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

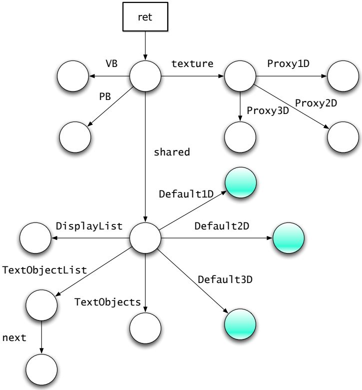
Leaks Detected

Leaks on Exception

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

Omission in Freeing Procedure

```
s = allocS();
...
freeS(s);
return;
```



Analysis Overview

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

Analysis Overview

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?
          8 Summary Categories
    2. Instantiation at the call sites: context-sensitivity
- **Unsound Decisions** for cost-accuracy balance
    - Reducing costs
    - Improving accuracy

Memory Leaks

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

$$(\texttt{arg}_i \mid \texttt{ret} \mid \texttt{global})(\texttt{*} \mid \texttt{.f})^*$$

Example of Explorations

Exploring Unknown Memory

```
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;
}
```

$$\begin{array}{rcl} \texttt{head} & \mapsto & \alpha \\ \alpha\texttt{.next} & \mapsto & \beta \\ \texttt{cur} & \mapsto & \beta \end{array}$$

Symbolic addresses α and β represent some addresses that already existed before the procedure is called

$$\begin{array}{rcl} \alpha & = & \texttt{arg *} \\ \beta & = & \texttt{arg *.next} \end{array}$$

Example of Explorations

Exploring Unknown Memory

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

$$\begin{array}{rcl} \texttt{head} & \mapsto & \alpha \\ \alpha\texttt{.next} & \mapsto & \beta \\ \texttt{cur} & \mapsto & \beta \end{array}$$

The symbolic address α is freed

$\langle$ Alloc, Free $\rangle = \langle \varnothing, \{\alpha\} \rangle$

Example of Explorations

Exploring Unknown Memory

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

$$\begin{array}{rcl} \texttt{head} & \mapsto & \alpha \\ \alpha\texttt{.next} & \mapsto & \beta \\ \texttt{cur} & \mapsto & \beta \\ \texttt{ret} & \mapsto & \beta \end{array}$$

Return address `ret` contains return value

Example of Explorations

Exploring Unknown Memory

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

$\texttt{head} \mapsto \alpha$
$\alpha\texttt{.next} \mapsto \beta$
$\texttt{cur} \mapsto \beta$
$\texttt{ret} \mapsto \beta$

This procedure

- frees α accessed from the argument with `arg*`
- returns β accessed from the argument with `arg*.next`

Memory Leak Related Behaviours

How can procedures affect leak detections?

No Effects

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

Effects

```
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

High-order Effects of Procedure

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

```
p = malloc();
fcall(p, free);
```

no leak!

We can not summarize all memory leak related behaviours

8 Summary Categories

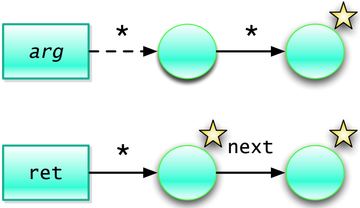
- To detect more leaks
- To avoid false positives
- To capture interprocedural aliasing

	free	global	argument	return
allocation	-	-	Alloc2Arg	Alloc2Ret
global	-	-	Glob2Arg	Glob2Ret
argument	Arg2Free	Arg2Glob	Arg2Arg	Arg2Ret

Examples of Summary Categories(1/2)

Allocation
- Alloc2Arg, Alloc2Ret

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



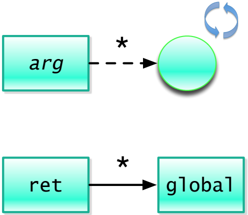
- - -▶ existed before the procedure is called

——▶ done by the procedure

Examples of Summary Categories(2/2)

Free & Globalization
- Arg2Free, Glob2Ret

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

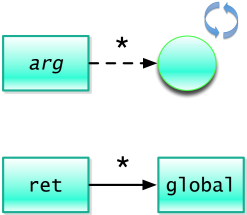


Examples of Summary Categories(2/2)

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

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

$\texttt{head} \mapsto \alpha$

$\alpha.\texttt{next} \mapsto \beta$

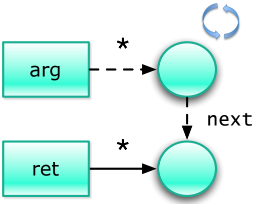
$\texttt{cur} \mapsto \beta$

$\texttt{ret} \mapsto \beta$

To

Arg2Free

Arg2Ret



$\alpha = \texttt{arg *}$

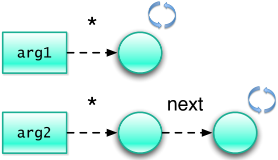
$\beta = \texttt{arg *.next}$

$\beta = \texttt{ret *}$

Summary Instantiation

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

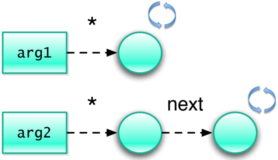
```
Node *a = malloc();
Node *b = a;
a->next = malloc();
foo(a,b);
```



$$\begin{array}{rcl} \texttt{a} & \mapsto & \ell_1 \\ \texttt{b} & \mapsto & \ell_1 \\ \ell_1\texttt{.next} & \mapsto & \ell_2 \end{array}$$

Summary Instantiation

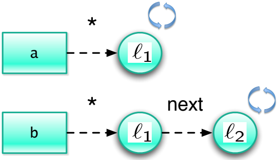
```
foo(Node *x, Node *y){
   free(y->next);
   free(x);
}
```



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

$$\begin{array}{rcl} \mathtt{a} & \mapsto & \ell_1 \\ \mathtt{b} & \mapsto & \ell_1 \\ \ell_1\mathtt{.next} & \mapsto & \ell_2 \end{array}$$

Parameterized addresses are instantiated



Performance Numbers

Sparrow The Early Bird, our analyzer www.spa-arrow.com

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

Comparison with Others(1/2)

Sparrow finds consistently more bugs than others on the same programs

C program	Tool	Bug Count	False Positives
SPEC2000 benchmark	Sparrow	81	15
	FastCheck '07 (Cornell)	59	8
binutils-2.13.1 & openssh-3.5.p1	Sparrow	246	29
	Saturn '05 (Stanford)	165	5
	Clouseau '03 (Stanford)	84	269

Table: On the same test programs

Comparison with Others(2/2)

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

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

$$\frac{\text{BugCount}/\text{KLOC}}{\text{FalsePositiveRatio}}$$

Prcatical Performance

For 1Million lines of code Sparrow

- takes 23 minutes
- detects 186 leaks
- with only 23 false positives

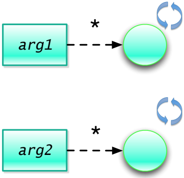
Unsound Escaping Effects from Path-insensitivity

Path-insensitive Analysis

```
f(int *x, int *y){
   int *p;
   if(···) p = x;
   else  p = y;
   free(p);
}
```

$$\begin{array}{lcc} \mathtt{x} & \mapsto & \alpha \\ \mathtt{y} & \mapsto & \beta \\ \mathtt{p} & \mapsto & \{\alpha, \beta\} \end{array}$$

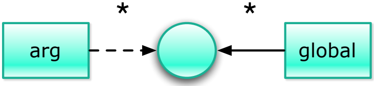
Unsound escaping effects on arguments



Global Variables Abstraction

The global node represents all the global variables

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



Interprocedural Overwritten on Global Variable

```
int n;
f(malloc());
f(&n);        overwritten leak!
```

Global Variables Abstraction

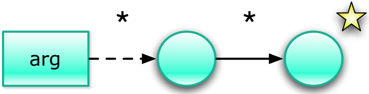
More categories are required to detect such leaks

	free	global	argument	return
allocation	-	Alloc2Glob	Alloc2Arg	Alloc2Ret
global	Glob2Free	Glob2Glob	Glob2Arg	Glob2Ret
argument	Arg2Free	Arg2Glob	Arg2Arg	Arg2Ret

Being Sensitive to Memory-Allocating Paths

Return Integer Values

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



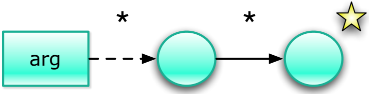
return integer values = { 0, 1 }

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

Being Sensitive to Memory-Allocating Paths

Return Integer Values

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



return integer values = { 1 }

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

k-bound Explorations

The number of explorations is limited up to k

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

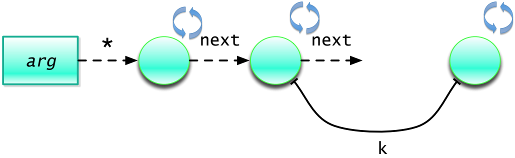
$\texttt{cur} \mapsto \alpha$

$\alpha\texttt{.next} \mapsto \beta$

$\beta\texttt{.next} \mapsto \gamma$

$\gamma\texttt{.next} \mapsto \delta$

⋮



Conclusion

- Practical Memory Leak Detector

pure soup	**Abstract Interpretation**
+	
unsound seasoning	**Procedural Summary** **Unsound Decisions**

- Procedural Summary
  - access path representation for exploring unknown memory
  - categorizing memory leak related behaviours