The missing leak

Jonas Maebe Michiel Ronsse Koen De Bosschere
Department of Electronics and Information Systems
Ghent University, Belgium
{jmaebe|ronsse|kdb}@elis.ugent.be

http://www.elis.ugent.be/diota/

***Abstract*— Memory leaks are caused by allocating memory blocks and not releasing them after their last use. In most modern programs, dynamic data structures are used quite extensively. In this case, only knowing the allocation site of leaked memory blocks, as reported by most current tools, is often not enough to fix said leaks. It is also possible that all pointers to a memory block were overwritten during the execution of a program, in which case more information is required as well. In this paper we describe an approach which uses a combination of dynamic instrumentation and garbage collection techniques to keep track of all memory blocks and their referring pointers. At the end of the execution, we can inform the user where exactly the last pointer to a memory block has been lost, as well as where this pointer was created. Another possibility is tracking a specific memory block during the life time of an execution and reporting the creation and destruction of all references to it. All this can be done without the need for recompilation or relinking.**

I. Introduction

Memory leaks are the result of a memory management problem whereby blocks of memory are never released. Although they can also occur in environments that use garbage collection, they primarily occur in systems where the programmer is directly responsible for the memory management. The consequences of memory leaks range from reduced system performance due to swapping to crashes because of exhaustion of swap or address space.

Several packages which can perform memory leak detection already exist. The necessary instrumentation can happen at different stages. Insure++ [Ins] rewrites the source code of an application. Many leak detectors operate at the library level by intercepting calls to memory management routines, such as in case of LeakTracer [And], memdebug, memprof and the Boehm Garbage Collector [Boe95].

Finally, it is possible to instrument at the machine code level. Purify [Rs] statically instruments the object code of an application and the libraries it uses. Dynamic instrumentors such as Valgrind [NS03] delay the instrumentation until run time.

Only Insure++ can show where the last pointer to a block of memory is lost, the other tools can only show the allocation sites of all leaked blocks. It can however not indicate where this pointer got its value, nor track memory blocks through the execution of a program. Additionally, since it is a source code instrumentation tool, it requires recompilation and cannot provide detailed information about leaks in third-party libraries of which the source code is unavailable.

In this paper, we present a technique that uses dynamic instrumentation at the machine code level to track all pointers to allocated blocks of memory. It is completely language- and compiler-independent and can show where the leaked blocks were allocated, lost and where the last references to these blocks were created.

In what follows, we first give a short overview of the instrumentation framework we use. Next, we discuss the kinds of memory leaks that exist and how they may occur. We then describe how these leaks can be detected and which information must be recorded for this purpose. Finally, we conclude after presenting a short evaluation and discussing our future plans.

II. Instrumentation

The dynamic instrumentation of the binaries, which allows us to perform the necessary analysis of the execution without requiring recompilation or relinking, happens using DIOTA (Dynamic Instrumentation, Optimization and Transformation of Applications). Its inner workings are explained in great detail in [MRB02]. It is a generic instrumentation framework which can be configured on the fly by so-called backends. These are shared libraries which link to the main DIOTA-library and which specify how DIOTA should rewrite the code.

The techniques we will describe rely on only two features of DIOTA: the ability to intercept calls to dynamically linked routines and being notified of memory operations. The former enables us to track the memory allocations and deallocations performed by the program, while the latter is used to track the pointers to the memory blocks as they are passed through the program.

III. Memory leaks

There are two kinds of memory leaks. ZeroFault Software [Gro] calls them logical and physical. A logical memory leak occurs when a block of memory is allocated and never freed afterwards, but at all times during the program execution a reachable pointer to this block of memory exists. A physical memory leak occurs when the last reachable pointer to a particular block of memory is lost.

In both cases, a way to track all pointers to a particular block of memory is required. In this sense, the problem is identical to the classic problem of garbage collection. One can therefore

also choose from the wide variety of known algorithms to perform garbage collection in order to find memory leaks.

Nevertheless, there is one big difference between finding memory leaks and performing garbage collection. In the former case, one wants to know as precisely as possible where exactly pointers to memory blocks are created and destroyed, while in the latter case a periodic cleanup is good enough.

For this reason, we have chosen to use reference counting [Wil92] as opposed to e.g. the more commonly used mark-and-sweep algorithm. Although this increases the overhead significantly and prevents us from detecting leaked cycles, we think that the added detailed information is worth it. Additionally, it is still possible to periodically perform a mark-and-sweep to detect leaked cycles.

IV. Detection

A. Memory blocks

In order to be able to track pointers to allocated memory blocks, one first has to know where those blocks are located. For this purpose, our DIOTA-backend intercepts all calls to malloc, calloc, realloc, free, new, delete and their variants.

For each allocated block, quite a bit of information is recorded, a.o. a reference count and the call stack at the time of allocation

B. References

The second piece of the puzzle is keeping track of all references to these blocks of memory. For each reference we keep track of where it was created, its address and the memory block it refers to.

The information about these references is stored in structures residing in two trees, with one tree reserved for the references residing on the stack. A first reason to separate the stack items from the rest, is that measurements showed that many more references are created and removed on the stack than elsewhere, and at the same time the maximum number of references located on the stack at a single time is often a factor 1000 smaller than the maximum of references residing on the heap.

It thus makes sense to keep the volatile but small group of references on the stack separate from the rest for performance reasons. Additionally, when the stack shrinks, we can keep removing the first item of the stack tree as long as this item's address lies below the new stack pointer, simplifying stack shrinking management considerably.

The bookkeeping of the references can be achieved by looking at the results of all store operations performed by the program. Load operations are largely irrelevant, as most of the time they only result in an extra reference when the value is stored back to memory. Register variables can be handled by looking at the contents of the registers when the reference count of a block drops to zero.

When a value equal to the start address of an allocated block is stored, we increase the reference count of said block. When a previously recorded reference is lost, the reference count of the block it referred to is decreased again. It can be lost in three ways:

- The reference is overwritten with a new value, or some value is added to it. In the latter case, it is possible that the original value will be restored later by subtracting this same value again, so one should take this into account to avoid false positives.
- The reference goes out of scope. For example, it was stored in a local variable or a parameter and the function exits.
- A block of memory containing the reference (to another block of memory) is freed.

C. Leaks

Finally, there are the memory leaks. When the reference count of a block of memory reaches zero, a new *potential* leak is created. The are called potential leaks because there may still be a reference to the possibly leaked block in a register, or it could be that a new reference will be calculated by the program later on (e.g. by substracting a value from a pointer that currently points to somewhere in the middle of that block).

Such a potential leak contains the call stacks of where it occurred, where the last reference, which was just lost, was created and where the leaked block was allocated.

All potential leaks are stored in a hash table, with the hash based on the recorded call stacks mentioned above. Apart from that data, we also record the cause of the leak (as explained in section IV-B). Finally, leaks also have an occurrence count.

Two potential leaks are deemed identical if their causes and their three recorded call stacks match. In such a case, the previously stored potential leak in the hash table is verified to see whether the block it refers to is still leaked.

If it still is, the previously recorded leak is deemed to be permanent and that leak's occurrence count is increased.

D. Reporting

By default, we only report all detected leaks at the end of the execution, but additionally this can also be done every *X* allocation operations. We only do this checking at allocation time, since if the program is not allocating any new memory, any leaks that may have happened are not going to have much influence on the program's operation. We also do not lose any information by delaying the reporting of the leaks.

V. Evaluation

We evaluated our techniques by analysing a few known free software programs (lynx and vim, which turned out not to contain any recurring memory leaks), as well as locally adapted versions of the SimpleScalar simulator and the Fortran Parallel Transformer (FPT) [DZW98]. The slowdown factor lies between 80 and 100 times, which is obviously very significant. The amount of required memory more or less doubles compared to the original execution.

Both SimpleScalar and FPT were known to contain memory leaks from testing with other tools, but without the exact

```
...
1    enode* result = new_enode(polynomial, exp+1, pos+1/*from 1 to m*/);
     for(int i=0;i<exp+1;i++) {
        set<map<lstring,int> > new_terms =
          find_terms_with_var_exp(terms, var_name, i);
5       // fix memory leak found by DIOTA
        value_clear(result->arr[i].d);
        value_clear(result->arr[i].x.n);
        result->arr[i] = translate_one_term(parameter_names,
                                            left_over_var_names,
10                                          new_terms);
     }
...
```

Fig. 1. Bug found in FPT using our technique

location of the actual leaking, fixing them proved to be very hard. An example from FPT is shown in figure 1.

Originally, the `d` and `x.n` fields in lines 6 and 7 were `long int`'s. Afterwards, they were changed into whole numbers with infinite precision from the GNU Multiprecision Library GMP. Before overwriting such values, one has to call the `value_clear()` macro to free previously allocated memory.

While adding such calls throughout 50000+ lines of C++ code, the two that are now at lines 6 and 7 were forgotten. Our tool pinpointed what is now line 8 in the fragment above as the place where the last reference to a block of memory was overwritten.

VI. Future plans

One of our main goals currently is to reduce the overhead of our backend. It has already become more than a factor 30 faster since the start of this project, and we are confident we can reduce it a lot more. We think that if we first run the program and check for memory leaks without tracking all pointers, and afterwards only track the pointers relevant to the leaked blocks, the combined execution time will probably be much lower than when tracking all pointers to all memory blocks.

VII. Conclusion

In this paper, we described how precise memory leak detector can be performed using the reference counting technique. We described the main principles and showed in our evaluation that although the current slowdown is quite big, the results provided by the technique help significantly with finding the root cause of memory leaks. We intend to speed up the implementation and technique in the future.

VIII. Acknowledgements

Jonas Maebe is supported by a grant from the Institute for the Promotion of Innovation by Science and Technology in Flanders (IWT). This research was also funded by Ghent University and by the Fund for Scientific Research-Flanders (FWO-Flanders).

The authors also wish to thank Kristof Beyls for providing ideas, testing consecutive implementations and his invaluable feedback throughout this process.

References

[And] Erwin S. Andreasen. Leaktracer. `http://www.andreasen.org/LeakTracer/`.

[Boe95] Hans Boehm. Dynamic memory allocation and garbage collection. In *Computers in Physics*, volume 9, pages 297–303, May 1995.

[DZW98] E. D'Hollander, F. Zhang, and Q. Wang. The fortran parallel transformer and its programming environment. *Journal of Information Science*, 106:293–317, 7 1998.

[Gro] The ZeroFault Group. Zerofault. `http://www.zerofault.com`.

[Ins] Insure++. `http://www.parasoft.com/`.

[MRB02] Jonas Maebe, Michiel Ronsse, and Koen De Bosschere. DIOTA: Dynamic Instrumentation, Optimization and Transformation of Applications. In *Compendium of Workshops and Tutorials, Held in conjunction with PACT'02: International Conference on Parallel Architectures and Compilation Techniques*, Charlottesville, Virginia, USA, September 2002.

[NS03] Nicholas Nethercote and Julian Seward. Valgrind: A program supervision framework. In Oleg Sokolsky and Mahesh Viswanathan, editors, *Electronic Notes in Theoretical Computer Science*, volume 89. Elsevier, 2003.

[Rs] Er Ro Rs. Purify: Fast detection of memory leaks and access errors. `http://citeseer.nj.nec.com/291378.html`.

[Wil92] Paul R. Wilson. Uniprocessor garbage collection techniques. In *Proc. Int. Workshop on Memory Management*, number 637, Saint-Malo (France), 1992. Springer-Verlag.