Random Testing for Security:

Blackbox vs. Whitebox Fuzzing

Patrice Godefroid

Microsoft Research

Acknowledgments

- Most of this talk presents recent results of joint work with Michael Y. Levin and David Molnar,
- Extending prior joint work with Nils Klarlund and Koushik Sen
- References: (see http://research.microsoft.com/users/pg)
  - DART: Directed Automated Random Testing, Godefroid-Klarlund-Sen, PLDI'2005
  - Compositional Dynamic Test Generation, Godefroid, POPL'2007
  - Automated Whitebox Fuzz Testing, Godefroid-Levin-Molnar, Technical Report MSR-TR-2007-58, Microsoft, May 2007
  - Active Property Checking, Godefroid-Levin-Molnar, Technical Report MSR-TR-2007-91, Microsoft, July 2007

Security is Critical

- 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: 100s of bugs found this way…

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 ("Fuzz"), 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 [Godefroid-Klarlund-Sen-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

Related Work

- Static Test Generation: impractical whenever symbolic execution is imprecise (see previous example)

- Dynamic test generation (Korel, Gupta-Mathur-Soffa, etc.)
  - Attempt to exercise a specific program path
  - Instead, DART attempts to cover all executable program paths (like model checking) while using runtime property checkers (Purify,…)

    DART = Systematic Dynamic Test Generation

  - + key engineering contributions + results with "real" examples (handles function calls, unknown functions, exploits simultaneous concrete and symbolic executions, has run-time checks to detect incompleteness, applied to large examples, found new bugs, etc.)

- DART extends systematic testing (VeriSoft,…) for concurrency to data nondeterminism (see [Colby-Godefroid-Jagadeesan, PLDI'98])

DART Implementations

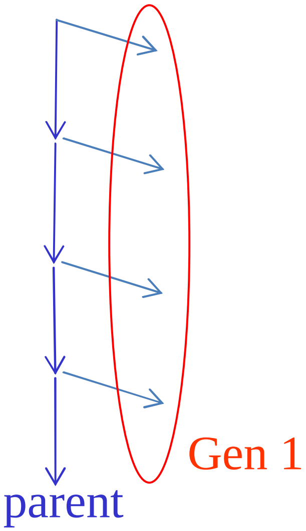
- Defined by symbolic execution, constraint generation and solving
  - Languages: C, Java, x86, .NET,…
  - Theories: linear arith., bit-vectors, arrays, uninterpreted functions,…
  - Solvers: lp_solve, CVCLite, STP, Disolver, Z3,…
- Examples of DART implementations:
  - EXE/EGT (Stanford): independent ['05-'06] closely related work
  - CUTE = same as first (old) DART implementation done at Bell Labs
  - SAGE (CSE/MSR) implements DART for x86 binaries and merges it with "fuzz" testing for finding security bugs (more later)
  - PEX (MSR) implements DART for .NET binaries in conjunction with "parameterized-unit tests" for unit testing of .NET programs
  - YOGI (MSR) implements DART to check the feasibility of program paths generated statically using a SLAM-like tool
  - Vigilante (MSR) implements DART to generate worm filters
  - BitScope (CMU/Berkeley) implements DART for malware analysis
  - Etc.

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 (SAGE)

- SAGE = "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: 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’

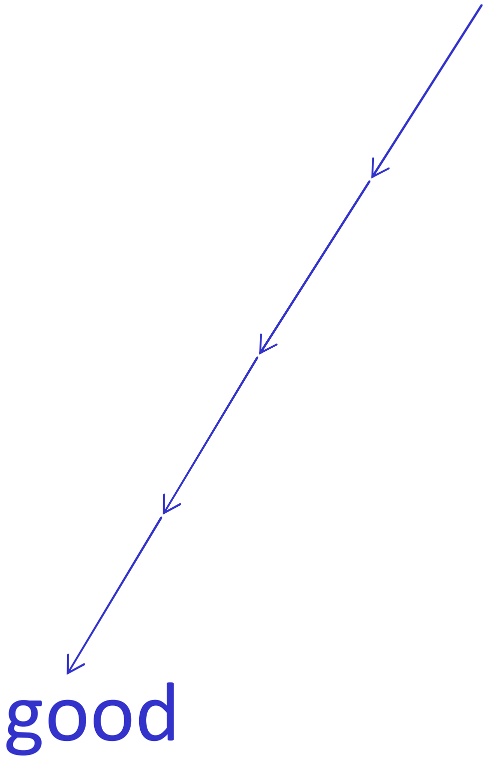
I_2 != ‘d’

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++;
   if (input[1] == ‘a’) cnt++;
   if (input[2] == ‘d’) cnt++;
   if (input[3] == ‘!’) cnt++;
   if (cnt >= 3) crash();
}
```

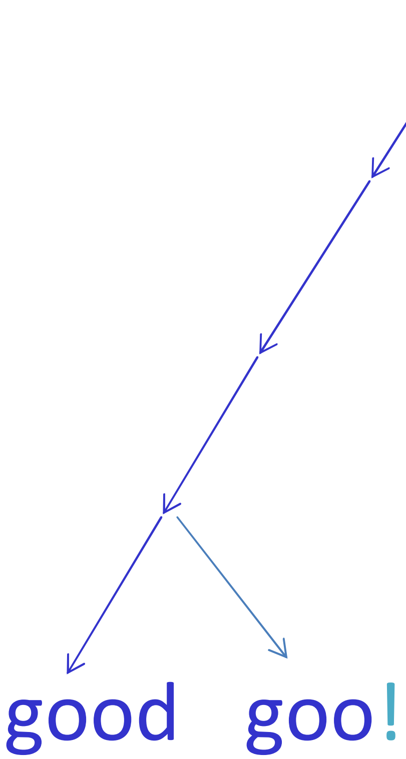
I_0 != ‘b’

I_1 != ‘a’

I_2 != ‘d’

I_3 != ‘!’

Depth-First Search



```
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();
}
```

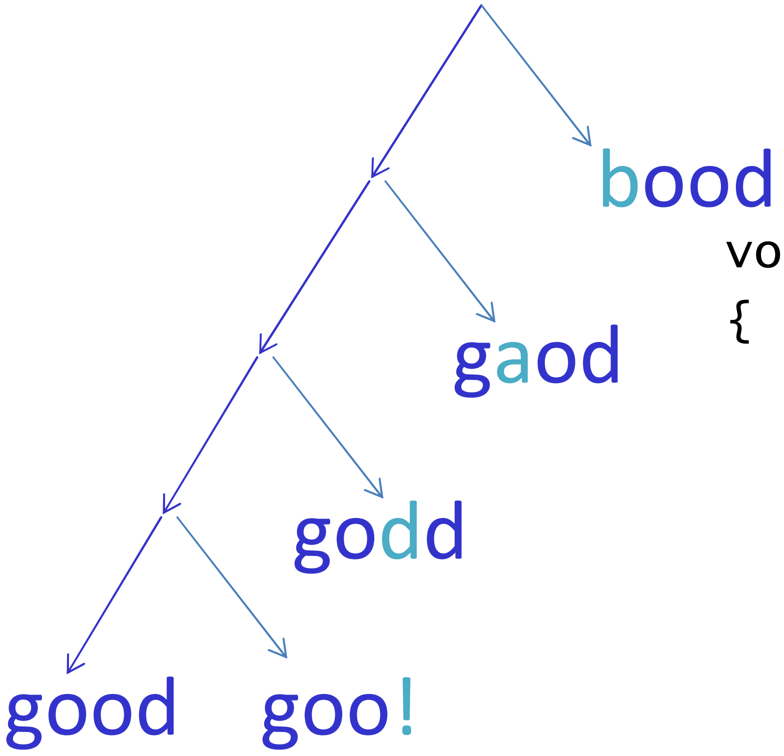
I_0 != ‘b’

I_1 != ‘a’

I_2 != ‘d’

I_3 == ‘!’

Generational Search



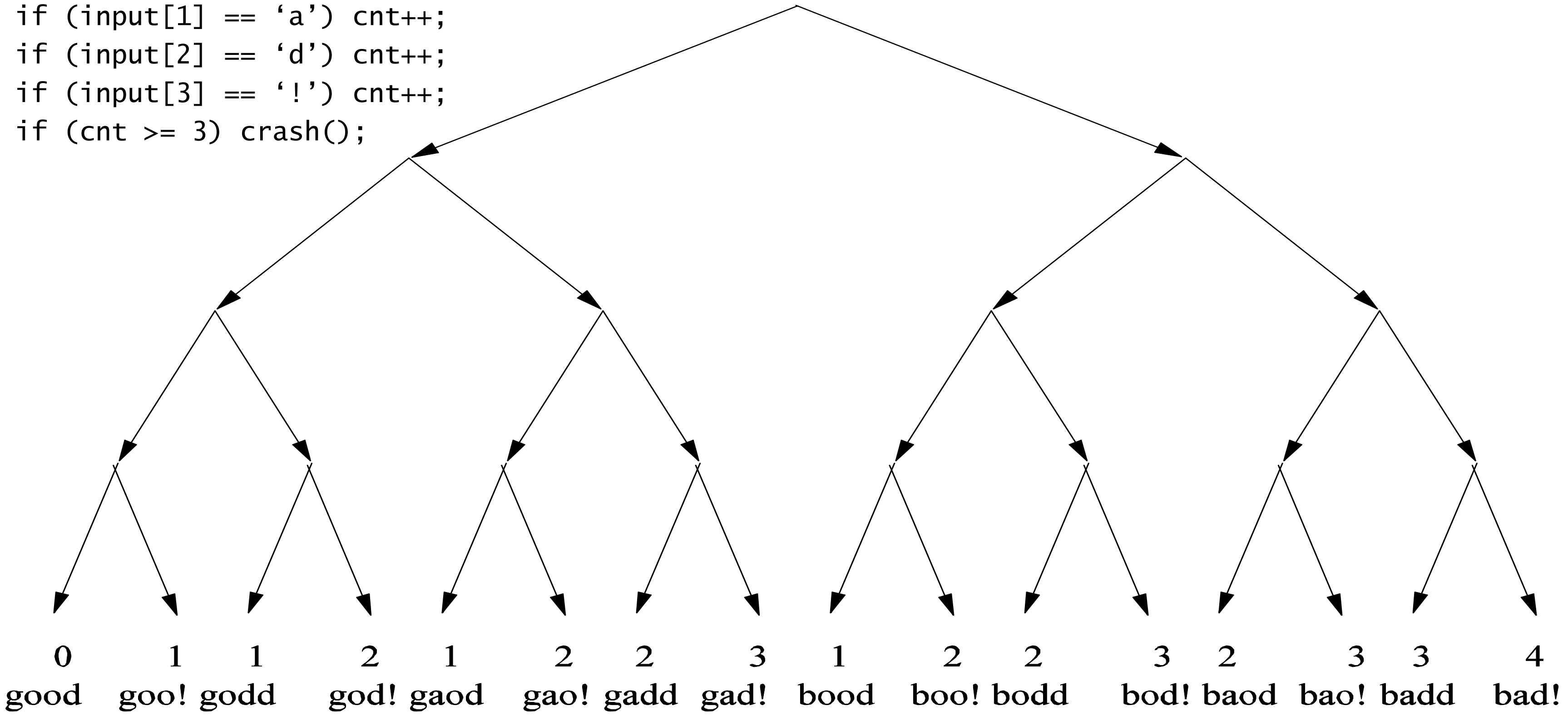
```
void top(char input[4])
{
   int cnt = 0;
   if (input[0] == ‘b’) cnt++;  I0 == ‘b’
   if (input[1] == ‘a’) cnt++;  I1 == ‘a’
   if (input[2] == ‘d’) cnt++;  I2 == ‘d’
   if (input[3] == ‘!’) cnt++;  I3 == ‘!’
   if (cnt >= 3) crash();
}
```

Four “Generation 1” test cases !

Note: in this ex, all branches are covered after Gen 1
(Helps detect bugs faster - see later experiments)

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();
}
```



Implementing a Generational Search

- Main loop:
  - Each symbolic execution (parent) generates many new tests (children)
  - Each new test is executed, runtime checked, and ranked by the new coverage it triggers (= “block-coverage heuristics”)
  - The highest ranked children becomes a parent; repeat the process
- Note: for each child, backtracking is prohibited above the backtracking point (to prevent redundant tests)
- Thus, such a generational search
  - maximizes the number of new tests per (expensive) symbolic execution
  - is resilient to divergences (unexpected redundant tests are ranked low)
  - is not trapped into “local minima” (unlike DFS, BFS,…)
  - explores “all depths at once”, hence discovers new code (+bugs) faster
  - is easy to parallelize

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)

Some Experiments

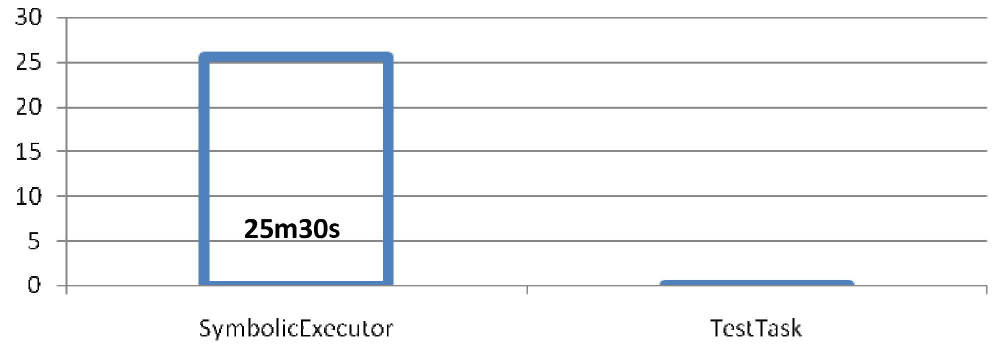
Most much (100x) bigger than ever tried before!

- 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

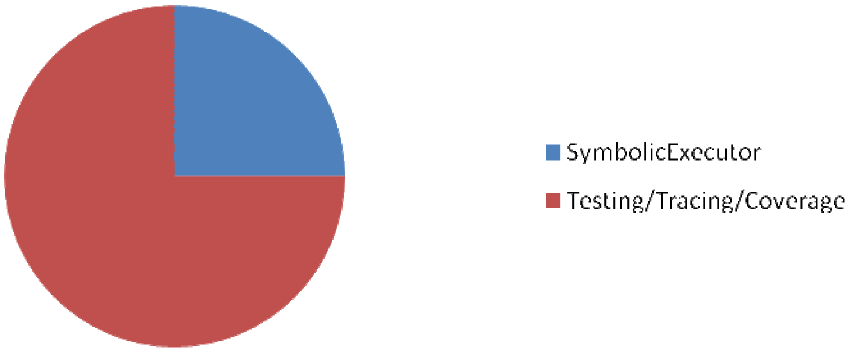
- Each symbolic execution is expensive



X1,000

- Yet, symbolic execution does not dominate search time

10 hours



Initial Experiences with SAGE

- Since 1st internal release in April'07: tens of new security bugs found (most missed by blackbox fuzzers, static analysis)
- 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"
- Credit is due to the entire SAGE team and users:
  - CSE: Michael Levin (DevLead), Christopher Marsh,
    Dennis Jeffries (intern'06), Adam Kiezun (intern'07);
    Mgmt: Hunter Hudson, Manuvir Das,…
    (+ symbolic exec. engine Nirvana/iDNA/TruScan contributors)
  - MSR: Patrice Godefroid, David Molnar (intern'07)
    (+ constraint solvers Disolver and Z3 contributors)
  - Plus work of many users who found and filed most of these bugs!

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:

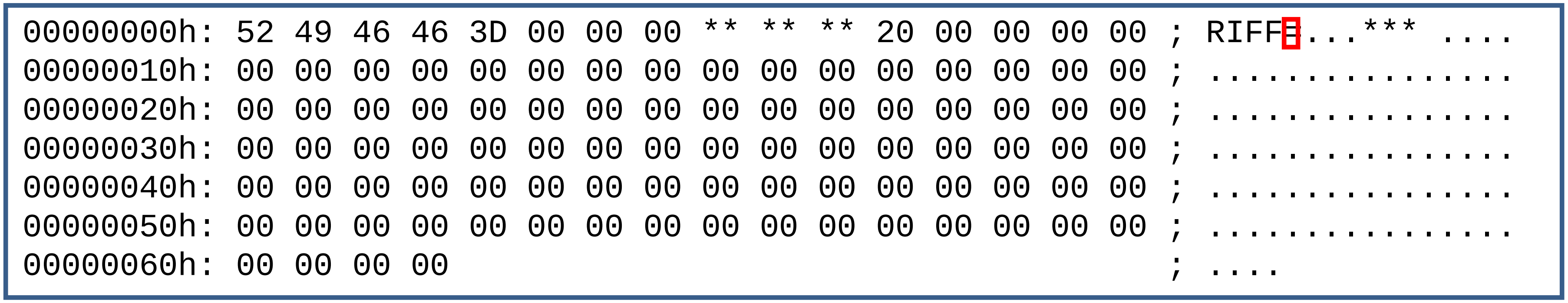
```
00000000h: 52 49 46 46 00 00 00 00 ** ** ** 20 00 00 00 00 ; RIFF....*** ....
00000010h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000020h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000030h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
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

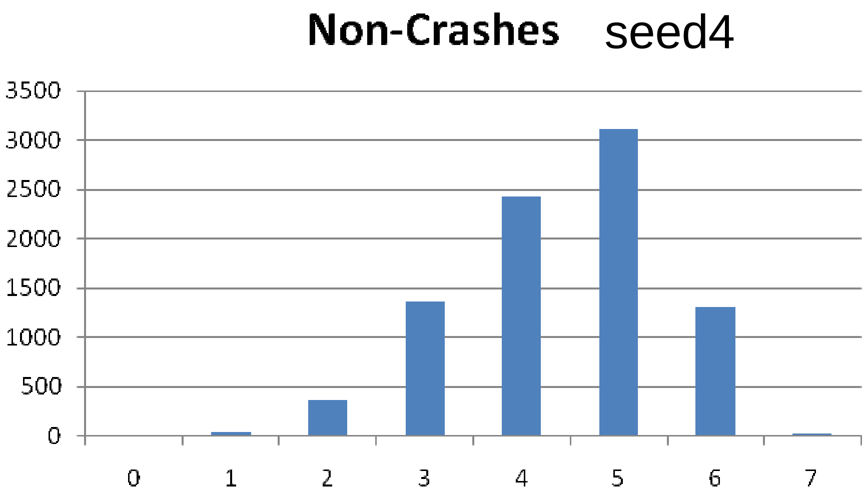
Different Seed Files, Different Crashes

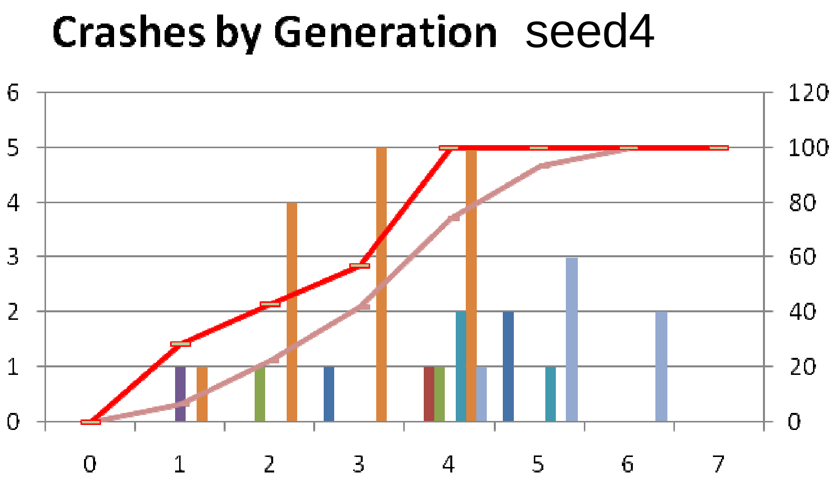
Bucket	seed1	seed2	seed3	seed4	seed5	100 zero bytes
1867196225	X	X	X	X	X	
2031962117	X	X	X	X	X	
612334691		X	X			
1061959981			X	X		
1212954973			X			X
1011628381			X	X		X
842674295				X		
1246509355			X	X		X
1527393075					X	
1277839407					X	
1951025690			X			

For the first time, we face bug triage issues!

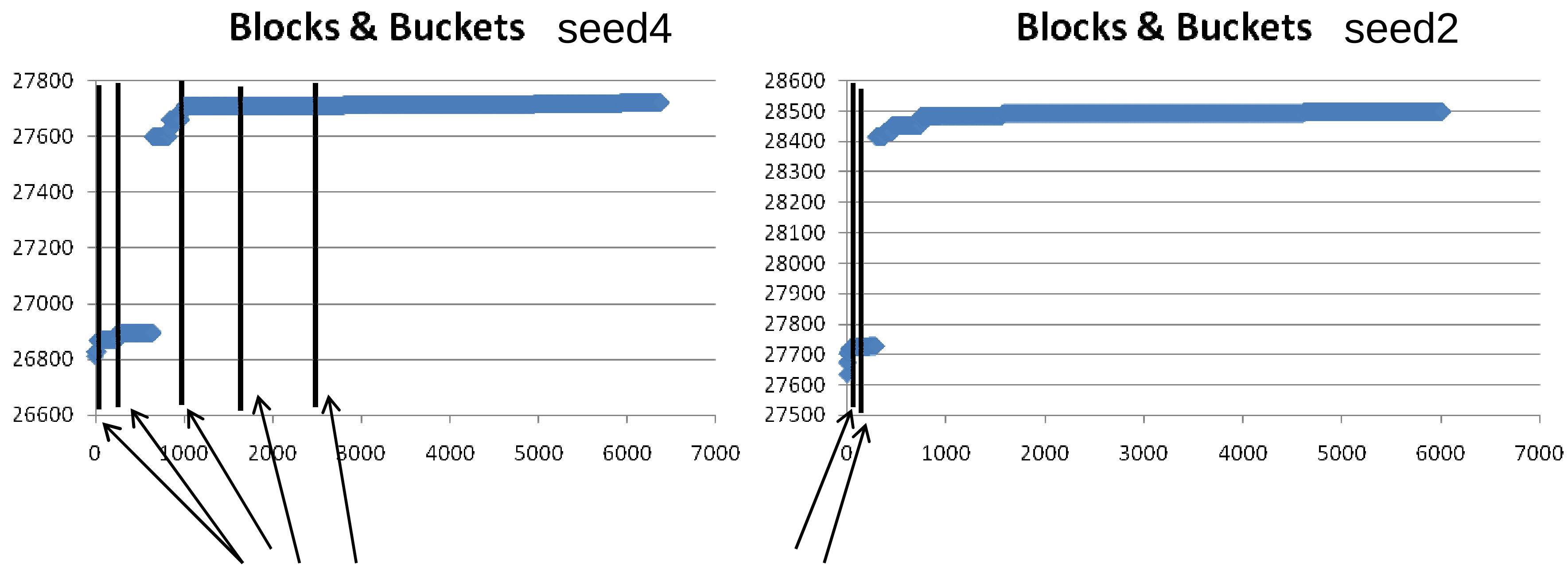
Media1: 60 machine-hours, 44598 total tests, 357 crashes, 12 unique buckets

Most Bugs Found are "Shallow"





Coverage and New Crashes: Low Correlation



New bucket found for the first time

SAGE Summary

- SAGE is so effective at finding bugs that, for the first time, we face “bug triage” issues with dynamic test generation
  - Claim: SAGE is today the most effective “DART implementation”
  - (no other team has ever reported so many bugs)
- 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

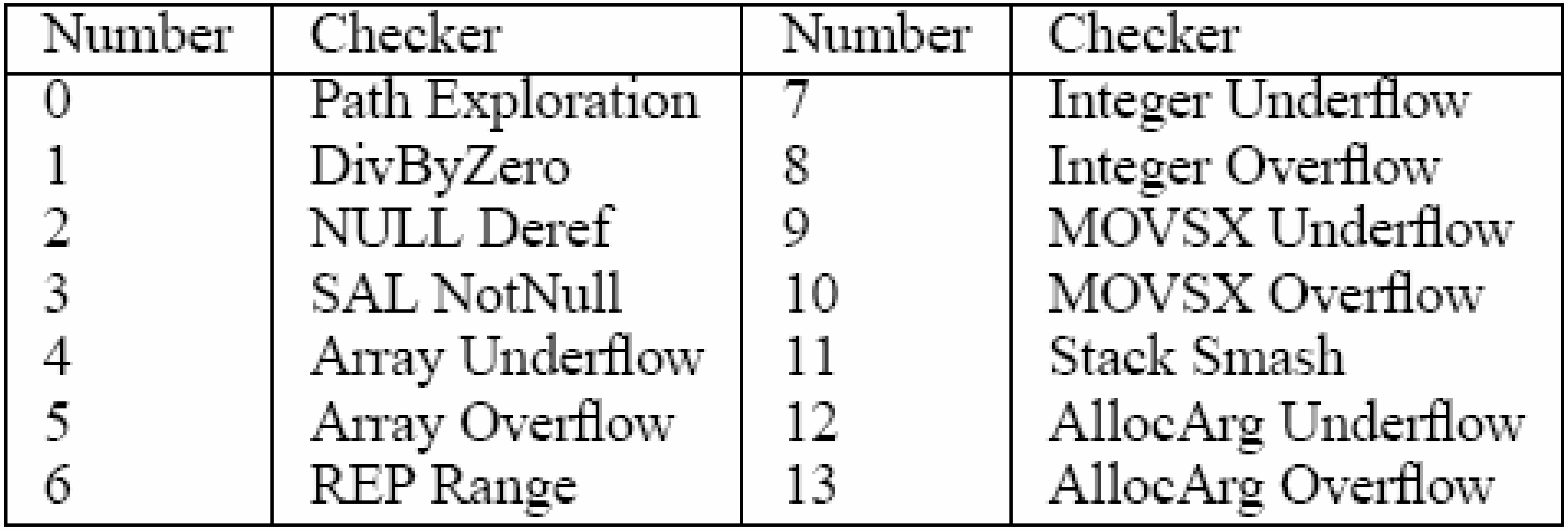
New: Active Property Checking

- 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!

Optimizations

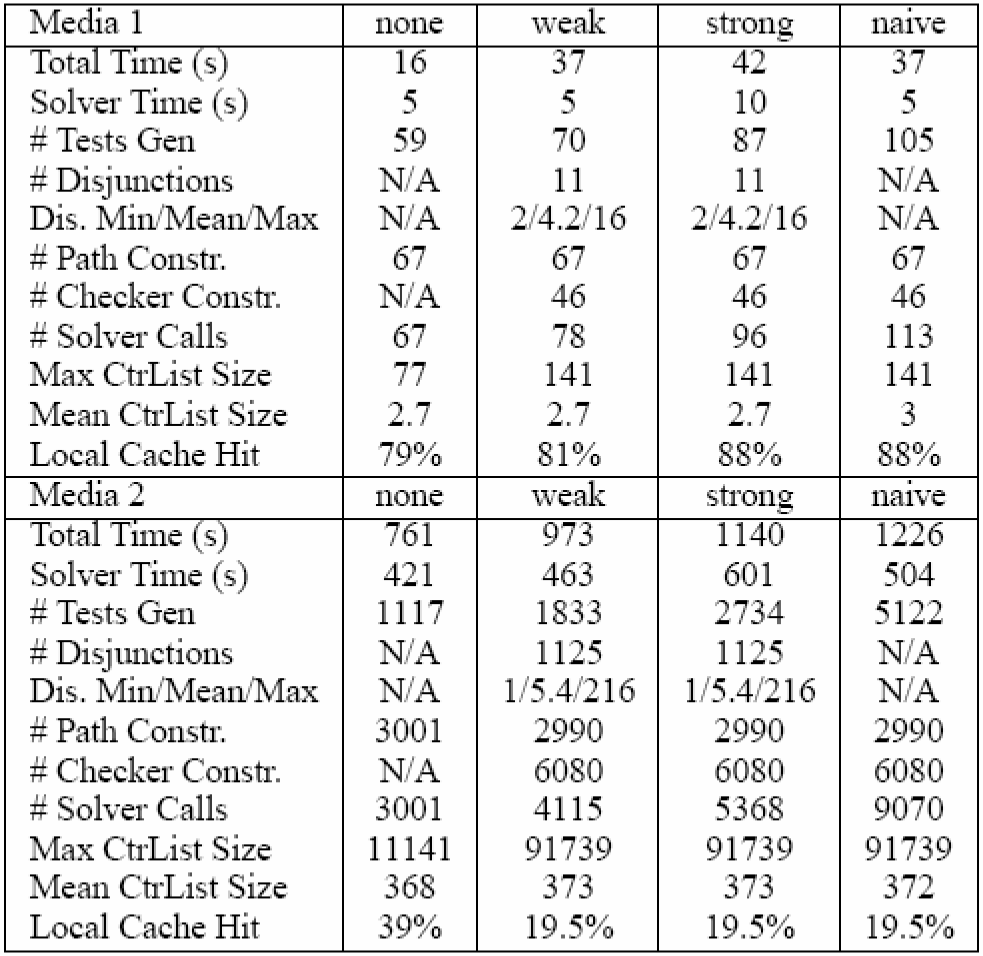
- More properties to check means more bugs found!
- Challenge: inject such constraints in an optimal way…
- Optimizations:
  - Minimize # solver calls
    - Given a path constraint *pc* and n checker constraints φC1,…, φCn
    - "**Naïve** approach": query each *pc* AND ¬ φCi
    - **Weak**: query *pc* AND (¬ φC1 OR…OR ¬ φCn)
    - **Strong**: query *pc* AND (¬ φC1 OR…OR ¬ φCn)
      - If result is UNSAT, done; otherwise remove satisfied disjuncts, repeat
  - Minimize formula size (unrelated constraint optimization)
  - Constraint caching (locally and globally)

Active Checkers in SAGE



Number	Checker	Number	Checker
0	Path Exploration	7	Integer Underflow
1	DivByZero	8	Integer Overflow
2	NULL Deref	9	MOVSX Underflow
3	SAL NotNull	10	MOVSX Overflow
4	Array Underflow	11	Stack Smash
5	Array Overflow	12	AllocArg Underflow
6	REP Range	13	AllocArg Overflow

Microbenchmarks



Media 1	none	weak	strong	naive
Total Time (s)	16	37	42	37
Solver Time (s)	5	5	10	5
# Tests Gen	59	70	87	105
# Disjunctions	N/A	11	11	N/A
Dis. Min/Mean/Max	N/A	2/4.2/16	2/4.2/16	N/A
# Path Constr.	67	67	67	67
# Checker Constr.	N/A	46	46	46
# Solver Calls	67	78	96	113
Max CtrList Size	77	141	141	141
Mean CtrList Size	2.7	2.7	2.7	3
Local Cache Hit	79%	81%	88%	88%
Media 2	none	weak	strong	naive
Total Time (s)	761	973	1140	1226
Solver Time (s)	421	463	601	504
# Tests Gen	1117	1833	2734	5122
# Disjunctions	N/A	1125	1125	N/A
Dis. Min/Mean/Max	N/A	1/5.4/216	1/5.4/216	N/A
# Path Constr.	3001	2990	2990	2990
# Checker Constr.	N/A	6080	6080	6080
# Solver Calls	3001	4115	5368	9070
Max CtrList Size	11141	91739	91739	91739
Mean CtrList Size	368	373	373	372
Local Cache Hit	39%	19.5%	19.5%	19.5%

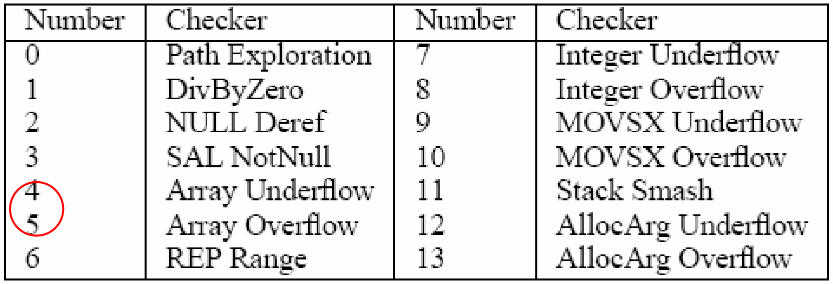
- Checkers produce more test cases, reasonable cost
  - Media 2 naive – 61% extra time, 4.5 times test cases
- Weak combination has lowest overhead
- Unrelated constraint elimination, caching are important for performance

Experiment: Buckets by Checker Type

Bucket	Kind	0	2	4	5	7	8
1867196225	NULL	No/W/S				W/S	W/S
1867196225	ReadAV	No/W					
1277839407	ReadAV	S					
1061959981	ReadAV		S			S	S
1392730167	ReadAV	S					
1212954973	ReadAV				S	S	S
1246509355	ReadAV				S	W/S	W/S
1527393075	ReadAV	S					
1011628381	ReadAV					S	W/S
2031962117	ReadAV	No/W/S					
26861377	ReadAV	No/S					
842674295	WriteAV		S	S		S	S

Media1 : 30 machine-hours, 41658 total tests, 783 crashes, 12 buckets

Checker Yields - Media1



Number	Checker	Number	Checker
0	Path Exploration	7	Integer Underflow
1	DivByZero	8	Integer Overflow
2	NULL Deref	9	MOVSX Underflow
3	SAL NotNull	10	MOVSX Overflow
4	Array Underflow	11	Stack Smash
5	Array Overflow	12	AllocArg Underflow
6	REP Range	13	AllocArg Overflow

	0	2	4	5	7	8
Injected	27612	26	13	13	11153	11153
Solved	18056	22	2	3	3179	5552
Crashes	339	22	2	3	139	136
Yield	1.9%	**100%**	**100%**	**100%**	4.4%	2.4%

Media1: 30 machine-hours, 41658 total tests, 783 crashes, 12 buckets

New: 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

SMART = Scalable DART

- Unlike interprocedural static analysis:
  - Summaries may include information about concrete values (to allow partial symbolic reasoning)
  - Each summary needs to be grounded in some concrete execution (to guarantee that no false alarm is ever generated): here, "must" summaries, not "may" summaries !
  - Bottom-up strategy for computing summaries is questionable (may generate spurious summaries and too few relevant ones)
  - Top-down strategy to compute summaries only for reachable calling contexts: SMART algorithm
  - SMART = Systematic Modular Automated Random Testing
  - Same path coverage as DART but can be exponentially faster!

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

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

SMART Properties

- Theorem: SMART provides same path coverage as DART
  - Corollary: same branch coverage, assertion violations,…
- Complexity: if b bounds the number of intraprocedural paths, number of runs by SMART is linear in b
  (while number of runs by DART can be exponential in b)
  - Similar to interprocedural static analysis, Hierarchical-FSM/Pushdown-system verification…
- Notes: arbitrary program blocks ok, recursion ok, concurrency is orthogonal (but arguably inherently non-compositional in general…)
- Full-fledged implementation under way…

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
  - Next: towards exhaustive testing of large application (verification)
  - How far can we go?