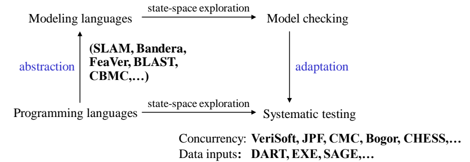
Lecture 5: Combining Static and Dynamic Software Model Checking

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:



Concurrency: **VeriSoft, JPF, CMC, Bogor, CHESS,…**
Data inputs: **DART, EXE, SAGE,…**

Overview

Note: DART: combines program analysis, testing, model checking and constraint solving (theorem proving)

- 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 [with Johannes Kinder]

Compositional May-Must Program Analysis:

Unleashing the Power of Alternation

[POPL'10]

P. Godefroid, A. Nori, S. Rajamani, S. Tetali

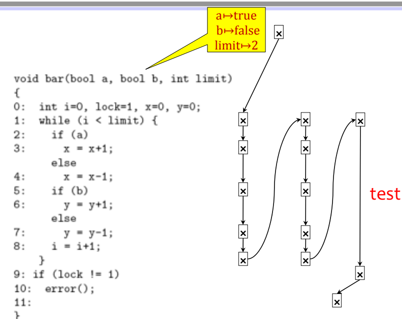
Compositional May/Must Program Analysis

- 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

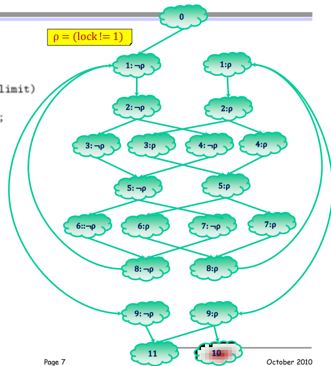
a↦true
b↦false
limit↦2

```
void bar(bool a, bool b, int limit)
{
0:  int i=0, lock=1, x=0, y=0;
1:  while (i < limit) {
2:    if (a)
3:      x = x+1;
      else
4:      x = x-1;
5:    if (b)
6:      y = y+1;
      else
7:      y = y-1;
8:    i = i+1;
    }
9:  if (lock != 1)
10:   error();
11:
}
```



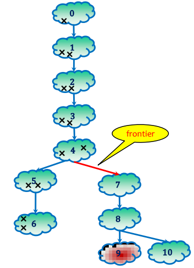
Proofs

```
void bar(bool a, bool b, int limit)
{
0:  int i=0, lock=1, x=0, y=0;
1:  while (i < limit) {
2:    if (a)
3:      x = x+1;
      else
4:      x = x-1;
5:    if (b)
6:      y = y+1;
      else
7:      y = y-1;
8:    i = i+1;
    }
9: if (lock != 1)
10:  error();
11:
}
```

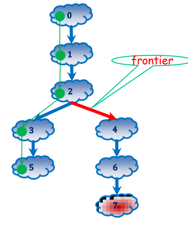


An Algorithm: SMASH = Compositional DASH

- (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

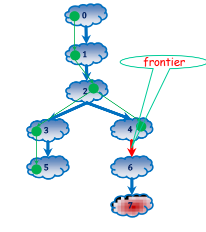


May-Must analysis



May-Must analysis

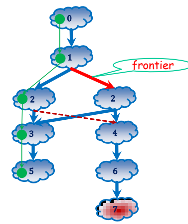
( )



May-Must analysis

( )

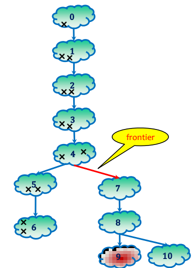
({ }) { } ( )



- 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

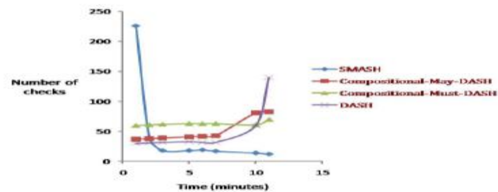


Figure 11. Comparison of SMASH with COMPOSITIONAL-MAY-DASH, COMPOSITIONAL-MUST-DASH and DASH on 303 checks.

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

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

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

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

```
…
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

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

```
…
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

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

```
…
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

Discharge to dynamic analysis as preconditions

- 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

Discharge to dynamic analysis as postconditions

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

```
…
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 →) report unsafe!

Static Precondition: Non-float: { edi, eax }

`fstp qword ptr [edi+eax]`

Static Postcondition: Float: {[edi + eax] : 8}

  - Postconditions for side effects: list of registers and memory locations that become float dependent will be dynamically tagged by SAGE as "FP-tag"

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?

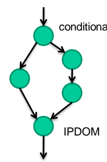
```
void process(double *inBuf, double *outBuf, int size)
{
 outBuf[i] = 2.5 * inBuf[i];

  if (outBuf[i] > 10.3)
  {
     outBuf[i] = inBuf[i] / 4.3;
  }
          …
}
```

- 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 `outBuf[i]` with FP-tag

Block Summaries by Static Analysis

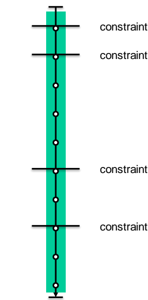
- Statically, for every conditional jump, summarize pre- and postconditions in both branches
  - Every value that is modified becomes FP-tagged (control dependence)
  - For function calls inside branches, recursively generate function summaries



- 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

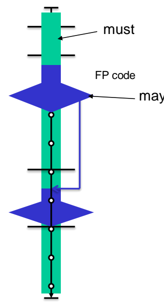
Old Symbolic Execution – Regular SAGE

- 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
```

true false

Experimental Results – Static Analysis

	DLLs	All instr.	FP instr.	Conditio-nals	Safe	Cond. Safe	Unsafe	Time
JPEG	16	2,127,862	15,334	212,158	6.4%	9.8%	83.8%	418s
GIF	19	2,860,801	41,455	275,635	6.4%	11.1%	82.5%	623s
ANI	15	1,753,916	7,774	172,652	5.5%	11.7%	82.8%	306s

- 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

		All instr.		FP instr.		Total FP cond.		Safe FP cond.		Unsafe FP cnd.	
		Full	Input	Full	Input	Full	Input	Full	Input	Full	Input
PEG	Occurr	26,712,705	21,983,468	7,826	7,320	45	4	39 (87%)	4	6 (13%)	0
	Unique	86,763		104	89	28	1	26 (93%)	1	2 (7%)	0
IF	Occurr	8,952,406	4,786,801	3,856	0	435	0	299(69%)	0	136 (31%)	0
	Unique	133,958		68	0	36	0	32 (89%)	0	4 (11%)	0
NI	Occurr	1,581,268	1,207,886	134	39	41	21	35 (85%)	15	6 (15%)	6
	Unique	29,722		16	13	27	7	25 (93%)	5	2 (7%)	2

- 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