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

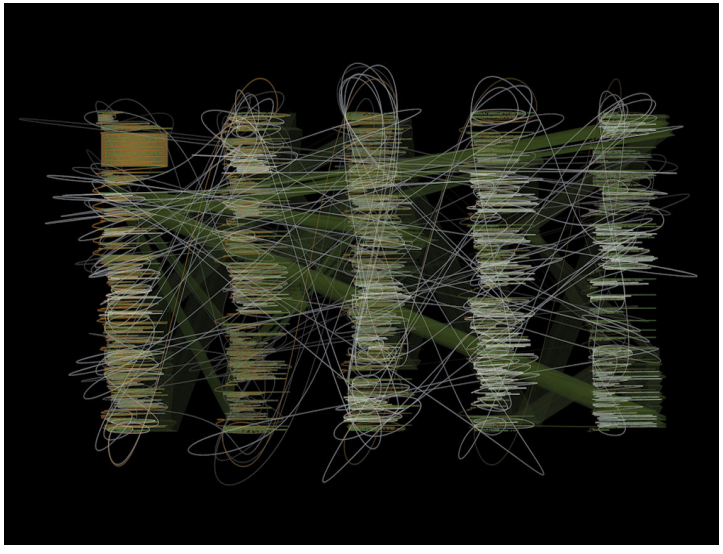
February 5, 2019

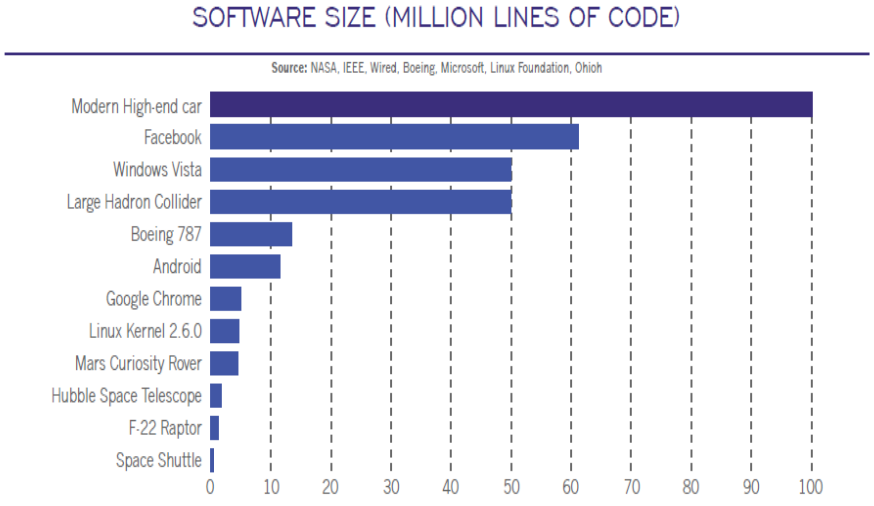
Outline

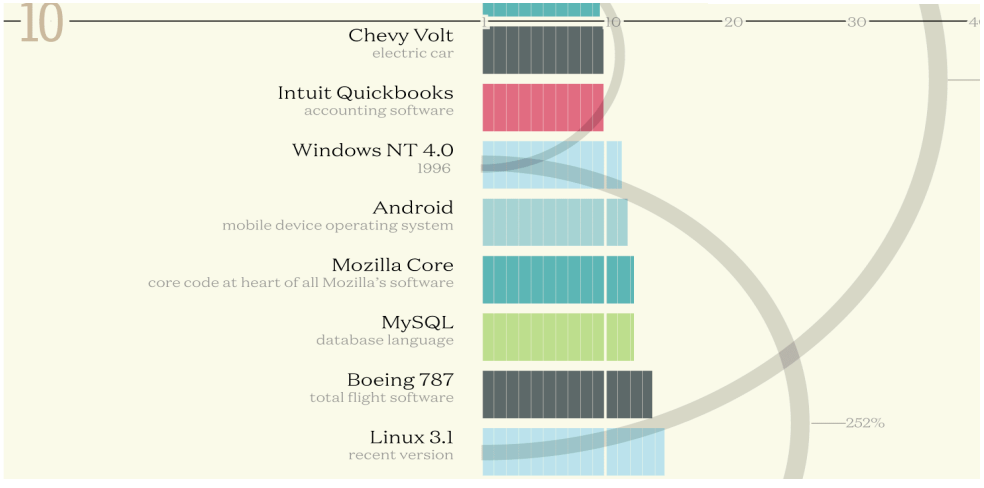
- **Existing software bugs and vulnerabilities**
- **Static analysis and dynamic analysis**
- **Technical contributions in developing scalable and precise program analysis**
  - Fundamental program analysis
    - Sparse value-flow analysis
    - On-demand value-flow analysis
    - Value-flow analysis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

Modern System Software

– Extremely Large and Complex







Software Becomes More Buggy

Software Becomes More Buggy

- Memory Leaks
- Buffer Overflows
- Null Pointers
- Use-After-Frees
- Data-races

Memory Leak

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

```
1  /* CVE-2012-0817 allows remote attackers to cause a denial of service through adversarial connection requests.*/
2  /* Samba --libads/ldap.c:ads_leave_realm */.
3
4  host = memAlloc(hostname);
5   ...
6   if (...)   {...;  return ADS_ERROR_SYSTEM(ENOENT);} // The programmer forgot to release host on error.
7
```

```
1  /* A memory leak in Php-5.5.11 */
2
3  for (...)  {
4       char* buf = readBuffer();
5       if (condition)
6          printf (buf);
7       else
8          continue;   // buf is leaked in else branch
9       freeBuf(buf);
10 }
11
```

Buffer Overflow

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

```
1  /* A simplified example from "Young and Mchugh, IEEE S&P 1987", exploited by attackers to bypass verfication*/
2
3    void verfiyPassword(){
4      char buff[15]; int pass = 0;
5       printf ("\n Enter the password : \n");
6      gets(buff);
7
8       if (strcmp(buff, "thegeekstuff")){     // return non-zero if the two strings do not match
9            printf ("\n Wrong Password \n");
10     }
11     else{                                   // return zero if two strings matched or a buffer overrun
12           printf ("\n Correct Password \n");
13          pass = 1;
14     }
15      if (pass)
16           printf ("\n Root privileges given to the user \n");
17   }
18
```

[1] Heartbleed, a well-known vulnerability in OpenSSL is also caused by buffer overflow (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 denial of service via memory corruption

```
1  /* An uninitialized variable vulnerability simplified from gnuplot, CVE-2017-9670 */
2
3  void load(){
4          switch (ctl) {
5                  case -1:
6                          xN = 0; yN = 0;
7                          break;
8                  case 0:
9                          xN = i; yN = -i;
10                         break;
11                 case 1:
12                         xN = i + NEXT_SZ; yN = i - NEXT_SZ;
13                         break;
14                 default:
15                         xN = -1; xN = -1; // xN is accidentally set twice while yN is uninitialized
16                         break;
17         }
18                 plot(xN, yN);
19 }
20
21
```

Use-After-Free

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

```
1  /* CVE-2015-6125 and CVE-2018-12377 with similar heap use after free patterns*/
2
3  char* msg = memAlloc(...);
4  ...
5  if (err) {
6          abrt = 1;
7           ....
8          free(msg);
9  }
10 ...
11 if (abrt) {
12          ...
13         logError("operation aborted before commit", msg); // try to access released heap variable
14 }
```

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
    - On-demand value-flow analysis
    - Value-flow analysis for multithreaded programs
  - Applications
    - Static value-flow analysis for detecting memory errors
    - Hybrid value-flow analysis to enforce memory safety
- **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 problems at source code level.
  - + Catch bugs at early the stage of the software development cycle.
  - - False alarms due to over-approximation.
  - - Precise analysis has scalability issue for analyzing large size programs.

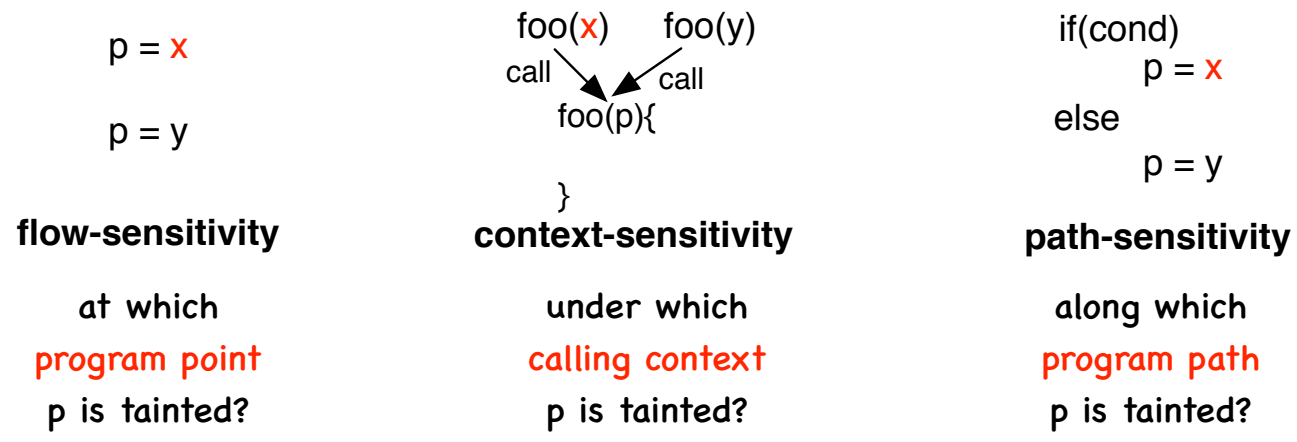
Static Analysis vs. Dynamic Analysis

Static Analysis

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

Levels of Abstractions

Assume x is a tainted value



Static Analysis vs. Dynamic Analysis

Dynamic Analysis

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

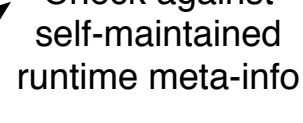
Static Analysis vs. Dynamic Analysis

Dynamic Analysis

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

Instrumentations

p = x[i]
Observe_and_check (&x, i)

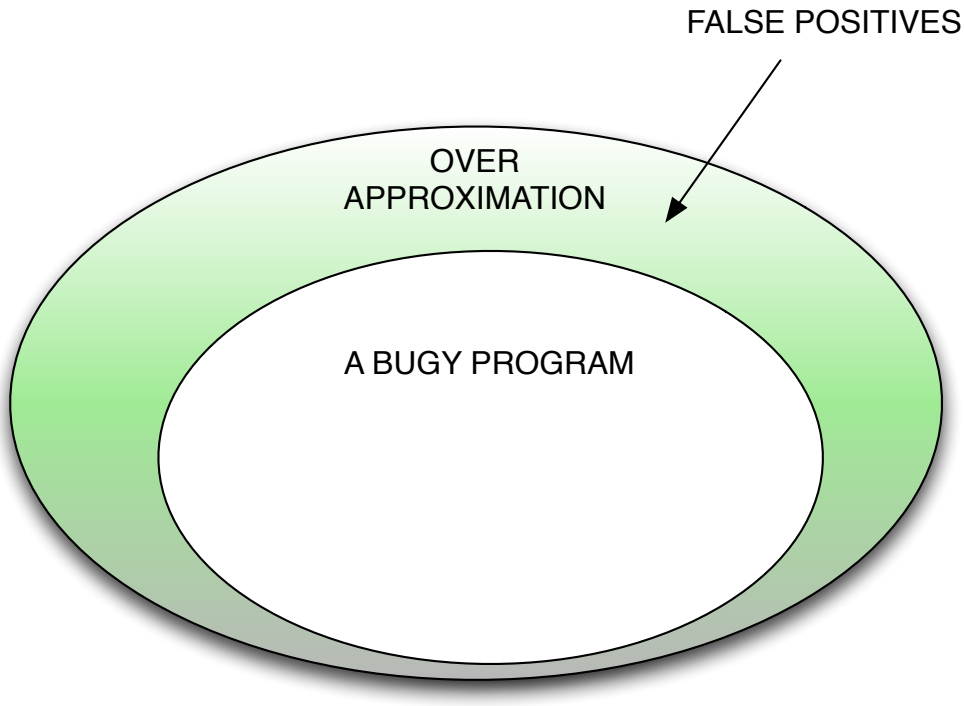


Check against self-maintained runtime meta-info

Limited Coverage

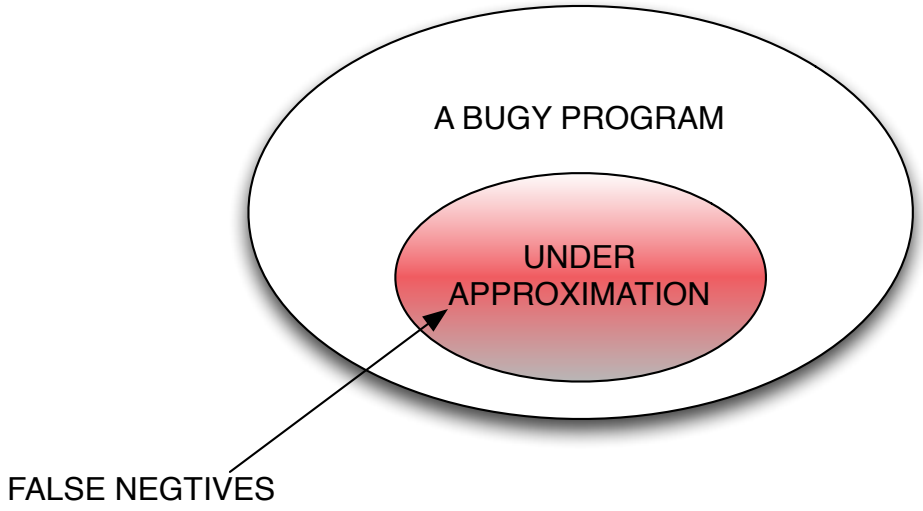
```
if(hard_to_satisfy)
        x=*p          // a null dereference
else
        x= *q         // a safe dereference
```

Bug Detection Philosophy



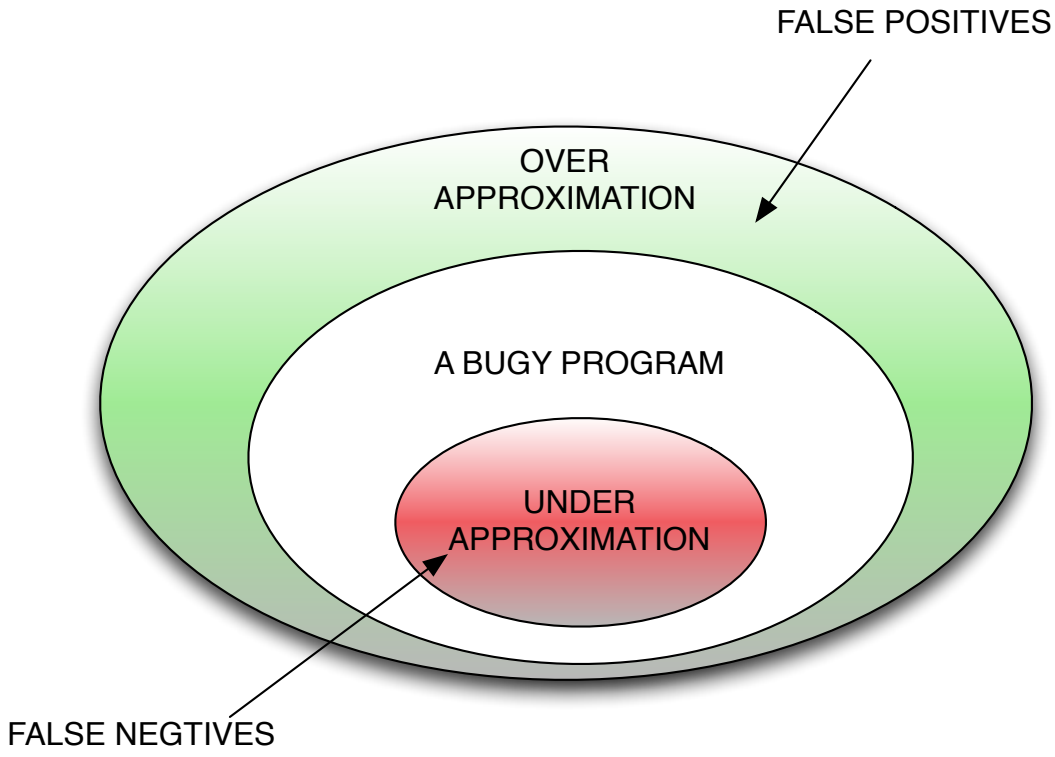
- Soundness : Over-Approximation (Static Analysis)
- Completeness : Under-Approximation (Dynamic Analysis)

Bug Detection Philosophy



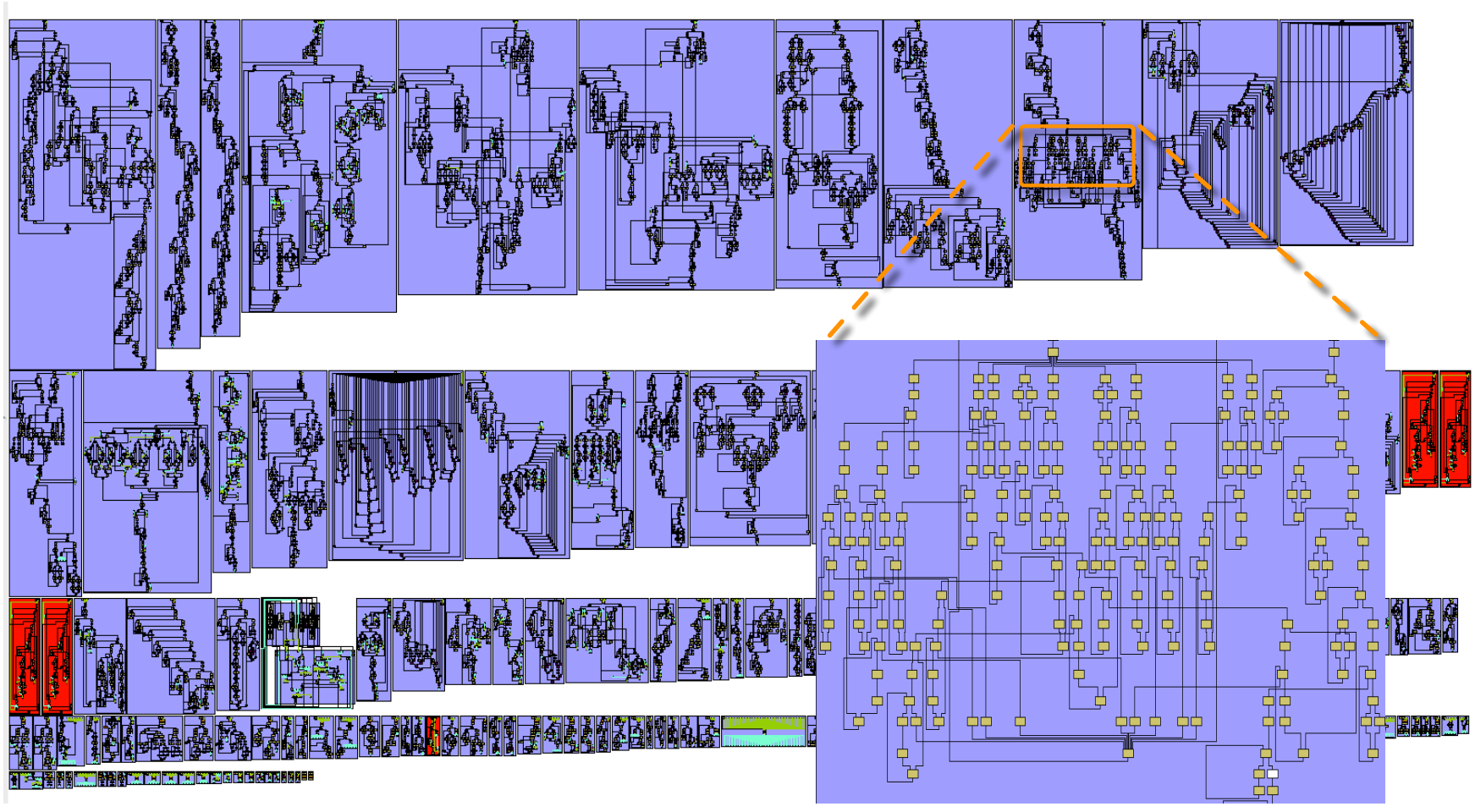
- Soundness : Over-Approximation (Static Analysis)
- Completeness : Under-Approximation (Dynamic Analysis)

Bug Detection Philosophy



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

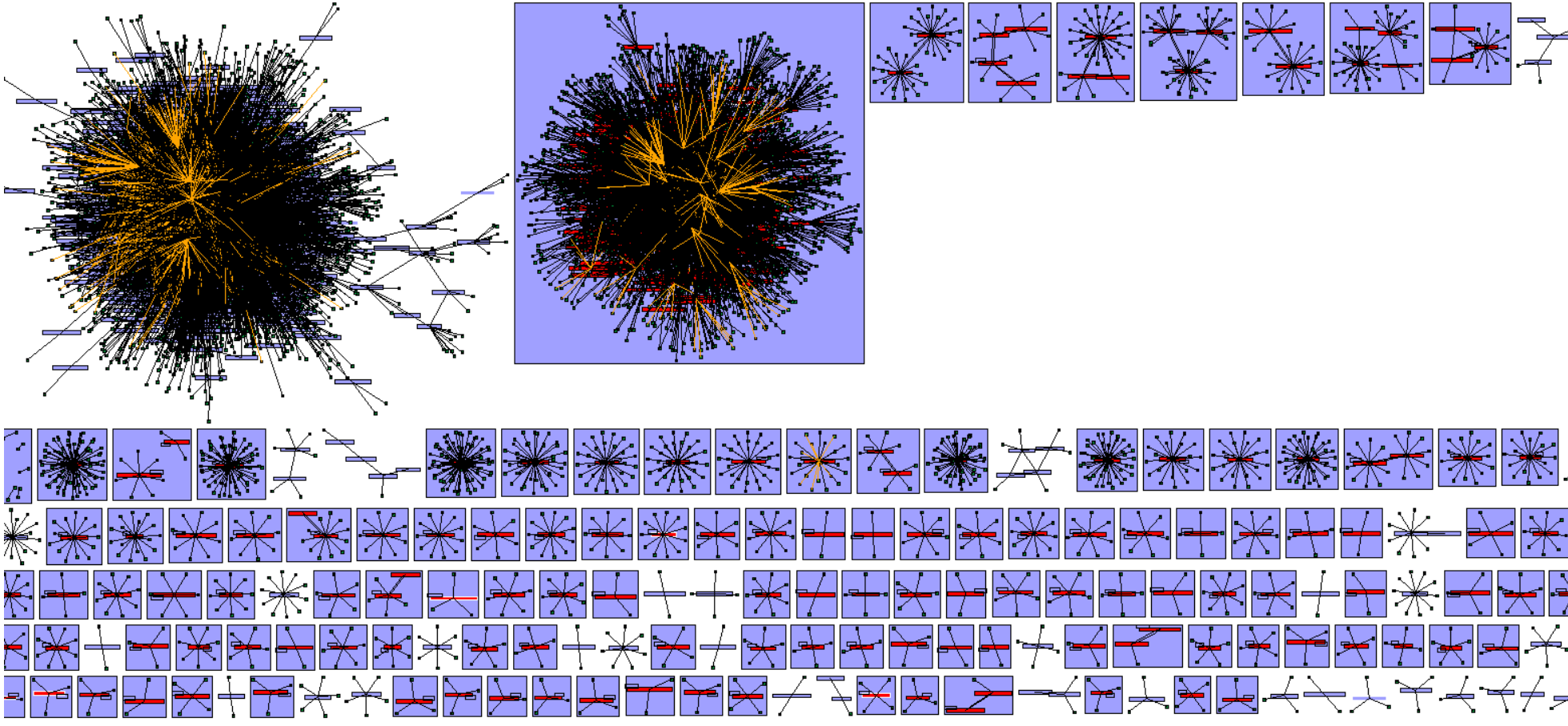
Whole-Program CFG of 300.twolf (20.5KLOC)



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

Costly to reason about flow of values on CFGs!

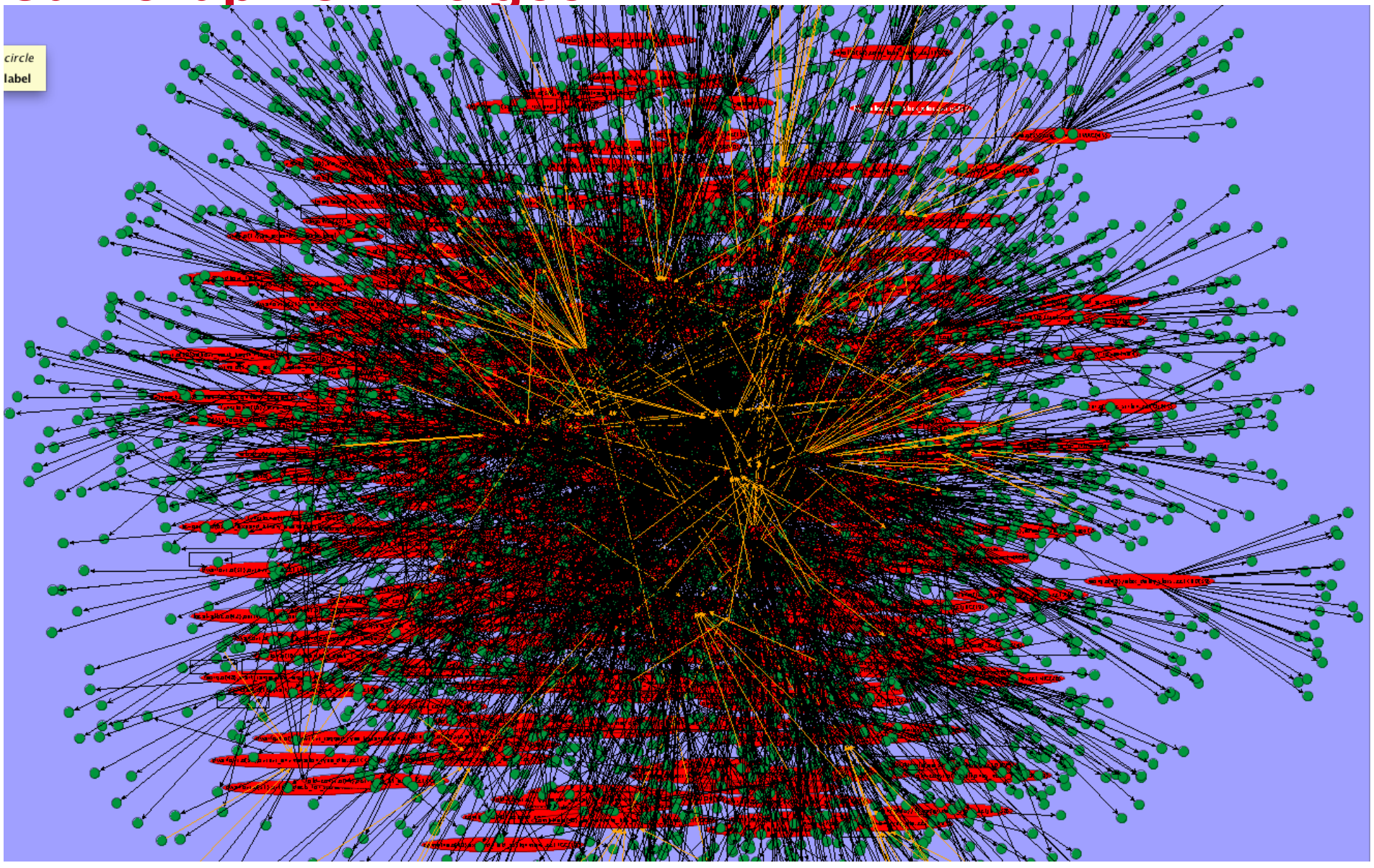
Call Graph of 176.gcc (230.5KLOC)



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

Costly to reason about flow of values on CFGs!

Call Graph of 176.gcc



My Past Research

Aims to develop practical **static and dynamic analysis** techniques that can efficiently and precisely **understand, detect and fix** bugs in the context of large-scale software with millions of lines of code.

- Fundamental Program Analyses
  - *Sparse value-flow analysis* (CC'16, FSE '16, ICSE '18, TSE '18)
  - *On-demand pointer analysis* (CGO '13, LCTES '16, TECS '18)
  - *Refinement-based analysis* (ECOOP '14, CGO '16, ICSE '18)
- Client Application: Software Bug Detection and Repair
  - *Memory leaks* (ISSTA '12, TSE '14)
  - *Buffer overflows* (ISSRE '14, IEEE Transaction on Reliability '15)
  - *Uninitialized variables* (CGO '14, FSE '16)
  - *Use-after-frees* (ACSAC '17, ICSE '18)
  - *Concurrency bugs* (CGO '16, ICST '19)
  - *Control-flow integrity protection* (ISSTA '17, ACISP '18)
  - *Program Repair* (SAC'16, ICSE '19)

SVF : Static Value-Flow Analysis

A sparse, on-demand, interprocedural program dependence analysis framework for both sequential and multithreaded programs.

- The SVF project
  - Started since early 2014, actively maintained. **Publicly available** at : `http://svf-tools.github.io/SVF`.
  - Implemented on top of LLVM compiler (the latest version 7.0.0) with over 100KLOC C/C++ code and **200+ stars** on Github.
  - Invited for a **plenary talk in EuroLLVM 2016** and **ICSE Distinguished Paper 2018**.

SVF : Static Value-Flow Analysis

A sparse, on-demand, interprocedural program dependence analysis framework for both sequential and multithreaded programs.

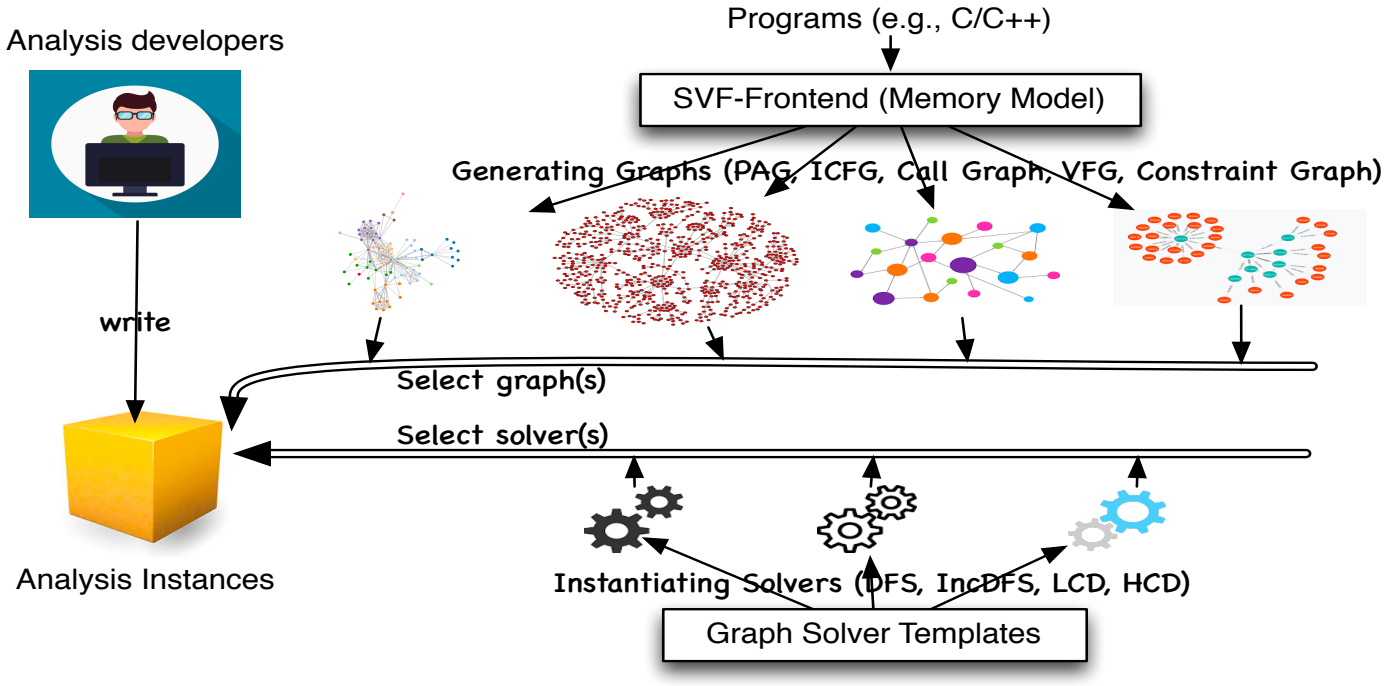
- The SVF project
  - Started since early 2014, actively maintained. **Publicly available** at : `http://svf-tools.github.io/SVF`.
  - Implemented on top of LLVM compiler (the latest version 7.0.0) with over 100KLOC C/C++ code and **200+ stars** on Github.
  - Invited for a **plenary talk in EuroLLVM 2016** and **ICSE Distinguished Paper 2018**.
- Value-Flow Analysis: resolves both control and data dependence.
  - 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?

SVF : Static Value-Flow Analysis

A sparse, on-demand, interprocedural program dependence analysis framework for both sequential and multithreaded programs.

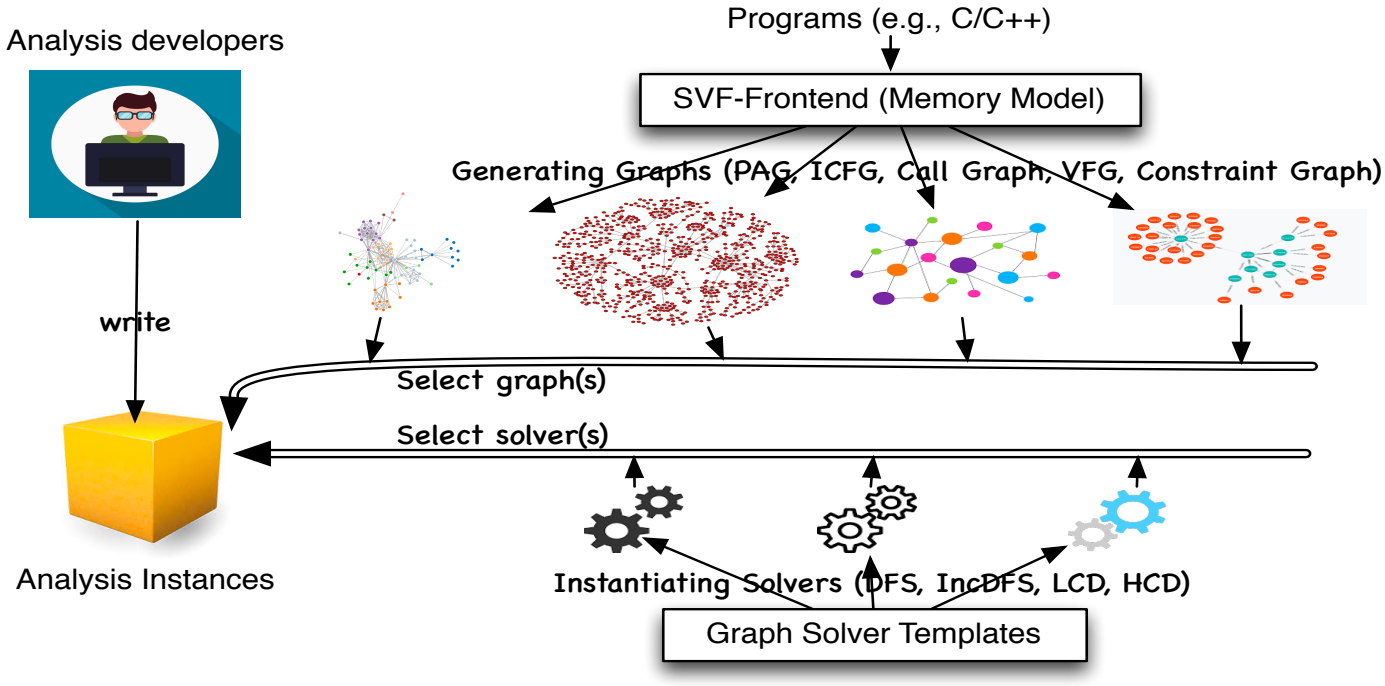
- The SVF project
  - Started since early 2014, actively maintained. **Publicly available** at : `http://svf-tools.github.io/SVF`.
  - Implemented on top of LLVM compiler (the latest version 7.0.0) with over 100KLOC C/C++ code and **200+ stars** on Github.
  - Invited for a **plenary talk in EuroLLVM 2016** and **ICSE Distinguished Paper 2018**.
- Value-Flow Analysis: resolves both control and data dependence.
  - 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 features of SVF
  - Sparse: compute and maintain the data-flow facts where necessary
  - On-demand : reason about program parts based on user queries.
  - Refinement : support mixed analyses for precision and efficiency trade-offs.

SVF: Design Principle



- Serving as an open-source foundation for building practical value-flow analysis
  - Minimize the efforts of implementing sophisticated analysis (extendable, reusable, and robust via layers of abstractions)
  - Support developing different analysis variants (flow-, context-, heap-, field-sensitive analysis) in a sparse and on-demand manner.

SVF: Design Principle



- Serving as an open-source foundation for building practical value-flow analysis
  - Minimize the efforts of implementing sophisticated analysis (extendable, reusable, and robust via layers of abstractions)
  - Support developing different analysis variants (flow-, context-, heap-, field-sensitive analysis) in a sparse and on-demand manner.
- Client applications:
  - Static bug detection (e.g., memory leaks, null dereferences, use-after-frees and data-races)
  - Accelerate dynamic analysis (e.g., Google's Sanitizers and AFL fuzzing)

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

- Cited by leading program analysis and security groups, e.g., Chopped Symbolic Execution (from Imperial College London@ICSE'18), PinPoint (from HKUST@PLDI'18), Type-based CFI (from ACSAC'18@MIT and Northeastern University), Kernel Fuzzing (from Purdue@IEEE S&P'18), Directed Fuzzer (from NTU@CCS'18) and K-Miner (from TU Darmstadt@NDSS'18).
- Used, commented and contributed by researchers from IBM, UCSB, UIUC, Cambridge, Wisconsin-Madison through our Github (comments).

Outline

- **Existing software bugs and vulnerabilities**
- **Static analysis and dynamic analysis**
- **Technical contributions**
  - Fundamental program analysis
    - Sparse value-flow analysis (ISSTA '12, TSE '14, SPE '14, CC '16)
    - On-demand value-flow analysis (CGO '13, SAS '14, FSE '16, TSE '18)
    - Value-flow analysis for multithreaded programs (ICPP '15, CGO '16)
  - Applications
    - Static value-flow analysis for detecting memory errors (ISSTA '12, ACSAC '17, ICSE '18)
    - Hybrid value-flow analysis to enforce memory safety (CGO '14, ISSRE '14, ISSTA '17)
- **Research opportunities**

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

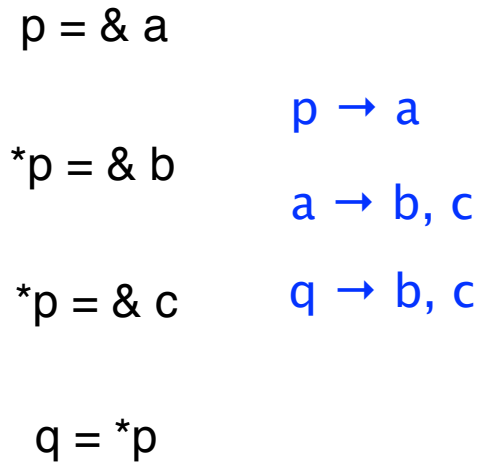
Flow-Insensitive v.s. Flow-Sensitive Analysis

Flow-insensitive pointer analysis:

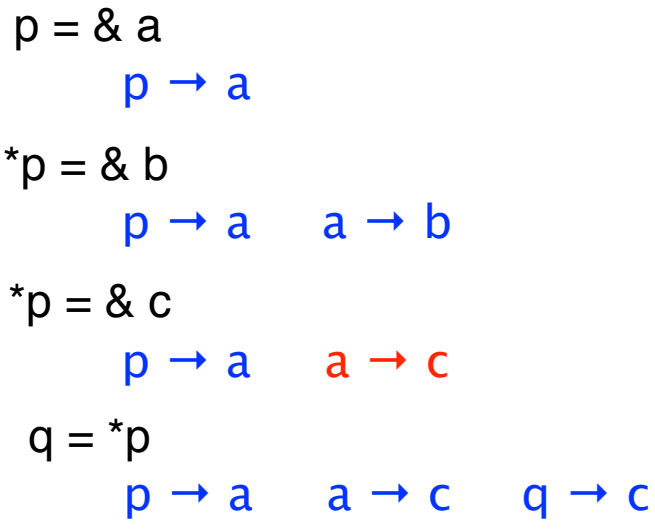
- **Ignore program execution order**
- **A single solution across whole program**

Flow-sensitive pointer analysis:

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



Flow-insensitive analysis



Data-flow-based flow-sensitive analysis

The Data-flow-based Flow-Sensitive Analysis

- **Propagates points-to along the control-flow without knowing whether the information will be used there or not.**

x = & m

x → m

p = & a

p → a x → m

*p = & b

p → a a → b x → m

*p = & c

p → a a → c x → m

*x = & d

p → a a → c x → m m → d

y = *x

p → a a → c x → m m → d y → d

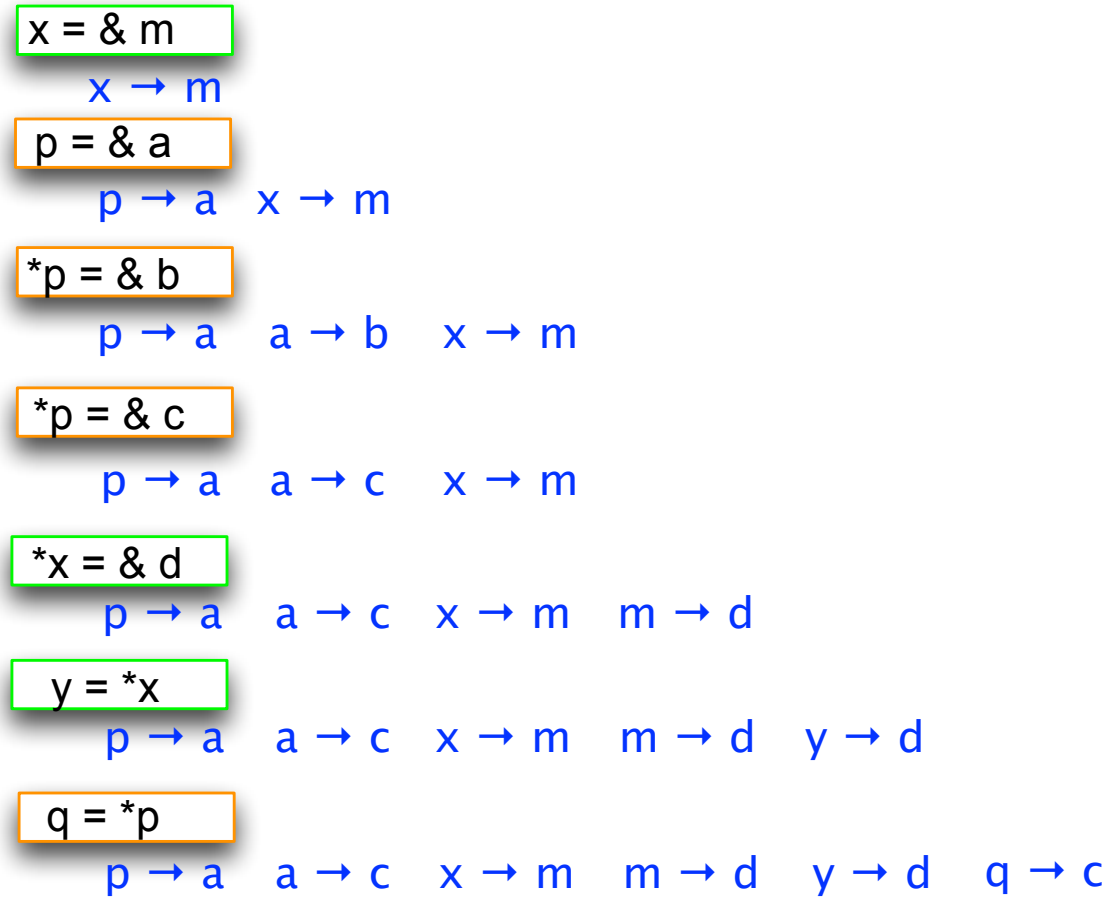
q = *p

p → a a → c x → m m → d y → d q → c

Data-flow-based flow-sensitive analysis

The Data-flow-based Flow-Sensitive Analysis

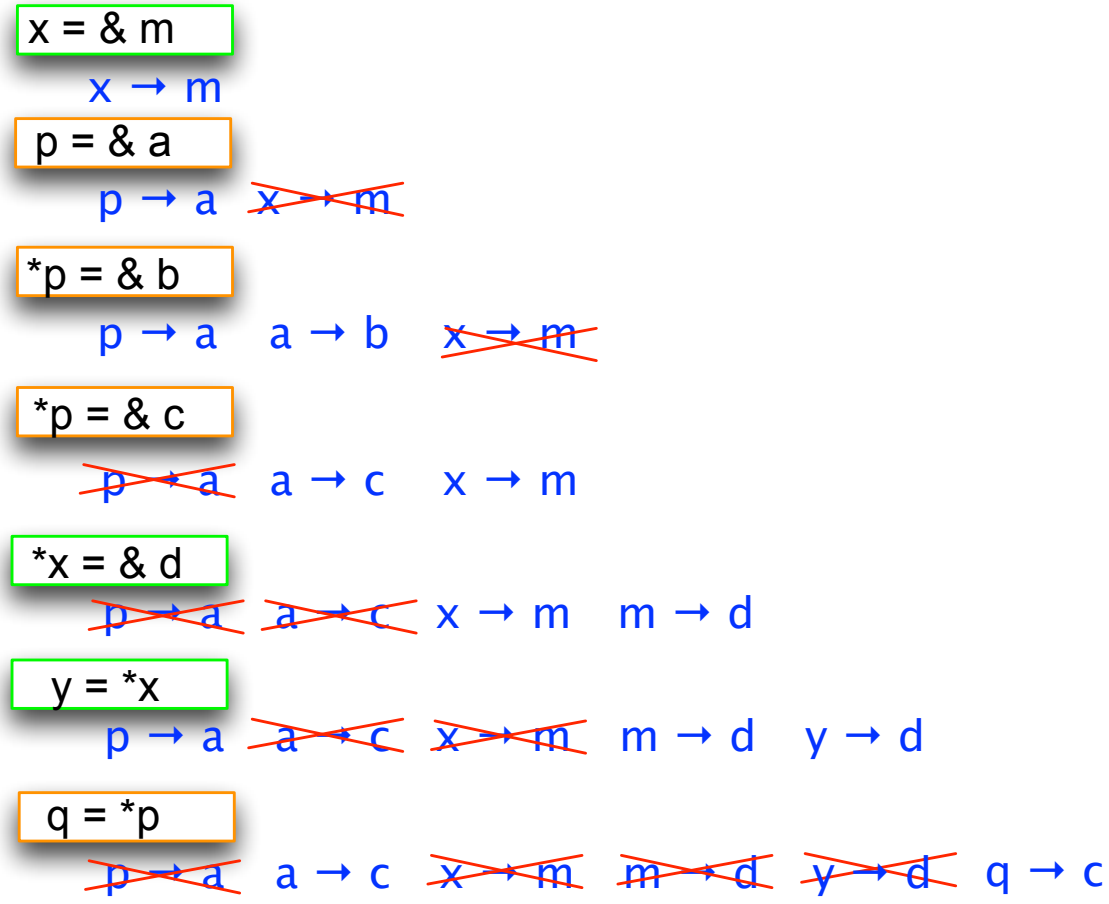
- **Propagates points-to along the control-flow without knowing whether the information will be used there or not.**



Data-flow-based flow-sensitive analysis

Sparse Flow-Sensitive Analysis

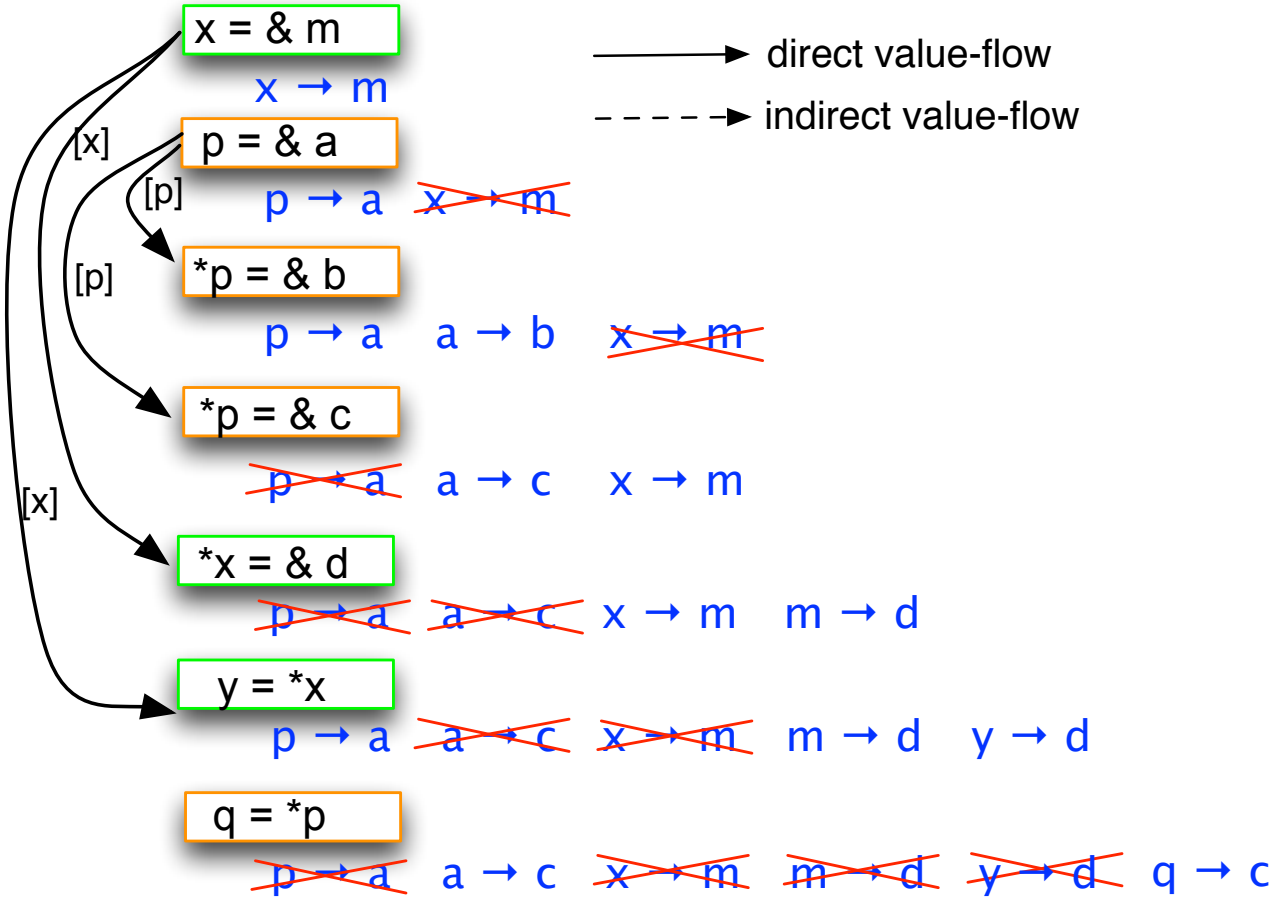
- **Propagate points-to information only along pre-computed def-use chains (a.k.a value-flows) instead of control-flow**



Data-flow-based flow-sensitive analysis

Sparse Flow-Sensitive Analysis

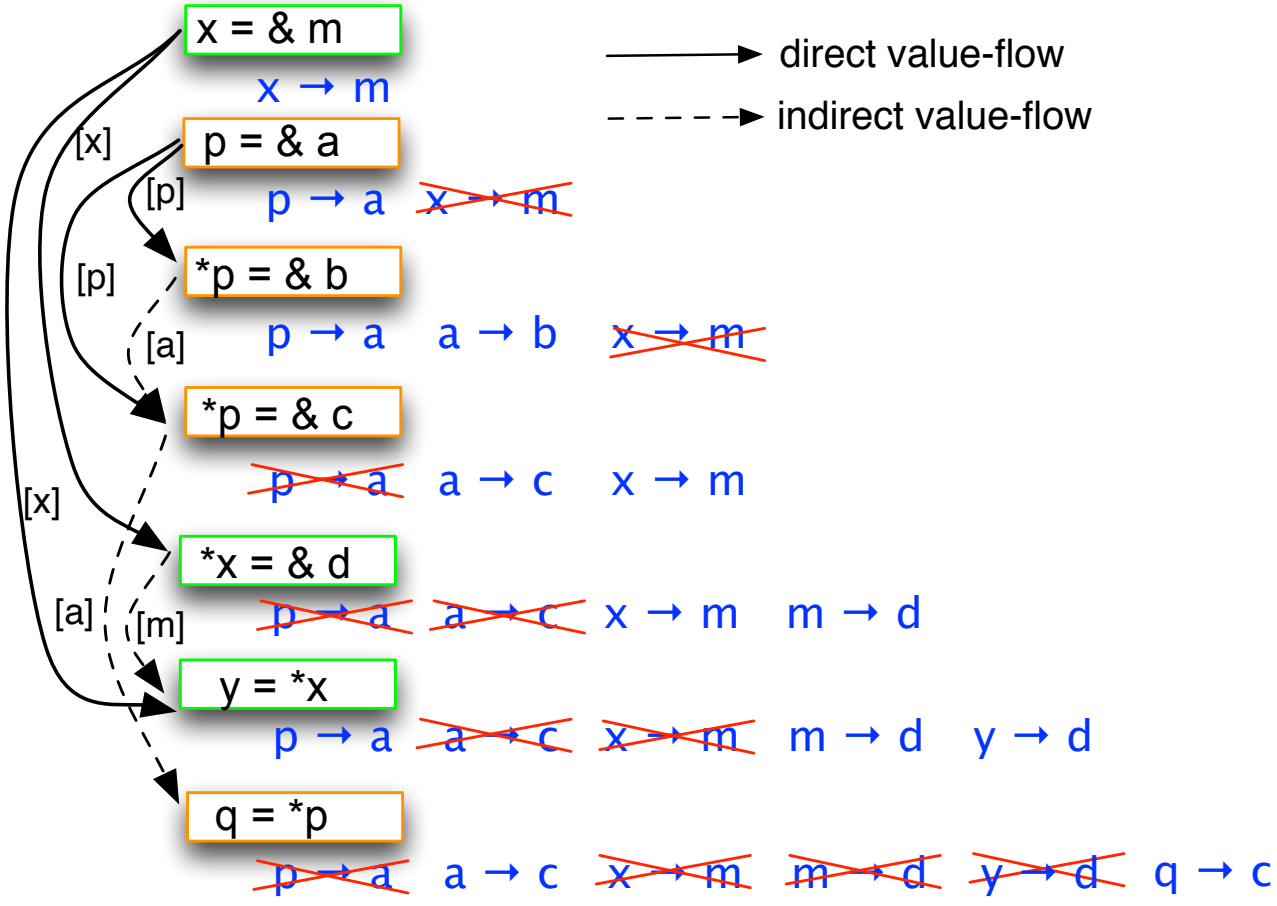
- **Propagate points-to information only along pre-computed def-use chains (a.k.a value-flows) instead of control-flow**



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

Sparse Flow-Sensitive Analysis

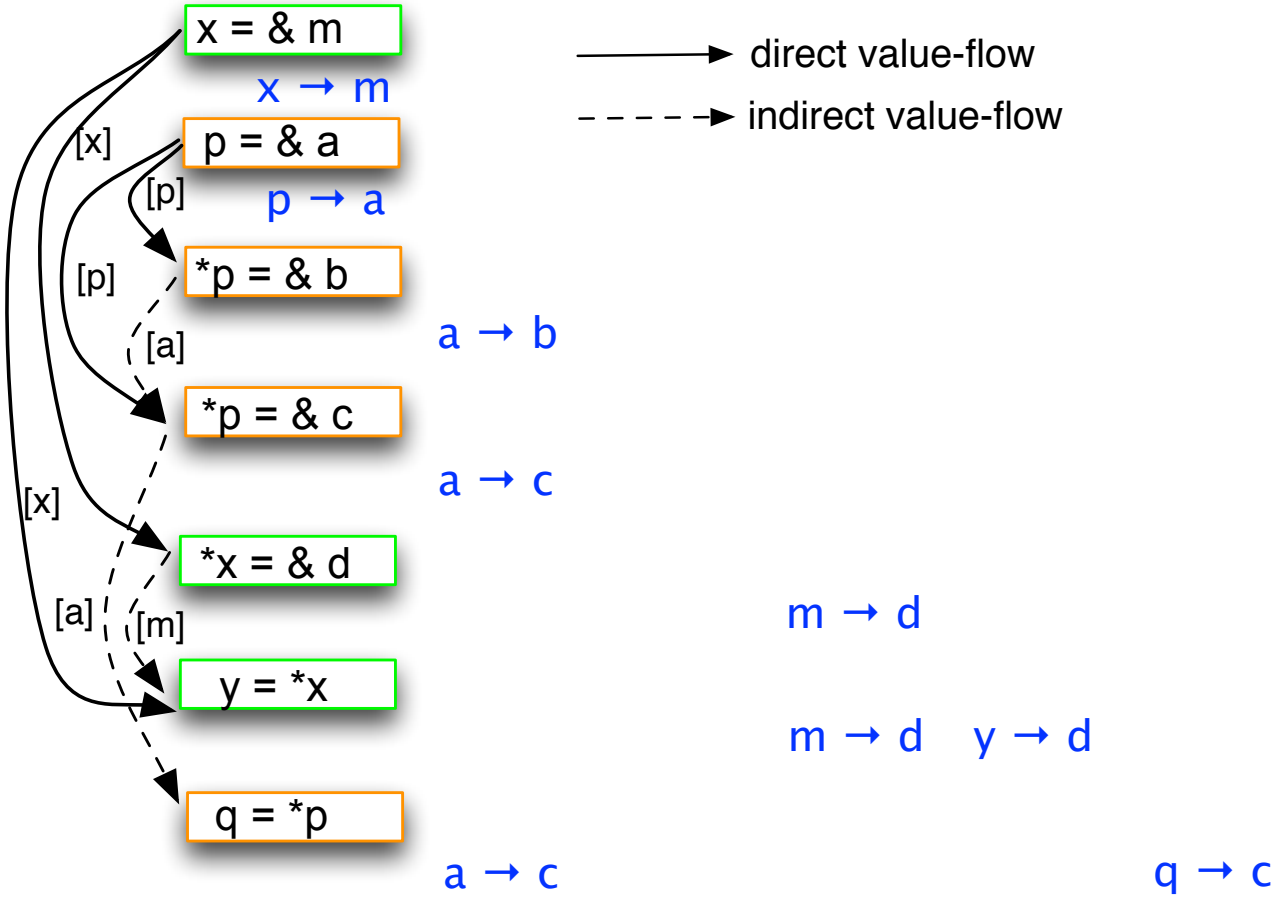
- **Propagate points-to information only along pre-computed def-use chains (a.k.a value-flows) instead of control-flow**



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

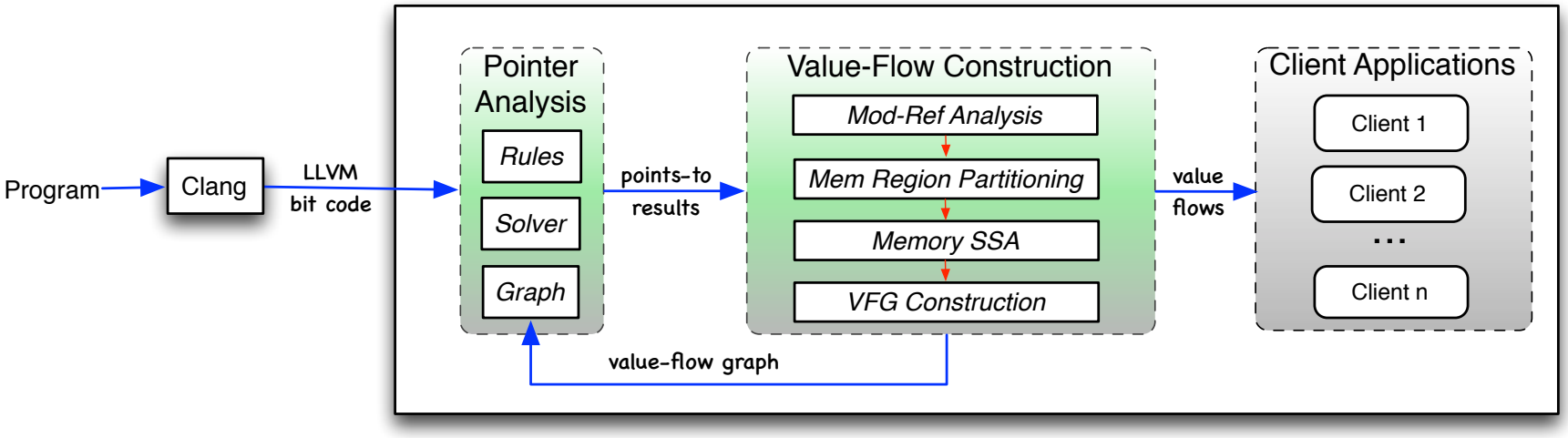
Sparse Flow-Sensitive Analysis

- **Propagate points-to information only along pre-computed def-use chains (a.k.a value-flows) instead of control-flow**



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

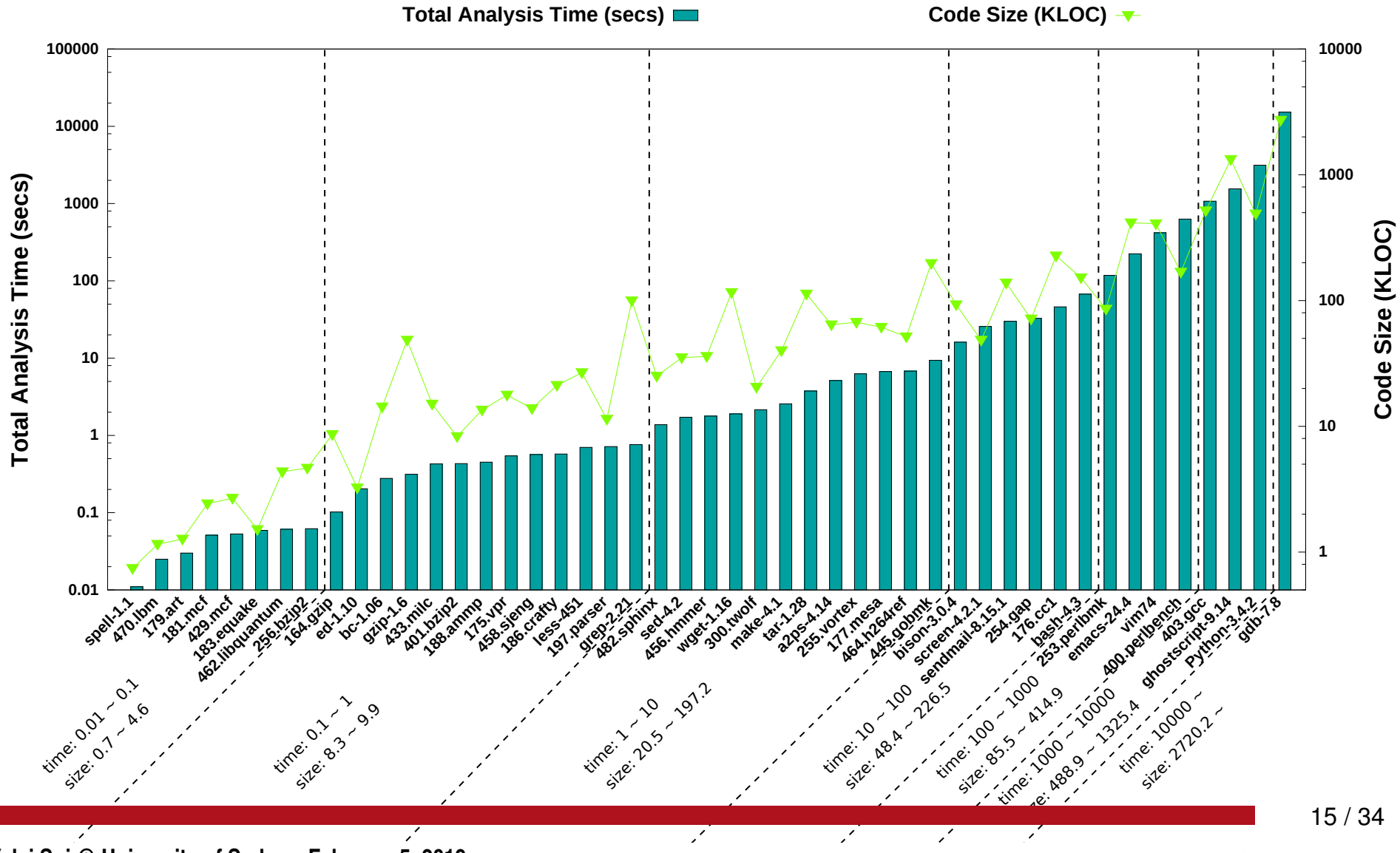
Evaluation and Results



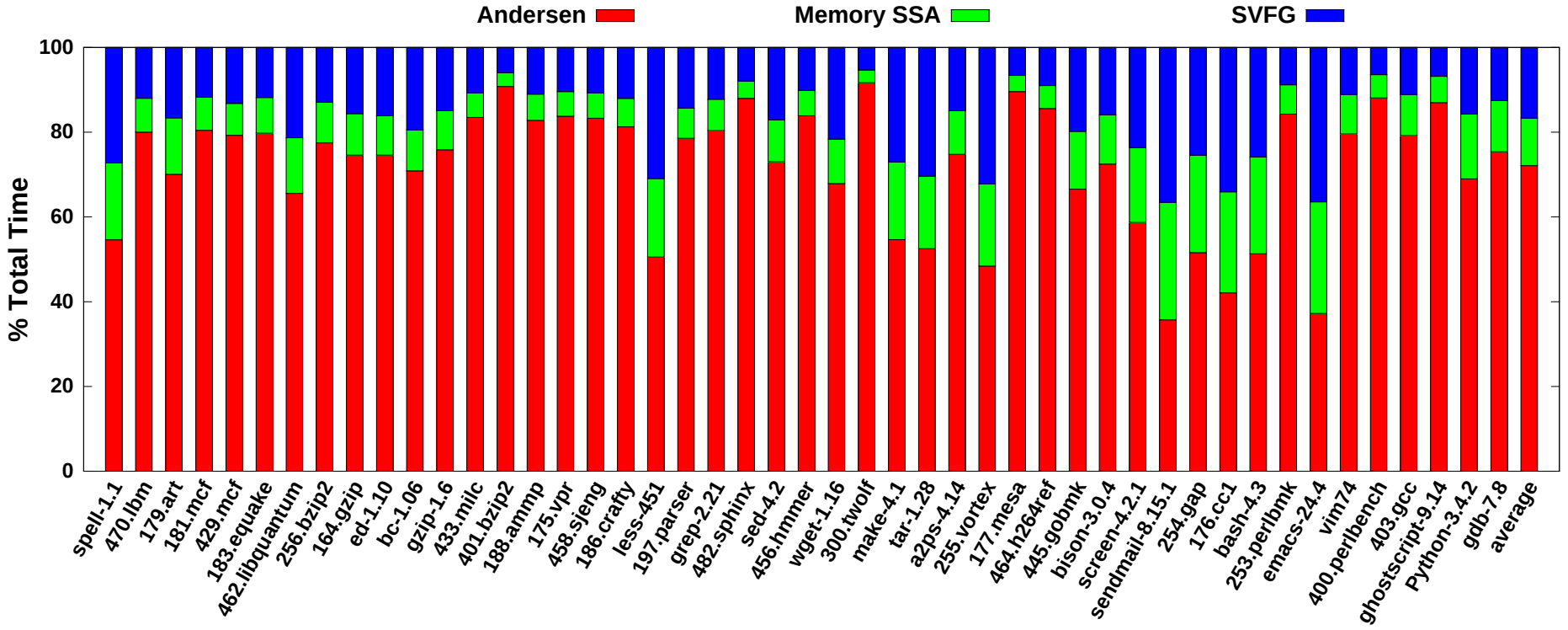
- Benchmarks:
  - All SPEC C benchmarks: 15 programs from CPU2000 and 12 programs from CPU2006
  - 20 Open-source applications: most of them are recent released versions.
  - Total lines of code evaluated: 8,005,872 LOC with maximum program size 2,720,279 LOC
- Machine setup:
  - Ubuntu Linux 3.11.0-15-generic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

Analysis Time

Total Analysis Time = Andersen + MemorySSA + VFG

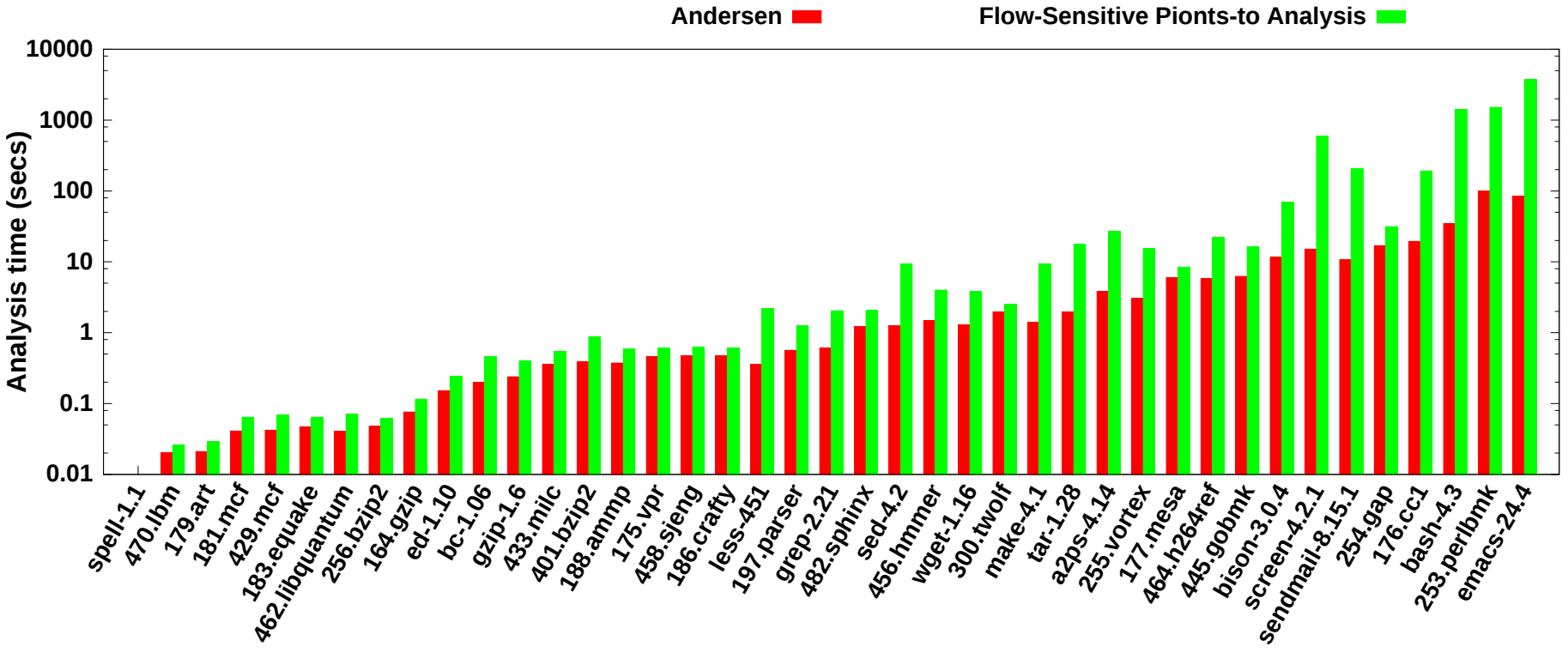


Analysis Time Distribution



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

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



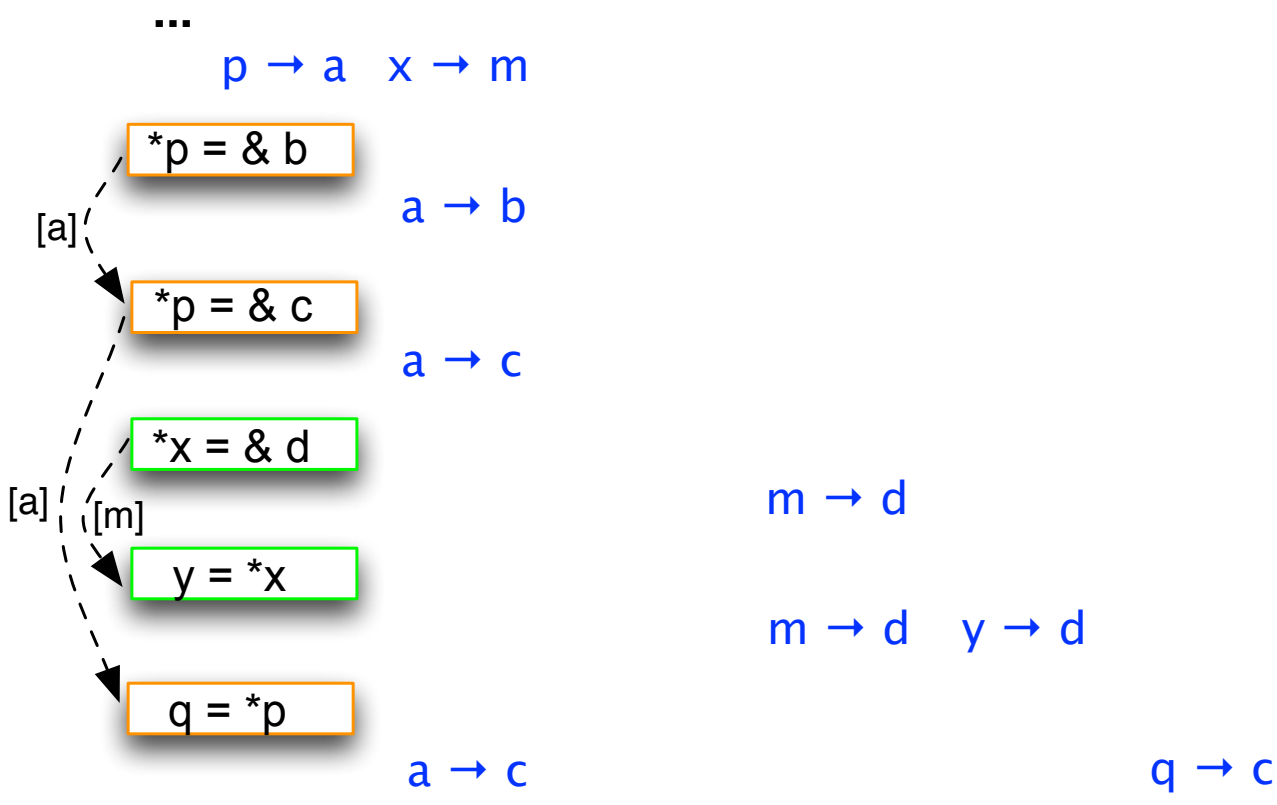
Flow-Sensitive Analysis Slowdowns: From 1.2× to 44×. On average 6.5×.

Outline

- **Existing software bugs and vulnerabilities**
- **Static analysis and dynamic analysis**
- **Technical contributions**
  - Fundamental program analysis
    - Sparse value-flow analysis (ISSTA '12, TSE '14, SPE '14, CC '16)
    - On-demand value-flow analysis (CGO '13, SAS '14, FSE '16, TSE '18)
    - Value-flow analysis for multithreaded programs (ICPP '15, CGO '16)
  - Applications
    - Static value-flow analysis for detecting memory errors (ISSTA '12, ACSAC '17, ICSE '18)
    - Hybrid value-flow analysis to enforce memory safety (CGO '14, ISSRE '14, ISSTA '17)
- **Research opportunities**

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

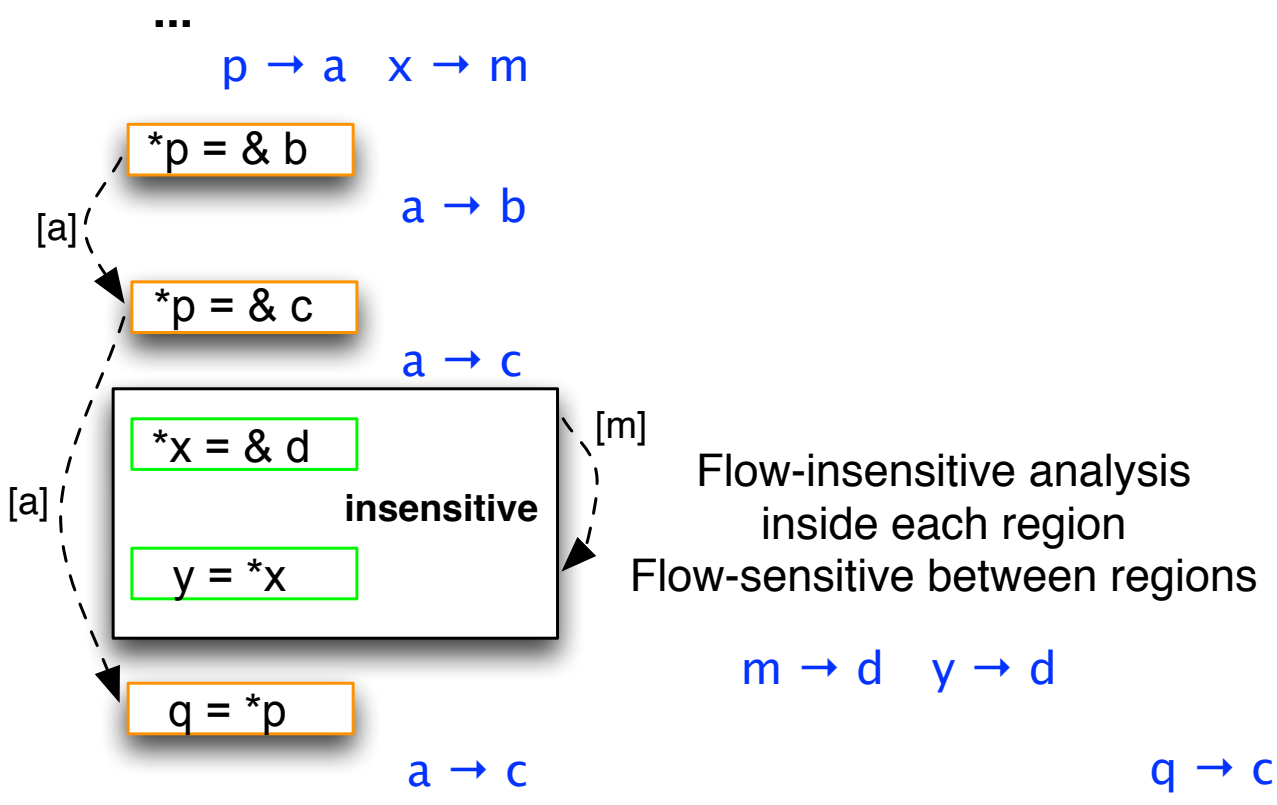
- **Hybrid flow-insensitive and sensitive pointer analysis that operate on the regions partitioned from a program**



Sparse flow-sensitive analysis

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

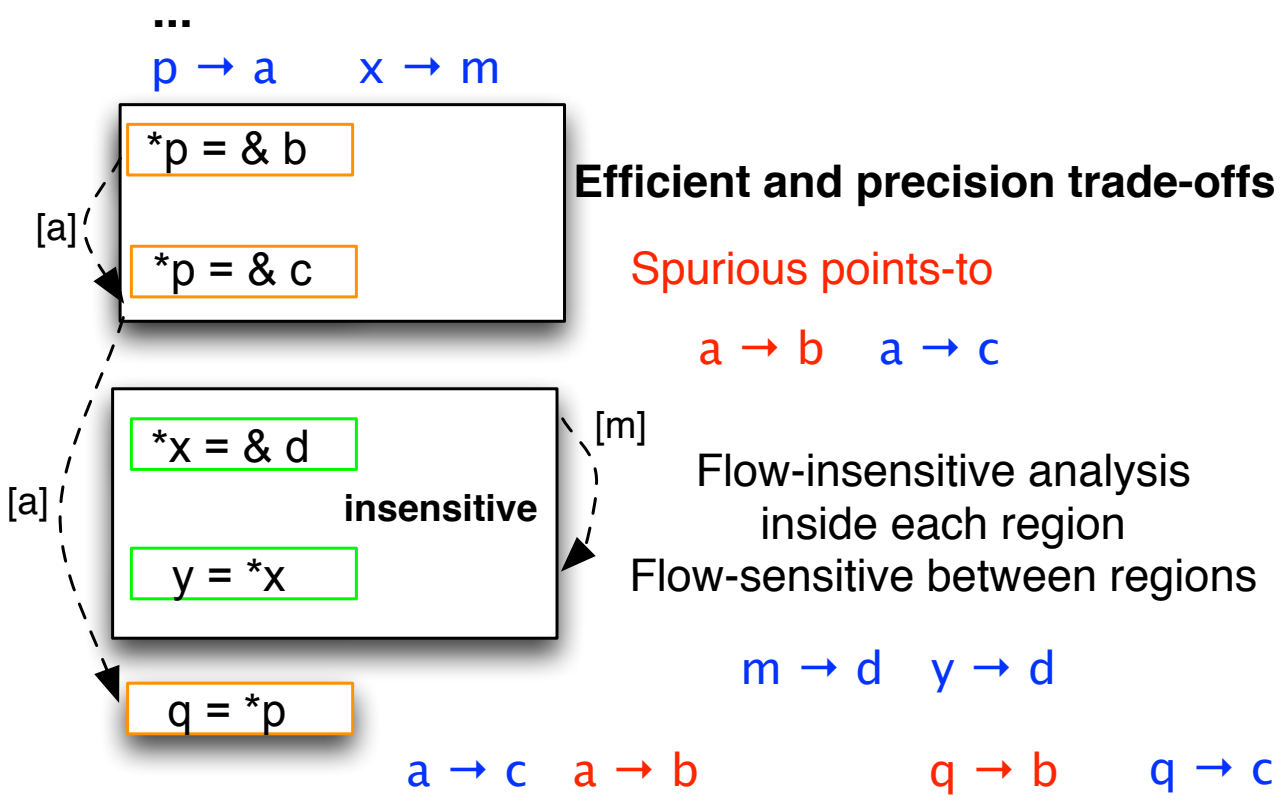
- **Hybrid flow-insensitive and sensitive pointer analysis that operate on the regions partitioned from a program**



Region-based flow-sensitive analysis

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

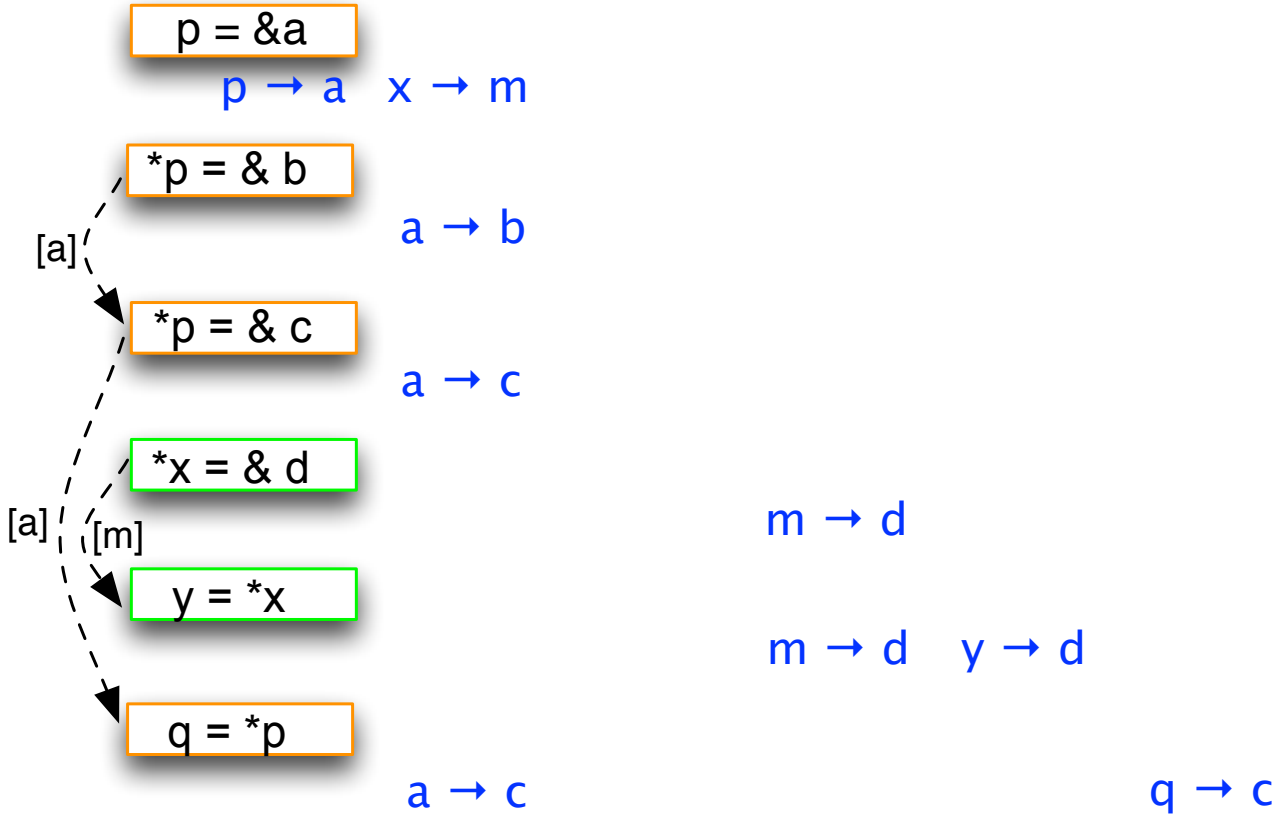
- **Hybrid flow-insensitive and sensitive pointer analysis that operate on the regions partitioned from a program**



Region-based flow-sensitive analysis (**2X** faster than sparse analysis).

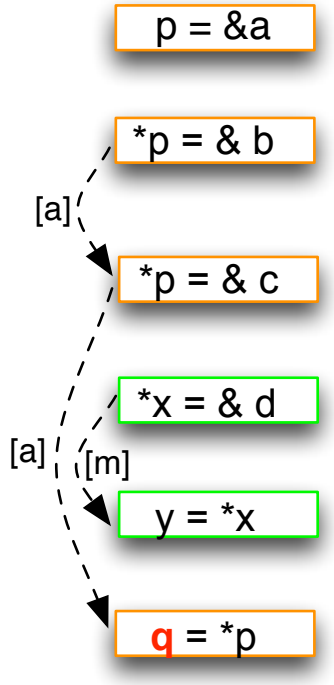
Value-flow-based demand-driven Analysis (FSE '16, TSE'18)

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



Value-flow-based demand-driven Analysis (FSE '16, TSE'18)

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

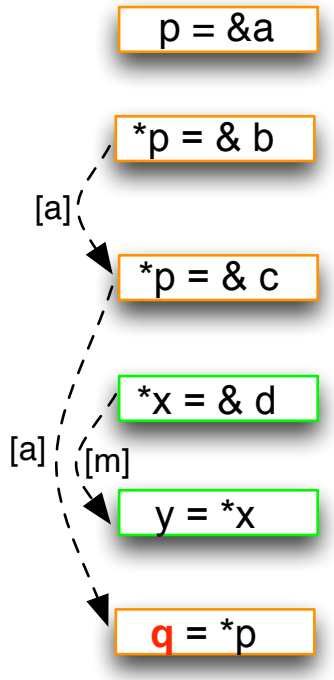


Query: points-to value of q ?

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

Value-flow-based demand-driven Analysis (FSE '16, TSE'18)

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



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

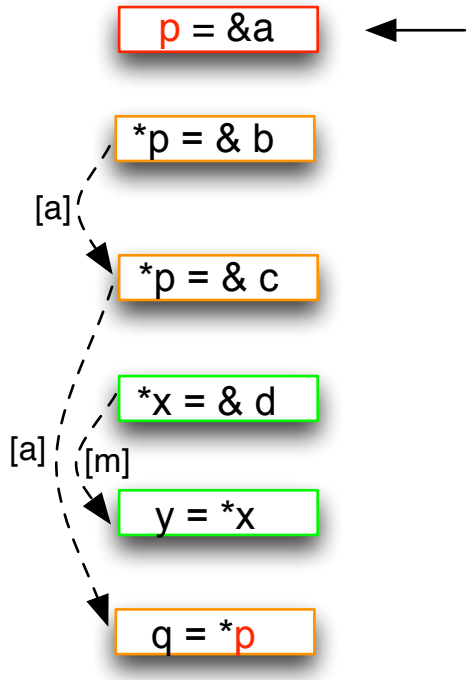
$$\overline{\textit{flowsTo}} \rightarrow (\overline{\text{assign}}\ |\ \text{load}\ \textit{alias}\ \text{store})^*\ \text{alloc}$$

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

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

Value-flow-based demand-driven Analysis (FSE '16, TSE'18)

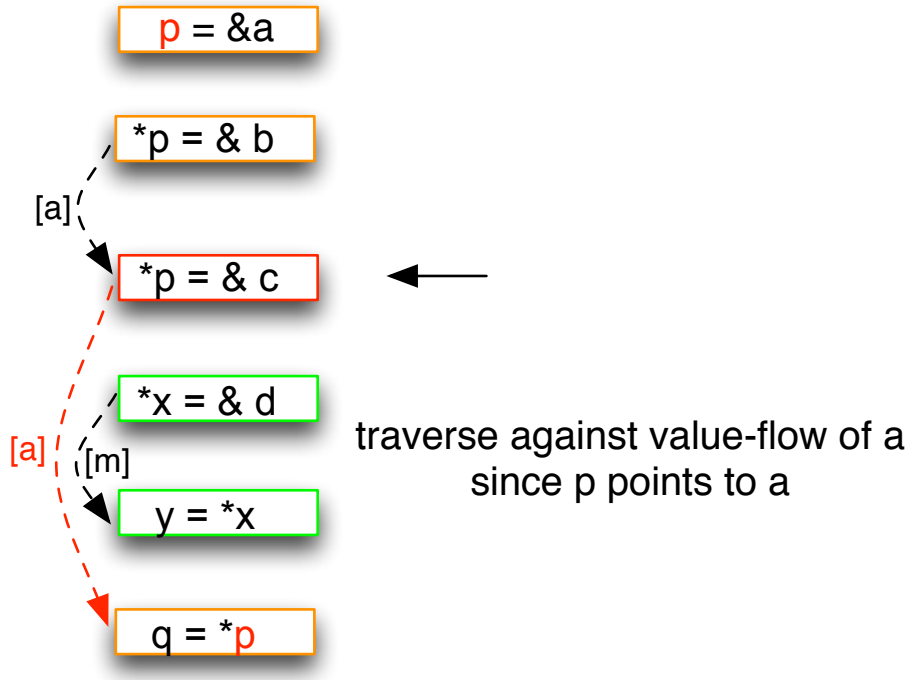
- **Demand-driven flow-sensitive analysis with strong updates via CFL-Reachability**



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

Value-flow-based demand-driven Analysis (FSE '16, TSE'18)

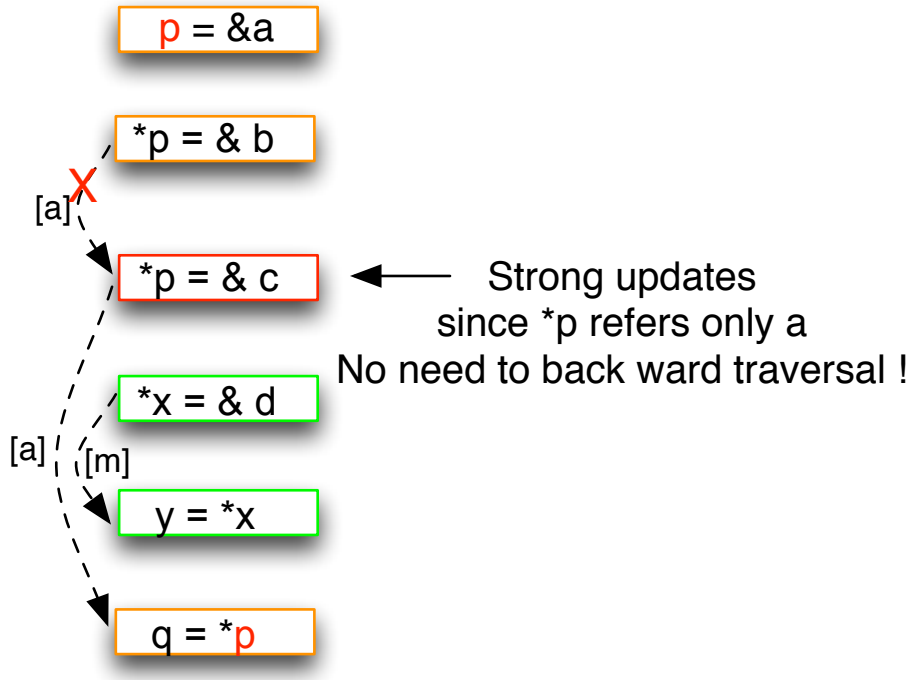
- **Demand-driven flow-sensitive analysis with strong updates via CFL-Reachability**



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

Value-flow-based demand-driven Analysis (FSE '16, TSE'18)

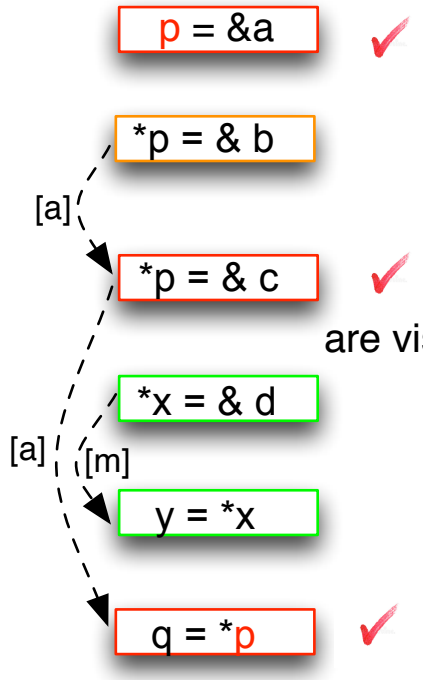
- **Demand-driven flow-sensitive analysis with strong updates via CFL-Reachability**



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

Value-flow-based demand-driven Analysis (FSE '16, TSE'18)

- **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, TSE'18)

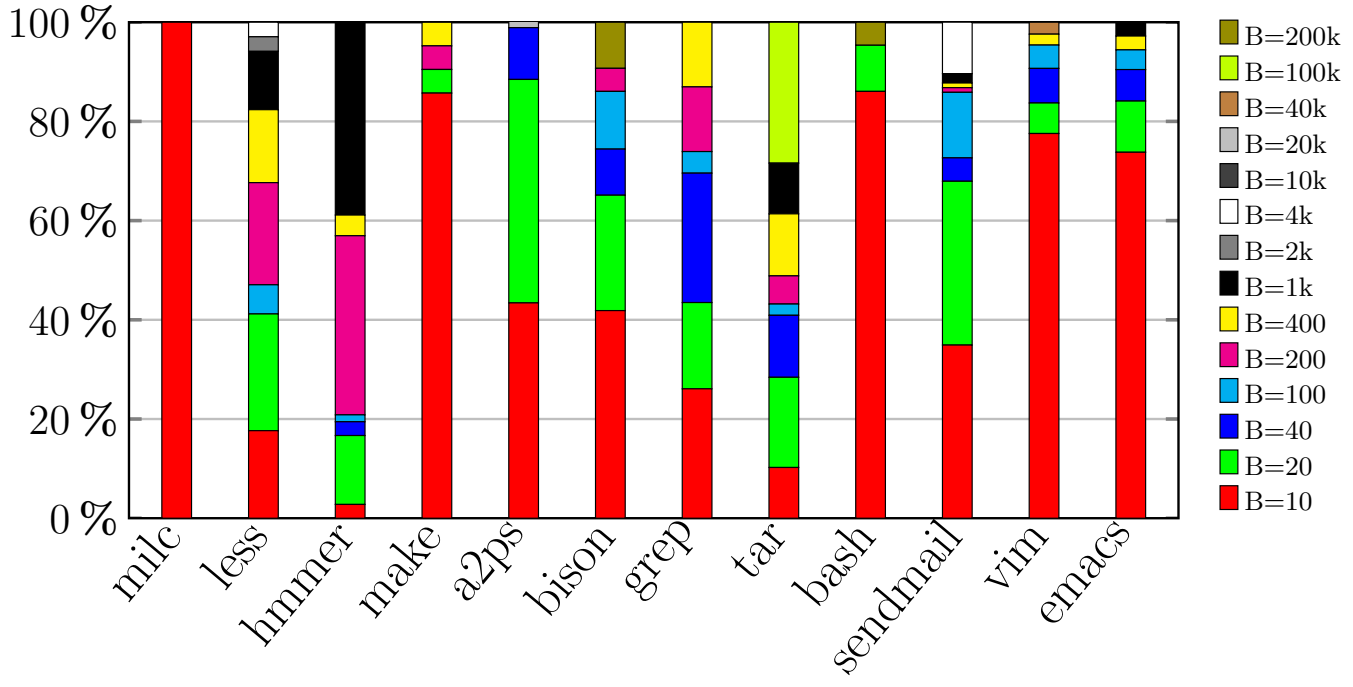


Figure: Percentage of queried variables proved to be safe (initialized) by demand-driven analysis over whole-program analysis under different budgets

Value-flow-based demand-driven Analysis (FSE '16, TSE'18)

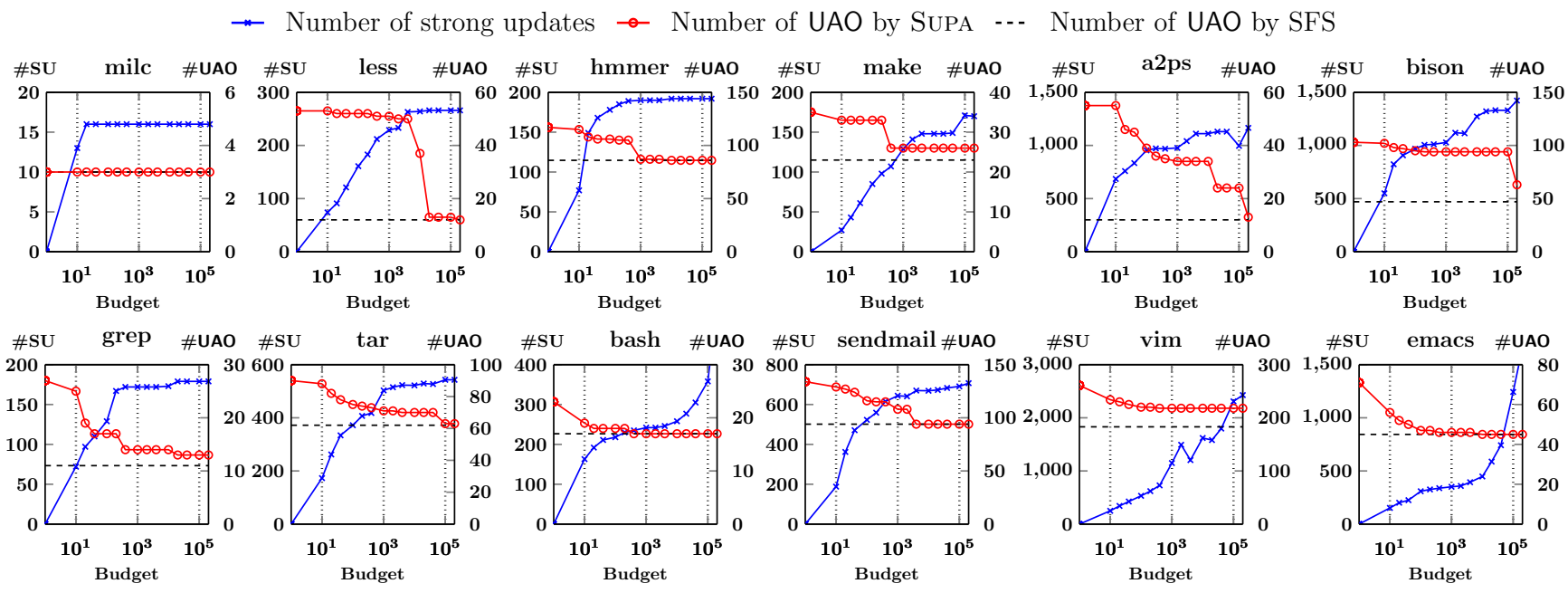
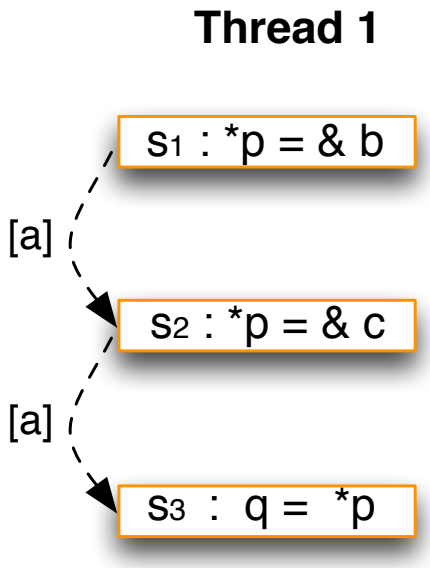


Figure: Correlating the number of strong updates with the number of unintialized variables detected under different budgets

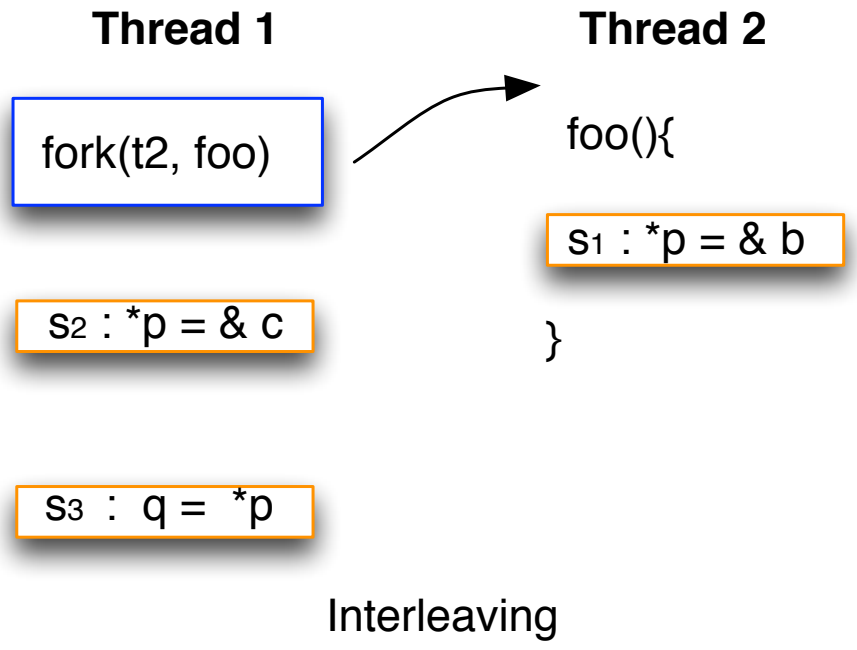
Outline

- **Existing software bugs and vulnerabilities**
- **Static analysis and dynamic analysis**
- **Technical contributions**
  - Fundamental program analysis
    - Sparse value-flow analysis (ISSTA '12, TSE '14, SPE '14, CC '16)
    - On-demand value-flow analysis (CGO '13, SAS '14, FSE '16, TSE '18)
    - Value-flow analysis for multithreaded programs (ICPP '15, CGO '16)
  - Applications
    - Static value-flow analysis for detecting memory errors (ISSTA '12, ACSAC '17, ICSE '18)
    - Hybrid value-flow analysis to enforce memory safety (CGO '14, ISSRE '14, ISSTA '17)
- **Research opportunities**

Flow-Sensitivity Under Thread Interleaving (CGO '16)

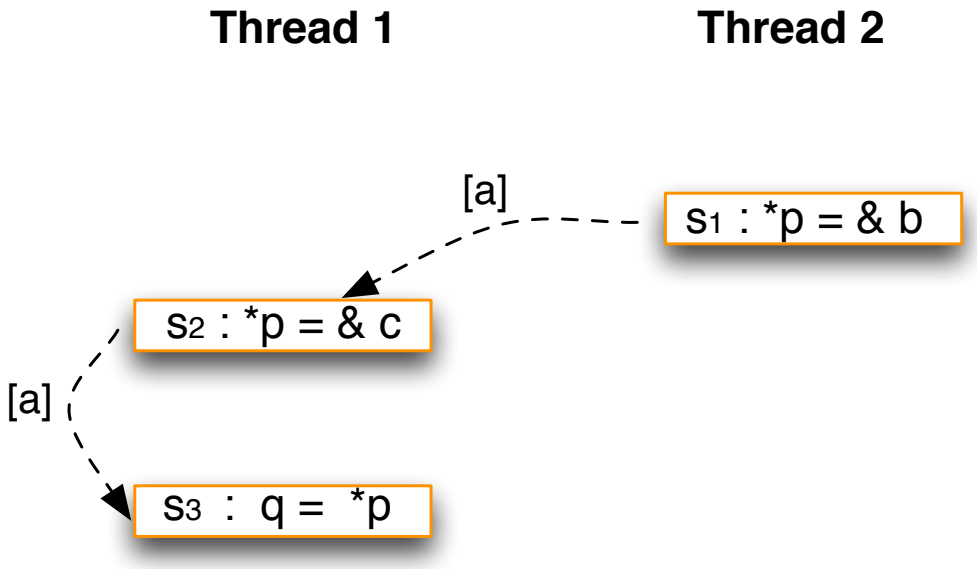


Flow-Sensitivity Under Thread Interleaving (CGO '16)



Flow-Sensitivity Under Thread Interleaving (CGO '16)

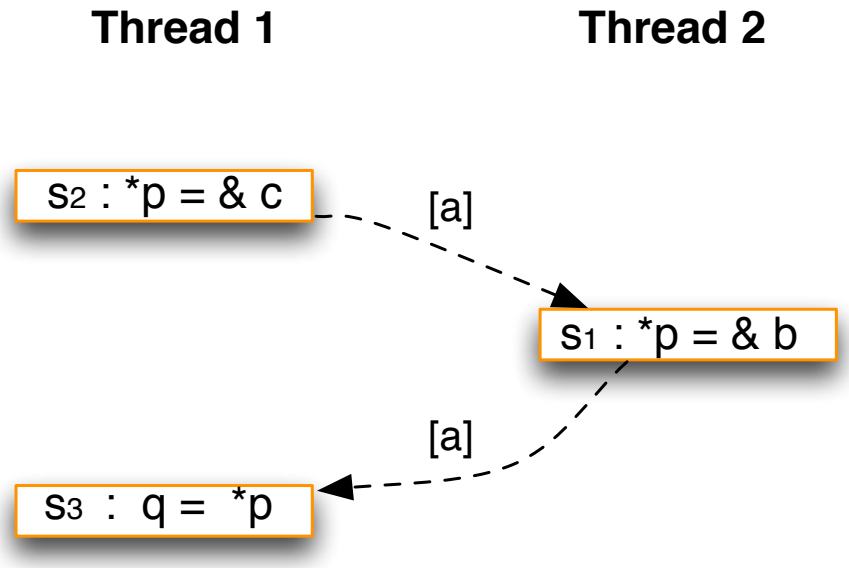
Scenario 1:



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

Flow-Sensitivity Under Thread Interleaving (CGO '16)

Scenario 2:

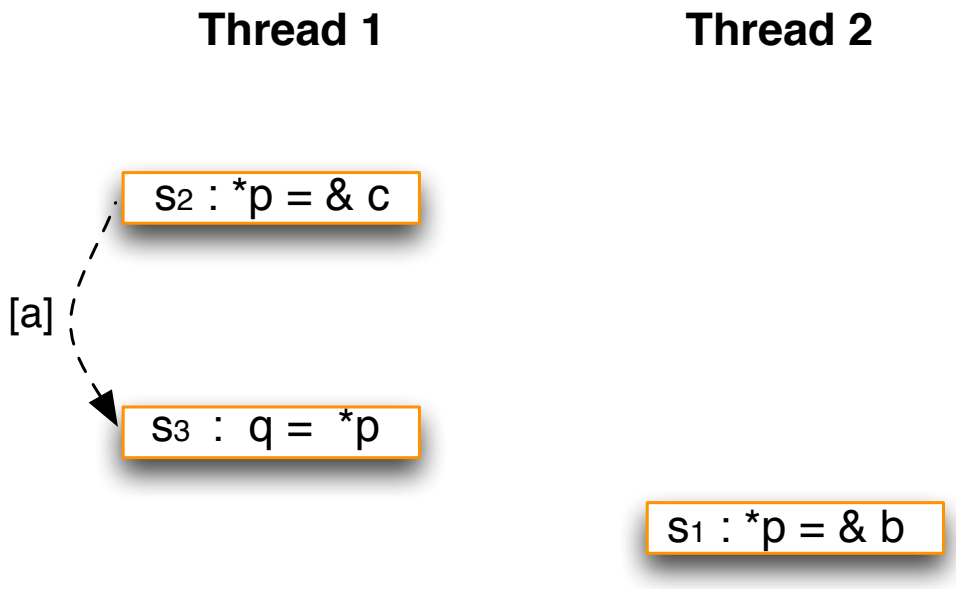


execution sequence : s_1, s_2, s_3 points-to of q : pt(q) = {c}

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

Flow-Sensitivity Under Thread Interleaving (CGO '16)

Scenario 3:

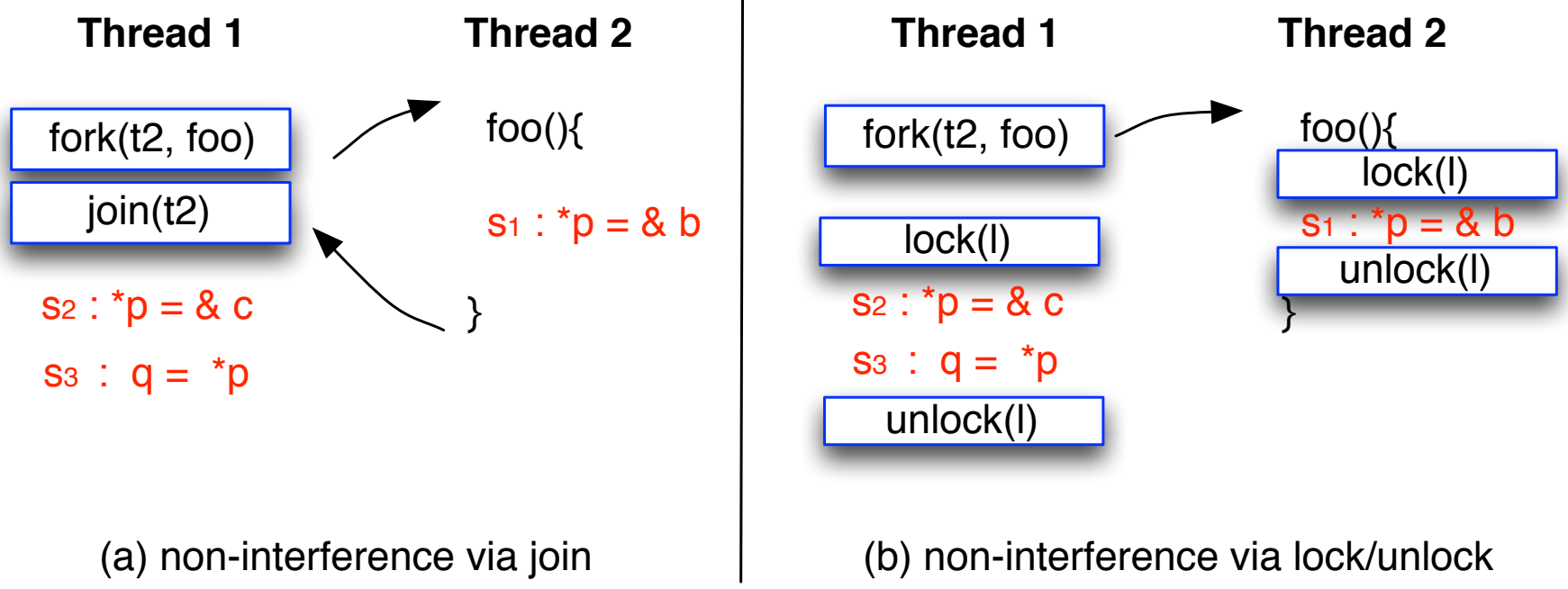


execution sequence : s_1, s_2, s_3 points-to of q : pt(q) = {c}

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)

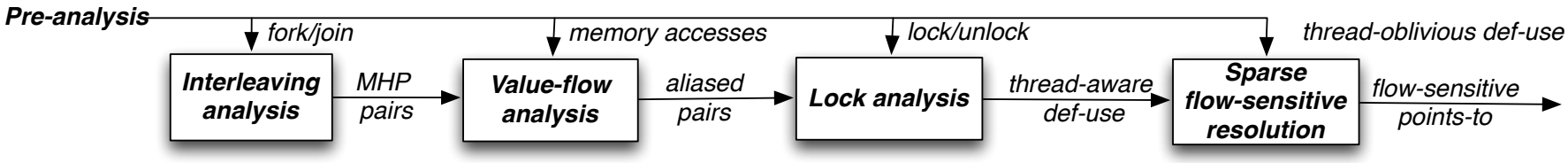


(a) non-interference via join

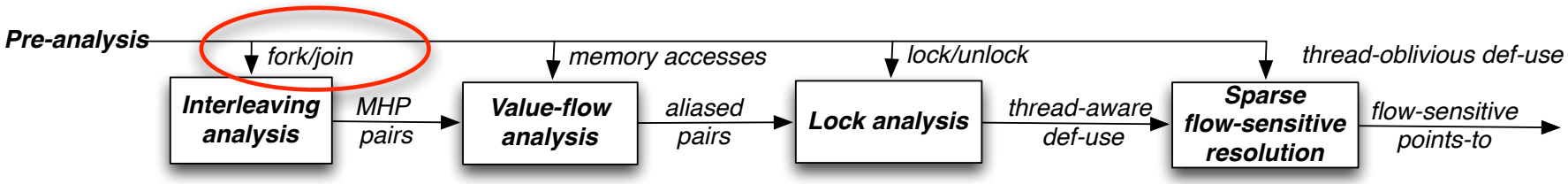
(b) non-interference via lock/unlock

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

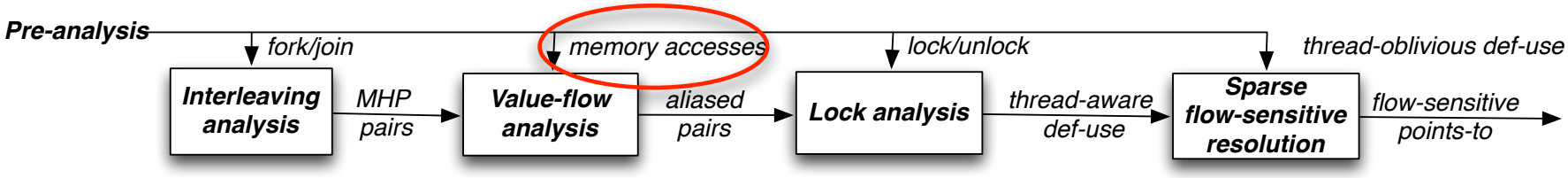
FSAM: Sparse Flow-Sensitive Analysis For Multithreaded Programs (CGO '16)



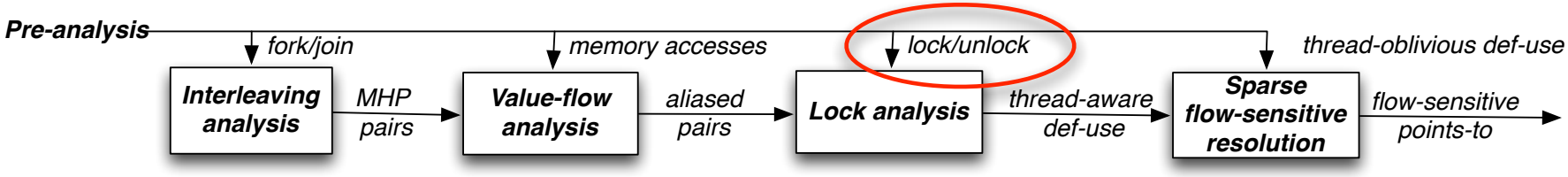
FSAM: Sparse Flow-Sensitive Analysis For Multithreaded Programs (CGO '16)



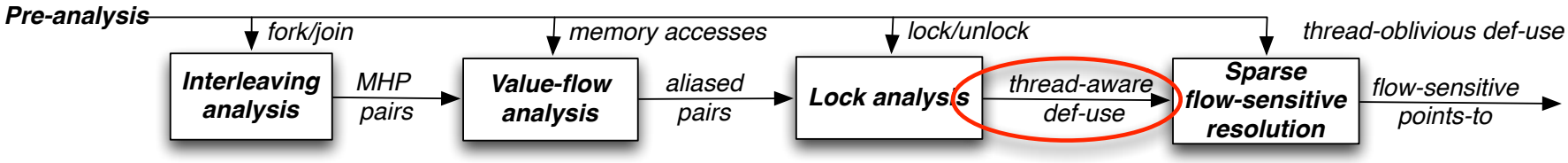
FSAM: Sparse Flow-Sensitive Analysis For Multithreaded Programs (CGO '16)



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



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



Analysis Time and Memory Usage

Comparing FSAM v.s. NONSPARSE iterative flow-sensitive pointer analysis[2]

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

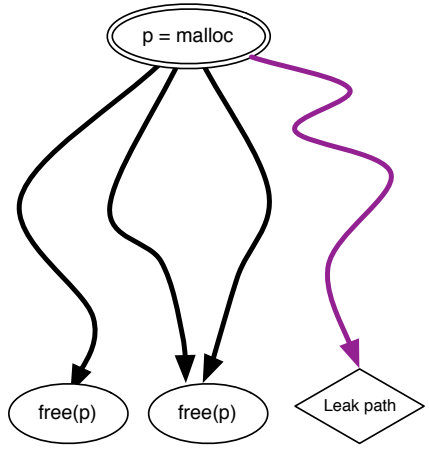
FSAM is 12x faster and uses 28x less memory.

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

Outline

- **Existing software bugs and vulnerabilities**
- **Static analysis and dynamic analysis**
- **Technical contributions**
  - Fundamental program analysis
    - Sparse value-flow analysis (ISSTA ’12, TSE ’14, SPE ’14, CC ’16)
    - On-demand value-flow analysis (CGO ’13, SAS ’14, FSE ’16, TSE ’18)
    - Value-flow analysis for multithreaded programs (ICPP ’15, CGO ’16)
  - Applications
    - Static value-flow analysis for detecting memory errors (ISSTA ’12, ACSAC ’17, ICSE ’18)
    - Hybrid value-flow analysis to enforce memory safety (CGO ’14, ISSRE ’14, ISSTA ’17)
- **Research opportunities**

Value-Flow Analysis For Memory Leak Detection (ISSTA '12, TSE '14)



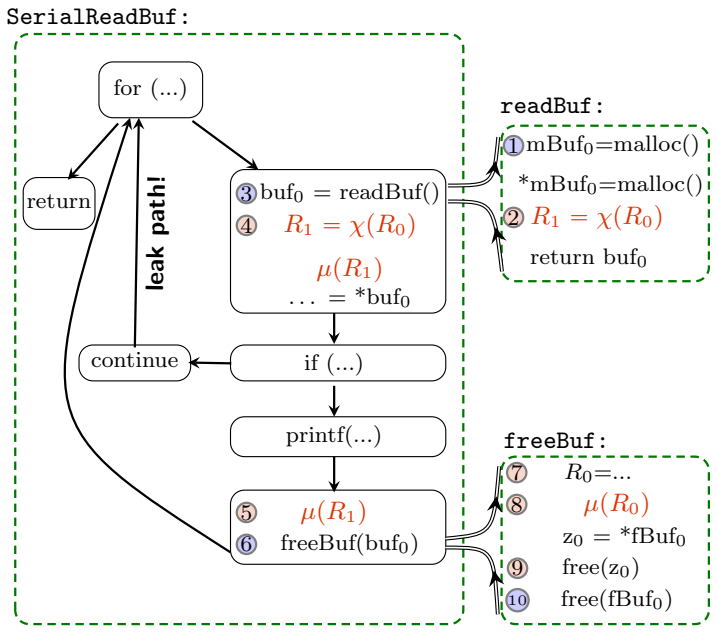
Yulei Sui, Ding Ye, Jingling Xue. Static Memory Leak Detection Using Full-Sparse Value Flow Analysis. *2012 International Symposium on Software Testing and Analysis (**ISSTA '12**)*

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

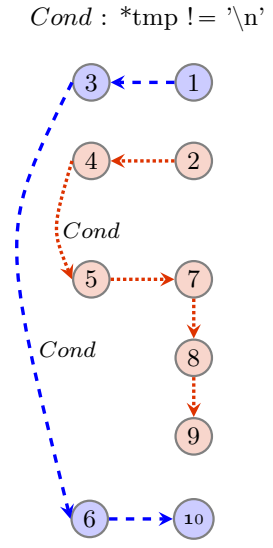
Value-Flow Analysis For Memory Leak Detection (ISSTA '12, TSE '14)

```
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, TSE '14)

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**)*

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

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]

$$\mathbb{ST}\big(\texttt{free}(p@l_f), \texttt{use}(q@l_u)\big) \quad := \quad (1)$$

$$\exists\, (\rho_f, \rho_u) \in \mathcal{P}(l_f) \times \mathcal{P}(l_u) : (\rho_f, l_f) \rightsquigarrow (\rho_u, l_u) \;\wedge\; (\rho_f, p) \cong (\rho_u, q)$$

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]

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

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

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

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

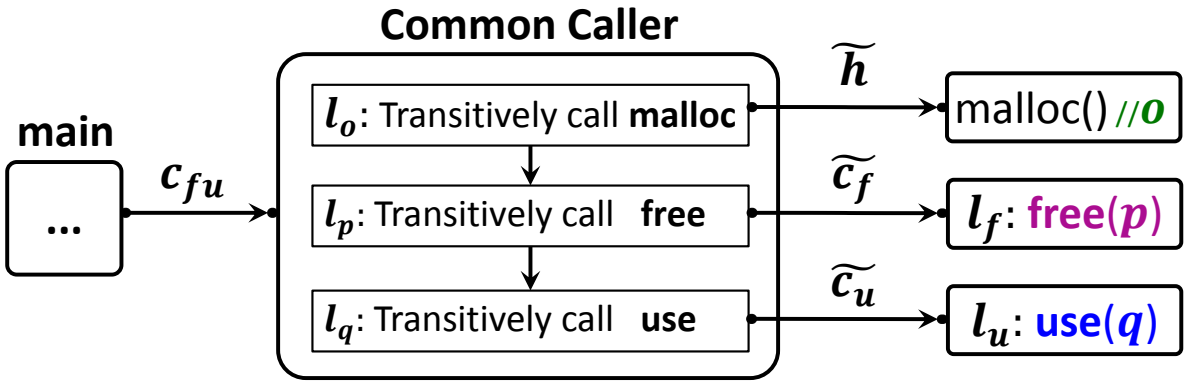
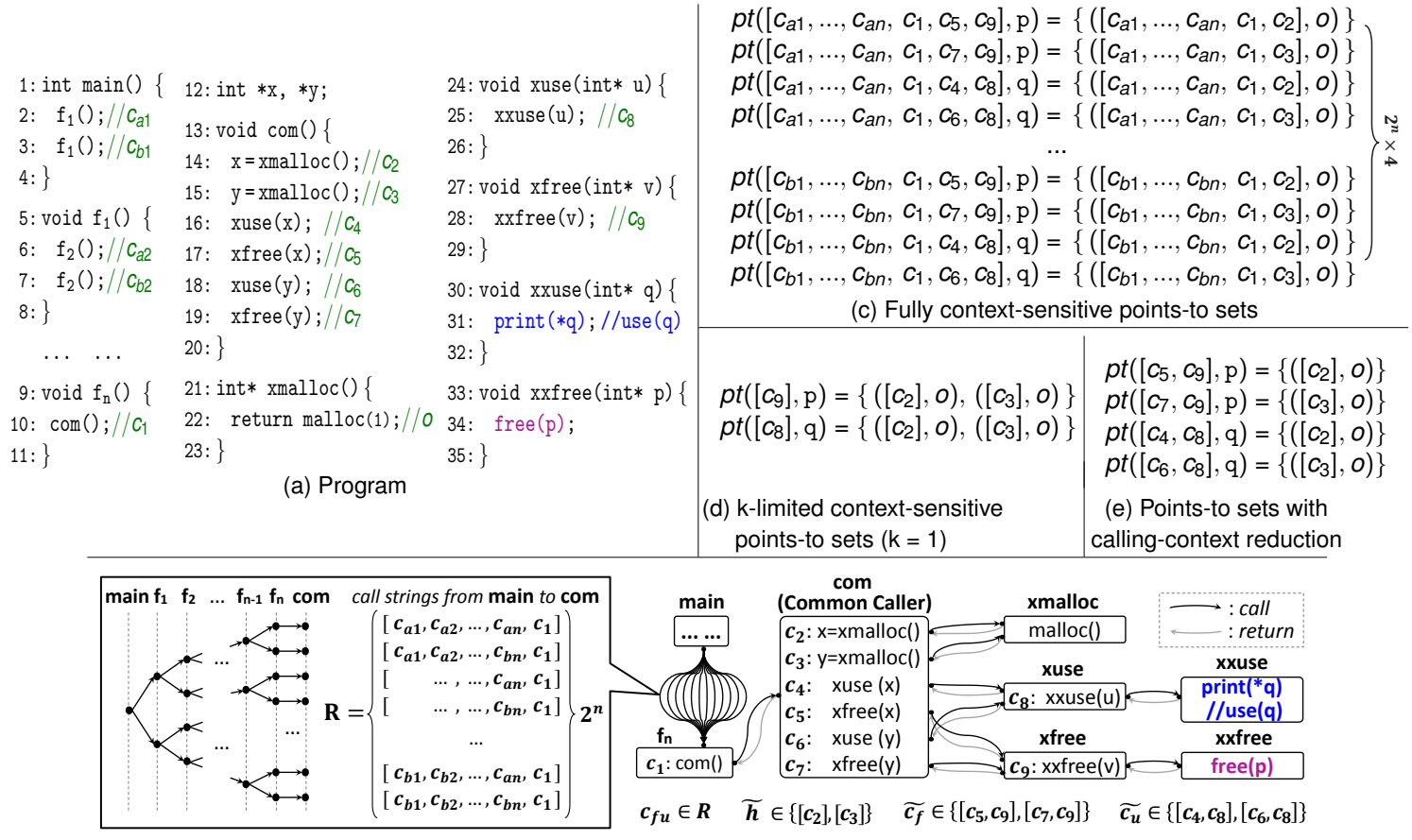


Figure: Context reduction, illustrated conceptually with an oracle fully-context-sensitive pointer analysis.

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

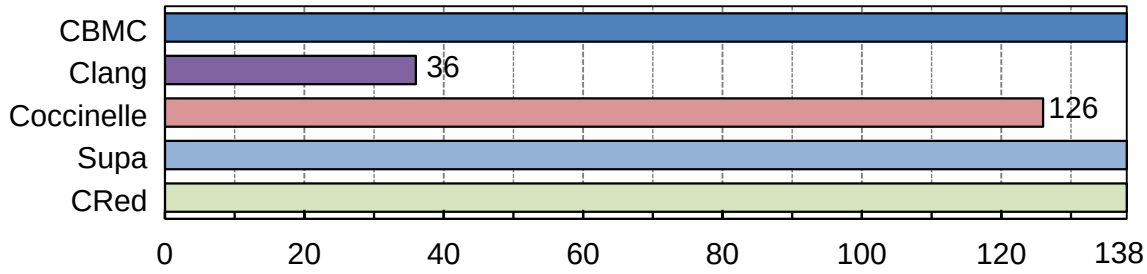


Figure: Hit rates for the 138 bugs in JTS: CBMC (100%), CLANG (26%), COCCINELLE (91%), SUPA (100%) and STC (100%).

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

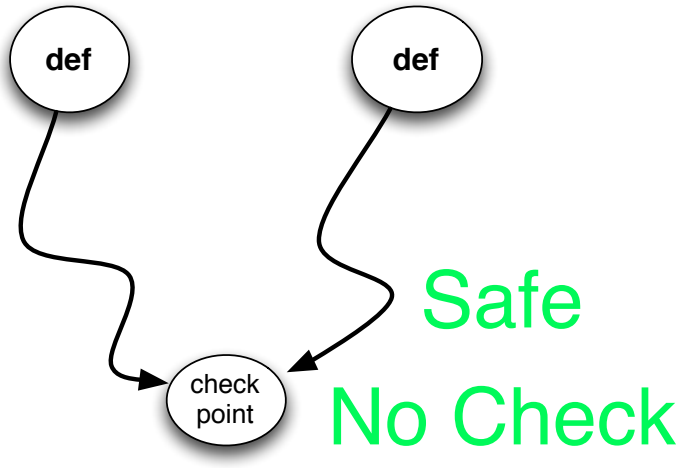
Table: #T:#True Positives (Bugs) and #F: #False Positives (i.e., False Alarms).

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

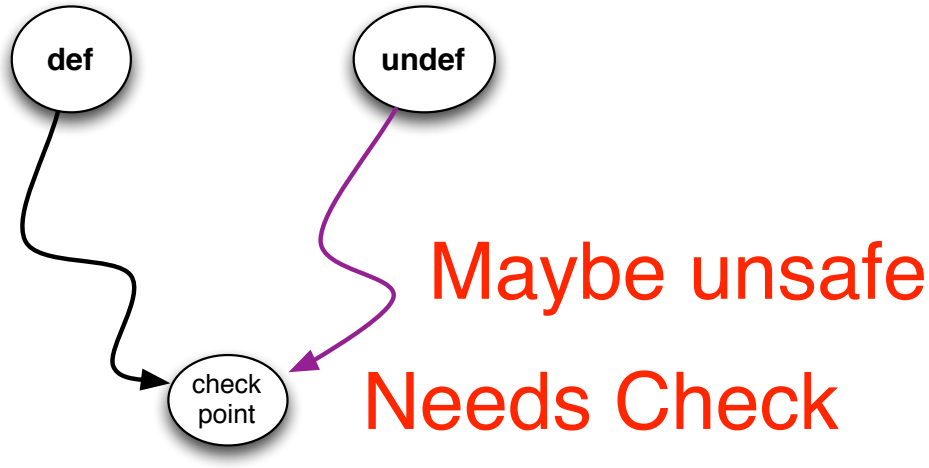
Outline

- **Existing software bugs and vulnerabilities**
- **Static analysis and dynamic analysis**
- **Technical contributions**
  - Fundamental program analysis
    - Sparse value-flow analysis
    - On-demand value-flow analysis
    - Value-flow analysis for multithreaded programs (ICPP ’15, CGO ’16)
  - Applications
    - Static value-flow analysis for detecting memory errors
    - Hybrid value-flow analysis to enforce memory safety (CGO ’14, ISSRE ’14, ISSTA ’17)
- **Research opportunities**

Hybrid Analysis For Uninitialized Variable Detection (CGO '14)



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

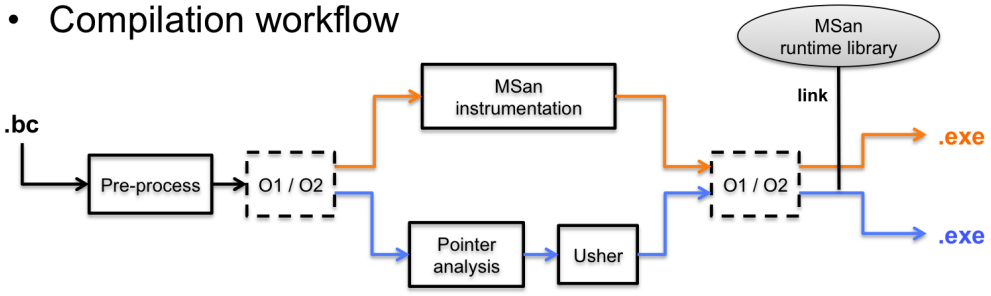


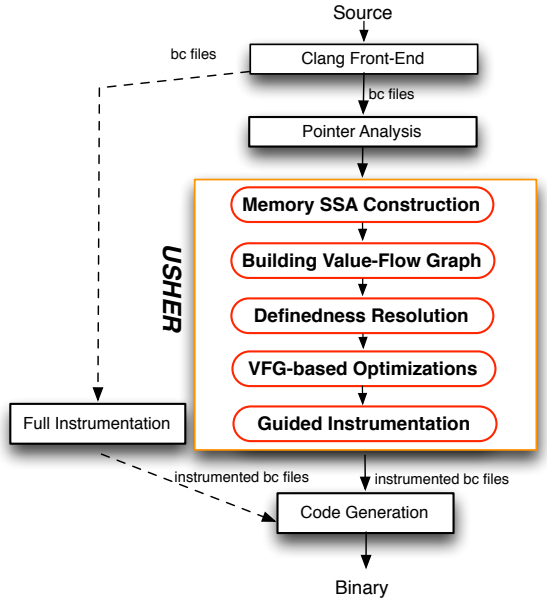
Reachable from an undefined allocation sites along at least one value-flow path

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] = ...;
  }
}
```

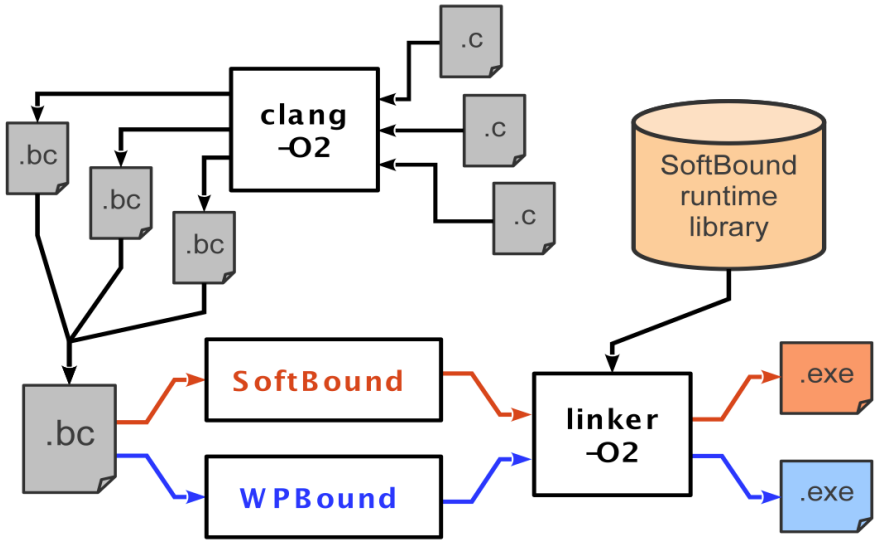
Use conservatively approximated WPs! [true, WP]

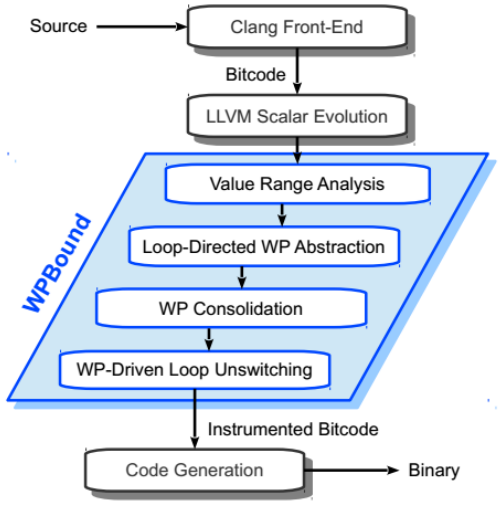
```
inline wpCheck(ptr, sz, base, bound) {
   return (ptr < base || ptr + sz >= bound);
}
```

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**)*

Ongoing and Future Opportunities on SVF

- **Mobile Security**: Android malware detection by combing program analysis and hybrid representation learning (Mr. Yanxin Zhang).
- **Program Analysis for Incomplete code**: Analysis for programs in the presence of incomplete code (Mr. Mahamad).
- **Program Repair**: Precise program repair using static taint analysis (Mr. Mingshan Jia).
- **Secure Machine Learning**: Program analysis to discover vulnerabilities in ML libraries, which may cause adversarial attacks and misclassifications.

Thanks!

Q & A

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(){          void foo(){

cs1: foo();           cs3: fork(t1, bar);
cs2: foo();               }

}
```

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();

}
```

```
void foo(){

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

```
void main(){

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

}
```

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

t is multi-forked thread

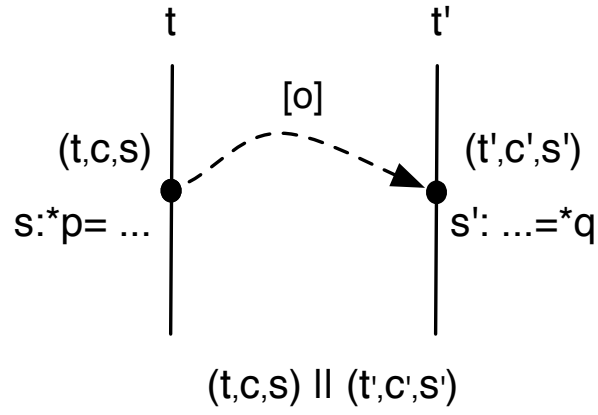
A thread *t* always refers to a context-sensitive fork site, i.e., a unique runtime thread unless $t \in \mathcal{M}$ is *multi-forked*, in which case, t may represent more than one runtime thread.

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

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



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.

Context-sensitive Thread 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

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

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

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

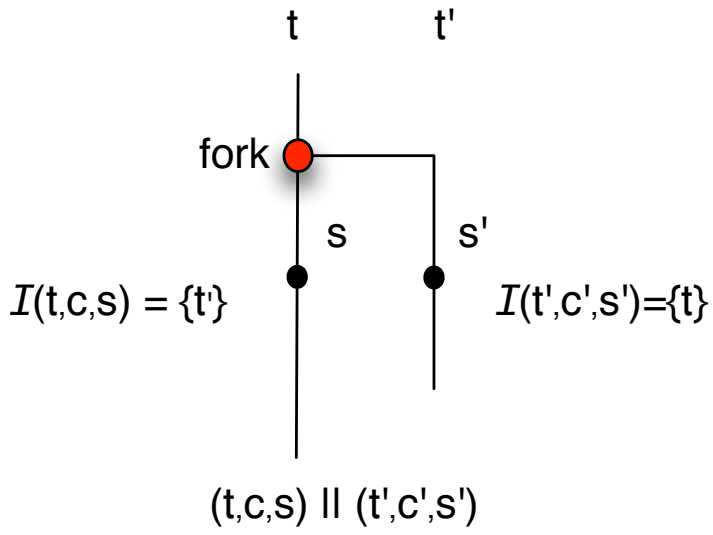
$$\texttt{[I-CALL]}\quad \frac{(t,c,l) \xrightarrow{call_i} (t,c',l') \quad c' = c.\mathit{push}(i)}{\mathcal{I}(t,c,l) \subseteq \mathcal{I}(t,c',l')}$$

$$\texttt{[I-INTRA]}\quad \frac{(t,c,l) \rightarrow (t,c,l')}{\mathcal{I}(t,c,l) \subseteq \mathcal{I}(t,c,l')}$$

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

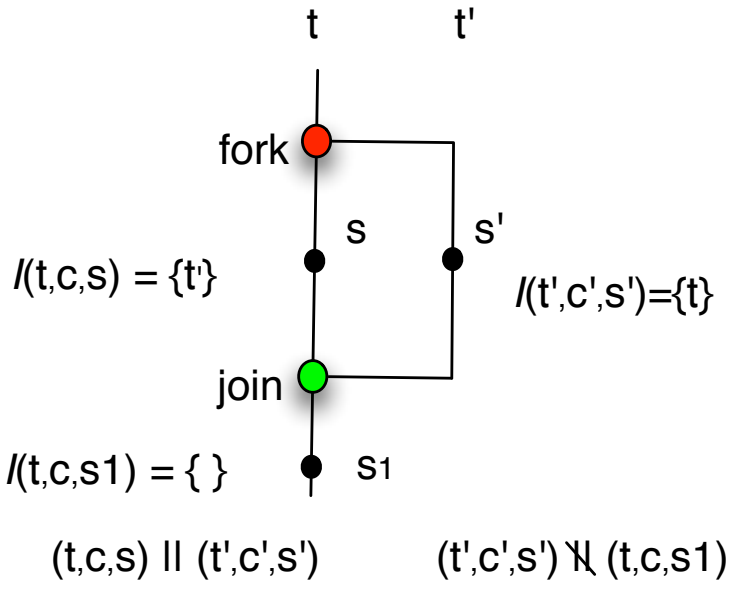
Interleaving Analysis Rule (CGO '16)

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



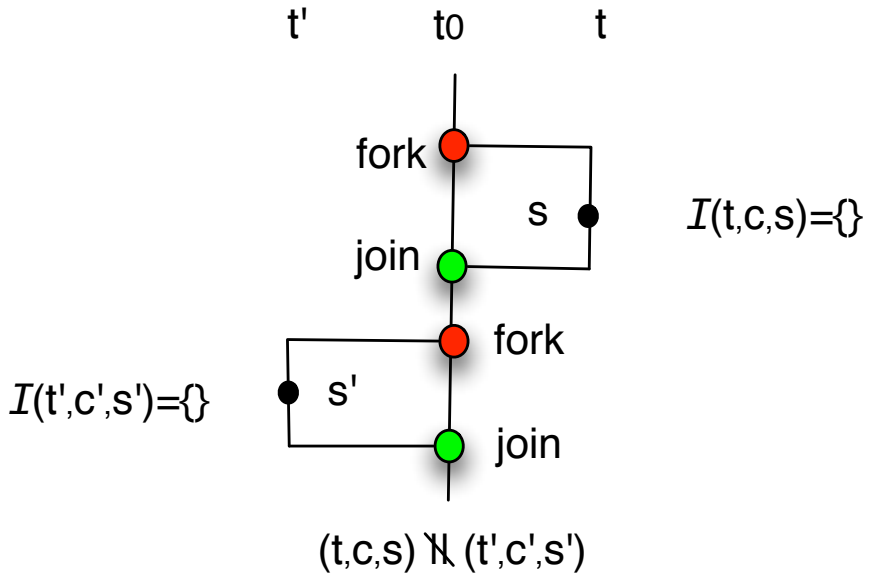
Interleaving Analysis Rule (CGO '16)

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



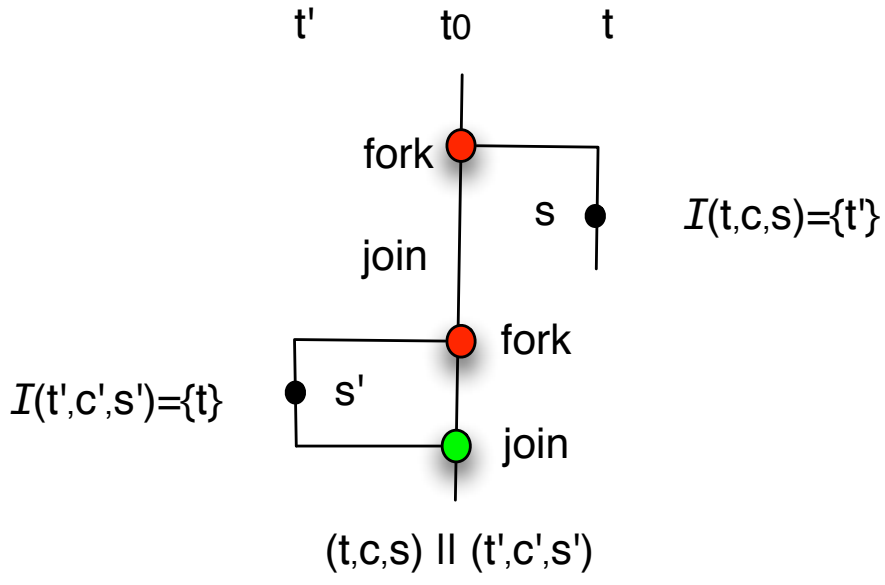
Interleaving Analysis Rule (CGO '16)

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



Interleaving Analysis Rule (CGO '16)

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



Lock Analysis (CGO '16)

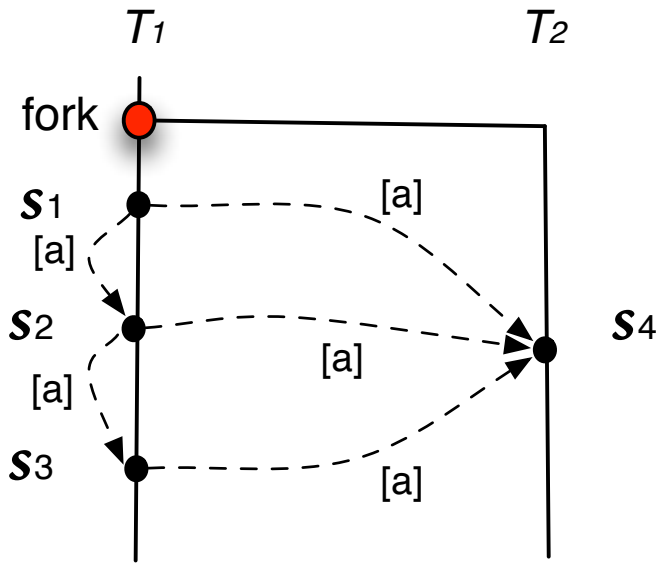
Statements from different mutex regions are interference-free if these regions are protected by a common lock.

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)
}
```