Memory Leaks Detection: A Survey

Haoyu Gong
M.S. Computer Engineering, Syracuse University
hgong06@syr.edu

Abstract

Memory leaks have significant impact on software availability, performance, and security. This paper talks about memory leaks in C/C++ itself, and two main methods of detecting them, static analysis and dynamic analysis. Then, it collects the main information of ten most recent tools and compares them in several features. In terms of this comparison, it discusses a few related questions and proposes a possible solution in the future.

Keywords: C/C++; memory leaks; static analysis; dynamic analysis

I. Introduction

The allocation and recovery of memory in the C/C++ language is mainly completed by the programmer when writing the source code. The advantage is that the cost of memory management is low, and the program has higher execution efficiency. However, the drawback is that this management is highly dependent on the level of the programmer. As the overall code size is increased, it's almost inevitable to miss the step of releasing the memory space. Besides, some irregular programming behaviors can also make their program a security risk. According to the US-CERT Vulnerability Notes Databases [1], 39% of all reported vulnerabilities since 1991 were caused by memory leaks or memory corruption. Many system software applications, such as operating system, compiler, development environment, etc, are implemented by C/C++ language. If there is a memory leak problem, it would cause tones of serious consequences. Therefore, both industry and academia are seeking the best ways to check and eliminate memory leaks.

II. Memory leaks

A program will have a memory leak if, during execution, the program loses its ability to address a portion of memory because of a programming error. For example, if a programmer creates an object and forgets to release it after it is no longer needed, the memory space addressed by this object cannot be reused. That will lead to the whole available memory becoming smaller and smaller. Assuming a very large scale program/project has too many memory-leak points, they may cause the program or operating system to crash. That's the reason why people study how to pinpoint and eliminate the memory leaks problem. This article will focus on those memory leak situations in C/C++ language.

II.1. The difference between C++ and Java on memory leaks

Speaking of C++ language, I always think about Java. In fact, the types of memory leaks are quite different between these two languages. In general, there are two scenarios in memory leaks.

In one case, such as in the C/C++ language, allocated memory in the heap, deletes all the ways to access the

memory before it is released (such as a pointer reassigning). Another scenario is that the memory object still retains the memory and the way of how it is accessed (reference) when it is no longer needed.

In the first case, the introduction of a garbage collection mechanism (GC) in Java has been well addressed. Therefore, memory leaks in Java mainly refer to the second case. The C++, on the other hand, contains both of these cases. Therefore, Java's memory breach is manifested as a memory object that lasts longer than the program takes. We sometimes refer to it as "object free."

For example, there are two main types on Java memory leaks. First, it's likely to have a memory leak when long-life objects hold references to short-life-cycle objects. Although short-life objects are no longer needed, they cannot be recycled because long-life objects hold their references.

```
public class Simple {
        Object object;
    public void method1(){
        object = new Object();
        //...other code
    }
```

Fig. 1. Java memory leak sample code

The object here, in fact, we expect it to only work in the method1() method. And it will no longer be used anywhere else. But when the method1() method is executed, the memory allocated by the object "object" is not immediately considered to be an object that can be freed. Only when the object created by the class "Simple" is released, it would be released, which is strictly a memory leak.

```
Vector v = new Vector(10);
for (int i=1;i<100; i++)
{
    Object o = new Object();
    v.add(o);
    o = null;
}
```

Fig. 2. Solution of Fig. 1

The solution is to use "object" as a local variable in the method1() method.

Second, memory leaks in collections, such as HashMap, ArrayList, and so on, are common. For example, when they are declared as static objects, their life cycle is as long as the life cycle of an application, which can easily result in memory shortages.

```
public class Simple {
    Object object;
    public void method1(){
        object = new Object();
        //...other code
        object = null;
    }
}
```

Fig. 3. Sample about collection memory leak

In this example, we loop through the “Object” object and put the requested object into a Vector, and if we only release the reference itself, then Vector still references the object. So the object is not recyclable for the GC. Therefore, if the object must also be removed from Vector after it is joined, the easiest way is to set the Vector object to null.

II.2. The two main techniques in C++ memory leaks

II.2.1. Static memory leaks detection

Based on the reason of memory leaking in C/C++, people attempted both static [2, 3, 4, 5, 6] and dynamic [7, 8, 9, 10] techniques to pinpoint this phenomenon. The static analysis usually searches for memory allocations and the corresponding pairing of those release points. It means, that is to verify all the paths will have the correct memory releasing. Once a path does not contain related memory releasing operations, will be considered a suspected leak. So the static analysis is always to be sensitive to the path. Nowadays, commercial static analysis tools include, for example, HP Fortify [11], Klocwock [12], and Coverity [13]. Since a sound static analysis needs not run the code and does not introduce any execution overhead, these tools have been widely used by real-world developers to find leaks and other memory errors.

II.2.2. Dynamic memory leaks detection

Relatively, the dynamic analysis requires a program to run and monitors the allocation and release of memory resources to determine if a memory leak has occurred. Compared with static technique, the dynamic analysis can detect errors more accurately and find hidden memory leaks. However, it requires a large operating cost, including time and space. One of the most famous dynamic tools is PurifyPlus [14].

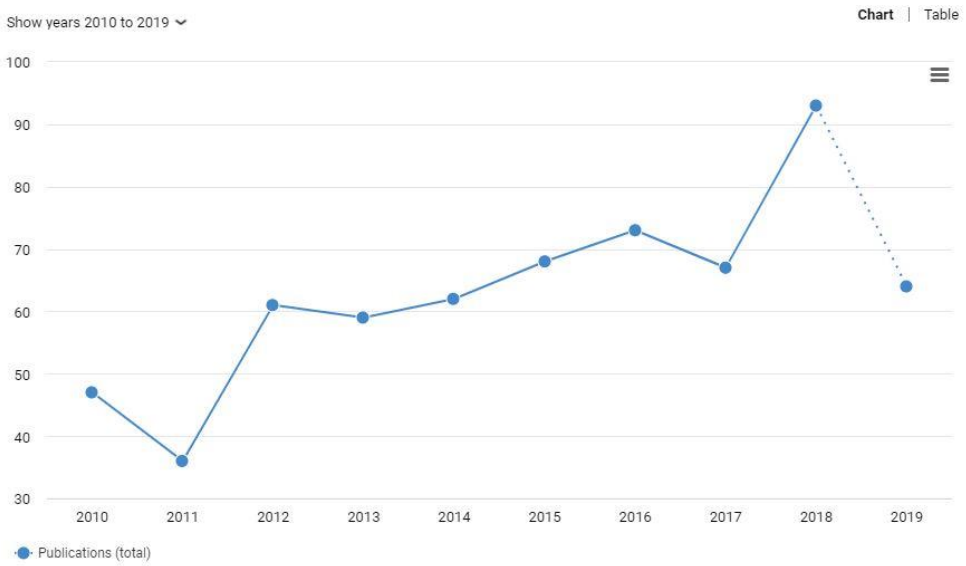

Fig. 4. Publications per year from 2010 to 2019

Fig. 4. shows the number of papers published on memory leak per year from 2010 to 2019. After some early papers on this subject, we can notice a growing number of papers published every tear starting from 2013. If we compare the number of papers published in the early 2010, we can notice an order of magnitude increment in the number of published papers. That demonstrates a growing interesting of the research community on this subject.

III. Software for detecting memory leaks

III.1. Software

III.1.1. Memcheck

http://valgrind.org/docs/manual/mc-manual.html

General Information

Memcheck is Valgrind-1.0.X's checking mechanism bundled up into a tool. All reads and writes of memory are checked. Memcheck will work at two different times: while process is being run, and after the process exits. This tool will keep track of all memory blocks issued in response to calls to malloc/calloc/realloc/new. When the program exits, it gets the information about which blocks have not been returned. It also can detect many other memory-related errors that are common in C and C++ programs and that can lead to crashes and unpredictable behavior:

- Accessing memory after it has been freed;
- Use of uninitialized memory;
- Incorrect freeing of heap memory;
- Overlapping source and destination blocks;
- Passing a fishy (presumably negative) value to the size parameter of a memory allocation function.

Languages supported: C, C++.

Platforms supported: X86/Linux, AMD64/Linux, ARM/Linux, PPC32/Linux, PPC64/Linux, S390X/Linux, MIPS/Linux, ARM/Android (2.3.x and later), X86/Android (4.0 and later), X86/Darwin and AMD64/Darwin (Mac OS X 10.6 and 10.7, with limited support for 10.8).

III.1.2. GlowCode

https://www.glowcode.com/

General Information

The goal of GlowCode is to analyze and optimize application performance, speed and resource use. While the program runs, GlowCode shows the duration, frequency and use of function calls, and identifies which functions play the most significant role in time-intensive tasks, and which execution nodes are the source of multiple memory leaks. This tool is a dedicated commercial product for Windows and .NET Framework, and supports native, managed and mixed code. The other functionalities are

- Finding performance bottlenecks;
- Profiling and tuning code;
- Tracing real-time program execution;
- Ensuring coding coverage;
- Isolating boxing errors;
- Identifying excessive memory usage;
- Finding hyperactive and loitering objects.

Languages supported: 64-bit and 32-bit code written in C, C++, C#, or any .NET Framework-compliant language.

Platforms supported: Windows 10, Windows Server 2019, and prior versions.

III.1.3. AQTime Pro

https://smartbear.com/product/aqtime-pro/overview/

General Information
AQTime Pro is an industrial-level profiling/debugging workbench. It collects necessary performance, memory and resource allocation information at runtime and forms a detailed report. AQTime Pro has both static and runtime profilers. Profilers of the static analysis do not run the program. But the runtime profiler launch the program. It gives the developers full options to analyze their application. AQTime Pro also includes profiling for:

- Performance bottlenecks;
- Code coverage analysis;
- Fault simulation.

Languages supported: C/C++ applications (Visual C++, C++Builder, GCC, Intel C++), Delphi applications, .NET 1.x–4.5 applications, Silverlight 4 applications, Java 1.5 and 1.6 applications, 64-bit applications, JavaScript, JScript and VBScript scripts.

Platforms supported: Windows

III.1.4 WinDbg

https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/debugger-download-tools

General Information
WinDbg is a great, free multipurpose debugger. It is more powerful than Visual Studio's built-in debugger, but is much harder to use. The WinDbg debugger plugin uses Microsoft's Debugging Engine COM interfaces from the Debugging Tools package. It is used to identify kernal memory dumps and examine the CPU register.

Languages supported: C++, etc

Platforms supported: Windows

III.1.5 BoundsChecker

https://www.microfocus.com/products/devpartner/

General Information
BoundsChecker is a tool for memory and API validation tool for C++ software. Right now, it belongs to a

larger tool suite, called DevPartner. It can detect memory leaks by monitoring API and COM calls. BoundsChecker has two different modes: ActiveCheck detects memory leaks by monitoring API and COM calls, or FinalCheck can detect buffer overflow and undefined memory. Other functionalities:

- Memory overrun detection;
- API call logging;
- .NET analysis;
- Deadlock analysis.

Languages supported: C++

Platforms supported: Intel x86 platform.

III.1.6. Deleaker

https://www.deleaker.com/

General Information

Deleaker is a tool for Visual C++, .Net and Delphi developers designed to help them find various leak types and memory leaks in the code. Deleaker is a dynamic tool to detect memory leaks. Basically, it hooks into every possible resource allocation function and its counterparts. When the program is closed, Deleaker reports leaks that were not released to the system.

Languages supported: C++, C#, .NET, and Delphi

Platforms supported: Windows

III.1.7. Dr. Memory

https://drmemory.org/

General Information

Dr.Memory is a memory checking tool that operates on both Windows and Linux applications. It handles the complex and not fully documented Windows environment, and avoids reporting false positive memory leaks that plague traditional leak locating algorithms. Similar in style to Memcheck, which works only on Linux and MacOS. Dr Memory is an open source memory checking tool built on top of a JIT-based instrumentation framework called DynamoRIO. In terms of detecting memory leaks problem, Dr.Memory has a light mode, and runs faster than Memcheck. It also could identify memory-related errors such as:

- Accesses of uninitialized memory;
- Accesses to unaddressable memory (including outside of allocated heap units and heap underflow and overflow);
- Accesses to freed memory, double frees;
- GDI API usage errors;
- Accesses to un-reserved thread local storage slots.

Languages supported: C++, etc

Platforms supported: Windows, Mac, Linux, or Android on commodity IA-32, AMD64, and ARM hardware.

III.1.8 Intel Inspector XE

https://software.intel.com/intel-inspector-xe

General Information

Intel Inspector XE helps for early detection of memory leaks and helps to reduce expenses for fixing memory leaks. The highlight of this tool is that it performs both static and dynamic analysis to identify the reason of memory leaks. However, it does not support GNU OpenMP runtime and may report false positives for OpenMP codes compiled by GCC. Other functionalities:

- Dangling pointers;
- Uninitialized variables;
- Use of invalid memory references;
- Mismatched memory;
- Allocation and deallocation;
- Stack memory checks;
- Stack trace with controllable stack trace depth.

Languages supported: C, C++, and Fortran

Platforms supported: Windows, and Linux.

III.1.9. Insure++

https://www.parasoft.com/products/insure

General Information

Unlike those tools above, Insure++ is a static tool. It is a C source code instrumentor that can verify memory accesses. It is a debugging tool, but since many vulnerabilities result from bugs, it can also be used as a security tool. The source code of the program being checked is required. Like other tools of this kind, Insure works by adding code that prints error messages when a bad memory access is made. It also can be used to find:

- Memory abuse;
- Buffer overflow/underflow;
- Pointer abuse;
- Many other causes of possible undefined behavior or implementation-defined behavior.

Languages supported: C, C++

Platforms supported: Windows, Linux, AIX, Solaris, HPUX

III.1.10. Visual Leak Detector for Visual C++ 2008-2015

https://kinddragon.github.io/vld/

General Information

Visual Leak Detector is a free, open-source memory leak detection tool for C/C++. When you run your program under the Visual Studio debugger, Visual Leak Detector will output a memory leak report at the end of your debugging session. The leak report includes the full call stack showing how any leaked memory blocks were allocated. Therefore, customizable and detailed memory leak reports are the best feature of this tool.

Languages supported: C, C++

Platforms supported: Windows

III.2. Analysis

	Languages Supported	**Platforms Supported**	**Method**	**Availability**
Memcheck	C, C++	Linux, Android, Darwin	Dynamic	Free
GlowCode	C, C++, C#, .NET	Windows	Dynamic	Need to pay
AQTime Pro	C, C++, Delphi, .NET, Sliverlight, Java, JavaScript, VBScript	Windows	Static / Dynamic	Need to pay, free trial included
WinDbg	C++, etc	Windows	Dynamic	Free
BoundsChecker	C++	Intel x86 platform	Dynamic	unknown
Deleaker	C++, C#, .NET, Delphi	Windows	Dynamic	Need to pay, free trial included
Dr. Memory	C++, etc	Windows, MacOS, Linux, or Android	Dynamic	Free
Intel Inspector XE	C, C++, Fortran	Windows, Linux	Static / Dynamic	Need to pay
Insure++	C, C++	Windows, Linux, AIX, Solaris, HPUX	Static	Need to pay
Visual Leak Detector for Visual C++ 2008-2015	C, C++	Windows	Dynamic	Free

Table. 1. Analysis of the selected tools

III.2.1. Why a few static tools?

In this chapter, I selected 10 tools which are the most recent and the most widely used. Looking at the table above, it's not difficult to find that almost every popular tool has their own dynamic method to achieve the goal, but just a few of them have static ways. In fact, since the static analysis does not run code, and the degree of automation is high, it has been widely used in the academic and industrial sectors. However, because static analysis adopts a conservative strategy, the results often contain a large number of false positives, and need to be further manual confirmation to identify false positives. A highly precise static analysis usually cannot scale to large code base. Besides, manual confirmation of static analysis results time-consuming and error-prone, and seriously limits the practicality of static analysis technology. Therefore, it's not surprising that more and more commercial tools choose dynamic analysis.

III.2.2. Verification of static analysis results and elimination of false positives

The truth is, many scholars are looking for corresponding solutions. And many of them have already made a progress. Ruthruff et al. [15] predicts the type of static alerts after you mine potential information in existing alerts and associates code to establish a statistical-based regression model. Kim and Ernst [16] use software change history to find the relationship between static analysis alerts and actual bug fixes, and then propose a history-based alert priority (HWP, History-based Warning Priority) processing algorithm. This takes advantage of the history of the fix to sort the alerts. Heckman et al. [17] propose a technique to reduce false positives and validate it with a baseline assembly, FAULTBENCH. FEEDBACK-RANK [18] uses known corrections to static analytical alerts, and develops a probability-based sorting method to reduce the number of false positives. Z-Ranking [19] prioritizes alerts by manually validating statistics and frequencies for static analysis results. Boogerd and Moonen [20] sort static analysis alerts by using a plausible analysis.

III.2.3. Why not mix them up?

In these related work, most of the results of static analysis are performed secondary lying, and not combined with dynamic implementation. Therefore, it's not too hard to think of mixing them up. Since both two methods have their own pros and cons, then why not develop a mixed tool to handle this issue?

In fact, many developers will select their own favorite tools, which highly suits with their ongoing project. The core functionality of detecting memory leaks is to ensure software not vulnerable. Therefore, it's not worthy spending too many times on this topic. Besides, some company provided a static profiler and a dynamic profiler for their customer to choose, such as AQTime Pro. It's pretty difficult to design a complete version to detection all the types of memory leaks.

III.2.4. Mixed execution testing

There is another technique, named mixed execution testing (concolic testing [21]). It is the combination between concrete execution and symbolic execution. The main idea of mixed execution testing is about testing the program. It collects both symbolic execution and symbolic path constraints in the process of concrete execution, and to use constraint solution to generate test input, so as to explore the program path space and find code defects.

Mixed execution testing has many advantages. For example, in symbol execution, once we want to get input from a path, we need to first get all the condition constraints in that path to be constrained. While the mixed execution testing is generated by the edge execution, and the specific values can be used to simplify the

constraints of symbol execution. However, mixed execution testing do not take the prior knowledge of target defects as a guide, resulting in a large number of useless test inputs, wasting time and resource overhead, and affecting test efficiency.

Till now, some people have also combined dynamic and static techniques with each other. The DyTa tool [22] combines static program validation with dynamic test generation to reduce false positives and improve efficiency. The DSD-Crasher tool [23] uses a dynamic-static-dynamic approach in turn to find program errors. Zhang and Saff, etc. [24], propose a static combination method for unit testing, which is used to automatically produce effective and diverse sequences of method calls. In the future, I believe mixed execution testing would make a contribution to many fields, including memory leaks detection.

IV. Conclusion

This paper analyzes two main methods in detecting memory leaks, and ten the most recent tools. From the table.1., we could realize the difference among those ten and the mainstream trend of commercial market. Based on the truth that dynamic tools occupies the market, I found out the reason why static tools are not that popular. In the end, I discussed why not mix both them up, and a novel testing method called mixed execution testing, which might be a new break-through point of memory leaks problem.

No matter how, market is changing fast. And ten tools cannot represent the whole market right now. Many old and traditional tools are not mentioned in this paper, such as Purify. In the future, I will try to add more tools into analysis and find more better possible solutions.

V. References

[1] US-CERT vulnerability notes database. http://www.kb.cert.org/vuls.

[2] M. Orlovich and R. Rugina. Memory leak analysis by contradiction. In SAS, pages 405–424, 2006.

[3] Y. Xie and A. Aiken. Context- and path-sensitive memory leak detection. In FSE, pages 115–125, 2005.

[4] D. L. Heine and M. S. Lam. Static detection of leaks in polymorphic containers. In ICSE, pages 252–261, 2006.

[5] D. L. Heine and M. S. Lam. A practical flow-sensitive and context-sensitive C and C++ memory leak detector. In PLDI, pages 168–181, 2003.

[6] S. Cherem, L. Princehouse, and R. Rugina. Practical memory leak detection using guarded value-flow analysis. In PLDI, pages 480–491, 2007.

[7] R. Hastings and B. Joyce. Purify: A tool for detecting memory leaks and access errors in C and C++ programs. In Winter 1992 USENIX Conference, pages 125–138, 1992.

[8] F. Qin, S. Lu, and Y. Zhou. SafeMem: Exploiting ECC-memory for detecting memory leaks and memory corruption during production runs. In HPCA, pages 291–302, 2005.

[9] G. Novark, E. D. Berger, and B. G. Zorn. Efficiently and precisely locating memory leaks and bloat. In PLDI, pages 397–407, 2009.

[10] J. Clause and A. Orso. LEAKPOINT: pinpointing the causes of memory leaks. In ICSE, pages 515–524, 2010.

[11] HP Fortify. https://www.microfocus.com/en-us/products/static-code-analysis-sast/overview.

[12] The Klocwork static analysis tool. http://www.klocwork.com/.

[13] The Coverity static analysis tools. http://www.coverity.com/, 2012.

[14] PurifyPlus. https://www.teamblue.unicomsi.com/products/purifyplus/.

[15] J. R. Ruthruff, J. Penix, J. D. Morgenthaler, S. Elbaum, and G. Rothermel. Predicting accurate and actionable static analysis warnings: An experimental approach. In ICSE, pages 341–350, 2008.

[16] S. Kim and M. D. Ernst. Prioritizing warning categories by analyzing software history. In MSR, pages 27–, 2007.

[17] S. Heckman and L. Williams. On establishing a benchmark for evaluating static analysis alert prioritization and classification techniques. In ESEM, pages 41–50, 2008.

[18] T. Kremenek, K. Ashcraft, J. Yang, and D. R. Engler. Correlation exploitation in error ranking. In FSE, pages 83–93, 2004.

[19] T. Kremenek and D. Engler. Z-ranking: using statistical analysis to counter the impact of static analysis approximations. In SAS, pages 295–315, 2003.

[20] C. Boogerd and L. Moonen. Prioritizing software inspection results using static profiling. In SCAM, pages 149–160, 2006.

[21] Sen, K., Marinov, D., Agha, G.: CUTE: a concolic unit testing engine for C. In: Wermelinger, M., Gall, H.C. (eds.) FSE 2005, pp. 263–272. ACM (2005).

[22] X. Ge, K. Taneja, T. Xie, and N. Tillmann. DyTa: dynamic symbolic execution guided with static verification results. In ICSE, pages 992–994, 2011.

[23] C. Csallner, Y. Smaragdakis, and T. Xie. DSD-Crasher: A hybrid analysis tool for bug finding. ACM Trans. Softw. Eng. Methodol., 17(2):8:1–8:37, May 2008.

[24] S. Zhang, D. Saff, Y. Bu, and M. D. Ernst. Combined static and dynamic automated test generation. In ISSTA, pages 353–363, 2011.