From Blackbox Fuzzing to Whitebox Fuzzing towards Verification

Patrice Godefroid
Microsoft Research
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),
    Cindy Rubio-Gonzalez (08,09), Johannes Kinder (09),…

• Thanks to the entire SAGE team and users!
  - Z3: Nikolaj Bjorner, Leonardo de Moura,…
  - Windows: Nick Bartmon, Eric Douglas,…
  - Office: Tom Gallagher, Eric Jarvi, Octavian Timofte,…
  - SAGE users all across Microsoft!
References

- See http://research.microsoft.com/users/pg
  - DART: Directed Automated Random Testing, with Klarlund and Sen, PLDI’2005
  - Compositional Dynamic Test Generation, POPL’2007
  - Automated Whitebox Fuzz Testing, with Levin and Molnar, NDSS’2008
  - Demand-Driven Compositional Symbolic Execution, with Anand and Tillmann, TACAS’2008
  - Grammar-Based Whitebox Fuzzing, with Kiezun and Levin, PLDI’2008
  - Active Property Checking, with Levin and Molnar, EMSOFT’2008
  - Precise Pointer Reasoning for Dynamic Test Generation, with Elkarablieh and Levin, ISSTA’2009
  - Compositional May-Must Program Analysis: Unleashing The Power of Alternation, with Nori, Rajamani and Tetali, POPL’2010
  - Proving Memory Safety of Floating-Point Computations by Combining Static and Dynamic Program Analysis, with Kinder, ISSTA’2010
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
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
  - \( \rightarrow \) 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
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,...)
DART = Directed Automated Random Testing

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
  - The number of program paths can be infinite: may not terminate!
  - Still, DART works well for small programs (1,000s LOC)
  - Significantly improves code coverage vs. random testing
DART Implementations

• Defined by symbolic execution, constraint generation and solving
  - Languages: C, Java, x86, .NET,…
  - Theories: linear arith., bit-vectors, arrays, uninterpreted functions,…
  - Solvers: lp_solve, CVCLite, STP, Disolver, Z3,…

• Examples of tools/systems implementing DART:
  - EXE/EGT (Stanford): independent ['05-'06] closely related work
  - CUTE = same as first DART implementation done at Bell Labs
  - SAGE (CSE/MSR) for x86 binaries and merges it with “fuzz” testing for finding security bugs (more later)
  - PEX (MSR) for .NET binaries in conjunction with “parameterized-unit tests” for unit testing of .NET programs
  - YOGI (MSR) for checking the feasibility of program paths generated statically using a SLAM-like tool
  - Vigilante (MSR) for generating worm filters
  - BitScope (CMU/Berkeley) for malware analysis
  - CatchConv (Berkeley) focus on integer overflows
  - Splat (UCLA) focus on fast detection of buffer overflows
  - Apollo (MIT) for testing web applications

...and more!
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 >= 3) crash();
}

input = "good"
```

Path constraint:

- $I_0 !='b'$ \implies $I_0 = 'b'$
- $I_1 !='a'$ \implies $I_1 = 'a'$
- $I_2 !='d'$ \implies $I_2 = 'd'$
- $I_3 !='!'$ \implies $I_3 = '!'$

Negate each constraint in path constraint
Solve new constraint $\implies$ new input
void top(char input[4])
{
    int cnt = 0;
    if (input[0] == 'b') cnt++;
    if (input[1] == 'a') cnt++;
    if (input[2] == 'd') cnt++;
    if (input[3] == '!') cnt++;
    if (cnt >= 3) crash();
}
SAGE (Scalable Automated Guided Execution)

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

SAGE was mostly developed by CSE
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!
WEX Fuzzing Lab Bug Yield for Win7

How fuzzing bugs found (2006-2009):

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

SAGE is running 24/7 on 100s machines: “the largest usage ever of any SMT solver”
N. Bjorner + L. de Moura (MSR, Z3 authors)
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 execution? PLDI’05
  - Must under-approximations

- 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
  - Symbolic test summaries (more later)

- How to reason precisely about pointers? ISSTA’09
  - New memory models leveraging concrete memory addresses and regions

- How to deal with floating-point instructions? ISSTA’10
  - Prove “non-interference” with memory accesses

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

    ![Diagram showing the relationship between modeling languages, programming languages, state-space exploration, model checking, systematic testing, and adaptations.

    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 Win7 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 = 4730400000000000000000000 secs = $2^{60}$ secs!
      → MUST be “symbolic”! 😊 How far can we go?

- **Practical goals: (easier?)**
  - Eradicate all remaining buffer overflows in all Windows parsers
  - Better coverage guarantees to justify “no new bug found”
  - Reduce costs & risks for Microsoft: when to stop fuzzing?
  - Increase costs & risks for Black Hats!
    - Many have probably moved to greener pastures already... (Ex: Adobe)
    - Ex: <5 security bulletins in all the SAGE-cleaned Win7 parsers
    - If noone can find bugs in P, P is observationally equivalent to “verified”!
How to Get There?

1. Identify and patch holes in symbolic execution + constraint solving

2. Tackle “path explosion” with compositional testing and symbolic test summaries [POPL’07, TACAS’08, POPL’10]

→ Fuzzing in the (Virtual) Cloud (Sagan)

- New centralized server collecting stats from all SAGE runs!
- Track results (bugs, concrete & symbolic test coverage), incompleteness (unhandled tainted x86 instructions, Z3 timeouts, divergences, etc.)
- Help troubleshooting (SAGE has 100+ options...)
- Tell us what works and what does not
The Art of Constraint Generation

- **Static analysis**: abstract away “irrelevant” details
  - Good for focused search, can be combined with DART (Ex: [POPL’10])
  - But for bit-precise analysis of low-level code (function pointers, in-lined assembly,...)? In a non-property-guided setting? Open problem...

- **Bit-precise VC-gen**: statically generate 1 formula from a program
  - Good to prove complex properties of small programs (units)
  - Does not scale (huge formula encodings), asks too much of the user

- **SAT/SMT-based “Bounded Model Checking”**: stripped-down VC-gen
  - Emphasis on automation
  - Unrolling all loops is naïve, does not scale

- **“DART”**: the only option today for large programs (Ex: Excel)
  - Path-by-path exploration is naïve, but “whitebox fuzzing” can scale it to large executions (Z3 is not the bottleneck) + zero false alarms!
  - But suffers from “path explosion”...
DART is Beautiful

- Generates formulas where the only “free” symbolic variables are whole-program inputs
  - When generating tests, one can only control inputs!

- **Strength: scalability to large programs**
  - Only tracks “direct” input dependencies (i.e., tests on inputs); the rest of the execution is handled with the best constant-propagation engine ever: running the code on the computer!
  - (The size of) path constraints only depend on (the number of) program tests on inputs, not on the size of the program
    = the right metric: complexity only depends on nondeterminism!

- **Price to pay: “path explosion”** [POPL’07]
  - Solution = symbolic test summaries
Example

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

Path constraint:
input = “good”

I_0!='b'
I_1!='a'
I_2!='d'
I_3!='!'
Compositionality = Key to Scalability

• 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 \( \phi(g) \) is a disjunction of path constraints expressed in terms of \( g \)'s input preconditions and \( g \)'s output postconditions:
    \[
    \phi(g) = \bigvee \phi(w) \quad \text{with} \quad \phi(w) = \text{pre}(w) \land \text{post}(w)
    \]
  - \( g \)'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)
  - Don’t fall into the “VC-gen or BMC traps” ! 😊

• 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
  - THE KEY to scalable bit-precise whole-program analysis ?
    • It is necessary! But in what form(s)? Is it sufficient?
    • Stay tuned...
Summaries Cure Search Redundancy

- Across different program paths
- Across different program versions
  - [“Incremental Compositional Dynamic Test Generation”, with S. Lahiri and C. Rubio-Gonzalez, MSR TR, Feb 2010]
- Across different applications
- Summaries avoid unnecessary work
- What if central server of summaries for all code?... Sagan 2.0
Conclusion: Impact of SAGE (In Numbers)

• 100+ machine-years  
  - Runs in the largest dedicated fuzzing lab in the world

• 100+ million constraints  
  - Largest computational usage ever for any SMT solver

• 100s of apps, 100s of bugs (missed by everything else)

• Bug fixes shipped to 1 Billion+ computers worldwide

• 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)
Myth: Blackbox is Cheaper than Whitebox

- Wrong!
  - A whitebox fuzzer is slower than a blackbox fuzzer
    - Whitebox needs more time to generate new tests (since smarter)
  - But over time, blackbox is more expensive than whitebox
    - Blackbox does not know when to (soundly) stop!

Example:
```c
int magic(int input)
{
    if (input==645723) error();
    return 0;
}
```

![Graph showing coverage versus time with whitebox and blackbox strategies.](chart.png)
What Next? Towards Verification

- Tracking all(?) sources of incompleteness
- Summaries (on-demand...) against path explosion
- How far can we go?
  - Reduce costs & risks for Microsoft: when to stop fuzzing?
  - Increase costs & risks for Black Hats (goal already achieved?)
- For history books:


Blackbox Fuzzing Whitebox Fuzzing Verification

There is one thing stronger than all the armies in the world; and that is an idea whose time has come. -- Victor Hugo
- For history books:

<table>
<thead>
<tr>
<th>Year</th>
<th>2000</th>
<th>2005</th>
<th>2010</th>
<th>2015</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Blackbox Fuzzing</td>
<td>Whitebox Fuzzing</td>
<td>Verification</td>
<td></td>
</tr>
</tbody>
</table>

*There is one thing stronger than all the armies in the world; and that is an idea whose time has come. -- Victor Hugo*