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
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.

<table>
<thead>
<tr>
<th>Compiler Optimisation (e.g., SIMD)</th>
<th>Memory Error Detection (e.g., Null pointer)</th>
<th>Concurrency Bug Detection (e.g., Data race)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Software Testing (e.g., Accelerating dynamic analysis)</td>
<td>Debugging (e.g., Slicing)</td>
<td>Security (e.g., CFI)</td>
</tr>
</tbody>
</table>

Pointer Analysis
Pointer Analysis

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

Effectiveness of Clients

- Compiler Optimisation (e.g., SIMD)
- Memory Error Detection (e.g., Null pointer)
- Concurrency Bug Detection (e.g., Data race)
- Software Testing (e.g., Accelerating dynamic analysis)
- Debugging (e.g., Slicing)
- Security (e.g., CFI)

Pointer Analysis

Precision
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.

---

**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: \( p = \&a; \)  \( p \rightarrow a \)
L2: \( q = p; \)  \( q \rightarrow a \)
L3: \( *p = \&b; \)  \( a \rightarrow b, c \)
L4: \( *q = \&c \)  \( r \rightarrow b, c \)
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: \( p = &a; \)

\[ p \rightarrow a \]

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

spurious data dependence and points-to relation may cause false alarms in bug detectors
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;
     q → a
L3:  *p = &b;
     p → a  q → a  a → b
L4:  *q = &c
     p → a  q → a  a → c
     r = *p;
     p → a  q → a  a → b  a → c  r → c
     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.

```c
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
Ignored 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 \rightarrow b, d
  } else{
    p = &c
  }
  cs2: y = foo(p); y \rightarrow b, d
}

foo(p)
{
  *p = &d;
  return *p;
}
```

*\textit{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]

*\textit{p} refers to two objects a and b under cs2

Flow-Sensitive Analysis
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

Existing Strong Update Analyses

Gap

Practical Strong Update Analysis

Single Analysis Choice:

Single analysis choice for all queries

Multiple Analysis Choices:

Multiple analysis choices for different queries

Demand-Driven

Analysis with Budgets
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.

Diagram:

1. Program → Pre-analysis
2. Value-Flows
3. On-Demand Reachability Solver
4. Out-Of-Budget[i]? Yes/No
5. Stages: Stage[0] ... Stage[N-1]
6. Budgets: No/Yes
7. Select i
8. i++
9. Refine Efficiency → Precision

Program → Pre-analysis → Value-Flows → On-Demand Reachability Solver → Out-Of-Budget[i]? → Stages (Stage[0] ... Stage[N-1]) → Select i → i++ → Refine Efficiency → Precision
An Example

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

points-to query pts(r)?
An Example

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: \( r = *p; \)
\[ p \rightarrow a \quad q \rightarrow a \quad a \rightarrow b \quad a \rightarrow c \quad r \rightarrow b \quad r \rightarrow c \]
An Example

FSE 2016, November 16th, Seattle
value-flow traces:
starting from L5: r = ..
backward tracing against value-flows

p = &a;
q = p;
*p = &b
*q = &c
r = *p;

Starting from L5:
- p = &a;
- q = p;
- *p = &b
- *q = &c
- r = *p;

Fall back to less precise analysis e.g., Andersen's

The fewer the number of statements is traversed.
L3 does not need to be analyzed
during backward reachability

Analysis Budget < 4 ?

L5:
- p = &a;
- q = p;
- *p = &b
- *q = &c
- r = *p;

L3 does not need to be analyzed

L1:
- p = &a;

L2:
- q = p;

L4:
- *q = &c

L5:
- r = *p;

def-use of top-level pointers
def-use of address-taken objects

value-flow traces:

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

fall back to less precise analysis e.g., Andersen's

r points to both b and c
r points to o3 only
p = &a;
q = p;
*p = &b
*q = &c
r = *p;

def-use of top-level pointers
def-use of address-taken objects

*p refers to object a

The earlier a strong update is performed,
dependence
spurious
dependence
spurious
dependence

fall back to less precise analysis e.g., Andersen's

L5:
- p = &a;
- q = p;
- *p = &b
- *q = &c
- r = *p;

L3 does not need to be analyzed
during backward reachability

L1:
- p = &a;

L2:
- q = p;

L3:
- *p = &b

L4:
- *q = &c

L5:
- r = *p;

def-use of top-level pointers
def-use of address-taken objects

*p refers to object a

The earlier a strong update is performed,
An Example

\[
\begin{align*}
\text{L1:} & \quad p = \&a; \\
\text{L2:} & \quad q = p; \\
\text{L3:} & \quad *p = \&b \\
\text{L4:} & \quad *q = \&c \\
\text{L5:} & \quad r = *p;
\end{align*}
\]

**value-flow traces:**

\[
\text{L1} \rightarrow \text{L2} \rightarrow \text{L3} \rightarrow \text{L4} \rightarrow \text{L5} \]

\[
\text{points-to query } \text{pts}(r)\?
\]

\[
p \rightarrow a
q \rightarrow a
* \rightarrow b
q \rightarrow a
a \rightarrow c
\]

**def-use of top-level pointers**

**def-use of address-taken objects**

Fall back to less precise analysis e.g., Andersen’s
An Example

L1: \[ p = &a; \]

L2: \[ q = p; \]

L3: \[ *p = &b; \]

L4: \[ *q = &c \]

L5: \[ r = *p; \]

\textbf{value-flow traces:}

- \textbf{def-use of top-level pointers}
- \textbf{def-use of address-taken objects}

*p refers to object a
An Example

L1: \( p = \&a; \)

L2: \( q = p; \)

L3: \( *p = \&b; \)

L4: \( *q = \&c; \)

L5: \( r = *p; \)

value-flow traces:

def-use of top-level pointers

def-use of address-taken objects

Fall back to less precise analysis e.g., Andersen's

The fewer the number of statements is traversed.

L3 does not need to be analyzed during backward reachability

Analysis Budget < 4 ?

p = &a;

q = p;

*p = &b

*q = &c

r = *p;

L1:

L2:

L3:

L4:

L5:

points-to query pts(r)?

p → a

q → a

a → b

q → a

a → c

r → b

r → c

L1:

L2:

L3:

L4:

L5:
An Example

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

[p]  
[q]  
[a]  
[a]  
[p]  

def-use of top-level pointers

def-use of address-taken objects

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

value-flow traces:

[p] L1
1  2
[a]  [a]  
L3 [a]
L4 [q]  L2 [p]  L2
L5 [p]  L1

spurious dependence

stop backward tracing due to strong updates

r points to o3 only

The fewer the number of statements is traversed.
The earlier a strong update is performed,
spurious dependence
Fall back to less precise analysis e.g., Andersen's
r points to both b and c
L3 does not need to be analyzed
during backward reachability
p → a
q → a
p → a
q → a
p → a
q → a
p → a
q → a
r → b
r → c

FSE 2016, November 16th, Seattle
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

spurious dependence

Fall back to less precise analysis e.g., Andersen’s

The fewer the number of statements is traversed.

Analysis Budget < 4 ?

p = &a;
q = p;
*p = &b;
*q = &c;
r = ... 

L1:
L2:
L3:
L4:
L5:

[p]
[p]
[p]
[a]
[a]
[r]
[r]
[r]
An Example

L1: \( p = &a; \)

L2: \( q = p; \)

L3: \( *p = &b; \)

L4: \( *q = &c; \)

L5: \( r = *p; \)

---

**value-flow traces:**

\( p = &a; \)

\( q = p; \)

\( *p = &b; \)

\( *q = &c; \)

---

**def-use of top-level pointers**

**def-use of address-taken objects**

---

**stop backward tracing due to strong updates**

**spurious dependence**

---

**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. SFS¹

---

¹Ben 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

- 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**

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

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)

<table>
<thead>
<tr>
<th>Program</th>
<th>SUPA-FSCI</th>
<th>SUPA-FSCS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time (ms)</td>
<td>#UAO</td>
</tr>
<tr>
<td>milc</td>
<td>0.02</td>
<td>3</td>
</tr>
<tr>
<td>less</td>
<td>15.15</td>
<td>37</td>
</tr>
<tr>
<td>hmmer</td>
<td>11.41</td>
<td>86</td>
</tr>
<tr>
<td>make</td>
<td>124.40</td>
<td>26</td>
</tr>
<tr>
<td>a2ps</td>
<td>126.01</td>
<td>34</td>
</tr>
<tr>
<td>bison</td>
<td>465.54</td>
<td>94</td>
</tr>
<tr>
<td>grep</td>
<td>124.46</td>
<td>14</td>
</tr>
<tr>
<td>tar</td>
<td>26.31</td>
<td>70</td>
</tr>
<tr>
<td>bash</td>
<td>188.69</td>
<td>17</td>
</tr>
<tr>
<td>sendmail</td>
<td>200.32</td>
<td>94</td>
</tr>
<tr>
<td>vim</td>
<td>168.67</td>
<td>218</td>
</tr>
<tr>
<td>emacs</td>
<td>159.22</td>
<td>45</td>
</tr>
</tbody>
</table>
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).
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)

<table>
<thead>
<tr>
<th>Program</th>
<th>Program</th>
<th>Pre-Analysis Times</th>
<th>Analysis Time of SFS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Shared by SUPA and SFS</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Andersen’s Analysis</td>
<td>SVFG</td>
</tr>
<tr>
<td>milc</td>
<td>0.42</td>
<td>0.1</td>
<td>0.52</td>
</tr>
<tr>
<td>less</td>
<td>0.42</td>
<td>0.37</td>
<td>0.79</td>
</tr>
<tr>
<td>hmmner</td>
<td>1.57</td>
<td>0.46</td>
<td>2.03</td>
</tr>
<tr>
<td>make</td>
<td>1.74</td>
<td>1.17</td>
<td>2.91</td>
</tr>
<tr>
<td>a2ps</td>
<td>7.34</td>
<td>1.31</td>
<td>8.65</td>
</tr>
<tr>
<td>bison</td>
<td>8.18</td>
<td>3.66</td>
<td>11.84</td>
</tr>
<tr>
<td>grep</td>
<td>1.44</td>
<td>0.17</td>
<td>1.61</td>
</tr>
<tr>
<td>tar</td>
<td>2.73</td>
<td>1.71</td>
<td>4.44</td>
</tr>
<tr>
<td>bash</td>
<td>53.48</td>
<td>44.07</td>
<td>97.55</td>
</tr>
<tr>
<td>sendmail</td>
<td>24.05</td>
<td>23.43</td>
<td>47.48</td>
</tr>
<tr>
<td>vim</td>
<td>445.88</td>
<td>85.69</td>
<td>531.57</td>
</tr>
<tr>
<td>emacs</td>
<td>135.93</td>
<td>146.94</td>
<td>282.87</td>
</tr>
</tbody>
</table>
// 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 }

(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;
77 switch (c) {
81 case ' ': ...
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 }

(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) {
115 char *p;
117 sscanf(p, "%d", ...);
119 }
120 int qcdhdr_get_int32x(char *s,QCDheader *hdr,...) {
122 char *p;
123 sscanf(p, "%x", ...);
128 }
129 int qcdhdr_get_double(char *s, QCDheader *hdr, ...) {
130 char *p;
131 sscanf(p, "%lf", ...);
135 }

(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 }

(d) Code snippet from tar-1.28