Lecture 4

Advanced Topics

Combining Static and Dynamic Software Model Checking

Patrice Godefroid

Microsoft Research
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
    - (SLAM, Bandera, FeaVer, BLAST, CBMC, YOGI,...)
    - state-space exploration
  - Programming languages
    - state-space exploration
  - Model checking
    - Systematic testing
    - adaptation
  - Lecture 3
    - abstraction

Lecture 1: Concurrency: VeriSoft, JPF, CMC, Bogor, CHESS,...
Lecture 2: Data inputs: DART, EXE, SAGE, PEX,...
Lecture 4: Advanced Topics
Recap: Software Model Checking Approaches

Two complementary approaches to software model checking:

- **Modeling languages**
  - **Model checking**
    - State-space exploration
  - **abstraction**
  - **Modeling languages**

- **Programming languages**
  - **Systematic testing**
  - **Systematic testing**
    - **adaptation**
    - State-space exploration

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

• SMASH: Compositional May-Must Program Analysis: Unleashing the Power of Alternation [POPL’10, with Aditya Nori, Sriram Rajamani, Sai Deep Tetali]

• Proving Memory Safety of Floating-Point Computations by Combining Static and Dynamic Program Analysis [ISSTA’10, with Johannes Kinder]

• Higher-Order Test Generation [PLDI’11]

• Summary of all 4 lectures
Compositional May-Must Program Analysis:

Unleashing the Power of Alternation

[POPL’10]

P. Godefroid, A. Nori, S. Rajamani, S. Tetali
Compositional May/Must Program Analysis

- **May**: over-approximation
  - Sound proofs

- **Must**: under-approximation
  - Sound bugs

- **May/Must**: 3-valued world (Sound bugs and proofs!)
  - How connected?
    - Shared abstract states (Modal Transition Systems, etc.)
    - Shared transitions: Synergy/Dash (more later)

- **Compositional May/Must**: (this paper)
  - memoize intermediate results as may/must summaries
  - Allows fine-grained coupling and alternation
Tests

void bar(bool a, bool b, int limit)
{
  int i=0, lock=1, x=0, y=0;
  while (i < limit) {
    if (a)
      x = x+1;
    else
      x = x-1;
    if (b)
      y = y+1;
    else
      y = y-1;
    i = i+1;
  }
  if (lock != 1)
    error();
}

a→true
b→false
limit→2

---

Patrice Godefroid  
Page 7  
August 2012
void bar(bool a, bool b, int limit) {
    int i=0, lock=1, x=0, y=0;
    while (i < limit) {
        if (a) {
            x = x+1;
            else {
                x = x-1;
        }
        if (b) {
            y = y+1;
            else {
                y = y-1;
        }
        i = i+1;
    }
    if (lock != 1) {
        error();
    }
}
An Algorithm: SMASH = Compositional DASH

- (Not-)May = predicate abstraction (as in SLAM)
- Must = symbolic execution (precise, whole-program path, as in DART)
- **Frontier**: Boundary between tested and untested regions (as in Synergy/DASH)
  - Intersection of the not-may (backward) and must (forward) abstractions
  - Extend the frontier → the not-may and must abstractions are refined in one step
May-Must analysis
May-Must analysis

\[ \begin{align*}
\phi_1 \in \Pi_{n_1} & \quad \phi_2 \in \Pi_{n_2} \\
\Omega_{n_1} \cap \phi_1 & \neq \emptyset \quad \Omega_{n_2} \cap \phi_2 = \emptyset \\
e & = (n_1, n_2) \in E_P \\
\Omega_{n_2} \cap \phi_2 & = \emptyset \\
\theta & \subseteq \text{Post}(\Gamma_e, \Omega_{n_1} \cap \phi_1) \\
\varphi_2 \cap \theta & \neq \emptyset \\
\Omega_{n_2} & := \Omega_{n_2} \cup \theta \\
\end{align*} \]

[MUST – POST]
May-Must analysis

\[ \varphi_1 \in \Pi_{n_1} \quad \varphi_2 \in \Pi_{n_2} \quad e = (n_1, n_2) \in E_p \]

\[ \Omega_{n_1} \cap \varphi_1 \neq \emptyset \quad \Omega_{n_2} \cap \varphi_2 = \emptyset \quad \theta \supseteq Pre(\Gamma_{e}, \varphi_2) \quad \theta \cap \Omega_{n_1} = \emptyset \]

\[ \Pi_{n_1} \coloneqq (\Pi_{n_1} \setminus \{\varphi_1\}) \cup \{\varphi_1 \cap \theta, \varphi_1 \cap \neg \theta\} \quad N_e \coloneqq N_e \cup \{\varphi_1 \cap \neg \theta, \varphi_2\} \]

[NOTMAY – PRE]

- Synergy/Dash [FSE ’06, ISSTA ‘08]
An Algorithm: SMASH = Compositional DASH

- (Not-)May = predicate abstraction (as in SLAM)
- Must = symbolic execution (precise, whole-program path, as in DART)
- Frontier: Boundary between tested and untested regions (as in Synergy/DASH)
  - Intersection of the not-may (backward) and must (forward) abstractions
  - Extend the frontier → the not-may and must abstractions are refined in one step
- SMASH = Compositional DASH
  - Do DASH intraprocedurally
  - Memoize and re-use may/must summaries
SMASH is implemented in YOGI (in SDV)

Experiments with 69 Win7 device drivers (342KLOC), 85 properties

We have unleashed the power of alternation!
Summary

• **SMASH** is a unified framework for compositional may-must program analysis

• We have explained **SMASH** in the context of existing analyses (SLAM, DART, Synergy/Dash ...) in the area

• Empirical evaluation shows that **SMASH** can significantly outperform may-only, must-only and non-compositional may-must algorithms

• [http://research.microsoft.com/yogi](http://research.microsoft.com/yogi)
Remarks

- **C code is first abstractly interpreted (= simplified)**
  - No pointer arithmetic (e.g., *(p+1) is treated as *p)
    - Strictly speaking, neither sound nor complete (as in SLAM)
  - Logic encoding: propositional logic, linear arithmetic and uninterpreted functions

- The environment is modeled abstractly (as in SDV)

- Each property is checked one by one
  - This is a “property-guided” setting (unlike DART and SAGE)
Proving Memory Safety of Floating-Point Computations by Combining Static and Dynamic Program Analysis

[ISSTA'10]

Patrice Godefroid        Johannes Kinder
SAGE: Current Limitations

• Symbolic execution is incomplete - a full implementation for x86 would have to model hundreds of instructions
  - Floating point
  - SIMD extensions (Intel SSE, SSE2, SSE3, ...)

• Input data that passes through these instructions will not show up in the path constraint

• Branches cannot be explored, bugs could be missed!

• These kinds of instructions are commonly used in media codecs: is this an issue?
Naïve Handling of FP / SIMD code

• Extend **bit-precise** symbolic execution to include all instructions and additional registers

• But: Z3 cannot even reason about floating point numbers, so extend that, too!

• **Collect and solve even more constraints, worsen path explosion problem**

Will this really help us find more bugs?
Header vs. Payload

- Bugs in media parsers are usually due to malformed header information about offsets, sizes, etc.

- Processing of the payload should not interfere with address calculations
  - like “data independence” in protocol verification [Wolper86]

- Intuition:
  - FP code is used for data-processing
  - Non-FP code is used for buffer allocation and indexing

- Idea: prove memory safety of FP code AND non-interference between FP and security critical non-FP code
  - SAGE can catch all unsafe memory accesses without understanding FP code
  - What level of precision is needed for the static analysis?
Example

- Floating point instructions invisible to SAGE

```c
void process(double *inBuf, double *outBuf, int i)
{
    ...
    outBuf[i] = 2.5 * inBuf[i];
    ...
}
```

```asm
...  
mov esi, [ebp + 8] ;inBuf
mov edi, [ebp + 12] ;outBuf
...
mov eax, [ebp + 16] ;i
fld qword ptr [const2.5] ;2.5
fld [esi + eax] ;inBuf[i]
fmulp ;*
fstp qword ptr [edi+eax] ;outBuf[i]
...  
```
Example

• Floating point instructions invisible to SAGE

• An FP instruction is **memory safe** if its addresses are within bounds

```c
void process(double *inBuf, double *outBuf, int i)
{
    ...
    outBuf[i] = 2.5 * inBuf[i];
    ...
}
```

```assembly
... mov esi, [ebp + 8] ;inBuf
mov edi, [ebp + 12] ;outBuf
...
mov eax, [ebp + 16] ;i
fld qword ptr [const2.5] ;2.5
fld [esi + eax] ;inBuf[i]
fmulp ;*
fstp qword ptr [edi+eax] ;outBuf[i]
...```
Example

- Floating point instructions invisible to SAGE
- An FP instruction is memory safe if its addresses are within bounds
- An FP instruction cannot interfere with critical code, if its output is never used for
  - Computing an address
  - Conditional jumps

```c
void process(double *inBuf, double *outBuff, int i)
{
    ...
    outBuff[i] = 2.5 * inBuff[i];
    ...
}
```

```assembly
... 
mov esi, [ebp + 8] ;inBuf
mov edi, [ebp + 12] ;outBuff
...
mov eax, [ebp + 16] ;i
fld qword ptr [const2.5] ;2.5
fld [esi + eax] ;inBuff[i]
fmlp ;*
fstp qword ptr [edi+eax] ;outBuff[i]
...
```
Example

- Floating point instructions are invisible to SAGE.
- An FP instruction is memory safe if its addresses are within bounds.
- An FP instruction cannot interfere with critical code, if its output is never used for:
  - Computing an address
  - Conditional jumps

```c
void process(double *inBuf, double *outBuf, int i) {
    …
    outBuf[i] = 2.5 * inBuf[i];
    …
}
```

```assembly
mov esi, [ebp + 8]      ;inBuf
mov edi, [ebp + 12]     ;outBuf
…
mov eax, [ebp + 16]     ;i
fld qword ptr [const2.5] ;2.5
fld [esi + eax]         ;inBuf[i]
fmulp
*fstp qword ptr [edi+eax] ;outBuf[i]
…
```
Statically Compute Pre/Postconditions

• Statically generate pre/postconditions for FP instrs:
  - Preconditions for memory safety: list of registers used to compute addresses
    • Non-float dependent values: handled by SAGE's bounds checker
    • Float dependent values: (not handled \(\rightarrow\)) report \textbf{unsafe}!
  - Postconditions for side effects: list of registers and memory locations that become float dependent will be dynamically tagged by SAGE as “FP-tag”

Static Precondition

\textbf{Non-float:}\ 
\{ edi, eax \}

\textbf{Static Postcondition}

\textbf{Float:}\ 
\{[edi + eax]: 8\}

\textbf{fstp qword ptr [edi+eax]}
Dynamically Check & Enforce at Runtime

• Dynamically check preconditions for memory safety
  - Preconditions disallow addressing using FP-tagged variables

• Dynamically enforce postconditions:
  - Tag variables at runtime as float dependent “FP-tag” (i.e., unusable in constraint solving)
  - Variables tagged as FP-tag propagate through SAGE’s regular symbolic execution when handled by non-FP code
  - **Disallow** conditional jumps that depend on FP-tagged variables (by reporting **unsafe**!)

Too restrictive since floating point values are used in conditional statements: too many “unsafe” are reported!
Example – Floating Point Conditional

• SAGE cannot flip branches that depend on floating point values

• Do we really need to flip the branch to prove memory safety?

• Memory safety of the complete block depends only on \( i \)

• Idea: same scheme but applied to FP-tainted conditional blocks!
  • Precondition for entire block: \( i \) is neither float nor “input dependent”
    • That is, \( i \) has no symbolic value
    • = new “attacker memory safety” (=“not directly attacker-controllable”)

• Postconditions require only to taint \( \text{outBuf}[i] \) with FP-tag
Block Summaries by Static Analysis

- **Statically**, for every conditional jump, summarize pre- and postconditions in both branches
  - Every value that is modified becomes FP-tagged (control dependence)
  - For function calls inside branches, recursively generate function summaries

- **Dynamically**, for every FP-tag dependent conditional jumps, SAGE
  - reads the static block summary and checks the preconditions for memory safety of the entire block
  - injects FP tags at the immediate postdominator for all variables in the postcondition
Old Symbolic Execution - Regular SAGE

- Start with concrete trace containing several conditional jumps which have either been taken or not
- Generate constraints over inputs for conditionals
- Symbolic values generalize concrete ones
- Note: Only some conditionals are input dependent and become part of the path constraint
New FP-Aware Symbolic Execution

- Start with concrete trace
- When encountering FP code, tag variables according to postconditions
- For FP dependent conditionals, use statically generated may-summary to over-approximate ALL executions of the if-then-else block
Real World Issues

- C-runtime contains some SSE optimized assembly code

- SSE2 versions of memzero and memcpy
  - Used depending on memory alignment
  - Handle non-payload data, inject lots of FP-tags, lots of false alarms!

- memzero
  - Assigns a constants, no tag needed
  - Extend static analysis to detect that

- memcpy
  - Detect memcpy idiom in binary
  - Special postcondition for copying input and FP tags

Memzero: Clear 128 bytes per iteration

```
loc_70D19BFD:
  movdqa oword ptr [edi], xmm0
  movdqa oword ptr [edi+10h], xmm0
  movdqa oword ptr [edi+20h], xmm0
  movdqa oword ptr [edi+30h], xmm0
  movdqa oword ptr [edi+40h], xmm0
  movdqa oword ptr [edi+50h], xmm0
  movdqa oword ptr [edi+60h], xmm0
  movdqa oword ptr [edi+70h], xmm0
  lea   edi, [edi+80h]
  dec   ecx
  jnz   short loc_70D19BFD
```
Experimental Results - Static Analysis

<table>
<thead>
<tr>
<th></th>
<th>DLLs</th>
<th>All instr.</th>
<th>FP instr.</th>
<th>Conditionals</th>
<th>Safe</th>
<th>Cond. Safe</th>
<th>Unsafe</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>JPEG</strong></td>
<td>16</td>
<td>2,127,862</td>
<td>15,334</td>
<td>212,158</td>
<td>6.4%</td>
<td>9.8%</td>
<td>83.8%</td>
<td>418s</td>
</tr>
<tr>
<td><strong>GIF</strong></td>
<td>19</td>
<td>2,860,801</td>
<td>41,455</td>
<td>275,635</td>
<td>6.4%</td>
<td>11.1%</td>
<td>82.5%</td>
<td>623s</td>
</tr>
<tr>
<td><strong>ANI</strong></td>
<td>15</td>
<td>1,753,916</td>
<td>7,774</td>
<td>172,652</td>
<td>5.5%</td>
<td>11.7%</td>
<td>82.8%</td>
<td>306s</td>
</tr>
</tbody>
</table>

- 14 DLLs shared by all three parsers
- All conditionals processed, only dynamic analysis discriminates FP / non-FP
  - Safe: Precondition is true
  - Unsafe: Precondition is false
  - Conditionally safe: otherwise
Experimental Results - Dynamic Analysis

- 12 different seed files (~1Kbytes) per format, 1 execution per file

- JPEG & GIF: unsafe warnings appear before any input is read - not attacker controllable, therefore safe

- ANI: Same warnings, but after input is read (math error handler)

- Runtime overhead: ~20% compared to regular symbolic execution
Limitations

• Depends on soundness of SAGE
  - symbolic execution of the integer part

• Depends on soundness of Vulcan
  - Dominator information inaccurate
  - Control flow information sometimes unreliable

• Control flow through exceptions is not supported by the static analysis
Conclusions and Future Work

• Proving memory safety of floating point computations by combining a lightweight static analysis and a precise dynamic analysis (SAGE)

• Analysis of FP computations for JPEG, GIF, ANI
  - Static analysis quickly pre-processes large binaries
  - Intuition was correct - FP computations did not interfere with memory safety

• Future work:
  - More experiments!
  - New bugs?
  - Identify other opportunities for cheap over-approximation instead of exploring precise paths
Higher-Order Test Generation

[PLDI'2011]
Why **Dynamic** Test Gen.? Most Precise!

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 \((x != 567) \lor (x != \text{hash}(y))\)
- constraint too complex
  - simplify it: \(x \neq 567\)
- 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:
- “Unknown/complex symbolic expressions can be simplified using concrete runtime values” [DART, PLDI’05]
- Let’s call this step “concretization” (ex: hash(y) → 567)
- **Dynamic test generation extends static test generation** with additional runtime information: it is **more powerful**

How often? When exactly? Why? → this work!
Unsound and Sound Concretization

• Concretization is not always sound

```c
int foo(int x, int y) {
    if (x==hash(y)) {
        if (y==10) error();
    }
} ...
```

Run: `x=567, y=42`
```
pc: x==567 and y!=10
New pc: x==567 and y==10
New inputs: x=567, y=10
Divergence!
```

pc and new pc are unsound!

• Definition: A path constraint pc for a path w is sound if every input satisfying pc defines an execution following w

• Sound concretization: add concretization constraints

```c
pc: y==42 and x==567 and y!=10   (sound)
```
```
New pc: y==42 and x==567 and y==10 (sound)
```

• Theorem: path constraint is now always sound. Is this better? No

  - Forces us to detect all sources of imprecision (expensive/impossible…)
  - Can prevent test generation and “good” divergences
Idea: Using Uninterpreted Functions

• Modeling imprecision with uninterpreted functions
  
  int obscure(int x, int y) {  
    if (x==hash(y)) error();  
    return 0;  
  }

  Run: x=33, y=42
  pc: x != h(y)
  New pc: x == h(y)

• How to generate tests?
  - Is (\exists x,y,h:) x=h(y) SAT? Yes, but so what? (ex: x=y=0, h(0)=0)
  - Need universal quantification!

    (\forall h:) \exists x,y: x=h(y)  is this first-order logic formula valid?
    Yes. Solution (strategy): “fix y, set x to the value of h(y)”

• Test generation from validity proofs! (not SAT models)
  - Necessary but not sufficient: what “value of h(y)”?

  Run:
  x=33, y=42
  pc: x != h(y)
  New pc: x == h(y)
Need for Uninterpreted Function Samples

• Record I/O UF samples
  ```c
  int obscure(int x, int y) {
    if (x==hash(y)) error();
    return 0;
  }
  ```
  Run: x=33, y=42
  Record: 567 == h(42)
  pc: x!=h(y)

• Use UF samples to interpret a validity proof/strategy
  - “fix y, set x to the value of h(y)” → set y=42, x=567

• Or new pc: (∀h:) (∃x,y: (567=h(42)) => (x=h(y))) is valid?

• Higher-order test generation =
  - models imprecision using Uninterpreted Functions
  - records UF samples as concrete input/output value pairs
  - generates tests from validity proofs of FOL formulas
  Key: a “higher-order” logic representation of path constraints
Higher-Order Test Generation is Powerful

- Theorem: HOTG is as powerful as sound concretization
  - Can simulate it (both UFs and UF samples are needed for this)

- Higher-Order Test Generation is more powerful

  Ex 1: \((\forall h:\) \exists x, y: h(x) = h(y)\) is valid (solution: set \(x = y\))

  Ex 2: \((\forall h:\) \exists x, y: h(x) = h(y) + 1\) is invalid

    But \((\forall h:\) \exists x, y: (h(0) = 0 \land h(1) = 1) \implies h(x) = h(y) + 1\) is valid

      (solution: set \(x = 1, y = 0\))

  Ex 3:

  ```
  int foo(int x, int y) {
    if (x == hash(y)) {
      if (y == 10) error();
    } ...
  }
  ```

  Run: \(x = 567, y = 42\)

  pc: \(x == h(y)\) and \(y != 10\)

  New pc: \((\forall h:\) \exists x, y: (h(42) = 567) \implies x == h(y) \land y = 10\)

  is valid. Solution: set \(y = 10\), set \(x = h(10)\)

  2-step test generation:

    - run1 with \(y = 10, x = 567\) to learn \(h(10) = 66\)
    - run2 with \(y = 10, x = 66\)!
Implementability Issues

- Tracking **all** sources of imprecision is problematic
  - Excel on a 45K input bytes executes 1 billion x86 instructions

- Imprecision cannot always be represented by UF
  - Unknown input/output signatures, nondeterminism,…

- Capturing all input/output pairs can be very expensive

- Limited support from current SMT solvers
  - $\exists X: \Phi(F,X) \text{ is valid if and only if } \forall X: \neg \Phi(F,X) \text{ is UNSAT}$
  - Little support for generating+parsing UNSAT ‘saturation’ proofs

- In practice, HOTG can be used for **targeted** reasoning about specific user-defined complex/unknown functions
Application: Lexers with Hash Functions

- **Parsers with input lexers using hash functions for fast keyword recognition**
  
  Initially, forall language keywords: addsym(keyword, hashtable)  
  When parsing the input:  
  X=findsym(inputChunk, hashtable); // is inputChunk in hashtable?  
  if (x==52) ... // how to get here?

- **With higher-order test generation:**
  
  - Represent hashfunct by one UF h  
  - Capture all pairs (hashvalue,h(keyword))  
  - If “h(inputChunk)==52” and “(52,h('while'))” -> inputChunk='while'  
  - This effectively inverses hashfunct only for all keywords  
  - Sufficient to drive executions through the lexer!  
  - See [PLDI'2011] for details
Other Related Work

• Modeling imprecision with UFs is well-known in program verification of universal properties
  - “may” abstractions, universal quantifiers only, validity checks
  - Here, novelty is for existential properties

• Test generation is only one way to verify existential properties of programs
  - More generally, one can build “must” abstractions
  - Alternation $\forall \exists$ is also used then

• Test generation as a game is not new
  - in model-based testing, testing for reactive systems, etc.
  - but from validity proofs of FOL formulas with UFs is new
Summary (see [PLDI'2011])

• Higher-order test generation = UFs + UF samples + test generation from validity checks of FOL formulas

• A new powerful form of test generation

• Tracking all sources of incompleteness is unrealistic, targeted use of UFs is more practical (ex: lexer with h)

• A formal tool to define the limits of test generation

• Explains in what sense dynamic test generation is more powerful than static test generation
  - only in its ability to record concrete values in path constraints
  - concrete values are ultimately needed for test generation
Summary: 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:

Lecture 1 —> Concurrency: VeriSoft, JPF, CMC, Bogor, CHESS,…
Lecture 2 —> Data inputs: DART, EXE, SAGE, PEX,…
Lecture 3 —> abstractions
Lecture 4: Advanced Topics —> adaptation

Modeling languages —> state-space exploration —> Model checking
Programming languages —> state-space exploration —> Systematic testing

(SLAM, Bandera, FeaVer, BLAST, CBMC, YOGI,...)
Summary of Lecture 1

• SMC via Systematic Testing - Dealing with Concurrency
  - How to systematically explore the state spaces of concurrent reactive software implementations, dynamically?
    • With dynamic semantics and a runtime scheduler - no static analysis
  - How to efficiently do so without storing any visited states?
    • With partial-order reduction (POR)
  - How to find million-dollar bugs in telecommunication products?
    • With VeriSoft
  - What are the strengths and weaknesses of this SMC approach?
  - How to extend this approach to multi-threaded software?
    • Dynamic POR, preemption bounding, concurrency heuristics, etc.
Summary of Lecture 2

• SMC via Systematic Testing - Dealing with Data Inputs
  - How to systematically explore efficiently the state spaces of sequential programs to find bugs due to malformed inputs?
    • With symbolic execution and dynamic test generation
  - Why is dynamic test generation more powerful than static?
    • Leverage concrete runtime values to build must abstractions
  - Can it scale to large programs (like Microsoft Excel)?
    • Yes, when extended to whitebox fuzzing
  - What is today’s largest computational usage of SMT solvers?
    • SAGE @ Microsoft
      - 400 machine years, Billions of constraints, runs 24/7 on 200+ machines
      - Ex: 1/3 of all file fuzzing bugs found during Windows 7 development
      - Millions of $ saved by avoiding expensive security patches
Summary of Lecture 3

• Software Model Checking via Abstraction
  - How to systematically explore state spaces symbolically?
    • By encoding sets of states using logic formulas
  - How to abstract programs and then refine those abstractions?
    • With predicate abstraction and automatic abstraction refinement
  - How to check the correctness of Windows device drivers?
    • With SLAM and SDV
  - How to abstract programs for verification and finding bugs?
    • By using 3-valued May/Must abstractions and logics
  - Can 3-valued model checking be more precise?
    • Yes, with “Generalized Model Checking”
      = checking whether there exists an implementation of a given abstraction
      that satisfies a given temporal logic formula
Summary of Lecture 4

- Can static and dynamic software model checking be combined?
  - Yes, for example as in YOGI (part of Microsoft’s SDV)

- Are there other ways to combine static and dynamic analyses?
  - Yes, for instance, to (may) over-approximate floating-point computations in (must) dynamic symbolic execution

- What is the most precise form of test generation known today?
  - Dynamic test generation
  - Higher-order test generation
Conclusion: Software Model Checking

- Several dimensions:
  - Proofs vs. Bugs (verification vs. testing)
  - May vs. Must (universal vs. existential)
  - Static vs. Dynamic (abstract and/or concrete)

- Dijkstra vs. Model Checking
  - “Testing can only prove the existence of bugs, not their absence.”
  - Unless it is exhaustive! This is the “model checking thesis”
  - In practice, verification is not binary: it is a continuum