Towards a Globally Scalable Semantics-based Static Analysis

Kwangkeun Yi

School of Computer Science & Engineering
Seoul National University

11/15/2010 @ CSE, HKUST
(co-work with my students and postdocs)
A Fundamental Challenge in SW

How can we reduce/eliminate errors in software?
- statically: before execution, before sell/embed
- automatically: against explosive sw size
- to find bugs or verify their absence

Our approach:
- “semantics-based static analysis”
- *abstract interpretation, type system, program logics, programming language theories*
- and hodge-podge of engineering
Our Goal

R&D of static software analysis tools:

“SW MRI” “SW fMRI” “SW PET”
Our Activities

We publicize our works in:

- POPL('06, '11), CAV('08), VMCAI('10, '11), ICSE('11), SAS, OOPSLA, FSE, etc.
- ACM TOPLAS, TCS, JFP, SP&E, Acta Informatica, etc.

A commercialization:

Research areas: *static analysis*, *abstract interpretation*, *programming language theory*, *type system*, & *whatever relevant*
Talk Outline

1. Short overview of SPARROW: scalable yet non-global bug finder for C
2. New localization technology for scalable & global SPARROW
3. Concluding remark
Motivation

- prove by ourselves that static analysis is “useful in real world”

Status

- shallow property, full automation, scalable
  - C’s buffer overrun, memory leak, null deref., / by zero, · · ·
  - for non domain-specific C code
- detection ratio: \( \sim 6/\text{KLOC} \), speed: \( \sim 100\text{Loc/sec} \)

From the elusive three (deep property, scalability, automation)

- deep property
- domain-independence
Towards a Globally Scalable Semantics-based Static Analysis
Customers

Domestic market at the moment

- electronics companies: Samsung, LG, ···
- network switching system sw developers
- other embedded sw developers
- bank system sw developers
- etc.

Complementing others (Coverity, GrammaTech, Klockworks, Polyspace).

- BMT at a site (a network device OS, ∼ 700KLOC):
**Sparrow** is a one-button solution with four steps:
### Performance Numbers (1/2)

Memory leak detection (SPEC2000 and open sources)

<table>
<thead>
<tr>
<th>Programs</th>
<th>Size KLOC</th>
<th>Time (sec)</th>
<th>True Alarms</th>
<th>False Alarms</th>
</tr>
</thead>
<tbody>
<tr>
<td>art</td>
<td>1.2</td>
<td>0.68</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>equake</td>
<td>1.5</td>
<td>1.03</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>mcf</td>
<td>1.9</td>
<td>2.77</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>bzip2</td>
<td>4.6</td>
<td>1.52</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>gzip</td>
<td>7.7</td>
<td>1.56</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>parser</td>
<td>10.9</td>
<td>15.93</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ammp</td>
<td>13.2</td>
<td>9.68</td>
<td>20</td>
<td>0</td>
</tr>
<tr>
<td>vpr</td>
<td>16.9</td>
<td>7.85</td>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td>crafty</td>
<td>19.4</td>
<td>84.32</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>twolf</td>
<td>19.7</td>
<td>68.80</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>mesa</td>
<td>50.2</td>
<td>43.15</td>
<td>9</td>
<td>0</td>
</tr>
<tr>
<td>vortex</td>
<td>52.6</td>
<td>34.79</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>gap</td>
<td>59.4</td>
<td>31.03</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>gcc</td>
<td>205.8</td>
<td>130.33</td>
<td>44</td>
<td>1</td>
</tr>
<tr>
<td>gnu-5.07</td>
<td>17.8</td>
<td>9.44</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>tc8.4.14</td>
<td>17.9</td>
<td>266.09</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>hanterm-3.16</td>
<td>25.6</td>
<td>13.66</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>sed-4.0.8</td>
<td>26.8</td>
<td>13.68</td>
<td>29</td>
<td>31</td>
</tr>
<tr>
<td>tar-1.13</td>
<td>28.3</td>
<td>13.88</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>grep-2.5.1a</td>
<td>31.5</td>
<td>22.19</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>openssh-3.5p1</td>
<td>36.7</td>
<td>10.75</td>
<td>18</td>
<td>4</td>
</tr>
<tr>
<td>bison-2.3</td>
<td>48.4</td>
<td>48.60</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>openssh-4.3p2</td>
<td>77.3</td>
<td>177.31</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>fftw-3.1.2</td>
<td>184.0</td>
<td>15.20</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>httpd-2.2.2</td>
<td>316.4</td>
<td>102.72</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>net-snmp-5.4</td>
<td>358.0</td>
<td>201.49</td>
<td>40</td>
<td>20</td>
</tr>
<tr>
<td>binutils-2.13.1</td>
<td>909.4</td>
<td>712.09</td>
<td>228</td>
<td>25</td>
</tr>
</tbody>
</table>
## Performance Numbers (2/2)

Buffer overrun detection (SPEC2000 and open sources)

<table>
<thead>
<tr>
<th>Programs</th>
<th>Size KLOC</th>
<th>Time (sec)</th>
<th>True Alarms</th>
<th>False Alarms</th>
</tr>
</thead>
<tbody>
<tr>
<td>art</td>
<td>1.2</td>
<td>0.45</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>equake</td>
<td>1.5</td>
<td>2.89</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>mcf</td>
<td>1.9</td>
<td>0.33</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>bzip2</td>
<td>4.6</td>
<td>10.90</td>
<td>23</td>
<td>29</td>
</tr>
<tr>
<td>gzip</td>
<td>7.7</td>
<td>3.38</td>
<td>18</td>
<td>24</td>
</tr>
<tr>
<td>parser</td>
<td>10.9</td>
<td>260.94</td>
<td>4</td>
<td>13</td>
</tr>
<tr>
<td>twolf</td>
<td>19.7</td>
<td>8.59</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ammp</td>
<td>13.2</td>
<td>10.20</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>vpr</td>
<td>16.9</td>
<td>11.15</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>crafty</td>
<td>19.4</td>
<td>139.80</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>mesa</td>
<td>50.2</td>
<td>47.88</td>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>vortex</td>
<td>52.6</td>
<td>40.12</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>gap</td>
<td>59.4</td>
<td>28.48</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>gzip-1.2.4</td>
<td>9.1</td>
<td>8.55</td>
<td>0</td>
<td>17</td>
</tr>
<tr>
<td>gnuchess-5.07</td>
<td>17.8</td>
<td>179.58</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>tcl8.4.14/unix</td>
<td>17.9</td>
<td>585.99</td>
<td>1</td>
<td>14</td>
</tr>
<tr>
<td>hanterm-3.1.6</td>
<td>25.6</td>
<td>52.25</td>
<td>34</td>
<td>1</td>
</tr>
<tr>
<td>sed-4.0.8</td>
<td>26.8</td>
<td>49.34</td>
<td>2</td>
<td>11</td>
</tr>
<tr>
<td>tar-1.13</td>
<td>28.3</td>
<td>57.98</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>grep-2.5.1a</td>
<td>31.5</td>
<td>47.26</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>bison-2.3</td>
<td>48.4</td>
<td>281.84</td>
<td>0</td>
<td>18</td>
</tr>
<tr>
<td>openssh-4.3p2</td>
<td>77.3</td>
<td>97.69</td>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td>fftw-3.1.2</td>
<td>184.0</td>
<td>102.17</td>
<td>9</td>
<td>4</td>
</tr>
<tr>
<td>httpd-2.2.2</td>
<td>316.4</td>
<td>265.43</td>
<td>10</td>
<td>33</td>
</tr>
<tr>
<td>net-snmp-5.4</td>
<td>358.0</td>
<td>899.73</td>
<td>3</td>
<td>36</td>
</tr>
</tbody>
</table>
in Linux Kernel 2.6.4
625 for (minor = 0; minor < 32 && acm_table[minor]; minor++);
... ...
713 acm_table[minor] = acm;

in a proprietary code
if (length >= NET_MAX_LEN)
    return API_SET_ERR_NET_INVALID_LENGTH;
...
buff[length] |= (num << 4);

in a proprietary code
imi_send_to_daemon(PM_EAP, CONFIG_MODE, set_str, sizeof(set_str));
...
imi_send_to_daemon(int module, int mode, char *cmd, int len)
{
...
    strncpy(cmd, reply.str, len);
    cmd[len] = 0;
Sparrow-detected Leak Errors (2/3)

- **in sed-4.0.8/regexp_internal.c**

  948: new_nexts = re_realloc (dfa->nexts, int, dfa->nodes_alloc);
  949: new_indices = re_realloc (dfa->org_indices, int, dfa->nodes_alloc);
  950: new_edests = re_realloc (dfa->edests, re_node_set, dfa->nodes_alloc);
  951: new_eclosures = re_realloc (dfa->eclosures, re_node_set, dfa->nodes_alloc);
  952: new_inveclosures = re_realloc (dfa->inveclosures, re_node_set, dfa->nodes_alloc);
  955: if (BE (new_nexts == NULL || new_indices == NULL || new_edests == NULL || new_eclosures == NULL || new_inveclosures == NULL, 0))
  956: return -1;

- **in proprietary code**

  line = read_config_read_data(ASN_INTEGER, line,
  &StorageTmp->traceRouteProbeHistoryHAddrType,
  &tmpint);

  ...

  line = read_config_read_data(ASN_OCTET_STR, line,
  &StorageTmp->traceRouteProbeHistoryHAddr,
  &StorageTmp->traceRouteProbeHistoryHAddrLen);

  ...

  if (StorageTmp->traceRouteProbeHistoryHAddr == NULL) {
    config_perror
    (‘invalid specification for traceRouteProbeHistoryHAddr’);
    return SNMPERR_GENERR;
  }
in mesa/osmesa.c (in SPEC 2000)

276:  osmesa->gl_ctx = gl_create_context( osmesa->gl_visual );
...
287:  gl_destroy_context( osmesa->gl_ctx );
------------------
1164:  GLcontext *gl_create_context( GLvisual *visual,
       GLcontext *share_list,
       void *driver_ctx )
...
1183:  ctx = (GLcontext *) calloc( 1, sizeof(GLc
...
1211:  ctx->Shared = alloc_shared_state();
---------------------
476:  static struct gl_shared_state *alloc_shared
477:  {
...
489:  ss->Default1D = gl_alloc_texture_object(ss,
490:  ss->Default2D = gl_alloc_texture_object(ss,
491:  ss->Default3D = gl_alloc_texture_object(ss, 0, 3);
----------------------
1257:  void gl_destroy_context( GLcontext *ctx )
1258:  {
...
1274:  free_shared_state( ctx, ctx->Shared );
SPARROW Approach: Pure Soup + Impure Catalysts

Pure soup
- firm formalization of basis: a simple & “sound” abstract interpretation

Impure catalysts
- unorthodox & unsound techniques to refine the bug-finding performance

As a pragmatic approach to find prevalent true errors
- miss rare true cases, reduce many false alarms
A New Localization Technology of Abstract Memories
Motivation

Big cost of global analysis by Sparrow

- Adopted many cost reduction techniques
- Worklist order, localization, selective operators, ...

<table>
<thead>
<tr>
<th>Program</th>
<th>LOC</th>
<th>Time</th>
<th>Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td>bc-1.06</td>
<td>13,093</td>
<td>13,879s</td>
<td>335MB</td>
</tr>
<tr>
<td>twolf</td>
<td>19,700</td>
<td>27,230s</td>
<td>1,199MB</td>
</tr>
<tr>
<td>less-382</td>
<td>23,822</td>
<td>137,827s</td>
<td>1,480MB</td>
</tr>
</tbody>
</table>

1.6 days

Kwangkeun Yi
Towards a Globally Scalable Semantics-based Static Analysis
Real-World Programs

- complex interproc. behavior
- \#Procs: 382
- \#RecProcs: 55(46)

less-382, 23,822 LOC

Kwangkeun Yi  
Towards a Globally Scalable Semantics-based Static Analysis
In global analysis, procedures are wastefully re-analyzed by changes to input states.

```c
int f() {...}  // Suppose f does not access g
int m() {
    g = 1; f();
    g = 2; f();}  // Change of g has no impact on the analysis of f
```

One major reason for big global analysis cost.

<table>
<thead>
<tr>
<th>program</th>
<th>LOC</th>
<th>Repeated analyses of procedures</th>
</tr>
</thead>
<tbody>
<tr>
<td>gzip</td>
<td>7,327</td>
<td>211</td>
</tr>
<tr>
<td>parser</td>
<td>10,900</td>
<td>601</td>
</tr>
<tr>
<td>bc</td>
<td>13,093</td>
<td>1,151</td>
</tr>
<tr>
<td>less</td>
<td>23,822</td>
<td>755</td>
</tr>
</tbody>
</table>
When analyzing a procedure, pass only the memory parts that is necessary to analyze the procedure.

**Solution:**

- Localization reduces re-analysis.

```c
int f() {...}
int m() {
  g = 1; f();
  g = 2; f();
}
```

Assume `f` does not access `g`

Need not to analyze `f` again
How to find the memory parts that will be used during the analysis before the analysis

Finding resources required by analysis

Analysis
Status Quo: Reachability-based Approach (Abstract Garbage Collection)

- Approximate what will be used by removing the unreachable from parameters and globals
- Conventional in control flow analysis, shape analysis, ...

Reachable parts are collected before entering f

call f → entry

f
- Accessed parts are a tiny subset of the reachable
  - ex) only a small number of globals are actually accessed in a procedure

<table>
<thead>
<tr>
<th>Program</th>
<th>LOC</th>
<th>accessed memory / reachable memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>spell-1.0</td>
<td>2,213</td>
<td>5 / 453 (1.1%)</td>
</tr>
<tr>
<td>barcode-0.96</td>
<td>4,460</td>
<td>19 / 1175 (1.6%)</td>
</tr>
<tr>
<td>httptunnel-3.3</td>
<td>6,174</td>
<td>10 / 673 (1.5%)</td>
</tr>
<tr>
<td>gzip-1.2.4a</td>
<td>7,327</td>
<td>22 / 1002 (2.2%)</td>
</tr>
<tr>
<td>jwhois-3.0.1</td>
<td>9,344</td>
<td>28 / 830 (3.4%)</td>
</tr>
<tr>
<td>parser</td>
<td>10,900</td>
<td>75 / 1787 (4.2%)</td>
</tr>
<tr>
<td>bc-1.06</td>
<td>13,093</td>
<td>24 / 824 (2.9%)</td>
</tr>
<tr>
<td>less-290</td>
<td>18,449</td>
<td>86 / 1546 (5.6%)</td>
</tr>
</tbody>
</table>
Our Approach

- More aggressive than reachability
  - Exclude some reachable that will not be accessed
Basically, perform the analysis once more.

- using a further abstraction of the analysis

```
analysis  ⊆  pre-analysis

collect  ⊆  collect

accessed  ⊆  accessed
```
How to Get the Further Abstraction?

- Easy in our case, because the underlying analysis is designed as an abstract interpretation.
- Just apply conservative abstractions to the abstract semantics of the original analysis.

**Original analysis**

\[ \hat{D} = \text{Node} \rightarrow \hat{\text{Mem}} \]

\[ \hat{\mathcal{F}} : \hat{D} \rightarrow \hat{D} \]

\[ \hat{\mathcal{F}} = \lambda \tau. \lambda n. \hat{f}_n (\bigcup_{p \in \text{pred}_0(n)} T(p)) \]

**Pre-analysis**

\[ \hat{D}_{\text{pre}} = \hat{\text{Mem}} \]

\[ \hat{\mathcal{F}}_{\text{pre}} : \hat{D}_{\text{pre}} \rightarrow \hat{D}_{\text{pre}} \]

\[ \hat{\mathcal{F}}_{\text{pre}} = \lambda m. (\bigcup_{n \in \text{Node}} \hat{f}_n(m)) \]

ignore statements orders
Collecting Abstract Locations

- Access functions collects accessed abstract locations.
- Naturally derived from the definitions of the abstract semantics

abstract semantics

\[
\begin{align*}
\hat{\mathcal{V}}(n)(\hat{m}) &= \langle [n, n], \bot, \bot, \bot \rangle \\
\hat{\mathcal{V}}(e_1 + e_2)(\hat{m}) &= \hat{\mathcal{V}}(e_1)(\hat{m}) + \hat{\mathcal{V}}(e_2)(\hat{m}) \\
\hat{\mathcal{V}}(\text{lv})(\hat{m}) &= \bigcup \{ \hat{m}(\text{Addr}) \mid \text{Addr} \in \hat{\mathcal{L}}(\text{lv})(\hat{m}) \} \\
\hat{\mathcal{V}}(&\text{lv})(\hat{m}) &= \langle \bot, \hat{\mathcal{L}}(\text{lv})(\hat{m}), \bot, \bot \rangle \\
\hat{\mathcal{L}}(x)(\hat{m}) &= \{ x \} \\
\hat{\mathcal{L}}(\texttt{*e})(\hat{m}) &= \hat{\mathcal{V}}(e)(\hat{m}).2 \cup \{ l \mid \langle l, o, s \rangle \in \hat{\mathcal{V}}(e)(\hat{m}).3 \} \\
&\quad \cup \{ \langle l, x \rangle \mid l, \{ x \} \in \hat{\mathcal{V}}(e)(\hat{m}).4 \} \\
\hat{\mathcal{L}}(\texttt{e}[e_2])(\hat{m}) &= \{ l \mid \langle l, o, s \rangle \in \hat{\mathcal{V}}(e_2)(\hat{m}).3 \} \\
\hat{\mathcal{L}}(\texttt{e}.x)(\hat{m}) &= \{ \langle l, x \rangle \mid l, \{ x \} \in \hat{\mathcal{V}}(e)(\hat{m}).4 \}
\end{align*}
\]

access functions

\[
\begin{align*}
\hat{\mathcal{A}}V(n)(\hat{m}) &= \emptyset \\
\hat{\mathcal{A}}V(e_1 + e_2)(\hat{m}) &= \hat{\mathcal{A}}V(e_1)(\hat{m}) \cup \hat{\mathcal{A}}V(e_2)(\hat{m}) \\
\hat{\mathcal{A}}V(\text{lv})(\hat{m}) &= \hat{\mathcal{A}}L(\text{lv})(\hat{m}) \cup \hat{\mathcal{L}}(\text{lv})(\hat{m}) \\
\hat{\mathcal{A}}V(&\text{lv})(\hat{m}) &= \hat{\mathcal{A}}L(\text{lv})(\hat{m}) \\
\hat{\mathcal{A}}L(x)(\hat{m}) &= \emptyset \\
\hat{\mathcal{A}}L(\texttt{*e})(\hat{m}) &= \hat{\mathcal{A}}V(e)(\hat{m}) \\
\hat{\mathcal{A}}L(e_1[e_2])(\hat{m}) &= \hat{\mathcal{A}}V(e_1)(\hat{m}) \\
\hat{\mathcal{A}}L(e_.x)(\hat{m}) &= \hat{\mathcal{A}}V(e)(\hat{m})
\end{align*}
\]
1. Access-analysis

\{ f \rightarrow \text{[color: yellow]}, \ g \rightarrow \text{[color: orange]}, \ldots \}\}

2. Actual analysis

Pre-compute access information for each procedure
Localize input state using the access-info
Performance

On average 92% reduction in time

Reachability-based Approach

Our Approach

Towards a Globally Scalable Semantics-based Static Analysis
We can apply access-based localization to any code block C.

$$M = P \ast R$$

$$P \rightarrow Q$$

$$M' = Q \ast R$$

**procedures**

$$M = P \ast R$$

$$P \rightarrow Q$$

$$M' = Q \ast R$$

**loops**

$$M = P \ast R$$

$$M' = Q \ast R$$

**branches**

$$M = P \ast R$$

$$P \rightarrow Q$$

$$M' = Q \ast R$$
Performance

On average 31% further reduction in time
Semantics-based static analysis technology: mature to solve realistic problems for sequential C programs

Our humble proof: Sparrow

Practical, semantically-dense global analysis for 100 MLoc will soon be possible

But lots of problems still remain

- reducing false alarms: cost vs soundness vs reality vs specialization
- alarm explanation: analysis internals too complex to understand
- daunting properties to check/verify

Thank you.
Concluding Remark

- Semantics-based static analysis technology: mature to solve realistic problems for sequential C programs
- Our humble proof: SPARROW
- Practical, semantically-dense global analysis for 100 MLoc will soon be possible
- But lots of problems still remain
  - reducing false alarms: cost vs soundness vs reality vs specialization
  - alarm explanation: analysis internals too complex to understand
  - daunting properties to check/verify

Thank you.