Software Model Checking Improving Security of a Billion Computers

Patrice Godefroid

Microsoft Research

Acknowledgments

- Joint work with Michael Levin (CSE) and others:
  - Chris Marsh, Lei Fang, Stuart de Jong (CSE)
  - interns Dennis Jeffries (06), David Molnar (07), Adam Kiezun (07), Bassem Elkarablieh (08), …
- Thanks to the entire SAGE team and users !
  - MSR: Ella Bounimova,…
  - Z3: Nikolaj Bjorner, Leonardo de Moura,…
  - WEX (Windows): Nick Bartmon, Eric Douglas,…
  - Office: Tom Gallagher, Octavian Timofte,…
  - SAGE users all across Microsoft!

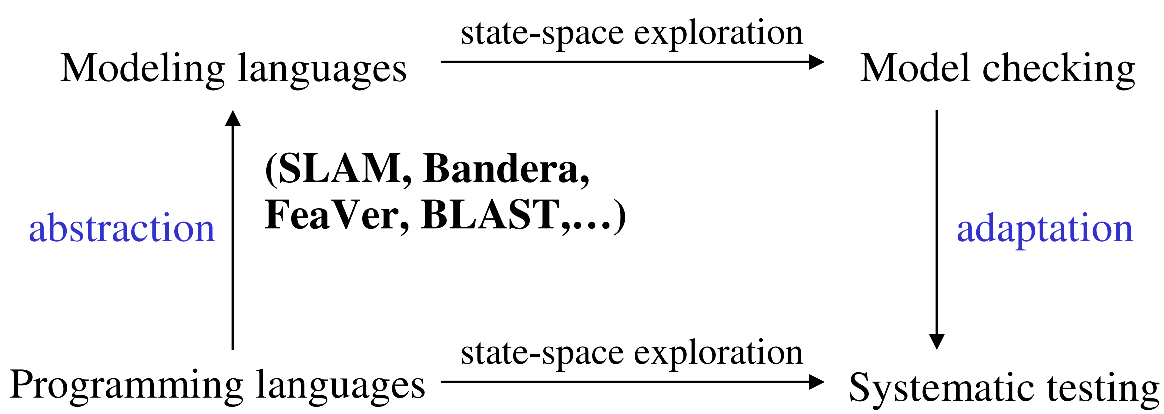
References

- see http://research.microsoft.com/users/pg
  - DART: Directed Automated Random Testing, with N. Klarlund and K. Sen, PLDI'2005
  - Compositional Dynamic Test Generation, POPL'2007
  - Automated Whitebox Fuzz Testing, with M. Levin and D. Molnar, NDSS'2008
  - Demand-Driven Compositional Symbolic Execution, with S. Anand and N. Tillmann, TACAS'2008
  - Grammar-Based Whitebox Fuzzing, with A. Kiezun and M. Levin, PLDI'2008
  - Active Property Checking, with M. Levin and D. Molnar, EMSOFT'2008
  - Precise Pointer Reasoning for Dynamic Test Generation, with B. Elkarablieh and M. Levin, ISSTA'2009

A Brief History of Software Model Checking

- How to apply model checking to analyze **software**?
  - "Real" programming languages (e.g., C, C++, Java),
  - "Real" size (e.g., 100,000's lines of code).

- Two main approaches to software model checking:



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

Killer app: security   biggest impact to date!

Security is Critical (to Microsoft)

- Software security bugs can be very expensive:
  - Cost of each Microsoft Security Bulletin: $Millions
  - Cost due to worms (Slammer, CodeRed, Blaster, etc.): $Billions
- Most security exploits are initiated via files or packets
  - Ex: Internet Explorer parses dozens of file formats
- Security testing: "hunting for million-dollar bugs"
  - Write A/V (always exploitable), Read A/V (sometimes exploitable), NULL-pointer dereference, division-by-zero (harder to exploit but still DOS attacks), etc.

Hunting for Security Bugs

- Main techniques used by "black hats":
  - Code inspection (of binaries) and
  - Blackbox fuzz testing
- Blackbox fuzz testing:
  - A form of blackbox random testing [Miller+90]
  - Randomly fuzz (=modify) a well-formed input
  - Grammar-based fuzzing: rules that encode "well-formed"ness + heuristics about how to fuzz (e.g., using probabilistic weights)
- **Heavily** used in security testing
  - Ex: July 2006 "Month of Browser Bugs"
  - Simple yet effective: many bugs found this way…
  - At Microsoft, fuzzing is mandated by the SDL

Blackbox Fuzzing

- Examples: Peach, Protos, Spike, Autodafe, etc.
- Why so many blackbox fuzzers?
  - Because anyone can write (a simple) one in a week-end!
  - Conceptually simple, yet effective…
- Sophistication is in the "add-on"
  - Test harnesses (e.g., for packet fuzzing)
  - Grammars (for specific input formats)
- Note: usually, no principled "spec-based" test generation
  - No attempt to cover each state/rule in the grammar
  - When probabilities, no global optimization (simply random walks)

Introducing Whitebox Fuzzing

- Idea: mix fuzz testing with dynamic test generation
  - Symbolic execution
  - Collect constraints on inputs
  - Negate those, solve with constraint solver, generate new inputs
  - do "systematic dynamic test generation" (=DART)
- Whitebox Fuzzing = "DART meets Fuzz"

  Two Parts:

  1. Foundation: DART (Directed Automated Random Testing)
  2. Key extensions ("Whitebox Fuzzing"), implemented in SAGE

Automatic Code-Driven Test Generation

Problem:

Given a sequential program with a set of input parameters, generate a set of inputs that maximizes code coverage

= "automate test generation using program analysis"

This is not "model-based testing"
(= generate tests from an FSM spec)

How? (1) Static Test Generation

- Static analysis to partition the program's input space [King76,…]

- Ineffective whenever symbolic reasoning is not possible
  - which is frequent in practice… (pointer manipulations, complex arithmetic, calls to complex OS or library functions, etc.)

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

- Run the program (starting with some random inputs), gather constraints on inputs at conditional statements, use a constraint solver to generate new test inputs

- Repeat until a specific program statement is reached [Korel90,…]

- Or repeat to try to cover ALL feasible program paths: DART = Directed Automated Random Testing = systematic dynamic test generation [PLDI'05,…]
  - detect crashes, assertion violations, use runtime checkers (Purify,…)

DART = Directed Automated Random Testing

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 (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:
  - Dynamic test generation extends static test generation with additional runtime information: it is more powerful
  - The number of program paths can be infinite: may not terminate!
  - Still, DART works well for small programs (1,000s LOC)
  - Significantly improves code coverage vs. random testing

DART Implementations

- Defined by symbolic execution, constraint generation and solving
  - Languages: C, Java, x86, .NET,…
  - Theories: linear arith., bit-vectors, arrays, uninterpreted functions,…
  - Solvers: lp_solve, CVCLite, STP, Disolver, Z3,…
- Examples of tools/systems implementing DART:
  - EXE/EGT (Stanford): independent ['05-'06] closely related work
  - CUTE = same as first DART implementation done at Bell Labs
  - SAGE (CSE/MSR) for x86 binaries and merges it with "fuzz" testing for finding security bugs (more later)
  - PEX (MSR) for .NET binaries in conjunction with "parameterized-unit tests" for unit testing of .NET programs
  - YOGI (MSR) for checking the feasibility of program paths generated statically using a SLAM-like tool
  - Vigilante (MSR) for generating worm filters
  - BitScope (CMU/Berkeley) for malware analysis
  - CatchConv (Berkeley) focus on integer overflows
  - Splat (UCLA) focus on fast detection of buffer overflows
  - Apollo (MIT) for testing web applications

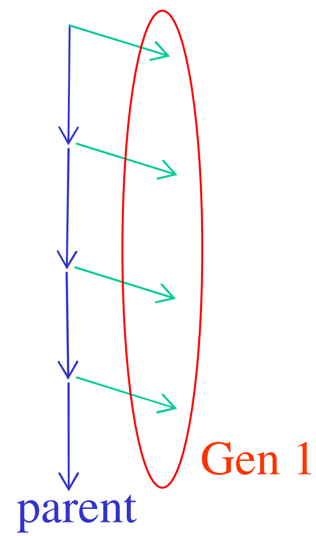
…and more!

DART Summary

- DART attempts to exercise all paths (like model checking)
  - Covering a single specific assertion (verification): hard problem (often intractable)
  - Maximize path coverage while checking thousands of assertions all over: easier problem (optimization, best-effort, tractable)
  - Better coverage than pure random testing (with directed search)
- DART can work around limitations of symbolic execution
  - Symbolic execution is an adjunct to concrete execution
  - Concrete values are used to simplify unmanageable symbolic expressions
  - Randomization helps where automated reasoning is difficult
- Comparison with static analysis:
  - No false alarms (more precise) but may not terminate (less coverage)
  - "Dualizes" static analysis: static may vs. DART must
    - Whenever symbolic exec is too hard, under-approx with concrete values
    - If symbolic execution is perfect, no approx needed: both coincide!

Whitebox Fuzzing [NDSS'08]

- Whitebox Fuzzing = "DART meets Fuzz"
- Apply DART to large applications (not unit)
- 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 !

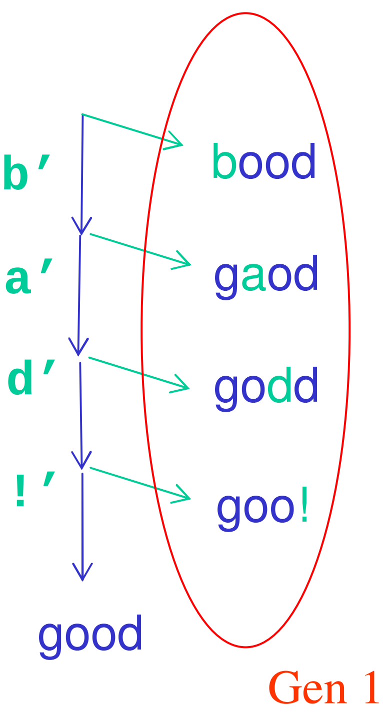
Example

```
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='!'

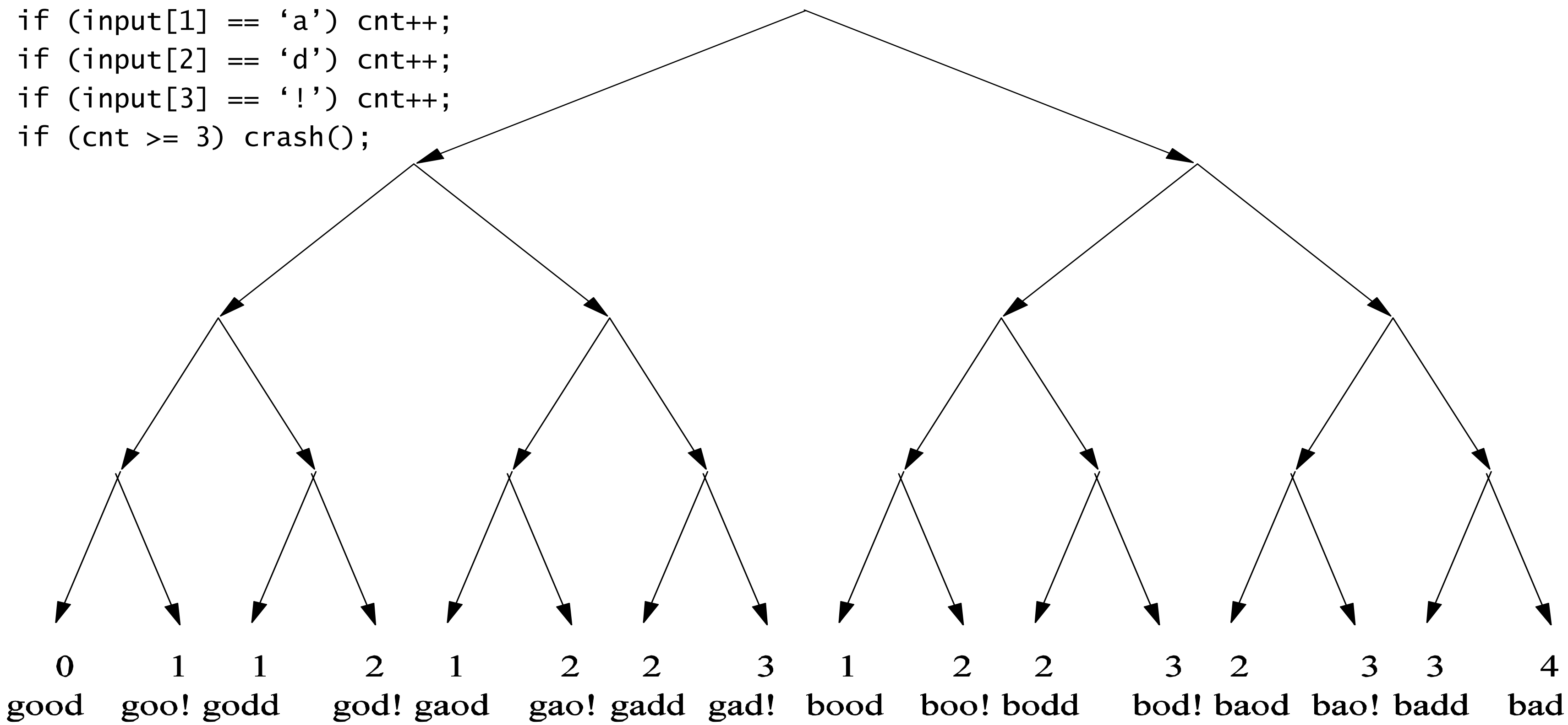


Negate each constraint in path constraint

Solve new constraint → new input

The Search Space

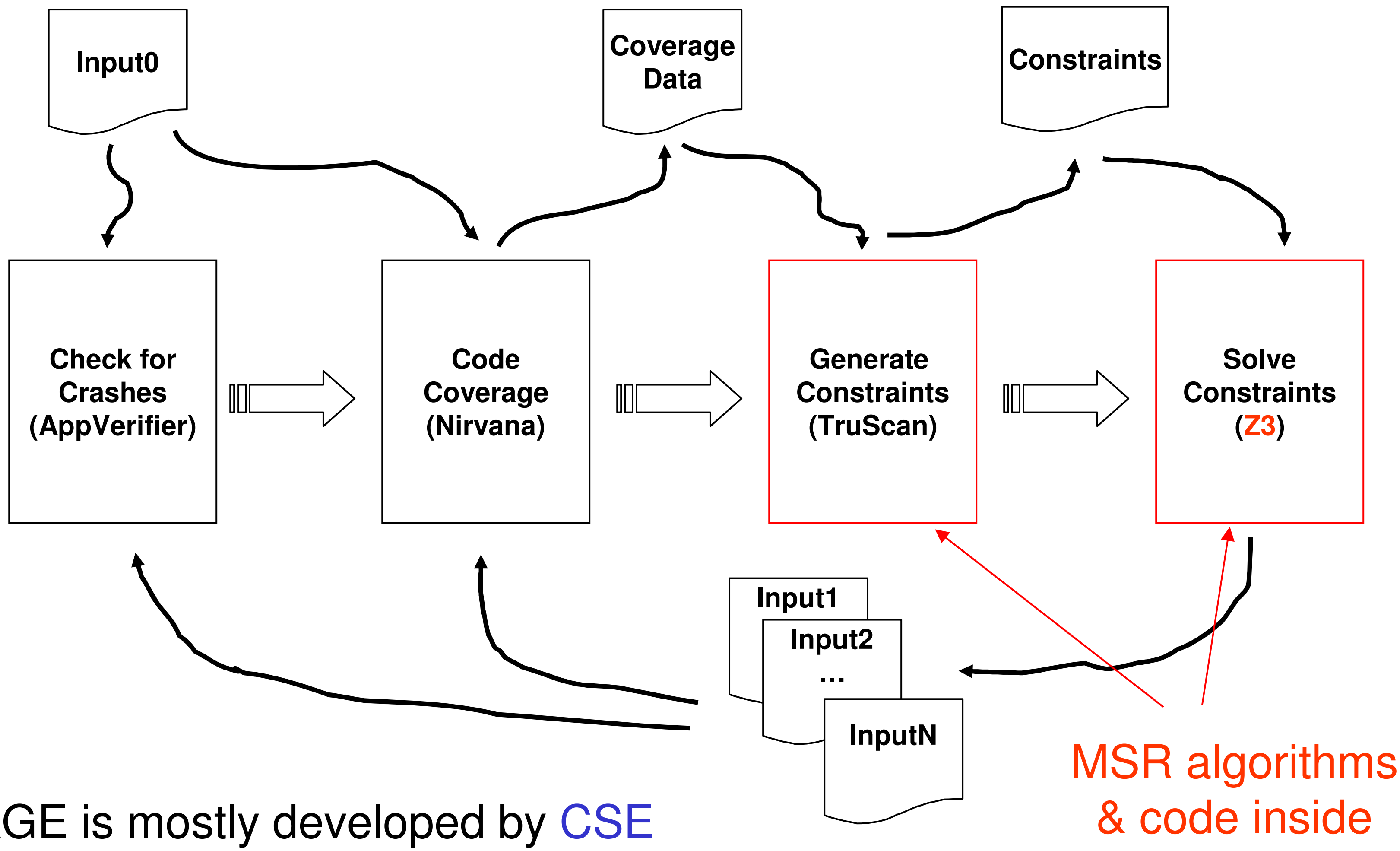
```
void top(char input[4])
{
   int cnt = 0;
   if (input[0] == 'b') cnt++;
   if (input[1] == 'a') cnt++;
   if (input[2] == 'd') cnt++;
   if (input[3] == '!') cnt++;
   if (cnt >= 3) crash();
}
```



SAGE (Scalable Automated Guided Execution)

- Generational search introduced in SAGE
- Performs symbolic execution of x86 execution traces
  - Builds on Nirvana, iDNA and TruScan for x86 analysis
  - Don't care about language or build process
  - Easy to test new applications, no interference possible
- Can analyse any file-reading Windows applications
- Several optimizations to handle huge execution traces
  - Constraint caching and common subexpression elimination
  - Unrelated constraint optimization
  - Constraint subsumption for constraints from input-bound loops
  - "Flip-count" limit (to prevent endless loop expansions)

SAGE Architecture



SAGE is mostly developed by CSE

Some Experiments

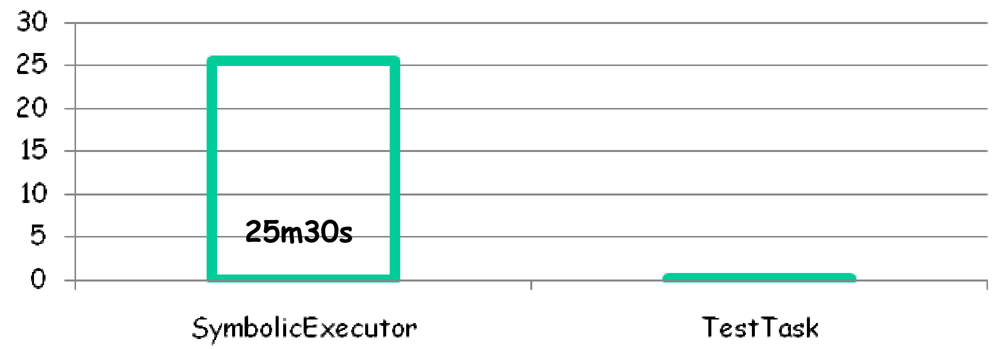
Most much (100x) bigger than ever tried before!

- Seven applications – 10 hours search each

App Tested	#Tests	Mean Depth	Mean #Instr.	Mean Input Size
ANI	11468	178	2,066,087	5,400
Media1	6890	73	3,409,376	65,536
Media2	1045	1100	271,432,489	27,335
Media3	2266	608	54,644,652	30,833
Media4	909	883	133,685,240	22,209
Compressed File Format	1527	65	480,435	634
OfficeApp	3008	6502	923,731,248	45,064

Generational Search Leverages Symbolic Execution

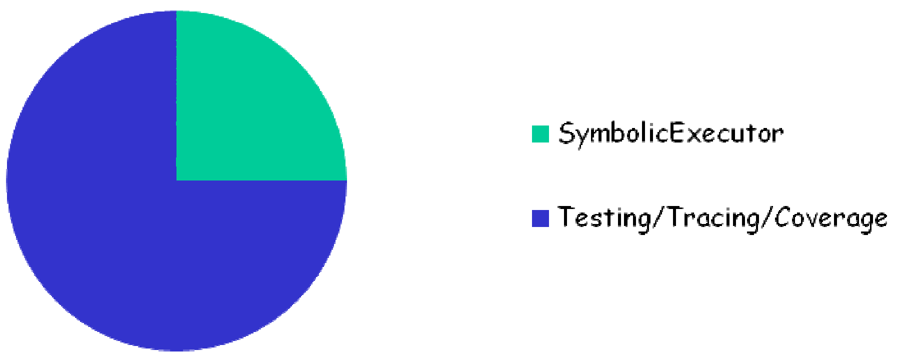
- Each symbolic execution is expensive



X1,000

- Yet, symbolic execution does not dominate search time

10 hours

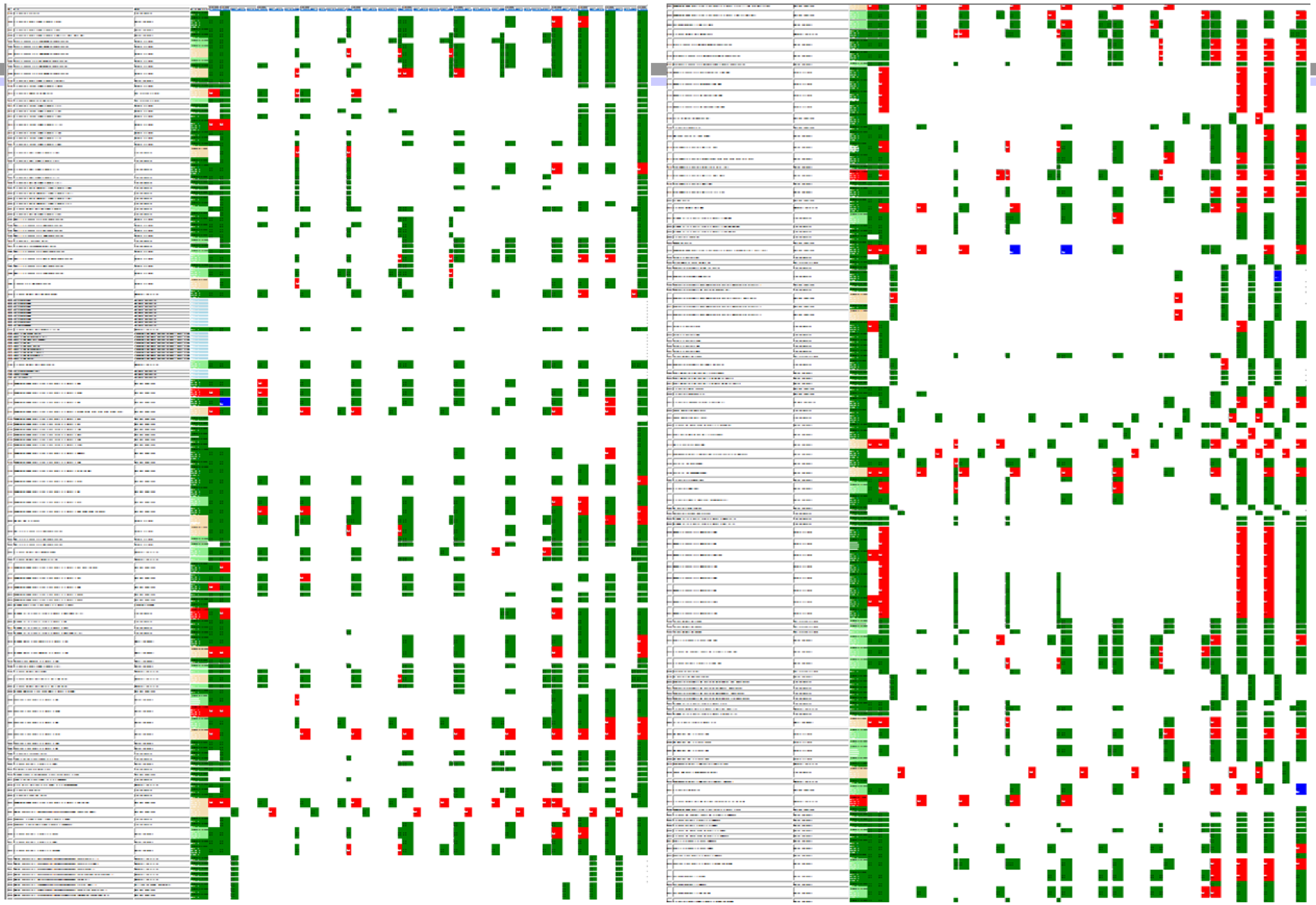


SAGE Results

Since April'07 1st release: many new security bugs found (missed by blackbox fuzzers, static analysis)

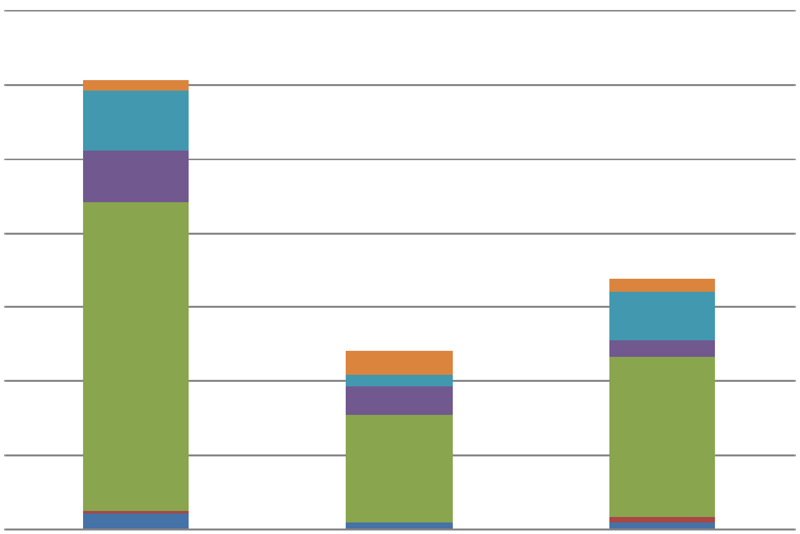
- Apps: image processors, media players, file decoders,...
- Bugs: Write A/Vs, Read A/Vs, Crashes,...
- Many triaged as "security critical, severity 1, priority 1" (would trigger Microsoft security bulletin if known outside MS)
- Example: WEX Security team for Win7
  - Dedicated fuzzing lab with 100s machines
  - 100s apps (deployed on 1billion+ computers)
  - ~1/3 of all fuzzing bugs found by SAGE !
- SAGE = gold medal at Fuzzing Olympics organized by SWI at BlueHat'08 (Oct'08)
- Credit due to entire SAGE team + users !

WEX Fuzz Dashboard Snippet



WEX Fuzzing Lab Bug Yield for Win7

How fuzzing bugs found (2006-2009) :



SAGE is running 24/7 on 100s machines:
"the largest usage ever of any SMT solver"
N. Bjorner + L. de Moura (MSR, Z3 authors)

- 100s of apps, total number of fuzzing bugs is confidential
- But SAGE didn't exist in 2006
- Since 2007 (SAGE 1st release), ~1/3 bugs found by SAGE
- But SAGE currently deployed on only ~2/3 of those apps
- Normalizing the data by 2/3, SAGE found ~1/2 bugs
- SAGE is more CPU expensive, so it is run last in the lab, so all SAGE bugs were missed by everything else!

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□äN
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!

Found after only 3 generations starting from seed3 file on next slide

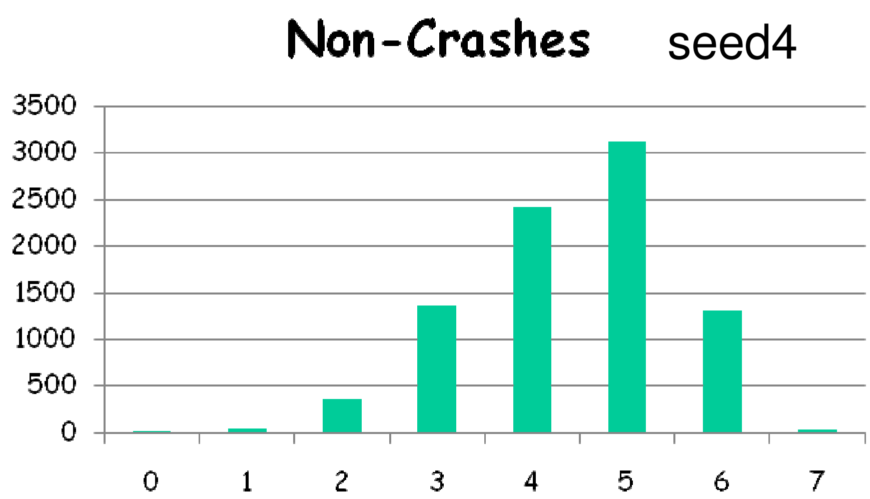
Different Seed Files, Different Crashes

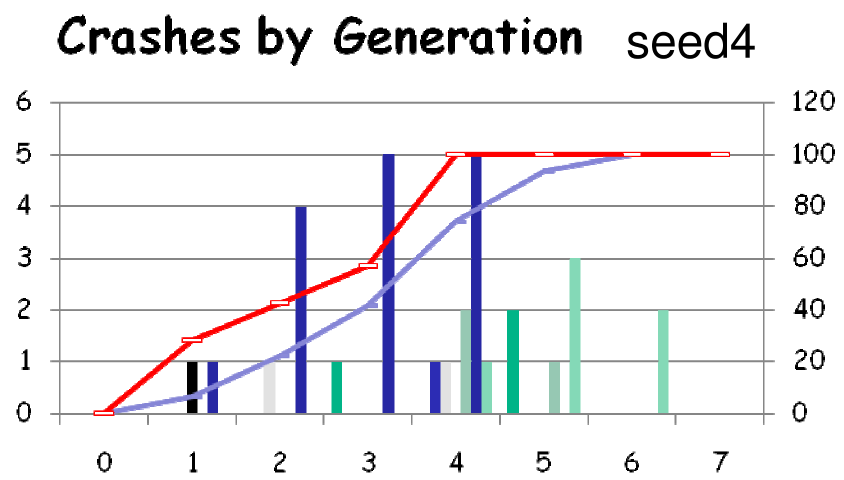
Bucket	seed1	seed2	seed3	seed4	seed5	100 zero bytes
1867196225	X	X	X	X	X	
2031962117	X	X	X	X	X	
612334691		X	X			
1061959981			X	X		
1212954973			X			X
1011628381			X	X		X
842674295				X		
1246509355			X	X		X
1527393075					X	
1277839407					X	
1951025690			X			

For the first time, we face bug triage issues!

Media1: 60 machine-hours, 44598 total tests, 357 crashes, 12 unique buckets

Most Bugs Found are "Shallow"





SAGE Summary

- SAGE is so effective at finding bugs that, for the first time, we face "bug triage" issues with dynamic test generation
- What makes it so effective?
  - Works on large applications (not unit test)
  - Can detect bugs due to problems across components
  - Fully automated (focus on file fuzzing)
  - Easy to deploy (x86 analysis – any language or build process !)
  - Now, used daily in various groups inside Microsoft

More On the Research Behind SAGE

- Challenges:
  - How to recover from imprecision in symbolic execution? PLDI'05
  - How to scale symbolic exec. to billions of instructions? NDSS'08
  - How to check efficiently many properties together? EMSOFT'08
  - How to leverage gram. specs for complex input formats? PLDI'08
  - How to deal with path explosion in large prgms? POPL'07, TACAS'08
  - How to reason precisely about pointers? ISSTA'09

+ research on constraint solvers (Z3, disolver,...)

Extension: Active Property Checking

- Traditional property checkers are "passive"
  - Purify, Valgrind, AppVerifier, TruScan, etc.
  - Check only the current concrete execution
  - Can check many properties at once
- Combine with symbolic execution "active"
  - Reason about all inputs on same path
  - Apply heavier constraint solving/proving
  - "Actively" look for input violating property
- Ex: array ref **a**[i] where i depends on input, **a** is of size c
  - Try to force buffer over/underflow: add "(i < 0) OR (i >= c)" to the path constraint; if SAT, next test should hit a bug!
- Challenge: inject/manage all such constraints efficiently…

Ext.: Grammar-Based Whitebox Fuzzing

- Input precondition specified as a context-free grammar
- Avoids path explosion in lexer and parser

- Faster, better and deeper coverage for applications with structured inputs (XML, etc.)

generation strategy (each ran 2 hours)	#inputs	total coverage	coverage in code gen
blackbox fuzzing	8658	14%	51%
whitebox fuzzing	6883	15%	54%
grammar-based blackbox fuzzing	7837	12%	61%
grammar-based whitebox fuzzing	2378	20%	82%

Ext.: Compositionality = Key to Scalability

- Problem: executing all feasible paths does not scale !
- Idea: compositional dynamic test generation
  - use summaries of individual functions (arbitrary program blocks) like in interprocedural static analysis
  - If f calls g, test g separately, summarize the results, and use g's summary when testing f
  - A summary $\varphi(g)$ is a disjunction of path constraints expressed in terms of input preconditions and output postconditions:

    $\varphi(g) = \vee\, \varphi(w)$ with $\varphi(w) = pre(w) \wedge post(w)$

    expressed in terms of g's inputs and outputs
  - g's outputs are treated as symbolic inputs to a calling function f
- Can provide same path coverage exponentially faster !

Conclusion: Blackbox vs. Whitebox Fuzzing

- Different cost/precision tradeoffs
  - Blackbox is lightweight, easy and fast, but poor coverage
  - Whitebox is smarter, but complex and slower
  - Note: other recent "semi-whitebox" approaches
    - Less smart (no symbolic exec, constr. solving) but more lightweight: Flayer (taint-flow, may generate false alarms), Bunny-the-fuzzer (taint-flow, source-based, fuzz heuristics from input usage), etc.
- Which is more effective at finding bugs? It depends…
  - Many apps are so buggy, any form of fuzzing find bugs in those !
  - Once low-hanging bugs are gone, fuzzing must become smarter: use whitebox and/or user-provided guidance (grammars, etc.)
- Bottom-line: in practice, use both! (We do at Microsoft)

Future Work (The Big Picture)

- During the last decade, code inspection for standard programming errors has largely been automated with static code analysis
- Next: automate testing (as much as possible)
  - Thanks to advances in program analysis, efficient constraint solvers and powerful computers
- Whitebox testing: automatic code-based test generation
  - Like static analysis: automatic, scalable, checks many properties
  - Today, we can exhaustively test small applications, or partially test large applications
  - Biggest impact so far: whitebox fuzzing for (Windows) security testing
    - Improved security for a billion computers worldwide!
  - Next: towards exhaustive testing of large application (verification)
  - How far can we go?