A Practical Memory Leak Detector Based on Parameterized Procedural Summaries *

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

Abstract

We present a static analyzer that detects memory leaks in C programs. It achieves relatively high accuracy at a relatively low cost on SPEC2000 benchmarks and several open-source software packages, demonstrating its practicality and competitive edge against other reported analyzers: for a set of benchmarks totaling 1,777 KLOCs, it found 332 bugs with 47 additional false positives (a 12.4% false-positive ratio), and the average analysis speed was 720 LOC/sec.

We separately analyze each procedure's memory behavior into a summary that is used in analyzing its call sites. Each procedural summary is parameterized by the procedure's call context so that it can be instantiated at different call sites. What information to capture in each procedural summary has been carefully tuned so that the summary should not lose any common memory-leak-related behaviors in real-world C programs.

Because each procedure is summarized by conventional fixpoint iteration over the abstract semantics (à la abstract interpretation), the analyzer naturally handles arbitrary call cycles from direct or indirect recursive calls.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification; D.2.5 [*Software Engineering*]: Testing and Debugging—Symbolic Execution; D.3.4 [*Programming Languages*]: Processors—Memory Management(garbage collection)

General Terms Experimentation, Languages, Verification.

Keywords program analysis, memory management, error detection, abstract interpretation, memory leaks, shape analysis.

1. Introduction

A memory leak in a C program is sometimes fatal; it may silently ail the program until memory is exhausted and the program is aborted. A procedure leaks heap memory whenever (1) memory is allocated while the procedure is active and (2) this memory is neither recycled nor visible to its caller after its return.

* This work was partially supported by Fasoo.com and Brain Korea 21 Project of Korea Ministry of Education and Human Resources.

ISMM'08, June 7-8, 2008, Tucson, Arizona, USA.

In this article, we present a practical, fully automatic static analyzer (called SPARROW) that locates memory leaks in C programs. In comparison with other published memory leak detectors [14, 8, 2, 11], our analyzer consistently detects more bugs for the same published benchmark software. Our analysis speed is 720LOC/sec, next to that of the fastest analyzer, FastCheck [2]. Our false-positive ratio (the percentage of alarms that are not true bugs) is 12.4%, which is beaten only by Saturn [14].

C program	Tool	Bug Count	False Alarm Count
SPEC2000 benchmark	SPARROW	81	15
	FastCheck [2]	59	8
binutils-2.13.1 & openssh-3.5.p1	SPARROW	236	19
	Saturn [14]	165	5
	Clouseau [8]	84	269

Table 1. Performance comparison for the same C programs. Other tools' data are from the cited papers. SPARROW found more bugs than others with a reasonable false-alarm ratio.

Tool	C Size KLOC	Speed LOC/s	Bug Count	False Alarm Ratio(%)
Saturn [14]	6,822	50	455	10%
Clouseau [8]	1,086	500	409	64%
FastCheck [2]	671	37,900	63	14%
Contradiction [11]	321	300	26	56%
SPARROW	1,777	720	332	12%

Table 2. Overall comparison with other memory leak detectors. Other tools' data are from [2]. Note that these tools are applied to different programs.

SPARROW separately analyzes each procedure's memory behavior into a summary. Each procedure's analysis summary is used at its call sites. The summary is parameterized so that it can be instantiated differently depending on the memory state at each call site.

Because extracting each procedure's summary is done by the conventional fixpoint iterations over abstract semantics (à la abstract interpretation), the analyzer naturally handles loops, arbitrary call cycles, and aliases within the abstract semantics.

The choice of summary categories has been empirically tuned. The summary categories are chosen after other choices have been tested against realistic C programs. The abstraction decision focuses on not neglecting common memory-leak-related behaviors in realistic C programs.

SPARROW cannot soundly determine that a program is free from memory leaks; it detects some memory leaks but not all.

1.1 Analysis Overview

1.1.1 Summaries and Their Use

The analysis consists of two interlocking processes: (1) the summarization of procedures' memory behavior and (2) the use of these summaries at the procedures' call sites. The summary is then used in analyzing its call site, when its caller's procedural summary is extracted. In the program analysis, summarization of a procedure is triggered when a call site of that procedure is encountered. The order in which procedures are summarized is thus the reverse topological order of the call graph. In the case of call cycles, all procedures in a call cycle are analyzed together within a single fixpoint iteration. In the case of dynamic call edges (due to function pointers), the caller's summarization is delayed until the callee's summary becomes ready.

- **Example** Consider the following example:

```
int *f(List *p) {
   if (n) return malloc(4);
   else {
      p->next = malloc(10);
      return 0;
   }
}
```

```
int *g(){
     List k;
     int *a = f(&k);
     return a;
 }
```

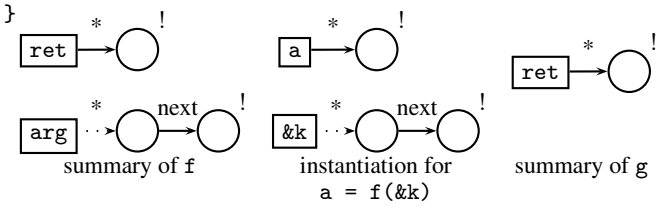


Figure 1. Procedural summary, instantiation, leak detection

The summary representation is explained in Section 2.1. The summary for procedure `f` says that the return value can be a pointer to an allocated cell (the ! marked circle) and the argument pointer's `next` field can point to another allocated cell. Solid arrows are for references created in the procedure body. Dotted arrows are for those that already existed before the procedure was called. The call site `f(&k)` inside `g` uses this summary by instantiating the return and argument boxes with `a` and `&k`, respectively. Among the two allocated cells visible inside `g` (through `a` and `k`), only the one reachable from `a` can be returned, hence `g` can leak memory (reachable from `k`). □

1.1.2 From Memory Effects to Summaries

Summarization of procedures consists of two sequential steps: (1) estimating memory effects and (2) using this estimate to create a summary consisting of information useful in identifying possible memory leaks. The memory-effect estimation step is based on abstract interpretation [3], using fixpoint iteration on our abstract semantics of the C language.

The memory effect for each procedure consists of three pieces of information: allocated addresses, freed addresses, and the exit memory state (memory state at the end of the procedure). From these three pieces of information, it is straightforward to summarize effects related to memory leaks. From the exit memory state, we collect the addresses that are potentially reachable from outside the procedure (via the global variables, the pointer arguments, and the return value). We then examine which among these locations are allocated ones, freed ones, or aliased ones. The results are summarized into the procedure's summary.

Parameterizing Memory Effects by Access Paths One major obstacle in estimating the exit memory state for each procedure is how to derive the exit memory state without knowing the input memory state (call context). We have to parameterize the exit memory state by the procedure's input memory state. How?

Our first observation is that we do not need the whole image of the input memory but only those locations that are accessed by the procedure. Our second observation is that C procedures access the input memory through either arguments or global variables. Our third observation is that although we cannot collect the accessed locations themselves unless we have the input memory, we can determine the "access path" with which those locations are accessed. Such access paths are explicit in the procedure source.

- **Example** For the previous example procedure `f`, the analyzed exit memory states at the two exit points are:

`n`	$[-\infty,-1][1,\infty]$
`ret`	ℓ_1

`n`	$[0,0]$
`p`	α
α`.next`	ℓ_2
`ret`	`null`

First consider the table at the right, which shows the state of the memory at the exit of the `false` branch. The memory has only entries for the accessed locations. The accessed locations are `n` (because of the if-condition), `p` (because of `p->`), α`.next` (because of `p->next`), and `ret` to store the return value.

Note that the abstract location "α`.next`" is the access path along which locations of the input memory can be accessed. The α here is a parameter, which is unknown now but will be instantiated at `f`'s different call sites.

Location ℓ_2 is the abstract, symbolic location allocated at `malloc(10)`. This ℓ_2 will be instantiated as a new symbolic location at each of `f`'s call sites.

The table at the left shows the state of the memory at the exit of the `true` branch. The accessed locations in the `true` branch are `n` (because of the if-condition) and `ret` (to store the return value). The location `ret` has ℓ_1, which is the abstract, symbolic location allocated at `malloc(4)`. This ℓ_1 will also be instantiated as a new symbolic location at each call site of `f`. The location `n` has the shown pair of integer intervals because we don't know anything about this global variable except that it cannot be zero. □

As implied by the above example, we use the well-known integer interval analysis [3] for abstraction of numeric values. Numeric values dependent on the call context are not parameterized; instead, we assume the worst-case: $[-\infty,\infty]$.

Summarizing Each procedural summary describes accessed locations that are reachable from the outside of the procedure. All accessed locations, in parameterized form as access paths, occur as location entries in the exit memory state. Among them, reachable locations from the outside of the procedure are those reachable from global variables, the pointer arguments, or the return value. In the summary, from such reachable locations and the sets of allocated and freed locations we summarize the procedure's behavior.

For the previous example, in the two exit memory states, the reachable locations from outside the procedure are those reachable from `n` (global variable), `p` (pointer parameter), and `ret` (return value). They are ℓ_2 (from `p` via α`.next`) and ℓ_1 (from `ret`). Hence `f` is simply summarized (Figure 1).

1.1.3 Instantiating Summaries

In analyzing a procedure, when we meet a call site we instantiate the called procedure's summary with the call site's memory state. The instantiation consists simply of using the call site's memory to

fill in the blanks in the summary that were parameterized by summarization. Alias information captured at the call site's memory is reflected in instantiation.

The instantiation's output consists of the memory state after the call, along with the updated sets of allocated and freed locations. We track these three pieces of information to the exits of the current procedure using fixpoint iteration, and then we record this information in the current procedure's summary, as explained above.

- **Example** Consider the following procedure `f` and its call site in procedure `g`.

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

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

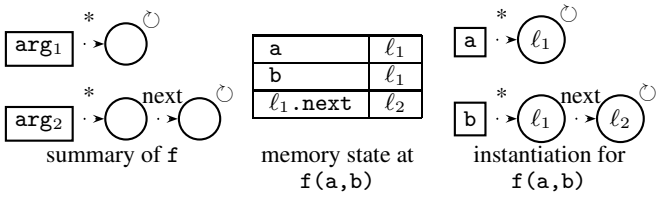


Figure 2. Summary instantiation, reflecting call context.

The summary of `f` shows that locations pointed to by the first argument can be freed (the ⟲ marked node), and locations pointed to by the second argument via access path `next` also can be freed.

At the call site of `f` inside `g`, the two actual parameters `a` and `b` both point to the same location ℓ_1. Instantiating `f`'s summary with this memory state results in the actual summary use (the right-hand-side one) that concludes both ℓ_1 and ℓ_2 can be freed. □

1.2 Contribution

- Our approach tends to be both faster and more accurate than existing analyzers that make a comparable cost-accuracy trade-off. In comparison with other published memory leak detectors [14, 8, 2, 11] using the same benchmark software, our analyzer consistently detects more bugs than the others. The analysis speed (720 LOC/sec) is second only to FastCheck [2][1], and the false-positive ratio (12.4%) is the second smallest, beaten only by Saturn [14]: (10%).
- We present what information to collect in the procedural summary to find an effective trade-off point without path-sensitive and/or global analysis. The information is categorized into eight classes (Section 2).
- In memory leak detection for realistic C programs, we report design decisions (Section 3) of the analysis, some of which are even unsound yet effectively increase the analysis accuracy without much increase in cost.
- We present an analysis method for separately summarizing each procedure's memory leak behavior. We separately analyze each procedure's memory behavior to produce a parameterized summary of it, which will be instantiated in analyzing its call sites. Each procedure's summarization is done by conventional fixpoint iteration over the abstract semantics (à la abstract interpretation [3]). The summary is parameterized for its call context. The call context is the collection of locations accessed by the procedure. The accessed locations are expressed by the access path forms that are explicit in the procedure's source text.

[1] FastCheck's reported speed of 37,900LOC/sec does not count the pointer analysis cost [2] though.

2. Procedural Summary

We analyze memory-leak-related effects of procedures, and we focus on effects visible to the outside of the procedures. Locations visible to the outside of a procedure are those reachable from the global variables, the pointer arguments, and the return value of the procedure.

Understanding a procedure's memory-leak-related effects needs three pieces of information: allocations, deallocations, and aliases. That is, we need to know which allocations inside procedures become visible to the outside, which locations visible to the outside are freed, and which locations visible to the outside are aliased.

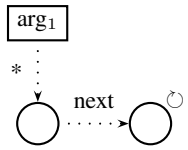
	free	global	argument	return
allocation	-	-	Alloc2Arg	Alloc2Ret
global	-	-	Glob2Arg	Glob2Ret
argument	Arg2Free	Arg2Glob	Arg2Arg	Arg2Ret

Table 3. Eight categories of our procedural summary. The reachable locations from outside and the sets of allocated and freed locations give us memory leak related information.

1. Recording allocations inside procedures that become visible to the outside of procedures: Allocated locations are visible to the outside of a procedure only when they are returned or assigned to the locations of the caller's environment. In C, locations of the caller's environment are reachable via only globals or pointer arguments.

   Thus we record which allocated locations are returned (category Alloc2Ret), or assigned to locations reachable from arguments (category Alloc2Arg). We do not record which allocated locations are assigned to globals (we don't use the category Alloc2Glob), because addresses reachable from global variables are accessible from any environment in the program. However, we miss some leaks that come from interprocedural overwriting of allocated addresses on the same global variable.
2. We record which existing locations are freed or made accessible via another alias.

   For locations that were allocated before the procedure call yet visible inside the procedure, there are two cases. Such locations are accessible via pointer arguments or globals. For those reachable from arguments, we record which locations are explicitly freed (category Arg2Free) and which locations are reachable from the return value (category Arg2Ret) or assigned to locations reachable from globals (category Arg2Glob) or from other arguments (category Arg2Arg). For allocated locations already reachable from globals we don't record which are freed. Locations reachable from globals remain visible from anywhere in the program until the global variable is overwritten with a new address, so they are not of concern for detecting non-interprocedural memory leaks.
3. Recording aliases of locations visible to the outside of procedures: Be reminded that locations visible to the outside are those reachable from the globals, pointer arguments, and return values. Aliases between these three classes of locations happen by assignments between them. Among the nine combinations only five cases are meaningful: assigning locations reachable from globals to locations reachable from arguments (category Glob2Arg) or from return values (category Glob2Ret), assigning locations reachable from arguments to those from globals (category Arg2Glob), to those from arguments (category




```
struct List{
   struct List * next;
   int *val;
};

freeNext(List *lst){
   free(lst->next);
}
```

Figure 3. Arg2Free case: The procedure frees addresses reachable from arguments. The freed location is marked by "♲".

Arg2Arg), or to those from return values (category Arg2Ret). (Note that Arg2Ret, Arg2Arg, and Arg2Glob are already used also for recording the deallocation effects before.) Three of the remaining cases (categories Ret2Glob, Ret2Arg, Ret2Ret) are impossible, because the return value only exists (as such) at the exit of the procedure. In other words, once the procedure returns a value, the procedure is done, so it cannot assign the returned value (as such) to anything. The aliases between globals Glob2Glob are not considered in our analysis because we use a single abstract location for all global variables.

2.1 Examples of the Eight Summary Categories

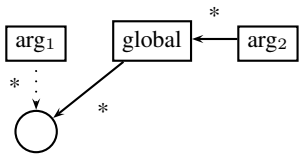
We show examples of how the selected eight categories are captured in the procedural summary. In the examples we will use the same pictorial forms for procedural summaries as we already used before.

A summary is represented by a directed graph. Each node represents one abstract address. Circle nodes are for heap locations (dynamically allocated addresses). Rectangular nodes are for stack locations (for variables, arguments, and globals). Newly allocated heap locations in the current procedure are marked by "!". Freed locations are marked by "♲". Each directed edge from node a to node b indicates that location a may point to location b. The label on the edge indicates the manner in which the predecessor points to the successor. The label can be "$*$" (dereferencing) or the name of a pointer field in a structure. A dotted edge indicates a relationship that already existed before the procedure was called. These locations (those connected by dotted edges) are parameterized locations that will be instantiated by the call site's memory state. A solid edge indicates an effect done by the procedure.

- Arg2Free (Fig. 3): Procedure `freeNext` in the example frees a location reachable from the argument. Its summary is shown in the graph figure.
- Arg2Glob and Glob2Arg (Fig. 4): In the example, the first argument node has one dotted edge and the second argument node has one solid edge. The dotted edge linked at the first argument represents Arg2Glob. In the `attachGlob` procedure, the address node pointed to by first argument `p1` becomes reachable from the global variables node. All global variables in program are abstracted into one global variable node. The solid edge from the second argument represents Glob2Arg.

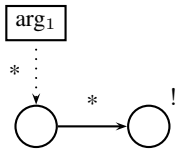
```
int *p1 = malloc();
int **p2;
attachGlob(p1,&p2);
*p2 = malloc();
```

The difference is clearly revealed when instantiating the summary. All allocated addresses in the above code become reachable from global variables. Procedure `attachGlob` makes the allocated location pointed to by `p1` reachable from a global variable. It also makes the pointer `*p2` become an alias of a location



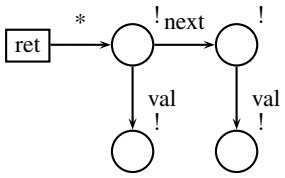

```
int *gInt;
List gLst;

attachGlob(int *p1,
           int ***p2){
   gInt = p1;
   *p2 = &(gLst.val);
}
```

Figure 4. Arg2Glob and Glob2Arg cases: The `attachGlob` procedure attaches some locations reachable from arguments to global variables and attaches locations reachable from global variables to arguments.




```
makeArray(int ** p){
   *p = malloc();
}
```

Figure 5. Alloc2Arg case: The `makeArray` procedure attaches an allocated address to the pointer argument `p`. The allocated node is marked by "!".



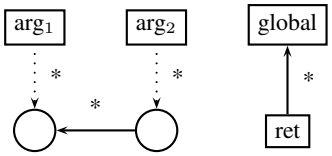

```
List * make2List(){
    List * lst = malloc();
    lst->val = malloc();
    lst->next = malloc();
    (lst->next)->val = malloc();
    return lst;
 }
```

Figure 6. Alloc2Ret case: The `make2List` procedure returns an allocated list of length two.

reachable from a global variable, and therefore the allocated address pointed to by `*p2` after the procedure call is reachable from a global variable.

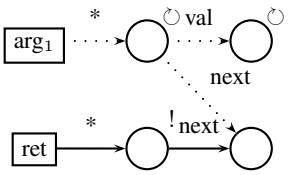
- Alloc2Arg (Fig. 5): In real C programs some procedures attach allocated addresses to pointer arguments. More leaks can be detected by capturing this situation.
- Alloc2Ret (Fig. 6): In real C programs many objects are allocated via procedure calls. It is the most common way to return allocated heap objects. The structures of heap objects are captured.
- Glob2Ret and Arg2Arg (Fig. 7): Some procedures in the Linux kernel return an object from a global table. Allocated addresses attached to this object must not to be reported as leaks.

  Traces of addresses passed through arguments to other arguments should be kept. In the example, the address pointed to by first argument is passed to second argument. This information enables the analyzer to know interprocedural aliases via arguments.
- Arg2Ret (Fig. 8): Some library functions in C (e.g. "memcpy" and "strcpy") return a pointer argument. Variable `ret` and the




```
List * argPassing
(List *lst1,
 List **lst2){
   *lst2 = lst1;
   return &gLst;
}
```

Figure 7. Glob2Ret and Arg2Arg cases: The `argPassing` procedure passes an address from the first argument to the second argument and returns global pointer.




```
List * renewList(List * lst){
  List * ret = malloc();
  ret->next = lst->next;
  free(lst->val);
  free(lst);
  return ret;
}
```

Figure 8. Arg2Ret case: The `renewList` procedure returns addresses reachable from an argument

pointer argument `lst` share a commonly reachable location. This interprocedural aliasing can be captured.

2.2 Interprocedural Summary Instantiation

We show how such procedural summaries are instantiated with simple C code (Figure 9). The `leak` procedure leaks a memory block

```
  void leak(){
1:  List *lst1,*lst2;
2:  int  **ptr;
3:  lst1 = make2List();
4:  lst2 = renewList(lst1);
5:  attachGlob((lst2->next)->val, &ptr);
6:  makeArray(ptr);
7:  freeNext(lst2);
  }
```

Figure 9. The `leak` procedure calls some procedures presented above

pointed to by `lst2` at the exit of the procedure. At line 3, procedure `make2List` is called. The procedure have been analyzed and its summary (Figure 6) can be used. The return value of the summary is instantiated with variable `lst1`. Pointer variable `lst1` becomes a pointer to newly allocated list of length two. At line 4, procedure `renewList` (Figure 8) is called. The first formal parameter and the returned address are instantiated with variables `lst1` and `lst2` respectively. Procedure `renewList` frees two nodes reachable from the first argument. We can trace which addresses are freed by following the access path of the summary. The allocated addresses `*lst1` and `*(*lst1).val` at line 3 are freed. The freed addresses are removed from the allocation set and added to the freed set. The memory state after line 4 is described in Figure 10.

At line 5, one allocated address `(lst2->next)->val` is globalized by procedure `attachGlob`. The pointer `ptr` is aliased with

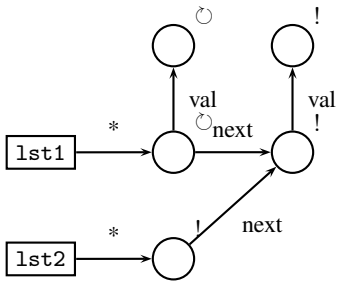


Figure 10. The memory state after line 4 of the code in Figure 9: some allocated addresses are freed and the other allocated addresses are reachable from the pointer variable `lst2`.

a global variable. At line 6, procedure `makeArray` makes one allocated address pointed to by `ptr`. At line 7, one allocated address `lst2->next` is freed by `freeNext`. The final memory state of `clean` is represented in Figure 11.

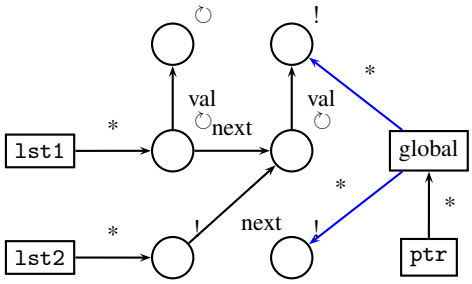


Figure 11. The exit memory state of `leak`: the one allocated address (marked by "!") pointed to by `lst2` is not reachable from global variables, hence leaked.

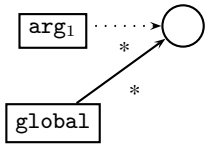
We can see that all addresses but one in the allocation set become reachable from global variables. Hence we conclude that there is a leak in the `clean` procedure.

3. Unsound Decisions for Cost-Accuracy Tradeoff

We use the following techniques (some of which violate the analysis soundness) to silence false alarms and lower the analysis costs, accepting tradeoff of missing a small number of memory leaks.

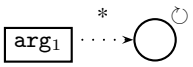
- **Global Variables Abstraction** All global variables in the program are abstracted into one global variable node in the procedural summary. We miss some leaks that come from interprocedural overwriting of allocated addresses stored in same global variable. For example, in the following program, a memory leak (involving the block allocated by `malloc(4)`) is not reported.

```
int *gp;
f(int *p){ gp = p; }
...
   int *p = malloc(4);
   f(p);
   p = malloc(8);
   f(p);
```



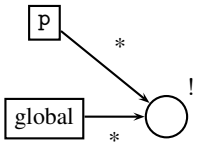
- **Unsound Escaping Effects from Path-insensitivity** Arg2Free collects all freed addresses regardless of the path. So, we might fail to detect a memory leak if the leaked block of memory would have been freed on a different path than the one that is actually taken. For example, the following example program's summary has no path-dependency information:

```
f(int *p, int n){
   if (n) free(p);
}
```



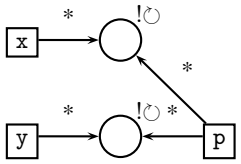
Similarly, assignments to global variables are collected regardless of the path. For example, the following example program's memory states has no path-dependency information:

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



We assume that function `free` frees all locations that may be pointed to by the argument. In the following example program, the memory states indicate that both locations pointed to by the arguments are subject to being freed, though only one is actually freed.

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



- **K-bound Exploration** Deriving memory images from loops is k-bounded, and hence the summaries are finite (Figure 12).

```
freeList(List *p){
   List *t = p;
   while(t != 0){
      free(t);
      p = p->next;
      t = p;
   }
}
```

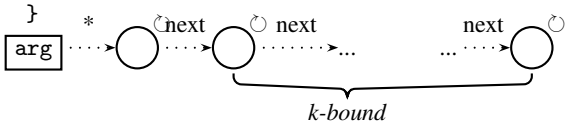


Figure 12. From a k-bounded exit memory state, the summary is k-bounded.

- **Being Sensitive to Memory-Allocating Paths** We have observed that almost all false positives come from the lack of interprocedural path-sensitivity.

```
int foo(int **pp){
 if(n==0) return 0;
 *pp = malloc(n);
 return 1;
}
void bar(){
  int *p;
  if(foo(&p) == 0) return;
  ...
}
```

Procedure `foo` returns the integer 1 whenever a newly allocated location is assigned to its argument. But the summarized return value of this procedure would to be the interval [0,1] (i.e., the return value must be either 0 or 1) because of the `return 0` statement. Whenever procedure `foo` is called, our analyzer assumes that pointer `p` points to a newly allocated address and the return value is the interval [0, 1]. Hence we lose the information regarding the relation between the returned integer and allocating action. So our analyzer falsely reports the allocated address as a potential leak.

Our remedy is, when we collect summary categories from all the exits, if some paths allocate new locations and others do not, instead of joining the all possibilities we choose to summarize only the allocation paths. Hence the return value is determined by only the allocation paths. The summary of the `foo` function is that it returns 1 and attaches an allocated address to the pointer argument.

- **Following Loop Iteration Effects** At flow join points (e.g., a loop head), the allocation set L and the freed set L' of all predecessors are collected. For loops like in Figure 13, it may cause some false positives. Allocated address ℓ remains in the allocation set at the exit even if it is definitely freed in the loop body.

```
p = malloc();
for(i=0;i<10;i++){
   ...
   free(p);
}
return;
```

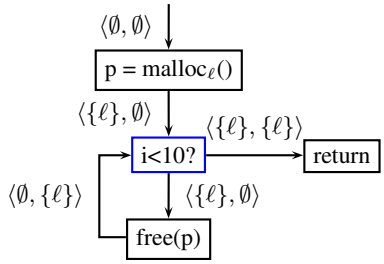
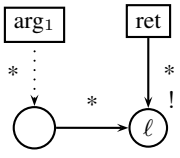


Figure 13. Tuple $\langle L, L' \rangle$ at each edge is the set of allocated and, respectively, freed locations. At the exit, allocated address ℓ remains not to be freed.

When a loop iterates more than once, we do not join with the $\langle \{\ell\}, \emptyset \rangle$ tuple of the initial input memory at the loop head. This choice is based on the heuristic that most loops in programs iterate at least once.

- **Using Names in Addition To Paths in Summaries** This is not unsound. Some procedures return an allocated address and attach the same address to an argument. Because extracting a summary from the exit memory state derives locations in terms of access paths, two different paths whose ends are the same allocated location can be confused to mean different locations. For example, analyzer can misunderstand the semantics of procedure `foo` below as if it allocates two different addresses for the return value and the argument. We use allocation site identifiers in addition to access paths when summarizing procedures.




```
int * foo(int **p){
   int *ret = malloc_ℓ(4);
   *p = ret;
   return ret;
}
```

4. Summarizing Procedures via Fixpoint Iterations

To summarize a procedure we must know what the procedure does. Based on the abstract interpretation framework [3], our analyzer does fixpoint iteration to find memory states at the exits of the procedures in the input program.

During fixpoint iteration, some symbolic addresses are introduced to represent accessed locations in the unknown input memory. From these symbolic addresses an abstract input memory image is derived. Starting with this input memory image, we track the procedure's memory behavior. The procedure's behavior is then summarized from the memory states at the procedure's exit points.

```
int *f(int n, List *p){
 if(n)
    return malloc_ℓ(n);
 return_i p->val;
}
```

Memory states at return statements

n	$([-\infty,-1],[1,\infty])$
ret	ℓ
n	$[0,0]$
p	$\langle \mathtt{p}, i\rangle$
$\langle \mathtt{p}, i\rangle$.val	$\langle\langle \mathtt{p}, i\rangle.\mathtt{val}, i\rangle$
ret	$\langle\langle \mathtt{p}, i\rangle.\mathtt{val}, i\rangle$

Categories of summary

Alloc2Ret	{ret*}
Arg2Ret	{(arg$_2$*.val, ret*)}

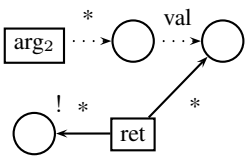


Figure 14. Example code, memory state at return statements, categories of summary and procedural summary.

4.1 Abstract Domains

The input memory state of a procedure is derived from the expressions in the procedure. We introduce "explore-address" $\widehat{Explore}$ (represented as α in previous Section 1.1.2) in order to represent unknown addresses in the input memory states. We always assume that these symbolic addresses are all distinct when summarizing. However these addresses can be instantiated as the same address at a call context (See Figure 2) when instantiating. An explore-address is a pair of an address $\widehat{a}$ and a program point i. It denotes that the address $\widehat{a}$ pointed to this explore address $\langle \widehat{a},\ i\rangle$ at this program point i.

The termination of our analyzer is guaranteed by our abstraction. The *k-bound* exploration[2] limits the number of generating explore-addresses to *k* at every program point. When the number of generating explore-addresses exceeds *k*, the *k*-th explore-address is used instead of generating a new explore-address. Infinitely many allocated addresses are abstracted to their static call sites $\widehat{Region}$. The allocation site is freshly renamed to represent context-sensitive allocations. We abstract numeric values into pairs of integer intervals. We use a widening operation [3] to avoid infinite divergence.

$$
\begin{array}{rcrcl}
\widehat{T} & \in & \widehat{Table} & = & Block \xrightarrow{\text{fin}} \widehat{Memory} \\
\widehat{m} & \in & \widehat{Memory} & = & \widehat{Map} \times \widehat{AllocFree} \\
\widehat{M} & \in & \widehat{Map} & = & \widehat{Addr} \xrightarrow{\text{fin}} \widehat{Value} \\
\widehat{L} & \in & \widehat{Address} & = & 2^{\widehat{Addr}} \\
\ell & \in & \widehat{Region} & = & AllocSite \\
i & \in & PgmPoint & & \\
\langle \widehat{a}, i\rangle & \in & \widehat{Explore} & = & \widehat{Addr} \times PgmPoint \\
\widehat{a} & \in & \widehat{Addr} & = & GVar + ProcId \times Var \\
 & & & & +\widehat{Region} + \widehat{Addr} \times FieldId \\
 & & & & +\widehat{Explore} + Null \\
\widehat{V} & \in & \widehat{Value} & = & \widehat{\mathbb{Z}} + \widehat{Address} \\
 & & \widehat{AllocFree} & = & \widehat{Alloc} \times \widehat{Free} \\
\widehat{AL} & \in & \widehat{Alloc} & = & 2^{\widehat{Region}} \\
\widehat{FR} & \in & \widehat{Free} & = & 2^{\widehat{Region}+\widehat{Explore}}
\end{array}
$$

Table 4. Abstract domains for our fixpoint iteration

The abstract domains for fixpoint iteration are presented in Table 4. $Block$ is the set of all basic blocks in the input program. $GVar$ is the memory storing the global variables (considered as an amalgamated unit). $ProcId \times Var$ is for local variables and the return address. $FieldId$ is the set of field labels for all structures in the program. $\widehat{\mathbb{Z}}$ is a pair of intervals to collect numeric values. The pair of intervals is useful to represent non-zero, negative, and positive integer ranges. The allocation set $\widehat{Alloc}$ and the freed set $\widehat{Free}$ keep allocated addresses and freed addresses respectively. Note that an $\widehat{Explore}$ address can be included in the free set $\widehat{Free}$. The semantics of allocation is to add an allocated address to $\widehat{Alloc}$. The semantics of free is to remove the freed address from $\widehat{Alloc}$ and add it to $\widehat{Free}$.

The analysis result of the fixpoint iteration is a map from each basic block in the procedure to its memory state. Figure 14 shows example code and its analysis results. The value of n is pruned with the condition expression. At the first return statement, the value of n can't be zero and the return address ret points to newly allocated address. At the second return statement, n must be zero. The exploration of memory follows the semantics of expressions. We assume that p points to an explore-address $\langle \mathtt{p}, i\rangle$ when we try to access *p. The i is the program point of the second return statement. Again we assume that $\langle \mathtt{p}, i\rangle$.val points to an explore-address $\langle\langle \mathtt{p}, i\rangle.\mathtt{val}, i\rangle$ when we try to access (*p).val. This address is assigned to the return address ret of the procedure.

[2] We chose five for *k* in the implementation.

4.2 Procedure Summarization from Memories at Exits

A procedure is summarized from the memory states at the exits of the procedure. We need to know which addresses are reachable from certain addresses in order to evaluate all the eight categories of procedural summary. Furthermore we have to know how those addresses can be reached. The *anchor* concept is introduced to represent such access.

$$\psi \in Anchor = (\mathtt{ret} \mid \mathtt{arg}_i \mid \mathtt{global})(\mathtt{*} \mid \mathtt{.f})^*$$

An *anchor* is a list of which the first element is the root address. The symbolic address $\mathtt{arg}_i$ denotes the formal parameters of a procedure, where i denotes the i-th parameter. The symbol "*" is used for dereferencing. The notation ".f" denotes an access through a field pointer of a structure. We present the procedural summary of the f procedure in Figure 14. The Alloc2Ret is a set of *anchors*. Each anchor in Alloc2Ret indicates that a newly allocated address is reachable via the return value. Arg2Ret is a set of tuples of two anchors, the first being an argument and the second being the return value. Each element in Arg2Ret illustrates which address, reachable from an argument in the input memory state, is aliased with an address reachable from the return value in the exit memory states.

Summarizing from Exit Memory State We use several functions for computing certain summary categories from the exit heap state. Given map $\widehat{M}$, $\text{reach}_{\widehat{M}}\,\widehat{L}$ collects all reachable addresses and their *anchors* i.e., the access paths to those addresses from each address in $\widehat{L}$:

$$
\begin{array}{rcl}
\text{reach}_{\widehat{M}} & : & 2^{\widehat{Addr}} \to 2^{\widehat{Addr}\times Anchor} \\
X, S & \in & 2^{\widehat{Addr}\times Anchor} \\
\text{reach}_{\widehat{M}}\,\widehat{L} & = & \text{lfp}^{\subseteq}\,\lambda S.X \cup (F\ S)
\end{array}
$$

where

$$\begin{array}{rcl} X & = & \{(\widehat{a},\widehat{a}) \mid \widehat{a} \in \widehat{L}\} \\ F\ S & = & \bigcup\{(\widehat{a}',\psi\texttt{*}) \mid \widehat{a}' \in \widehat{M}(\widehat{a}), (\widehat{a},\psi) \in S\} \\ & & \bigcup\{((\widehat{a},\texttt{f}),\psi\texttt{.f}) \mid (\widehat{a},\texttt{f}) \in dom(\widehat{M}), (\widehat{a},\psi) \in S\} \end{array}$$

X is the root tuple set of the addresses in $\widehat{L}$. $F\ S$ finds addresses and accesses that can be directly reached from S. A memory location is investigated from the input tuples if reachable via dereferencings or field accesses. The reach operation collects all the reachable addresses and their accesses. The `addr` function picks up all addresses from S.

$$\begin{array}{rcl} S \trianglelefteq S' & = & \texttt{addr}\ S \subseteq \texttt{addr}\ S' \\ \texttt{addr}\ S & = & \{\widehat{a} \mid (\widehat{a},\_) \in S\} \end{array}$$

Let $\widehat{m} = (\widehat{M}, (\widehat{AL}, \widehat{FR}))$ be the memory at an exit of a procedure. Let $\widehat{GL}$ be the reachable addresses from the global variables. Let $\widehat{RL}$ be the addresses reachable from the return value.

The following four categories are calculated from the exit memory state.

$$\begin{array}{rcl} \mathsf{Glob2Arg} & = & \bigcup_i \text{reach}_{\widehat{M}}\{\texttt{arg}_i\} \Cap \widehat{GL} \\ \mathsf{Alloc2Arg} & = & \bigcup_i \text{reach}_{\widehat{M}}\{\texttt{arg}_i\} \Cap (\widehat{AL} - \widehat{GL}) \\ \mathsf{Alloc2Ret} & = & \text{reach}_{\widehat{M}}\{\texttt{ret}\} \Cap (\widehat{AL} - \widehat{GL}) \\ \mathsf{Glob2Ret} & = & \text{reach}_{\widehat{M}}\{\texttt{ret}\} \Cap \widehat{GL} \end{array}$$

The operation $S \Cap \widehat{L}$ collects all anchors in S whose address is also in $\widehat{L}$.

$$S \Cap \widehat{L} = \{\psi \mid (\widehat{a},\psi) \in S, \widehat{a} \in \widehat{L}\}$$

The Glob2Arg is the intersection of reachable addresses from arguments with the reachable addresses from global variables. The Alloc2Arg is the intersection of reachable addresses from arguments with the allocation set. Alloc2Ret and Glob2Ret are similar to Alloc2Arg and Glob2Arg respectively.

Summarizing from Input Memory State In order to calculate the other categories we have to know the input memory state. We infer the input memory state from the output memory state. The use of the input memory state is recorded in the output memory state. We define function Sreach to compute symbolic reachability. This function is exactly equal to the reach function except for the dereferencing part in the $F\ S$.

$$F\ S \quad = \quad \bigcup\{(\widehat{a}',\psi\texttt{*}) \mid \langle\widehat{a},i\rangle \in dom(\widehat{M}), (\widehat{a},\psi) \in S\}$$

For an input address $\widehat{a}$, we follow the explore-address $\langle\widehat{a},i\rangle$ instead of looking up the memory with $\widehat{a}$. For field addresses, Sreach acts the same as reach. Using the function Sreach, the other four categories are computed as follows.

$$\begin{array}{rcl} \mathsf{Arg2Free} & = & \bigcup_i S\text{reach}_{\widehat{M}}\{\texttt{arg}_i\} \Cap \widehat{FR} \\ \mathsf{Arg2Glob} & = & \bigcup_i S\text{reach}_{\widehat{M}}\{\texttt{arg}_i\} \Cap \widehat{GL} \\ \mathsf{Arg2Arg} & = & \bigcup_{ij} \texttt{common}\ (\text{reach}_{\widehat{M}}\{\texttt{arg}_j\})\ (S\text{reach}_{\widehat{M}}\{\texttt{arg}_i\}) \\ \mathsf{Arg2Ret} & = & \texttt{common}\ (\text{reach}_{\widehat{M}}\{\texttt{ret}\})\ (S\text{reach}_{\widehat{M}}\{\texttt{arg}_i\}) \end{array}$$

The `common` $S\ S'$ function collects all pairs of anchors whose addresses belong to both S and S'.

$$\texttt{common}\ S\ S' = \{(\psi,\psi') \mid (\widehat{a},\psi) \in S, (\widehat{a},\psi') \in S'\}$$

The categories Arg2Free and Arg2Glob are the intersections of addresses that were reachable from the arguments with the freed set and the globally reachable set, respectively. These two categories are sets of anchors. The categories Arg2Arg and Arg2Ret are the intersections of addresses that were reachable from the arguments with addresses reachable from other arguments and from the return value, respectively. These two categories are sets of pairs of anchors.

If there are several return statements in the procedure, then, for each category, we take the union of the contents of that category at all the possible exit points. For example, in Figure 14, procedure `f` returns an allocated address at one path and returns a different address at the other path. We summarize this procedure such that it returns an allocated address and returns an address pointed to by the argument.

Reporting Leaks Among the addresses in the allocation set, if there exist addresses that are not reachable from the global variables, the arguments, or the return value. then the addresses are leaked in this procedure:

$$\mathsf{LeakedAddress} = \widehat{AL} - \widehat{GL} - (\texttt{addr}\ \text{reach}_{\widehat{M}}(\bigcup_i \{\texttt{arg}_i\} \cup \{\texttt{ret}\}))$$

For each address in LeakedAddress, the analyzer reports the positions of the allocation site and the return statements. The procedural summaries concerning each leaked address are displayed to help users inspect reported alarms.

4.3 Instantiation of Procedural Summary

A procedure's summary is instantiated at each of its call sites. The values of the actual parameters, addresses that will contain the return value, and the input memory state are the inputs for instantiation. Let $(\widehat{M}, (\widehat{AL}, \widehat{FR}))$ be an input memory state of a procedure call site.

The freed set is updated only by the Arg2Free component of the called procedure's summary. This component contains the set of all access paths that freed memory reachable via arguments. Each access path locates an address which must be freed. The allocation and freed sets are updated:

$$\widehat{AL}' = \widehat{AL} - \widehat{L} \quad \widehat{FR}' = \widehat{FR} \cup \widehat{L}$$

$$\text{where } \widehat{L} = \bigcup_{\psi \in \mathsf{Arg2Free}} \Psi(\widehat{L}_i, \psi, \widehat{M})$$

Note that the source of ψ is $\texttt{arg}_i$. The $\widehat{L}_i$ represents the address of the i-th actual parameter. Function Ψ is used to locate the target addresses. It returns all the addresses that are reachable from $\widehat{L}$ via ψ.

$$\Psi \quad : \quad \widehat{Address} \times Anchor \times \widehat{Map} \to \widehat{Address}$$

$$\begin{array}{rcl} \Psi(\widehat{L},\widehat{a}::t,\widehat{M}) & = & \Psi(\widehat{L},t,\widehat{M}) \\ \Psi(\widehat{L},\texttt{*}::t,\widehat{M}) & = & \Psi(\{\widehat{a}' \mid \widehat{a}' \in M(\widehat{a}), \widehat{a} \in \widehat{L}\},t,\widehat{M}) \\ \Psi(\widehat{L},\texttt{.f}::t,\widehat{M}) & = & \Psi(\{\langle\widehat{a},\texttt{f}\rangle \mid \widehat{a} \in \widehat{L}\},t,\widehat{M}) \\ \Psi(\widehat{L},\texttt{nil},\widehat{M}) & = & \widehat{L} \end{array}$$

This function Ψ is also used to instantiate other components of the summary. The Alloc2Arg and Alloc2Ret in the summary affect both the allocation set and the output memory.

For instantiating Arg2Arg and Arg2Ret, aliases are found by using Ψ. From Glob2Arg, Arg2Glob and Glob2Ret, some addresses are globalized.

5. Experiment Results

We implemented the analysis of this paper in SPARROW . The performance results are presented in the Table 5. We ran our analyzer on a 3.2GHz Pentium 4 machine with 4GB of memory under Linux.

Programs	Size KLOC	Time (sec)	Bug Count	False Alarm
ammp	13.2	9.68	20	0
art	1.2	0.68	1	0
bzip2	4.6	1.52	1	0
crafty	19.4	84.32	0	0
equake	1.5	1.03	0	0
gap	59.4	31.03	0	0
gcc	205.8	1330.33	44	1
gzip	7.7	1.56	1	4
mcf	1.9	2.77	0	0
mesa	50.2	43.15	9	0
parser	10.9	15.93	0	0
twolf	19.7	68.80	5	0
vortex	52.6	34.79	0	1
vpr	16.9	7.85	0	9
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

Table 5. Analysis results on programs from SPEC2000 benchmark and open source programs.

Comparison with FastCheck We have experimented with programs from SPEC2000 benchmarks to compare SPARROW with FastCheck [2]. Analyzing the same set of programs as in [2] except for "perlmbk"[3], SPARROW found 81 bugs among 96 reported alarms and FastCheck found 59 bugs among 67 reported alarms. SPARROW caught all the bugs found by FastCheck except for only two bugs from the "gcc" program.

```
261: osmesa = (OSMesaContext) calloc( 1, sizeof( ...
262: if (osmesa) {
263:   osmesa->gl_visual = gl_create_visual( rgbmode,
     ...
272:   if (!osmesa->gl_visual) {
273:     return NULL;
274:   }

276:   osmesa->gl_ctx = gl_create_context( ...
     ...
279:   if (!osmesa->gl_ctx) {
280:     gl_destroy_visual( osmesa->gl_visual );
281:     free(osmesa);
282:     return NULL;
283:   }
284:   osmesa->gl_buffer = gl_create_framebuffer( ...
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:   }
```

Figure 15. Example code from "mesa"(a SPEC2000 benchmark).

The Figure 15 shows two reported memory leaks from the "mesa" program. From line 261 to 263, the pointer variable `osmesa` points to an allocated heap structure and `osmesa->gl_visual` points to another allocated heap structure. SPARROW successfully captures that procedure `gl_create_visual` returns an allocated heap structure. If the procedure `gl_create_visual` returns a null pointer then the current procedure returns the null pointer at line 273 without freeing the heap structure pointed to by `osmesa`. SPARROW reports this leak. SPARROW is silent at line 282, because all allocated heap Structures (allocated at line 261 and 263) are freed. At line 289, SPARROW reports that some addresses allocated at line 276 are leaked. It seems a false positive at first glance because there is a `gl_destroy_context` function call at line 287. However, they are indeed leaked. By inspecting our procedural summaries we find out that some heap locations allocated by the procedure `gl_create_context` are not freed by the procedure `gl_destroy_context`. FastCheck missed this leak. Figure 16 shows the procedural summary of the creator and the addresses not freed by the destroyer. The analyzer can capture the creation and the destruction of such a complex heap structure through its procedural summaries.

In SPEC2000 benchmarks, our false positives come from several sources: the limitation of our pruning operation for if-conditions (for "gcc"); path-insensitivity between the return value and the condition for allocation (for "gzip"); inaccurate approximation of the number of loop iterations (for "vortex"); and over-approximation on a two-dimensional array (for "vpr").

Comparison with Saturn We analyzed four open source packages: binutils, openssh, httpd and tar. We used older versions of the first two in order to compare with the results reported in existing memory leak detection tools [14, 8]. Open source software packages have several target platforms (e.g. binary, library, ...). Table 5 lists the analysis results for one target that generates the largest number of alarms.

While Saturn found 29 bugs in openssh, SPARROW found 18 bugs. As mentioned above (Section 3), SPARROW misses some bugs due to path-insensitivity in the openssh program: at a flow junction, if one path attaches an address to a global variable and the other path doesn't, then SPARROW assumes that the address is attached to the global, and hence concludes no leaks.

On the other hand, SPARROW found more bugs (228) than Saturn (136) in analyzing binutils. This is because our procedural summary is finer than Saturn's. Saturn fails to follow allocated addresses if they become reachable from the procedure's parameters (category Alloc2Arg). In binutils, more procedures pass newly allocated memory back to their caller via their parameters (491) than via their return value (160). Even if allocated addresses are returned from a procedure, Saturn cannot capture the structure (shape) of the allocated addresses (category Alloc2Ret).

6. Related Work

Whaley and Rinard developed a compositional pointer and escape analysis for Java [13]. The analysis uses parameterized points-to escape graphs that keeps information regarding which memory blocks escape from methods. This information is similar to our procedural summary, but we needed more information in order to detect memory leaks.

Heine and Lam [8, 9] presented a flow-sensitive and context-sensitive C and C++ memory leak detector, Clouseau [8, 9]. They developed a type system to formalize a practical ownership model of memory management. In their ownership model, every object is pointed to by only one owning pointer. The owning pointer takes the responsibility of freeing the object or passing its obligation to another pointer. From this concept they generate constraints for the input program. If constraints are unsatisfiable then there are memory leaks or double deletions. Their analysis generates more false positives than ours.

Orlovich and Rugina [11] proposed a leak detection algorithm that assumes the presence of leaks and runs a reverse heap analysis

[3] We could not analyze the 'perlmbk' program because our parser can not accept many of its files.

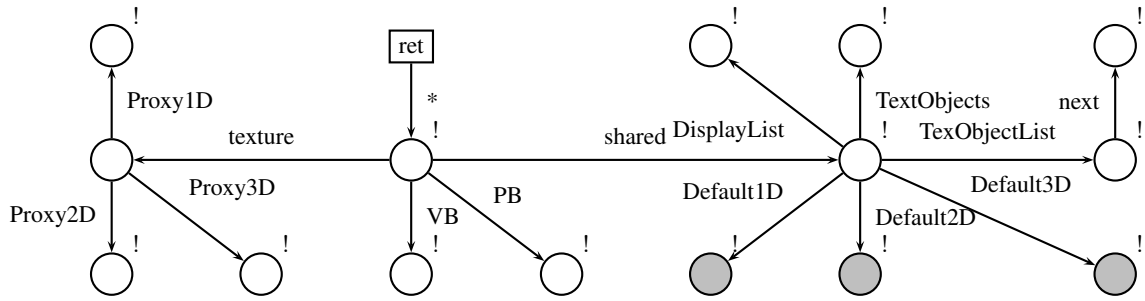


Figure 16. Procedural summary of `gl_create_context`. Nodes are shaded if they are **not** freed by procedure `gl_destroy_context`.

to disprove the assumption. Our analysis finds more bugs with a lower false positive ratio on SPEC2000 benchmarks.

Recently Rugina et al. proposed a new analyzer FastCheck [2] using guarded value-flow analysis. They model memory leak detecting problems using *source-sink* properties. They simplify the program to guarded value flows by reaching definition and branch condition expressions. Their analysis is very fast but additional region analysis is required. They found bugs with a low false positive ratio. Our analyzer can detect more bugs with a similar false positive ratio (differences are 2∼3%) on SPEC2000 benchmarks.

Xie and Aiken [14] presented a Saturn-based memory leak detector. Saturn [14, 15] exploits path-sensitivity from modeling the input program as Boolean formulas. Memory leak detection is reduced to a Boolean satisfiability problem. Their analyzer is context- and path- sensitive, but loops and recursion are handled heuristically. Their analysis takes more time than our analysis.

7. Conclusion

We have presented a practical memory leak detector (named SPARROW) for C programs. In comparison with other published memory leak detectors [14, 8, 2, 11], SPARROW detects consistently more bugs for the same published benchmark software, while the analysis speed is next to the fastest and the false-positive ratio is next to the smallest.

SPARROW analyzes programs in a compositional way; it analyzes each procedure's memory behavior separately and produces a summary of it. The summary is parameterized by the memory state at its call site so that it can be instantiated at different call sites. Our summary categories enable us to find an effective trade-off point without path-sensitive or global analysis. We report design decisions of the analysis, some of which are unsound yet increase the analysis accuracy without much increase in cost.

Acknowledgments

We thank our all ROPAS members for their contributions to SPARROW . We thank Will Klieber for his very kind and detailed comments on our English, as well as technical, presentation. We thank the anonymous reviewers for their helpful comments. We also thank Lonnie Princehouse for his help to compare our analyzer with FastCheck [2].

References

[1] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérome Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. A static analyzer for large safety-critical software. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 196–207, New York, NY, USA, 2003. ACM Press.

[2] Sigmund Cherem, Lonnie Princehouse, and Radu Rugina. Practical memory leak detection using guarded value-flow analysis. *SIGPLAN Not.*, 42(6):480–491, 2007.

[3] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of ACM Symposium on Principles of Programming Languages*, pages 238–252, January 1977.

[4] M. Das, S. Lerner, and M. Seigle. Path-sensitive program verification in polynomial time, 2002.

[5] David Evans. Static detection of dynamic memory errors. In *SIGPLAN Conference on Programming Language Design and Implementation (PLDI '96)*, 1996.

[6] Bolei Guo, Neil Vachharajani, and David I. August. Shape analysis with inductive recursion synthesis. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 256–265, New York, NY, USA, 2007. ACM Press.

[7] Seth Hallem, Benjamin Chelf, Yichen Xie, and Dawson Engler. A system and language for building system-specific, static analyses. In *PLDI 2002*. PLDI, December 2002.

[8] David L. Heine and Monica S. Lam. A practical flow-sensitive and context-sensitive c and c++ memory leak detector. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, pages 168–181, 2003.

[9] David L. Heine and Monica S. Lam. Static detection of leaks in polymorphic containers. In *ICSE '06: Proceeding of the 28th international conference on Software engineering*, pages 252–261, New York, NY, USA, 2006. ACM.

[10] Erick M.Nystrom, Hong-Seok Kim, and Wen mei W.Hwu. Bottom-up and top-down context-sensitive summary-based pointer analysis. In *The proceedings of the 11th Annual International Static Analysis Symposium*, Lecture Notes in Computer Science. Springer, 2004.

[11] M Orlovich and R Rugina. Memory leak analysis by contradiction. In *SAS 2006: 13th Annual International Static Analysis Symposium*, Lecture Notes in Computer Science. Springer, 2006.

[12] J. Reynolds. Separation logic: a logic for shared mutable data structures. In *In Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, 2002.

[13] John Whaley and Martin Rinard. Compositional pointer and escape analysis for Java programs. *ACM SIGPLAN Notices*, 34(10):187–206, 1999.

[14] Yichen Xie and Alex Aiken. Context- and path-sensitive memory leak detection. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 115–125, New York, NY, USA, 2005. ACM.

[15] Yichen Xie and Alex Aiken. Scalable error detection using boolean satisfiability. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 351–363, New York, NY, USA, 2005. ACM.