Dynamic Software Model Checking for Security

Patrice Godefroid

Microsoft Research
“Model Checking” (~1981)

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

- **Once thoroughly checked, models can be compiled and used as the core of the implementation**
Insight: Model Checking is Super Testing

• Simple yet effective technique for finding bugs

![Graph showing cost vs. coverage with model checking, verification, and testing categories.](Image)
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:

  Modeling languages  \[\xrightarrow{\text{state-space exploration}}\] Model checking
  \[\xrightarrow{\text{abstraction}}\] (SLAM, Bandera, CBMC, FeaVer, BLAST, …)

  Programming languages  \[\xrightarrow{\text{state-space exploration}}\] Systematic testing
  \[\xrightarrow{\text{adaptation}}\] Concurrency: VeriSoft, JPF, CMC, CHESS, Cuzz,…
  Data inputs: DART, EXE, SAGE, Pex,…

Dynamic Killer app: security → biggest impact to date!
Dynamic Software Model Checking

Dealing with Data Inputs
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”

Example: Powerpnt.exe <filename>

- Millions of lines of C/C++, complex input format, dynamic memory allocation, data structures of various shapes and sizes, pointers, loops, procedures, libraries, system calls, etc.
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), use **dynamic symbolic execution**, 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 blend with model checking!
  - 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,…)

EPFL Page 8 October 2018
DART = Directed Automated Random Testing

Example:

```c
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 arithmetic, 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 (became KLEE)
  - **CUTE** = same as first DART implementation done at Bell Labs
  - **SAGE** (MSR) for x86 binaries and merges it with “fuzz” testing for finding security bugs (more next)
  - **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 many more!
An Application: SAGE @ Microsoft

• #1 application of SMT solvers today (CPU usage)

• Why? Security Testing

• 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 vulnerabilities are in file & packet parsers
  - 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 [NDSS’08]

Idea: mix fuzz testing with dynamic test generation

- Dynamic symbolic execution to collect constraints on inputs, negate those, solve new constraints to get new tests, repeat → “systematic dynamic test generation” (= DART)

( Why dynamic ? Because most precise ! [PLDI’05, PLDI’11] )

- Apply 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
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"

Path constraint:

```
I₀!=`b' → I₀=`b'
I₁!=`a' → I₁=`a'
I₂!=`d' → I₂=`d'
I₃!=`!' → I₃=`!'
```

Negate each constraint in path constraint
Solve new constraint → new input
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 >= 4) crash();
}

If symbolic execution is perfect and search space is small, this is verification!
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>Excel</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!
SAGE (Scalable Automated Guided Execution)

- Whitebox fuzzing 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)

Input0 → Coverage Data → Constraints

Input1 → Input2 → … → InputN
Since 2007: many new security bugs found (missed by blackbox fuzzers, static analysis)

- Apps: image decoders, media players, document processors,…
- 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 1 billion+ computers)
  - ~1/3 of all fuzzing bugs found by SAGE!
Impact of SAGE (in Numbers)

• 1000+ machine-years
  - Ran in the world-largest dedicated fuzzing lab, now in the cloud
  - 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 and the world)

• “Practical Verification”:
  - Eradicate all buffer overflows in all Windows parsers
    - <5 security bulletins in all SAGE-cleaned Win7 parsers, 0 since 2011
    - If nobody can find bugs in P, P is observationally equiv to “verified”!
    - Reduce costs & risks for Microsoft, increase those for Black Hats

<table>
<thead>
<tr>
<th>Year</th>
<th>Blackbox Fuzzing</th>
<th>Whitebox Fuzzing</th>
<th>“Practical Verification”</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2005</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2010</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2015</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
More On the Research Behind SAGE

- How to recover from imprecision in symbolic exec.? PLDI'05, PLDI'11
- How to scale symbolic exec. to billions of instructions? NDSS'08
- How to check efficiently many properties together? EMSOFT'08
- How to leverage grammars for complex input formats? PLDI'08
- How to deal with path explosion? POPL'07, TACAS'08, POPL'10, SAS'11
- How to reason precisely about pointers? ISSTA'09
- How to deal with floating-point instructions? ISSTA'10
- How to deal with input-dependent loops? ISSTA'11
- How to synthesize x86 circuits automatically? PLDI'12
- How to run 24/7 for months at a time? ICSE'13

+ research on constraint solvers

References: see http://research.microsoft.com/users/pg
What Next? Open Problems

Towards Formal Verification

- When can we safely stop testing?
- When we know that there are no more bugs! = “Verification”
- Two main problems:
  • Imperfect symbolic execution and constraint solving
  • Path explosion
- Active area of research...

Terra Incognita

scalability

testing

Dynamic SMC (SAGE, ...)

verification

Static SMC (BMC, PA, ...)

From Program to Logic

• VC-gen/BMC: one formula for the entire program
  - Tracks all (data+control) dependencies in one formula
  - Great when applicable, but does not scale to large programs!

• Dynamic Symbolic Execution: one formula per program path
  - Tracks only input dependencies
  - Scales to long paths in large programs, but too many paths!

• Trade-off: compositional dynamic test generation \[^{[POPL'07]}\]
  - use symbolic test summaries of single functions (or prgm blocks,...)
    • like in interprocedural static analysis
    • but here “must” formulas generated dynamically and incrementally
    • A summary is a disjunction of intraprocedural path constraints
  - In theory, can provide same path coverage exponentially faster!
  - In practice, heavy machinery... Worth the trouble?
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

Other Ideas

• **Combine with static analysis** [POPL’2010, ISSTA’2010]
  - For property-guided device-driver verification (YOGI): yes! 😊
  - For memory safety (SAGE): no… 😞
    • Not property guided (memory accesses everywhere!)
    • No sound static analysis
    • Too complicated, not enough benefits…

• **Incremental analysis** [SAS’2011]
  - Re-analyze only program parts which have changed
  - In practice, too complicated, not enough benefits…
Other Ideas (Continued)

- **Machine Learning for Input Fuzzing [ASE’2017]**
  - How to learn (generative) input grammars using RNNs
  - Case study: the PDF-object format (~2/3 of 1,300 pages)

  ![PDF data objects of different types](image)

  ![A char-RNN model to generate PDF objects](image)

  - Takeaway: better learning does not imply better fuzzing
  - More research needed, just scratching the surface...

- **Other problems of interest:**
  - How to secure network and cloud services
  - How to measure security
Project Springfield (2015-)

• A simple idea:
  - centralized fuzzing at Microsoft (starting ~2006)
  - Let’s package centralized fuzzing as a cloud service!
  - Project Springfield = 1st commercial fuzzing service in the cloud

What is Project Springfield?

Project Springfield is Microsoft’s unique fuzz testing service for finding security critical bugs in software. Project Springfield helps customers quickly adopt practices and technology battle-tested over the last 15 years at Microsoft.

"Million Dollar" Bugs

Battle tested tech

Fast, Consistent Roll-out

Available now

Goal = build, sell and operate a cloud fuzzing service (FaaS)
How does Springfield work?

1. Upload Binaries
   One-time configuration and sample inputs

2. Run multiple fuzzers
   Get data on every run

3. Identify high-value bugs
   Report actionable test cases

4. Fix bugs
   Review report and take action
Why Fuzzing in The Cloud

- **Faster**: easier adoption, time to market
- **Cheaper**: shared dev costs, no upfront cost, pay as you go
- **Better**: centralization (better quality, big data optimizations, more scenarios hence more security), elasticity (compute)

Is there a market? **Yes** - testing the market by shipping
  - Private preview in 2015, public since Sep 2016
  - Outsourcing fuzzing to Springfield: higher quality at lower cost

How Big is the market? **Unclear but security is booming**

May 2017: Springfield is now **Microsoft Security Risk Detection**

Example: **better beer!** ([video link](#))

How Springfield helped **OSIsoft** help **Deschutes Brewery**
Finally, a Secure Beer!

microsoft.com/Springfield

May 2017: Springfield is now Microsoft Security Risk Detection
Thank You!

Questions?