500 Machine-Years of Software Model Checking and SMT Solving

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.
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
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:

```c
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, Valgrind, AppVerifier,...)
Example:

```java
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
    - see [DART in PLDI'05], [PLDI'11]
  - 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/IBM) for testing web applications

...and more!
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!
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();
}

Negate each constraint in path constraint
Solve new constraint $\rightarrow$ new input

Path constraint:

input = “good”

SMT solver

Gen 1

$\rightarrow$ SAT

good

$\rightarrow$ bood
good
goodd

goo!

$\rightarrow$ bood
good
goodd
goo!
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)
SAGE Architecture

Check for Crashes (AppVerifier) → Code Coverage (Nirvana) → Generate Constraints (TruScan) → Solve Constraints (Z3)

SAGE was mostly developed by CSE (2006-2008)

MSR algorithms & code inside (2006-2014)
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>Media1</td>
<td>6890</td>
<td>73</td>
<td>3,409,376</td>
<td>65,536</td>
</tr>
<tr>
<td>Media2</td>
<td>1045</td>
<td>1100</td>
<td>271,432,489</td>
<td>27,335</td>
</tr>
<tr>
<td>Media3</td>
<td>2266</td>
<td>608</td>
<td>54,644,652</td>
<td>30,833</td>
</tr>
<tr>
<td>Media4</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
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!
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!
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.? \textit{PLDI’05, PLDI’11}
  - Must under-approximations
- How to scale symbolic exec. to billions of instructions? \textit{NDSS’08}
  - Techniques to deal with large path constraints
- How to check efficiently \textbf{many properties} together? \textit{EMSOFT’08}
  - Active property checking
- How to leverage \textit{grammars} for \textbf{complex} input \textit{formats}? \textit{PLDI’08}
  - Lift input constraints to the level of symbolic terminals in an input grammar
- How to deal with \textbf{path explosion}? \textit{POPL’07, TACAS’08, POPL’10, SAS’11}
  - Symbolic test summaries (\textit{more later})
- How to reason precisely about \textbf{pointers}? \textit{ISSTA’09}
  - New memory models leveraging concrete memory addresses and regions
- How to deal with \textbf{floating-point instructions}? \textit{ISSTA’10}
  - Prove “non-interference” with memory accesses
- How to deal with input-dependent \textbf{loops}? \textit{ISSTA’11}
  - Automatic dynamic loop-invariant generation and summarization

\textbf{+} research on \textit{constraint solvers}
"Practical Verification"

• Since 2009: 500+ machine-years of running SAGE

• Practical goals:
  - Eradicate all remaining buffer overflows in all Windows parsers
    • Ex: <5 security bulletins in all the SAGE-cleaned Win7 parsers, 0 over the last 3 years
  - Reduce costs & risks for Microsoft, increase those for Black Hats!
    • Many have probably moved to greener pastures already...
      (Ex: Adobe, Java, Browsers,...)

• If nobody can find bugs in P, P is observationally equivalent to “verified”!

• This is “practical verification” or “security bug eradication”!
What Next? Towards “Verification”

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

- Two main approaches to software model checking:

<table>
<thead>
<tr>
<th>Programming languages</th>
<th>Modeling languages</th>
<th>Model checking</th>
</tr>
</thead>
<tbody>
<tr>
<td>state-space exploration</td>
<td></td>
<td></td>
</tr>
<tr>
<td>abstraction</td>
<td>(SLAM, Bandera,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>FeaVer, BLAST,...</td>
<td></td>
</tr>
</tbody>
</table>

Concurrency: VeriSoft, JPF, CMC, Bogor, CHESS,…
Data inputs: DART, EXE, SAGE,…
Exhaustive Testing?

- Model checking is always “up to some bound”
  - Limited (often finite) input domain, for specific properties, under some environment assumptions
    - Ex: exhaustive testing of Win JPEG parser up to 1,000 input bytes
      - 8000 bits → $2^{8000}$ possibilities → if 1 test per sec, $2^{8000}$ secs
      - FYI, 15 billion years = $473040000000000000$ secs = $2^{60}$ secs!
      → MUST be “symbolic”! 😊 How far can we go?

- This is “formal verification” (model checking)
How Far from “Formal Verification”? 

Two main problems:

1. Identify and patch holes in symbolic execution + constraint solving
   - Log unhandled input-tainted x86 instructions: Sagan
   - Extend symbolic execution engine manually
   - Or semi-automatically: see “Automated Synthesis of Symbolic Instruction Encodings from I/O Samples” [PLDI’12]

2. Tackle “path explosion”
From Program to Logic, Today

• **VC-gen/BMC**: one formula for the entire program
  - Tracks all (data+control) dependencies in one formula
  - Great when it works! (constraint solver faster than prg testing)
  - But does not scale to large programs!

• **DART**: one formula per program path
  - Tracks only input dependencies
  - Scales to long paths and large programs
  - But too many paths!

• Can we get the best of both worlds?
  - In theory, yes: compositional testing (symbolic test summaries)
  - In practice, the devil is in the details, and those are still open...
Compositionality = Scalability for Verification

- Idea: **compositional** dynamic test generation [POPL'07]
  - use **summaries** of individual functions (or program blocks, etc.)
    - like in interprocedural static analysis
    - but here "must" formulas generated dynamically
  - If f calls g, test g, summarize the results, and use g’s summary when testing f
  - A summary \( \varphi(g) \) is a disjunction of path constraints expressed in terms of g’s input preconditions and g’s output postconditions:
    \[
    \varphi(g) = \lor \varphi(w) \quad \text{with} \quad \varphi(w) = \text{pre}(w) \land \text{post}(w)
    \]
  - g’s outputs are treated as **fresh symbolic inputs** to f, all bound to prior inputs and can be “eliminated” (for test generation)

- Can provide same path coverage exponentially faster!
  - See details and refinements in [POPL'07,TACAS'08,POPL'10]
The Engineering of Test Summaries

• Systematically summarizing everywhere is foolish
  - Very expensive and not necessary (costs outweigh benefits)
  - Not scalable without user help (see work on VC-gen and BMC)

• Summarization on-demand: (100% algorithmic)
  - When? At search bottlenecks (with dynamic feedback loop)
  - Where? At simple interfaces (with simple data types)
  - How? With limited side-effects (to be manageable and “sound”)

• Goal: use summaries intelligently
  - How? In what form(s)?
    - Computed statically? [POPL’10, ISSTA’10]
Ex: ANI Windows Image Parser Verification

- The ANI Windows parser
  350+ fcts in 5 DLLs, parsing in ~110 fcts in 2 DLLs, core = 47 fcts in user32.dll

- Is "attacker memory safe"
  = no attacker-controllable buffer overflow

- How? Compositional exhaustive testing
  - "perfect" symbolic execution in SAGE
    (max precision, no divergences, no x86 incompleteness, no Z3 timeouts, etc.),
    - manual bounding of input-dependent loops
      (only ~10 input bytes + file size), and
    - 5 user-guided simple summaries

- And modulo fixing a few bugs... 😊

- 100% dynamic (=zero static analysis)

- 1st Windows image parser proved attacker memory safe

- See "Proving Memory Safety of the ANI Windows Image Parser using Compositional Exhaustive Testing", MSR-TR-2013-120, with intern Maria Christakis
Conclusion: Impact of SAGE (In Numbers)

- **500+ machine-years**
  - Runs in the largest dedicated fuzzing lab in the world

- **4 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-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)
What Next? Towards “Verification”

- Tracking all(?) sources of incompleteness
- Summaries (on-demand...) against path explosion
- How far can we go?
  - Practical Verification: yes!
  - Formal Verification?
- For history books?

Blackbox Fuzzing  Whitebox Fuzzing  “Verification”

“Practical V.”: yes!  “Formal V.”: ?
Acknowledgments

• Joint work with:
  - **MSR:** Ella Bounimova, David Molnar,…
  - **CSE:** Michael Levin, Chris Marsh, Lei Fang, Stuart de Jong,…
  - **Interns** Dennis Jeffries (06), David Molnar (07), Adam Kiezun (07), Bassem Elkarablieh (08), Marius Nita (08), Cindy Rubio-Gonzalez (08,09), Johannes Kinder (09), Daniel Luchaup (10), Nathan Rittenhouse (10), Mehdi Bouaziz (11), Ankur Taly (11), Louis Jachiet (12), Gennady Pekhimenko (12), Maria Christakis (13,14), Rijnard Van Tonder (14)…

• Thanks to the entire SAGE team and users!
  - **Z3** (MSR): Nikolaj Bjorner, Leonardo de Moura,…
  - **Windows:** Nick Bartmon, Eric Douglas, Dustin Duran, Elmar Langholz, Isaac Sheldon, Dave Weston,…
    - Windows 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](http://research.microsoft.com/users/pg)