Static and Dynamic Program Analysis to Improve Code Reliability and Security

Yulei Sui

http://yuleisui.github.io

Faculty of Engineering and Information Technology
University of Technology Sydney, Australia

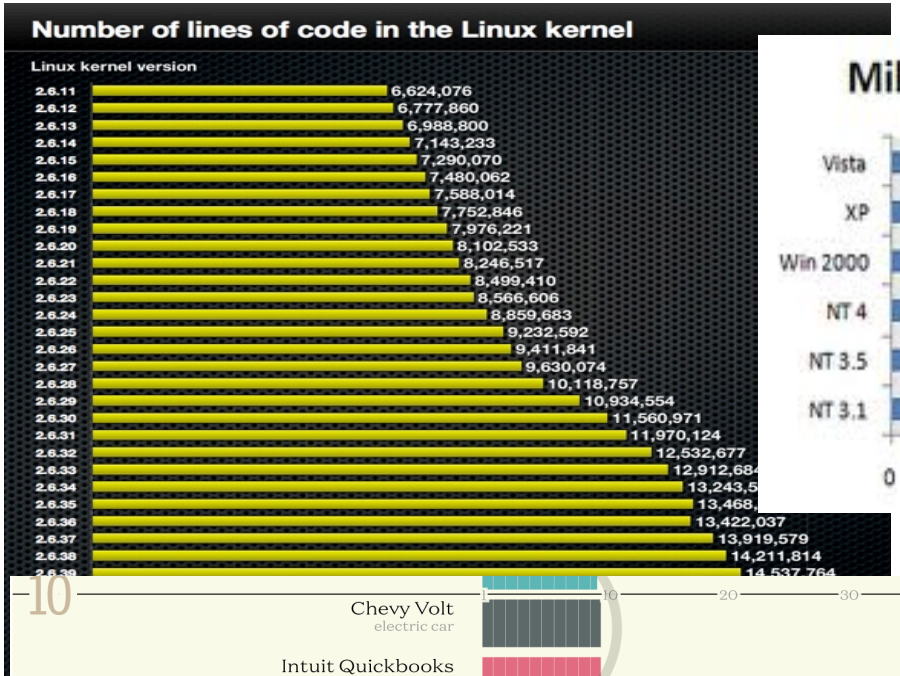
August 14, 2018

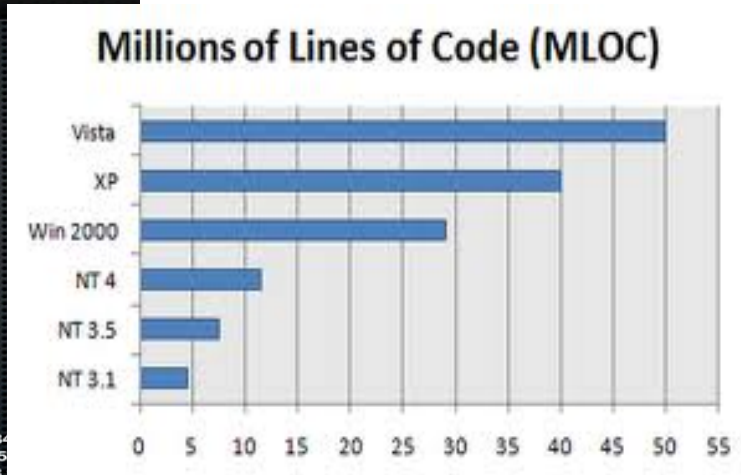
Outline

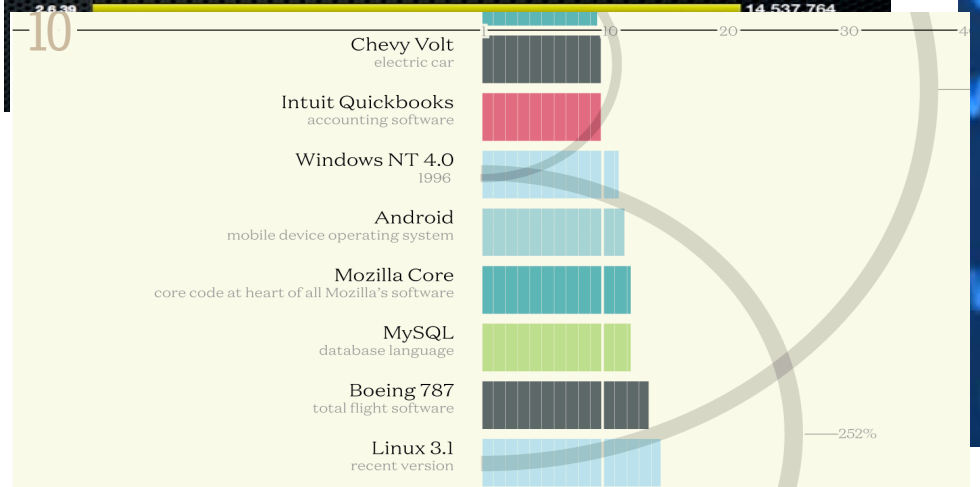
- **Existing software bugs and vulnerabilities**
- **Static analysis and dynamic analysis**
- **Technical contributions in developing scalable and precise program analysis**
  - Fundamental program analysis
    - Sparse value-flow analysis
    - Selective and on-demand value-flow analysis
  - Applications
    - Static value-flow analysis for detecting memory errors
    - Hybrid value-flow analysis to enforce memory safety and control-flow integrity
- **Research opportunities**

Software Is Everywhere

Software Becomes Larger and More Complex







Software Becomes More Buggy

floating point error

28 soldiers died

buffer overflow

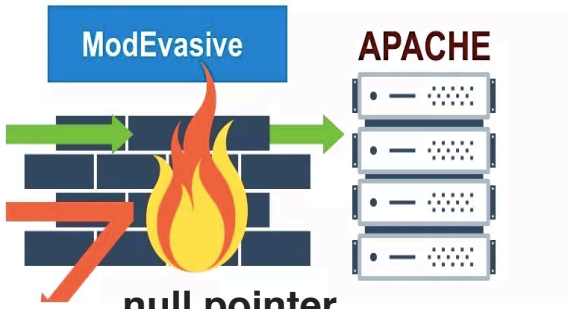
66% websites affected

uninitialized variables

password leakage via *tar* on Solaris OS

use-after-free

exploit price up to $100k per bug in Chrome



null pointer

denial of service affecting millions of servers worldwide

data race

11 civilians died

Memory Leak

- A dynamically allocated object is not freed along some execution path of the program
- A major concern of long running server apps due to gradual loss of available memory

```
1  /* Samba --libads/ldap.c:ads_leave_realm */
2     host = strdup(hostname);
3      if (...)  {...; return ADS_ERROR_SYSTEM(ENOENT);} // The programmer forgot to free host on error.
4
```

```
1  for (...) {
2      char* buf = readBuffer();
3      if (condition)
4          printf (buf);
5      else
6          continue;   // buf is leaked in else branch
7      freeBuf(buf);
8  }
9
10
```

Buffer Overflow

- Attempts to put more data in a buffer than it can hold.
- Program crashes, undefined behavior or zero-day exploit[1].

```
1   void verfiyPassword(){
2     char buff[15]; int pass = 0;
3      printf ("\n Enter the password : \n");
4     gets(buff);
5
6      if (strcmp(buff, "thegeekstuff")){     // return 0 if buffer overrun
7           printf ("\n Wrong Password \n");
8     }
9     else{
10          printf ("\n Correct Password \n");
11         pass = 1;
12    }
13     if (pass)
14          printf ("\n Root privileges given to the user \n");
15  }
16
```

[1] Heartbleed is a buffer overflow security bug disclosed in April 2014 in the OpenSSL (It took more than 2 years to discover and fix it since first patch, and over 500,000 websites were affected). Vulnerability is exploited when more data can be read than should be allowed.

Uninitialized Variable

- Stack variables in C and C++ are not initialized by default.
- Undefined behavior or zero-day exploit.

```
1  void drawCoordinate(){
2          switch (ctl) {
3                  case -1:
4                          xN = 0; yN = 0;
5                          break;
6                  case 0:
7                          xN = i; yN = -i;
8                          break;
9                  case 1:
10                         xN = i + NEXT_SZ; yN = i - NEXT_SZ;
11                         break;
12                 default:
13                         xN = -1; xN = -1; // xN is accidentally set twice while yN is uninitialized
14                         break;
15         }
16                 repaint(xN, yN);
17 }
18
19
```

Use After Free

- Attempt to access memory after it has been freed.
- Program crashes, undefined behavior or zero-day exploit.

```
1  /* heap based use-after-free*/
2  char* ptr = (char*) malloc (SIZE);
3  ...
4  if (err) {
5          abrt = 1;
6          free(ptr);
7  }
8  ...
9  if (abrt) {
10         logError("operation aborted before commit", ptr); // try to access released heap variable
11 }
```

My Past Work on Software Vulnerability Detection and Program Optimizations

program analysis techniques that can efficiently and precisely detect bugs and optimize program performance in the context of large-scale software.

- Fundamental Program Analyses
  - *Value-flow analysis* (CC'16, FSE '16, ICSE '18)
  - *Pointer analysis for optimizations* (CGO '13, SPE '13)
  - *Java reflection analysis* (ECOOP '14)
  - *SIMD vectorization* (LCTES '16, TECS '18)
  - *Probabilistic programming* (SAS '17)
- Software Bug Detection
  - *Memory leaks* (ISSTA '12, TSE '14, SAC '16)
  - *Uninitialized variables* (CGO '14, FSE '16)
  - *Buffer overflows* (ISSRE '14, TR '15)
  - *Use-after-frees* (ACSAC '17, ICSE '18)
  - *Concurrency bugs* (CGO '16, PMAM '16)
  - *Control-flow integrity protection* (ISSTA '17, ACISP '18)
  - *Mobile app analysis* (IToF '15, ICSE '18)

Outline

- **Existing software bugs and vulnerabilities**
- **Static analysis and dynamic analysis**
- **Technical contributions in developing scalable and precise program analysis**
  - Fundamental program analysis
    - Sparse value-flow analysis
    - Selective and on-demand value-flow analysis
  - Applications
    - Static value-flow analysis for detecting memory errors
    - Hybrid value-flow analysis to enforce memory safety and virtual call integrity
- **Research opportunities**

Static Analysis vs. Dynamic Analysis

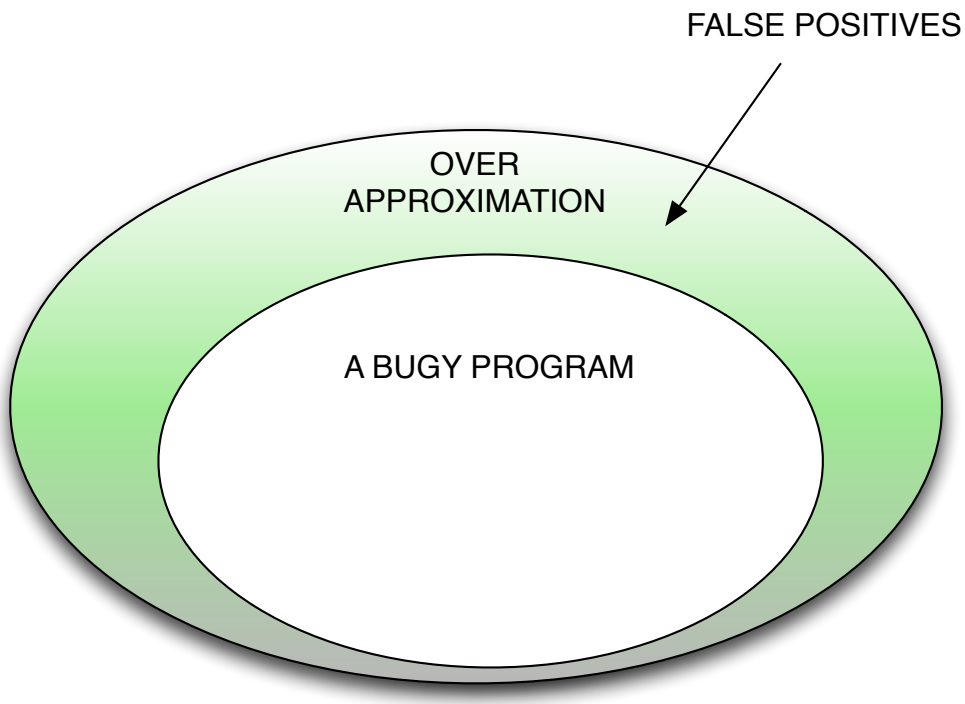
Static Analysis

- *Analyze a program without actually executing it – inspection of its source code by examining all possible program paths*
  - + Pin-point bugs at source code level.
  - + Catch bugs at early the stage of the software development cycle.
  - - False alarms due to over-approximation.
  - - Scalability issue of precise analysis for large size programs.

Dynamic Analysis

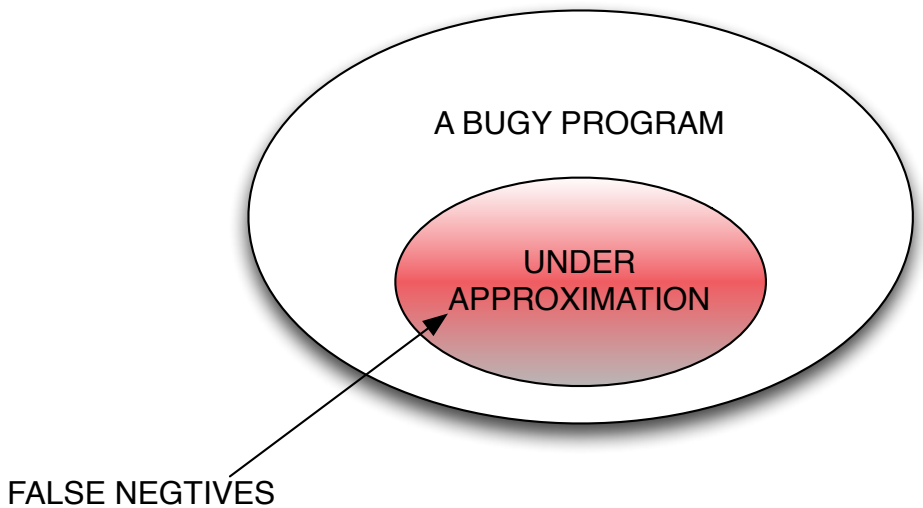
- *Analyze a program at runtime – inspection of its running program by examining some executable paths depending on test inputs*
  - + Identify bugs at runtime (catch it when you observe it).
  - + Zero or low false alarm rates.
  - - May miss bugs (false negative) due to under-approximation.
  - - Runtime overhead due to code instrumentation.

Bug Detection Philosophy



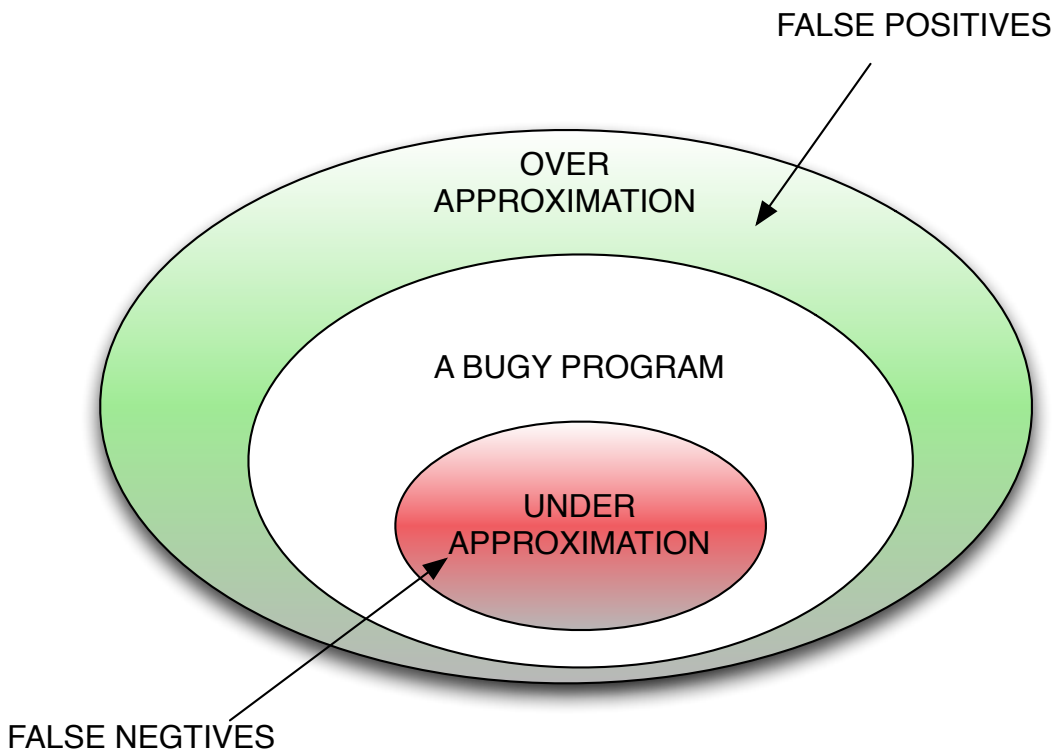
- Soundness : Over-Approximation (Static Analysis)
- Completeness : Under-Approximation (Dynamic Analysis)

Bug Detection Philosophy



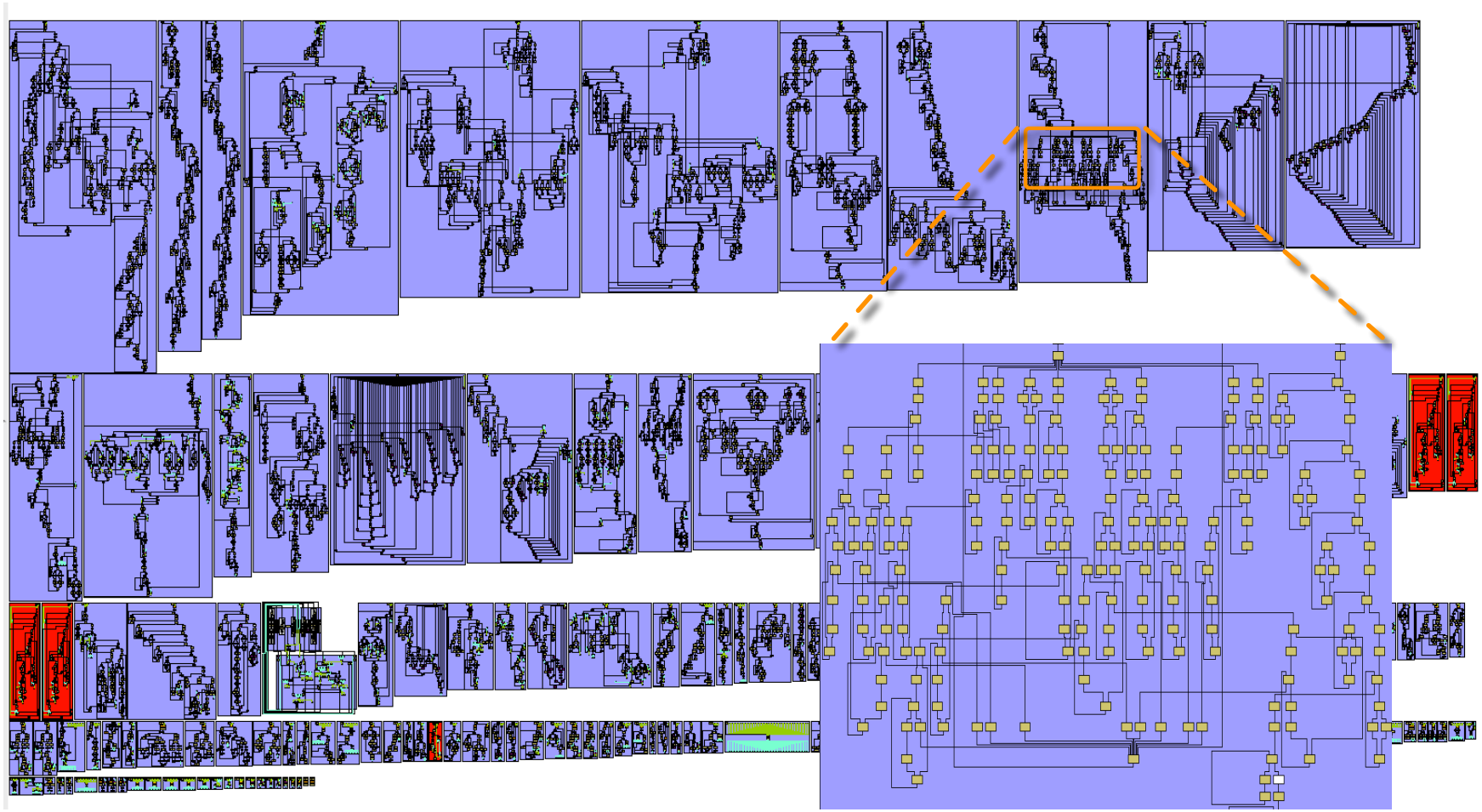
- Soundness : Over-Approximation (Static Analysis)
- Completeness : Under-Approximation (Dynamic Analysis)

Bug Detection Philosophy



- Soundness : Over-Approximation (Static Analysis)
- Completeness : Under-Approximation (Dynamic Analysis)

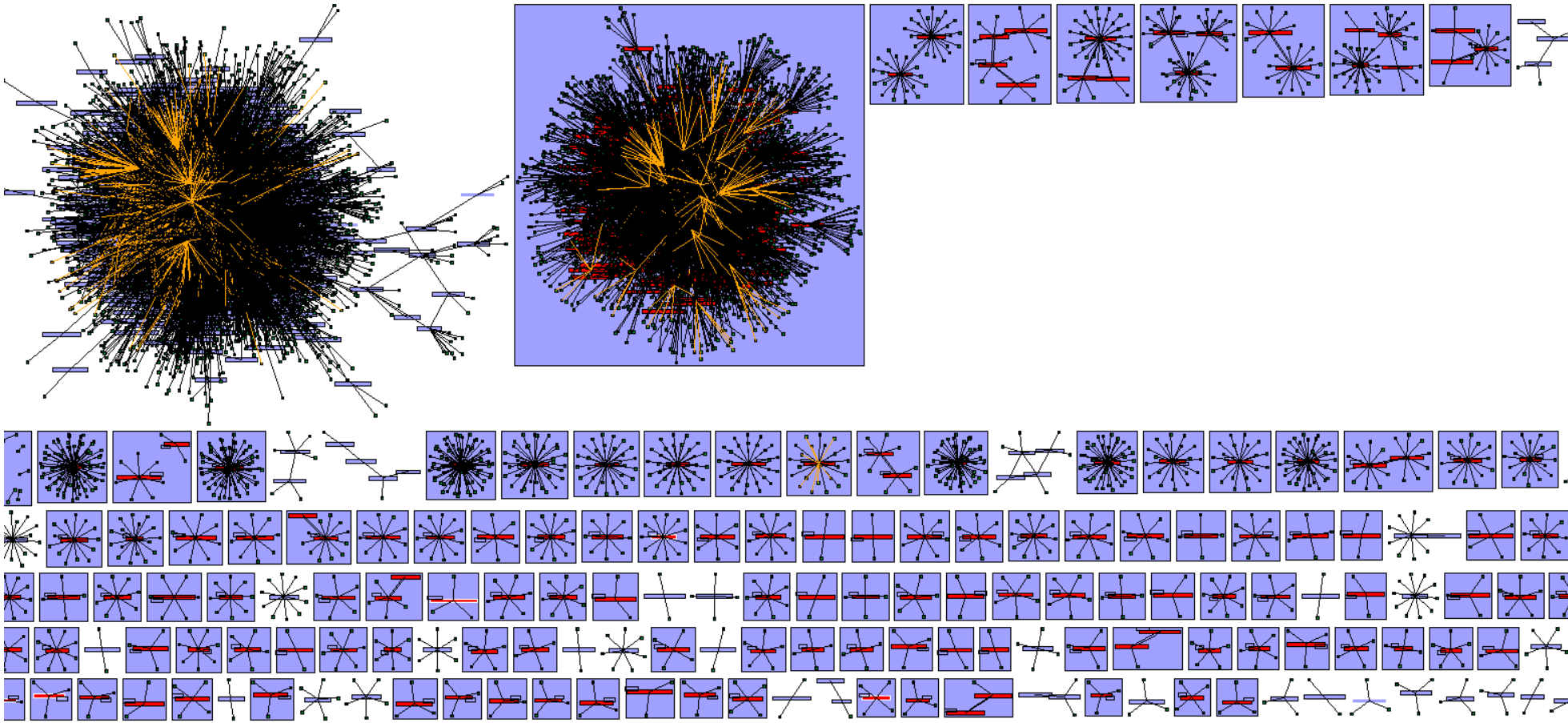
Whole-Program CFG of 300.twolf (20.5KLOC)



#functions: 194 #pointers: 20773 #loads/stores: 8657

Costly to reason about flow of values on CFGs!

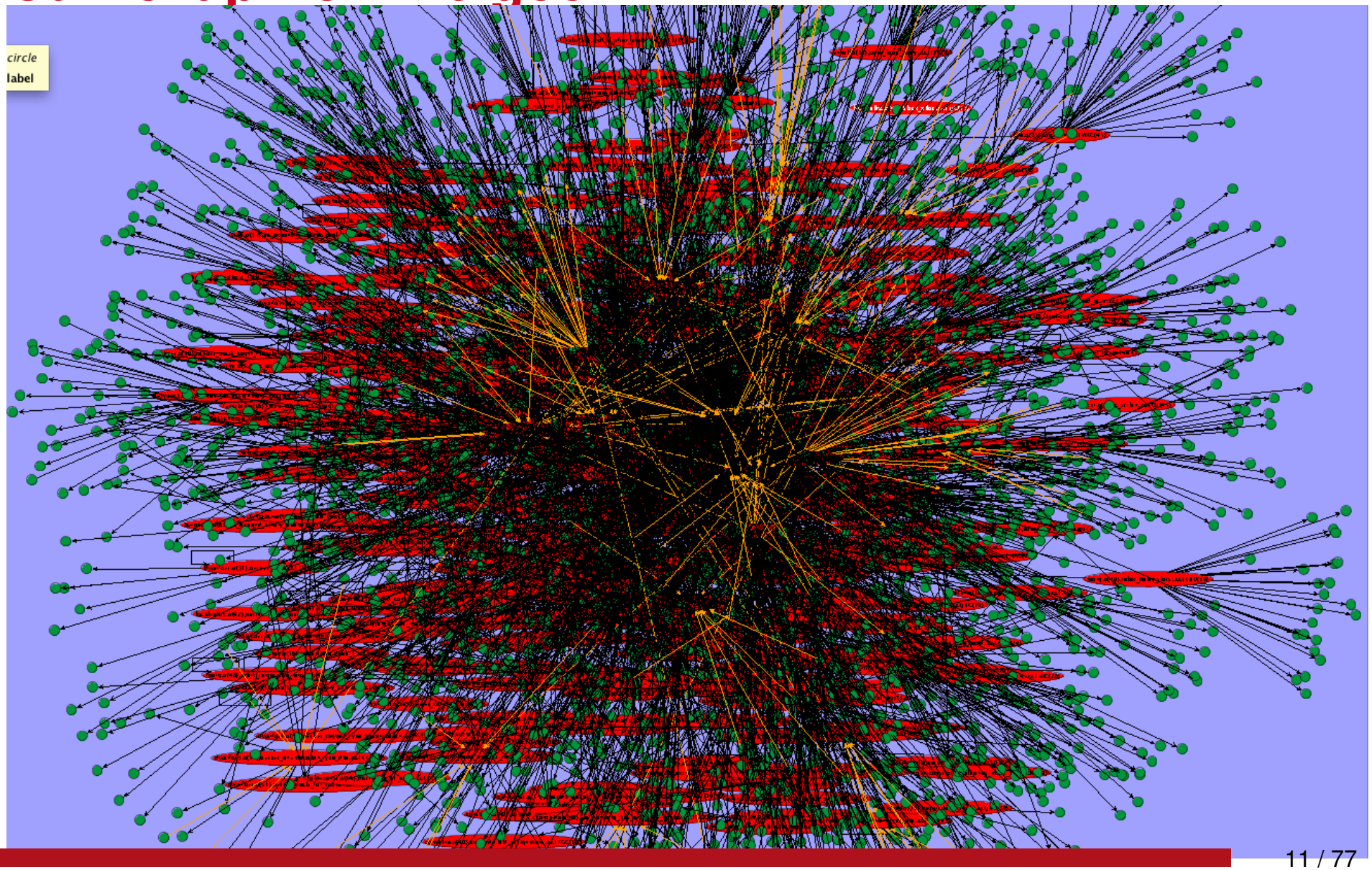
Call Graph of 176.gcc (230.5KLOC)



#functions: 2256 #pointers: 134380 #loads/stores: 51543

Costly to reason about flow of values on CFGs!

Call Graph of 176.gcc



Outline

I have contributed a number of novel and practical solutions:

- **Sparse static value-flow analysis**
  - ISSTA ’12, TSE ’14, SPE ’14, CC ’16, ICSE ’18
- **Selective and on-demand analysis**
  - CGO ’13, ECOOP ’14, SAS ’14, FSE ’16, SAS ’17
- **Static value-flow analysis for detecting memory errors**
  - ICPP ’15, CGO ’16, ACSAC ’17, ICSE ’18
- **Hybrid static and dynamic analysis to enforce memory safety and CFI**
  - ISSRE ’14, CGO ’14, ISSTA ’17, ACISP ’18

Public Available Tools

I am an author of the following public available tools:

- **SVF**: a static tool that enables scalable and precise interprocedural dependence analysis in LLVM.
  - `https://github.com/SVF-tools/SVF`
- **SELFS**: a region-based selective interprocedural flow-sensitive pointer analysis.
  - `https://yuleisui.github.io/selfs`
- **ELF**: a static reflection analysis tool for Java
  - `http://www.cse.unsw.edu.au/~corg/elf/`
- **FSAM**: sparse pointer analysis for multithreaded programs
  - `https://yuleisui.github.io/fsam/`

Outline

- **Existing software bugs and vulnerabilities**
- **Static analysis and dynamic analysis**
- **Technical contributions in developing scalable and precise program analysis**
  - Fundamental program analysis
    - Sparse value-flow analysis
    - Selective and on-demand value-flow analysis
  - Applications
    - Static value-flow analysis for detecting memory errors
    - Hybrid value-flow analysis to enforce memory safety and virtual call integrity
- **Research opportunities**

Value-Flow/Information-Flow Analysis

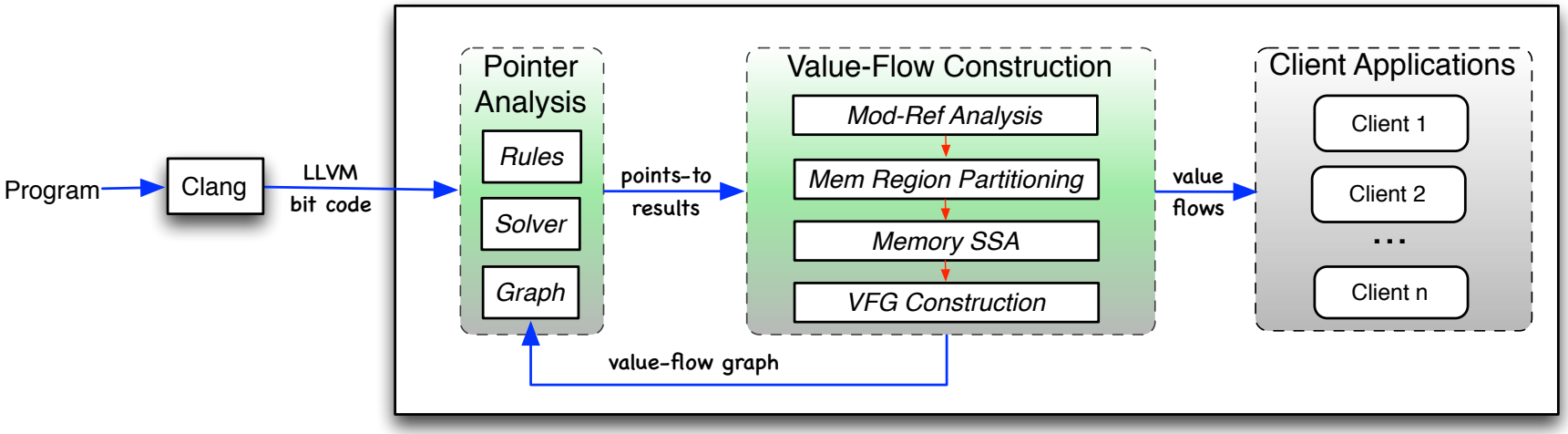
Resolve both control and data dependence of a program.

- Does the information generated at program point A flow to another program point B along some execution paths?
- Can function F be called either directly or indirectly from some other function F'?
- Is there an unsafe memory access that may trigger a bug or security risk?

Key components:

- Pointer analysis (flow-, context-, and path-sensitive data-flow analysis)
- Symbolic execution (whitebox fuzzing)

SVF: Static Value-Flow Analysis



- Publicly available at: https://github.com/SVF-tools/SVF.
- Implemented on top of industrial-strength compiler (LLVM-6.0.0) with over 100KLOC C/C++ code.
- Over 2000 downloads since its first release in July 2015 (133 stars and 50 forked repositories)
- Invited for a plenary talk in EuroLLVM 2016.
- Serves as a foundation for developing other analyses, with participants and contributors from both industry and academia, including IBM, Intel, Google, Huawei, Qualcomm, Veracode and Pathscale.

Flow-Insensitive v.s. Flow-Sensitive Analysis

Flow-insensitive pointer analysis:

- **Ignore program execution order**
- **A single solution across whole program**

Flow-Insensitive v.s. Flow-Sensitive Analysis

Flow-insensitive pointer analysis:

- **Ignore program execution order**
- **A single solution across whole program**

Flow-sensitive pointer analysis:

- **Respect program control-flow**
- **A separate solution at each program point**

Flow-Insensitive v.s. Flow-Sensitive Analysis

Flow-insensitive pointer analysis:

- **Ignore program execution order**
- **A single solution across whole program**

Flow-sensitive pointer analysis:

- **Respect program control-flow**
- **A separate solution at each program point**

```
p = & a

*p = & b

*p = & c

q = *p
```

Flow-insensitive analysis

Flow-Insensitive v.s. Flow-Sensitive Analysis

Flow-insensitive pointer analysis:

- **Ignore program execution order**
- **A single solution across whole program**

Flow-sensitive pointer analysis:

- **Respect program control-flow**
- **A separate solution at each program point**

p = & a

*p = & b

*p = & c

q = *p

p → a

a → b, c

q → b, c

Flow-insensitive analysis

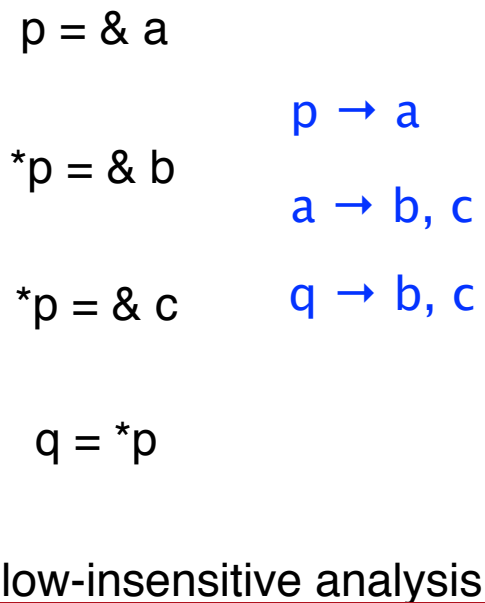
Flow-Insensitive v.s. Flow-Sensitive Analysis

Flow-insensitive pointer analysis:

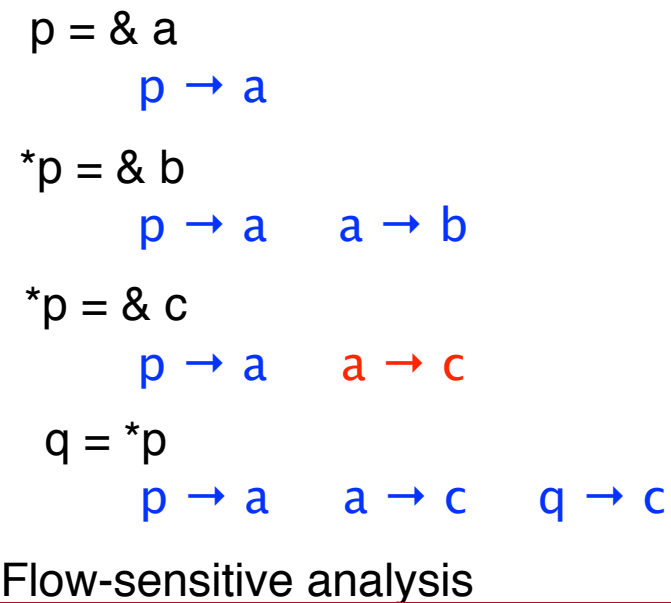
- **Ignore program execution order**
- **A single solution across whole program**

Flow-sensitive pointer analysis:

- **Respect program control-flow**
- **A separate solution at each program point**



Flow-insensitive analysis



Flow-sensitive analysis

Sparse Flow-Sensitive Analysis

- **Propagate points-to information only along pre-computed def-use chains instead of control-flow**

x = & m

$x \rightarrow m$

p = & a

$p \rightarrow a \quad x \rightarrow m$

*p = & b

$p \rightarrow a \quad a \rightarrow b \quad x \rightarrow m$

*p = & c

$p \rightarrow a \quad a \rightarrow c \quad x \rightarrow m$

*x = & d

$p \rightarrow a \quad a \rightarrow c \quad x \rightarrow m \quad m \rightarrow d$

y = *x

$p \rightarrow a \quad a \rightarrow c \quad x \rightarrow m \quad m \rightarrow d \quad y \rightarrow d$

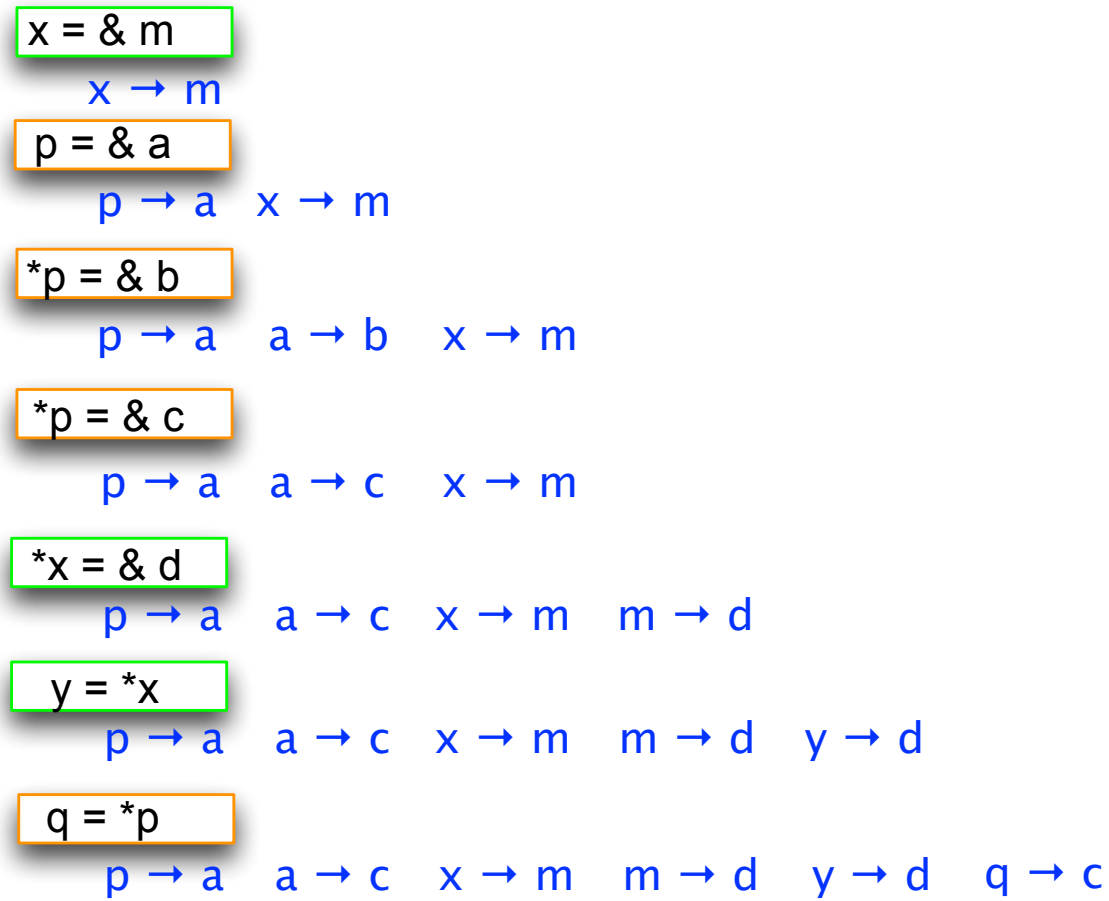
q = *p

$p \rightarrow a \quad a \rightarrow c \quad x \rightarrow m \quad m \rightarrow d \quad y \rightarrow d \quad q \rightarrow c$

Data-flow-based flow-sensitive analysis

Sparse Flow-Sensitive Analysis

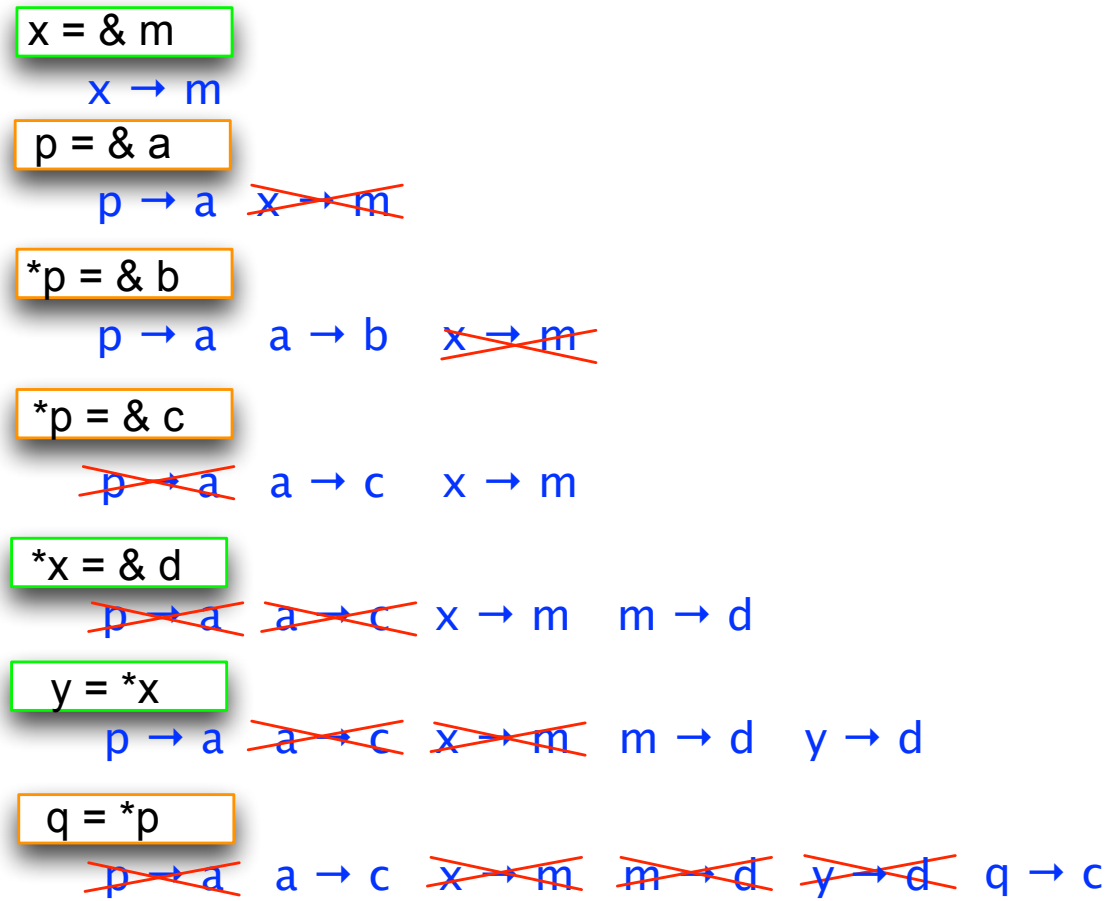
- **Propagate points-to information only along pre-computed def-use chains instead of control-flow**



Data-flow-based flow-sensitive analysis

Sparse Flow-Sensitive Analysis

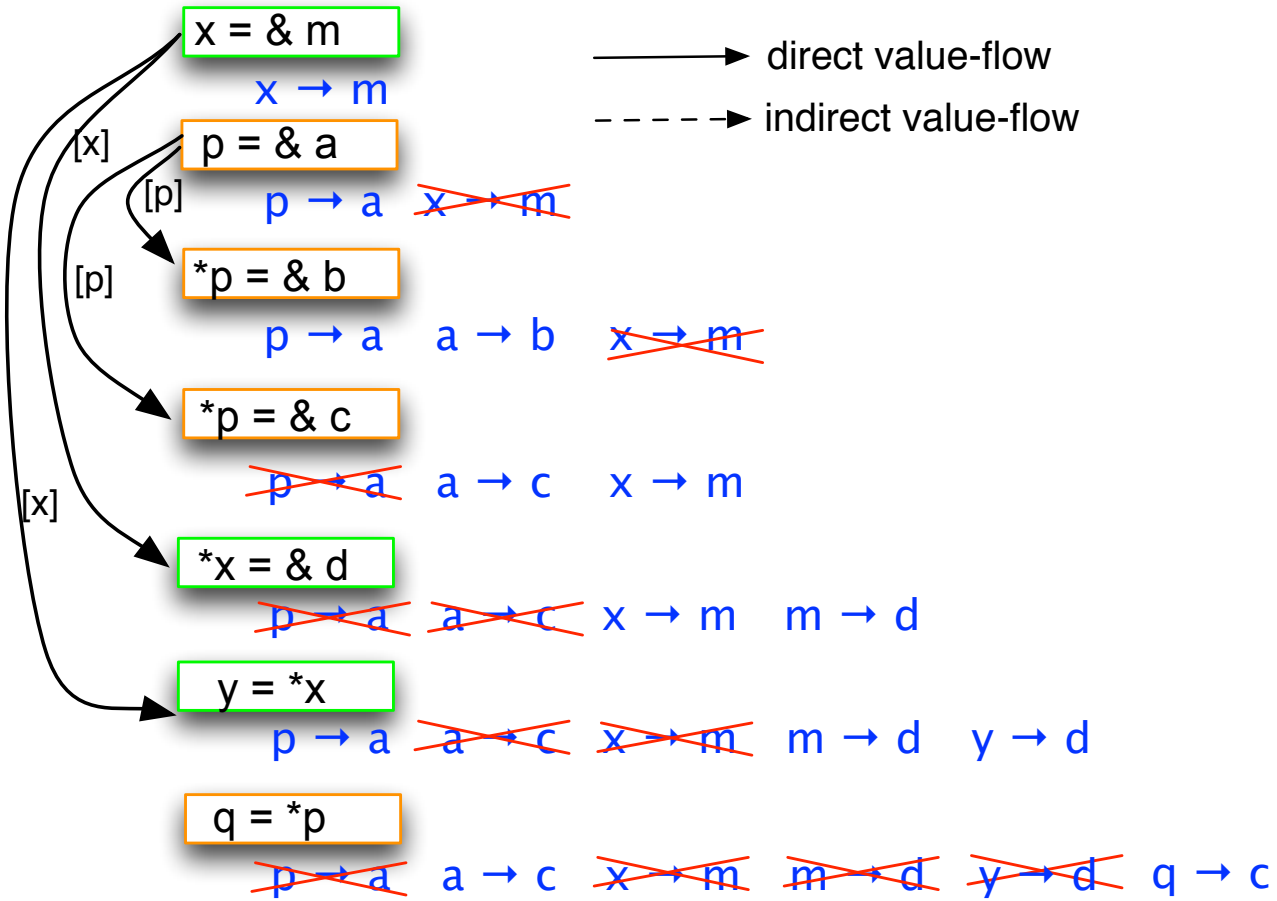
- **Propagate points-to information only along pre-computed def-use chains instead of control-flow**



Data-flow-based flow-sensitive analysis

Sparse Flow-Sensitive Analysis

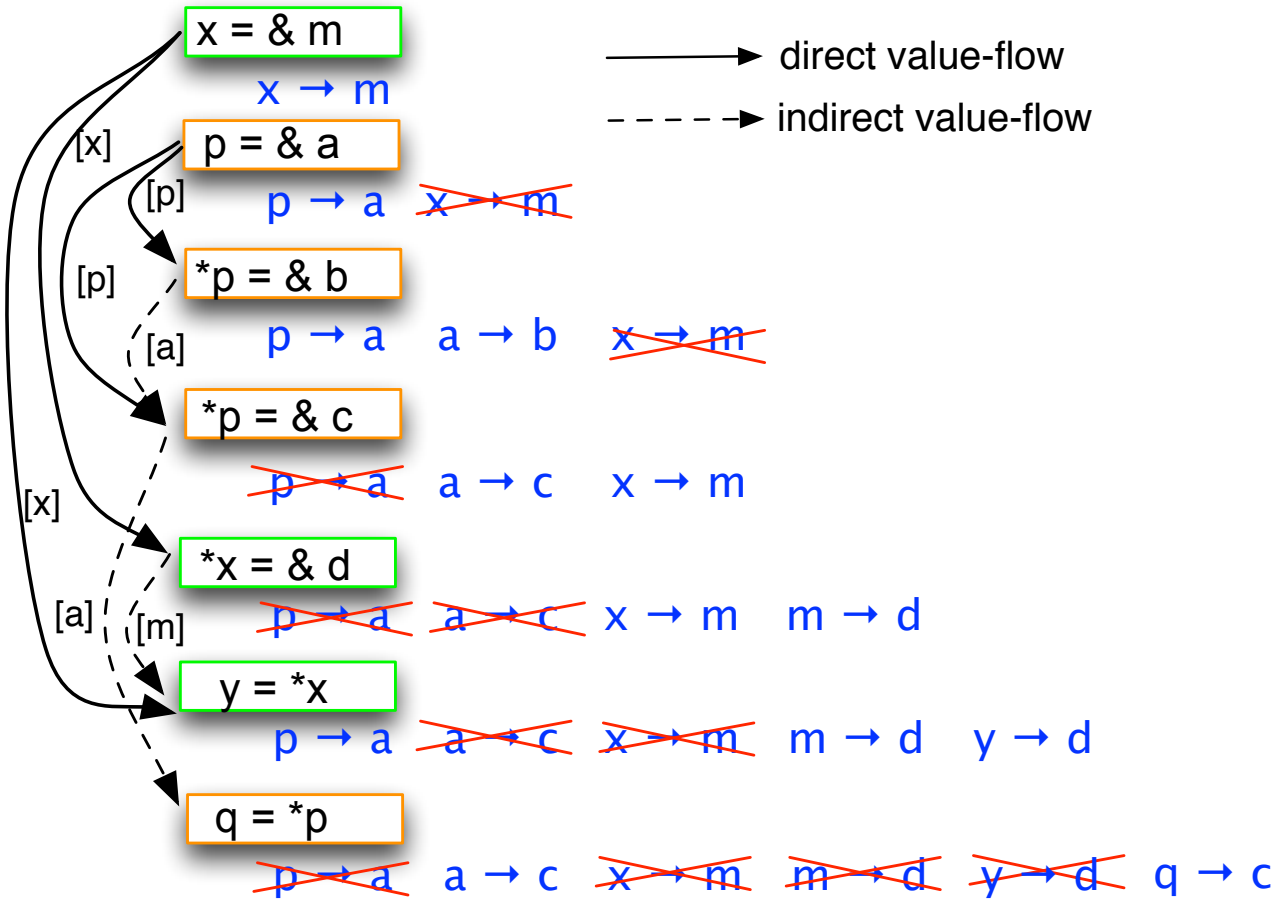
- **Propagate points-to information only along pre-computed def-use chains instead of control-flow**



Sparse analysis (*ISSTA '12*, *SAS '14*, *TSE '14*, *CC '16*)

Sparse Flow-Sensitive Analysis

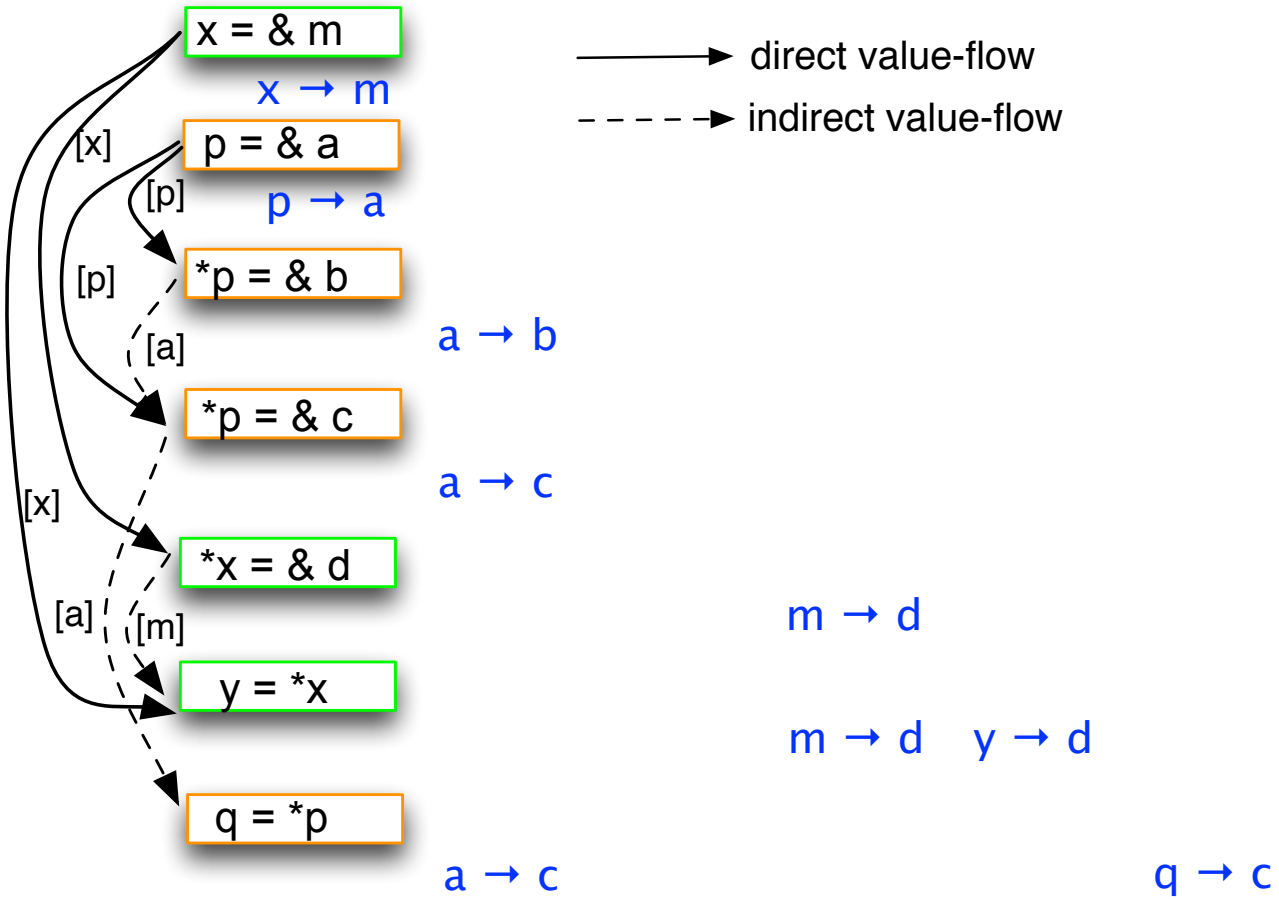
- **Propagate points-to information only along pre-computed def-use chains instead of control-flow**



Sparse analysis (*ISSTA '12*, *SAS '14*, *TSE '14*, *CC '16*)

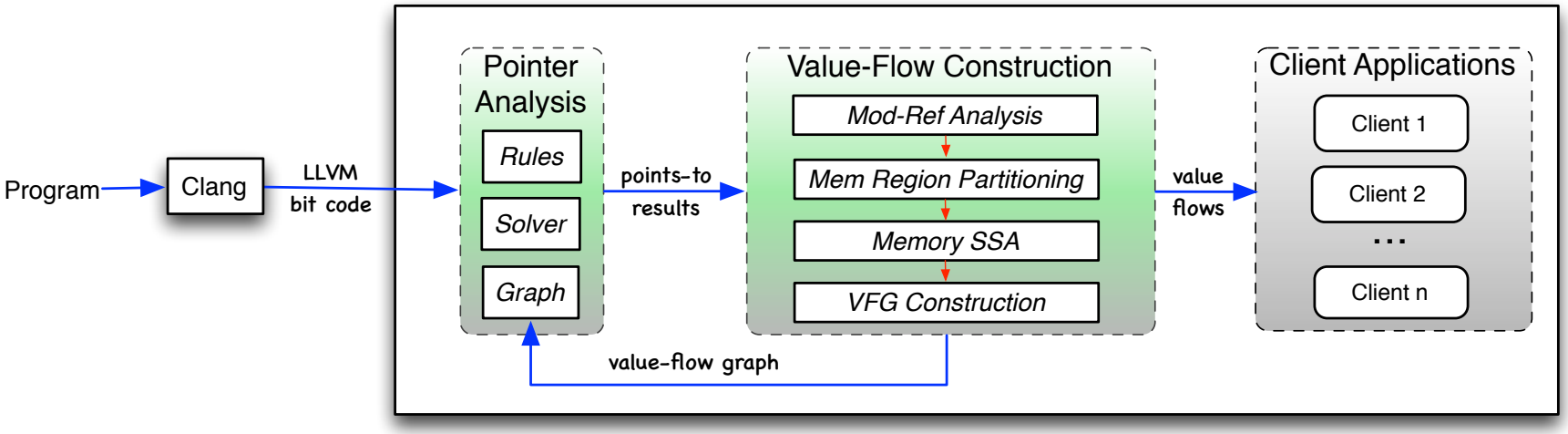
Sparse Flow-Sensitive Analysis

- **Propagate points-to information only along pre-computed def-use chains instead of control-flow**



Sparse analysis (*ISSTA '12*, *SAS '14*, *TSE '14*, *CC '16*)

SVF: Static Value-Flow Analysis



- Publicly available at: https://github.com/SVF-tools/SVF.
- Implemented on top of industrial-strength compiler (LLVM-6.0.0) with over 100KLOC C/C++ code.
- Over 2000 downloads since its first release in July 2015 (133 stars and 50 forked repositories)
- Invited for a plenary talk in EuroLLVM 2016.
- Serves as a foundation for developing other analyses, with participants and contributors from both industry and academia, including IBM, Intel, Google, Huawei, Qualcomm, Veracode and Pathscale.

SVF – Static Value-Flow Analysis

What is SVF?

SVF is a static tool that enables scalable and precise interprocedural dependence analysis for C and C++ programs. SVF allows value-flow construction and pointer analysis to be performed iteratively, thereby providing increasingly improved precision for both.

What kind of analyses does SVF provide?

- Call graph construction for C and C++ programs
- Field-sensitive Andersen's pointer analysis
- Sparse flow-sensitive pointer analysis
- Value-flow dependence analysis
- Interprocedural memory SSA
- Detecting source-sink related bugs, such as memory leaks and incorrect file-open close errors.
- An Eclipse plugin for examining bugs

License

GPLv3

SVF – Static Value-Flow Analysis

User Download History

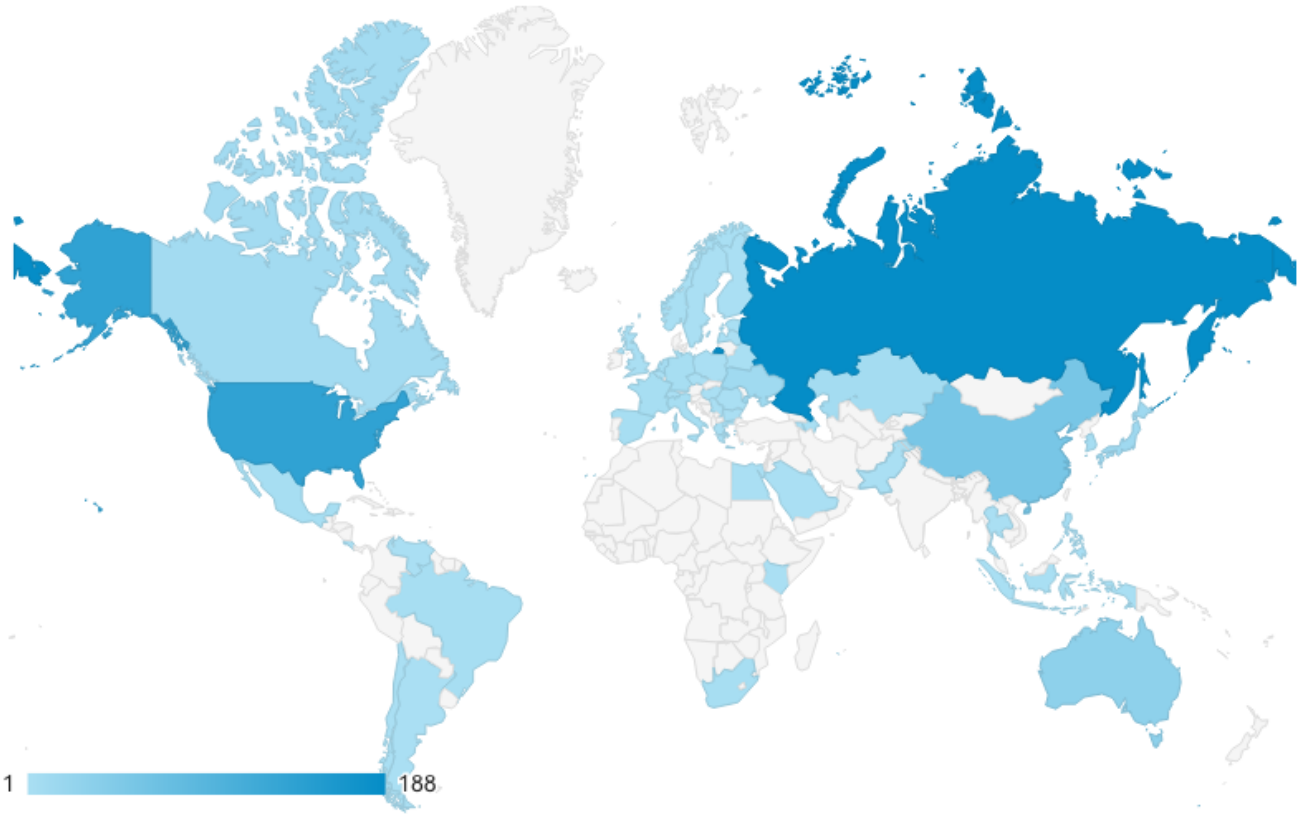
Primary Dimension: Country City Continent Sub Continent

Plot Rows | Secondary dimension | Sort Type: Default | advanced

Country	Acquisition			Behaviour			Conversions		
	Sessions	% New Sessions	New Users	Bounce Rate	Pages/Session	Avg. Session Duration	Goal Conversion Rate	Goal Completions	Goal Value
	4,008 % of Total: 100.00% (4,008)	61.20% Avg for View: 61.20% (0.00%)	2,453 % of Total: 100.00% (2,453)	75.95% Avg for View: 75.95% (0.00%)	1.48 Avg for View: 1.48 (0.00%)	00:01:34 Avg for View: 00:01:34 (0.00%)	0.00% Avg for View: 0.00% (0.00%)	0 % of Total: 0.00% (0)	US$0.00 % of Total: 0.00% (US$0.00)
1. Australia	1,262 (31.49%)	28.76%	363 (14.80%)	75.36%	1.63	00:01:40	0.00%	0 (0.00%)	US$0.00 (0.00%
2. United States	628 (15.67%)	89.01%	559 (22.79%)	78.50%	1.46	00:00:47	0.00%	0 (0.00%)	US$0.00 (0.00%
3. Russia	376 (9.38%)	2.93%	11 (0.45%)	44.41%	1.81	00:07:24	0.00%	0 (0.00%)	US$0.00 (0.00%
4. China	352 (8.78%)	74.43%	262 (10.68%)	84.38%	1.24	00:01:04	0.00%	0 (0.00%)	US$0.00 (0.00%
5. (not set)	335 (8.36%)	100.00%	335 (13.66%)	88.06%	1.24	00:00:00	0.00%	0 (0.00%)	US$0.00 (0.00%
6. United Kingdom	318 (7.93%)	99.37%	316 (12.88%)	87.42%	1.25	00:00:01	0.00%	0 (0.00%)	US$0.00 (0.00%
7. Germany	95 (2.37%)	76.84%	73 (2.98%)	48.42%	1.94	00:00:05	0.00%	0 (0.00%)	US$0.00 (0.00%
8. Spain	61 (1.52%)	47.54%	29 (1.18%)	73.77%	1.64	00:01:11	0.00%	0 (0.00%)	US$0.00 (0.00%
9. Hong Kong	54 (1.35%)	68.52%	37 (1.51%)	74.07%	1.41	00:00:46	0.00%	0 (0.00%)	US$0.00 (0.00%
10. Austria	51 (1.27%)	98.04%	50 (2.04%)	9.80%	2.71	00:00:08	0.00%	0 (0.00%)	US$0.00 (0.00%
11. Japan	50 (1.25%)	100.00%	50 (2.04%)	96.00%	1.04	00:00:07	0.00%	0 (0.00%)	US$0.00 (0.00%
12. Brazil	39 (0.97%)	97.44%	38 (1.55%)	97.44%	1.05	00:00:01	0.00%	0 (0.00%)	US$0.00 (0.00%
13. Netherlands	39 (0.97%)	100.00%	39 (1.59%)	76.92%	1.41	00:00:20	0.00%	0 (0.00%)	US$0.00 (0.00%
14. Singapore	37 (0.92%)	78.38%	29 (1.18%)	78.38%	1.27	00:00:34	0.00%	0 (0.00%)	US$0.00 (0.00%
15. Canada	32 (0.80%)	81.25%	26 (1.06%)	90.62%	1.09	00:00:09	0.00%	0 (0.00%)	US$0.00 (0.00%
16. India	31 (0.77%)	83.87%	26 (1.06%)	100.00%	1.00	00:00:00	0.00%	0 (0.00%)	US$0.00 (0.00%
17. France	25 (0.62%)	92.00%	23 (0.94%)	100.00%	1.00	00:00:00	0.00%	0 (0.00%)	US$0.00 (0.00%
18. South Korea	24 (0.60%)	91.67%	22 (0.90%)	87.50%	1.12	00:01:17	0.00%	0 (0.00%)	US$0.00 (0.00%
19. Iraq	19 (0.47%)	100.00%	19 (0.77%)	100.00%	1.00	00:00:00	0.00%	0 (0.00%)	US$0.00 (0.00%
20. Iran	11 (0.27%)	100.00%	11 (0.45%)	81.82%	1.18	00:00:52	0.00%	0 (0.00%)	US$0.00 (0.00%
21. Taiwan	10 (0.25%)	100.00%	10 (0.41%)	90.00%	1.30	00:01:30	0.00%	0 (0.00%)	US$0.00 (0.00%
22. Italy	9 (0.22%)	100.00%	9 (0.37%)	100.00%	1.00	00:00:00	0.00%	0 (0.00%)	US$0.00 (0.00%

SVF – Static Value-Flow Analysis

User Download History



Comments on SVF

- *"Looking for llvm pointer analysis I stumbled over SVF and was quite impressed by it's capabilities - thanks a lot for making it open source!"*
  — Matthias Neugschwandtner ⟨*EUG@zurich.ibm.com*⟩, a postdoctoral researcher working in the Cloud and Storage Security Group at IBM Research.
- *"Your tests have been most helpful! . . . I'm trying to convince Arthur that SVF can solve some (all?) of the design requirement laid out by Chandler (Google)."*
  — C Bergstrom ⟨*cbergstrom@pathscale.com*⟩ CTO at Pathscale, HPC Compiler Company.
- *"First, I'd like to thank you for your work on SVF and for making it public! The academic LLVM community is in dire need of such a framework and I am very interested in seeing your framework become the foundation for many analyses moving forward. "*
  — Will Dietz ⟨*wdietz2@illinois.edu*⟩, Researcher at UIUC and ALLVM.
- *"I came across SVF recently and am really interested in using it. . . . it's a really nice set of tools! Again thanks! "*
  — Jared Carlson ⟨*jared.carlson23@gmail.com*⟩ Sr. Security Researcher at Veracode, a leading company working in software application security.
- *" I remember watching your presentation on SVF and thought: that's cool, was so excited to be able to reach you. . . . Would be great to be able to chat about how maybe you and our team might be able to work on interesting things together"*
  – Dean Michael Berris ⟨*dean.berris@gmail.com*⟩, Software Engineer at Google.
- *'I hope to use SVF for the DARPA TRACE grant application.'*
  — Stephen N. Lee from University of Wisconsin-Madison

Comments on SVF

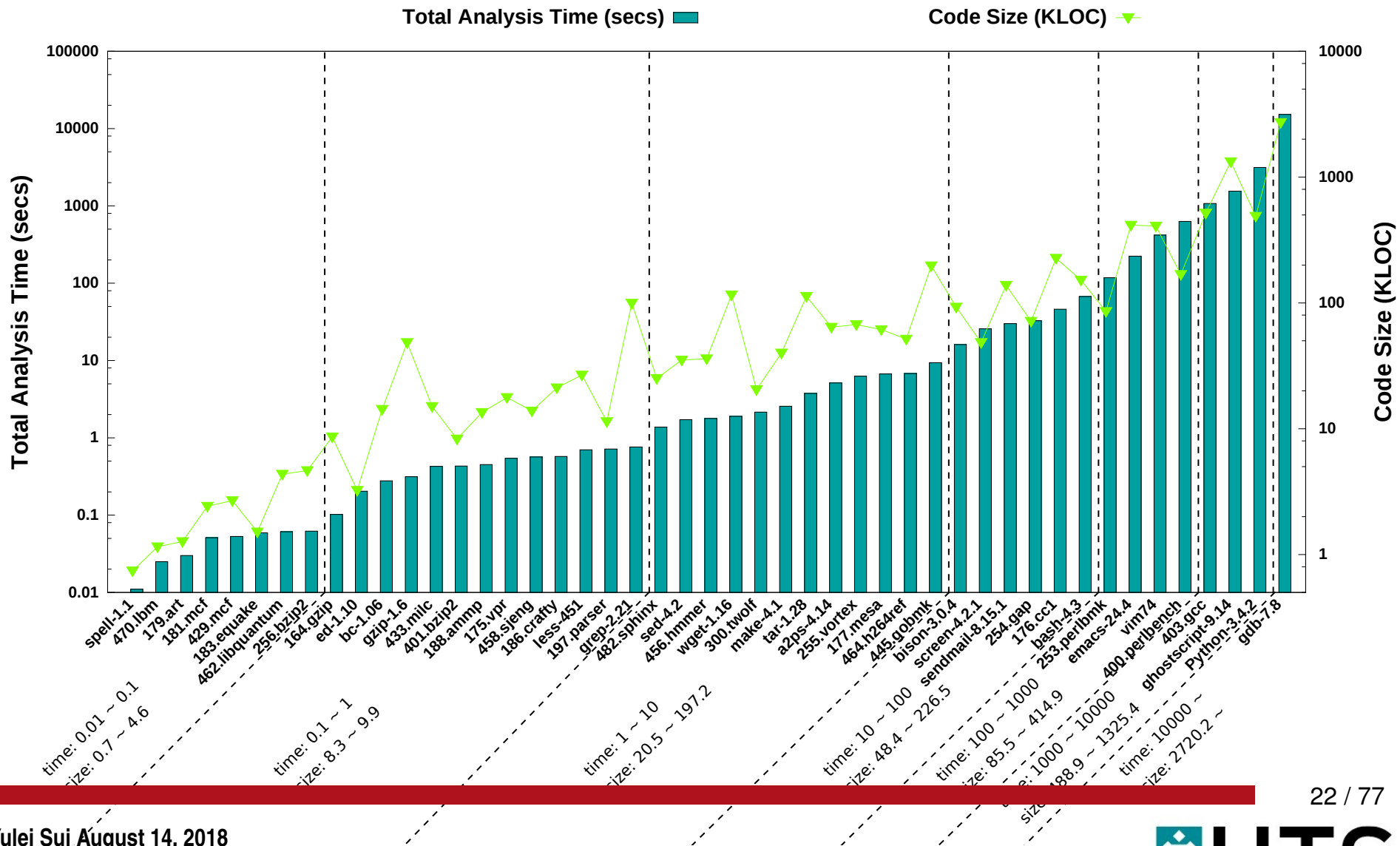
- *"I'm currently working with your awesome SVF analysis."*
  — Simon Schmitt ⟨*symenschmitt@web.de*⟩ from TU Darmstadt.
- *"I've run into your paper: Region-based Selective Flow-Sensitive Pointer Analysis. It's a great work!"*
  — Marek Chalupa ⟨*mchalupa@mail.muni.cz*⟩.
- *"Thank you again and I think that the SVF will be promising for my work."*
  — Muhammad Refaat Soliman ⟨*mrefaat@uwaterloo.ca*⟩.
- *" We very much appreciate your work and would like to use the static pointer analysis you developed."*
  — Alexandra Jimborean ⟨*alexandra.jimborean@it.uu.se*⟩.
- *" I'm very interested in the SVF analysis that can be performed with your software."*
  — Timo Bressmer ⟨*timo.bressmer@uni − ulm.de*⟩.
- *"We heard a lot about SABER: Static Memory Leak Detection Using Full-Sparse Value-Flow Analysis. We would like to test our improvement ideas for SABER "*
  — Ahmed Tamrawi ⟨*atamrawi@iastate.edu*⟩
- *"I recently tested SVF (commit 5355fc2). Great piece of work from my point of view! . . . Thank you for providing the code!"*
  — Oliver Braunsdorf at TU Ilmenau ⟨*oliver.braunsdorf@tu − ilmenau.de*⟩
- *" As far as I can tell, the tool fixes all issues I have with musl libc. Thank you."*
  — Anh Quach ⟨*aquach1@binghamton.edu*⟩
- *"Thanks for your great project. "*
  – Qingkai (Thomas) Shi ⟨*qingkaishi@gmail.com*⟩ at Hong Kong University of Science and Technology.
- *" Thanks for the quick answer:) I am looking at SVF implementation, in order to test one of our ideas. "*
  – David Trabish at Tel-Aviv University
- *"Perfect. Thank You. This is a great project!! "*
  – Machiry Aravind Kumar at UCSB

Evaluation and Results

- Benchmarks:
  - All SPEC C benchmarks: 15 programs from CPU2000 and 12 programs from CPU2006
  - 20 Open-source applications: most of them are recent released versions.
  - Total lines of code evaluated: 8,005,872 LOC with maximum program size 2,720,279 LOC
  - Gold Plugin is used to combine multiple bitcode files into one
- Machine setup:
  - Ubuntu Linux 3.11.0-15-generic Intel Xeon Quad Core HT, 3.7GHZ, 64GB

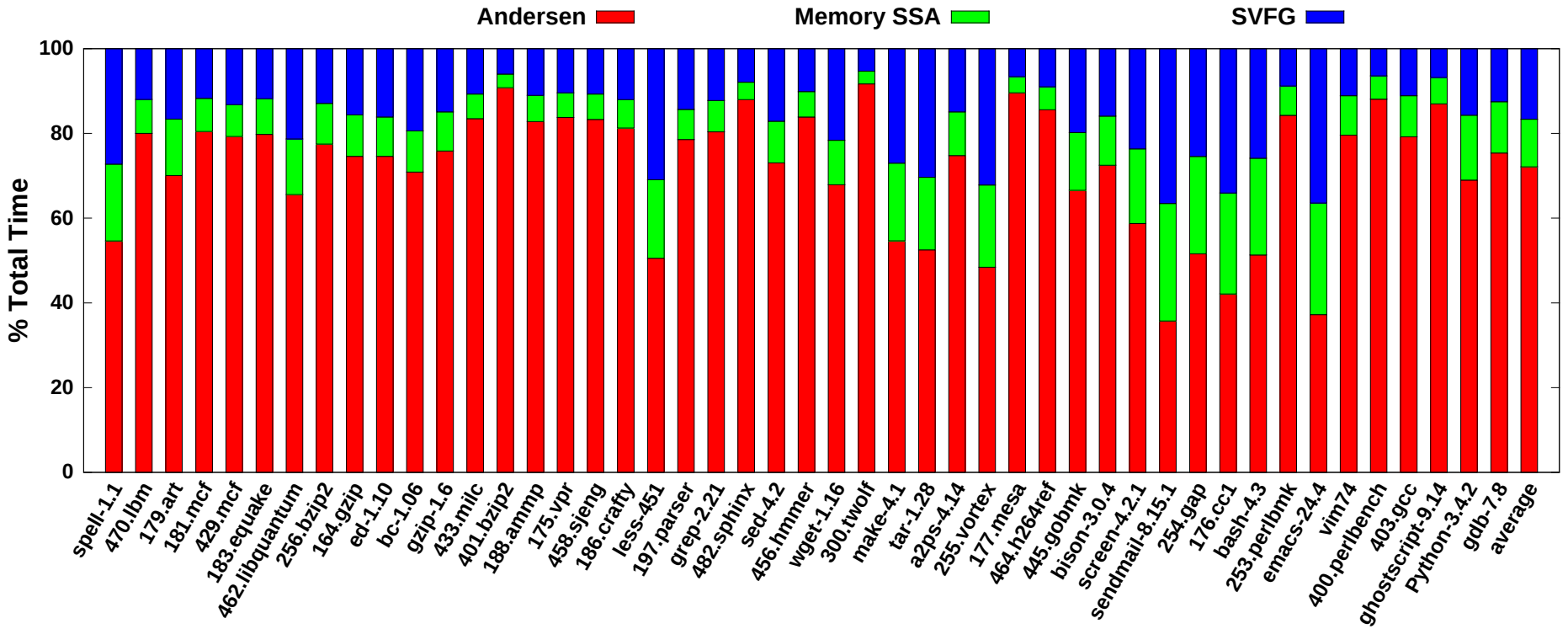
Analysis Time

Total Analysis Time = Andersen + MemorySSA + VFG



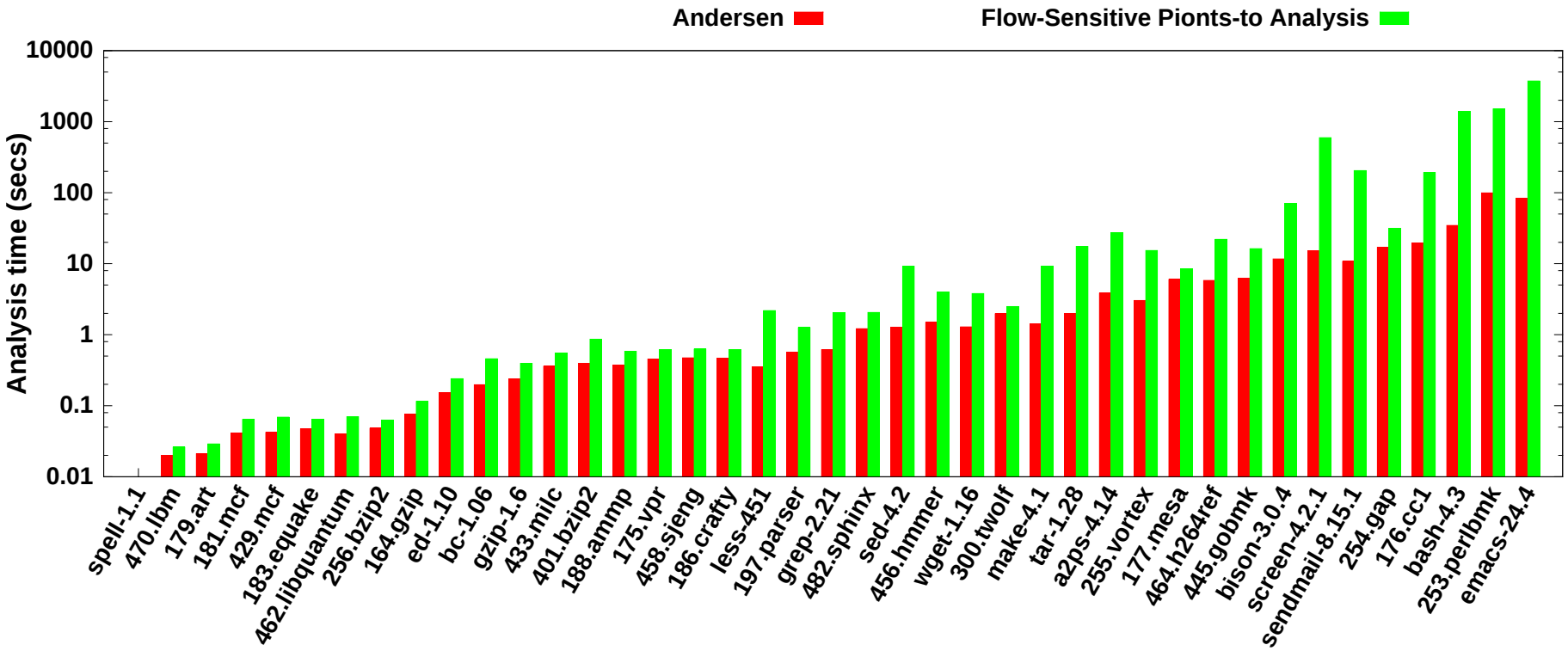
UNIVERSITY OF TECHNOLOGY, SYDNEY

Analysis Time Distribution



Average Percentage: Andersen (71.9%), Memory SSA (11.3%), VFG (16.8%)

Analysis Time : Andersen v.s. Sparse Flow-Sensitive Points-to Analysis



Flow-Sensitive Analysis Slowdowns: From 1.2× to 44×. On average 6.5×.

Outline

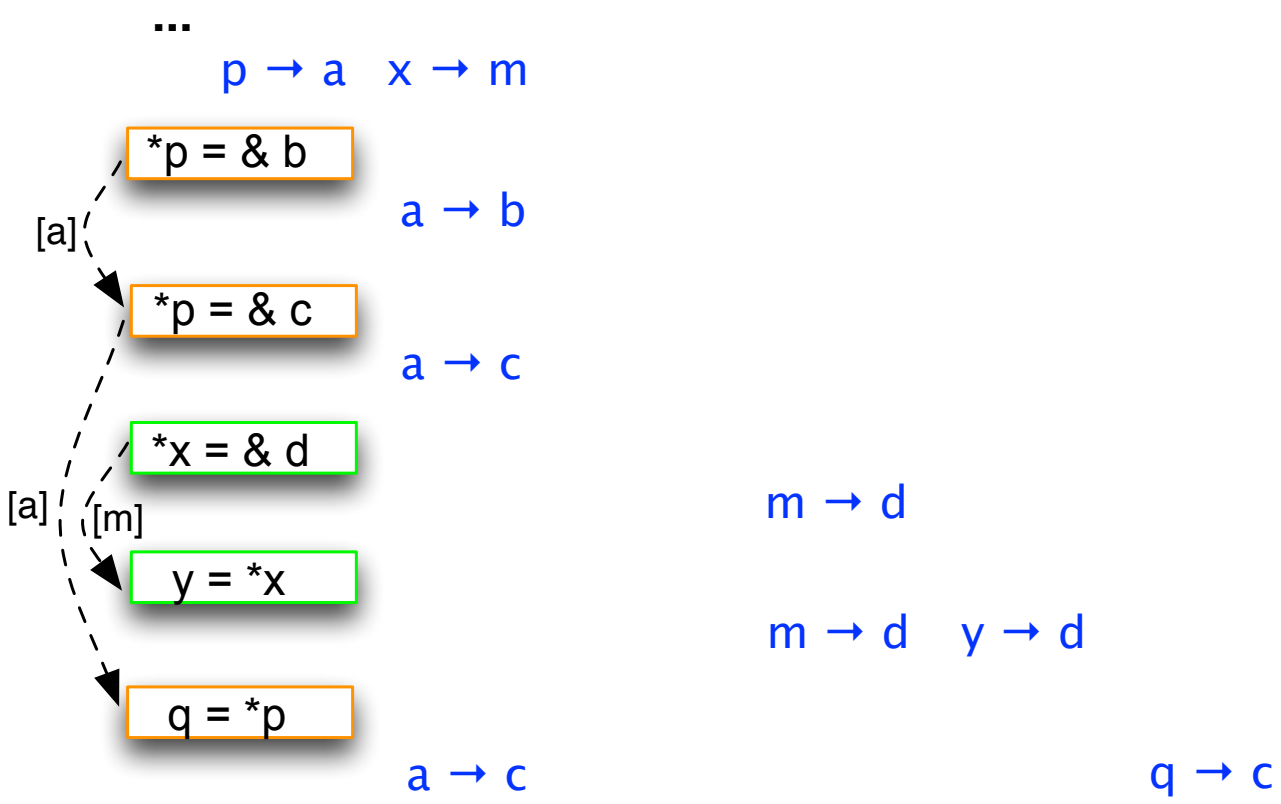
- **Existing software bugs and vulnerabilities**
- **Static analysis and dynamic analysis**
- **Technical contributions in developing scalable and precise program analysis**
  - Fundamental program analysis
    - Sparse value-flow analysis
    - Selective and on-demand value-flow analysis
  - Applications
    - Static value-flow analysis for detecting memory errors
    - Hybrid value-flow analysis to enforce memory safety
- **Research opportunities**

Selective and On-demand Program Analysis

- Precise whole-program analysis is expensive
  - **Selective** analysis for important parts of a program
  - **On-demand** analysis for answering client queries

Region-based Selective Flow-Sensitivity (SAS '14)

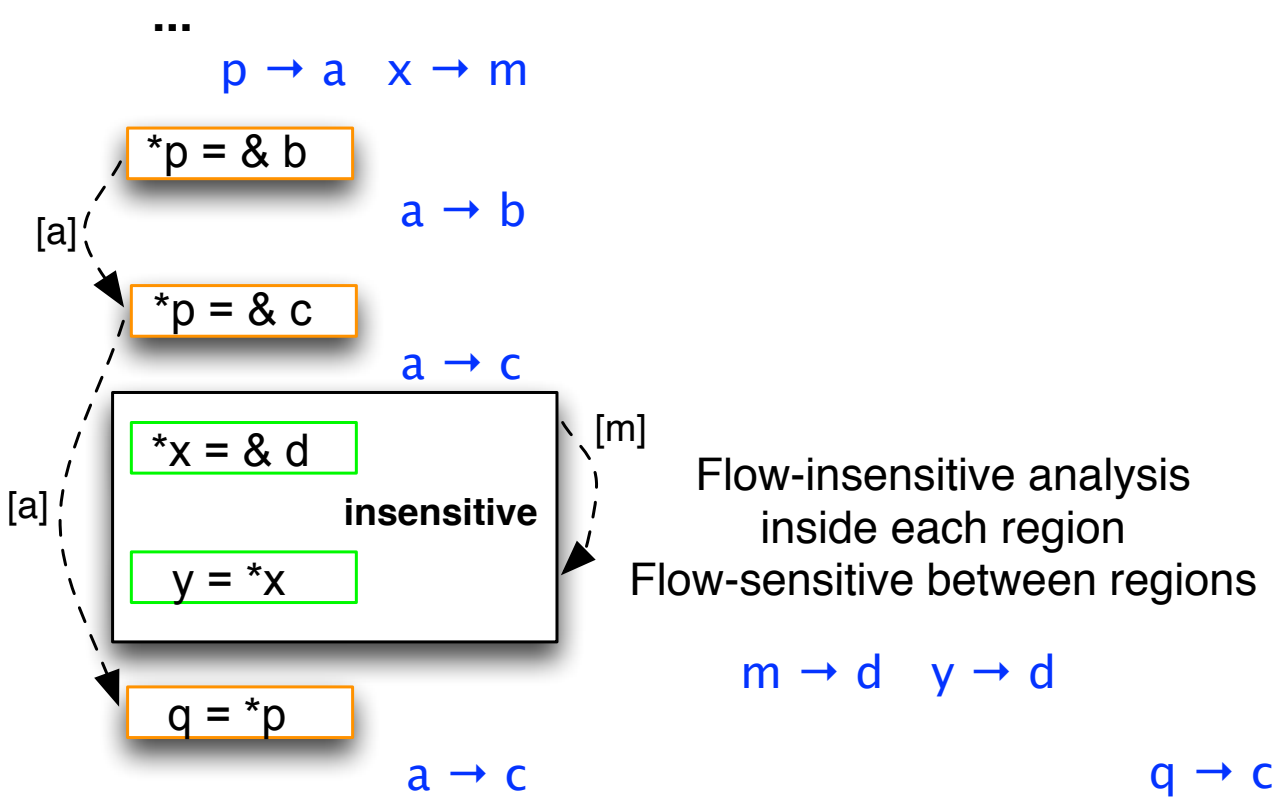
- **Hybrid flow-insensitive and sensitive pointer analysis that operate on the regions partitioned from a program**



Sparse flow-sensitive analysis

Region-based Selective Flow-Sensitivity (SAS '14)

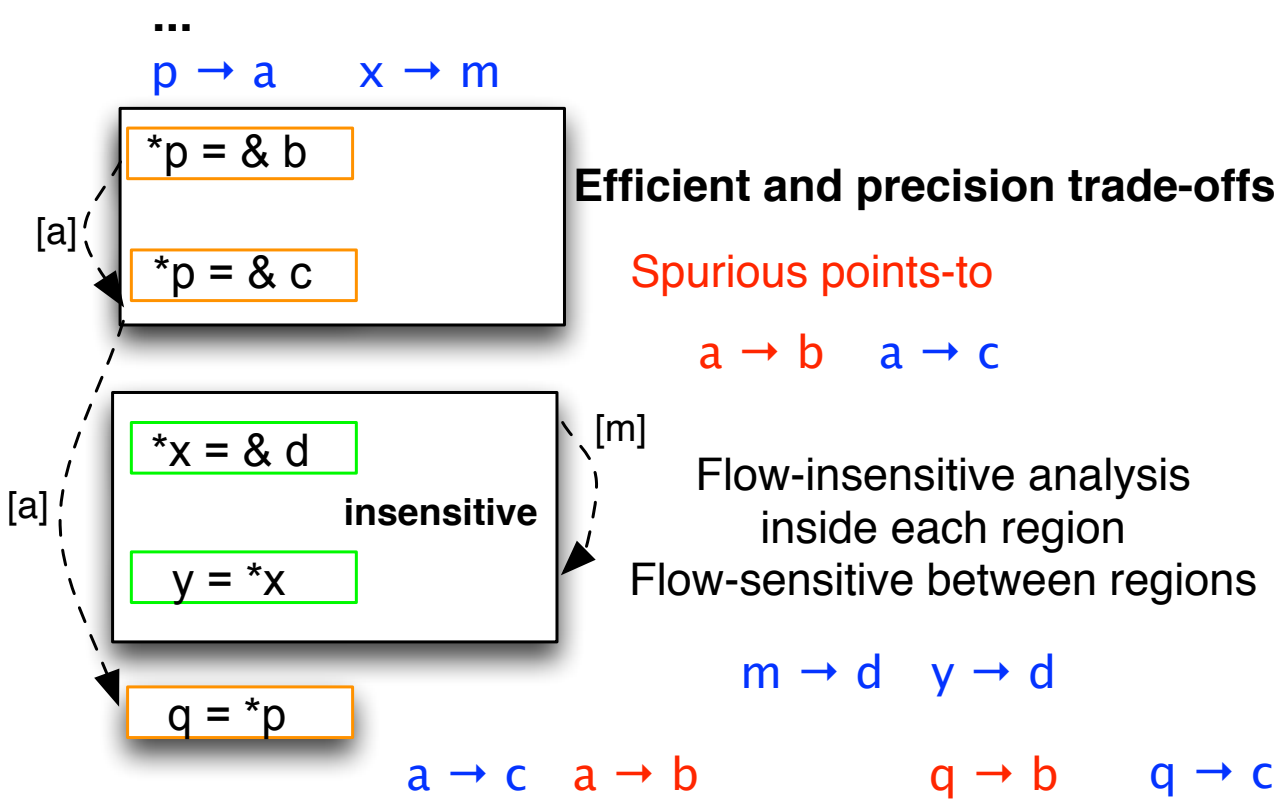
- **Hybrid flow-insensitive and sensitive pointer analysis that operate on the regions partitioned from a program**



Region-based flow-sensitive analysis

Region-based Selective Flow-Sensitivity (SAS '14)

- **Hybrid flow-insensitive and sensitive pointer analysis that operate on the regions partitioned from a program**



Region-based flow-sensitive analysis

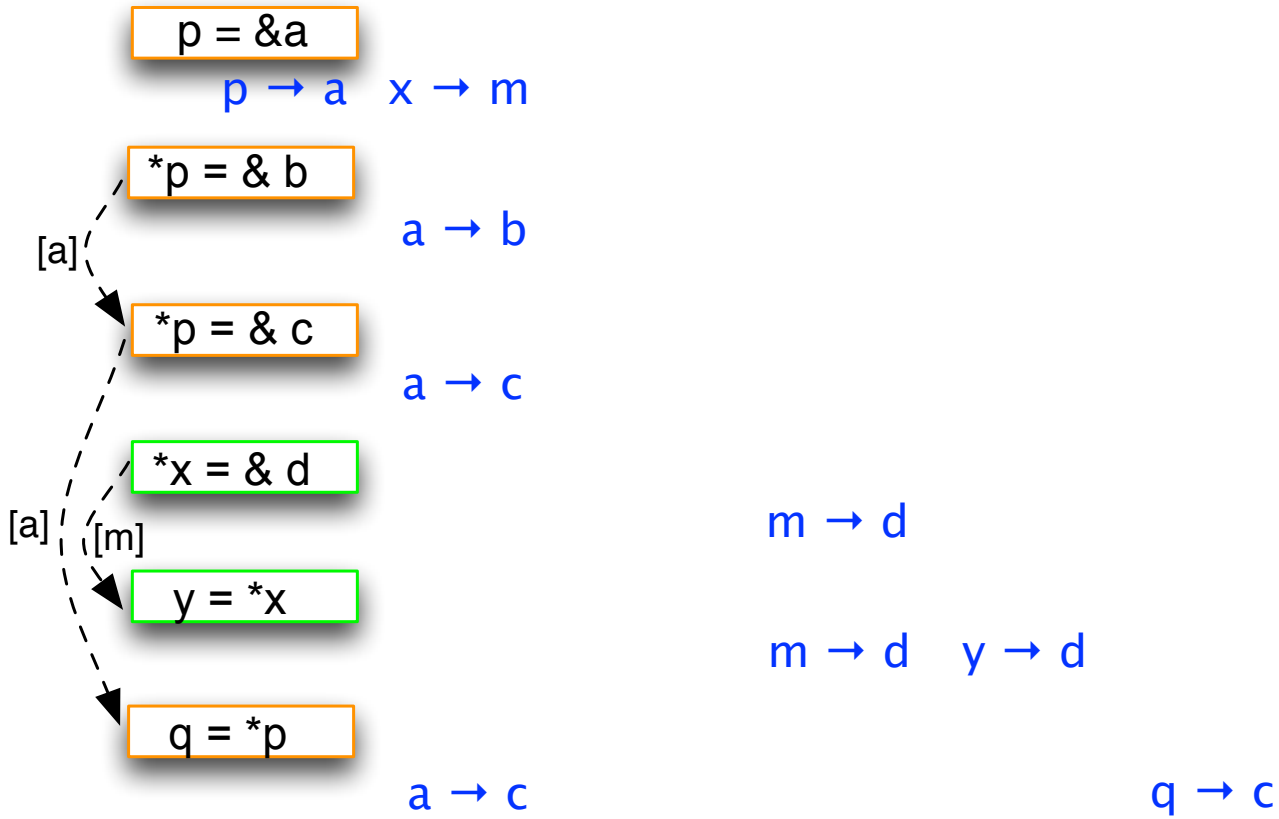
Region-based Selective Flow-Sensitivity (SAS '14)

Program	Size KLOC	Analysis Times (Secs)			Type of Regions (of L, S and W Types)				
		Sparse	Selective	Speedup	#L	#S	#W	#Avg	#Max
ammp	13.4	0.31	0.31	1.00	538	0	187	1.53	16
crafty	21.2	0.30	0.31	0.97	377	0	328	1.95	276
gcc	230.4	826.34	408.93	2.02	10690	294	6867	2.23	401
h264ref	51.6	40.84	5.48	7.45	3523	159	1460	1.94	128
hmmer	36.0	0.42	0.52	0.81	1170	59	487	1.59	56
mesa	61.3	1096.63	180.37	6.08	3801	0	2211	1.40	50
milc	15.0	0.28	0.26	1.08	566	8	253	1.69	66
parser	11.4	3.80	2.31	1.65	820	11	931	1.47	110
perlbmk	87.1	407.86	143.25	2.85	7514	513	4451	1.78	189
sjeng	13.9	0.07	0.19	0.37	463	0	524	1.50	14
sphinx3	25.1	1.16	1.23	0.94	953	14	598	1.98	42
twolf	20.5	1.07	1.02	1.05	1798	1	494	3.51	184
vortex	67.3	86.01	37.92	2.27	2369	198	2061	2.11	830
vpr	17.8	0.30	0.27	1.11	768	0	305	2.36	39

Figure: Comparing sparse flow-sensitive analysis and region-based selective flow-sensitive analysis.

Value-flow-based demand-driven Analysis (FSE '16)

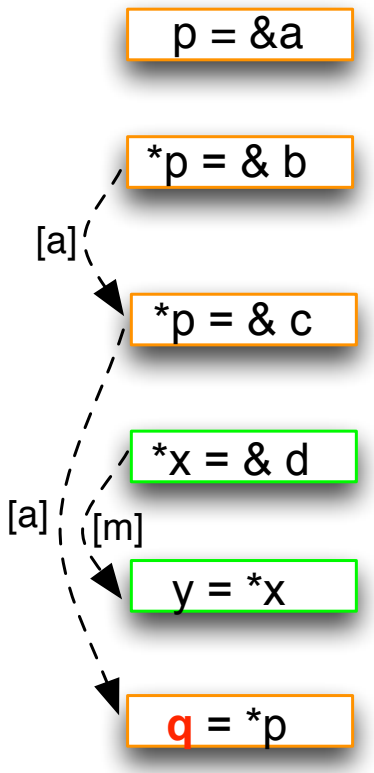
- **Demand-driven flow-sensitive analysis with strong updates via CFL-Reachability**



Sparse flow-sensitive analysis

Value-flow-based demand-driven Analysis (FSE '16)

- **Demand-driven flow-sensitive analysis with strong updates via CFL-Reachability**

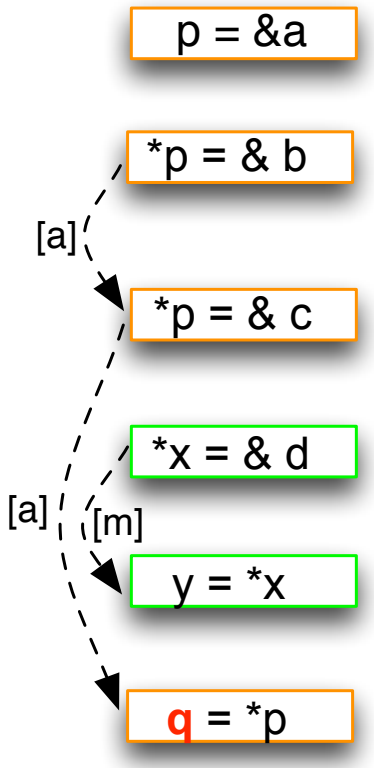


Query: points-to value of q ?

Demand-driven value-flow analysis with user-specified budgets

Value-flow-based demand-driven Analysis (FSE '16)

- **Demand-driven flow-sensitive analysis with strong updates via CFL-Reachability**



$$\textit{flowsTo} \rightarrow \text{alloc}\ (\text{assign}\ |\ \text{store}\ \textit{alias}\ \text{load})^*$$

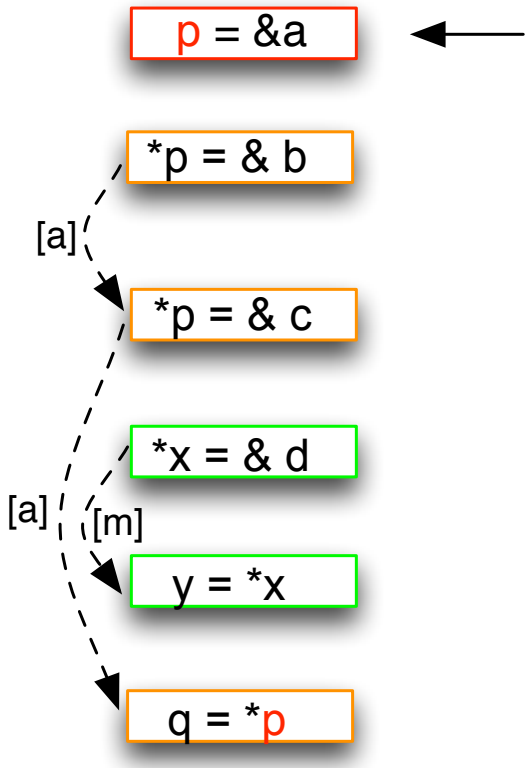
$$\overline{\textit{flowsTo}} \rightarrow (\overline{\text{assign}}\ |\ \text{load}\ \textit{alias}\ \text{store})^*\ \text{alloc}$$

$$\textit{alias} \rightarrow \overline{\textit{flowsTo}}\ \textit{flowsTo}$$

Demand-driven value-flow analysis with user-specified budgets

Value-flow-based demand-driven Analysis (FSE '16)

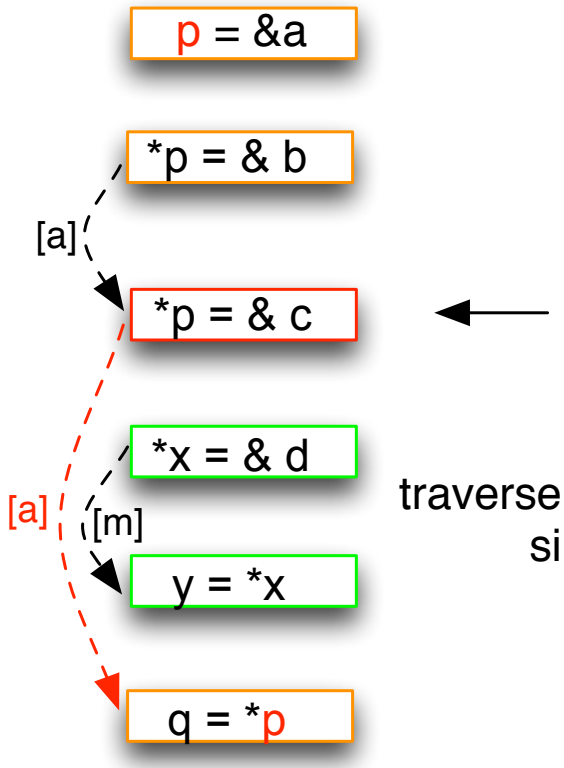
- **Demand-driven flow-sensitive analysis with strong updates via CFL-Reachability**



Demand-driven value-flow analysis with user-specified budgets

Value-flow-based demand-driven Analysis (FSE '16)

- **Demand-driven flow-sensitive analysis with strong updates via CFL-Reachability**

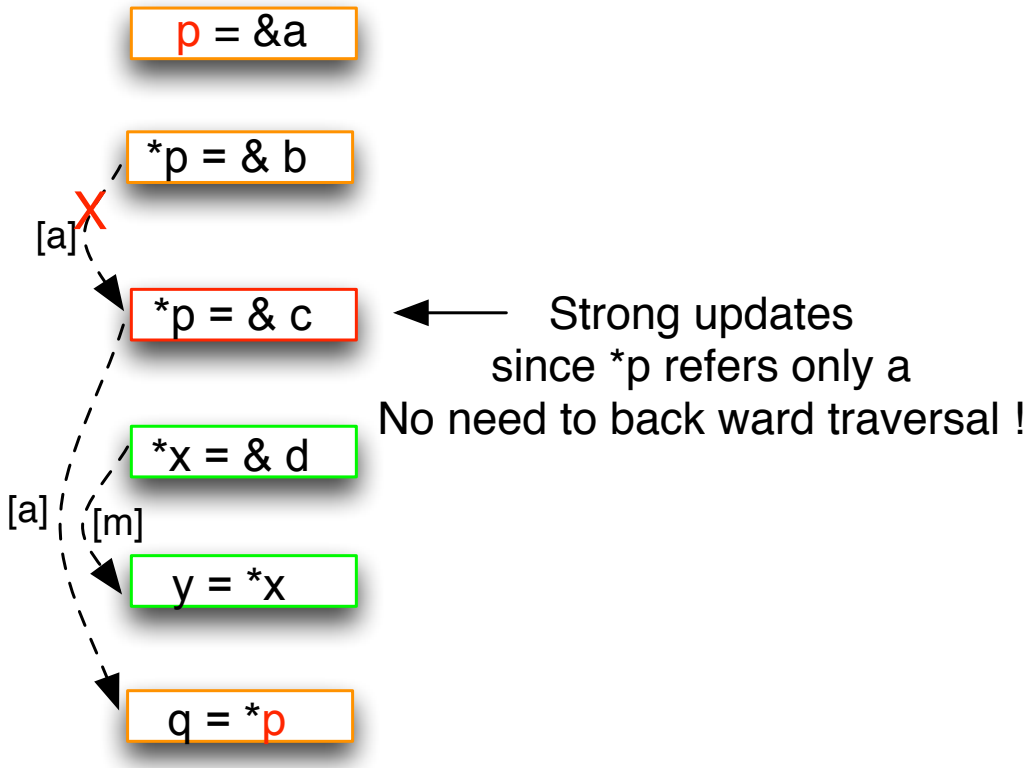


traverse against value-flow of a since p points to a

Demand-driven value-flow analysis with user-specified budgets

Value-flow-based demand-driven Analysis (FSE '16)

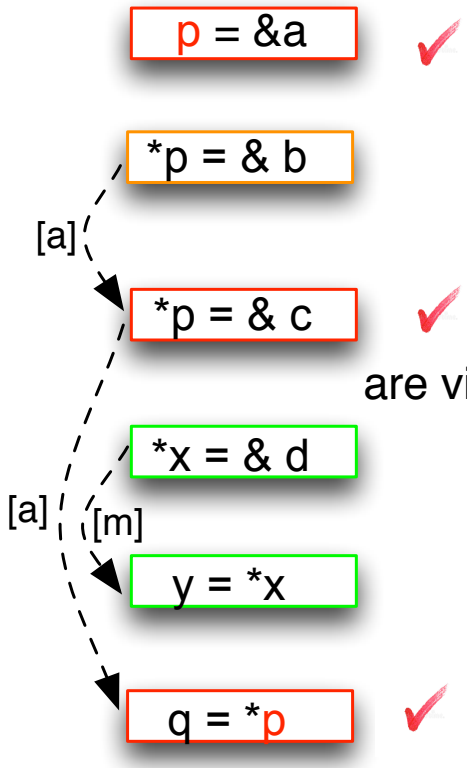
- **Demand-driven flow-sensitive analysis with strong updates via CFL-Reachability**



Demand-driven value-flow analysis with user-specified budgets

Value-flow-based demand-driven Analysis (FSE '16)

- **Demand-driven flow-sensitive analysis with strong updates via CFL-Reachability**



Only three statements
are visited to obtain the points-to result:
q points to c

Demand-driven value-flow analysis with user-specified budgets

Value-flow-based demand-driven Analysis (FSE '16)

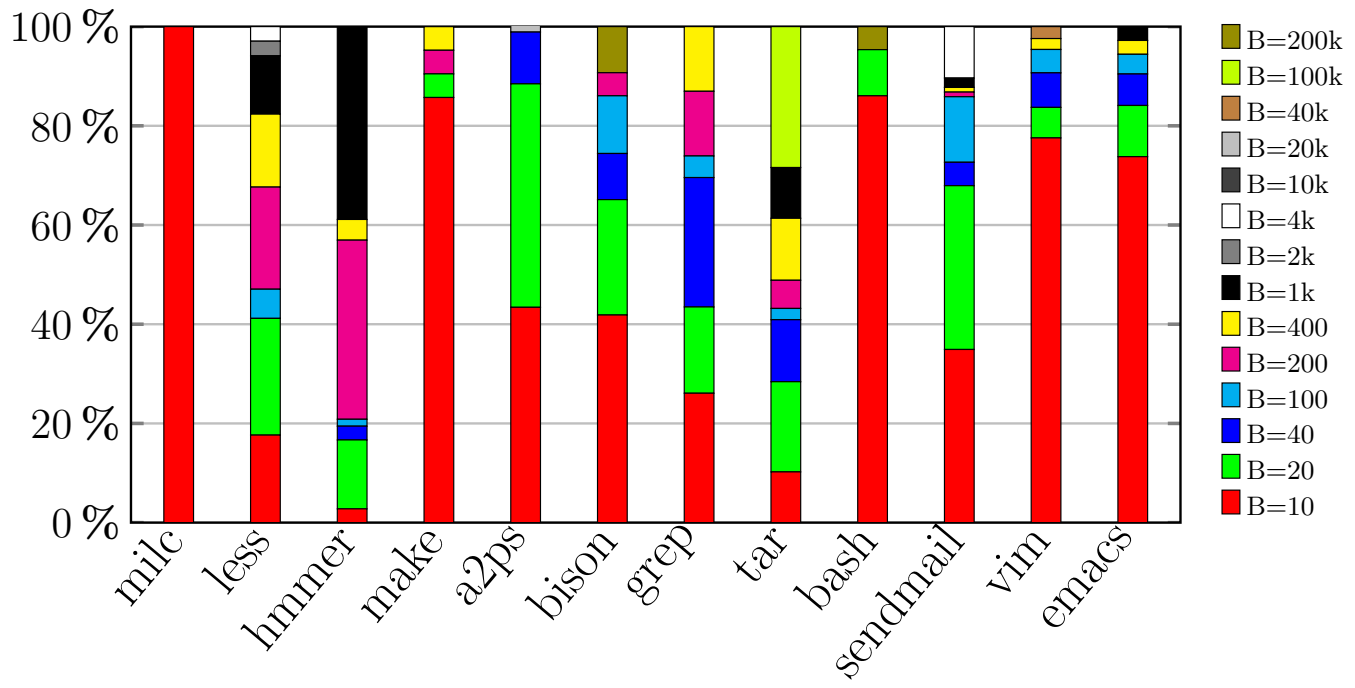


Figure: Percentage of queried variables proved to be safe (initialized) by demand-driven analysis over whole-program analysis under different budgets

Value-flow-based demand-driven Analysis (FSE '16)

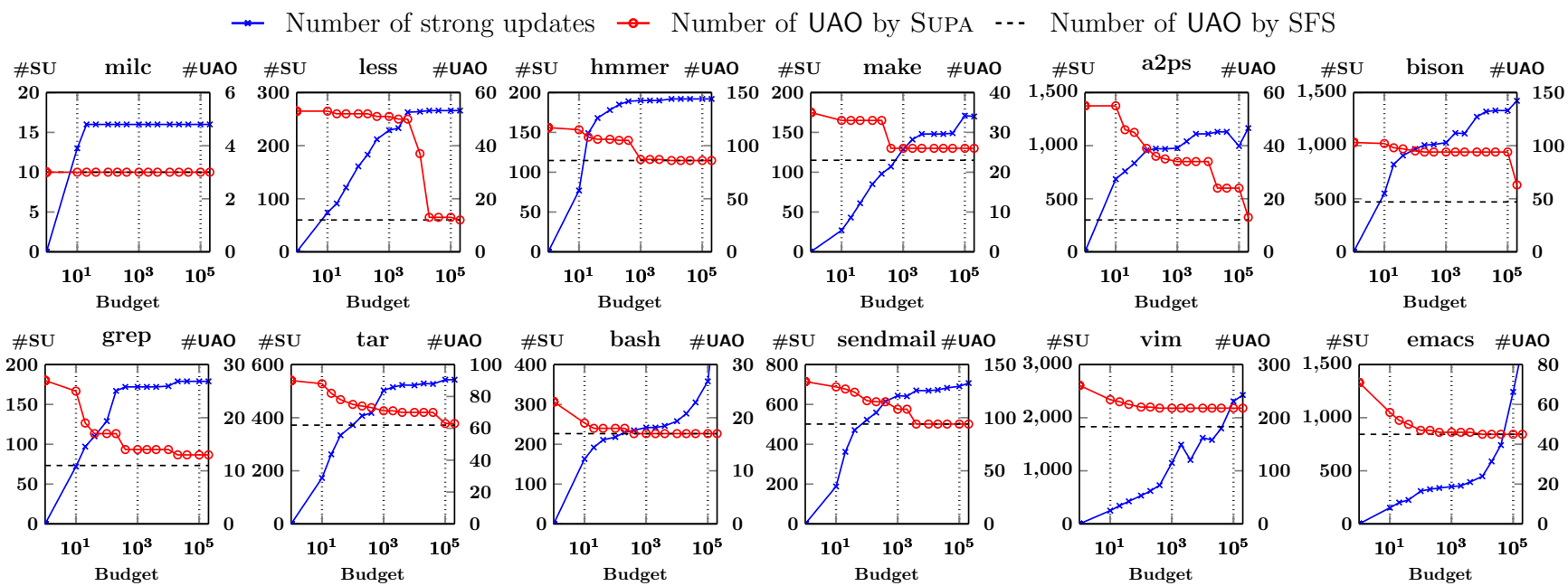
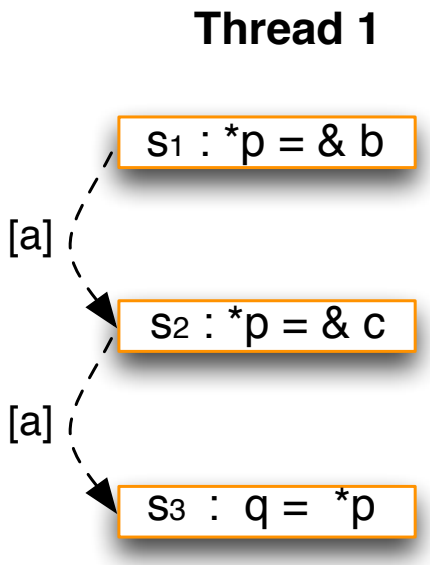
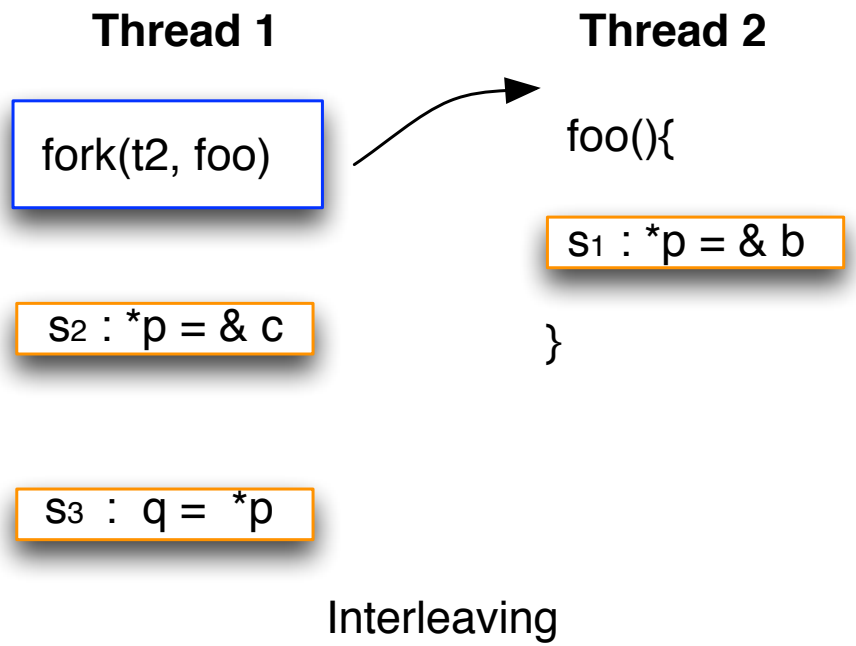


Figure: Correlating the number of strong updates with the number of unintialized variables detected under different budgets

Flow-Sensitivity Under Thread Interleaving (CGO '16)

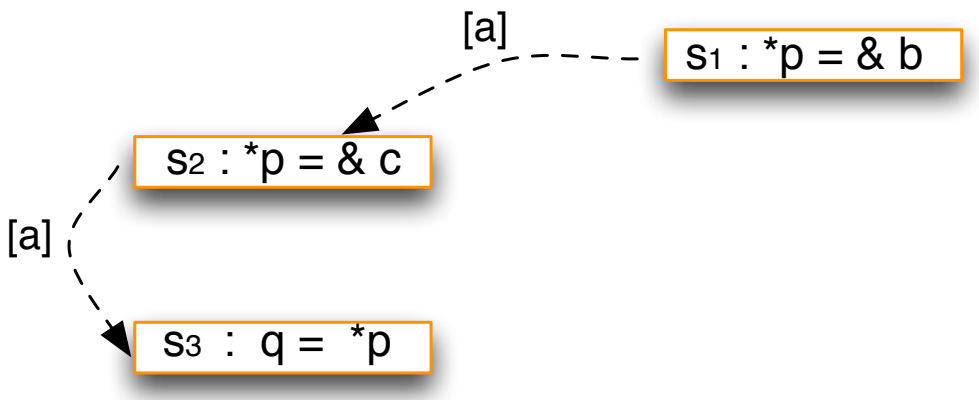


Flow-Sensitivity Under Thread Interleaving (CGO '16)



Flow-Sensitivity Under Thread Interleaving (CGO '16)

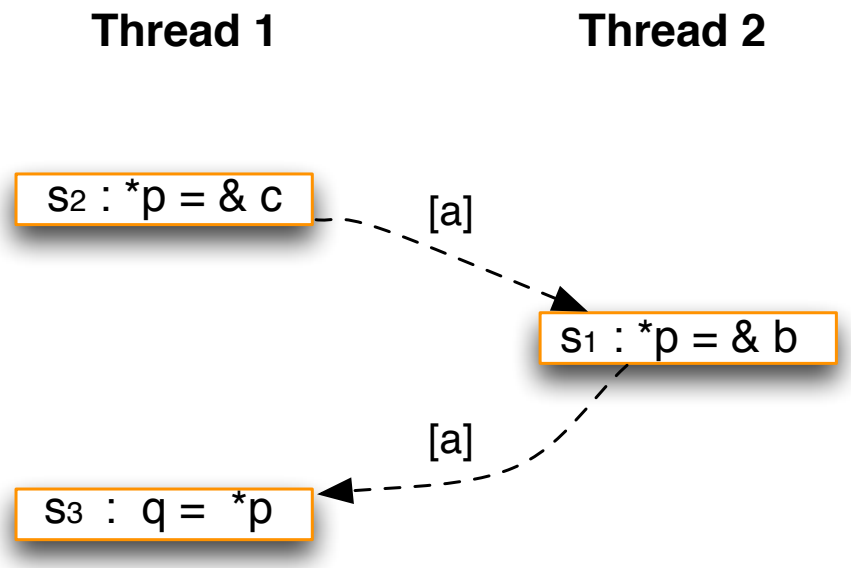
Scenario 1:



execution sequence : s_1, s_2, s_3 points-to of q : pt(q) = {c}

Flow-Sensitivity Under Thread Interleaving (CGO '16)

Scenario 2:

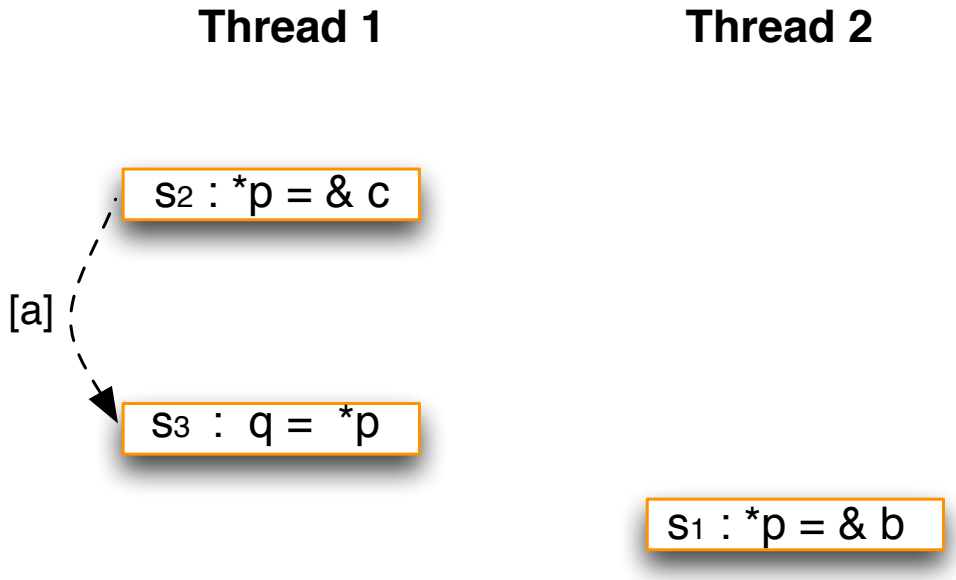


execution sequence : s_1, s_2, s_3 points-to of q : pt(q) = {c}

execution sequence : s_2, s_1, s_3 points-to of q : pt(q) = {b}

Flow-Sensitivity Under Thread Interleaving (CGO '16)

Scenario 3:

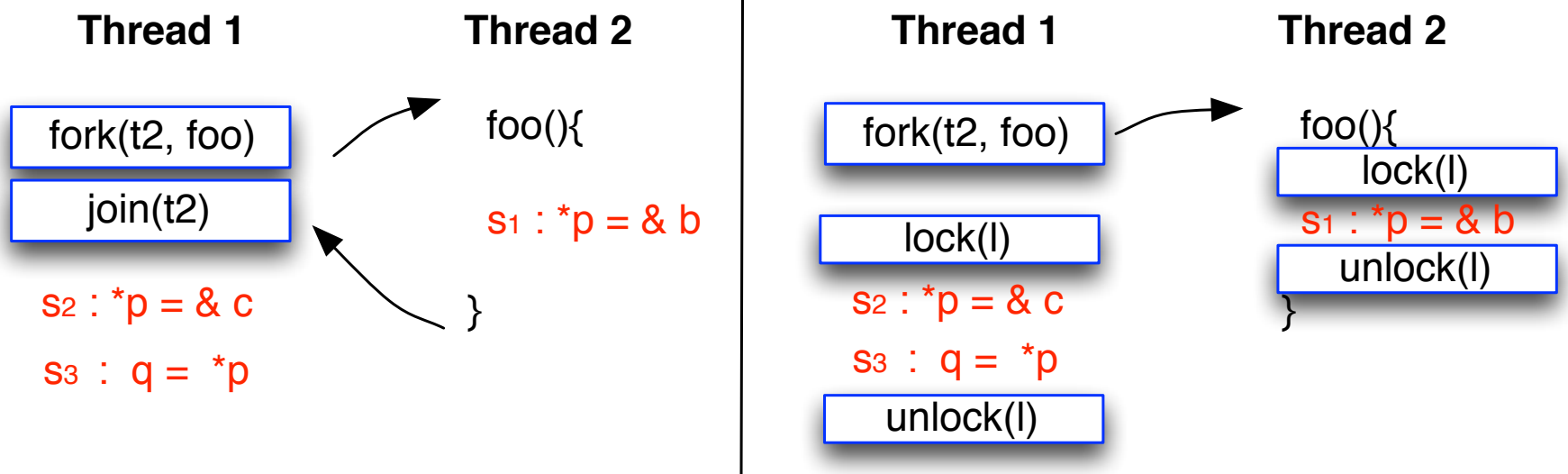


execution sequence : s_1, s_2, s_3 points-to of q : pt(q) = {c}

execution sequence : s_2, s_1, s_3 points-to of q : pt(q) = {b}

execution sequence : s_2, s_3, s_1 points-to of q : pt(q) = {c}

Flow-Sensitivity Under Thread Interleaving (CGO '16)

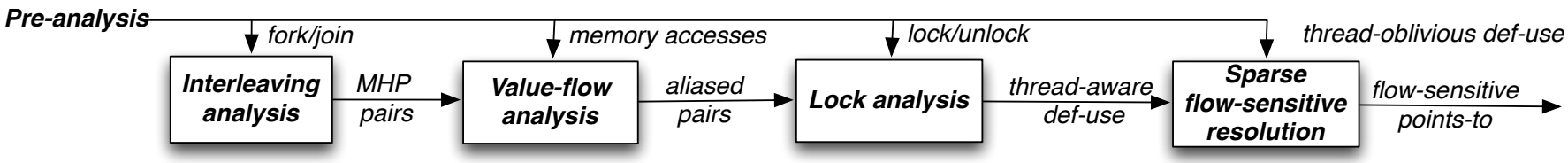


(a) non-interference via join

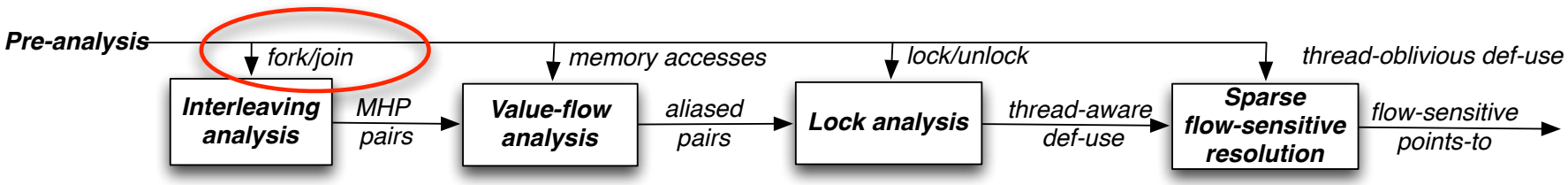
(b) non-interference via lock/unlock

points-to of q: pt(q) = {c}

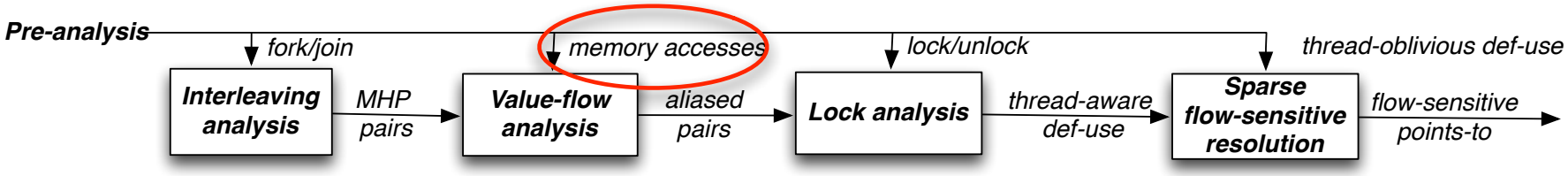
FSAM: Sparse Flow-Sensitive Analysis For Multithreaded Programs (CGO '16)



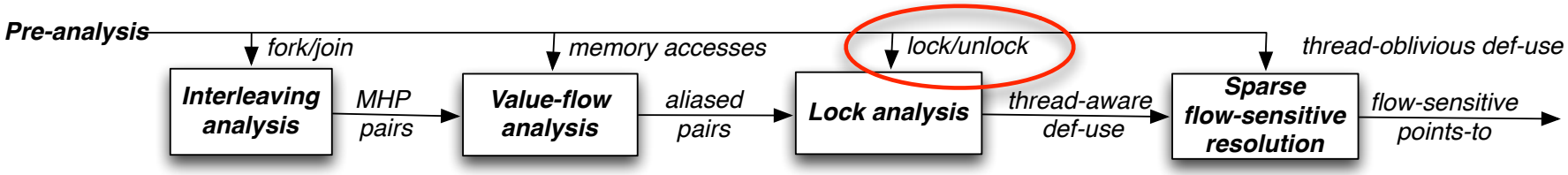
FSAM: Sparse Flow-Sensitive Analysis For Multithreaded Programs (CGO '16)



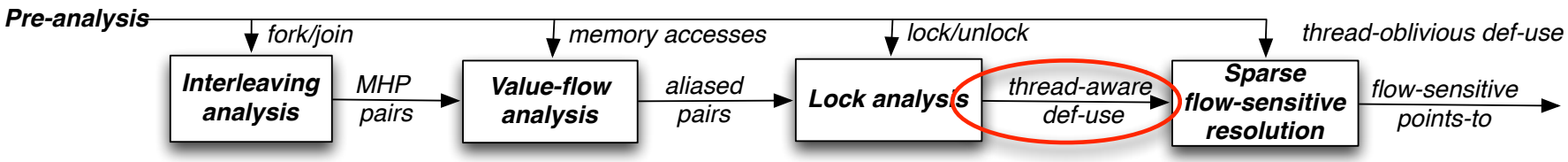
FSAM: Sparse Flow-Sensitive Analysis For Multithreaded Programs (CGO '16)



FSAM: Sparse Flow-Sensitive Analysis For Multithreaded Programs (CGO '16)



FSAM: Sparse Flow-Sensitive Analysis For Multithreaded Programs (CGO '16)



Context-Sensitive Abstract Threads (CGO '16)

An abstract thread *t* refers to a call of pthread_create() at a context-sensitive fork site during the analysis.

```
void main(){

cs1: foo();
cs2: foo();

}
```

```
void foo(){

cs3: fork(t1, bar);
}
```

t1 *refers to fork site under context [1,3]*

t1' *refers to fork site under context [2,3]*

t1 and t1' are context-sensitive threads

Context-Sensitive Abstract Threads (CGO '16)

An abstract thread *t* refers to a call of pthread_create() at a context-sensitive fork site during the analysis.

```
void main(){

cs1: foo();
cs2: foo();

}
```

t1 *refers to fork site under context [1,3]*

```
void foo(){

cs3: fork(t1, bar);
   }
```

t1' *refers to fork site under context [2,3]*

t1 and t1' are context-sensitive threads

```
void main(){

   for(i=0;i<10;i++){
      fork(t[i], foo)
   }

}
```

t is multi-forked thread

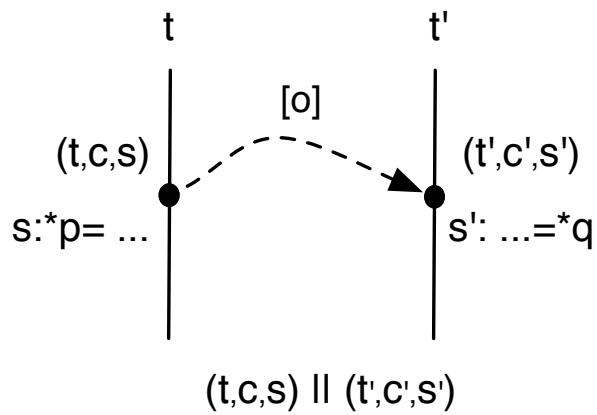
A thread *t* always refers to a context-sensitive fork site, i.e., a unique runtime thread unless $t \in \mathcal{M}$ is *multi-forked*, in which case, t may represent more than one runtime thread.

Thread-Aware Value-Flows (CGO '16)

A thread-aware def-use is added if a pair of statements (t, c, s) and (t', c', s')

- (1) may access same memory using pre-computed results.
- (2) may happen in parallel

$$\frac{s : *p = \_ \qquad s' : \_ = *q \quad \text{or} \quad *q = \_ \qquad (t, c, s) \parallel (t', c', s') \qquad o \in Alias(*p, *q)}{s \overset{o}{\hookrightarrow} s'}$$



Context-sensitive Thread Interleaving Analysis (CGO '16)

$(t_1, c_1, s_1) \parallel (t_2, c_2, s_2)$ holds if:

$$\begin{cases} t_2 \in \mathcal{I}(t_1, c_1, s_1) \wedge t_1 \in \mathcal{I}(t_2, c_2, s_2) & \text{if } t_1 \neq t_2 \\ t_1 \in \mathcal{M} & \text{otherwise} \end{cases}$$

where $\mathcal{I}(t, c, s)$: denotes a set of interleaved threads may run in parallel with s in thread t under calling context c,
$\mathcal{M}$ is the set of multi-forked threads.

Interleaving Analysis (CGO '16)

Computing $\mathcal{I}(t, c, s)$ is formalized as a forward data-flow problem $(V, \sqcap, F)$.

- V: the set of all thread interleaving facts.
- $\sqcap$: meet operator ($\cup$).
- F: $V \rightarrow V$ transfer functions associated with each node in an ICFG.

Interleaving Analysis Rule

$$\texttt{[I-DESCENDANT]}\quad \frac{t \xrightarrow{(c,fk_i)} t' \quad (t,c,fk_i) \rightarrow (t,c,l) \quad (c',l') = \mathit{Entry}(\mathcal{S}_{t'})}{\{t'\} \subseteq \mathcal{I}(t,c,l) \quad \{t\} \subseteq \mathcal{I}(t',c',l')}$$

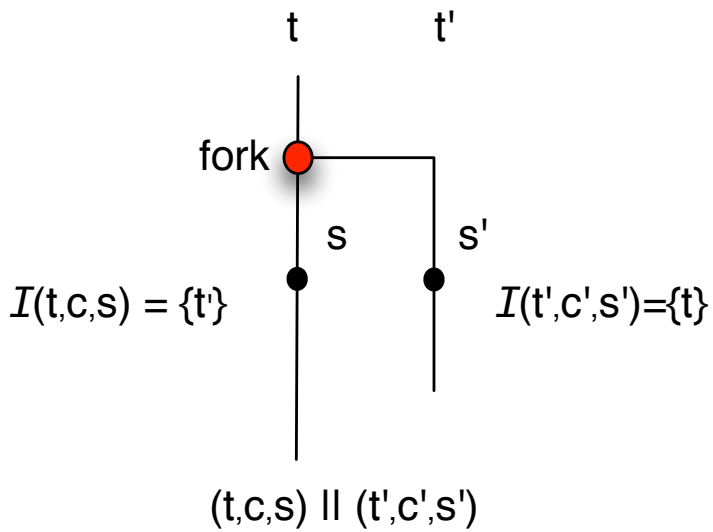
$$\texttt{[I-SIBLING]}\quad \frac{t \bowtie t' \quad (c,l) = \mathit{Entry}(\mathcal{S}_t) \quad (c',l') = \mathit{Entry}(\mathcal{S}_{t'}) \quad t \not\succ t' \wedge t' \not\succ t}{\{t\} \subseteq \mathcal{I}(t',c',l') \quad \{t'\} \subseteq \mathcal{I}(t,c,l)}$$

$$\texttt{[I-JOIN]}\quad \frac{t \xleftarrow{(c,jn_i)} t'}{\mathcal{I}(t,c,jn_i) = \mathcal{I}(t,c,jn_i) \backslash \{t'\}} \qquad \texttt{[I-CALL]}\quad \frac{(t,c,l) \xrightarrow{call_i} (t,c',l') \quad c' = c.\mathit{push}(i)}{\mathcal{I}(t,c,l) \subseteq \mathcal{I}(t,c',l')}$$

$$\texttt{[I-INTRA]}\quad \frac{(t,c,l) \rightarrow (t,c,l')}{\mathcal{I}(t,c,l) \subseteq \mathcal{I}(t,c,l')} \qquad \texttt{[I-RET]}\quad \frac{(t,c,l) \xrightarrow{ret_i} (t,c',l') \quad i = c.\mathit{peek}() \quad c' = c.\mathit{pop}()}{\mathcal{I}(t,c,l) \subseteq \mathcal{I}(t,c',l')}$$

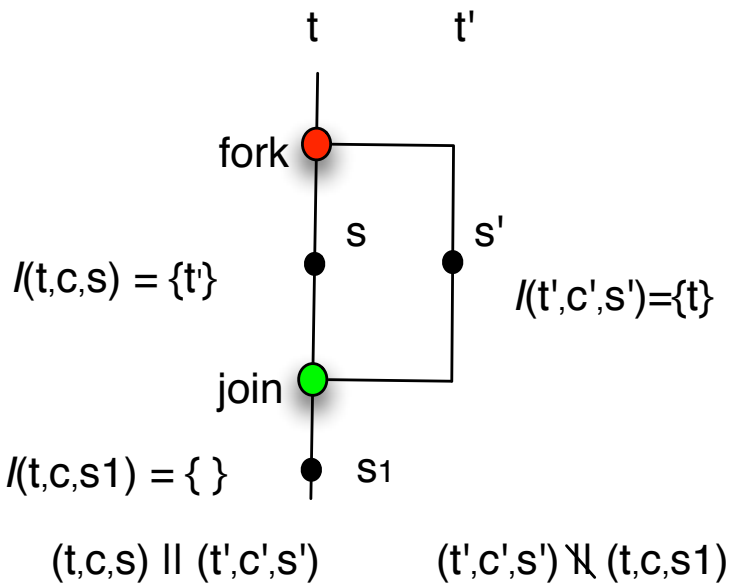
Interleaving Analysis Rule (CGO '16)

$$\texttt{[I-DESCENDANT]}\quad \frac{t \xrightarrow{(c,fk_i)} t' \quad (t,c,fk_i) \rightarrow (t,c,l) \quad (c',l') = Entry(\mathcal{S}_{t'})}{\{t'\} \subseteq \mathcal{I}(t,c,l) \quad \{t\} \subseteq \mathcal{I}(t',c',l')}$$



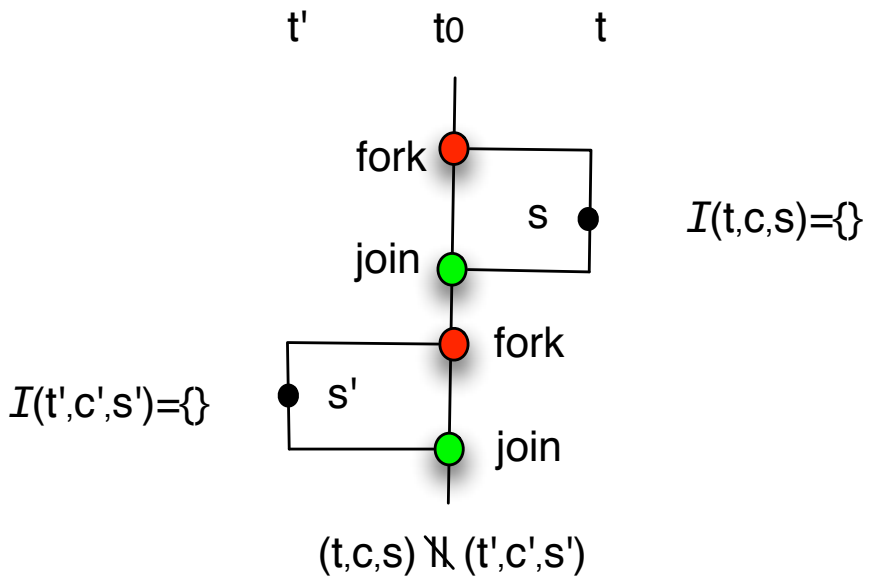
Interleaving Analysis Rule (CGO '16)

$$\texttt{[I-JOIN]}\quad \frac{t \xleftarrow{(c,jn_i)} t'}{\mathcal{I}(t,c,jn_i) = \mathcal{I}(t,c,jn_i) \backslash \{t'\}}$$



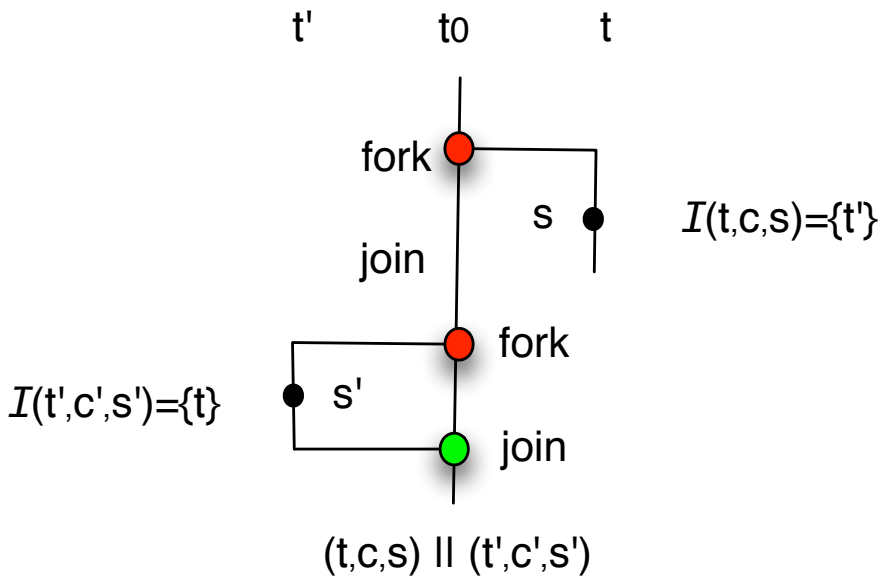
Interleaving Analysis Rule (CGO '16)

$$\texttt{[I-SIBLING]}\quad \frac{t \bowtie t' \quad (c, l) = \mathit{Entry}(\mathcal{S}_t) \quad (c', l') = \mathit{Entry}(\mathcal{S}_{t'}) \quad t \not\succ t' \wedge t' \not\succ t}{\{t\} \subseteq \mathcal{I}(t', c', l') \quad \{t'\} \subseteq \mathcal{I}(t, c, l)}$$



Interleaving Analysis Rule (CGO '16)

$$\texttt{[I-SIBLING]}\quad \frac{t \bowtie t' \quad (c, l) = \mathit{Entry}(\mathcal{S}_t) \quad (c', l') = \mathit{Entry}(\mathcal{S}_{t'}) \quad t \not\succ t' \wedge t' \not\succ t}{\{t\} \subseteq \mathcal{I}(t', c', l') \quad \{t'\} \subseteq \mathcal{I}(t, c, l)}$$



Interleaving Analysis Rule (CGO '16)

$$\texttt{[I-DESCENDANT]}\quad \frac{t \xrightarrow{(c,fk_i)} t' \quad (t,c,fk_i) \rightarrow (t,c,l) \quad (c',l') = \mathit{Entry}(\mathcal{S}_{t'})}{\{t'\} \subseteq \mathcal{I}(t,c,l) \quad \{t\} \subseteq \mathcal{I}(t',c',l')}$$

$$\texttt{[I-SIBLING]}\quad \frac{t \bowtie t' \quad (c,l) = \mathit{Entry}(\mathcal{S}_t) \quad (c',l') = \mathit{Entry}(\mathcal{S}_{t'}) \quad t \not\succ t' \wedge t' \not\succ t}{\{t\} \subseteq \mathcal{I}(t',c',l') \quad \{t'\} \subseteq \mathcal{I}(t,c,l)}$$

$$\texttt{[I-JOIN]}\quad \frac{t \xleftarrow{(c,jn_i)} t'}{\mathcal{I}(t,c,jn_i) = \mathcal{I}(t,c,jn_i) \backslash \{t'\}} \qquad \texttt{[I-CALL]}\quad \frac{(t,c,l) \xrightarrow{call_i} (t,c',l') \quad c' = c.push(i)}{\mathcal{I}(t,c,l) \subseteq \mathcal{I}(t,c',l')}$$

$$\texttt{[I-INTRA]}\quad \frac{(t,c,l) \rightarrow (t,c,l')}{\mathcal{I}(t,c,l) \subseteq \mathcal{I}(t,c,l')} \qquad \texttt{[I-RET]}\quad \frac{(t,c,l) \xrightarrow{ret_i} (t,c',l') \quad i = c.peek() \quad c' = c.pop()}{\mathcal{I}(t,c,l) \subseteq \mathcal{I}(t,c',l')}$$

Lock Analysis (CGO '16)

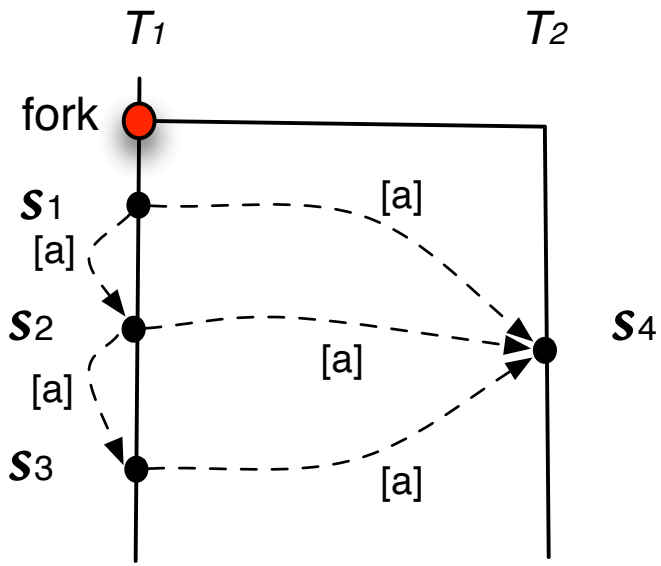
Statements from different mutex regions are interference-free if these regions are protected by a common lock.

Thread 1

```
main(){
fork(t2, foo)
 s1 : *p = & c
 s2 : *p = & d
 s3 : *p = & e
}
```

Thread 2

```
foo(){
 s4 : q = *p
}
```



Lock Analysis

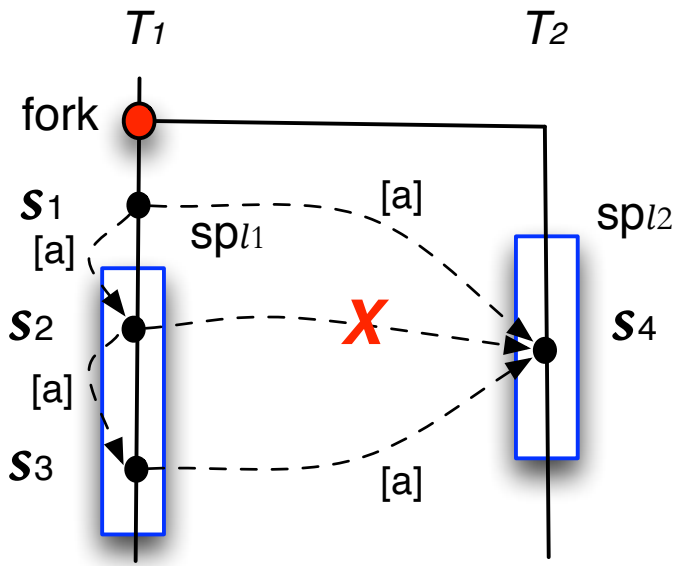
Statements from different mutex regions are interference-free if these regions are protected by a common lock.

Thread 1

```
main(){
 fork(t2, foo)
  s1 : *p = & c
       lock(l)
  s2 : *p = & d
  s3 : *p = & e
       unlock(l)
}
```

Thread 2

```
 foo(){
  lock(l)
  s4 : q = *p
  unlock(l)
}
```



Outline

- Background and Motivation
- Our approach: FSAM
- Evalution

Evaluation

- Implementation:
  - On top of our previous open-source tool SVF (http://svf-tools.github.io/SVF/) (CC '16)
  - Around 4,000 LOC core source code
  - Field-sensitivity: each field instance of a struct is treated as a separate object, arrays are considered monolithic.
  - On-the-fly call graph construction.

[2] *Radu Rugina and Martin Rinard*, Pointer Analysis for Multithreaded Programs PLDI '99

Evaluation

- Implementation:
  - On top of our previous open-source tool SVF (http://svf-tools.github.io/SVF/) (CC '16)
  - Around 4,000 LOC core source code
  - Field-sensitivity: each field instance of a struct is treated as a separate object, arrays are considered monolithic.
  - On-the-fly call graph construction.
- Methodology
  - FSAM v.s. NONSPARSE iterative flow-sensitive analysis following RR algorithm[2]

[2] *Radu Rugina and Martin Rinard*, Pointer Analysis for Multithreaded Programs PLDI '99

Evaluation

- Implementation:
  - On top of our previous open-source tool SVF (http://svf-tools.github.io/SVF/) (CC '16)
  - Around 4,000 LOC core source code
  - Field-sensitivity: each field instance of a struct is treated as a separate object, arrays are considered monolithic.
  - On-the-fly call graph construction.
- Methodology
  - FSAM v.s. NONSPARSE iterative flow-sensitive analysis following RR algorithm[2]
- Benchmarks:
  - Two largest C benchmarks from `Phoenix-2.0`
  - Five largest C benchmarks from `Parsec-3.0`
  - Three open-source applications
- Machine setup:
  - Ubuntu Linux 3.11 Intel Xeon Quad Core, 3.7GHZ, 64GB

[2] *Radu Rugina and Martin Rinard*, Pointer Analysis for Multithreaded Programs PLDI '99

Benchmarks

Table: Program statistics.

Benchmark	Description	LOC
word_count	Word counter based on map-reduce	6330
kmeans	Iterative clustering of 3-D points	6008
radiosity	Graphics	12781
automount	Manage autofs mount points	13170
ferret	Content similarity search server	15735
bodytrack	Body tracking of a person	19063
httpd_server	Http server	52616
mt_daapd	Multi-threaded DAAP Daemon	57102
raytrace	Real-time raytracing	84373
x264	Media processing	113481
Total		380,659

RR only evaluated their analysis with benchmarks with up to 4500 lines of Cilk code.

Analysis Time and Memory Usage

Table: Analysis time and memory usage.

Program	Time (Secs)		Memory (MB)	
	FSAM	NONSPARSE	FSAM	NONSPARSE
word_count	3.04	17.40	13.79	53.76
kmeans	2.50	18.19	18.27	53.19
radiosity	6.77	29.29	38.65	95.00
automount	8.66	83.82	27.56	364.67
ferret	13.49	87.10	52.14	934.57
bodytrack	128.80	2809.89	313.66	12410.16
httpd_server	191.22	2079.43	55.78	6578.46
mt_daapd	90.67	2667.55	37.92	3403.26
raytrace	284.61	OOT	135.06	OOT
x264	531.55	OOT	129.58	OOT

FSAM is 12x faster and uses 28x less memory.

Impact of FSAM's three thread interference analysis

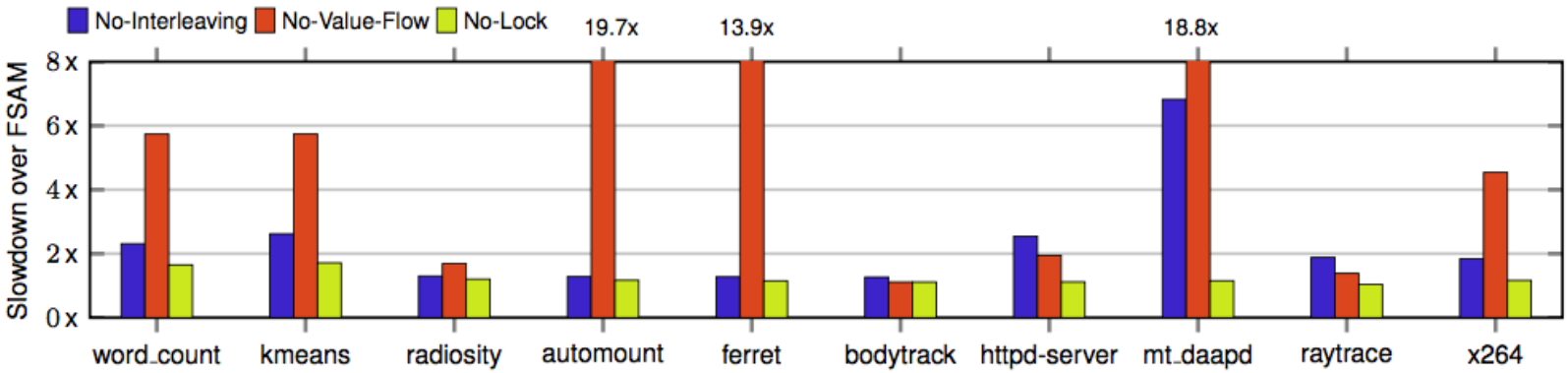
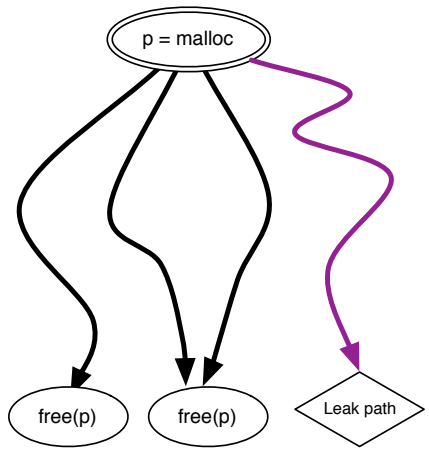


Figure: Impact of FSAM's three thread interference analysis phases on the performance of flow-sensitive points-to resolution.

Outline

- **Existing software bugs and vulnerabilities**
- **Static analysis and dynamic analysis**
- **Technical contributions in developing scalable and precise program analysis**
  - Fundamental program analysis
    - Sparse value-flow analysis
    - Selective and on-demand value-flow analysis
  - Applications
    - Static value-flow analysis for detecting memory errors
    - Hybrid value-flow analysis to enforce memory safety
- **Research opportunities**

Value-Flow Analysis For Memory Leak Detection (ISSTA '12))



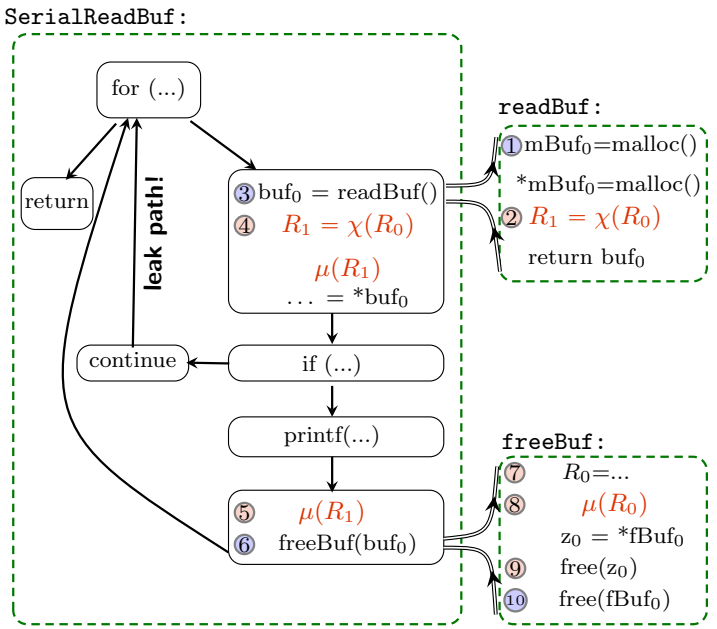
Yulei Sui, Ding Ye, Jingling Xue. Static Memory Leak Detection Using Full-Sparse Value Flow Analysis. *2012 International Symposium on Software Testing and Analysis (**ISSTA '12**)*

Yulei Sui, Ding Ye, Jingling Xue. Detecting Memory Leaks Statically with Full-Sparse Value-Flow Analysis. *IEEE Transactions on Software Engineering (**TSE '14**)*

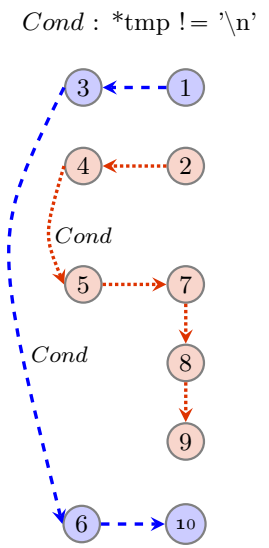
Value-Flow Analysis For Memory Leak Detection (ISSTA '12))

```
1  void SerialReadBuf() {
2    for (n=0; n<100; n++) {
3      char** buf = readBuf();
4      char* tmp = *buf;
5      if (*tmp != '\n')
6        printf("%s",*tmp);
7      else
8        continue;
9      freeBuf(buf);
10   }
11 }
12 char** readBuf() {
13   char** mBuf = malloc(); //o
14   *mBuf = malloc(); //o'
15   //... (write into **mBuf);
16   return mBuf;
17 }
18 void freeBuf(char** fBuf) {
19   char* z = *fBuf;
20   free(z);
21   free(fBuf);
22 }
```

(a) Input program



(b) Full-sparse SSA on Inter-procedural CFG



(c) SVFG (with its unlabelled edges being guarded by true)

Yulei Sui, Ding Ye, Jingling Xue. Static Memory Leak Detection Using Full-Sparse Value Flow Analysis. *2012 International Symposium on Software Testing and Analysis (**ISSTA '12**)*

Yulei Sui, Ding Ye, Jingling Xue. Detecting Memory Leaks Statically with Full-Sparse Value-Flow Analysis. *IEEE Transactions on Software Engineering (**TSE '14**)*

Value-Flow Analysis For Memory Leak Detection (ISSTA '12))

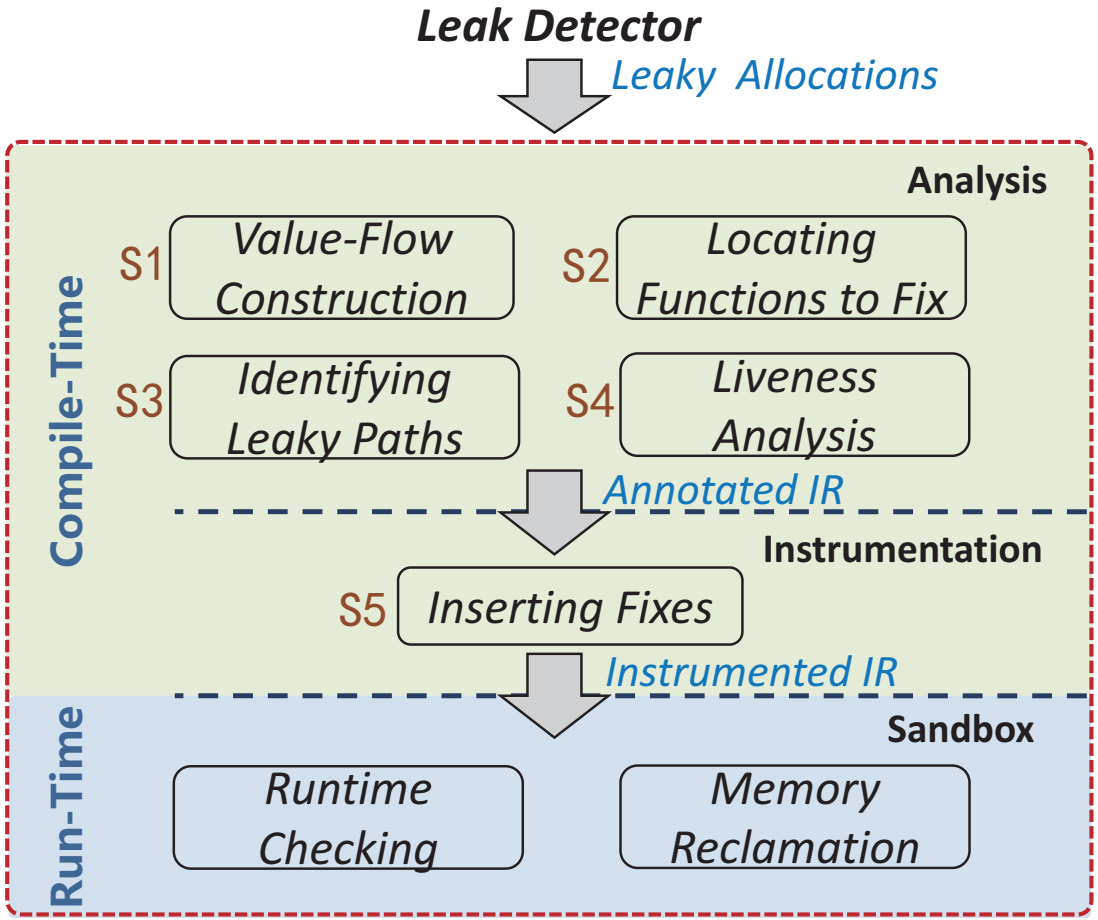
Leak Detector	Speed (LOC/sec)	Bug Count	FP Rate(%)
Athena	50	53	10
CONTRADICTION	300	26	56
CLANG	400	27	25
SPARROW	720	81	16
FASTCHECK	37,900	59	14
SABER	**10,220**	**85**	**19**

Comparing SABER with other static detectors on analysing SPEC2000 C programs

Yulei Sui, Ding Ye, Jingling Xue. Static Memory Leak Detection Using Full-Sparse Value Flow Analysis. *2012 International Symposium on Software Testing and Analysis (**ISSTA '12**)*

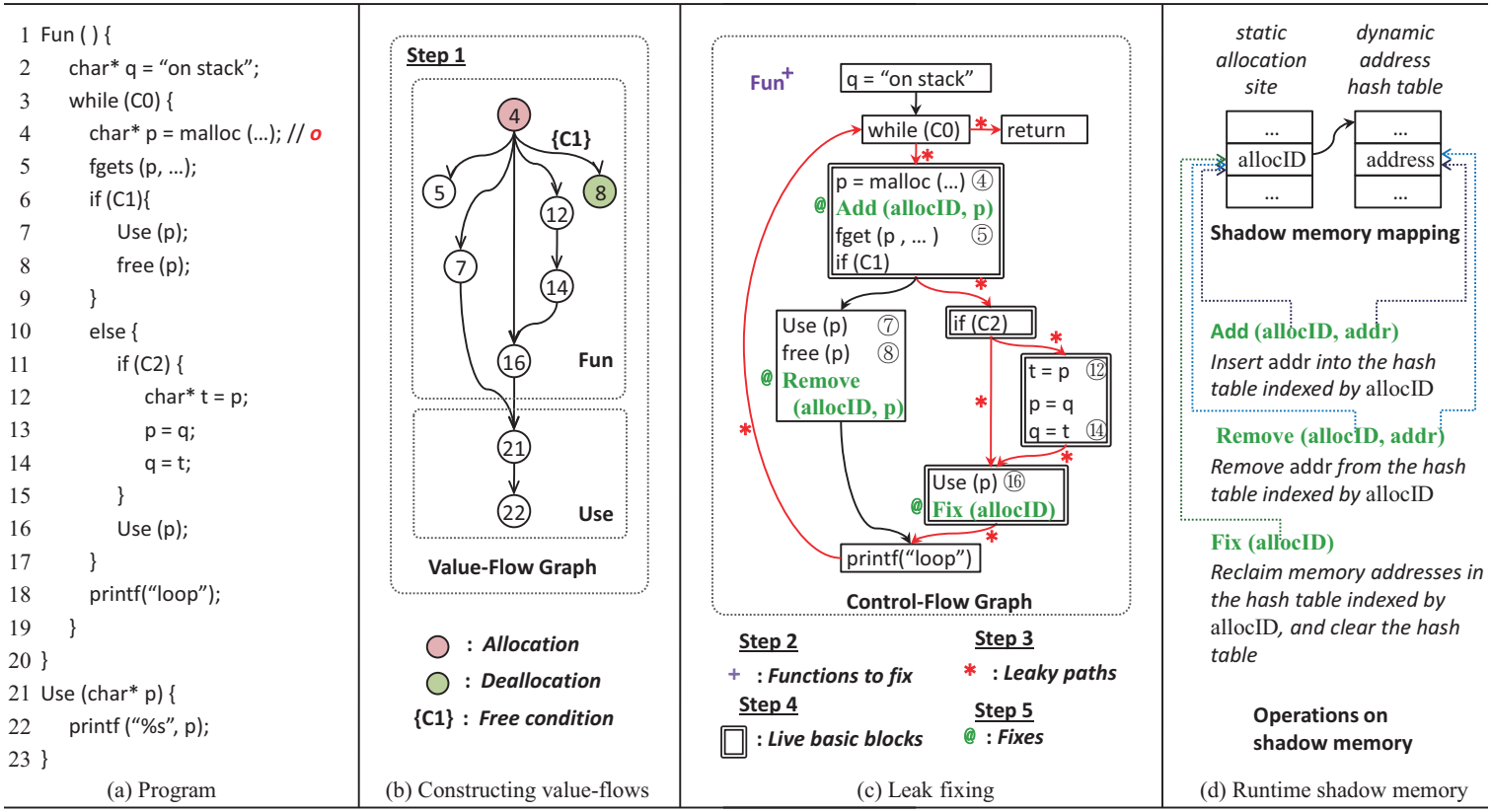
Yulei Sui, Ding Ye, Jingling Xue. Detecting Memory Leaks Statically with Full-Sparse Value-Flow Analysis. *IEEE Transactions on Software Engineering (**TSE '14**)*

Value-Flow Analysis for Automatic Bug Fixing (SAC '16)



Hua Yan, Yulei Sui, Shiping Chen, Jingling Xue. Automated Memory Leak Fixing on Value-Flow Slices For C Programs. *31st ACM Symposium on Applied Computing (**SAC '16**)*

Value-Flow Analysis for Automatic Bug Fixing (SAC '16)



Hua Yan, Yulei Sui, Shiping Chen, Jingling Xue. Automated Memory Leak Fixing on Value-Flow Slices For C Programs. *31st ACM Symposium on Applied Computing (**SAC '16**)*

Detecting Use-After-Free Vulnerabilities (ICSE '18)

Use-after-free (UAF)

- UAF is also known as dangling pointer dereference, i.e., referencing a memory object after it has been freed
- In Common Weakness Enumeration (CWE-416)
- Use-after-free is one of the most serious security vulnerabilities
  - Crashes
  - Data corruption
  - Information leakage
  - Control-flow hijacking

Detecting Use-After-Free Vulnerabilities (ICSE '18)

A simple attack model

```
 1: typedef void (*func_ptr)();

 2: void foo() {…}

 3: int main() {
 4:   func_ptr* p = malloc(4);
 5:   func_ptr* q = p;
 6:   *p = &foo;
 7:   free(p);
 8:   long int* r = malloc(4);
 9:   *r = userInput();
10:   (*q)();  // UAF bug
    }
```

Runtime memory layout

Detecting Use-After-Free Vulnerabilities (ICSE '18)

A simple attack model

```
 1: typedef void (*func_ptr)();

 2: void foo() {…}

 3: int main() {
 4:   func_ptr* p = malloc(4);
 5:   func_ptr* q = p1;
 6:   *p = &foo;
 7:   free(p);
 8:   long int* r = malloc(4);
 9:   *r = userInput();
10:   (*q)();  // UAF bug
    }
```

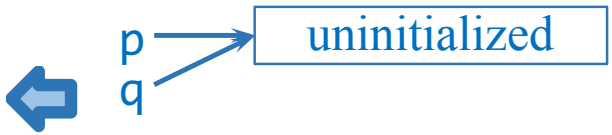
Runtime memory layout

Detecting Use-After-Free Vulnerabilities (ICSE '18)

A simple attack model

```
 1: typedef void (*func_ptr)();

 2: void foo() {…}

 3: int main() {
 4:   func_ptr* p = malloc(4);
 5:   func_ptr* q = p1;
 6:   *p = &foo;
 7:   free(p);
 8:   long int* r = malloc(4);
 9:   *r = userInput();
10:   (*q)();  // UAF bug
    }
```

Runtime memory layout



Detecting Use-After-Free Vulnerabilities (ICSE '18)

A simple attack model

```
 1: typedef void (*func_ptr)();

 2: void foo() {…}

 3: int main() {
 4:   func_ptr* p = malloc(4);
 5:   func_ptr* q = p1;
 6:   *p = &foo;
 7:   free(p);
 8:   long int* r = malloc(4);
 9:   *r = userInput();
10:   (*q)();  // UAF bug
    }
```

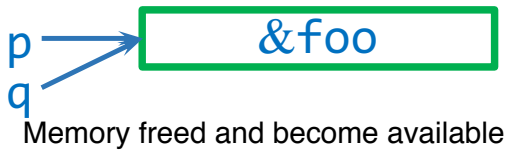
Runtime memory layout

Detecting Use-After-Free Vulnerabilities (ICSE '18)

A simple attack model

```
1:  typedef void (*func_ptr)();

2:  void foo() {…}

3:  int main() {
4:    func_ptr* p = malloc(4);
5:    func_ptr* q = p1;
6:    *p = &foo;
7:    free(p);
8:    long int* r = malloc(4);
9:    *r = userInput();
10:   (*q)();  // UAF bug
    }
```

Runtime memory layout

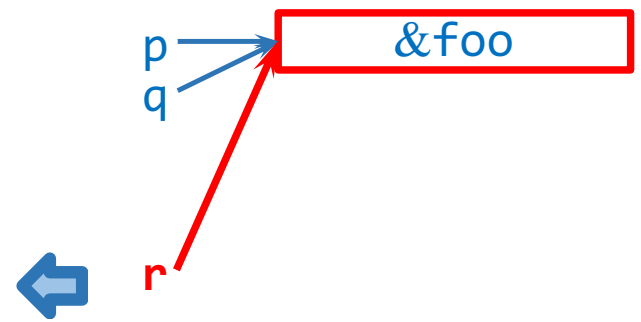


Detecting Use-After-Free Vulnerabilities (ICSE '18)

A simple attack model

```
1:  typedef void (*func_ptr)();

2:  void foo() {…}

3:  int main() {
4:    func_ptr* p = malloc(4);
5:    func_ptr* q = p1;
6:    *p = &foo;
7:    free(p);
8:    long int* r = malloc(4);
9:    *r = userInput();
10:   (*q)();  // UAF bug
    }
```

Runtime memory layout

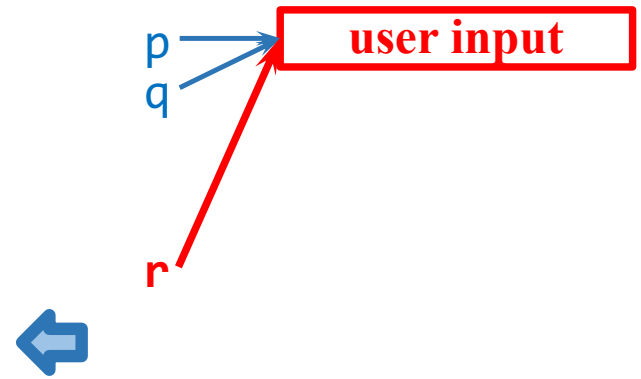


Detecting Use-After-Free Vulnerabilities (ICSE '18)

A simple attack model

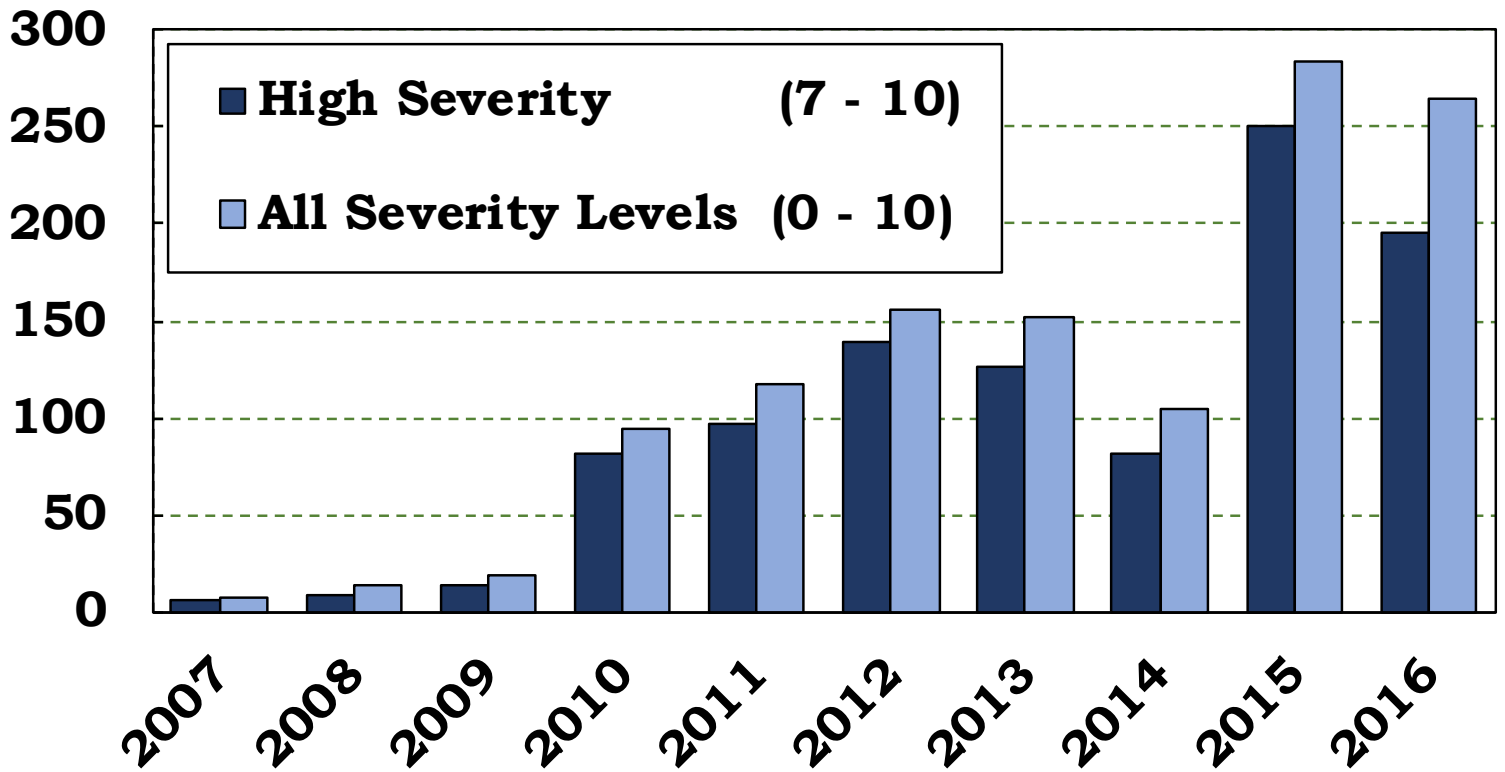
```
1:  typedef void (*func_ptr)();

2:  void foo() {…}

3:  int main() {
4:    func_ptr* p = malloc(4);
5:    func_ptr* q = p1;
6:    *p = &foo;
7:    free(p);
8:    long int* r = malloc(4);
9:    *r = userInput();
10:   (*q)();  // UAF bug
    }
```

Runtime memory layout



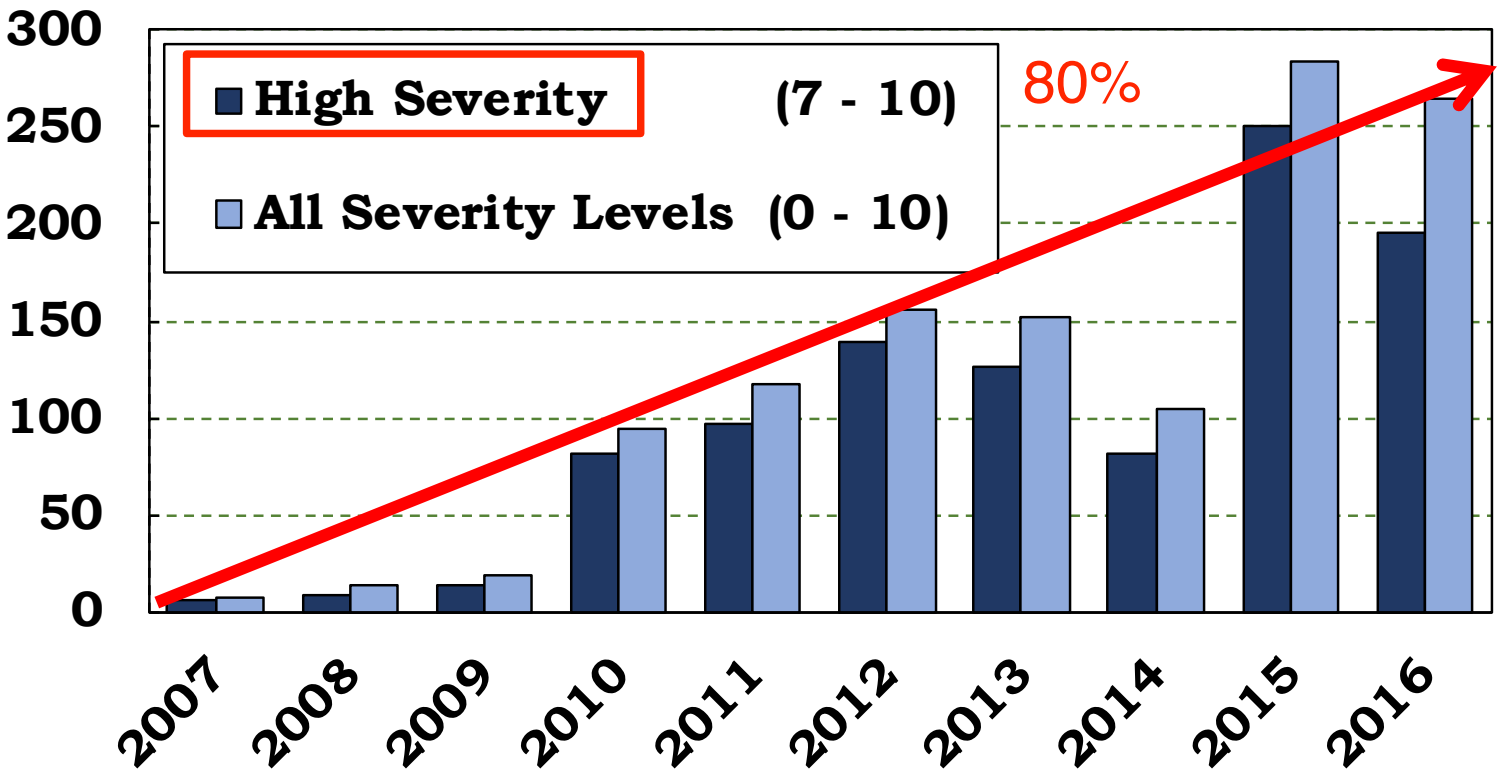
Detecting Use-After-Free Vulnerabilities (ICSE '18)

UAF in US National Vulnerability Database



Detecting Use-After-Free Vulnerabilities (ICSE '18)

UAF in US National Vulnerability Database



UNIVERSITY OF TECHNOLOGY, SYDNEY

Related Work – Dynamic Approaches

- Detection
  - *Full memory safety*: e.g., CETS [ISMM'10]
  - *Taint tracking*: e.g., Undangle [ISSTA'12]
  - *Redzone*: e.g., AddressSanitizer [Usenix ATC'12]
  - *Optimization*: e.g., DangSan [EuroSys'17]
- Mitigation
  - *Safe allocator*: e.g., DieHarder [CCS'10], Cling [Security'10], FreeGuard [CCS'17]
  - *Safe deallocator*: e.g., VTPin [ACSAC'16]
  - *Nullification*: e.g., DangNull [NDSS'15], FreeSentry [NDSS'15]
  - *Control-flow integrity*: e.g., CFI [CCS'05], PathArmor [CCS'15], ShrinkWrap [ACSAC'15]

Related Work – Static Approaches

- *Buffer overflow* E.g., Archer [FSE'03], Marple [FSE'08], Parfait [FSE'10]
- *Memory leak* E.g., Saturn [FSE'05], FastCheck [PLDI'07], Saber [ISSTA'12]
- *Information flow* E.g., TAJ [PLDI'09], Merlin [PLDI'14], DroidSafe [NDSS'15]
- *Data race* E.g., RacerX [SOSP'03], LockSmith [PLDI'06], DroidRacer [PLDI'14]
- *UAF* **Relatively unexplored**

Challenges

- Large program with complex features E.g., indirect calls, control-flow cycles, complex data structures
- Number of free-use pairs `php-5.6.8`: **1,391 frees x 244,917 uses = 340 million pairs** with **billions** of calling contexts.
- Inter-procedural analysis
- Pointer analysis

Detecting Use-After-Free Vulnerabilities (ICSE '18)

Spatio-temporal correlation

Problem Statement

Consider a pair of statements, $\big(\texttt{free}(p@l_f), \texttt{use}(q@l_u)\big)$, where p and q are pointers and l_f and l_u are line numbers. Let $\mathcal{P}(l)$ be the set of all feasible (concrete) program paths reaching line l from `main()`. The pair is a UAF if and only if the following holds:

[Spatio-Temporal Correlation]

$$\mathbb{ST}\big(\texttt{free}(p@l_f), \texttt{use}(q@l_u)\big) \quad := \quad \exists\, (\rho_f, \rho_u) \in \mathcal{P}(l_f) \times \mathcal{P}(l_u) : (\rho_f, l_f) \rightsquigarrow (\rho_u, l_u) \;\wedge\; (\rho_f, p) \cong (\rho_u, q) \tag{1}$$

Detecting Use-After-Free Vulnerabilities (ICSE '18)

Conservative spatio-temporal correlation

[Spatio-Temporal Correlation with a High Level of Spuriosity]

$$\mathbb{ST}^{\text{SUPA}}\left(\texttt{free}(p@l_f), \texttt{use}(q@l_u)\right) \quad := \quad ([\,],l_f) \rightsquigarrow ([\,],l_u) \ \wedge \ ([\,],p) \cong ([\,],q) \tag{2}$$

$\mathcal{P}(l_f) \times \mathcal{P}(l_u)$ represents an extremely coarse abstraction, $\{[\,]\} \times \{[\,]\}$ where $[\,]$ represents all possible calling contexts (paths) reaching l.

Detecting Use-After-Free Vulnerabilities (ICSE '18)

Spatio-temporal context reduction

[Spatio-Temporal Context Reduction]

$$\mathbb{ST}^{\mathsf{STC}}\left(\texttt{free}(p@l_f), \texttt{use}(q@l_u)\right) \quad := \tag{3}$$

$$\exists\, (\widetilde{c}_f, \widetilde{c}_u) \in \widetilde{\mathcal{P}}(l_f) \times \widetilde{\mathcal{P}}(l_u) : (\widetilde{c}_f, l_f) \rightsquigarrow (\widetilde{c}_u, l_u) \;\wedge\; (\widetilde{c}_f, p) \cong (\widetilde{c}_u, q)$$

We ensure that $\mathbb{ST}^{\mathsf{STC}}$ is sound by requiring $\widetilde{\mathcal{P}}(l)$ to be a coarser abstraction of $\mathcal{P}(l)$ and scalable by requiring $|\widetilde{\mathcal{P}}(l_f) \times \widetilde{\mathcal{P}}(l_u)| \ll |\mathcal{P}(l_f) \times \mathcal{P}(l_u)|$, but to achieve as precise as that for full context-sensitive analysis.

Detecting Use-After-Free Vulnerabilities (ICSE '18)

Spatio-temporal context reduction

$$(c_{fu} \oplus \widetilde{h}\ , o) \in pt^{\infty}(c_{fu} \oplus \widetilde{c_f}\ , p) \cap pt^{\infty}(c_{fu} \oplus \widetilde{c_u}\ , q)$$

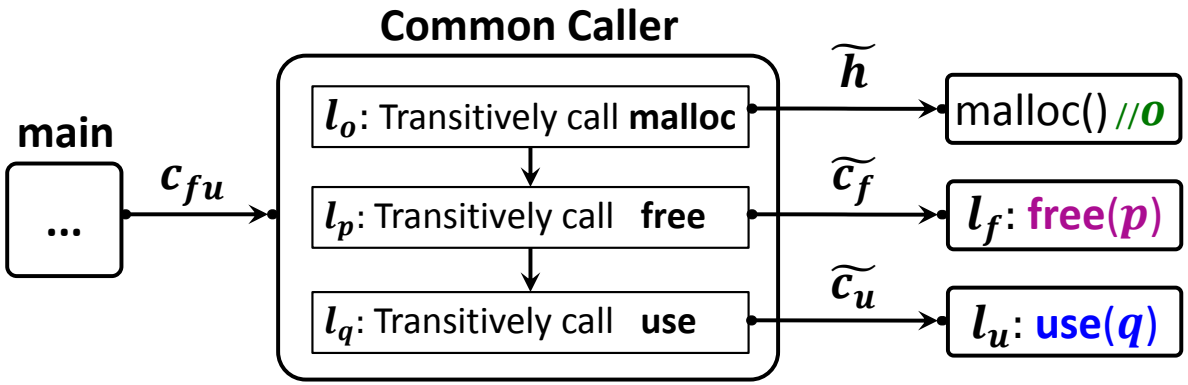
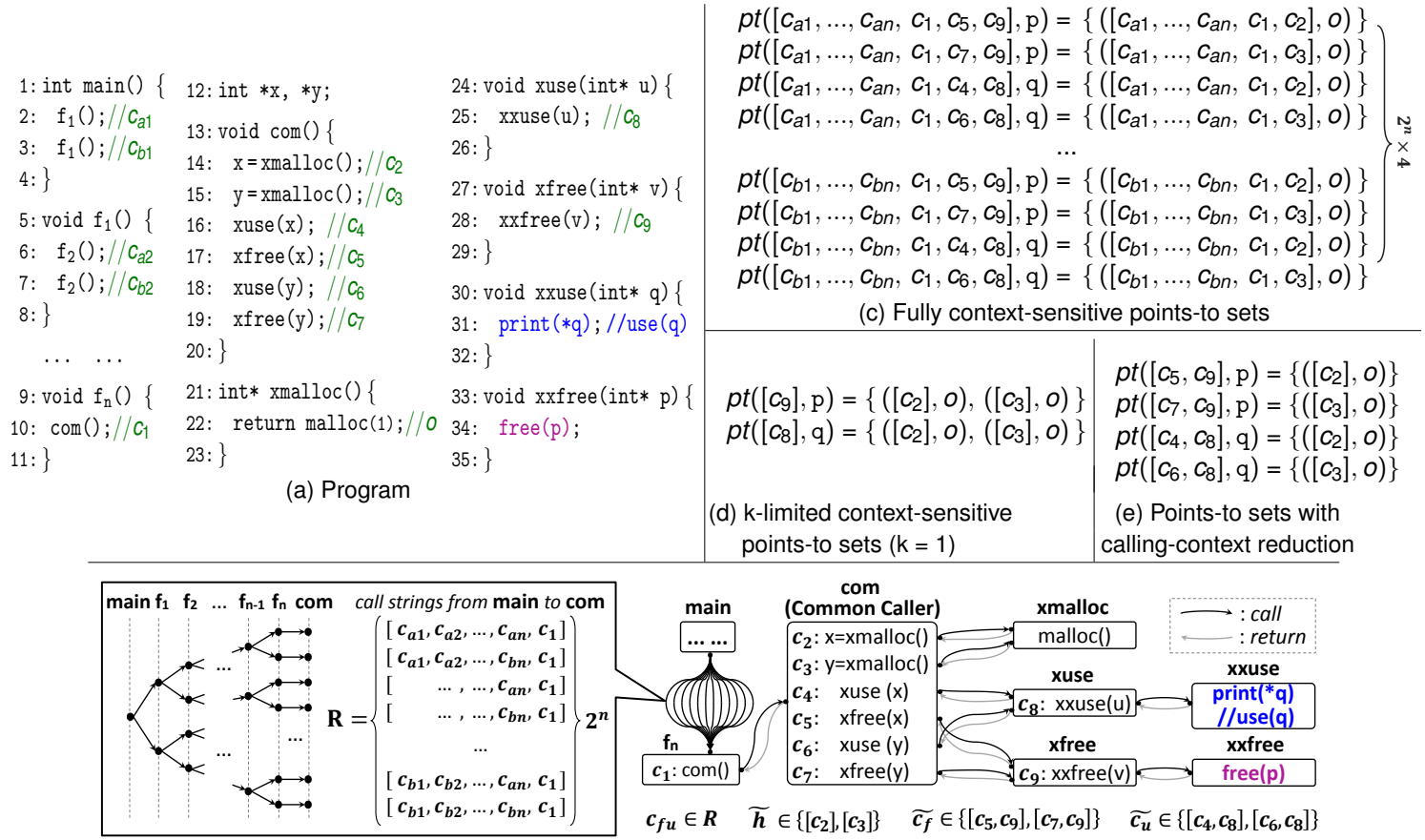


Figure: Context reduction, illustrated conceptually with an oracle fully-context-sensitive pointer analysis.

Detecting Use-After-Free Vulnerabilities (ICSE '18)

Spatio-temporal context reduction



(b) Interprocedural control flow graph (ICFG)

Figure: Calling-context reduction for overcoming the limitations of full and *k*-limited context-sensitivity in UAF detection.

Detecting Use-After-Free Vulnerabilities (ICSE '18)

Recall

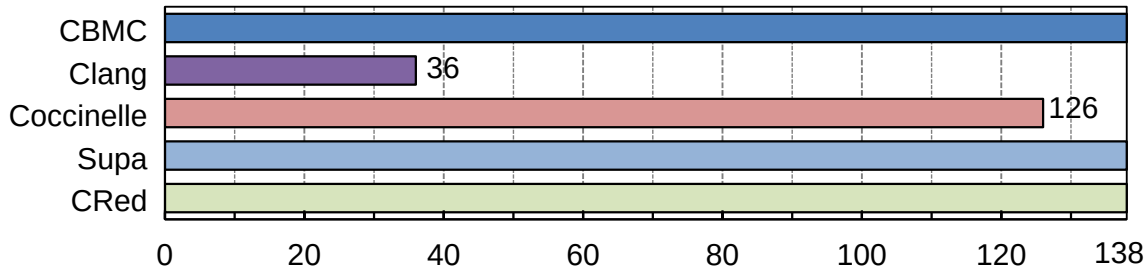


Figure: Hit rates for the 138 bugs in JTS: CBMC (100%), CLANG (26%), COCCINELLE (91%), SUPA (100%) and STC (100%).

Detecting Use-After-Free Vulnerabilities (ICSE '18)

CVE vulnerabilities found by our tool

Program	Known bugs		New bugs
	Identifier	Detected	#Detected
rtorrent	—	—	0
less	—	—	1
bitlbee	CVE-2016-10188	✓	0
nghttp2	CVE-2015-8659	✓	0
mupdf	BugID-694382	✓	0
h2o	CVE-2016-4817	✓	5
xserver	CVE-2013-4396	✓	0
php	CVE-2015-1351	✓	2

Detecting Use-After-Free Vulnerabilities (ICSE '18)

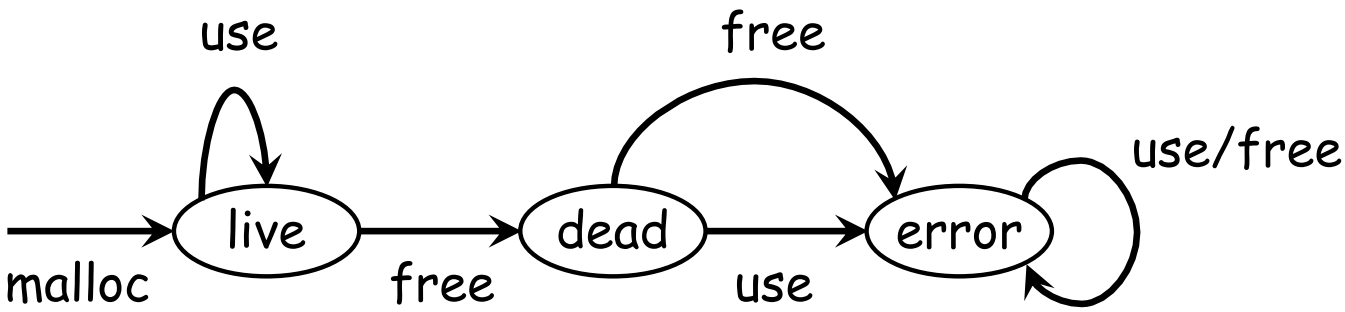
Comparing with other detectors

Table: #T:#True Positives (Bugs) and #F: #False Positives (i.e., False Alarms).

Program	CBMC			CLANG			COCCINELLE			SUPA			STC								
	Report		Time (secs)	Report		Time	Report		Time	Report		Time	#Warnings			Context Reduction		Report		Time
	#T	#F		#T	#F		#T	#F		#T	#F		Pre	CS	PS	Before	After	#T	#F	
bison	0	0	>259200	0	0	113	0	18	7	0	1044	1793	1640	352	1	7.3×10^{15}	2.0×10^{5}	0	1	1904
curl	0	0	>259200	0	0	355	0	8	53	0	694	27	699	82	0	3.3×10^{7}	8.2×10^{3}	0	0	668
ed	0	0	68553	0	0	18	0	0	1	0	34	1	34	32	2	6.3×10^{4}	3.6×10^{3}	0	2	4
grep	0	0	>259200	0	0	110	0	18	9	1	537	362	630	493	2	1.1×10^{7}	3.0×10^{5}	1	1	2023
ghostscript	0	0	>259200	0	0	2007	0	23	68	0	1944	2556	2630	1038	3	6.4×10^{15}	1.6×10^{5}	0	3	2805
gzip	0	0	>259200	1	0	68	0	12	3	1	381	3	382	117	1	7.1×10^{8}	3.6×10^{3}	1	0	4
phptrace	0	0	>259200	0	0	29	0	0	1	1	192	1	268	5	1	7.0×10^{5}	3.2×10^{3}	1	0	2
redis	0	0	>259200	0	2	836	0	5	7	16	4187	13333	11019	395	20	1.1×10^{15}	4.0×10^{3}	16	4	13551
sed	0	0	>259200	0	0	116	0	14	3	26	1887	160	2258	441	29	1.0×10^{9}	1.8×10^{5}	26	3	5102
zfs	0	0	>259200	0	0	790	0	5	30	40	12195	180	22283	2730	73	2.3×10^{14}	1.0×10^{6}	40	33	1271
Total	0	0	>2401353	1	2	4442	0	103	179	85	23095	18416	41843	5685	132	1.5×10^{16}	1.9×10^{6}	85	47	27334

Machine-Learning-Guided Type State Analysis For UAF Detection (ACSAC '17)

Automaton ? Temporal Property Specification



Value-Flow Analysis For Memory Leak Detection (ISSTA '12))

- Still too many false alarms by typestate analysis 19,000 in 8 programs (2 MLOC)
- UAF-related aliases/non-aliases
  - Alike
  - Predictable
  - Common characteristics
  - Classification using machine learning

Machine-Learning-Guided Type State Analysis For UAF Detection (ACSAC '17)

Observations

```
1: void foo(Apple* p) {
2:   free(p);
   }
   ...
3: void bar(Orange* q) {
4:   use(q);//Likely false UAF
   }
```

```
1: void foo(Apple* p) {
2:   free(p);
   }
   ...
3: void bar(Apple* q) {
4:   use(q);//Likely true UAF
   }
```

Machine-Learning-Guided Type State Analysis For UAF Detection (ACSAC '17)

Observations

```
1: void foo(Apple* p) {
2:   free(p);
   }
   ...
3: void bar(Apple* q) {
4:   use(q);//Likely false UAF
   }
```

Imprecise static over-approximation

$pt(p) = \{o_1, o_2, \ldots, \boldsymbol{o_{100}}\}$

$pt(q) = \{\boldsymbol{o_{100}}, o_{101}, \ldots, o_{200}\}$

```
1: void foo(Apple* p) {
2:   free(p);
   }
   ...
3: void bar(Apple* q) {
4:   use(q);//Likely true UAF
   }
```

Precise static over-approximation

$pt(p) = \{\boldsymbol{o_1}\}$

$pt(q) = \{\boldsymbol{o_1}\}$

Machine-Learning-Guided Type State Analysis For UAF Detection (ACSAC '17)

Observations

```
 1: void fun() {
 2:   ...
 3:   if (Cnd) {
 4:     free(p);
 5:     p = null;
 6:   }
 7:   ...
 8:   if (p != null) {
 9:     use(p);//Likely false UAF
10:   }
    }
```

```
 1: void fun() {
 2:   ...
 3:   if (Cnd) {
 4:     free(p);
 5:     //p = null;
 6:   }
 7:   ...
 8:   //if (p != null) {
 9:     use(p);//Likely true UAF
10:   }
    }
```

Machine-Learning-Guided Type State Analysis For UAF Detection (ACSAC '17)

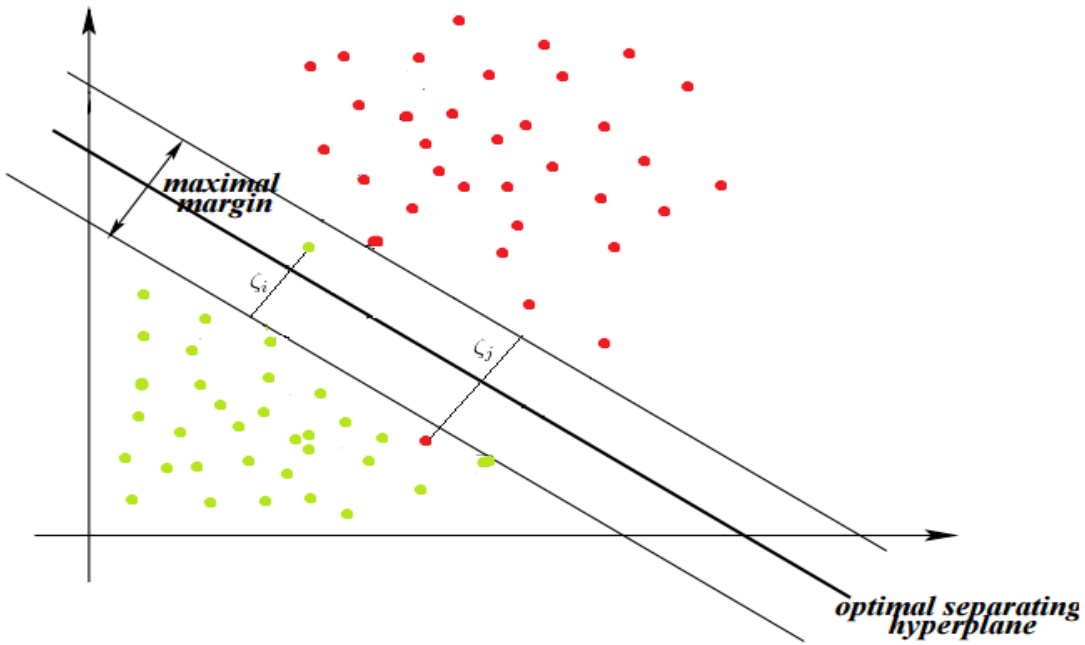
Observations

```
1: void fun() {
2:   ...
3:   for(...) {
4:     p = malloc(…);
5:     use(p); //Likely false UAF
6:     free(p);
     }
   }
```

```
1: void fun() {
2:   ...
3:   p = malloc(…);
4:   for(...) {
5:     use(p); //Likely true UAF
6:     free(p);
     }
   }
```

Machine-Learning-Guided Type State Analysis For UAF Detection (ACSAC '17)

Support Vector Macine



Figures shamelessly stolen from:
http://blog.hackerearth.com/simple-tutorial-svm-parameter-tuning-python-r

Machine-Learning-Guided Type State Analysis For UAF Detection (ACSAC '17)

Feature Engineering

- 35 Features in 4 groups
- Type information
  - E.g., array, struct, container, global, type compatibility
- Control flow
  - E.g., loop, recursion, distance, dominance, use before free
- Common characteristics
  - E.g., nullification, flags, reallocation, address comparison
- Classification using machine learning
  - E.g., size of points-to set, #UAF@free, #UAF@use, #aliases, points-to cycles

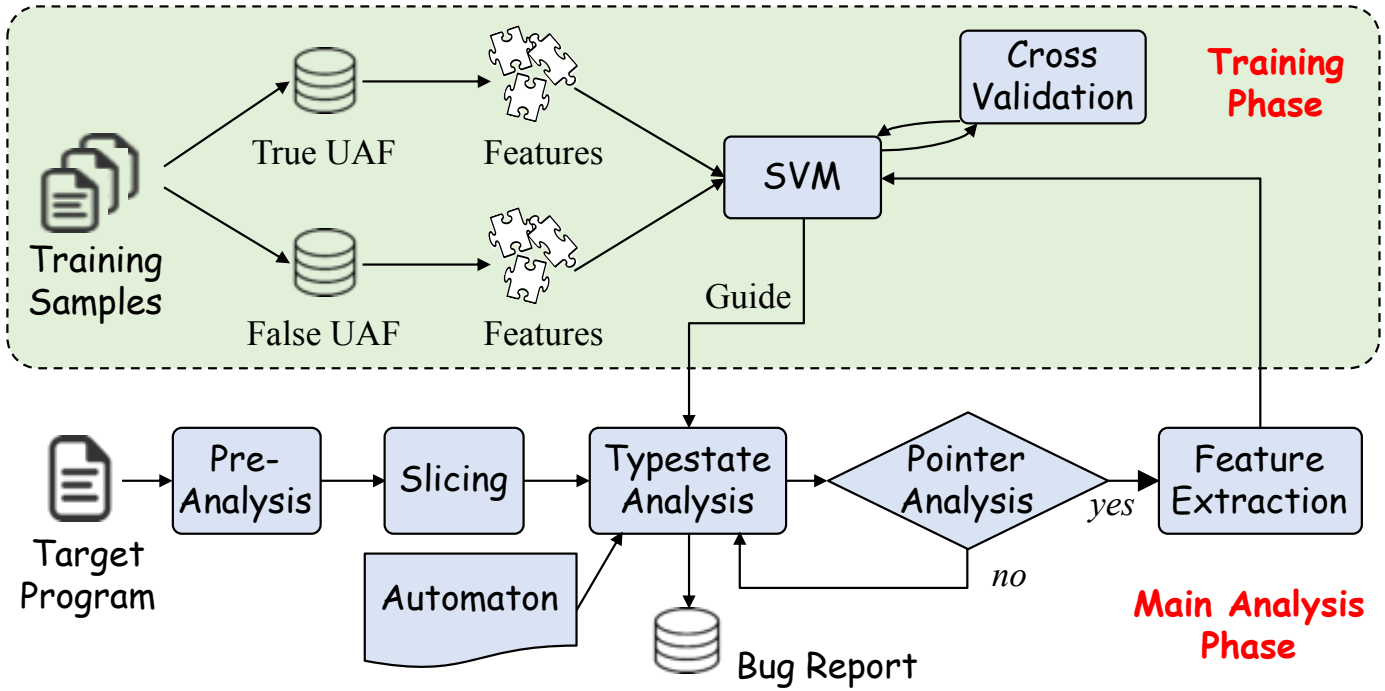
Machine-Learning-Guided Type State Analysis For UAF Detection (ACSAC '17)

Feature Engineering

Group	ID	Feature	Type	Description
Type Information	1	Array	Boolean	o is an array or an element of an array
	2	Struct	Boolean	o is a struct or an element of a struct
	3	Container	Boolean	o is a container (e.g., vector or map) or an element of a container
	4	IsLoad	Boolean	$use(q)$ is a load instruction
	5	IsStore	Boolean	$use(q)$ is a store instruction
	6	IsExtCall	Boolean	$use(q)$ is an external call
	7	GlobalFree	Boolean	$free(p)$, where p is a global pointer
	8	GlobalUse	Boolean	$use(q)$, where q is a global pointer
	9	CompatibleType	Boolean	p and q are type-compatible at $free(p)$ and $use(q)$
Control Flow	10	InSameLoop	Boolean	$free(p)$ and $use(q)$ are in the same loop
	11	InSameRecursion	Boolean	$free(p)$ and $use(q)$ are in the same recursion cycle
	12	#FunctionInBetween	Integer	number of functions in the shortest path from $free(p)$ to $use(q)$ in the program's call graph
	13	DiffIteration	Boolean	$use(q)$ appears after $free(p)$ via a loop back-edge
	14	Dom	Boolean	$free(p)$ dominates $use(q)$
	15	PostDom	Boolean	$use(q)$ post-dominates $free(p)$
	16	#IndCalls	Integer	number of indirect calls in the shortest path from $free(p)$ to $use(q)$ in the program's call graph
	17	UseBeforeFree	Boolean	a UAF pair, $free(p)$ and $use(q)$, is also a use-before-free
Common Programming Practices	18	NullifyAfterFree	Boolean	p is set to null immediately after $free(p)$
	19	ReturnConstInt	Boolean	a const integer is returned after $free(p)$
	20	ReturnBoolean	Boolean	a Boolean value is returned after $free(p)$
	21	Casting	Boolean	pointer casting is applied to q at $use(q)$
	22	ReAllocAfterFree	Boolean	p is redefined to point to a newly allocated object immediately after $free(p)$
	23	RefCounting	Boolean	o is an reference-counted object
	24	ValidatedFreePtr	Boolean	null checking for p before $free(p)$
	25	ValidatedUsePtr	Boolean	null checking for q before $use(q)$
Points-to Information	26	SizeOfPointsToSetAtFree	Integer	number of objects pointed to by p at $free(p)$
	27	SizeOfPointsToSetAtUse	Integer	number of objects pointed to by q at $use(q)$
	28	#UAFSharingSameFree	Integer	number of UAF pairs sharing the same $free(p)$
	29	#UAFSharingSameUse	Integer	number of UAF pairs sharing the same $use(q)$
	30	#Aliases	Integer	number of pointers pointing to o
	31	AllocInLoop	Boolean	o is allocated in a loop
	32	AllocInRecursion	Boolean	o is allocated in recursion
	33	LinkedList	Boolean	o is in a points-to cycle (signifying its presence in a linked-list)
	34	SamePointer	Boolean	p and q at $free(p)$ and $use(q)$ are the same pointer variable
	35	DefinedBeforeFree	Boolean	q at $use(q)$ is defined before $free(p)$

Machine-Learning-Guided Type State Analysis For UAF Detection (ACSAC '17)

Framework



Machine-Learning-Guided Type State Analysis For UAF Detection (ACSAC '17)

Results

Program	#Cand	#W^{NML}	R1	#W$^{\text{Tac}}$	R2	Time (s)	#True	FPR	TPR
rtorrent	803	229	71.5%	0	100.0%	90	0	—	—
less	4,628	790	82.9%	3	99.6%	316	1	66.7%	33.3%
bitlbee	529	113	78.6%	16	85.8%	151	9	43.8%	56.3%
nghttp2	975	210	78.5%	16	92.4%	83	7	56.3%	43.8%
mupdf	21,701	1,658	92.4%	50	97.0%	197	19	62.0%	38.0%
h2o	18,143	3,559	80.4%	23	99.4%	6,205	9	60.9%	39.1%
xserver	53,258	6,706	87.4%	102	98.5%	2,053	40	60.8%	39.2%
php	26,306	5,818	77.9%	56	99.0%	5,942	24	57.1%	42.9%
Total	126,343	19,083	—	266	—	15,037	109	Avg. 58.2%	Avg. 41.8%

$\#Cand$: Number of candidate UAF pair by pre-analysis

$\#W^{NML}$: Number of warnings by Tac without machine learning

$\#W^{TAC}$: Number of warnings by Tac

FPR: False positive rate

TPR: True positive rate

$$R1 = \frac{\#Cand - \#W^{NML}}{\#Cand}$$

$$R2 = \frac{\#W^{NML} - \#W^{TAC}}{\#W^{NML}}$$

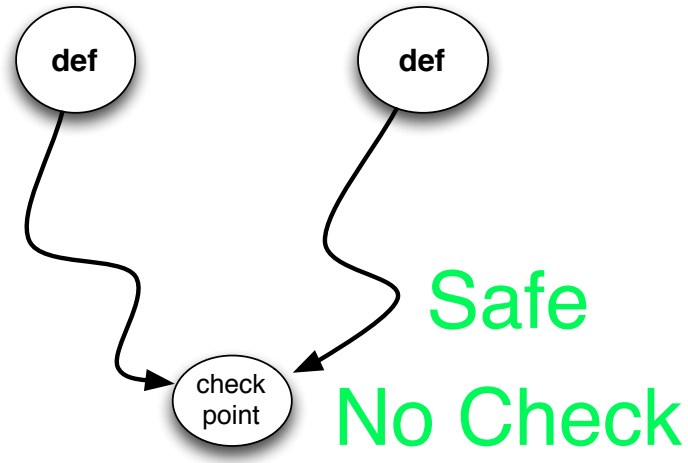
Outline

- **Existing software bugs and vulnerabilities**
- **Static analysis and dynamic analysis**
- **Technical contributions in developing scalable and precise program analysis**
  - Fundamental program analysis
    - Sparse value-flow analysis
    - Selective and on-demand value-flow analysis
  - Applications
    - Static value-flow analysis for detecting memory errors
    - Hybrid value-flow analysis to enforce memory safety
- **Research opportunities**

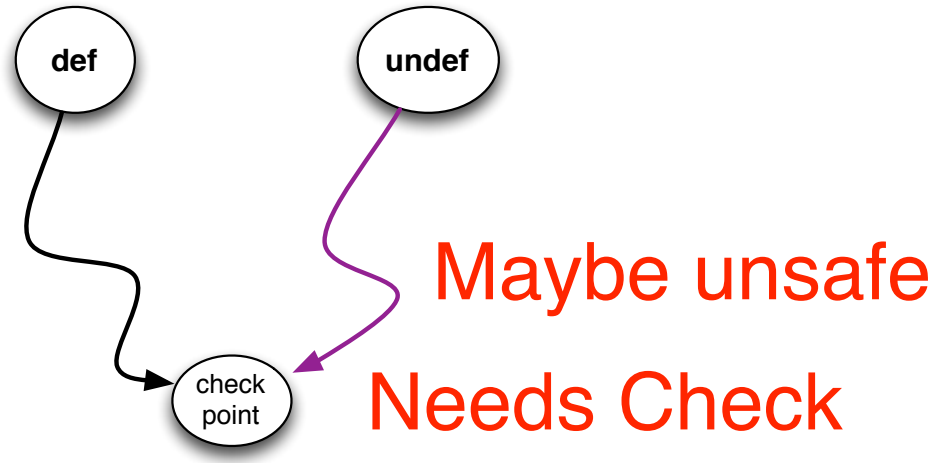
Hybrid Analysis to Enforce Memory Safety

- Static and dynamic analysis techniques have their own strengths and weaknesses
- Complementary to each other
  - Reduce dynamic analysis overhead using static analysis
  - Guide dynamic analysis to analyze important parts of a program

Hybrid Analysis For Uninitialized Variable Detection (CGO '14)



There is no value-flow reachable from an undefined allocation sites

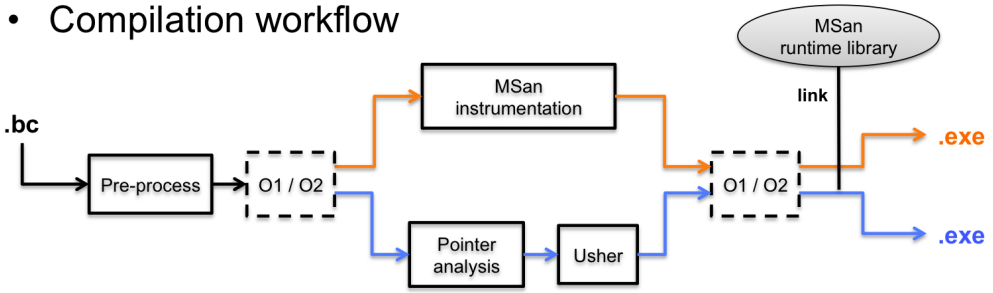


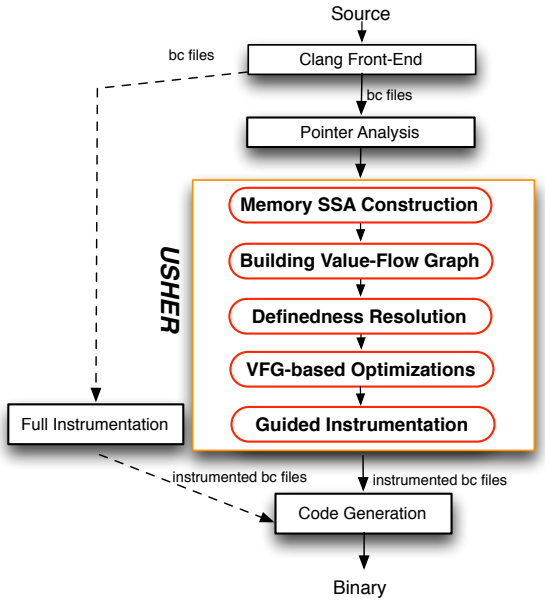
Reachable from an undefined allocation sites along at least one value-flow path

Ding Ye, Yulei Sui, Jingling Xue. Accelerating Dynamic Detection of Uses of Undefined Values with Static Value-Flow Analysis
*12th Annual IEEE/ACM International Symposium on Code Generation and Optimization (**CGO '14**)*

Hybrid Analysis For Uninitialized Variable Detection (CGO '14)

- Compilation workflow





Our approach successfully reduces the average overhead of Google's Memory Sanitizer from 302% to 123%

Ding Ye, Yulei Sui, Jingling Xue. Accelerating Dynamic Detection of Uses of Undefined Values with Static Value-Flow Analysis *12th Annual IEEE/ACM International Symposium on Code Generation and Optimization (**CGO '14**)*

Hybrid Analysis For Buffer Overflow Detection (ISSRE '14)

```
int *a = ..., *b = ...;

for (i = 1; i <= N; i++) {
  a[i - 1] = ...
  if (...)  {
     a[i] = b[i] + ...;
  }
}
```

Load / Store	Memory access range
a[i-1]	[a, a+4*N)
a[i]	[a+4, a+4*(N+1))
b[i]	[b+4, b+4*(N+1))

Ding Ye, Yu Su, Yulei Sui, Jingling Xue. WPBound: Enforcing Spatial Memory Safety Efficiently at Runtime with Weakest Preconditions *25th IEEE International Symposium on Software Reliability Engineering (**ISSRE '14**)*

Yulei Sui, Ding Ye, Yu Su, Jingling Xue. Eliminating Redundant Bounds Checks in Dynamic Buffer Overflow Detection Using Using Weakest Preconditions. *IEEE Transactions on Reliability (**TR '16**)*

Hybrid Analysis For Buffer Overflow Detection (ISSRE '14)

Weakest preconditions are usually hard to compute

```
int *p = ...;
wp = wpCheck(p, 4*N, p_base, p_bound);
for (i = 0; i < N; i++) {
  if (...)  {
    if (wp)  sCheck(&p[i], 4, p_base, p_bound);
    p[i] = ...;
  }
}
```

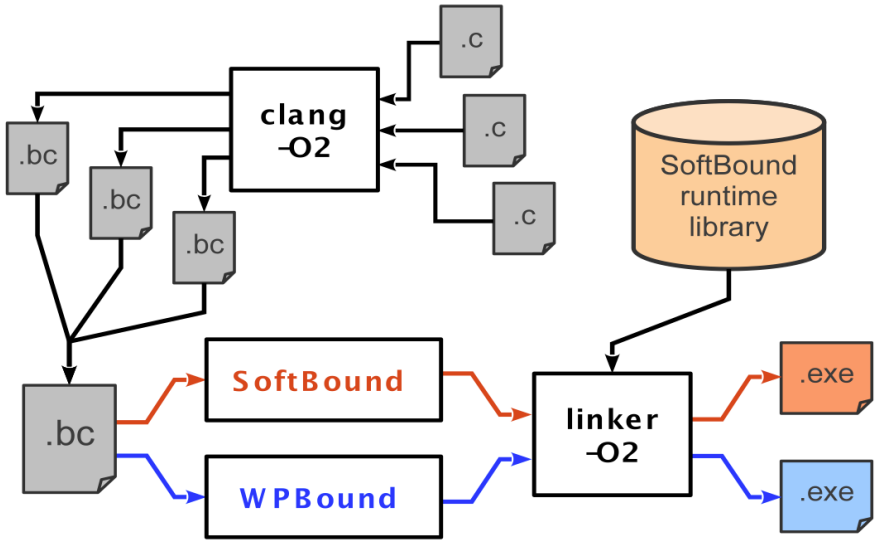
Use conservatively approximated WPs! [true, WP]

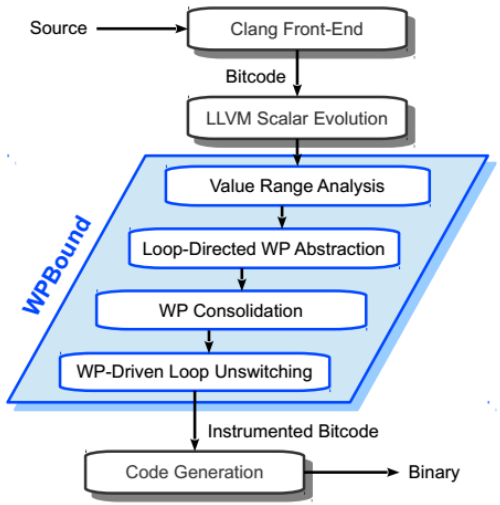
```
inline wpCheck(ptr, sz, base, bound) {
  return (ptr < base || ptr + sz >= bound);
}
```

Ding Ye, Yu Su, Yulei Sui, Jingling Xue. WPBound: Enforcing Spatial Memory Safety Efficiently at Runtime with Weakest Preconditions *25th IEEE International Symposium on Software Reliability Engineering (**ISSRE '14**)*

Yulei Sui, Ding Ye, Yu Su, Jingling Xue. Eliminating Redundant Bounds Checks in Dynamic Buffer Overflow Detection Using Using Weakest Preconditions. *IEEE Transactions on Reliability (**TR '16**)*

Hybrid Analysis For Buffer Overflow Detection (ISSRE '14)



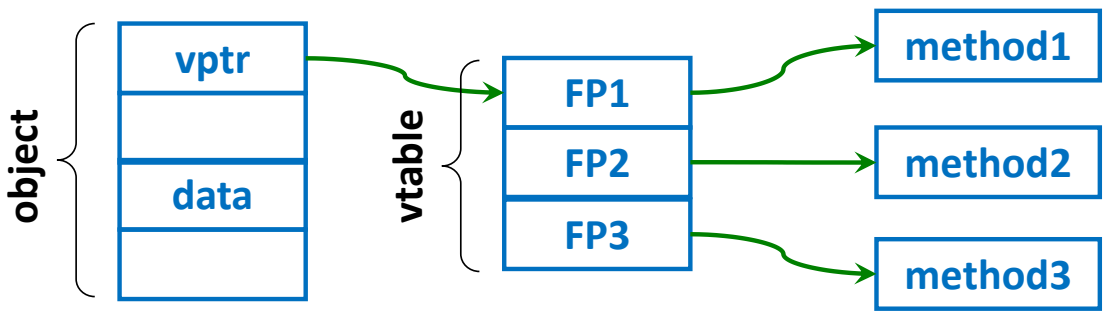


Our approach successfully reduces the average overhead of SOFTBOUND from 71% to 45%

Ding Ye, Yu Su, Yulei Sui, Jingling Xue. WPBound: Enforcing Spatial Memory Safety Efficiently at Runtime with Weakest Preconditions *25th IEEE International Symposium on Software Reliability Engineering (**ISSRE '14**)*

Yulei Sui, Ding Ye, Yu Su, Jingling Xue. Eliminating Redundant Bounds Checks in Dynamic Buffer Overflow Detection Using Using Weakest Preconditions. *IEEE Transactions on Reliability (**TR '16**)*

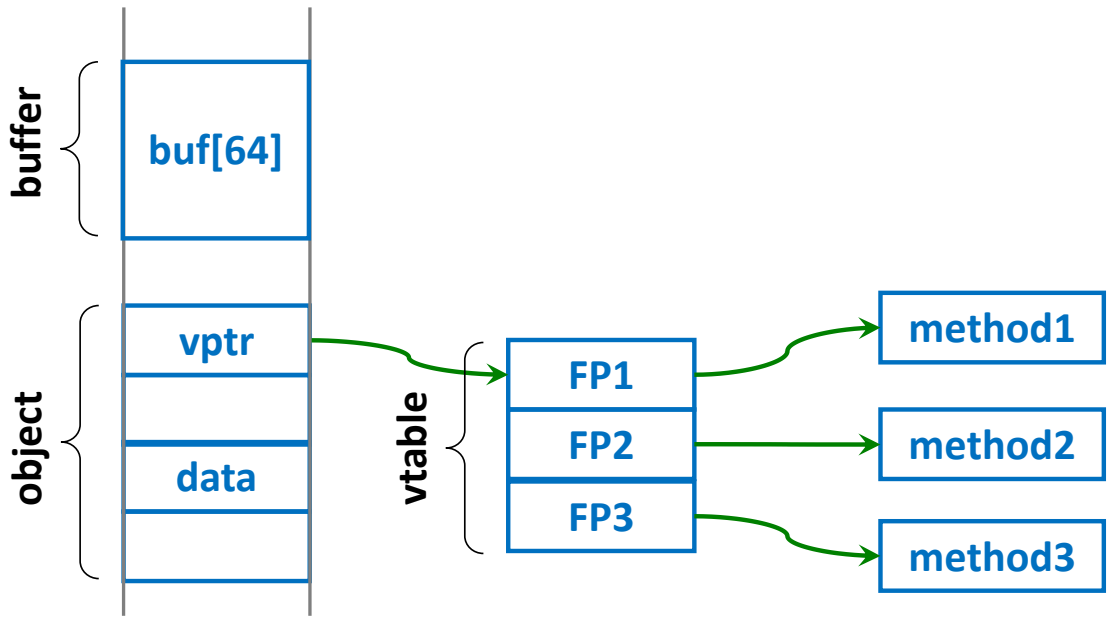
Compiler generated function pointers
(e.g C++ code)



Virtual Call Protection Against VTable Hijacking (ISSTA '17)

VTable Hijacking Attacks via Memory Corruption Bugs

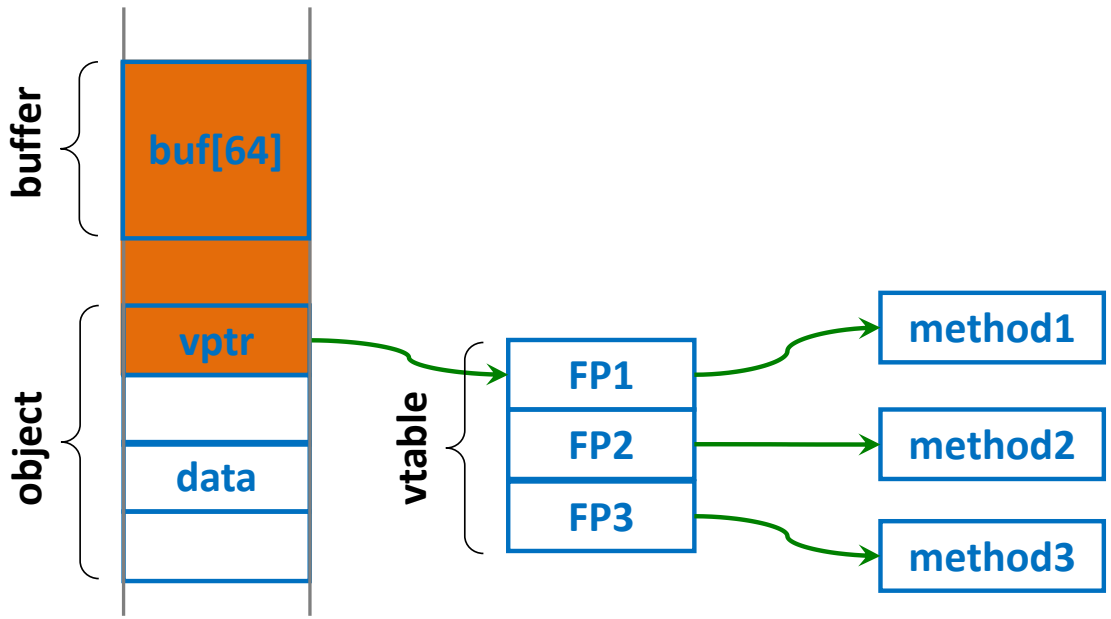
Control flow hijacking via *buffer overflow*



Virtual Call Protection Against VTable Hijacking (ISSTA '17)

VTable Hijacking Attacks via Memory Corruption Bugs

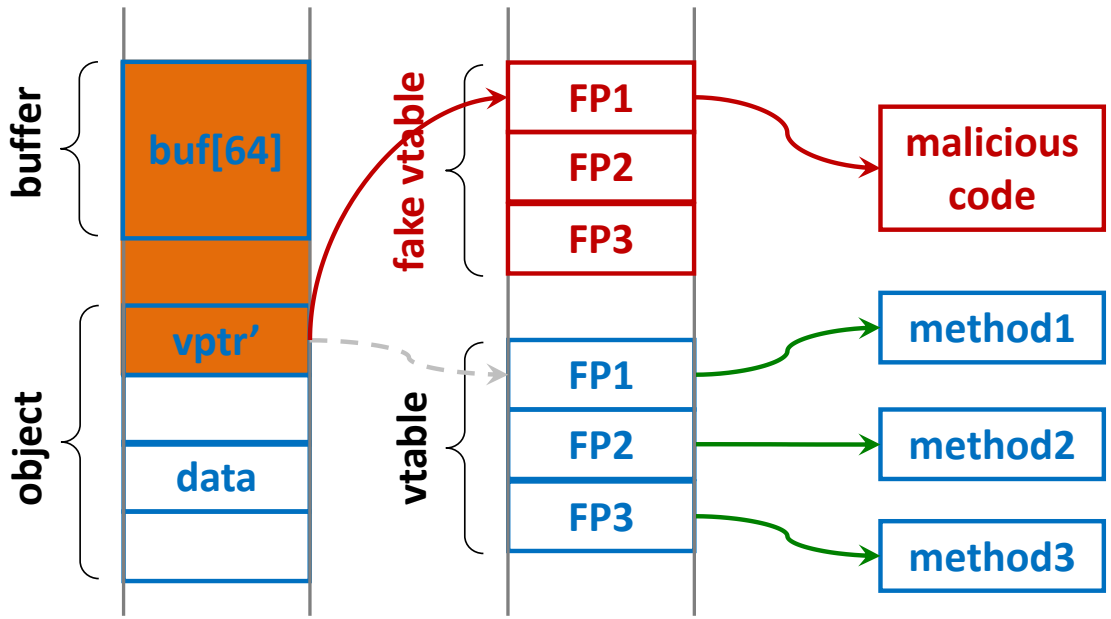
Control flow hijacking via *buffer overflow*



Virtual Call Protection Against VTable Hijacking (ISSTA '17)

VTable Hijacking Attacks via Memory Corruption Bugs

Control flow hijacking via *buffer overflow*

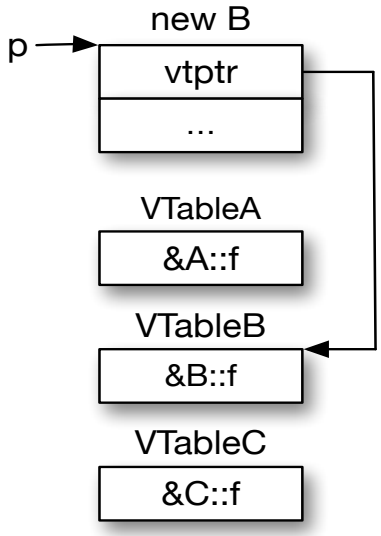


Virtual Call Protection Against VTable Hijacking (ISSTA '17)

An example

```
1  class A {
2    public: virtual void f() {...}
3  };
4  class B: public A {
5    public: virtual void f() {...}
6  };
7  class C {
8    public: virtual void f() {...}
9  };
10 ...
11 int main() {
12   A *p = new B;
13   p->f();
14 }
```

(a) Source code



(b) Object layout

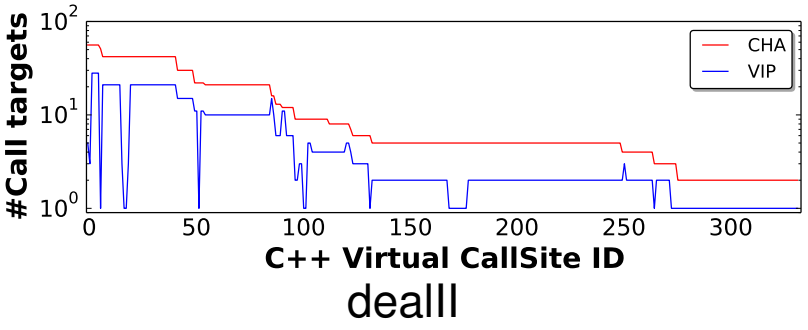
```
vtptr = *p;       //load vtable
vfn = &vtptr[0]; //the 1st vtable entry address
fp = *vfn;        //load virtual function
assert(fp==&A::f || fp==&B::f);
fp(p);           //call virtual function
```

(c) Pseudo LLVM instructions for virtual call `p->f()`

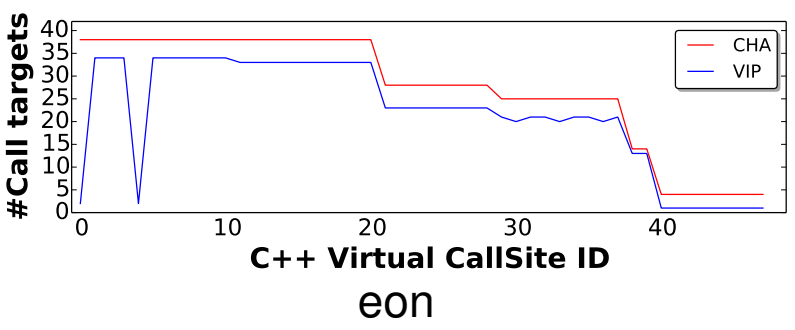
Figure: Imprecision of the CHA-based approach, which excludes `C::f`, but includes a spurious function `A::f`.

Virtual Call Protection Against VTable Hijacking (ISSTA '17)

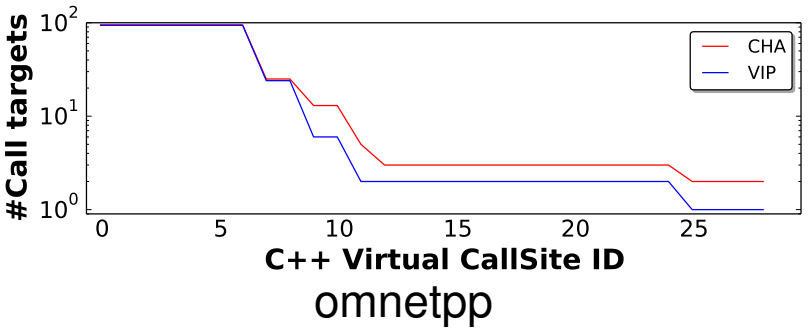
Precision Comparison



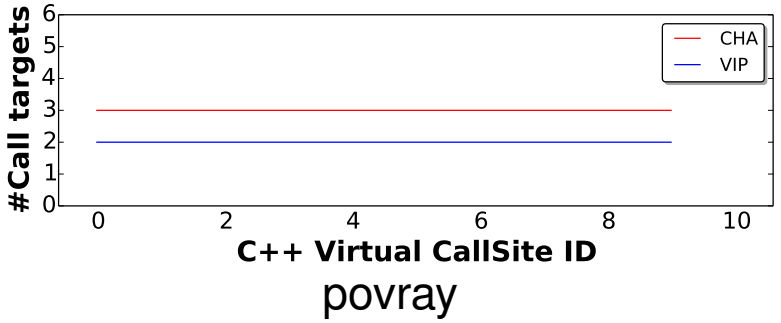
dealII



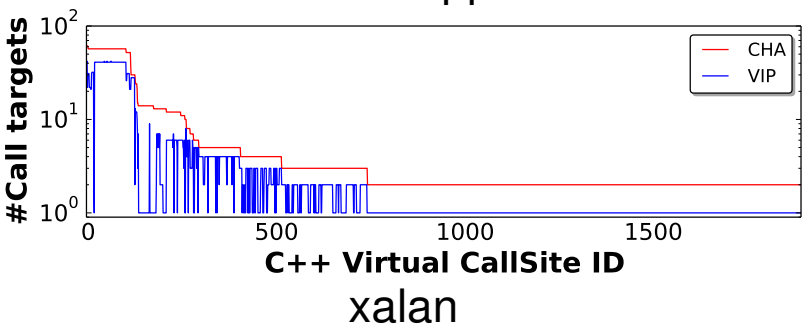
eon



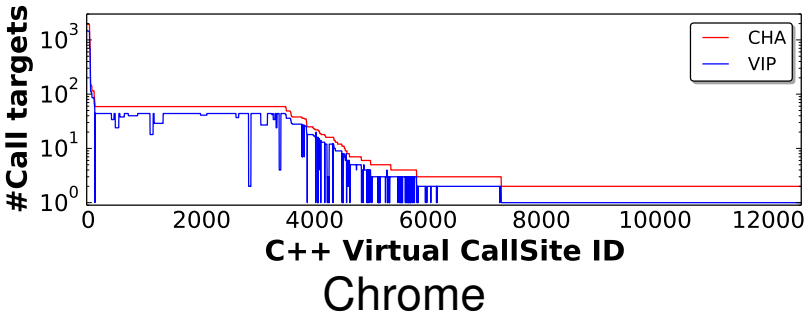
omnetpp



povray



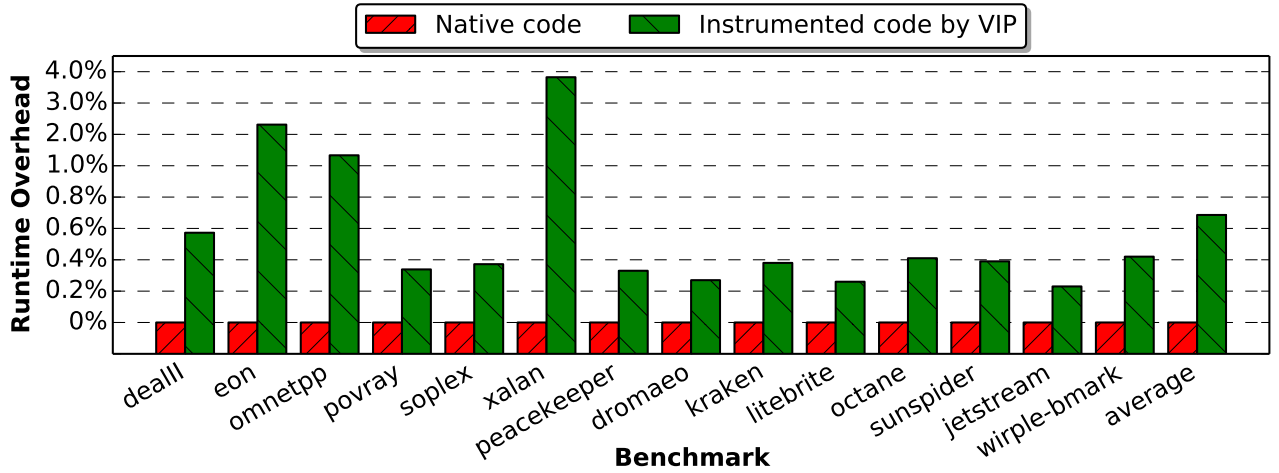
xalan

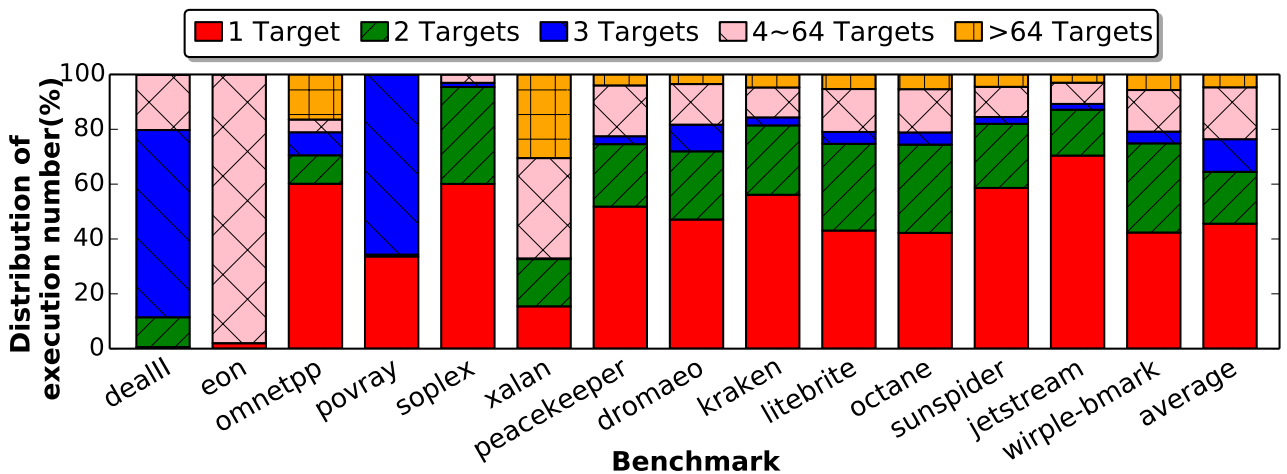


Chrome

Virtual Call Protection Against VTable Hijacking (ISSTA '17)

Runtime Overhead





What's More? Research Opportunities

- **Mobile Security**: Detecting information leakage in the mobile apps (e.g., Android, iOS) through static and dynamic analysis.
- **Program Analysis for Incomplete code**: Analysis for programs in the presence of incomplete code using NLP.
- **Big Data Security**: Apply program analysis techniques to track information flows in big data applications (e.g., hadoop and spark).
- **Program Analysis for Smart Contract Programs**: Apply program analysis techniques to find security vulnerabilities in smart contract programs (e.g., written in solidity language) in Block Chain Tools (e.g., Ethereum).

Thanks!

Q & A