Static 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

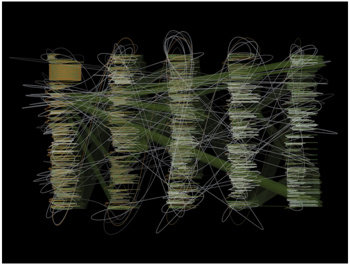
February 28, 2019

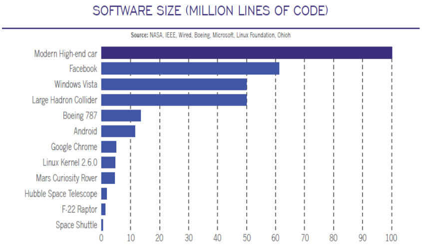
Outline

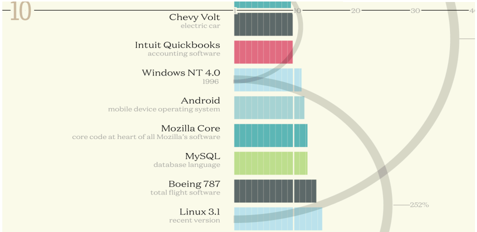
- Existing software bugs and vulnerabilities
- Static analysis and dynamic analysis
- Technical contributions in developing scalable and precise program analysis
  - Fundamental program analysis
    - Sparse value-flow analysis
    - Selective value-flow analysis
    - On-demand value-flow analysis
  - Applications
    - Value-flow analysis for detecting memory errors
- Research opportunities

Modern System Software

– Extremely Large and Complex







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.

```
/* CVE-2012-0817 allows remote attackers to cause a denial of service through adversarial connection requests.*/
/* Samba --libads/ldap.c:ads_leave_realm */.

host = memAlloc(hostname);
 ...
 if (...)  {...;  return ADS_ERROR_SYSTEM(ENOENT);} // The programmer forgot to release host on error.

```

```
/* A memory leak in Php-5.5.11 */

for (...) {
    char* buf = readBuffer();
     if (condition)
        printf (buf);
    else
       continue;  // buf is leaked in else branch
    freeBuf(buf);
}

```

Buffer Overflow

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

```
/* A simplified example from "Young and Mchugh, IEEE S&P 1987", exploited by attackers to bypass verfication*/

void verfiyPassword(){
  char buff[15]; int pass = 0;
  printf ("\n Enter the password : \n");
  gets(buff);

  if (strcmp(buff, "thegeekstuff")){     // return non-zero if the two strings do not match
      printf ("\n Wrong Password \n");
  }
  else{                                  // return zero if two strings matched or a buffer overrun
      printf ("\n Correct Password \n");
      pass = 1;
  }
  if (pass)
      printf ("\n Root privileges given to the user \n");
}
```

[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

```c
/* An uninitialized variable vulnerability simplified from gnuplot, CVE-2017-9670 */

void load(){
	switch (ctl) {
		case -1:
			xN = 0; yN = 0;
			break;
		case 0:
			xN = i; yN = -i;
			break;
		case 1:
			xN = i + NEXT_SZ; yN = i - NEXT_SZ;
			break;
		default:
			xN = -1; xN = -1; // xN is accidentally set twice while yN is uninitialized
			break;
	}
		plot(xN, yN);
}
```

Use-After-Free

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

```
/* CVE-2015-6125 and CVE-2018-12377 with similar heap use after free patterns*/

char* msg = memAlloc(...);
...
if (err) {
        abrt = 1;
         ....
        free(msg);
}
...
if (abrt) {
         ...
        logError("operation aborted before commit", msg); // try to access released heap variable
}
```

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 value-flow analysis
    - On-demand value-flow analysis
  - Applications
    - Value-flow analysis for detecting memory errors
- **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

```
p = x

p = y
```

flow-sensitivity

at which
program point
p is tainted?

```
foo(x)      foo(y)
  call        call
   foo(p){

   }
```

context-sensitivity

under which
calling context
p is tainted?

```
if(cond)
      p = x
else
      p = y
```

path-sensitivity

along which
program path
p is tainted?

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

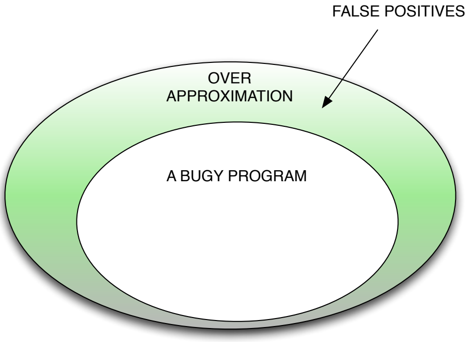
p = x[i]
Observe_and_check (&x, i)

Check against self-maintained runtime meta-info

Limited Coverage

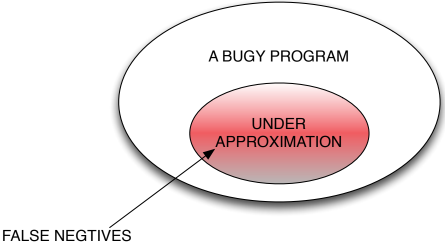
```
if(hard_to_satisfy)
        x=*p          // a null dereference
 else
        x= *q         // a safe dereference
```

Bug Detection Philosophy



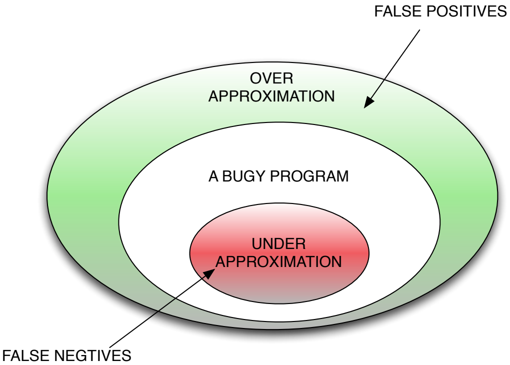
- Soundness : Over-Approximation (Static Analysis)
- Completeness : Under-Approximation (Dynamic Analysis)

Bug Detection Philosophy



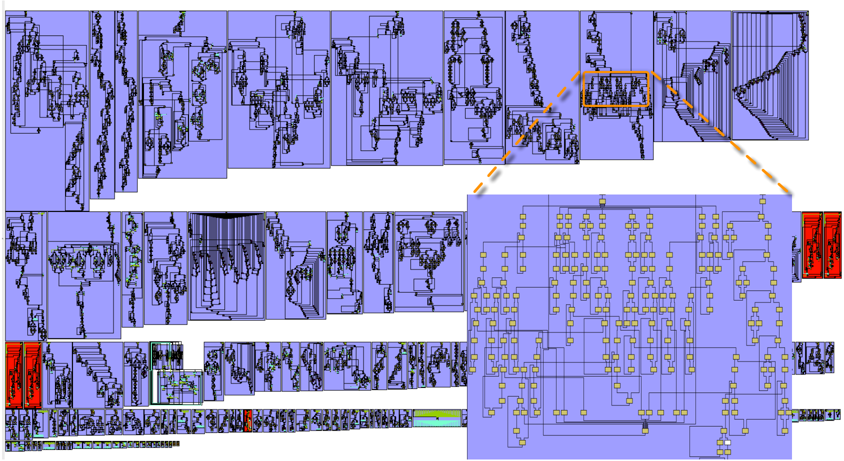
- Soundness : Over-Approximation (Static Analysis)
- Completeness : Under-Approximation (Dynamic Analysis)

Bug Detection Philosophy



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

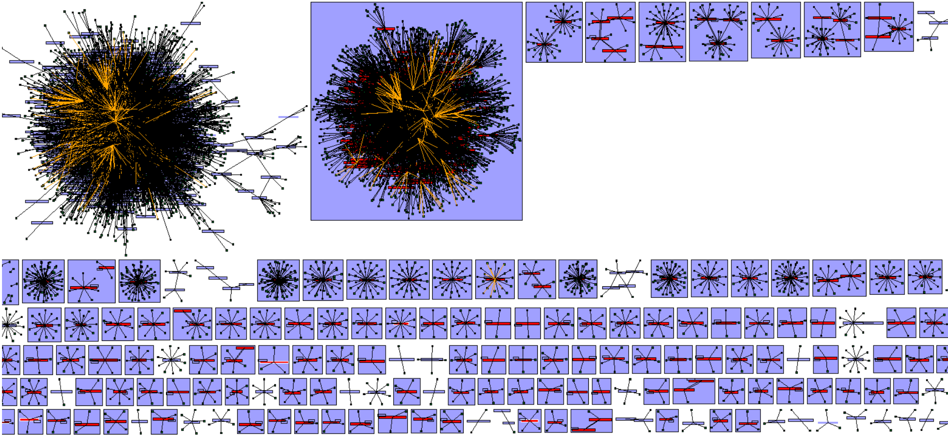
Whole-Program CFG of 300.twolf (20.5KLOC)



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

Costly to reason about flow of values on CFGs!

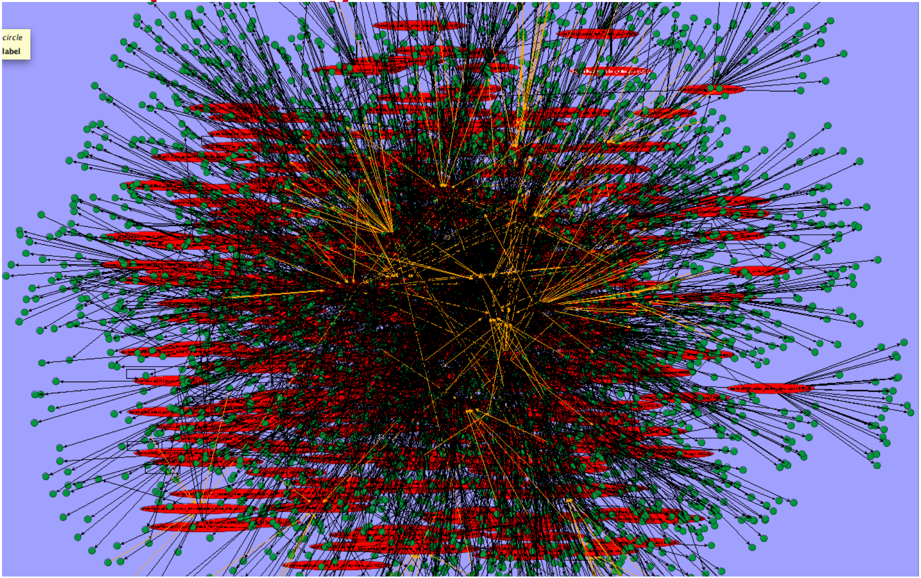
Call Graph of 176.gcc (230.5KLOC)



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

Costly to reason about flow of values on CFGs!

Call Graph of 176.gcc



Research Aim

Developing fundamental and practical **program 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 analysis* (CC'16, CGO '16, ICSE '18,)
  - *Selective analysis* (SAS '14, LCTES '16, TECS '18)
  - *On-demand analysis* (FSE '16, ICSE '18, TSE '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, selective and 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, selective and on-demand interprocedural program dependence analysis framework for both sequential and multithreaded programs.

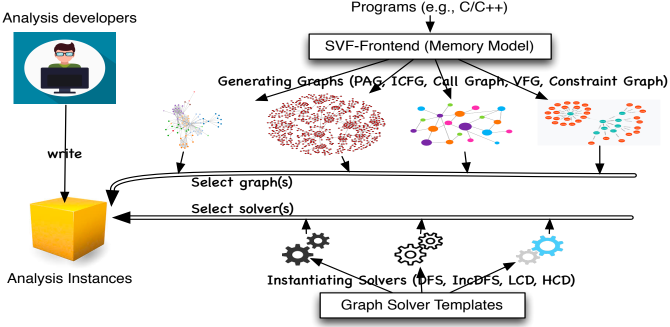
- The SVF project
  - Started since early 2014, actively maintained. **Publicly available** at : `http://svf-tools.github.io/SVF`.
  - Implemented on top of LLVM compiler (the latest version 7.0.0) with over 100KLOC C/C++ code and **200+ stars** on Github.
  - Invited for a **plenary talk in EuroLLVM 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, selective and on-demand interprocedural program dependence analysis framework for both sequential and multithreaded programs.

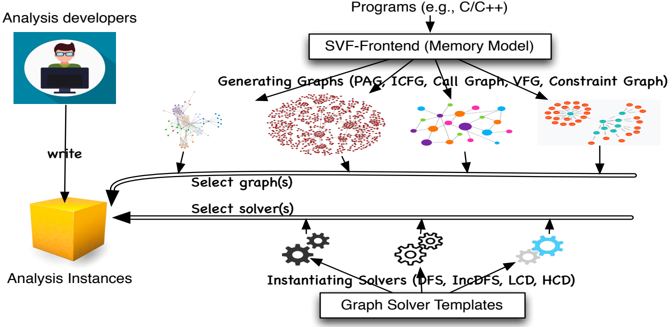
- The SVF project
  - Started since early 2014, actively maintained. **Publicly available** at : `http://svf-tools.github.io/SVF`.
  - Implemented on top of LLVM compiler (the latest version 7.0.0) with over 100KLOC C/C++ code and **200+ stars** on Github.
  - Invited for a **plenary talk in EuroLLVM 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
  - Selective : support mixed analyses for precision and efficiency trade-offs.
  - On-demand : reason about program parts based on user queries.

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

SVF

Pointer Analysis and Program Depedence Analysis in LLVM

View Wiki on GitHub | Download Source Code | Download Virtual Machine Image

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
    - Selective value-flow analysis
    - On-demand value-flow analysis
  - Applications
    - Value-flow analysis for detecting memory errors
- **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
```

$p \rightarrow a$

$a \rightarrow b, c$

$q \rightarrow b, c$

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**

Program	Points-to
p = & a	$p \to a$
*p = & b	$a \to b, c$
*p = & c	$q \to b, c$
q = *p	

Flow-insensitive analysis

Program	Points-to		
p = & a			
	$p \to a$		
*p = & b			
	$p \to a$	$a \to b$	
*p = & c			
	$p \to a$	$a \to c$	
q = *p			
	$p \to a$	$a \to c$	$q \to c$

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.**

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

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

Sparse Flow-Sensitive Analysis (ISSTA '12, TSE'14, CC'16)

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

$\boxed{x = \&m}$

x → m

$\boxed{p = \&a}$

p → a ~~x → m~~

$\boxed{*p = \&b}$

p → a a → b ~~x → m~~

$\boxed{*p = \&c}$

~~p → a~~ a → c x → m

$\boxed{*x = \&d}$

~~p → a~~ ~~a → c~~ x → m m → d

$\boxed{y = *x}$

p → a ~~a → c~~ ~~x → m~~ m → d y → d

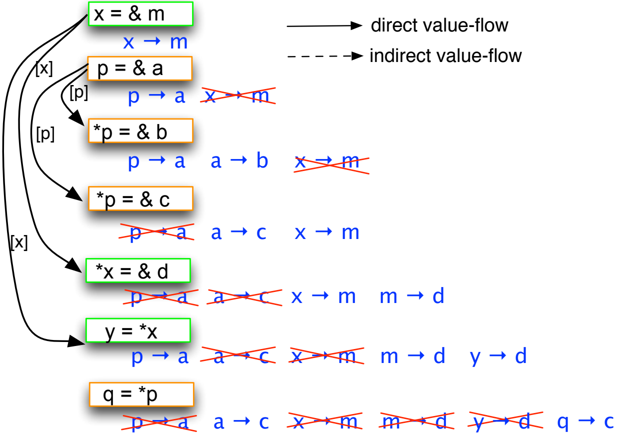
$\boxed{q = *p}$

~~p → a~~ a → c ~~x → m~~ ~~m → d~~ ~~y → d~~ q → c

Data-flow-based flow-sensitive analysis

Sparse Flow-Sensitive Analysis (ISSTA '12, TSE'14, CC'16)

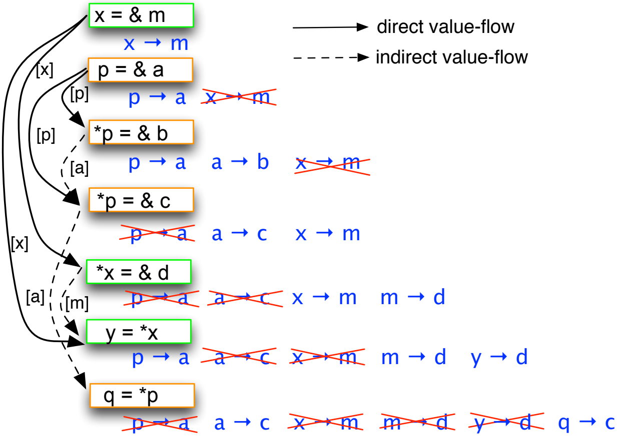
- **Propagate points-to information only along pre-computed def-use chains (a.k.a value-flows) instead of control-flow**



Sparse flow-sensitive analysis

Sparse Flow-Sensitive Analysis (ISSTA '12, TSE'14, CC'16)

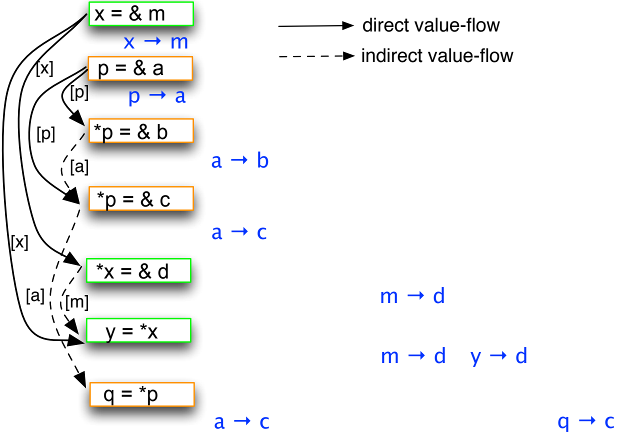
- **Propagate points-to information only along pre-computed def-use chains (a.k.a value-flows) instead of control-flow**



Sparse flow-sensitive analysis

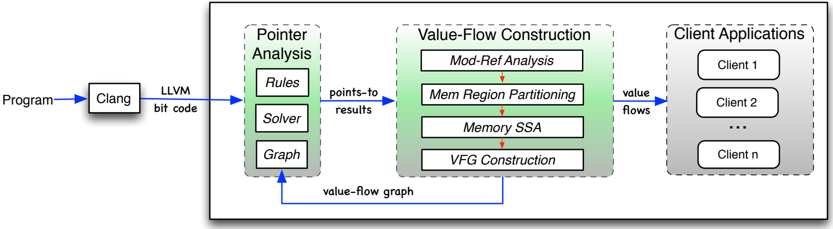
Sparse Flow-Sensitive Analysis (ISSTA '12, TSE'14, CC'16)

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



Sparse flow-sensitive analysis

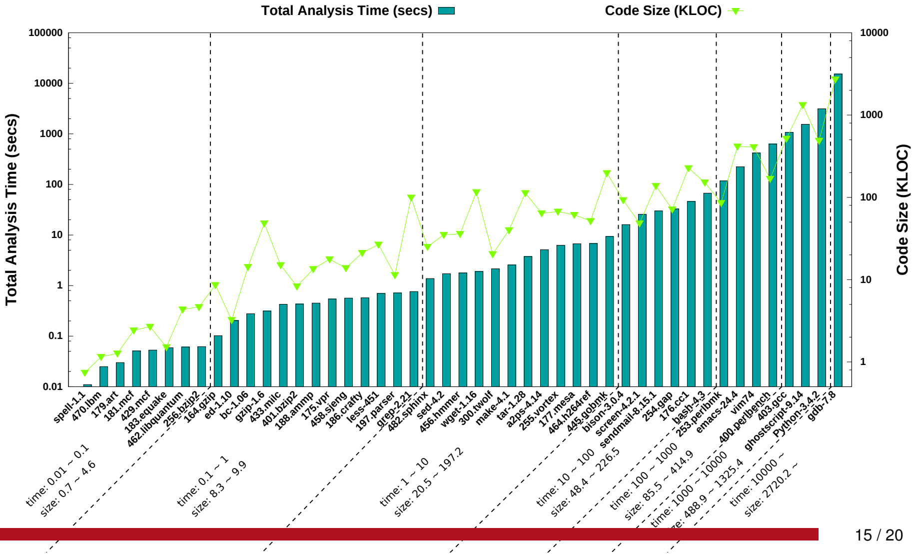
Evaluation and Results



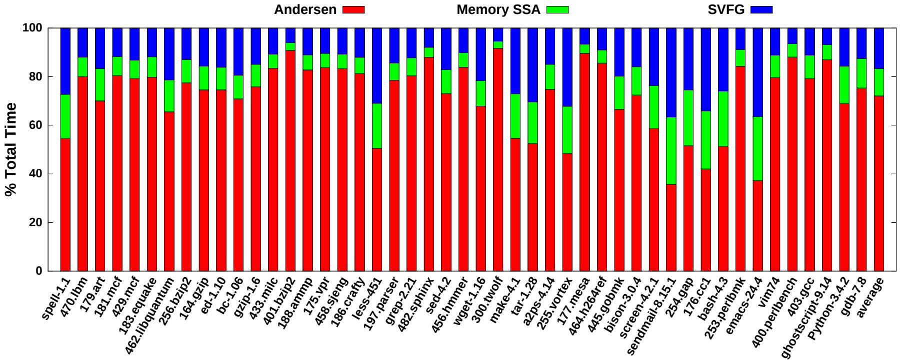
- Benchmarks:
  - All SPEC C benchmarks: 15 programs from CPU2000 and 12 programs from CPU2006
  - 20 Open-source applications: most of them are recent released versions.
  - Total lines of code evaluated: 8,005,872 LOC with maximum program size 2,720,279 LOC
- Machine setup:
  - Ubuntu Linux 3.11.0-15-generic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

Analysis Time

Total Analysis Time = Andersen + MemorySSA + VFG

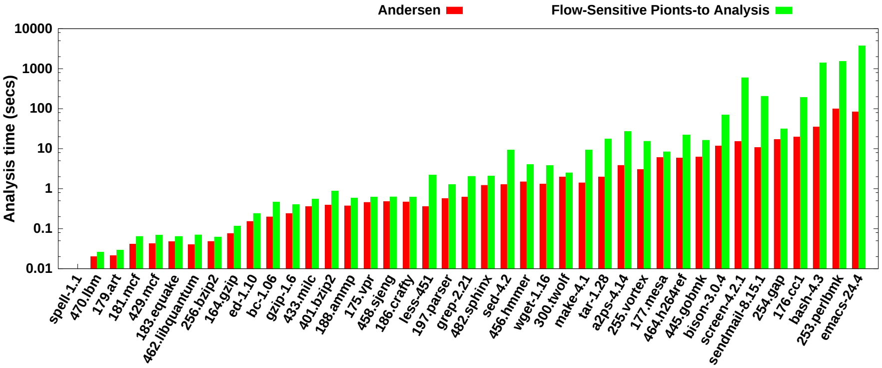


Analysis Time Distribution



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

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



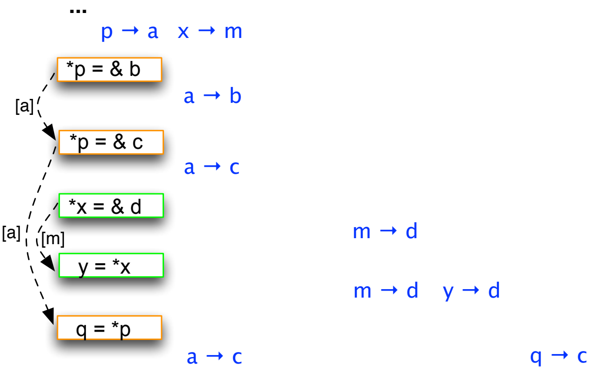
Flow-Sensitive Analysis Slowdowns: From 1.2$\times$ to 44$\times$. On average 6.5$\times$.

Outline

- **Existing software bugs and vulnerabilities**
- **Static analysis and dynamic analysis**
- **Technical contributions**
  - Fundamental program analysis
    - Sparse value-flow analysis
    - Selective value-flow analysis
    - On-demand value-flow analysis
  - Applications
    - Value-flow analysis for detecting memory errors
- **Research opportunities**

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

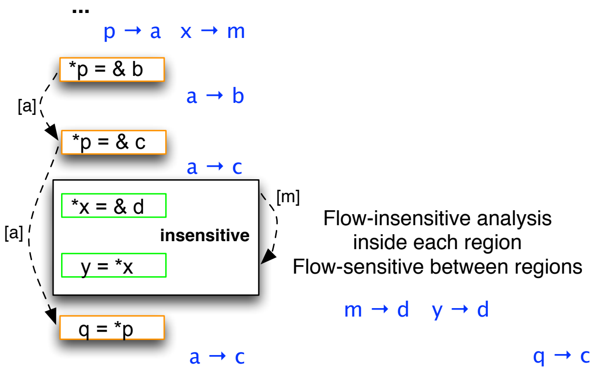
- **Hybrid flow-insensitive and sensitive pointer analysis that operate on the regions partitioned from a program**



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**

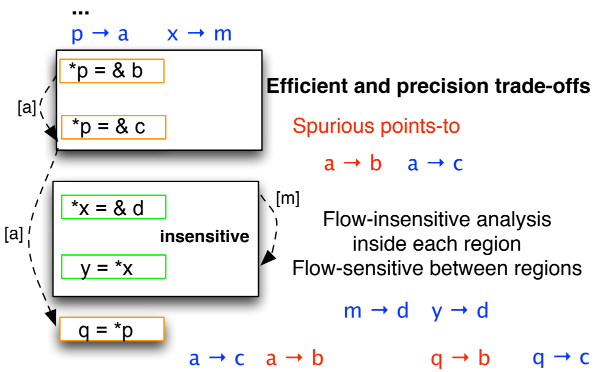


Flow-insensitive analysis
inside each region
Flow-sensitive between regions

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

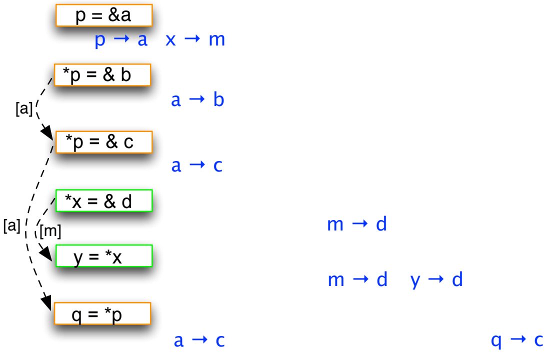
Load-precision-preserving analysis is **2X** faster than sparse analysis.

Outline

- **Existing software bugs and vulnerabilities**
- **Static analysis and dynamic analysis**
- **Technical contributions**
  - Fundamental program analysis
    - Sparse value-flow analysis
    - Selective value-flow analysis
    - On-demand value-flow analysis
  - Applications
    - Value-flow analysis for detecting memory errors
- **Research opportunities**

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

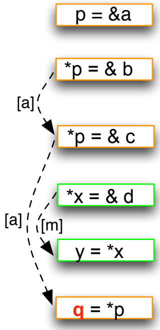
- **Demand-driven flow-sensitive analysis with strong updates via CFL-Reachability**



Sparse flow-sensitive analysis

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

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

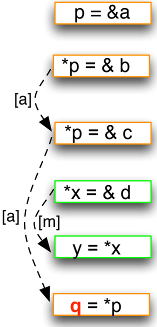


Query: points-to value of q ?

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**



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

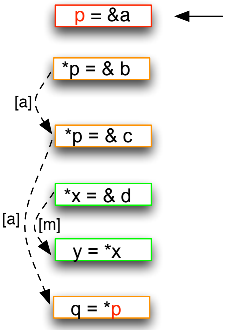
$$\overline{\textit{\textbf{flowsTo}}} \rightarrow (\overline{\text{assign}} \mid \text{load } \textit{\textbf{alias}} \text{ store})^{*} \;\text{alloc}$$

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

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

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

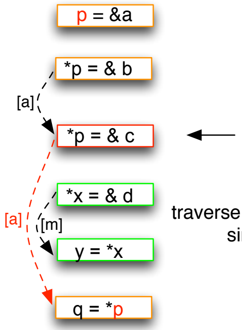
- **Demand-driven flow-sensitive analysis with strong updates via CFL-Reachability**



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**

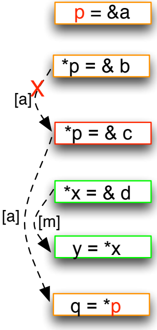


traverse against value-flow of a since p points to a

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

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

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

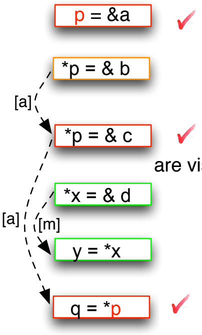


Strong updates
since *p refers only a
No need to back ward traversal !

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)

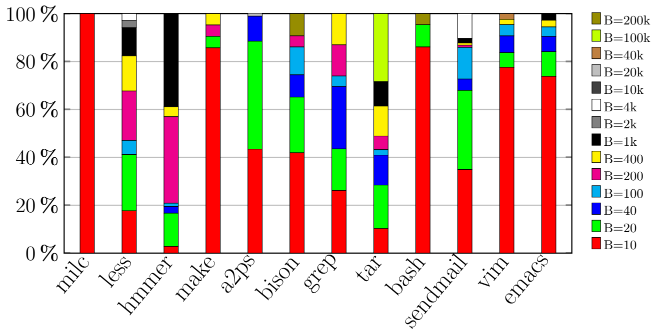


Figure: Percentage of queried variables proved to be safe (initialized) by demand-driven analysis over whole-program analysis under different budgets

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

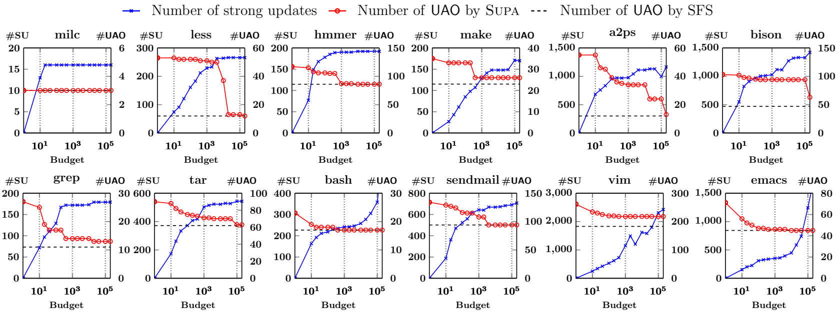


Figure: Correlating the number of strong updates with the number of unintialized variables detected under different budgets

Outline

- **Existing software bugs and vulnerabilities**
- **Static analysis and dynamic analysis**
- **Technical contributions**
  - Fundamental program analysis
    - Sparse value-flow analysis
    - On-demand value-flow analysis
    - Value-flow analysis for multithreaded programs
  - Applications
    - Value-flow analysis for detecting memory errors
- **Research opportunities**

Value-Flow Analysis for Memory Error Detection

- **Memory Leak Detection** (ISSTA'12 and TSE'14)
  - context-free reachability problem on sparse value-flow graph
  - report 40.7% more bugs than the fastest one with a slightly higher false positive rate but is only 3.7X slower

Value-Flow Analysis for Memory Error Detection

- **Memory Leak Detection** (ISSTA'12 and TSE'14)
  - context-free reachability problem on sparse value-flow graph
  - report 40.7% more bugs than the fastest one with a slightly higher false positive rate but is only 3.7X slower
- **Use-after-free Detection** (ACSAC'17 and ICSE'18)
  - spatio-temporal correlation problem and its context reduction
  - validated with 10 open-source applications (3+ MLOC) with 7 CVE bug found
- **Uninitialized Variable Detection** (CGO'14 and FSE'16)
  - reason about the definedness of values along sparse value-flow graph to remove redundant instrumentation
  - reduce the overhead of Google's Memory Sanitizer from 302% to 123%

Value-Flow Analysis for Memory Error Detection

- **Memory Leak Detection** (ISSTA'12 and TSE'14)
  - context-free reachability problem on sparse value-flow graph
  - report 40.7% more bugs than the fastest one with a slightly higher false positive rate but is only 3.7X slower
- **Use-after-free Detection** (ACSAC'17 and ICSE'18)
  - spatio-temporal correlation problem and its context reduction
  - validated with 10 open-source applications (3+ MLOC) with 7 CVE bug found
- **Uninitialized Variable Detection** (CGO'14 and FSE'16)
  - reason about the definedness of values along sparse value-flow graph to remove redundant instrumentation
  - reduce the overhead of Google's Memory Sanitizer from 302% to 123%
- **Buffer Overflow Detection** (ISSRE'14 and TRel'16)
  - eliminate expensive runtime checks inside loops through static weakest precondition analysis
  - reduce the runtime overhead of SOFTBOUND from 77% to 47%

Value-Flow Analysis for Memory Error Detection

- **Memory Leak Detection** (ISSTA'12 and TSE'14)
  - context-free reachability problem on sparse value-flow graph
  - report 40.7% more bugs than the fastest one with a slightly higher false positive rate but is only 3.7X slower
- **Use-after-free Detection** (ACSAC'17 and ICSE'18)
  - spatio-temporal correlation problem and its context reduction
  - validated with 10 open-source applications (3+ MLOC) with 7 CVE bug found
- **Uninitialized Variable Detection** (CGO'14 and FSE'16)
  - reason about the definedness of values along sparse value-flow graph to remove redundant instrumentation
  - reduce the overhead of Google's Memory Sanitizer from 302% to 123%
- **Buffer Overflow Detection** (ISSRE'14 and TRel'16)
  - eliminate expensive runtime checks inside loops through static weakest precondition analysis
  - reduce the runtime overhead of SOFTBOUND from 77% to 47%
- **Protecting Control-Flow Integrity** (ISSTA'17 and ACISP'18)
  - pointer analysis to identify and remove spurious call targets by class hierarchy analysis to raise the bar against code reuse attacks.
  - reduce the sets of legitimate targets permitted at 20.3% of the virtual callsites in Chrome

Ongoing and Future Opportunities on SVF

- **Automatic Program Repair**: Value-flow-guided precise automatic program repair.
- **Partial/Compositional Pointer Analysis**: Analysis for programs in the presence of incomplete code.
- **Machine-learning-guided static analysis**: enable machine learning (e.g., deep representation learning) to boost pointer analysis for analysing large-scale programs.

Thanks!

Q & A