✓ Artifact evaluated by FSE

On-Demand Strong Update Analysis via Value-Flow Refinement

Yulei Sui and Jingling Xue

School of Computer Science and Engineering
The University of New South Wales
2052 Sydney Australia

Nov. 16, 2016

UNSW THE UNIVERSITY OF NEW SOUTH WALES

Contributions

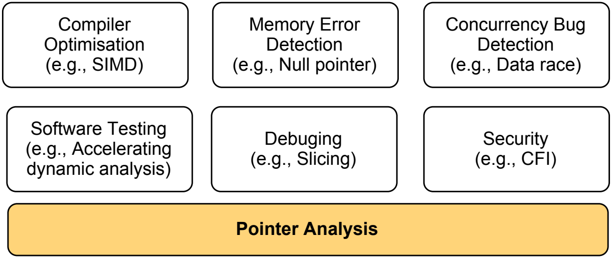
- Demand-driven pointer analysis with strong updates for C/C++ programs.
- Hybrid multi-stage analysis framework to performs strong update analysis precisely by refining imprecisely pre-computed value-flows away.
- Small analysis time and memory budgets (0.19 seconds and 36KB of memory per points-to query, on average).

Outline

- Background and Motivation
- Our approach: SUPA
- Experimental Results and Evaluation

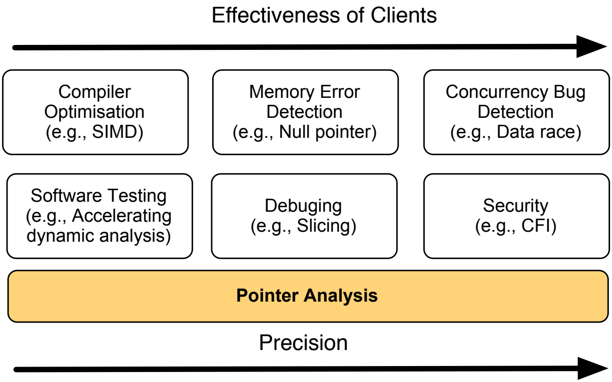
Pointer Analysis

- Statically approximate runtime values of a pointer
- A fundamental enabling technology for many clients.



Pointer Analysis

- Statically approximate runtime values of a pointer
- A fundamental enabling technology for many clients.



Flow-Sensitive Analysis with Strong Updates

- Key feature of flow-sensitivity to boost the precision of pointer analysis

--Strong updates: *overwrite contents of an abstract memory object with a new value.*

--Weak updates: *add new values to the existing values of an abstract object.*

Pointer Analysis

Precision

Strong Updates

Flow-Sensitive Analysis with Strong Updates

Flow-Insensitive Pointer Analysis

Ignore program execution order, i.e., a single solution across whole program.

Flow-Sensitive Pointer Analysis

Respect program control-flow, i.e., a separate solution at each program point.

```
L1:  p = &a;
L2:  q = p;
L3: *p = &b;
L4: *q = &c
L5:  r = *p;
```

Flow-insensitive analysis

Flow-Sensitive Analysis with Strong Updates

Flow-Insensitive Pointer Analysis

Ignore program execution order, i.e., a single solution across whole program.

Flow-Sensitive Pointer Analysis

Respect program control-flow, i.e., a separate solution at each program point.

L1: p = &a;

L2: q = p;

L3: *p = &b;

L4: *q = &c

L5: r = *p;

$p \rightarrow a$

$q \rightarrow a$

$a \rightarrow b, c$

$r \rightarrow b, c$

Flow-insensitive analysis

Flow-Sensitive Analysis with Strong Updates

Flow-Insensitive Pointer Analysis

Ignore program execution order, i.e., a single solution across whole program.

Flow-Sensitive Pointer Analysis

Respect program control-flow, i.e., a separate solution at each program point.

L1: p = &a;

$p \rightarrow a$

L2: q = p;

$p \rightarrow a \quad q \rightarrow a$

L3: *p = &b;

$p \rightarrow a \quad q \rightarrow a \quad a \rightarrow b$

L4: *q = &c

$p \rightarrow a \quad q \rightarrow a \quad a \rightarrow b \quad a \rightarrow c$

L5: r = *p;

$p \rightarrow a \quad q \rightarrow a \quad a \rightarrow b \quad a \rightarrow c \quad r \rightarrow b \quad r \rightarrow c$

Flow-sensitive analysis **without** strong updates

Flow-Sensitive Analysis with Strong Updates

Flow-Insensitive Pointer Analysis
Ignore program execution order, i.e., a single solution across whole program.

Flow-Sensitive Pointer Analysis
Respect program control-flow, i.e., a separate solution at each program point.

L1: p = &a;
p → a
L2: q = p;
p → a   q → a
L3: *p = &b;
p → a   q → a   a → b
L4: *q = &c
p → a   q → a   ~~a → b~~   a → c
L5: r = *p;
p → a   q → a   a → b   a → c   ~~r → b~~   r → c

**q refers to a single runtime memory location*

Strong updates for a

Flow-sensitive analysis **with** strong updates

Flow-Sensitive Analysis with Strong Updates

Flow-Insensitive Pointer Analysis

Ignore program execution order, i.e., a single solution across whole program.

Flow-Sensitive Pointer Analysis with strong updates

Respect program control-flow, i.e., a separate solution at each program point.

L1: p = &a;

p → a

L2: q = p;

p → a q → a

L3: *p = &b;

p → a q → a a → b

L4: *q = &c

p → a q → a ~~a → b~~ a → c

L5: r = *p;

p → a q → a a → b a → c ~~r → b~~ r → c

Flow-sensitive analysis **with** strong updates

spurious data dependence and points-to relation may cause false alarms in bug detectors

Flow- and Context-Sensitive Strong Updates

Flow-Insensitive Pointer Analysis

Ignore program execution order, i.e., a single solution across whole program.

Flow-Sensitive Pointer Analysis

Respect program control-flow, i.e., a separate solution at each program point.

```
bar(){

if(..) {
        p = &a;
       *p = &b;
cs1:  x = foo(p);    x → b, d
} else {
        p = &c
}
cs2:  y = foo(p);    y → b, d

}
```

Flow-Sensitive Analysis

```
foo(p){

   *p = &d;

 return *p;

}
```

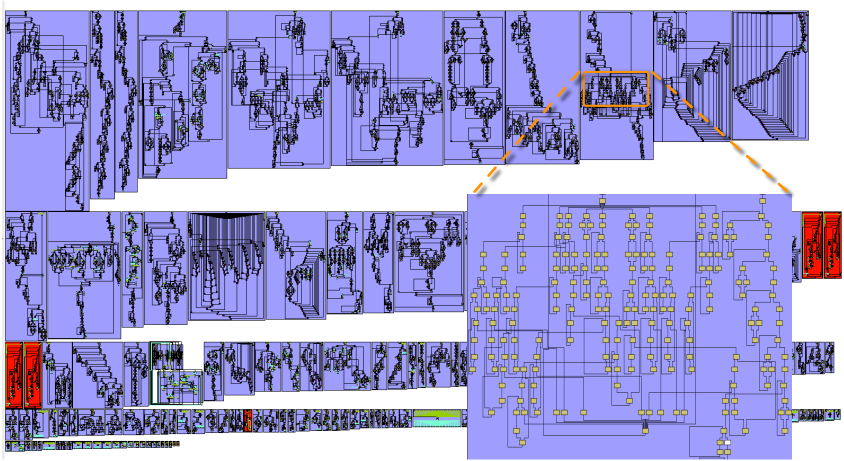
*p refers to unique object a under *cs1*

Strong updates for a ***under context [cs1]***

Weak updates for both a ***and*** b ***under context [cs2]***

*p refers to two objects a and b under *cs2*

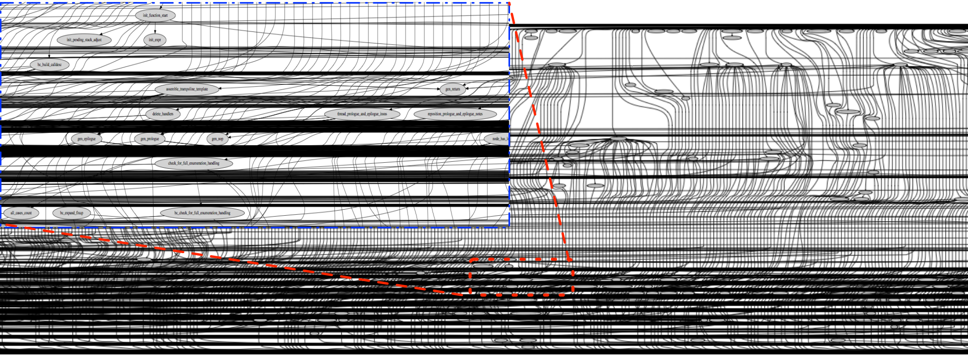
Whole-Program CFG of 300.twolf (20.5KLOC)



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

Costly to reason about whole-program control-flows!

Call Graph of emacs-24.4 (431.9KLOC)

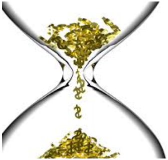


#functions: 3938 #pointers: 754746 #loads/stores: 52781

Costly to reason about whole-program calling contexts!

Limitations of Existing Strong Update Analyses

Whole-program



Time consuming

Single Analysis Choice:

Single analysis choice
for all queries

Existing Strong Update Analyses

Gap

Practical Strong Update Analysis

Demand-Driven

Analysis with Budgets

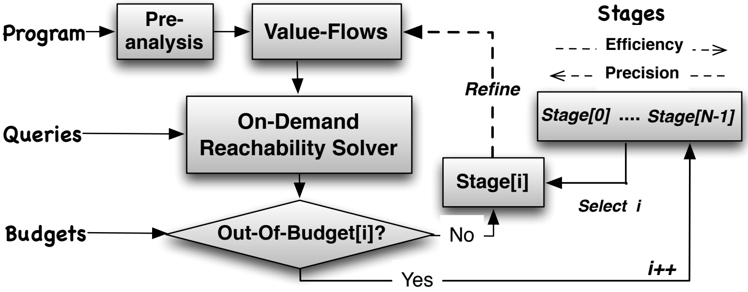
Multiple Analysis Choices:

Multiple analysis choices
for different queries

Outline

- Background and Motivation
- Our approach: SUPA
- Experimental Results and Evaluation

SUPA: On-Demand Strong-UPdate Analysis



- Pre-computed value-flows (def-use)
- Backward CFL-reachability analysis on value-flow graph under analysis budgets (value-flow edges traversed)
- Multi-stages include FSCS, FSCI and FICI stages.

An Example

```
L1:   p = &a

L2:   q = p;

L3:   *p = &b

L4:   *q = &c

L5:   r = *p;
```

points-to query pts(r)?

An Example

points-to query pts(r)?

L1: p = &a

p → a

L2: q = p;

p → a   q → a

L3: *p = &b

p → a   q → a   a → b

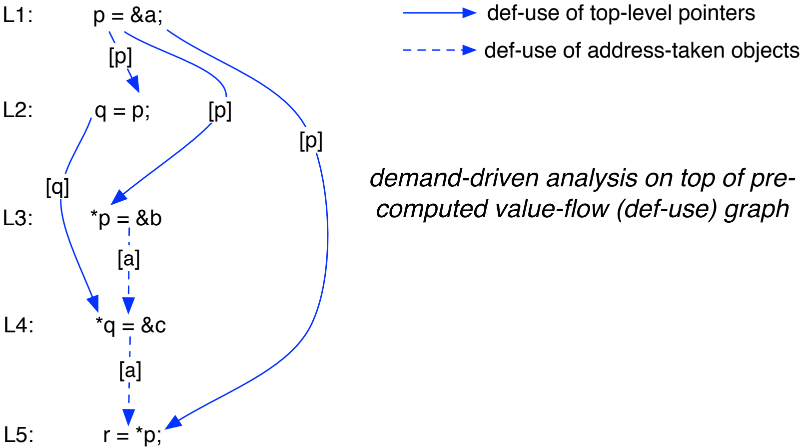
L4: *q = &c

p → a   q → a   a → b   a → c

L5: r = *p;

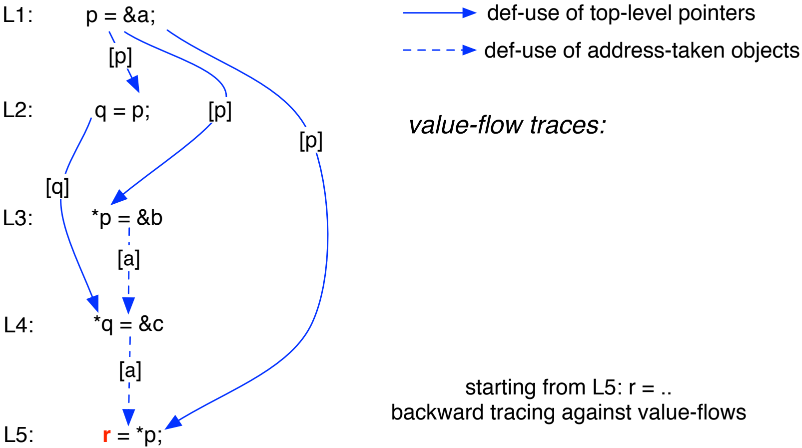
p → a   q → a   a → b   a → c   r → b   r → c

An Example



demand-driven analysis on top of pre-computed value-flow (def-use) graph

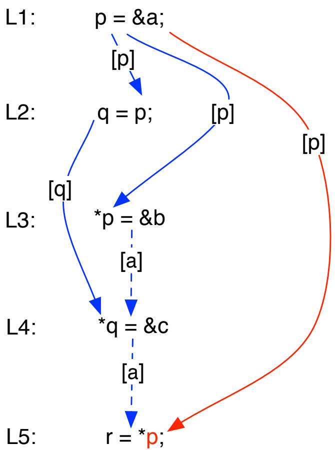
An Example

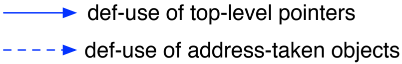


value-flow traces:

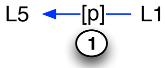
starting from L5: r = ..
backward tracing against value-flows

An Example

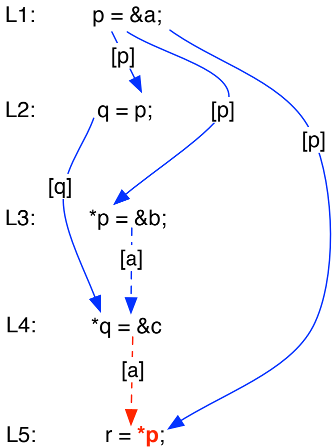




value-flow traces:



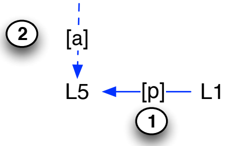
An Example



def-use of top-level pointers

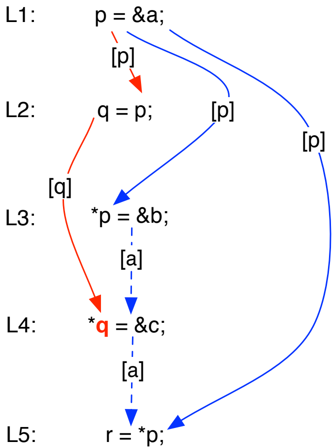
def-use of address-taken objects

value-flow traces:



*p refers to object a

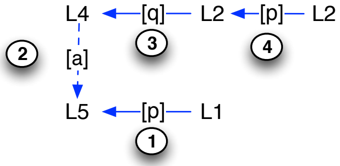
An Example



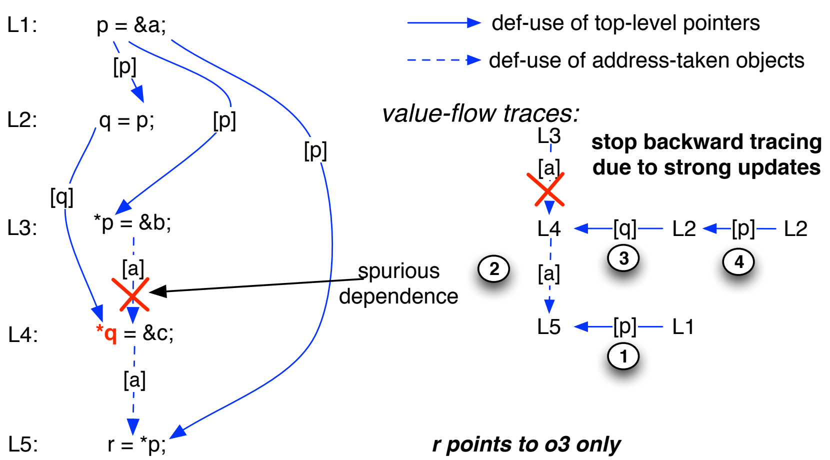
def-use of top-level pointers

def-use of address-taken objects

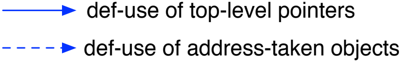
value-flow traces:



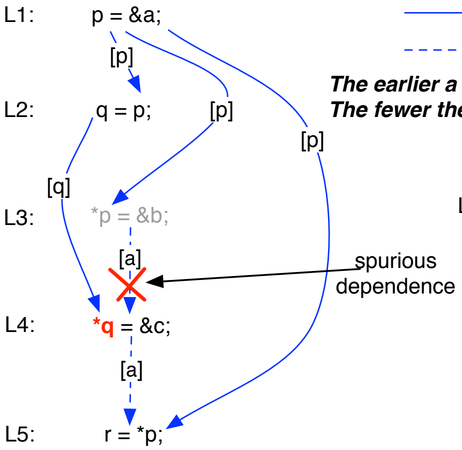
An Example


L1: p = &a;
L2: q = p;
L3: *p = &b;
L4: *q = &c;
L5: r = *p;
[p]
[p]
[p]
[q]
[a]
[a]
spurious dependence
def-use of top-level pointers
def-use of address-taken objects
value-flow traces:
L3
[a]
stop backward tracing due to strong updates
L4
[q]
L2
[p]
L2
3
4
2
[a]
L5
[p]
L1
1
r points to o3 only

An Example



The earlier a strong update is performed,
The fewer the number of statements is traversed.

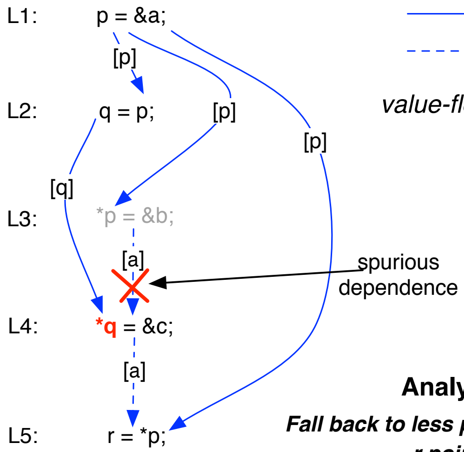


L3 does not need to be analyzed
during backward reachability

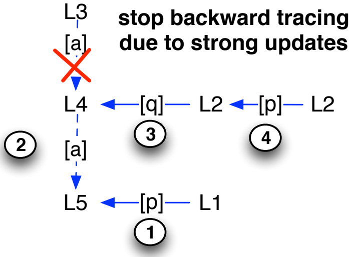
An Example

def-use of top-level pointers

def-use of address-taken objects



value-flow traces:



Analysis Budget < 4 ?

Fall back to less precise analysis e.g., Andersen's
r points to both b and c

Flow- and Context-Sensitive Strong Updates

- Every statement is parameterized additionally by a context i.e., a sequence of callsites.
- CFL-reachability on top of value-flow graph by matching calls and returns.
- Strong updates on singleton heap objects (objects with concrete contexts, not involved in recursion or loops).

Outline

- Background and Motivation
- Our approach: SUPA
- Experimental Results and Evaluation

Evaluation

- Implementation:
  - Implemented on top of our previous open-source tool SVF (http://unsw-corg.github.io/SVF/) (CC '16)
  - Core implementation of SUPA is round 5,000 LOC C++ code.
  - Field-sensitivity and on-the-fly call graph construction.

[1] *Ben Hardekopf and Calvin Lin*, Flow-sensitive pointer analysis for millions of lines of code CGO '11

Evaluation

- Implementation:
  - Implemented on top of our previous open-source tool SVF (http://unsw-corg.github.io/SVF/) (CC '16)
  - Core implementation of SUPA is round 5,000 LOC C++ code.
  - Field-sensitivity and on-the-fly call graph construction.
- Methodology
  - One major client, uninitialized pointer detection (add a special tainted object (UAO) pointed by every stack and heap objects at their allocation sites).
  - SUPA v.s. SFS[1]

[1] *Ben Hardekopf and Calvin Lin*, Flow-sensitive pointer analysis for millions of lines of code CGO '11

Evaluation

- Implementation:
  - Implemented on top of our previous open-source tool SVF (http://unsw-corg.github.io/SVF/) (CC '16)
  - Core implementation of SUPA is round 5,000 LOC C++ code.
  - Field-sensitivity and on-the-fly call graph construction.
- Methodology
  - One major client, uninitialized pointer detection (add a special tainted object (UAO) pointed by every stack and heap objects at their allocation sites).
  - SUPA v.s. SFS[1]
- Benchmarks:
  - 12 open-source programs, nine recently released applications, such as `make`, `bash`,`sendmail`,`vim`, and `emacs`.
- Machine setup:
  - Ubuntu Linux 3.11 Intel Xeon Quad Core, 3.7GHZ, 64GB

[1] *Ben Hardekopf and Calvin Lin*, Flow-sensitive pointer analysis for millions of lines of code CGO '11

Benchmarks

Table: Program characteristics

Program	KLOC	Statements	Pointers	Alloc Sites	Queries
milc-v6	15	11713	29584	865	3
less-451	27.1	6766	22835	1135	100
hmmer-2.3	36	27924	74689	1472	2043
make-4.1	40.4	14926	36707	1563	1133
a2ps-4.14	64.6	49172	116129	3625	5065
bison-3.0.4	113.3	36815	90049	1976	4408
grep-2.21	118.4	10199	33931	1108	562
tar-1.28	132	30504	85727	3350	909
bash-4.3	155.9	59442	191413	6359	5103
sendmail-8.15	259.9	86653	256074	7549	2715
vim-7.4	413.1	147550	466493	8960	6753
emacs-24.4	431.9	189097	754746	12037	4438
Total	1807.6	670761	2158377	49999	33232

Analysis Precision

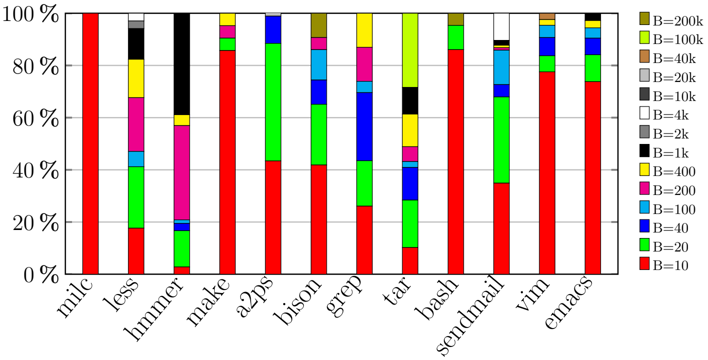
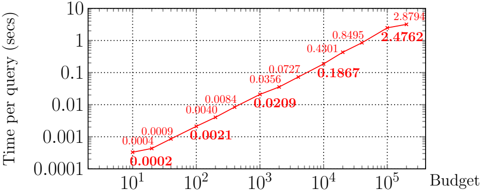


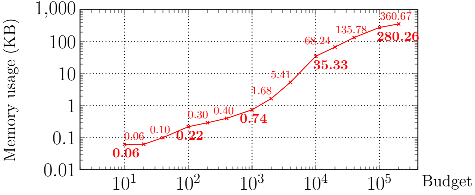
Figure: Percentage of queried variables proved to be initialized by SUPA over SFS under different budgets

SUPA answers correctly **97%** of all the queries as SFS under **10K** budget per query, and **the same** precision as SFS when increasing the budget to **200K**.

Analysis Time and Memory Usage



(a) Analysis Time



(b) Memory Usage

SUPA consumes about 0.19 seconds and 36KB of memory per query, on average (with a budget of 10000 value-flows traversed).

Precision

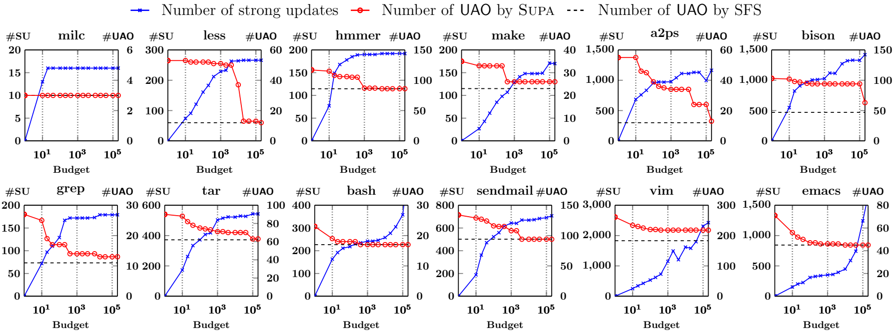


Figure: Correlating the number of strong updates with the number of UAO's under different budgets.

Context-Sensitive Results

Table: Average analysis times and UAO's reported by SUPA-FSCS (with a budget of 10000 in each stage) and SUPA-FSCI (with a budget of 10000 in total)

Program	SUPA-FSCI		SUPA-FSCS	
	Time (ms)	#*UAO*	Time (ms)	#*UAO*
milc	0.02	3	14.52	**0**
less	15.15	37	92.41	37
hmmer	11.41	86	135.05	**71**
make	124.40	26	229.44	26
a2ps	126.01	34	448.26	**32**
bison	465.54	94	529.20	**86**
grep	124.46	14	197.66	14
tar	26.31	70	83.10	**68**
bash	188.69	17	327.16	17
sendmail	200.32	94	250.19	**85**
vim	168.67	218	473.25	218
emacs	159.22	45	222.65	45

Conclusion

- Demand-driven pointer analysis with strong updates for C/C++ programs.
- Hybrid multi-stage analysis framework to performs strong update analysis precisely by refining imprecisely pre-computed value-flows away.
- Small analysis time and memory budgets (0.19 seconds and 36KB of memory per points-to query, on average).

✓ Artifact evaluated by FSE

Full replication package is publicly available online:
`http://www.cse.unsw.edu.au/~corg/supa/`

Thanks!

Q & A

Backup Slides: Pre-analysis and SFS Time

Table: Pre-processing times taken by pre-analysis shared by SUPA and SFS and analysis times of SFS (in seconds)

Program	Pre-Analysis Times Shared by SUPA and SFS			Analysis Time of SFS
	Andersen's Analysis	SVFG	Total	
milc	0.42	0.1	0.52	0.16
less	0.42	0.37	0.79	1.94
hmmer	1.57	0.46	2.03	1.07
make	1.74	1.17	2.91	13.94
a2ps	7.34	1.31	8.65	60.61
bison	8.18	3.66	11.84	44.16
grep	1.44	0.17	1.61	2.39
tar	2.73	1.71	4.44	12.27
bash	53.48	44.07	97.55	2590.69
sendmail	24.05	23.43	47.48	348.63
vim	445.88	85.69	531.57	13823
emacs	135.93	146.94	282.87	8047.55

Case Studies

```
    // symtab.c
114 static
115 void symbols_sort(symbol **first, symbol **second) {
      ...
119   symbol* tmp = *first;
120   *first  =  *second;

121   *second  =  tmp;
      ...
123 }

623 static void
624 user_token_number_redeclaration(...) {
      ...
627   symbols_sort (&st, &nd);
      ...
628   complain_indent (&nd->location, ...);

635 }
```

SU for nd

Query

pt(⟨ℓ628 ,*nd->location*⟩)

(a) Code snippet from bison-3.0.4

```
    // mark.c
68  static struct mark* getmark(int c){
72    register struct mark *m;  static struct mark sm;
75    switch (c) {
77    case '^':
81      m = &sm;
        ...
84      m->m_ifile  =  curr_ifile;
85      break;
108   case '\'':
112     m = &marks[LASTMARK];
113     break;
127   }
128   return (m);
129 }
179 public void gomark(int c) {
186     m = getmark(c);
208     if (m->m_ifile) { ...}
218 }
```

SU for sm.m_ifile

Query

pt(⟨ℓ208 ,*m->m_ifile*⟩)

(b) Code snippet from less-4.5.1

```
    //io_lat4.c
93  int qcdhdr_get_str(char *s, QCDheader *hdr, char **q) {
98    *q  =  (*hdr).value[i];
104 }
113 int qcdhdr_get_int(char *s,QCDheader *hdr,int *q) {
114   char *p;
115   qcdhdr_get_str(s,hdr,&p);
117   sscanf(p,"%d",...);
119 }
120 int qcdhdr_get_int32x(char *s,QCDheader *hdr,...) {
121   char *p;
123   qcdhdr_get_str(s,hdr,&p);
125   sscanf(p,"%x",...);
128 }
129 int qcdhdr_get_double(char *s, QCDheader *hdr, ...) {
130   char *p;
131   qcdhdr_get_str(s,hdr,&p);
133   sscanf(p,"%lf",...);
135 }
```

SU for q

Query

pt(⟨ℓ117 ,*p*⟩)

(c) Code snippet from milc-v6

```
     //argp-help.c
434  static struct hol * make_hol (...) {
442    struct hol *hol = malloc (sizeof (struct hol)); // Obj
501    return hol;
502  }
849  static void hol_append (struct hol *hol, ...) {
934    hol->short_options  =  short_options;
939  }
1386 static struct hol * argp_hol (...) {
1390   struct hol *hol = make_hol (argp, cluster);
1401   hol_append(hol, ... );
1405 }
1588 static void _help (...)
1617   hol = argp_hol (argp, 0);
1664   hol_usage (hol, fs);
1727 }
1346 static void hol_usage (struct hol *hol, ...) {
1353   strlen(hol->short_options);
1382 }
```

SU for Obj.short_options

Query

pt(⟨ℓ1353 ,*hol->short_options*⟩)

(d) Code snippet from tar-1.28