Automatic Memory-Leak Fixing for C Programs

Qing Gao, **Yingfei Xiong**, Lu Zhang, Bing Xie, Hong Mei
Peking University

2013-12-21

Outline

1. Memory Leak
2. Existing work
3. Approach
4. Experiments
5. Discussion

Memory Leak

- A program consumes memory but is unable to release it

```
int f(int num, int len){
1:    int *array = (int*)malloc(sizeof(int)*len);
2:    if (array == NULL)   return -1;
      …//operation on array

20:   int sum = 0;
21:   if (num < 0) free(array);
22:   else
23:         for (int i = 0; i < len; i++)
24:               if (array[i] != num)
25:                     sum += array[i];
26:               else
27:                     return 0;
28:    printf("%d", sum);
29:    return sum;
30: }
```

Memory leak [Clause and Orso, 2010]

- Difficult to detect by testing
- Potential to impact multiple applications
- Common even in mature applications

Outline

1. Memory Leak
2. Existing work
3. Approach
4. Experiments
5. Discussion

Existing work- Memory leak detection

- Static detection:
  - False positives
- Dynamic detection
  - Runtime overhead
- Programmers need to check programs to fix leaks
  - Need some time
  - Correctness is not ensured

Existing work- Dynamic pinpointing

- Need to run program
- Instrument nearly every instruction and overhead is high
- Programmers still need to check programs

Existing work- Compile-time object deallocation

- Focus on Java bytecode
  - May lead to double free
  - May lead to unreadable code
  - Cannot make sure whether a leak has been completely fixed

Outline

1. Memory Leak
2. Existing work
3. Approach
4. Experiments
5. Discussion

Correct fix definition

- A location in the code to insert “free(p)” where for any path covers this location
  - Pointer p points to an allocated chunk at the location (allocation and its reference)
  - There is no other deallocation statements that release the memory (no double frees)
  - There is no reference to the memory after the execution (no use after free)

Approach overview

- Building A-CFG
- Memory leak detection
- Memory Leak fixing
- Map A-CFG to code

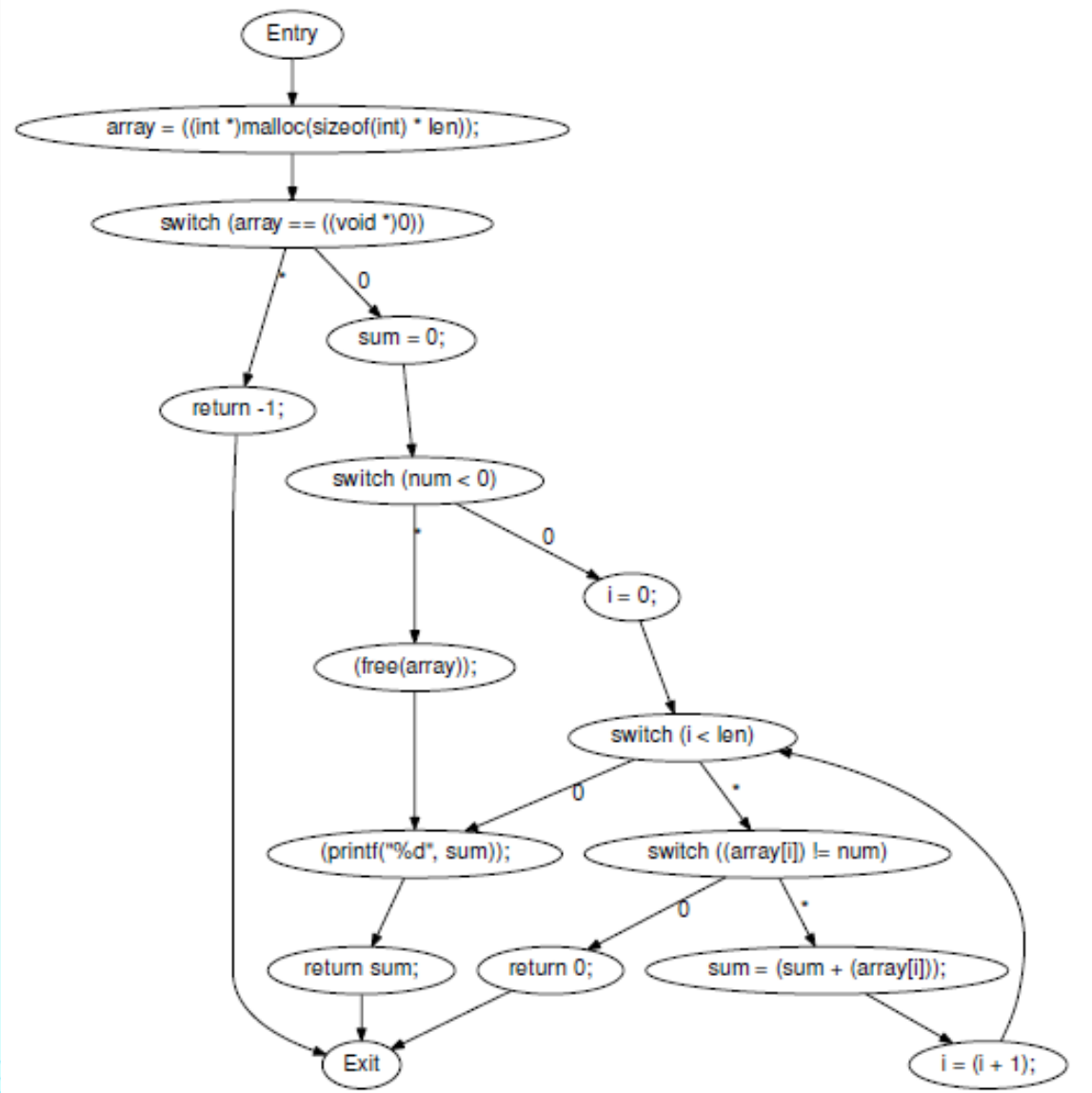
Building A-CFG

- A-CFG: Allocation-centric Control Flow Graph

- Transformed from CFG (control flow graph)
  - Have no cycles (no loops)
  - Abstract information only related to memory management

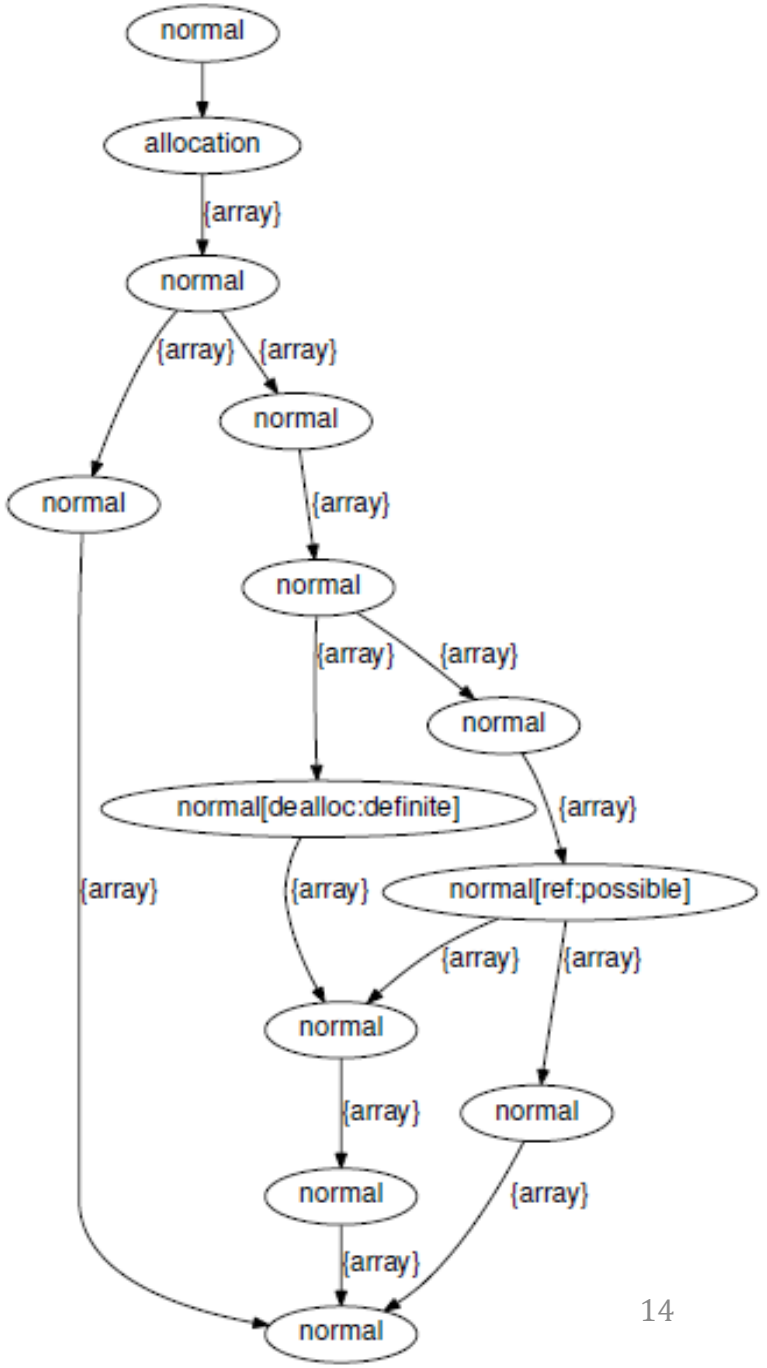
Building A-CFG

- Input: CFG



Building A-CFG

- Output: A-CFG
- Node type:
  - Normal
  - Allocation
- Node label:
  - Dealloc: definite/possible
  - Ref: possible
- Edge label: pointer set



Building A-CFG

- Inter-procedure analysis
  - Build procedure summaries
    - Method *m* with parameters *p*0, ..., *pn*

$$<\mathcal{S}_0, \mathcal{F}_0> \times \ldots \times <\mathcal{S}_n, \mathcal{F}_n> \rightarrow <\mathcal{S}_r>$$

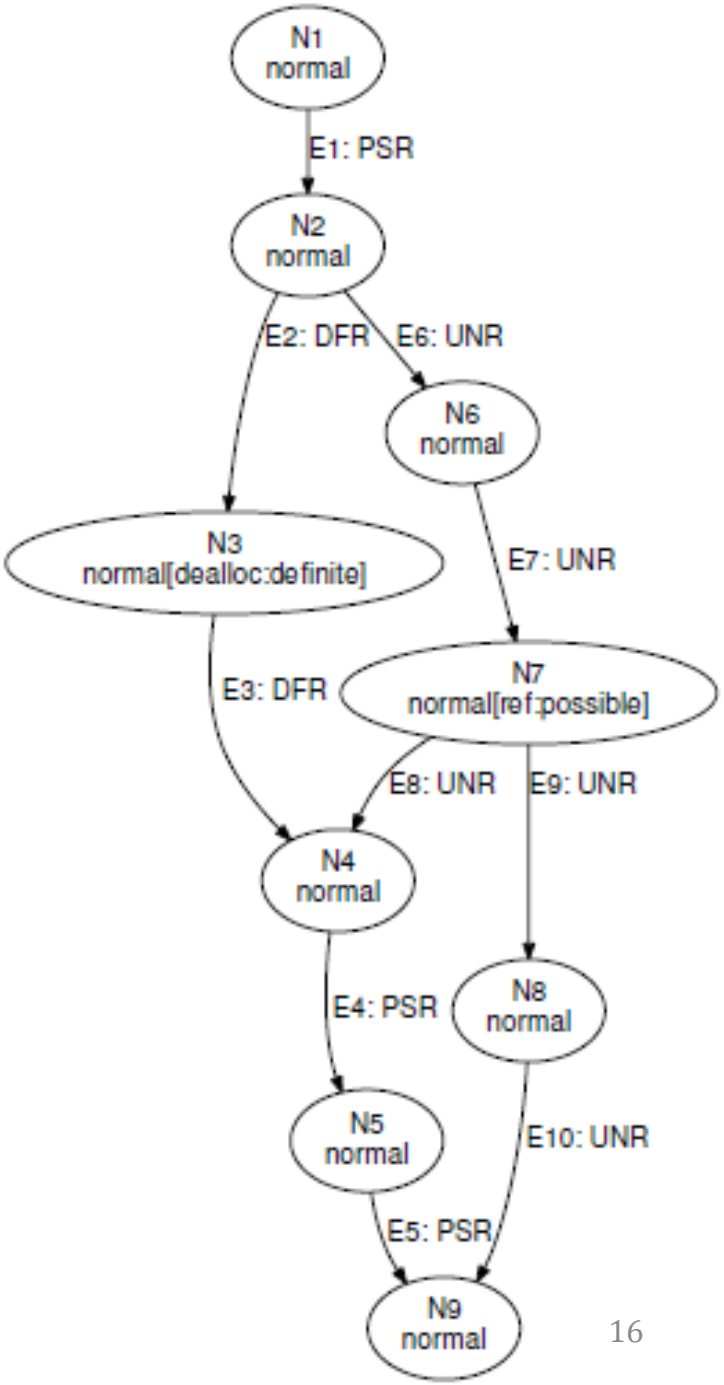
*Sj : what *pj will point to*

Fj : whether pj is freed

    - May and Must summaries
  - Map summaries to A-CFG nodes

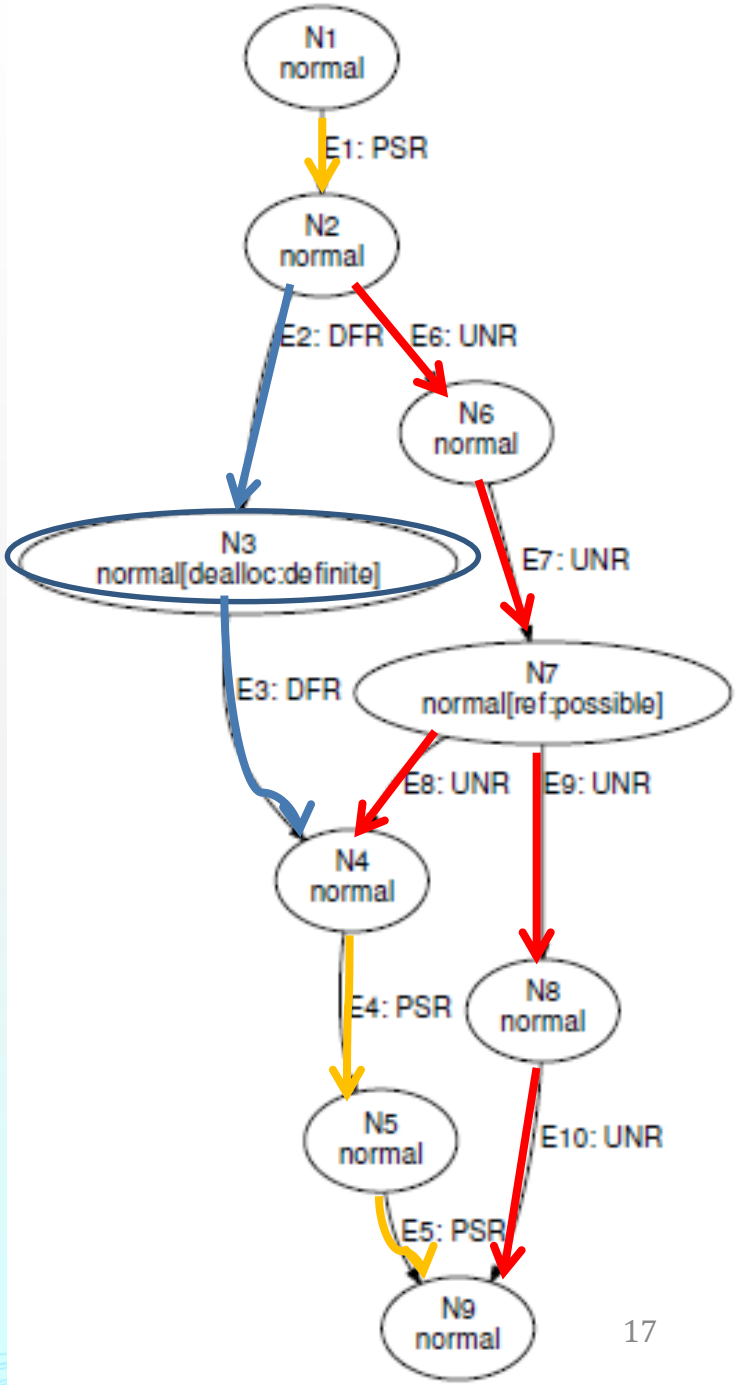
Memory Leak Detection

- Classify the edges by deallocation:
  - Blue: definitely reachable
  - Yellow: possibly reachable
  - Red: unreachable



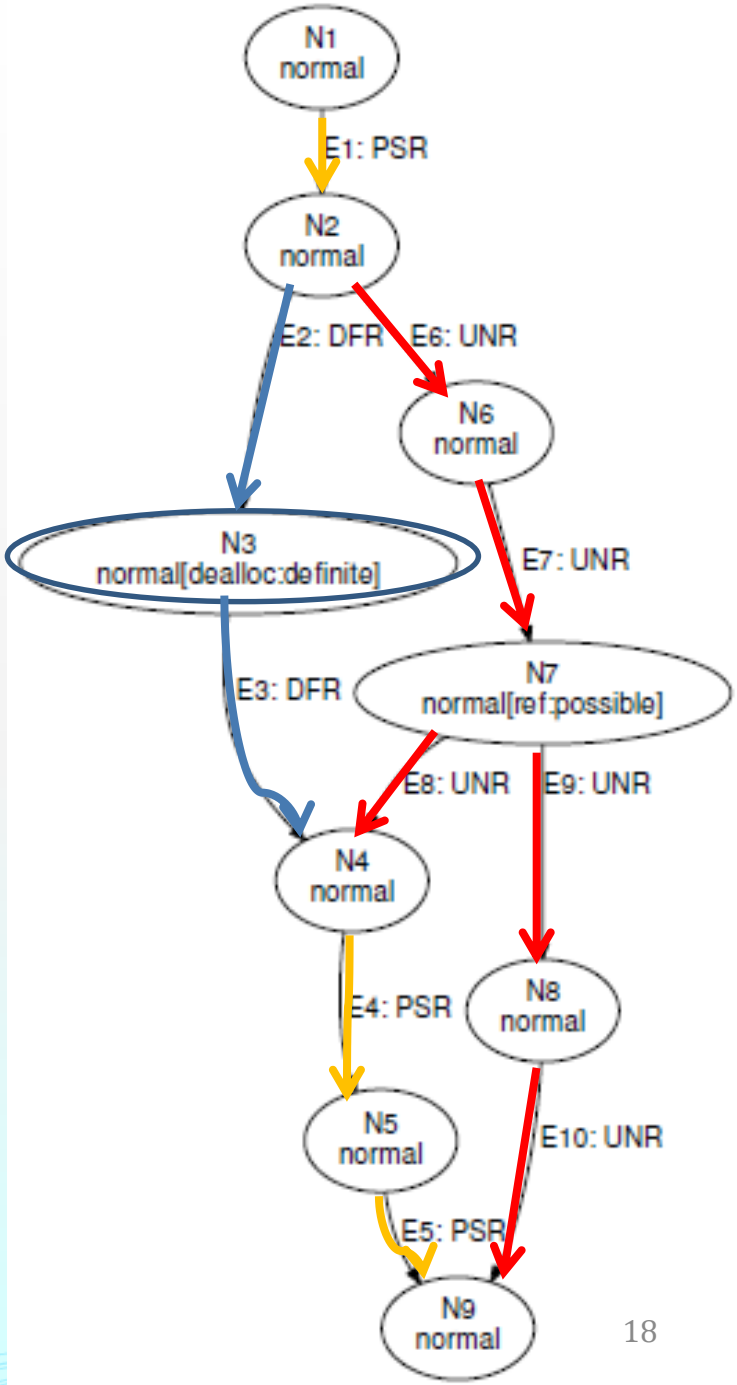
Memory Leak Detection

- Classify the edges by deallocation:
  - Blue: definitely reachable
  - Yellow: possibly reachable
  - Red: unreachable



Memory leak fixing

- Fix on edges whose pointer set is not empty:
  - allocation and its references
- Fix on red edges:
  - no double free
- Traverse backwardly:
  - no use after free



Outline

1. Memory Leak
2. Existing work
3. Approach
4. Experiments
5. Discussion

Experiment

- RQ1: How effective is our tool in fixing real-world memory leaks?
- RQ2: What are the execution time and memory consumption of our tool for memory leak fixing?

Experiment

- Benchmark: SPEC2000
  - Used by papers related to detection

Programs	Size (Kloc)	#Func	#Allocation
art	1.3	44	11
equake	1.5	45	29
mcf	1.9	44	3
bzip2	4.6	92	10
gzip	7.8	128	5
parser	10.9	342	1
ammp	13.3	197	37
vpr	17	290	2
crafty	18.9	127	12
twolf	19.7	209	2
mesa	49.7	1124	67
vortex	52.7	941	8
gap	59.5	872	2
gcc	205.8	2271	53

Experimental Results

- Effectiveness

Programs	#Fixed	#Maximum Detected	Percentage(%)	#Fixing Points	#Unnecessary Fixes
art	0	1	0	0	0
equake	0	0	N/A	0	0
mcf	0	0	N/A	0	0
bzip2	1(1,1)	1	100	1	0
gzip	1(1,1)	1	100	1	0
parser	0	0	N/A	0	0
ammp	15(7,15)	20	75	27	0
vpr	0	0	N/A	0	0
crafty	0	0	N/A	0	0
twolf	0	5	0	0	0
mesa	1(0,0)	9	11	1	0
vortex	0	0	N/A	0	0
gap	0	0	N/A	0	0
gcc	7(1,2)	44	16	10	0
total	25(10,19)	85	29	40	0

Leaks reported by detection tools

Programs	LC	Fastcheck	SPARROW	SABER
art	1(0)	1(0)	1(0)	1(0)
equake	0(0)	0(0)	0(0)	0(0)
mcf	0(0)	0(0)	0(0)	0(0)
bzip2	1(1)	0(0)	1(0)	1(0)
gzip	1(2)	0(0)	1(4)	1(0)
parser	0(0)	0(0)	0(0)	0(0)
ammp	20(4)	20(0)	20(0)	20(0)
vpr	0(0)	0(1)	0(9)	0(3)
crafty	0(0)	0(0)	0(0)	0(0)
twolf	0(0)	2(0)	5(0)	5(0)
mesa	2(0)	0(2)	9(0)	7(4)
vortex	0(26)	0(0)	0(1)	0(4)
gap	0(1)	0(0)	0(0)	0(0)
gcc	N/A	35(2)	44(1)	40(5)
total	25(34)	58(5)	81(15)	70(14)

Experimental Results

- Efficiency

Programs	Size (Kloc)	Crystal Time (sec)		LeakFix Time (sec)				Total Time (sec)
		Preprocessed Code Parsing	CFG Building	A-CFG Building	Detection	Fixing	Total	
art	1.3	0.187	0.034	0.031	0.001	0	0.032	0.253
equake	1.5	0.219	0.068	0.072	0	0	0.072	0.359
mcf	1.9	1.610	0.257	0.045	0	0	0.045	1.912
bzip2	4.6	0.424	0.109	0.097	0.002	0.001	0.010	0.633
gzip	7.8	2.104	0.498	0.276	0.003	0.001	0.280	2.882
parser	10.9	2.653	0.561	0.298	0.021	0	0.319	3.710
ammp	13.3	4.415	0.989	0.351	0.005	0.002	0.358	5.762
vpr	17	3.32	0.675	0.498	0.005	0	0.503	4.498
crafty	18.9	6.177	1.184	0.604	0.006	0	0.610	7.971
twolf	19.7	10.106	1.995	1.012	0.006	0	1.018	13.029
mesa	49.7	16.948	2.312	3.608	0.129	0.058	3.795	23.056
vortex	52.7	14.995	2.555	1.721	0.020	0	1.741	19.291
gap	59.5	8.201	2.170	1.325	0.014	0	1.339	11.710
gcc	205.8	18.701	4.669	4.973	0.526	0.107	5.606	28.976

Experimental Results

- Efficiency (cont.)

Programs	Size (Kloc)	Percentage(%)	Maximum Memory (Megabyte)
art	1.3	12.6	11.2
equake	1.5	20.0	10.6
mcf	1.9	2.4	8.4
bzip2	4.6	15.8	13.2
gzip	7.8	9.7	11.4
parser	10.9	8.6	13.8
ammp	13.3	6.2	15.3
vpr	17	11.2	18.2
crafty	18.9	7.7	21.7
twolf	19.7	7.8	19.7
mesa	49.7	16.5	34.9
vortex	52.7	9.0	40.5
gap	59.5	11.4	38.5
gcc	205.8	19.3	79.3

Example bugs

```
63: int u_v_nonbon( V, lambda )
    float *V,lambda;
65: {
…       …
103:   buffer = malloc( 3*i*sizeof(float) );
104:    if( buffer == NULL)
105:    {aaerror("cannot allocate memory in u_v_nonbon\n"); return 0;}
106:   vector = malloc( i*sizeof(float) );
107:    if( vector == NULL)
108:    {aaerror("cannot allocate memory in u_v_nonbon\n"); return 0;}
109:   atms = malloc( i*sizeof(ATOM *) );
110:    if( atms == NULL)
111:    {aaerror("cannot allocate memory in u_v_nonbon\n"); return 0;}
…      …
118:    for( jj=1; jj<imax; jj++,a1=bp){
…         …
122:       if( (*use)[used] == a1)
123:       { used += 1;}
124:       else { aaerror("error in uselist - must abort"); return 0;}
…          …
178:    }
…       …
183: }
```

Example bugs

```
319: struct gl_texture_image *gl_alloc_texture_image( void )
320: {
321:         return (struct gl_texture_image*) calloc(1, sizeof(struct
                    gl_texture_image) );
322: }

344: static struct gl_texture_image *
     image_to_texture( GLcontext *ctx, const struct gl_image *image,
                       GLenum internalFormat, GLint border )
347: {
349:     struct gl_texture_image *texImage;
…       …
362:    texImage = gl_alloc_texture_image();
…       …
451:        switch (texImage->Format) {
…           …
476:        default:
477:            gl_problem(ctx,"Bad format in image_to_texture");
478:            return NULL;
479:        }
…       …
787: }
```

Outline

1. Memory Leak
2. Existing work
3. Approach
4. Experiments
5. Discussion

Discussion

- Handling loops
  - Loops are handled independently
- Unnecessary fixes
- Related Allocations
  - free(p->q) v.s. free(p)
- Minimal number of fixes

```
if (condition1)
   statement1;
else
   statement2;
statement3;
```

Thanks!