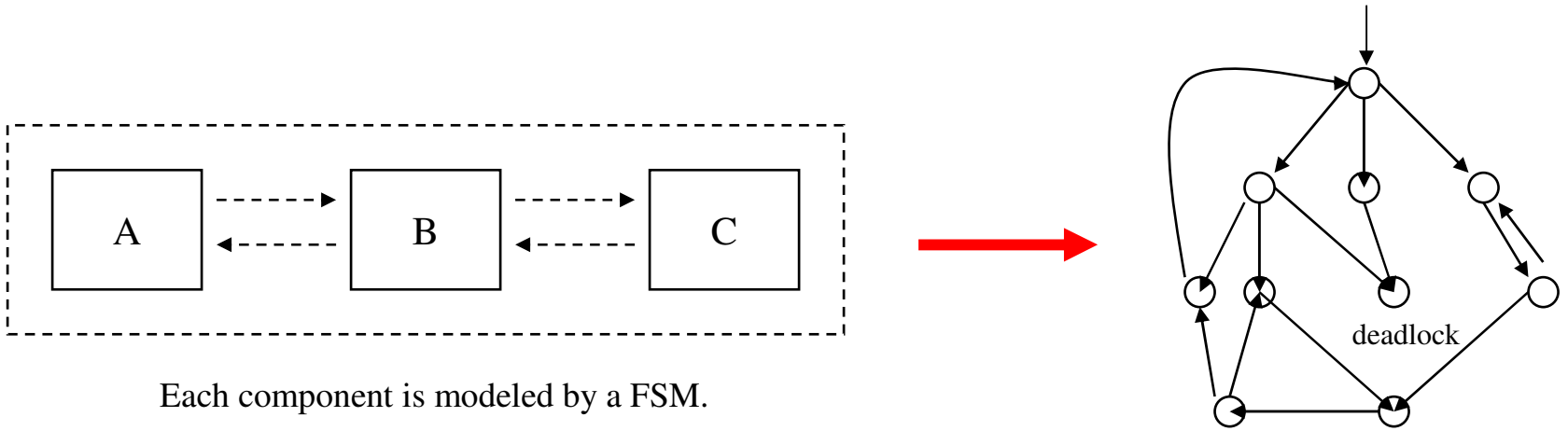
Software Model Checking 2.0

Patrice Godefroid

Microsoft Research

"Model Checking"

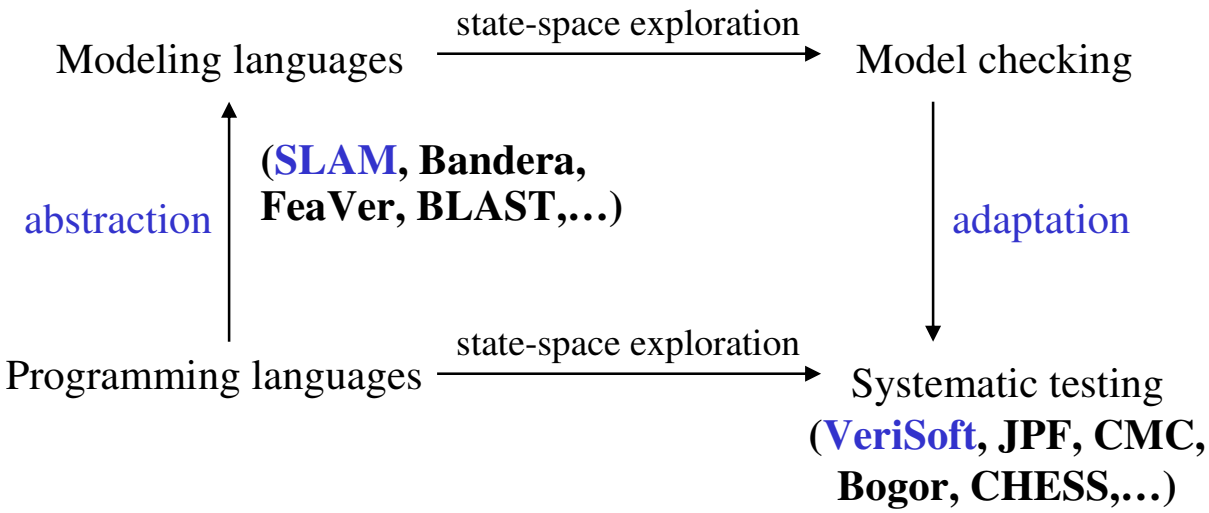


Each component is modeled by a FSM.

- Model Checking (MC) is
  - check whether a program satisfies a property by exploring its state space
  - systematic state-space exploration = exhaustive testing
  - "check whether the system satisfies a temporal-logic formula"
- Simple yet effective technique for finding bugs in high-level hardware and software designs (examples: FormalCheck for Hardware, SPIN for Software, etc.)
- Once thoroughly checked, models can be compiled and used as the core of the implementation (examples: SDL, VFSM, etc.)

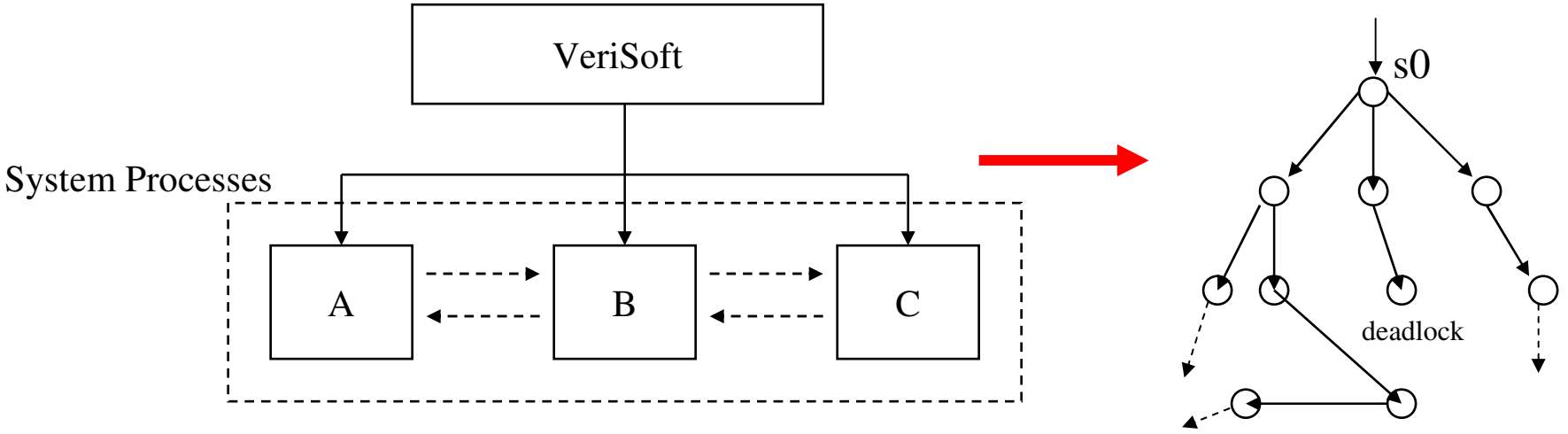
Model Checking of Software

- 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:



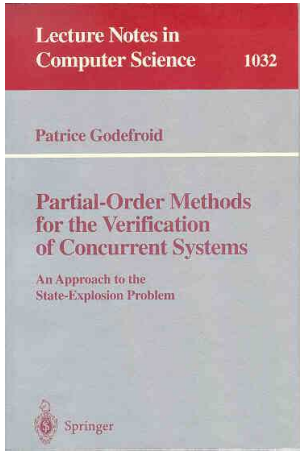
VeriSoft: Systematic Software Testing

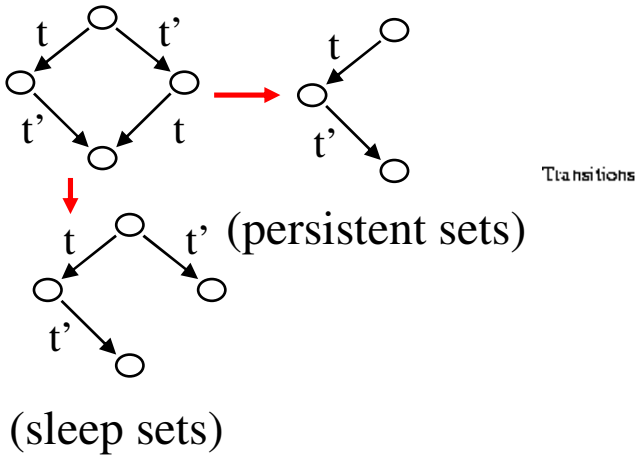
- State Space = "product of (OS) processes" (Dynamic Semantics)
- Systematically drive the system along all its state space paths (= automatically generate, execute and evaluate many scenarios)
- Control and observe the execution of concurrent processes by intercepting system calls (communication, assertion violations, etc.)
- From a given initial state, one can always guarantee a complete coverage of the state space up to some depth

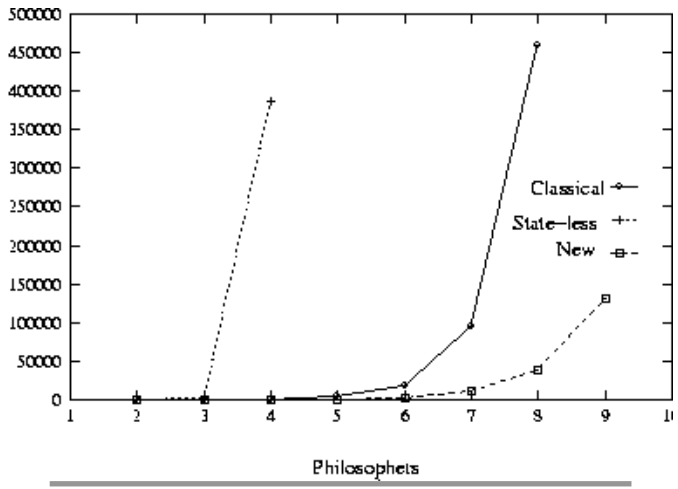


VeriSoft Innovations

- VeriSoft is the first systematic state-space exploration tool for concurrent systems composed of processes executing arbitrary code (e.g., C, C++,…) [POPL97]
  - No static analysis (programming language independent)
  - "VS_toss(int)" to simulate nondeterminism at run-time
  - "State-Less" search (no state encodings saved in memory)
  - Uses "partial-order reduction algorithms" to make a state-less search tractable





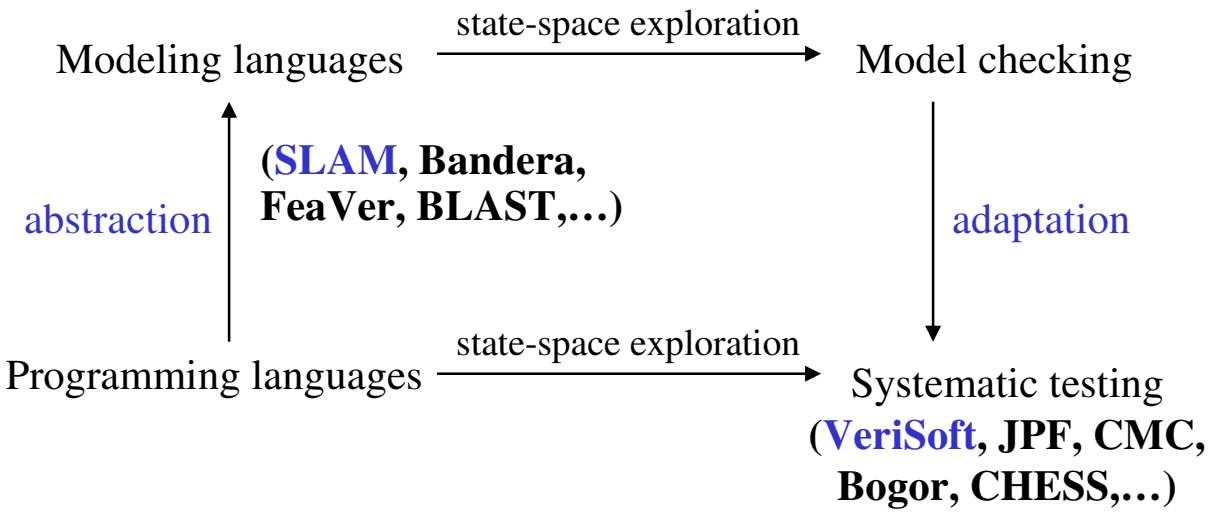


Applications & Discussion

- Examples of successful applications (at Lucent):
  - 4ESS Heart-Beat Monitor debugging and unit testing (1998)
  - WaveStar 40G R4 integration and system testing (1999-2000)
  - 3G Wireless CDMA call processing library testing (2000-2001)
  - Critical bugs found in each case ("$1M+ saved")
- VeriSoft is available outside Lucent since 1999
  - 100's of non-commercial (free) licenses in 25+ countries
- Conclusions: (Lessons Learned)
  - VeriSoft is not a "silver bullet": limited by state explosion, etc.
  - Used properly, VeriSoft is very effective at finding bugs
  - Those bugs would otherwise be found by the customer !
  - So the real question is: "How much ($) do you care about bugs?"

Model Checking of Software

- 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:



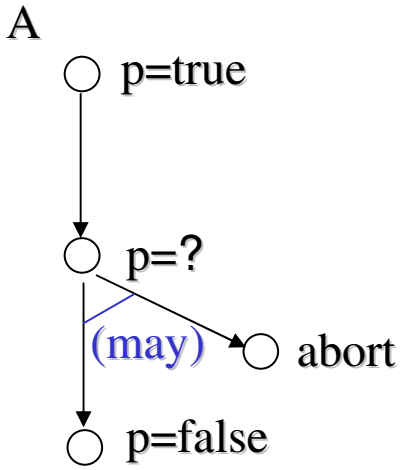
Static Approach: Automatic Abstraction

- Example: "Abstract-Check-Refine" Loop (SLAM)
  1. Abstract: generate a (may) abstraction via static analysis
     - Ex: predicate abstraction and boolean program
  2. Check: "model check" the abstraction
  3. Refine: map abstract error traces back to code, or refine the abstraction (e.g., by adding predicates); goto 1

```
Program P( ) {
  int x = 1;
  x = h(x);
  if (odd(x))
    abort(); // error!
  x = 0;
}
```

Predicate abstraction

p: "x is odd"

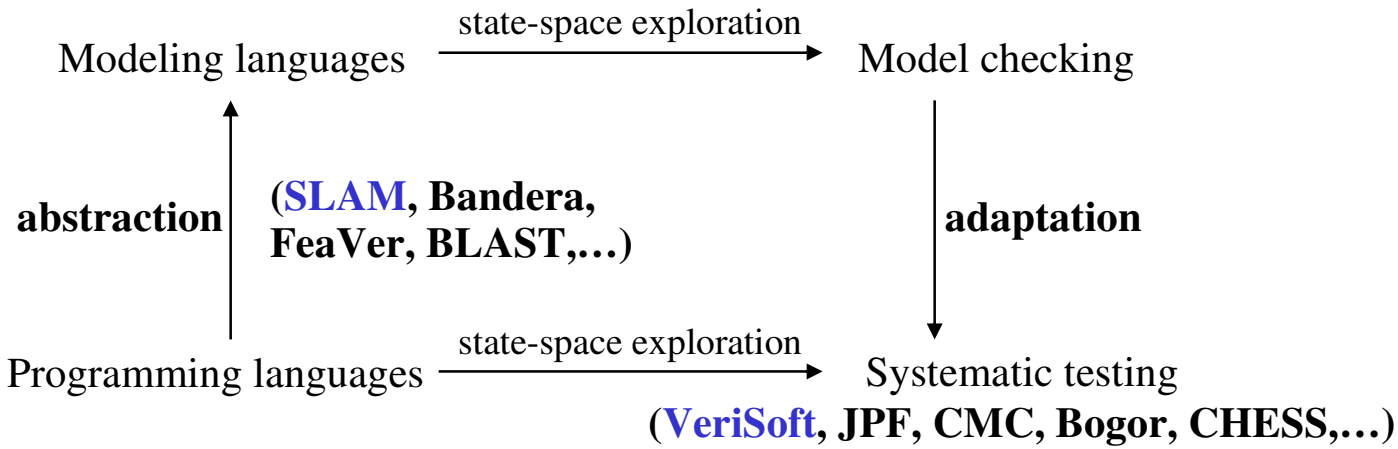


Main Ideas and Issues

1. Abstract: extract a "model" out of program via static analysis
   - Which programming languages are supported? (C, C++, Java, ?)
   - Additional assumptions? (Pointers? Recursion? Concurrency?…)
   - What is the target modeling language? ((C)(E)FSMs, PDAs,…)
   - Can/must the abstraction process be guided by the user? How?
2. Model check the abstraction
   - What properties can be checked? (Safety? Liveness?,…)
   - How to model the environment? (Closed or open system ?…)
   - Which model-checking algorithm? (PDAs, BDDs, SAT solvers…)
   - Is the abstraction "conservative"? (= is the analysis "sound"?)
3. Map abstract error traces back to code, refine the abstraction
   - Spurious behaviors may have been introduced during Step 1
   - How to map scenarios leading to errors back to the code?
   - When an error trace is spurious, how to refine the abstraction?

Software Model Checking 1.0

Two complementary approaches to software model checking:



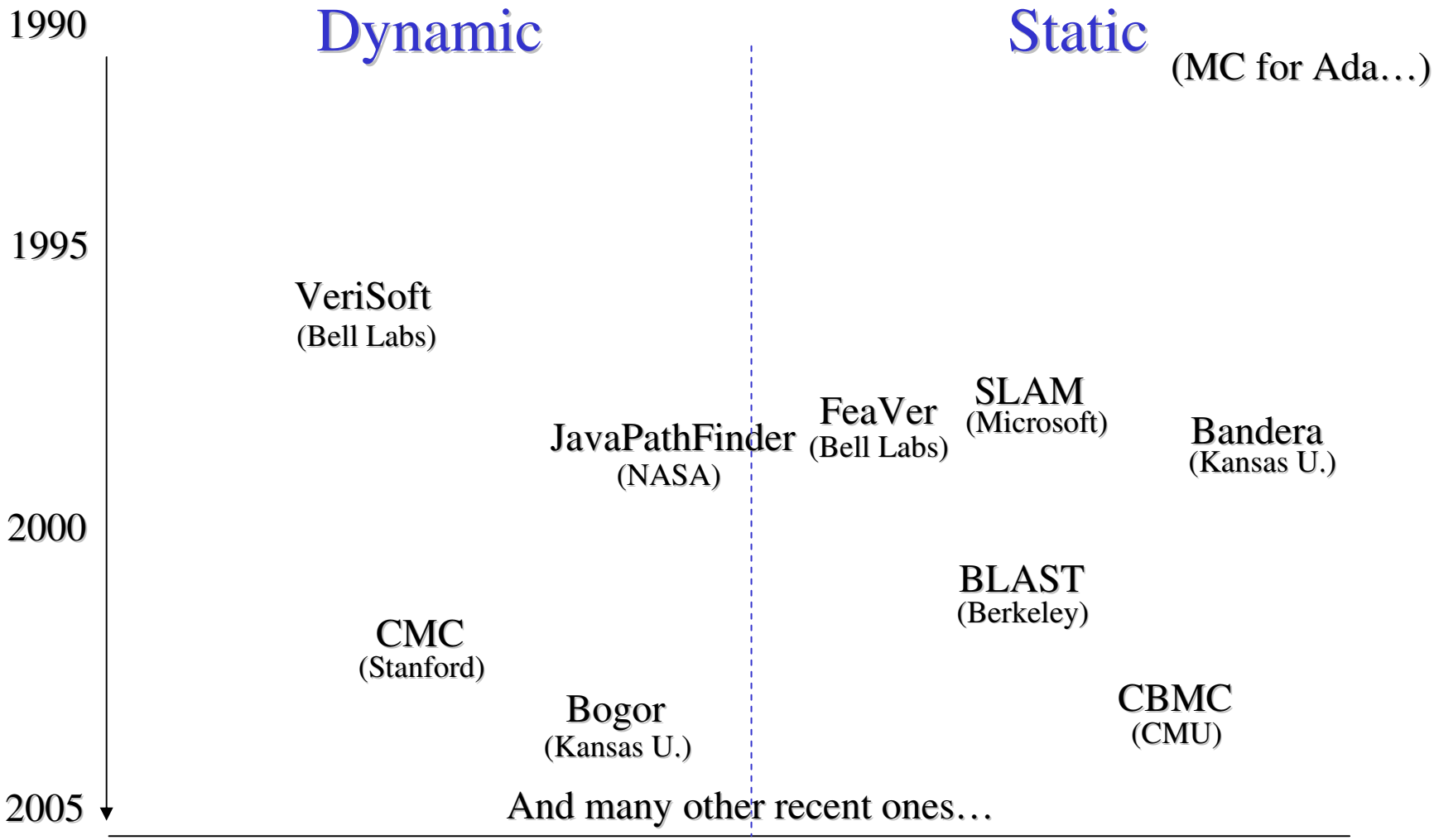
Automatic Abstraction (static analysis):

- Idea: parse code to generate an abstract model that can be analyzed using model checking
- No execution required but language dependent
- May produce spurious counterexamples (unsound bugs)
- Can prove correctness (complete) in theory (but not in practice…)

Systematic Testing (dynamic analysis):

- Idea: control the execution of multiple test-drivers/processes by intercepting systems calls
- Language independent but requires execution
- Counterexamples arise from code (sound bugs)
- Provide a complete state-space coverage up to some depth only (typically incomplete)

Software Model Checking Tools



What Next? Software Model Checking 2.0

- General idea: combine static and dynamic analysis
- Motivation: take the best of both approaches (precision of dynamic AND efficiency of static)
- Example: DART (Directed Automated Random Testing)
  - Aka "systematic dynamic test generation" [PLDI'05]
  - Can be viewed as extending the VeriSoft approach to data nondeterminism (see also [PLDI'98] for an earlier attempt)
  - Combines symbolic execution, testing, model checking and theorem proving
  - Recent extensions: whitebox fuzz testing and SAGE
  - Killer app: security

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: 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, AppVerifier,…)

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 !

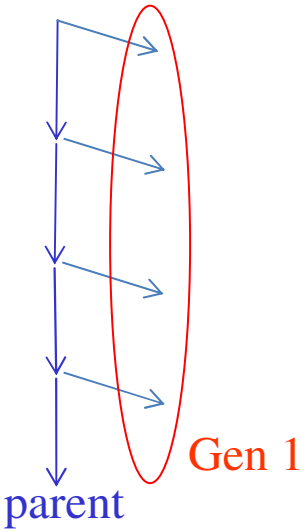
- Observations:
  - Dynamic test generation extends static test generation with additional runtime information: it is more powerful
  - The number of program paths can be infinite: may not terminate!
  - Still, DART works well for small programs (1,000s LOC)
  - Significantly improves code coverage vs. 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 DART implementations:
  - EXE/EGT (Stanford): independent ['05-'06] closely related work
  - CUTE = same as first 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
  - CatchConv (Berkeley) implements DART with focus on integer overflows
  - Splat (UCLA) implements DART with focus on fast detection of buffer overflows
  - Apollo (MIT) implements DART for testing web applications …and more!

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

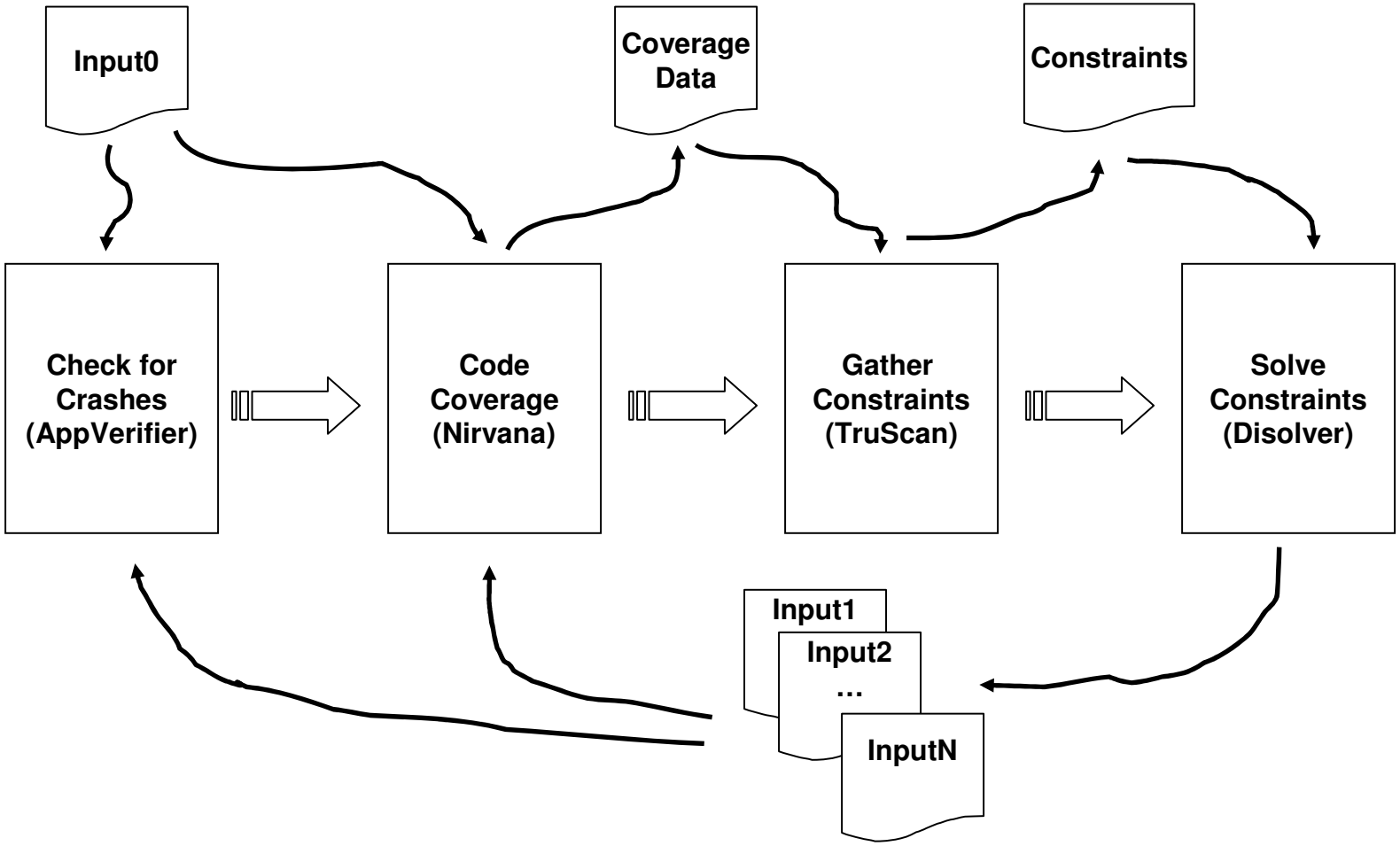


- Search spaces are huge, the search is partial… yet effective at finding bugs !

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 Results

Since April'07 1st release: tens of new security bugs found
(most 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"
- Credit is due to the entire SAGE team and users:
  - CSE: Michael Levin (DevLead), Christopher Marsh, Dennis Jeffries (intern'06), Adam Kiezun (intern'07)
  - MSR: Patrice Godefroid, David Molnar (intern'07)
  - Plus work of many users who found and filed most of these bugs!

Some Experiments

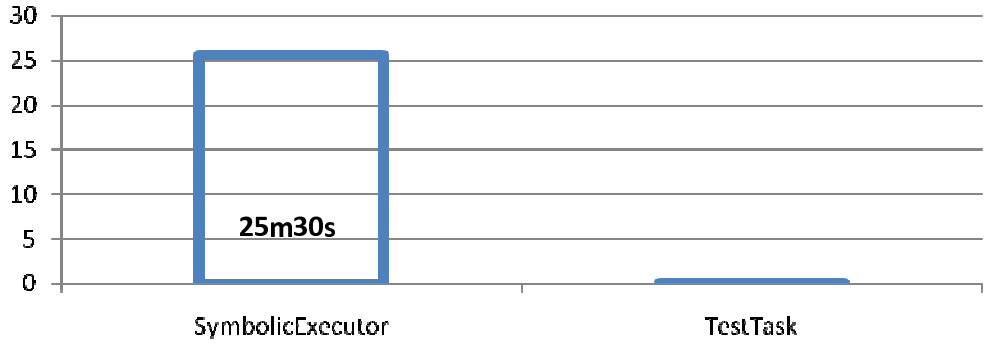
Most much (100x) bigger than ever tried before!

- 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

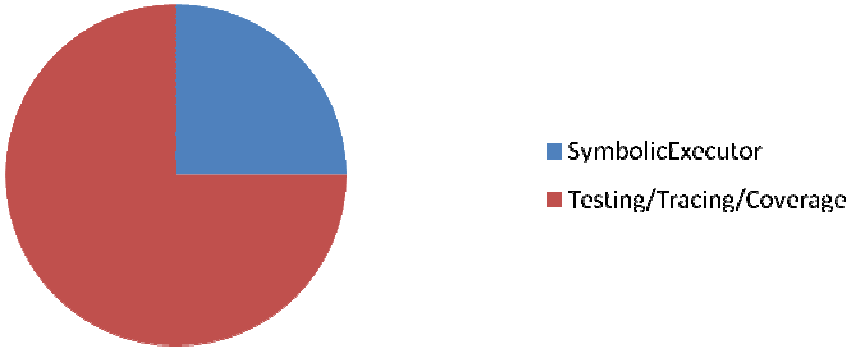
- Each symbolic execution is expensive



X1,000

- Yet, symbolic execution does not dominate search time

10 hours



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□aN
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

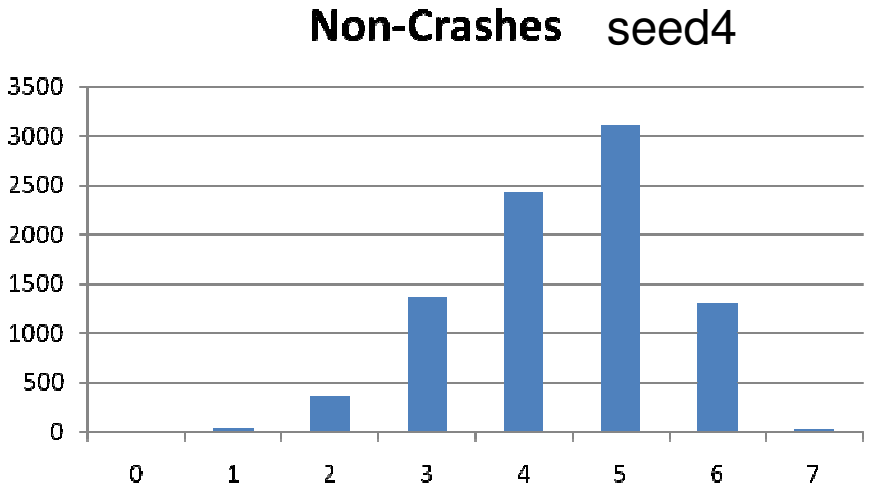
Different Seed Files, Different Crashes

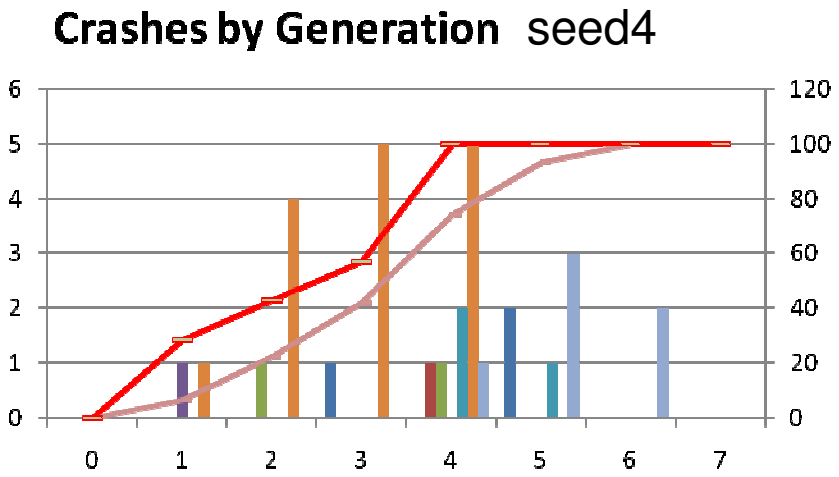
Bucket	seed1	seed2	seed3	seed4	seed5	100 zero bytes
1867196225	X	X	X	X	X	
2031962117	X	X	X	X	X	
612334691		X	X			
1061959981			X	X		
1212954973			X			X
1011628381			X	X		X
842674295				X		
1246509355			X	X		X
1527393075					X	
1277839407					X	
1951025690			X			

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"





Some Recent Extensions

- Scalability: compositional dynamic test generation
  - use function summaries like in interprocedural static analysis
  - If f calls g, test g separately, summarize the results, and use g's summary when testing f
  - Can get same path coverage exponentially faster!
- More checkers: active checkers to find more bugs
  - Ex: array reference **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 such constraints in an optimal way…
- Leverage input spec: grammar-based whitebox fuzzing
  - input precondition specified as a context-free grammar

SAGE Summary

- SAGE is so effective at finding bugs that we face "bug triage" issues with dynamic test generation for the first time

- What makes it so effective?
  - Works on large applications (not unit test)
  - Can detect bugs due to problems across components
  - Fully automated (current focus on file fuzzing)
  - Easy to deploy (x86 analysis – any language or build process)
  - Now, used in various groups inside Microsoft

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!

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)
    - Example of challenge: eradicate all buffer overflows in all MS codecs
  - How far can we go?

References

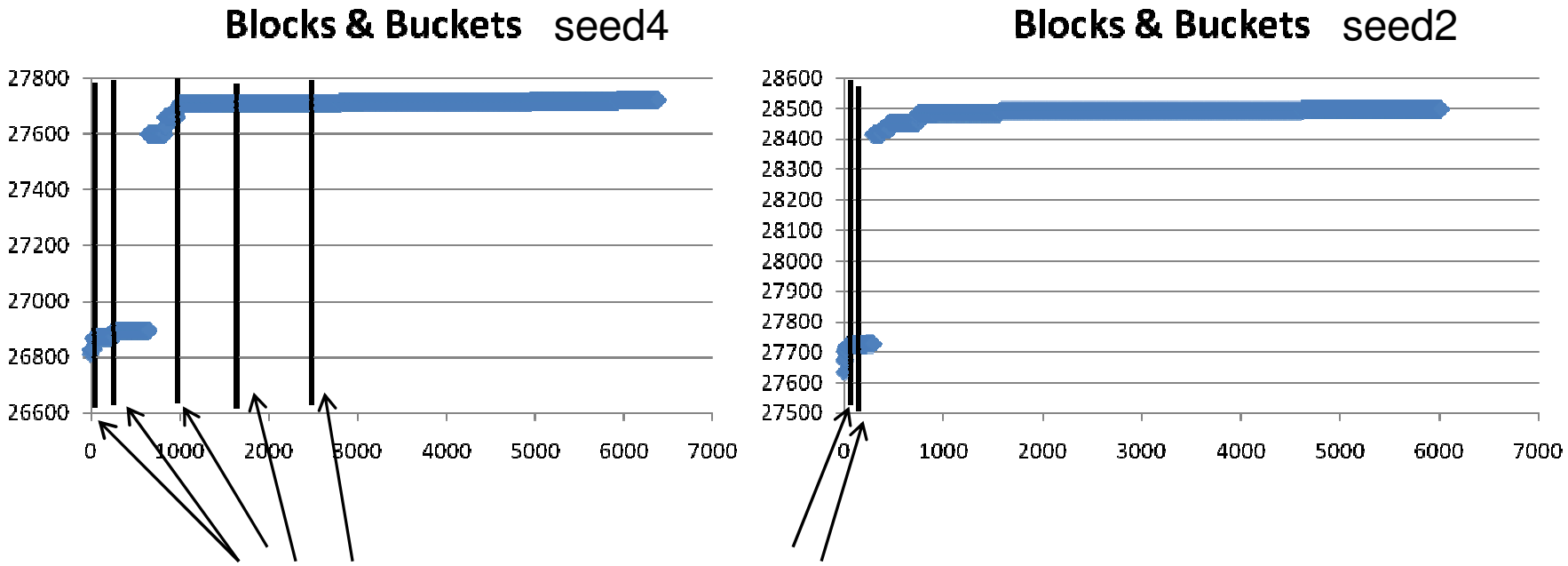
- see http://research.microsoft.com/users/pg
  - Model Checking for Programming Languages using VeriSoft, POPL'1997
  - DART: Directed Automated Random Testing, with N. Klarlund and K. Sen, PLDI'2005
  - Compositional Dynamic Test Generation, POPL'2007
  - Automated Whitebox Fuzz Testing, with M. Levin and D. Molnar, NDSS'2008
  - Grammar-based Whitebox Fuzzing, with A. Kiezun and M. Levin, to appear in PLDI'2008
  - Active Property Checking, with M. Levin and D. Molnar, MSR-TR-2007-138

Some Other Related Work in MSR

- Pex: automatic test generation to the desktop
  - Unit testing, OO languages, .NET managed code, VS integration
  - contracts, rich interfaces, mock-object creation, program repair
- Yogi: combine testing with static analysis
  - Testing is precise but incomplete
  - Static analysis is complete but imprecise
  - Focus on Windows device drivers

  } combine !
- More expressive constraint solvers (ex: Z3)
- Extend to concurrency (ex: CHESS)
- Etc. (active area of research)

Coverage and New Crashes: Low Correlation



New bucket found for the first time