Practical Memory Leak Detector Based on Parameterized Procedural Summaries

Yungbum Jung, Kwangkeun Yi
{dreameye, kwang}@ropas.snu.ac.kr

Programming Research Lab.
Seoul National University
Korea

June 8, 2008
International Symposium on Memory Management
Leaks Detected

Leaks on Exception

```c
p1 = malloc();
if(p1 == NULL) return 0;
p2 = malloc();
if(p2 == NULL) return 0;
```

Omission in Freeing Procedure

```c
s = allocS();
...
freeS(s);
return;
```
Static analysis for detecting memory leaks

- **Procedural Summaries**
  1. Summarizing callee procedures in reverse topological order of static call graph
     - How to analyze a procedure without knowing the input memory?
     - What can be memory leak related behaviours?
  2. Instantiation at the call sites: context-sensitivity

- **Unsound Decisions** for cost-accuracy balance
  - Reducing costs
  - Improving accuracy
Static analysis for detecting memory leaks

- **Procedural Summaries**
  1. Summarizing callee procedures in reverse topological order of static call graph
     - How to analyze a procedure without knowing the input memory? (Access Path)
     - What can be memory leak related behaviours?
  2. Instantiation at the call sites: context-sensitivity

- **Unsound Decisions** for cost-accuracy balance
  - Reducing costs
  - Improving accuracy
A procedure leaks a heap memory whenever

- the memory is allocated while the procedure active
- the memory is neither recycled nor visible after return

Detecting memory leaks needs three information

- allocated addresses
  
  ```
  int *p = malloc();
  ```

- aliases between addresses
  
  ```
  int *x = p;
  ```

- freed addresses
  
  ```
  free(x);
  ```
Exploring Unknown Input Memory

We collect three information without knowing the input memory

1. Only locations accessed by the procedure are important
2. C procedures access the input memory through either arguments or global variables
3. We can determine the “access path” with which those locations are accessed

\((\text{arg}_i \mid \text{ret} \mid \text{global})(\ast \mid .f)\)
Example of Explorations

Exploring Unknown Memory

```c
List * next(List *head) {
    List * cur = head->next;
    free(head);
    return cur;
}
```
Example of Explorations

Exploring Unknown Memory

List * next(List *head) {
    List * cur = head->next;
    free(head);
    return cur;
}

Symbolic addresses $\alpha$ and $\beta$ represent some addresses that already existed before the procedure is called

$$\alpha = \text{arg } *$$
$$\beta = \text{arg } *.next$$
Example of Explorations

Exploring Unknown Memory

List * next(List *head) {
    List * cur = head->next;
    free(head);
    return cur;
}

The symbolic address $\alpha$ is freed

\[ \langle \text{Alloc, Free} \rangle = \langle \emptyset, \{\alpha}\rangle \]
Example of Explorations

Exploring Unknown Memory

List * next(List *head) {
    List * cur = head->next;
    free(head);
    return cur;
}

head ↦ α
α.next ↦ β
cur ↦ β
ret ↦ β

Return address ret contains return value
Example of Explorations

Exploring Unknown Memory

List * next(List *head) {
    List * cur = head->next;
    free(head);
    return cur;
}

This procedure

- frees \( \alpha \) accessed from the argument with \( \text{arg} \cdot \text{arg}^* \)
- returns \( \beta \) accessed from the argument with \( \text{arg} \cdot \text{arg}^* \cdot \text{arg} \cdot \text{next} \)
Memory Leak Related Behaviours

How can procedures affect leak detections?

No Effects

```c
void local() {
    int *p = malloc();
    free(p);
}
```

Effects

```c
int *foo(int *p) {
    free(p);
    return malloc();
}
```

The `foo` procedure frees an address pointed to by the argument and returns an allocated address.
Memory Leak Related Behaviours?

Procedure `fcall` calls the function pointer

```c
void fcall(int *a, void (*fp)(int *)) {
    fp(a);
}
```

```c
p = malloc();
fcall(p, free);    no leak!
```

We can not summarize all memory leak related behaviours
8 Summary Categories

- To detect more leaks
- To avoid false positives
- To capture interprocedural aliasing

<table>
<thead>
<tr>
<th></th>
<th>free</th>
<th>global</th>
<th>argument</th>
<th>return</th>
</tr>
</thead>
<tbody>
<tr>
<td>allocation</td>
<td>-</td>
<td>-</td>
<td>Alloc2Arg</td>
<td>Alloc2Ret</td>
</tr>
<tr>
<td>global</td>
<td>-</td>
<td>-</td>
<td>Glob2Arg</td>
<td>Glob2Ret</td>
</tr>
<tr>
<td>argument</td>
<td>Arg2Free</td>
<td>Arg2Glob</td>
<td>Arg2Arg</td>
<td>Arg2Ret</td>
</tr>
</tbody>
</table>
Examples of Summary Categories (1/2)

Allocation
- Alloc2Arg, Alloc2Ret

List *f(int **p) {
    List *cur = malloc();
    cur->next = malloc();
    *p = malloc();
    return cur;
}

--- ➔ existed before the procedure is called

--- ➔ done by the procedure
Examples of Summary Categories (2/2)

Free & Globalization
- Arg2Free, Glob2Ret

Node gNode;
Node *g(int *p) {
    free(p);
    return &gNode;
}

Diagram:
- Node labeled 'arg' to node labeled 'global'
- Node labeled 'ret' to node labeled 'global'

Programming Research Lab. Seoul National University, Korea
Examples of Summary Categories (2/2)

Free & Globalization
- Arg2Free, Glob2Ret

Node gNode;
Node *g(int *p){
    free(p);
    return &gNode;
}

No Leaks

void f(){
    Node *node;
    int *p = malloc();
    node = g(p);
    node->next = malloc();
}
Summarization from Memory

From

```c
List *next(List *head) {
    List *cur = head->next;
    free(head);
    return cur;
}
```

To

- **Arg2Free**
  - `head` $\mapsto$ $\alpha$
  - $\alpha$.next $\mapsto$ $\beta$
  - `cur` $\mapsto$ $\beta$
  - `ret` $\mapsto$ $\beta$

- **Arg2Ret**
  - `arg` $\mapsto$ $\alpha$
  - `ret` $\mapsto$ $\beta$
  - `arg`.next $\mapsto$ $\beta$
  - `ret` $\mapsto$ `ret` *
foo(Node *x, Node *y){
    free(y->next);
    free(x);
}

Node *a = malloc();
Node *b = a;
a->next = malloc();
foo(a,b);
Summary Instantiation

```c
foo(Node *x, Node *y){
    free(y->next);
    free(x);
}
```

```c
Node *a = malloc();
Node *b = a;
a->next = malloc();
foo(a,b);
```

Parameterized addresses are instantiated

```
d a  \mapsto  \ell_1
b  \mapsto  \ell_1
\ell_1.next  \mapsto  \ell_2
```
Performance Numbers

Programs | Size (KLOC) | Time (sec) | Bug Count | False Positives
--- | --- | --- | --- | ---
art | 1.2 | 0.68 | 1 | 0
equake | 1.5 | 1.03 | 0 | 0
mcf | 1.9 | 2.77 | 0 | 0
bzip2 | 4.6 | 1.52 | 1 | 0
gzip | 7.7 | 1.56 | 1 | 4
parser | 10.9 | 15.93 | 0 | 0
ammp | 13.2 | 9.68 | 20 | 0
vpr | 16.9 | 7.85 | 0 | 9
crafty | 19.4 | 84.32 | 0 | 0
twolf | 19.7 | 68.80 | 5 | 0
mesa | 50.2 | 43.15 | 9 | 0
vortex | 52.6 | 34.79 | 0 | 1
gap | 59.4 | 31.03 | 0 | 0
gcc | 205.8 | 1330.33 | 44 | 1
binutils-2.13.1 | 909.4 | 712.09 | 228 | 25
openssh-3.5p1 | 36.7 | 10.75 | 18 | 4
httpd-2.2.2 | 316.4 | 74.87 | 0 | 0
tar-1.13 | 49.5 | 11.73 | 5 | 3

SPEC2000 benchmarks

Open source programs

Programming Research Lab. Seoul National University, Korea
Comparison with Others (1/2)

Sparrow finds consistently more bugs than others on the same programs

<table>
<thead>
<tr>
<th>C program</th>
<th>Tool</th>
<th>Bug Count</th>
<th>False Positives</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPEC2000 benchmark</td>
<td>Sparrow</td>
<td>81</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>FastCheck ’07 (Cornell)</td>
<td>59</td>
<td>8</td>
</tr>
<tr>
<td>binutils-2.13.1 &amp; openssh-3.5.p1</td>
<td>Sparrow</td>
<td>246</td>
<td>29</td>
</tr>
<tr>
<td></td>
<td>Saturn ’05 (Stanford)</td>
<td>165</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>Clouseau ’03 (Stanford)</td>
<td>84</td>
<td>269</td>
</tr>
</tbody>
</table>

Table: On the same test programs
### Comparison with Others (2/2)

<table>
<thead>
<tr>
<th>Tool</th>
<th>C size</th>
<th>Speed</th>
<th>Bug Count</th>
<th>False Positive Ratio (%)</th>
<th>Efficacy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Saturn ’05 (Stanford)</td>
<td>6,822</td>
<td>50</td>
<td>455</td>
<td>10%</td>
<td>1/150</td>
</tr>
<tr>
<td>Clouseau ’03 (Stanford)</td>
<td>1,086</td>
<td>500</td>
<td>409</td>
<td>64%</td>
<td>1/170</td>
</tr>
<tr>
<td>FastCheck ’07 (Cornell)</td>
<td>671</td>
<td>37,900</td>
<td>63</td>
<td>14%</td>
<td>1/149</td>
</tr>
<tr>
<td>Contradiction ’06 (Cornell)</td>
<td>321</td>
<td>300</td>
<td>26</td>
<td>56%</td>
<td>1/691</td>
</tr>
<tr>
<td>Sparrow</td>
<td>1,777</td>
<td>785</td>
<td>332</td>
<td>12%</td>
<td>1/66</td>
</tr>
</tbody>
</table>

**Table:** Overall Performance

In comparison with other published memory leak detectors

- Analysis speed: 785LOC/sec, next to the fastest FastCheck
- False-positive ratio: 12.4% next to the smallest Saturn
- Efficacy: Sparrow the biggest

$$\frac{\text{Bug Count}}{\text{KLOC}} \div \text{False Positive Ratio}$$
Practical Performance

For 1 Million lines of code Sparrow

- takes 23 minutes
- detects 186 leaks
- with only 23 false positives
Unsound Escaping Effects from Path-insensitivity

Path-insensitive Analysis

\[ f(\text{int } *x, \text{int } *y)\{
\text{int } *p;
\text{if}(\cdots) \ p = x;
\text{else} \ p = y;
\text{free}(p);
\} \]

\begin{align*}
x & \mapsto \alpha \\
y & \mapsto \beta \\
p & \mapsto \{\alpha, \beta\}
\end{align*}

Unsound escaping effects on arguments
Global Variables Abstraction

The global node represents all the global variables

```c
int *gp;
void f(int *p){ gp = p; }
```

Interprocedural Overwritten on Global Variable

```c
int n;
f(malloc());
f(&n);  // overwritten leak!
```
Global Variables Abstraction

More categories are required to detect such leaks

<table>
<thead>
<tr>
<th></th>
<th>free</th>
<th>global</th>
<th>argument</th>
<th>return</th>
</tr>
</thead>
<tbody>
<tr>
<td>allocation</td>
<td></td>
<td>Alloc2Glob</td>
<td>Alloc2Arg</td>
<td>Alloc2Ret</td>
</tr>
<tr>
<td>global</td>
<td>Glob2Free</td>
<td>Glob2Glob</td>
<td>Glob2Arg</td>
<td>Glob2Ret</td>
</tr>
<tr>
<td>argument</td>
<td>Arg2Free</td>
<td>Arg2Glob</td>
<td>Arg2Arg</td>
<td>Arg2Ret</td>
</tr>
</tbody>
</table>
Being Sensitive to Memory-Allocating Paths

Return Integer Values

```c
int *foo(int **a) {
    if (n == 0) return 0;
    *a = malloc(n);
    return 1;
}
```

```c
void bar() {
    if (foo(&p) == 0)
        return;  // false positive!
    ...
}
```

Return integer values = \{ 0, 1 \}
Being Sensitive to Memory-Allocating Paths

Return Integer Values

```c
int *foo(int **a){
    if(n == 0) return 0;
    *a = malloc(n);
    return 1;
}
```

```c
void bar(){
    if(foo(&p) == 0)
        return; // unreachable!
    ...
}
```

return integer values = \{ 1 \}
**k-bound Explorations**

The number of explorations is limited up to k

```c
freeList(List *cur)
{
    List *prev = cur;
    while(cur != NULL)
    {
        cur = cur->next;
        free(prev);
        prev = cur;
    }
}
```

![Diagram showing the process of explorations](image)

Programming Research Lab. Seoul National University, Korea
Conclusion

- Practical Memory Leak Detector
  - pure soup
  - Abstract Interpretation
  - unsound seasoning
- Procedural Summary
  - access path representation for exploring unknown memory
  - categorizing memory leak related behaviours

Programming Research Lab. Seoul National University, Korea