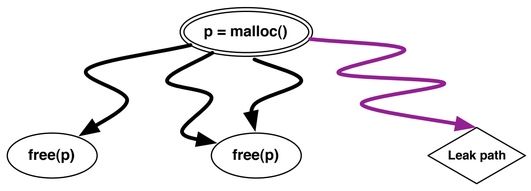
Static Memory Leak Detection Using Full-Sparse Value-Flow Analysis

Yulei Sui, Ding Ye, and Jingling Xue
{YSUI,DYE,JINGLING}@CSE.UNSW.EDU.AU
SCHOOL OF COMPUTER SCIENCE AND ENGINEERING, UNSW

PROBLEM

To find memory leaks **statically** in a program (without actually running it), a leak analysis reasons about a source-sink property: every object created at an allocation site (a source) must eventually reach a free site (a sink) during any execution of the program.



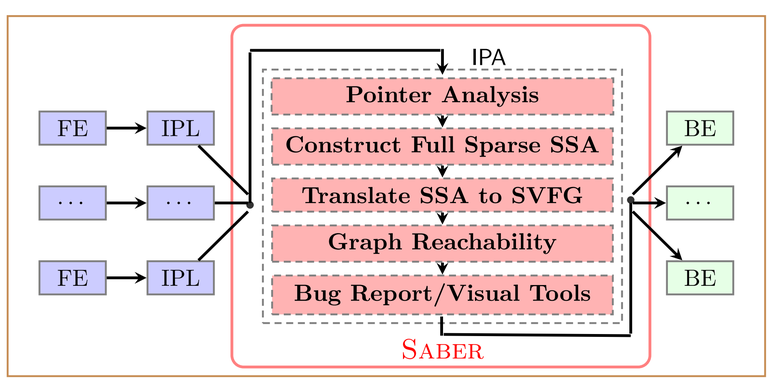
CONTRIBUTIONS

1. The first to find memory leaks by using a full-sparse value-flow analysis to track the flow of values through all memory locations.
2. The first to Leverage recent advances on sparse pointer analysis in a major client application.
3. Effective at detecting 211 leaks at a false positive rate of 18.5% in the 15 SPEC2000 and 5 open-source C programs (totalling in 2324.1 KLOC).

Leak Detector	Speed (LOC/sec)	Bug Count	False Positive Rate (%)
CONTRADICTION	300	26	56
CLANG	400	27	25
SPARROW	720	81	16
FASTCHECK	37,900	59	14
SABER	**10,220**	**83**	**19**

Comparing SABER with other detectors using SPEC2000

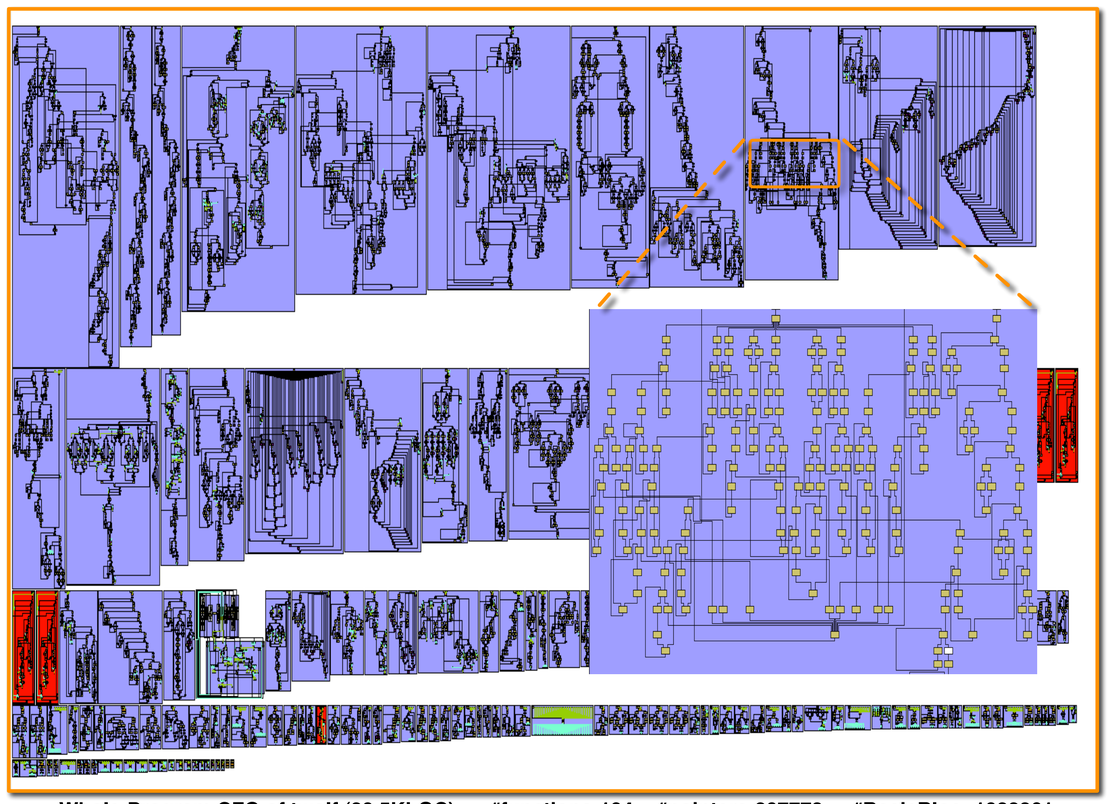
OUR TOOL FRAMEWORK



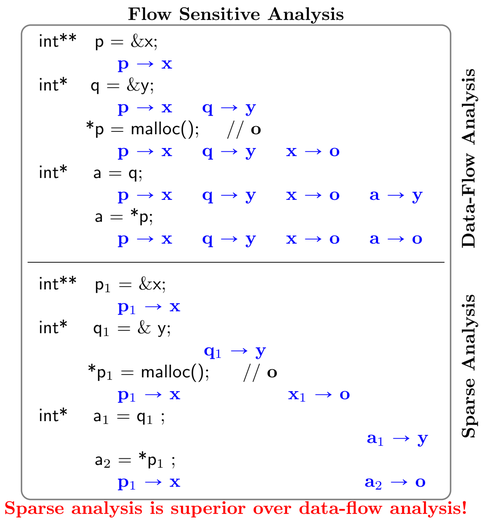
We have implemented our tool **SABER** in Open64, an open-source industry-strength compiler, at IPA (interprocedural analysis module) containing four phases.

IPA: Global analysis by combing information from IPL
IPL: Summary information local to a function
FE: Compiler Front End
BE: Compiler Backend End

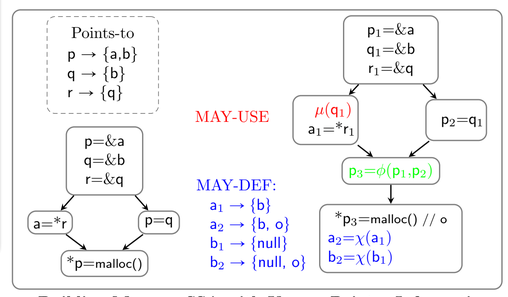
BACKGROUND



Whole-Program CFG of twolf (20.5KLOC) #functions:194 #pointers:207773 #BasicBloc: 1022261
Costly to reason about flow of values on CFGs!!



Sparse analysis is superior over data-flow analysis!

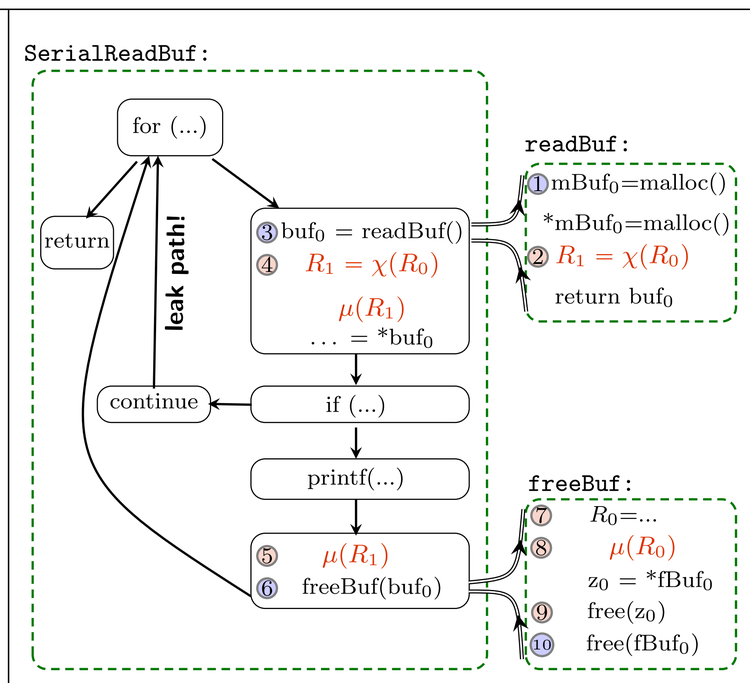


Building Memory SSA with Known Pointer Information

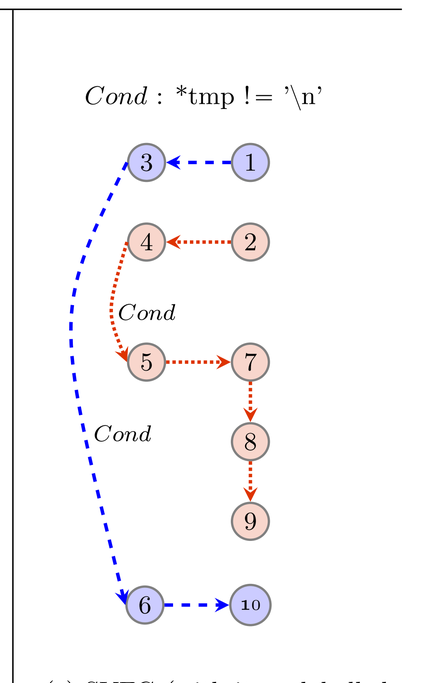
AN EXAMPLE

```
1  void SerialReadBuf() {
2    for (n=0; n<100; n++) {
3      char** buf = readBuf();
4      char* tmp = *buf;
5      if (*tmp != '\n')
6        printf("%s",*tmp);
7      else
8        continue;
9      freeBuf(buf);
10   }
11 }
12 char** readBuf() {
13   char** mBuf = malloc(); //o
14   *mBuf = malloc(); //o'
15   //... (write into **mBuf);
16   return mBuf;
17 }
18 void freeBuf(char** fBuf) {
19   char* z = *fBuf;
20   free(z);
21   free(fBuf);
22 }
```

(a) Input program



(b) Full-sparse SSA on Inter-procedural CFG



(c) SVFG (with its unlabelled edges being guarded by true)

This example, which is adapted from a real scenario in `wine`, demonstrates full-sparse value-flow analysis for leak detection. In this program, **readBuf** is called in a for loop in **SerialReadBuf**. Every time when **readBuf** is called, a single-char buffer formed by two objects is created: o at line 13 and o' at line 14. There are two cases. If the buffer contains a char that is not '\n', the char is printed and then both o and o' are freed. Otherwise, both objects are leaked.

EXPERIMENTAL RESULTS

Real Bugs Found By Saber

```
     //ungif.c
890: GifFileType *
891: DGifOpen(void *userData,
       InputFunc readFunc) {
898:  GifFile = malloc(sizeof(GType));
905:  Private = malloc(sizeof(PType));
911:  GifFile->Private = (void*)Private;
       ...
938:  return GifFile;
     }

944: int
945: DGifCloseFile(GifFileType* GifFile){
947:    GifFilePrivateType *Private;
952:    Private =  GifFile->Private;
964:    free(Private);
972:    free(GifFile);
974:    return GIF_OK;
     }

      //olepicture.c
1002: HRESULT OLEPictureImpl_LoadGif
       (OLEPictureImpl *This, ...){
1006: GifFileType  *gif;
1021: gif = DGifOpen((void*)&gd,...);
       ...
1030: if (gif->ImageCount<1){
1031:   FIXME("images not inside?\n");
1032:     return E_FAIL;
      }
              ...
1194: DGifCloseFile(gif);
1195: HeapFree(GetProcessHeap(),0,bytes);
1196: return S_OK;
     }
```

Relevant leaky code in wine

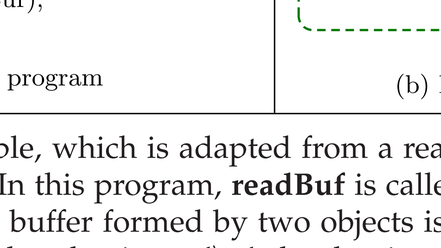
```
      //avl.c
42: avl_node *avl_node_new (void *key,
          avl_node *parent){
45:  avl_node* node = alloc(sizeof(avl));
47:  if (!node) {
48:     return NULL;
49:  }else {
50:     node->parent = parent;
51:     node->key = key;
58:     return node;
       }
     }

116: int avl_insert (avl_tree * ob,
                     void * key){
120:   avl_node* node = node_new(key);
121:   if (!node)
122:     return -1;
       ...
128:  }

      //auth_htpasswd.c
120: static void htpasswd_recheckfile
       (htpasswd_auth_state *passwd){
123:   avl_tree *new_users;
157:   new_users = avl_tree_new (users);
159:   while(get_line(passwdfile,line))
       {
161:    int len; htpasswd_user *entry;
174:    entry = calloc (1,len);
176:    entry->name = malloc (len);
        ...
180:    avl_insert(new_users,entry);
       }
     }
```

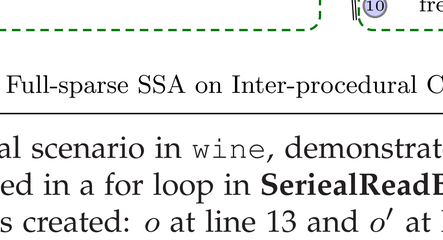
Relevant leaky code in icecast

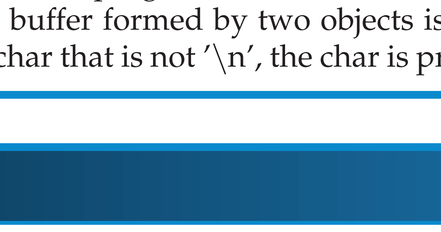
Two scenarios with conditional leaks requiring value-flow analysis for address-taken variables: wine-0.9.24 is a tool that allows windows applications to run on Linux and icecast-2.3.1 is a streaming media server.

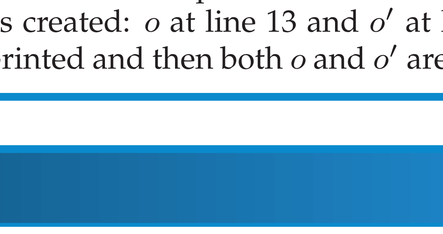
Forward Slices of Program vpr (17.8 KLOC) #SVFG nodes:1993 #SVFG edges:58554

Backward Slices of Program vpr (17.8 KLOC) #SVFG nodes:428 #SVFG edges:3038









Program	Size (KLOC)	Time (secs)	Bug Count	#False Alarm
ammp	13.4	0.55	20	0
art	1.2	0.01	1	0
bzip2	4.7	0.04	1	0
crafty	21.2	0.83	0	0
equake	1.5	0.04	0	0
gap	71.5	4.00	0	0
gcc	230.4	20.88	40	5
gzip	8.6	0.08	1	0
mcf	2.5	0.03	0	0
mesa	61.3	10.10	7	4
parser	11.4	0.28	0	0
perlbmk	87.1	18.52	8	4
twolf	20.5	2.12	5	0
vortex	67.3	2.90	0	4
vpr	17.8	0.31	0	3
bash	100.0	22.03	8	2
httpd	128.1	10.65	0	0
icecast	22.3	5.54	12	5
sendmail	115.2	32.97	2	0
wine	1338.1	390.7	106	21
Total	2324.1	522.58	211	48

SABER's bug counts and analysis times.