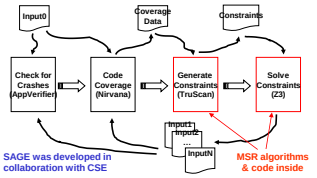
Lecture 6:

What's Next?
Compositional Testing and Verification

Patrice Godefroid

Microsoft Research

SAGE: Automated Whitebox Security Testing

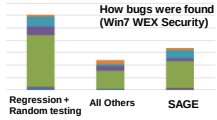


Basic idea:

1. Run the program with first inputs,
2. gather constraints on inputs at conditional statements,
3. use a constraint solver to generate new test inputs,
4. repeat - possibly forever!

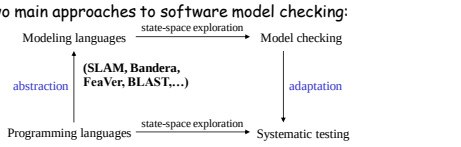
Results: since 1st internal release (April 2007)
many new security-critical bugs found! (number: confidential)
(would trigger MS security bulletins if known outside MS)
SAGE is now used daily in Windows, Office, DevDiv, etc.

Example: WEX Security team for Win7
Centralized security testing for Win7 WEX (=Windows client)
SAGE running 24/7 for 1+ year on (avg.) 100+ machines
~1/3 of all Win7 WEX security bugs found by SAGE !



What Next? Towards "Verification"

- When can we safely stop testing?
  - When we know that there are no more bugs ! = "Verification"
  - "Testing can only prove the existence of bugs, not their absence." [Dijkstra]
  - Unless it is exhaustive! This is the "model checking thesis"
  - "Model Checking" = exhaustive testing (state-space exploration)
  - Two main approaches to software model checking:



Concurrency: **VeriSoft, JPF, CMC, Bogor, CHESS,…**
Data inputs: **DART, EXE, SAGE,…**

Exhaustive Testing ?

- Model checking is always "up to some bound"
  - Limited (often finite) input domain, for specific properties, under some environment assumptions
    - Ex: exhaustive testing of Win7 JPEG parser up to 1,000 input bytes
      - 8000 bits → 2^8000 possibilities → if 1 test per sec, 2^8000 secs
      - FYI, 15 billion years = 473040000000000000 secs = 2^60 secs!
      - → MUST be "symbolic" ! ☺ How far can we go?
- Practical goals: (easier?)
  - Eradicate all remaining buffer overflows in all Windows parsers
  - Better coverage guarantees to justify "no new bug found"
  - Reduce costs & risks for Microsoft: when to stop fuzzing?
  - Increase costs & risks for Black Hats !
    - Many have probably moved to greener pastures already… (Ex: Adobe)
    - Ex: <5 security bulletins in all the SAGE-cleaned Win7 parsers
    - If noone can find bugs in P, P is observationally equivalent to "verified"!

How to Get There?

1. Identify and patch holes in symbolic execution + constraint solving
2. Tackle "path explosion" with compositional testing and symbolic test summaries [POPL'07,TACAS'08,POPL'10]

→ Fuzzing in the (Virtual) Cloud (Sagan)

- New centralized server collecting stats from all SAGE runs !
- Track results (bugs, concrete & symbolic test coverage), incompleteness (unhandled tainted x86 instructions, Z3 timeouts, divergences, etc.)
- Help troubleshooting (SAGE has 100+ options…)
- Tell us what works and what does not

The Art of Constraint Generation

- Static analysis: abstract away "irrelevant" details
  - Good for focused search, can be combined with DART (Ex: [POPL'10])
  - But for bit-precise analysis of low-level code (function pointers, in-lined assembly,…) ? In a non-property-guided setting? Open problem…
- Bit-precise VC-gen: statically generate 1 formula from a program
  - Good to prove complex properties of small programs (units)
  - Does not scale (huge formula encodings), asks too much of the user
- SAT/SMT-based "Bounded Model Checking": stripped-down VC-gen
  - Emphasis on automation
  - Unrolling all loops is naïve, does not scale
- "DART": the only option today for large programs (Ex: Excel)
  - Path-by-path exploration is naïve, but "whitebox fuzzing" can scale it to large executions (Z3 is not the bottleneck) + zero false alarms !
  - But suffers from "path explosion"…

DART is Beautiful

- Generates formulas where the only "free" symbolic variables are whole-program inputs
  - When generating tests, one can only control inputs !
- Strength: scalability to large programs
  - Only tracks "direct" input dependencies (i.e., tests on inputs); the rest of the execution is handled with the best constant-propagation engine ever: running the code on the computer !
  - (The size of) path constraints only depend on (the number of) program tests on inputs, not on the size of the program

  = the right metric: complexity only depends on nondeterminism!
- Price to pay: "path explosion" [POPL'07]
  - Solution = symbolic test summaries

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"

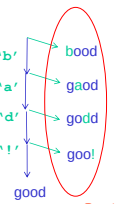
Path constraint:

I_0!='b' → I_0='b'

I_1!='a' → I_1='a'

I_2!='d' → I_2='d'

I_3!='!' → I_3='!'



Negate each constraint in path constraint

Solve new constraint → new input

Compositionality = Key to Scalability

- Idea: compositional dynamic test generation [POPL'07]
  - use summaries of individual functions (or program blocks, etc.)
    - like in interprocedural static analysis
    - but here "must" formulas generated dynamically
  - If f calls g, test g, summarize the results, and use g's summary when testing f
  - A summary φ(g) is a disjunction of path constraints expressed in terms of g's input preconditions and g's output postconditions:

    $\varphi(g) = \vee \varphi(w)$ with $\varphi(w) = pre(w) \wedge post(w)$
  - g's outputs are treated as fresh symbolic inputs to f, all bound to prior inputs and can be "eliminated" (for test generation)
- Can provide same path coverage exponentially faster !
  - See details and refinements in [POPL'07,TACAS'08,POPL'10]

Example

```
int is_positive(int x) {
  if (x>0) return 1;
  return 0;
}
#define N 100
void top(int s[N]) {//N inputs
  int i,cnt=0;
  for (i=0;i<N;i++)
    cnt=cnt+is_positive(s[i]);
  if (cnt == 3) error(); //(*)
  return;
}
```

Program P={top,is_positive} has 2^N feasible whole-program paths

DART will perform 2^N runs

SMART will perform only 4 runs !

• 2 to compute the summary

$\Phi = (x>0 \wedge ret=1) \vee (x=<0 \wedge ret=0)$

for function is_positive()

• 2 to execute both branches of (*), by solving the constraint

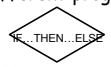
$[(s[0]>0 \wedge ret_0=1) \vee (s[0]=<0 \wedge ret_0=0)]$
$\wedge [(s[1]>0 \wedge ret_1=1) \vee (s[1]=<0 \wedge ret_1=0)]$
$\wedge \ldots \wedge [(s[N-1]>0 \wedge ret_{N-1}=1) \vee (s[N-1]=<0 \wedge ret_{N-1}=0)]$
$\wedge (ret_0+ret_1+\ldots+ret_{N-1} = 3)$

The Engineering of Test Summaries

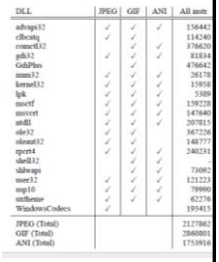
- Systematically summarizing everywhere is foolish
  - Very expensive and not necessary (costs outweigh benefits)
  - Don't fall into the "VC-gen or BMC traps" ! ☺
- Summarization on-demand: (100% algorithmic)
  - When? At search bottlenecks (with dynamic feedback loop)
  - Where? At simple interfaces (with simple data types)
  - How? With limited side-effects (to be manageable and "sound")
- Goal: use summaries intelligently
  - THE KEY to scalable bit-precise whole-program analysis ?
    - It is necessary! But in what form(s)? Is it sufficient?
    - Stay tuned…

Summaries Cure Search Redundancy

- Across different program paths

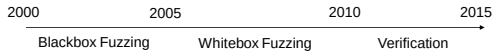


- Across different program versions
  - ["Incremental Compositional Dynamic Test Generation", with S. Lahiri and C. Rubio-Gonzalez, MSR TR, Feb 2010]
- Across different applications →
- Summaries avoid unnecessary work
- What if central server of summaries for all code?… Sagan 2.0



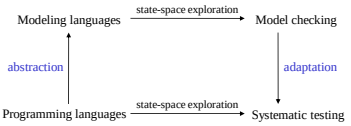
Conclusion: Towards Verification

- Tracking all(?) sources of incompleteness
- Summaries (on-demand…) against path explosion
- How far can we go?
  - Reduce costs & risks for Microsoft: when to stop fuzzing?
  - Increase costs & risks for Black Hats (goal already achieved?)
- For history books:



Blackbox Fuzzing | Whitebox Fuzzing | Verification

Conclusion: Software Model Checking



- Several independent dimensions:
  - May vs. Must (universal vs. existential)
  - Static vs. Dynamic (but what's the difference really?)
  - Proofs vs. Bugs (verification vs. testing)
- Dijkstra vs. Model Checking
  - "Testing can only prove the existence of bugs, not their absence."
  - Unless it is exhaustive! This is the "model checking thesis"
  - In practice, verification is not binary: it is a *continuum*