFs *is* new

Conclusions

- Higher-order test generation = UFs + UF samples + test generation from validity checks of FOL formulas
- A new powerful form of test generation
- Tracking all sources of incompleteness is unrealistic, targeted use of UFs is more practical (ex: lexer with h)
- A formal tool to define the limits of test generation
- Explains in what sense dynamic test generation is more powerful than static test generation
  - only in its ability to record concrete values in path constraints
  - concrete values are ultimately needed for test generation