**Lecture 2:**

**Software Model Checking via Systematic Testing**

**Dealing with Data Inputs**

Patrice Godefroid  
Microsoft Research

---

**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
- Many security exploits are initiated via files or packets
  - Ex: MS Windows includes parsers for hundreds 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.

---

**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
  - \( \rightarrow \) 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

---

**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:
  
<table>
<thead>
<tr>
<th>Modeling languages</th>
<th>State-space exploration</th>
<th>Model checking</th>
</tr>
</thead>
<tbody>
<tr>
<td>(SLAM, Bandera, FeaVer, BLAST, CBMC,...)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
  
<table>
<thead>
<tr>
<th>Programming languages</th>
<th>State-space exploration</th>
<th>Systematic testing</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Concurrency: VeriSoft, JPF, CMC, Bogor, CHESS,...</td>
</tr>
<tr>
<td>Data inputs: DART, EXE, SAGE,...</td>
<td></td>
<td>Killer app: security ( \rightarrow ) biggest impact to date!</td>
</tr>
</tbody>
</table>

---

**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
  - Simple yet effective: many bugs found this way...
  - At Microsoft, fuzzing is mandated by the SDL \( \rightarrow \)
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) {
    return 0;
  }
  ```

  Can’t statically generate values for x and y that satisfy “x==hash(y)”!

DART = Directed Automated Random Testing

- 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 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

DART Implementations

- Defined by symbolic execution, constraint generation and solving
  - Languages: C, Java, x86, .NET,
  - Theories: linear arithmetic, bit-vectors, arrays, uninterpreted functions,…
  - Solvers: lp_solve, CVLibe, 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
  - CatchCam (Berkeley) focus on integer overflows
  - Split (UCLA) focus on fast detection of buffer overflows
  - Apollo (MIT/IBM) 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)
  - “dualized” static analysis: static = tay 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

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

input = "good"

I

Negate each constraint in path constraint
Solve new constraint → new input

The Search Space

If symbolic execution is perfect and search space is small, this is verification.

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

- Seven applications - 10 hours search each

<table>
<thead>
<tr>
<th>App Tested</th>
<th>#Tests</th>
<th>Mean Depth</th>
<th>Mean #Instr</th>
<th>Mean Input Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANI</td>
<td>11468</td>
<td>178</td>
<td>2,066,087</td>
<td>5,400</td>
</tr>
<tr>
<td>Medi1</td>
<td>6890</td>
<td>73</td>
<td>3,409,376</td>
<td>65,536</td>
</tr>
<tr>
<td>Medi2</td>
<td>1045</td>
<td>1100</td>
<td>271,432,489</td>
<td>27,335</td>
</tr>
<tr>
<td>Medi3</td>
<td>2266</td>
<td>608</td>
<td>54,644,652</td>
<td>30,833</td>
</tr>
<tr>
<td>Medi4</td>
<td>909</td>
<td>883</td>
<td>133,685,240</td>
<td>22,209</td>
</tr>
<tr>
<td>Compressed File Format</td>
<td>1527</td>
<td>65</td>
<td>480,435</td>
<td>634</td>
</tr>
<tr>
<td>OfficeApp</td>
<td>3008</td>
<td>6502</td>
<td>923,731,248</td>
<td>45,064</td>
</tr>
</tbody>
</table>

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

Generational Search Leverages Symbolic Execution

- Each symbolic execution is expensive

- Yet, symbolic execution does not dominate search time

X3,000

10 hours
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 billions+ 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 Fuzzing Lab Bug Yield for Win7

• 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 was 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:

Generation 0 – seed file

Generation 1

Generation 2

Generation 3

Zero to Crash in 10 Generations

- Starting with 100 zero bytes ...
- SAGE generates a crashing test for Media1 parser:

Generation 0 – seed file

Generation 1

Generation 2

Generation 3
Zero to Crash in 10 Generations

- Starting with 100 zero bytes …
- SAGE generates a crashing test for Media1 parser:

**Generation 4**

- SAGE generates a crashing test for Media1 parser:

**Generation 5**

- SAGE generates a crashing test for Media1 parser:

**Generation 6**

- SAGE generates a crashing test for Media1 parser:

**Generation 7**

- SAGE generates a crashing test for Media1 parser:

**Generation 8**

- SAGE generates a crashing test for Media1 parser:

**Generation 9**

- SAGE generates a crashing test for Media1 parser:
Zero to Crash in 10 Generations

- Starting with 100 zero bytes ...
- SAGE generates a crashing test for Media1 parser:

  ```
  00000100h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  00000104h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  00000108h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0000010ch: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  00000110h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  00000114h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  00000118h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0000011ch: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  00000120h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  00000124h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  
  Generation 10 – crash bucket 1212954973!

```

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

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, like DART, EXE, etc.)
  - Can detect bugs due to problems across components
  - Fully automated (focus on file fuzzing)
  - Easy to deploy (x86 analysis - any language or build process!)
    - 1st tool for whole-program dynamic symbolic execution at x86 level
  - Now, used daily in various groups at Microsoft

More On the Research Behind SAGE

- How to recover from imprecision in symbolic exec.? PLDI’05, PLDI’11
  - Must under-approximate
- How to scale symbolic exec. to billions of instructions? NDSS’08
  - Techniques to deal with large path constraints
- How to check efficiently many properties together? EMSOFT’08
  - Active property checking
- How to leverage grammars for complex input formats? PLDI’08
  - Lift input constraints to the level of symbolic terminals in an input grammar
- How to deal with path explosion? POPL’07, TACAS’08, POPL’10, SAS’11
  - Symbolic test summaries (non-inter).
- How to reason precisely about pointers? ISSTA’09
  - How memory models leverage concrete memory addresses and regions
- How to deal with floating-point instructions? ISSTA’10
  - Prove "non-interference" with memory accesses
- How to deal with input-dependent loops? ISSTA’11
  - Automatic dynamic loop-invariant generation and summarization

Conclusion: Impact of SAGE (In Numbers)

- 200+ machine-years
  - Runs in the largest dedicated fuzzing lab in the world
- 1 Billion+ constraints
  - Largest computational usage ever for any SMT solver
- 100s of apps, 100s of bugs (missed by everything else)
- Bug fixes shipped quietly (no MSRCs) to 1 Billion+ PCs
- Millions of dollars saved
  - For Microsoft + time/energy savings for the world
- DART, Whitebox fuzzing now adopted by (many) others (10s tools, 100s citations)

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 diagnosis), Bunny (taint-flavor, may generate false alarms), Sunny-the-fuzzer (taint-flavor, 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.
  - Next: towards exhaustive testing of large applications (verification).
  - How far can we go?

Conclusion

- Software Model Checking via Systematic Testing
  - Lecture 2: Dealing with Data Inputs

    - Modeling languages → state-space exploration → Model checking
    - Programming languages → state-space exploration → Adaptation

    - (SLAM, Bandera, FuzzVoC, BLAST, CBMC,...)
    - (Concurency: VeriSoft, JPF, CMC, Bogor, CHESS,...)
    - Data inputs: DART, EXE, SAGE,...

Acknowledgments

- SAGE is joint work with:
  - MSR: Ella Bounimova, David Molnar,...
  - CSE: Michael Levin, Chris Marsh, Lei Fang, Stuart de Jong,...
  - Internes: Dennis Jeffries (06), David Molnar (07), Adam Kiezun (07), Bassem Elkarablieh (08), Marius Nita (08), Cindy Rubio-Gonzalez (08), Johannes Kinder (09), Daniel Luchaup (10), Nathan Rittenhouse (10), Mehdi Bouaziz (11),...

- Thanks to the entire SAGE team and users!
  - Z3 (MSR): Nikolaj Bjorner, Leonardo de Moura,...
  - Windows: Nick Bartman, Eric Douglas, Dustin Duran, Elmar Langholz, Isaac Sheldon, Dave Weston,...
  - Truscan support: Evan Tice, David Grant,...
  - Office: Tom Gallagher, Eric Jarvi, Octavian Timofte,...
  - SAGE users all across Microsoft!

- References: see http://research.microsoft.com/users/pg