Sparrow

Static Bug Finder

Yungbum Jung co-worked with
YI Jhee, MS Jin, DH Kim, SH Kong, HJ Lee, HJ Oh, DJ Park and KK Yi

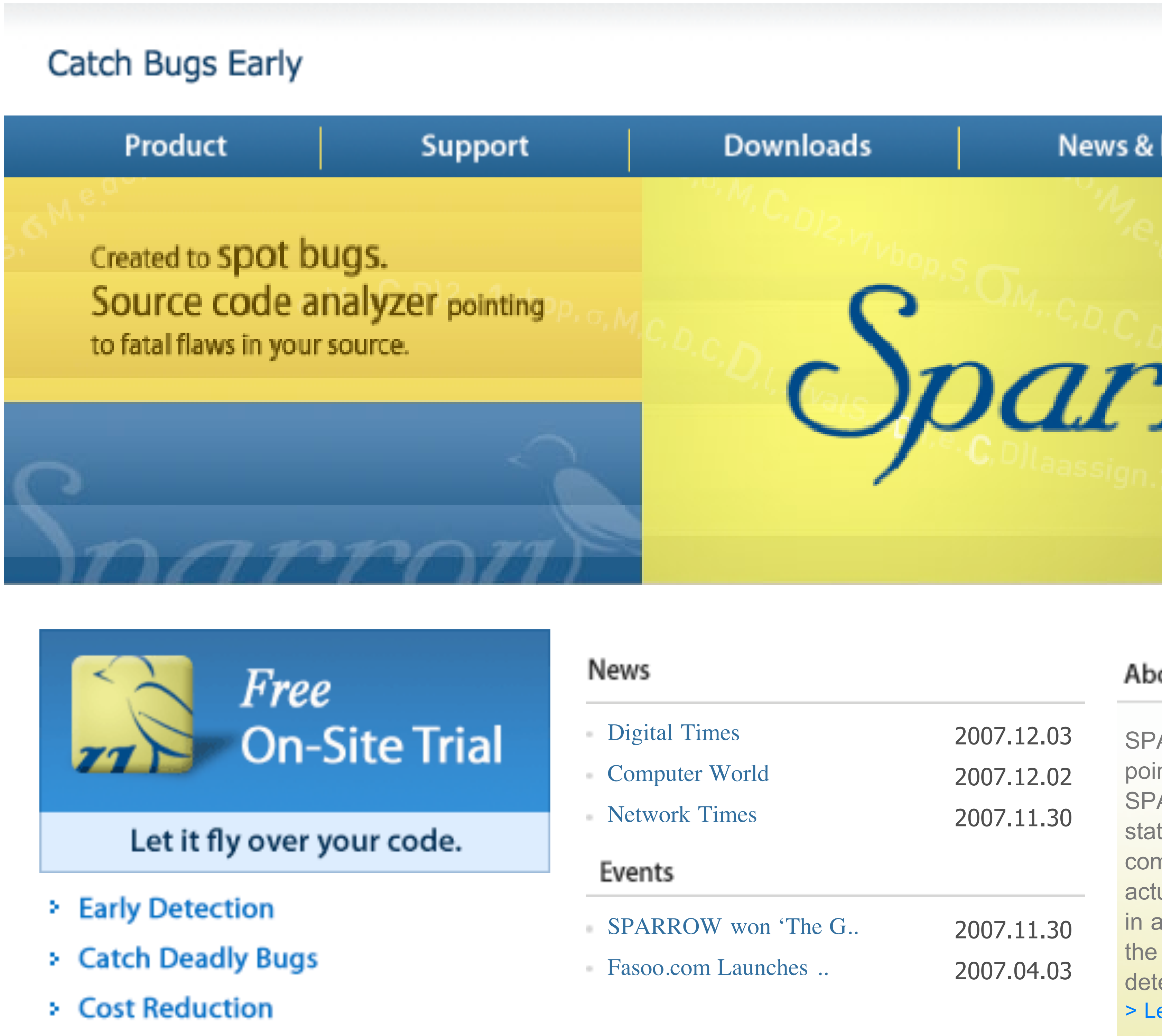
Programming Research Lab.
Seoul National Univ.

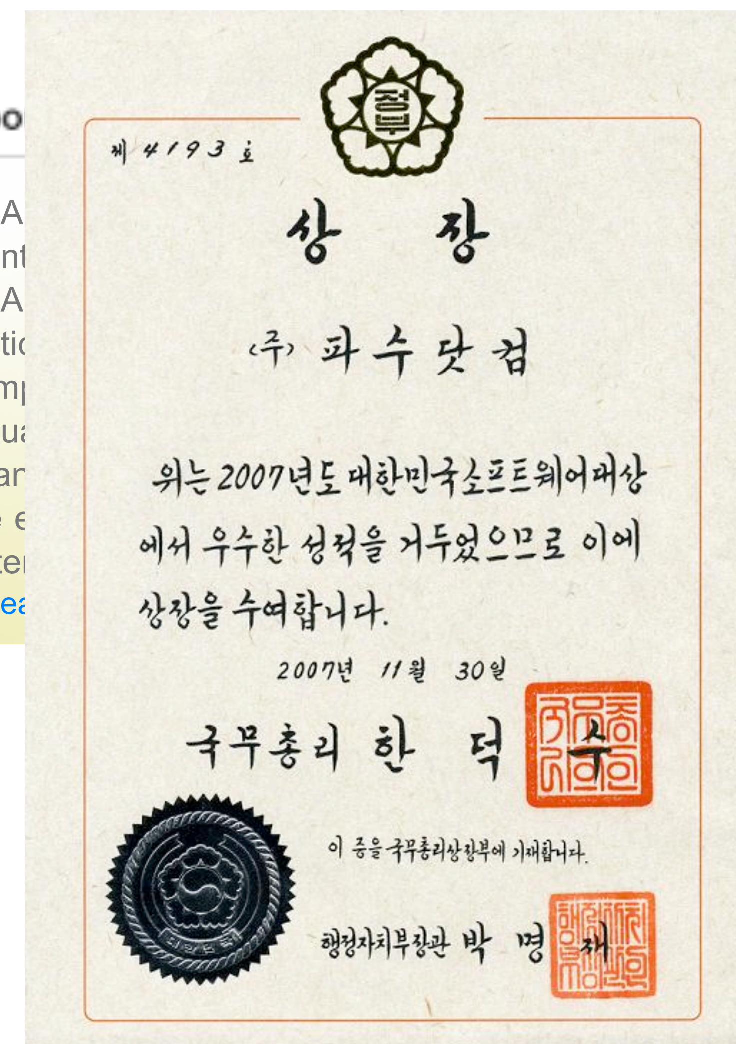
SIGPL Winter School, 01/31/2008 @ KAIST

Sparrow History




- 2004 - Airac: Array index range analyzer for C (abstract interpretation)
- 2005 - AiracV: improved Airac + statistical post analysis[SAS'05]
- 2006 - AiracV: loop-refinement Mairac: memory leak detector
- 2007 - Sparrow: edg parser + M/Airac engine + reason chain + UI
- 2008 - Sparrow 2.0: Sparrow Nest + path-sensitive analysis + more bugs checker (null-dereference, ...)



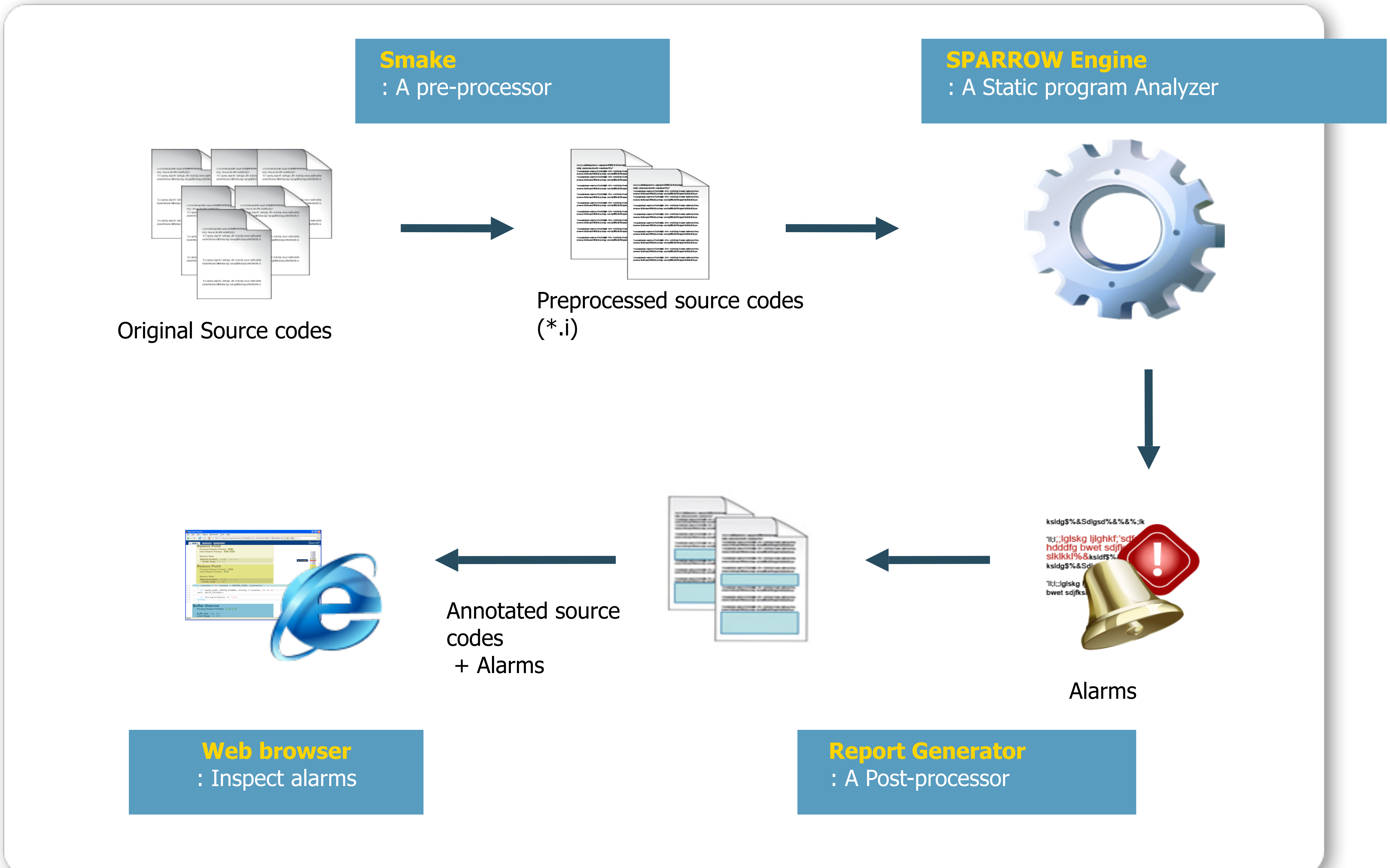


http://spa-arrow.com

Overview on Sparrow

- Sparrow is a static source code analyzer that points to fatal bugs in C
    - Airac: Buffer Overrun, Uninitialized Local Variables, Divided by Zero
    - Mairac: Memory Leak, Null Dereference, Double Free, Use After Free, Return Pointer to Local and Return Pointer to Freed
- Sparrow's analysis engines are created by semantics-based static analysis technology, abstract interpretation

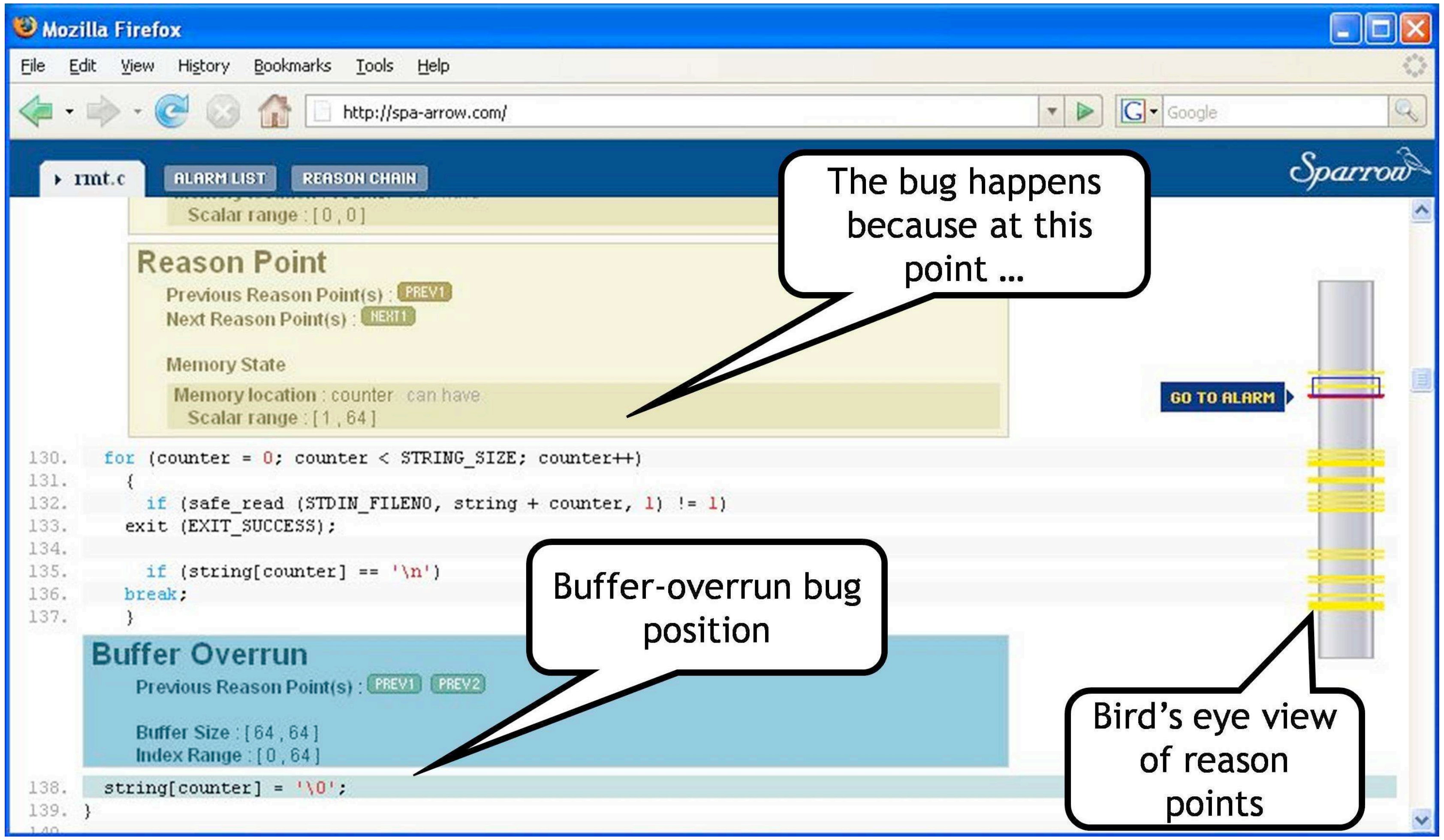
How Sparrow Works



Reporting Bugs

- Sparrow uses statistical post-analysis to rank the alarms, so that the user can check highly probable errors first

- Sparrow explains bugs



Sparrow Performance

Buffer overrun detection (SPEC2000 and open sources) (as of 01/04/2008)

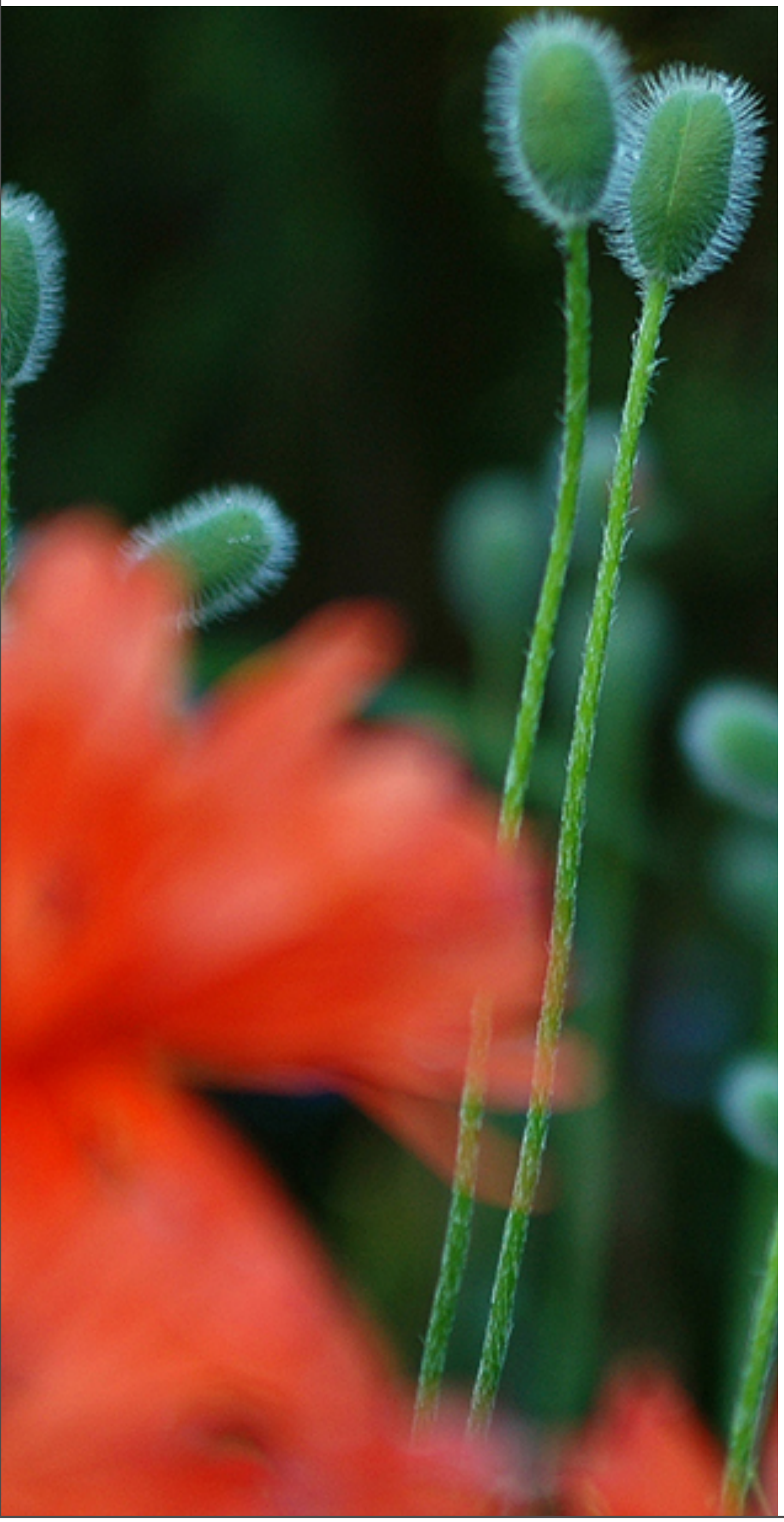
Programs	Size KLOC	Time (sec)	True Alarms	False Alarms
art	1.2	0.45	0	0
equake	1.5	2.89	0	1
mcf	1.9	0.33	0	0
bzip2	4.6	10.90	23	29
gzip	7.7	3.38	18	24
parser	10.9	260.94	4	13
twolf	19.7	8.59	0	0
ammp	13.2	10.20	6	0
vpr	16.9	11.15	0	3
crafty	19.4	139.80	1	5
mesa	50.2	47.88	2	10
vortex	52.6	40.12	2	0
gap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

Sparrow Performance

Memory leak detection (SPEC2000 and open sources) (as of 01/04/2008)

Programs	Size KLOC	Time (sec)	True Alarms	False Alarms
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
gnuchess-5.07	17.8	9.44	4	0
tcl8.4.14	17.9	266.09	4	4
hanterm-3.1.6	25.6	13.66	0	0
sed-4.0.8	26.8	13.68	29	31
tar-1.13	28.3	13.88	5	3
grep-2.5.1a	31.5	22.19	2	3
openssh-3.5p1	36.7	10.75	18	4
bison-2.3	48.4	48.60	4	1
openssh-4.3p2	77.3	177.31	1	7
fftw-3.1.2	184.0	15.20	0	0
httpd-2.2.2	316.4	102.72	6	1
net-snmp-5.4	358.0	201.49	40	20
binutils-2.13.1	909.4	712.0 9	228	25

Sparrow Performance



In comparison with other published memory leak detectors

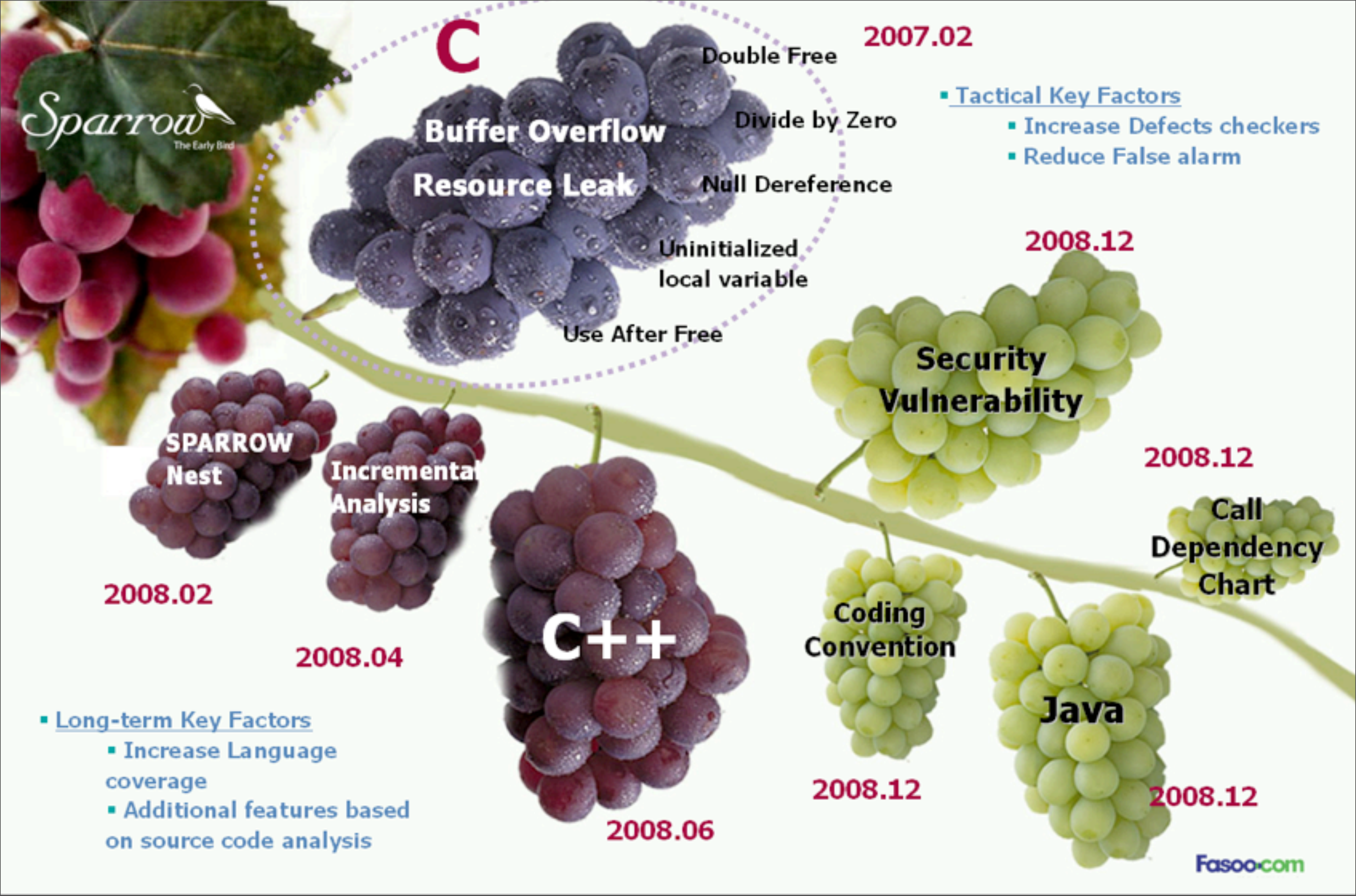
- Number of bugs: SPARROW finds consistently more bugs than others
- Analysis speed: 788LOC/sec, next to the fastest FastCheck.
- False-alarm ratio: 21%
- Efficacy (TrueAlarms/KLOC $\times$ 1/FalseAlarmRatio): biggest

Tool	C size KLOC	Speed LOC/s	True Alarms	False Alarm 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	2,543	720	433	21%	1/123

Table: Overall comparison

C program	Tool	True Alarms	False Alarm Count
SPEC2000 benchmark	SPARROW	81	15
	FastCheck '07 (Cornell)	59	8
binutils-2.13.1 & openssh-3.5.p1	SPARROW	236	19
	Saturn '05 (Stanford)	165	5
	Clouseau '03 (Stanford)	84	269

Table: Comparison for the same C programs



Sparrow Road Map

Memory Leak Analysis

Memory Leak Analysis on Airac

- Reporting not freed addresses when program terminates

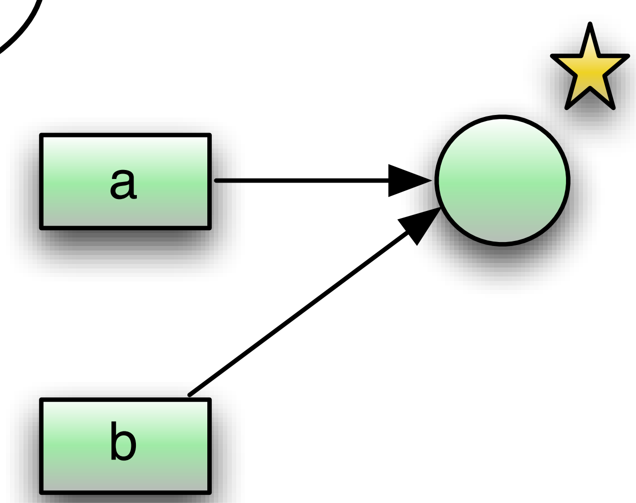
call site abstraction

```
while(1)
  p = malloc();
```

```
void * mymalloc(int size){
  return malloc(size);
 }

void main(){
  char *a = mymalloc(1);
  int  *b = mymalloc(4);
  free(a);
}
```

context insensitive



Problem Localizing (program $\longrightarrow$ procedure)

- How can we know that a procedure makes allocated addresses safe?

```
p=malloc;
```

  - freed

```
free(p);
```

  - return value

```
return p;
```

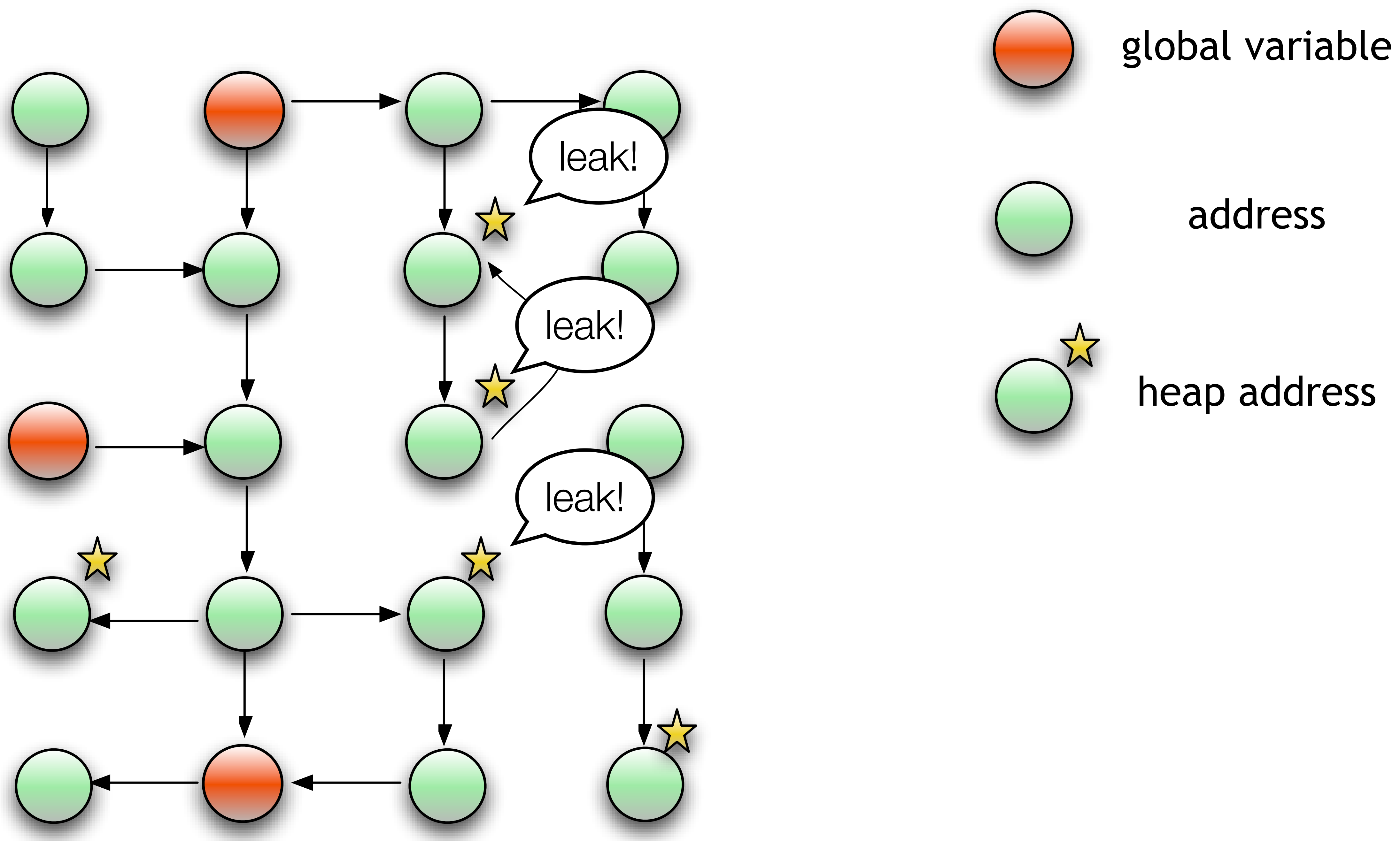
  - arguments passed to procedure

```
f(int **x){
  *x = p;
}
```

  - global variables

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

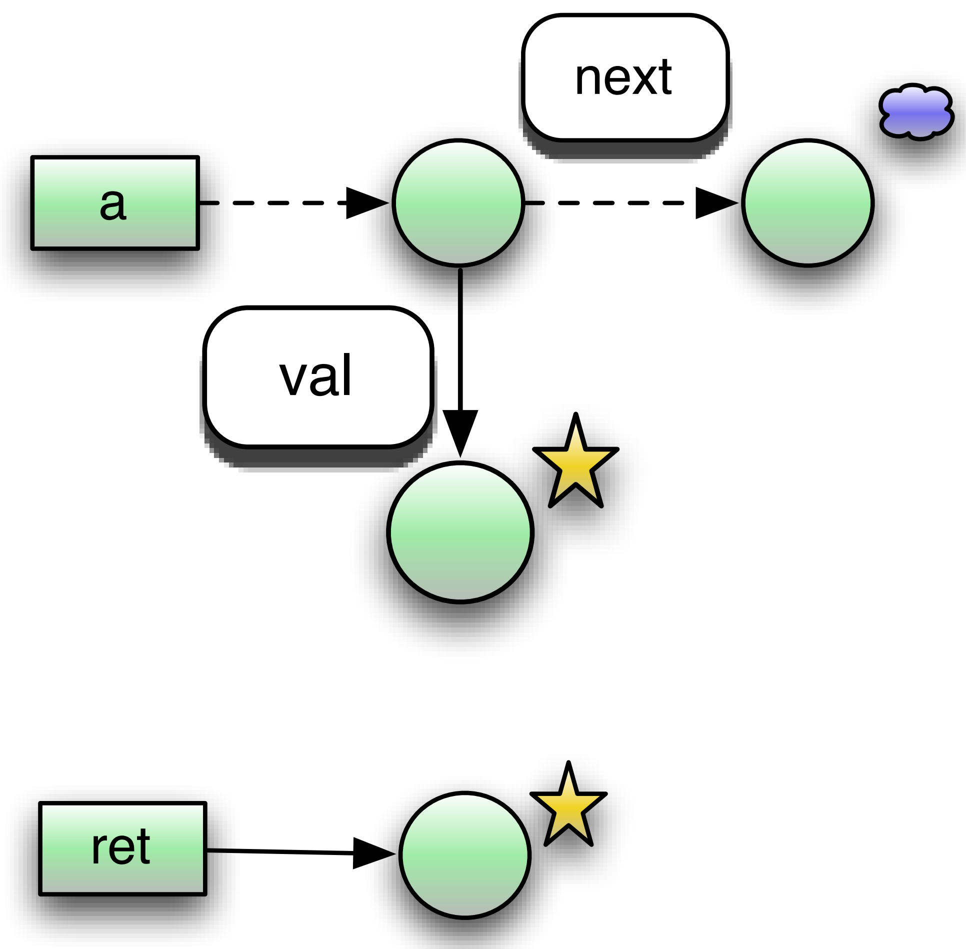
Memory Leak Problem = Graph Reachability Problem



Symbolic Address for Exploring Unknown Memory

- We can't know the input memory while analyzing one procedure

```
char * f(List * arg){
  free(arg->next);
  arg->val = malloc(10);
  return malloc(1);
}
```

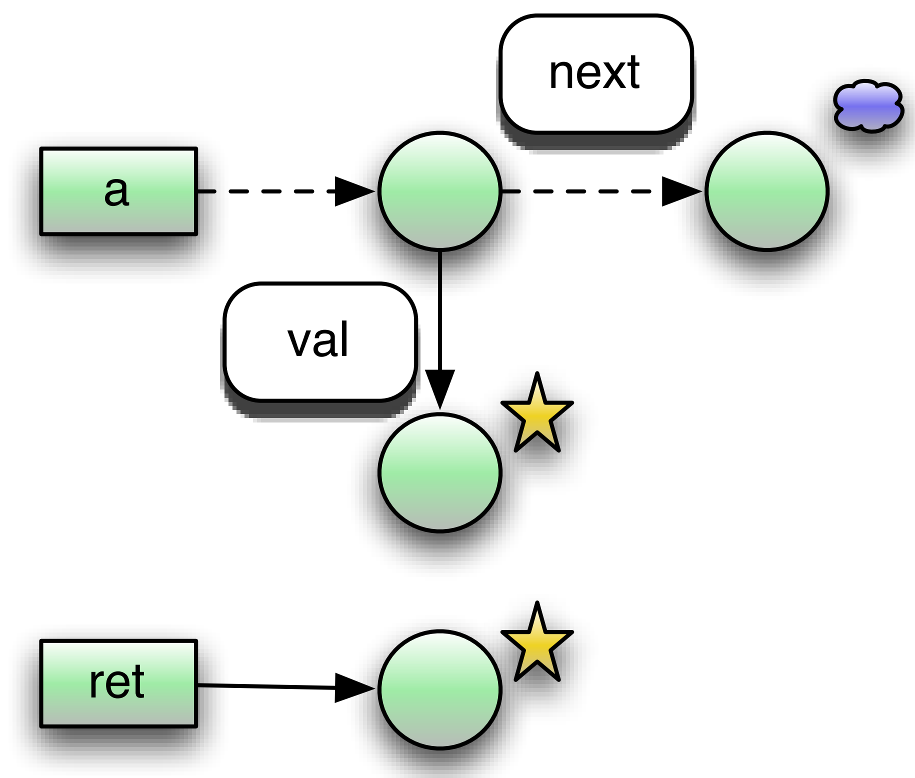


We can infer input memories from memory usages in the procedure

Procedural Summary

- How can we handle procedure call?

```
char * f(List * arg){
  free(arg->next);
  arg->val = malloc(10);
  return malloc(1);
}
```




```
void bar(){
  List * lst   alloc(sizeof(list));
  lst->next   lloc(sizeof(list));
  return f(lst);
}
```

leak!

leak!

Categories on Procedural Summary

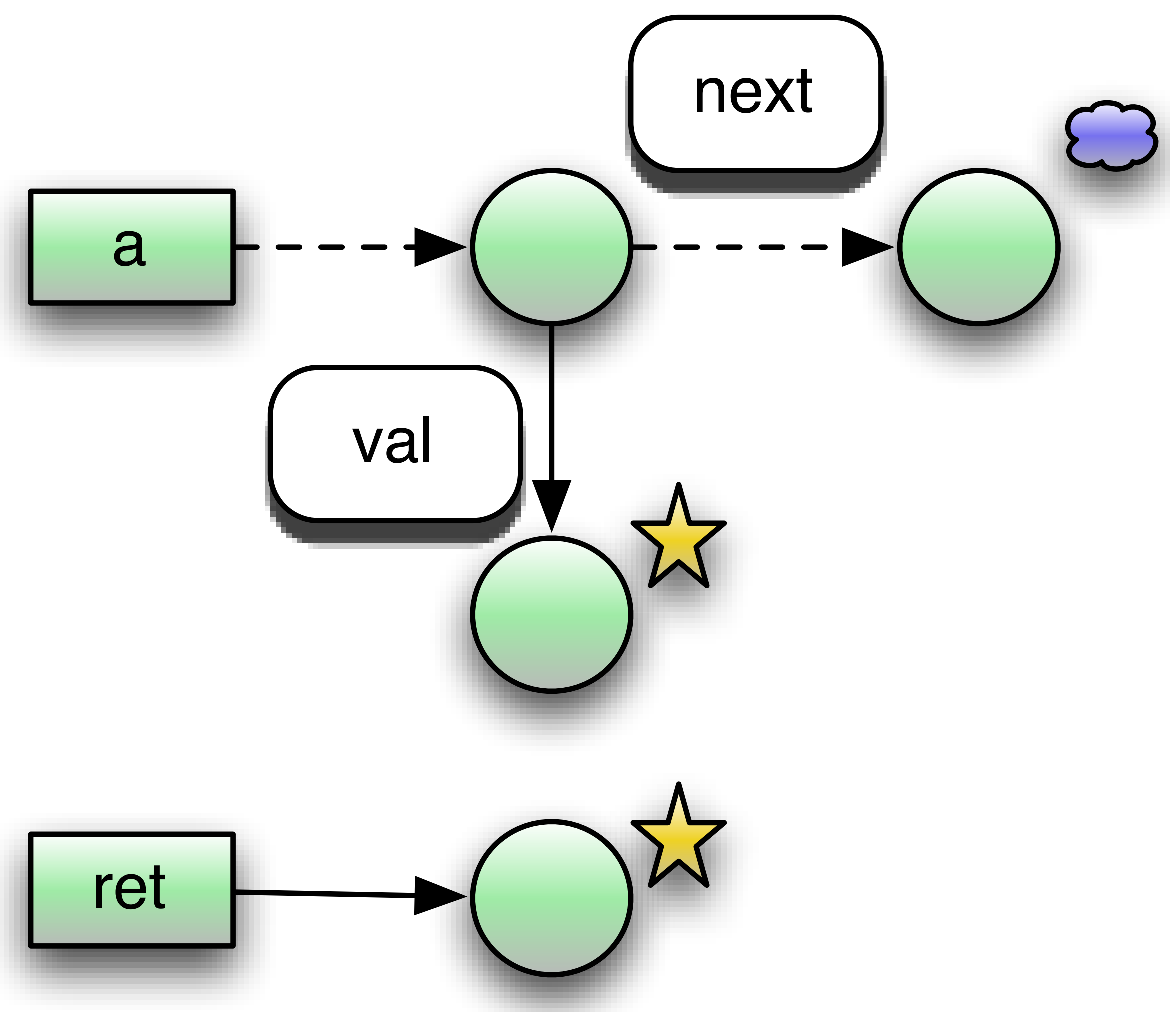
- We are interested in the following 8 categories for detecting memory leaks

	freeing	allocating	globalizing	aliasing
argument	FreeArg	Alloc2Arg	Arg2Glob Glob2Arg	Arg2Arg
return		Alloc2Ret	Glob2Ret	Arg2Ret

- It seems that the above categories are sufficient for most realistic programs
  - + exit, null return, varargs, returned number ...
  - - there always exist exceptions making analyzer fool

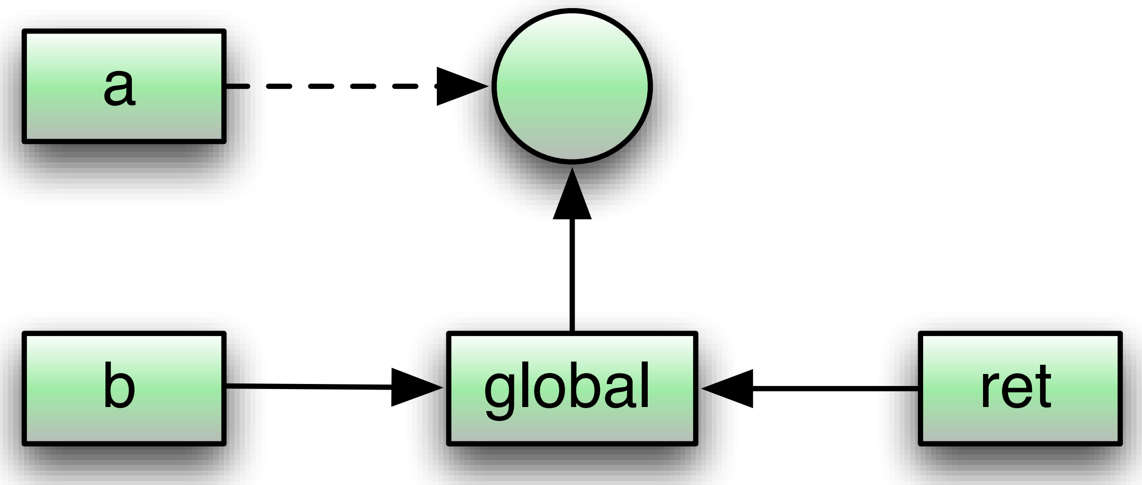
Freeing - FreeArg
Allocating - Alloc2Arg, Alloc2Ret

```
char * f(List * a){
  free(a->next);
  a->val = malloc(10);
  return malloc(1);
}
```

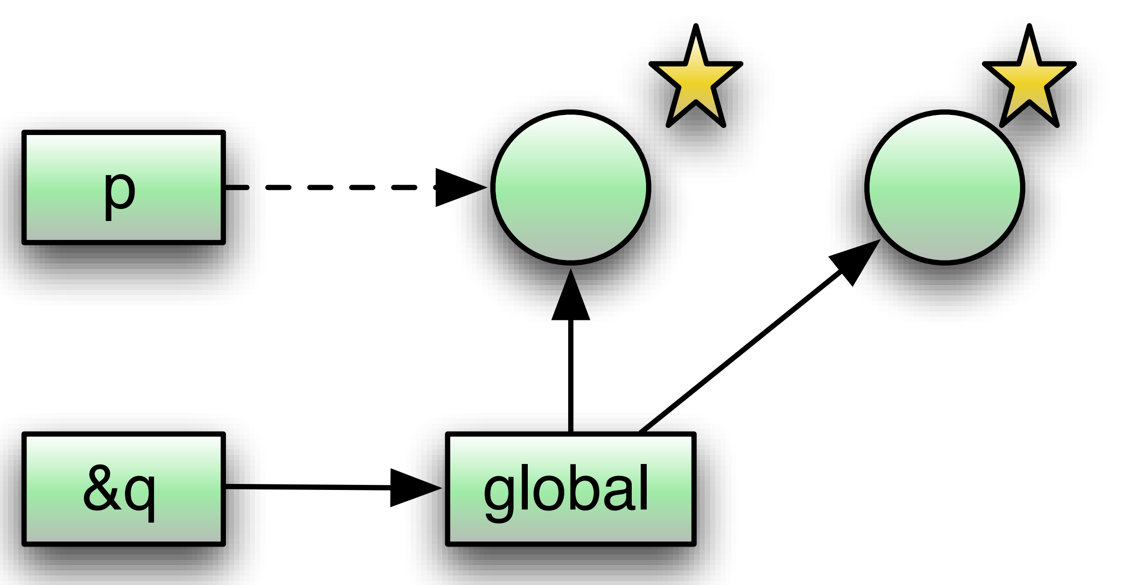


Globalizing - Glob2Arg, Arg2Glob, Glob2Ret

```
int *ga, *gb;
int gc;
int * glob(int *a, int **b){
  ga = a;
  b = &gb;
  return &gc;
}
```

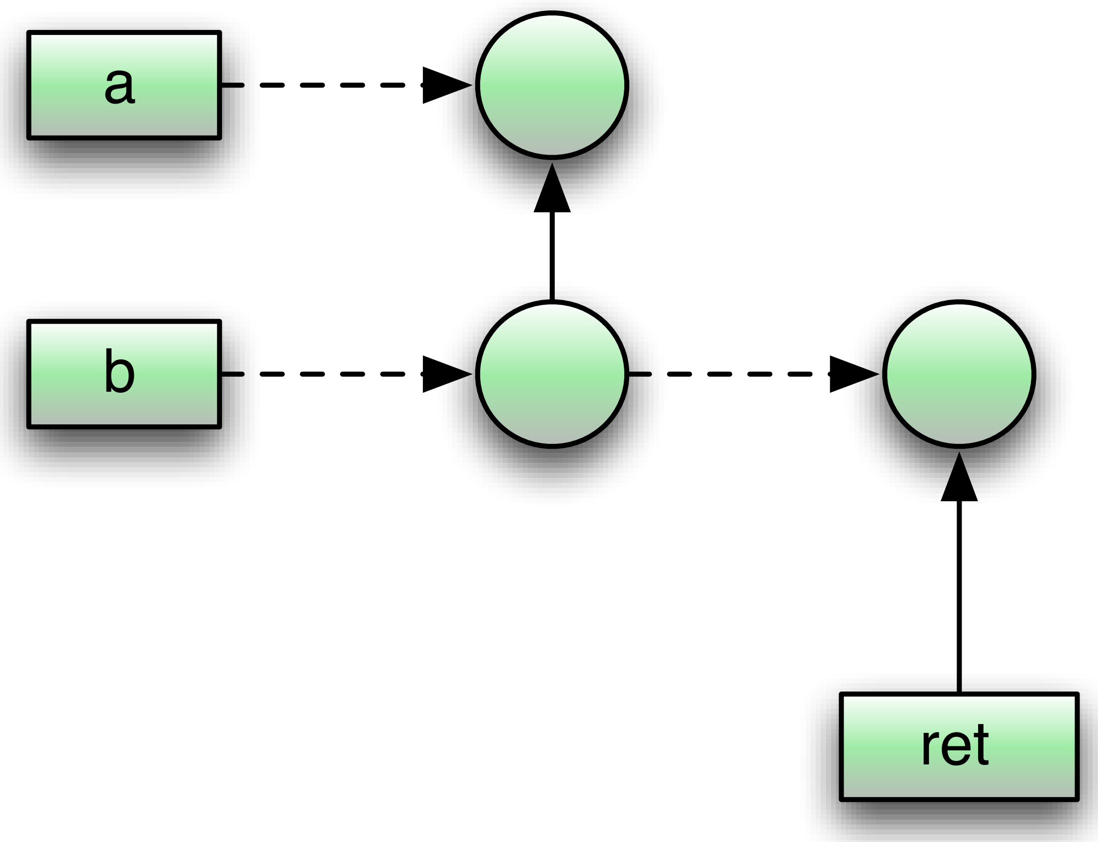



```
int *p = malloc();
int *q;
glob(p, &q);
q = malloc();
```



Aliasing - Arg2Arg, Arg2Ret

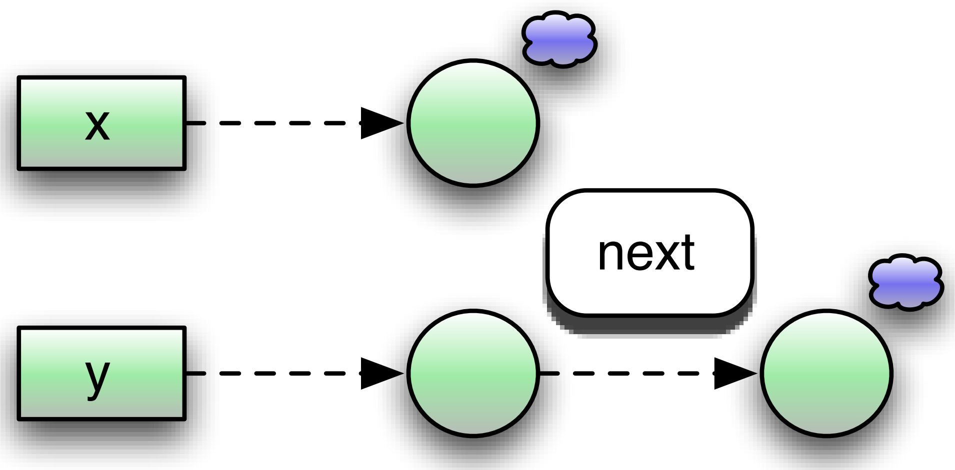
```
int *aliasing(int *a, int **b){
  int *ret = *b;
  *b = a;
  return ret;
}
```



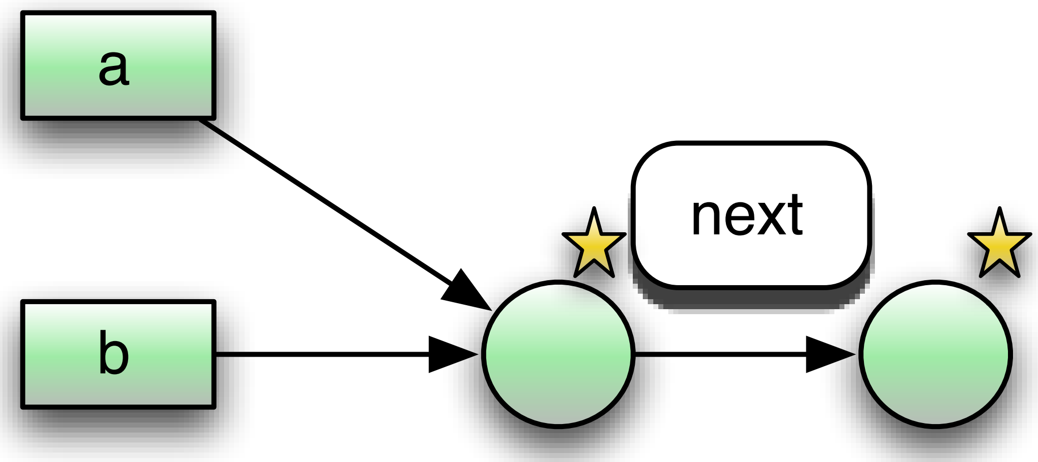
Summary Instantiation

- Procedural summaries are instantiated depending on calling contexts

```
f(List *x, List *y){
  free(y->next);
  free(x);
}
```



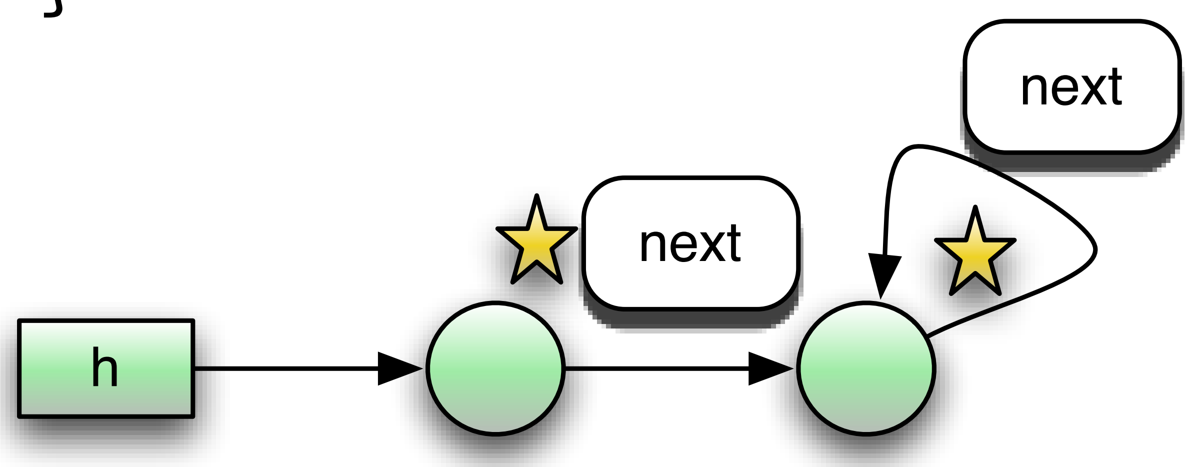

```
g(){
  List *a = malloc();
  List *b = a;
  a->next = malloc();
  f(a,b);
}
```



Abstraction

- Dynamically allocated addresses are abstracted to their static call sites
- The number of introducing symbolic addresses is constantly bounded
- Using a pair of intervals for number values (with widening)

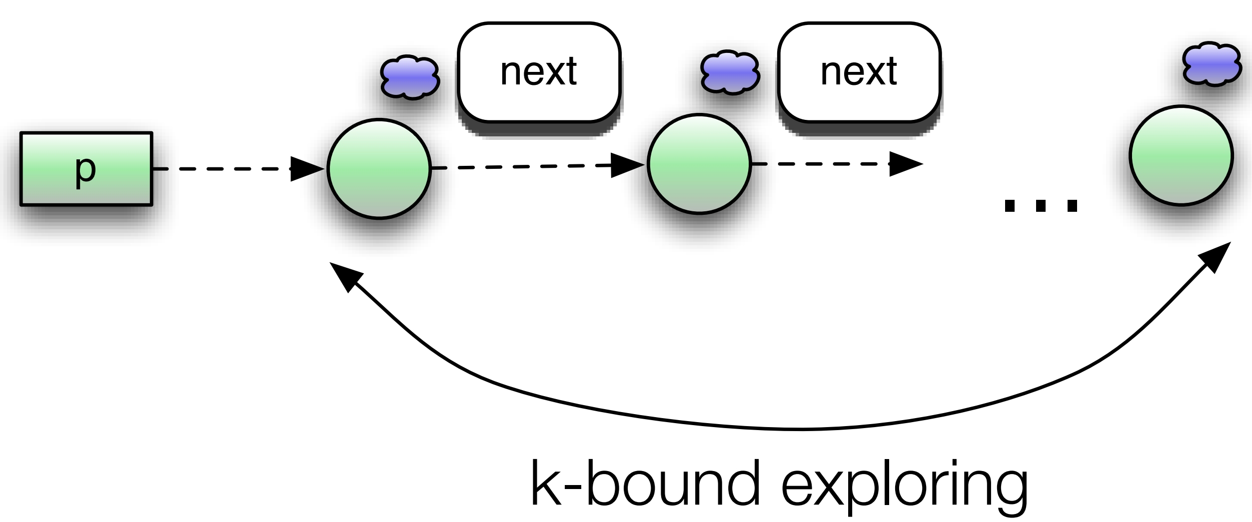
```
List * allocList(int n){
  List *h = malloc();
  List *c = h;
  for(i=1;i<n;i++){
    c->next = malloc();
    c = c->next;
  }
  return h;
}
```



Abstraction

- Dynamically allocated addresses are abstracted to their static call sites
- The number of introducing symbolic addresses is constantly bounded
- Using a pair of intervals for number values (with widening)

```
freeList(List *p){
  List *c = p;
  while(c != Null){
    p = p->next;
    free(c);
    c = p;
  }
}
```



Abstraction

- Dynamically allocated addresses are abstracted to their static call sites
- The number of introducing symbolic addresses is constantly bounded
- Using a pair of intervals for number values (with widening)

```
int foo(int n){
  int s = 0;
  for(i=0;i<n;i++){
    s++;
  }
  return s;
}
```

s = [0, 0], [0, 1], ... , [0,+oo]

Pair of intervals is useful for non-zero numbers
!0 = [-oo, -1] [1, +oo]

Memory Leaks in Real Programs

- in sed-4.0.8/regexp_internal.c

```
 new_nexts = re_realloc (dfa->nexts, int, dfa->nodes_alloc);
 new_indices = re_realloc (dfa->org_indices, int, dfa->nodes_alloc);
 new_edests = re_realloc (dfa->edests, re_node_set, dfa->nodes_alloc);
 new_eclosures = re_realloc (dfa->eclosures, re_node_set,
     dfa->nodes_alloc);
 new_inveclosures = re_realloc (dfa->inveclosures, re_node_set,
dfa->nodes_alloc);
 if (BE (new_nexts == NULL || new_indices == NULL
 || new_edests == NULL || new_eclosures == NULL
 || new_inveclosures == NULL, 0))
   return -1;
```

- in proprietary code

```
fp = fopen(SYSLOC_CONF,"r");
tp = fopen("/etc/syslog.tmp", "w");
...
if (!fp) 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;
}
```

Memory Leaks on Complex Heap Structure

- in mesa/osmesa.c(in SPEC 2000)

```
276:   osmesa->gl_ctx = gl_create_context( osmesa->gl_visual );
...
285:   if (!osmesa->gl_buffer) {
286:     gl_destroy_visual( osmesa->gl_visual );
287:     gl_destroy_context( osmesa->gl_ctx );
288:     free(osmesa);
289:     return NULL;
290:   }
------------------
1164: GLcontext *gl_create_context( GLvisual *visual,
                                    GLcontext *share_l
                                    void *driver_ctx )
...
1183:  ctx = (GLcontext *) calloc( 1, sizeof(GLc
...
1210:     /* allocate new group of display lists
1211:     ctx->Shared = alloc_shared_state();
---------------------
476: static struct gl_shared_state *alloc_shared
477: {
...
480: ss = (struct gl_shared_state*) calloc( 1, sizeof(struct gl
488: /* Default Texture objects */
489: ss->Default1D = gl_alloc_texture_object(ss, 0, 1);
490: ss->Default2D = gl_alloc_texture_object(ss, 0, 2);
491: ss->Default3D = gl_alloc_texture_object(ss, 0, 3);
----------------------
1257: void gl_destroy_context( GLcontext *ctx )
1258: {
      ...
```