Paper ID 5efd6136-ea37-4566-bc66-98d0c0ce9e29

Memory Leak Detection and Prevention Mechanisms

B.Sai Ashish[1],Nirbhay Kumar Pandey[2], *B.Vinay.*

***Abstract*—Memory leak are the blocks of allocated memory that the program in which there majorly of two types Explicit and Implicit.In the Explicit the memory leak is mistake made by the programmer by his negligence or the error in the program. An Implicit this occurs when a system behaviour or the function of the user such as Memory Fragmentation,Circular reference or by the External Factors.These both are the memory leaks[1] that we use to detect and debug these in especially in the Large components or large complex programs.These are the some for each buffer its finds and displays the imformation**

- **The location of the leaked memory**
- **The size of the leak (in bytes)**
- **The contents of the leaked buffer**

Causes of memory retention can stem from various sources. One common cause is the failure to properly release dynamically allocated memory, resulting in a gradual consumption of system resources. This can occur when references are lost without proper deallocation, or when resources like file handles are left unclosed.

***Index Terms*—Memory leak detection, Dynamic memory management, Memory deallocation, Memory leak detection tools.**

I. Introduction

Memory retention occur when the program get to fail release memory it has allocated, which can lead to reduced system performance and even crashes. Detecting and to deal with leaks is crucial for protect safety of software applications.

Memory leak detection involves using tools and techniques to identify areas in a program where memory is allocated but not properly released. This can be done from the use of specialized debugging tools, profilers, and memory analysis software that can pinpoint the specific locations in the code where memory leaks are occurring.

To prevent memory retention, developers[2] can implement best practices such as properly managing memory allocation and deallocation, using smart pointers and automated memory management techniques, and conducting regular code reviews to identify potential memory management issues.

By incorporating memory leak detection and prevention strategies into the software development process, developers can create more reliable and efficient applications that deliver a better user experience.

Static program analysis is an approach used to analyze computer programs without actually executing them. It involves examining the code structure, syntax, and dependencies to identify potential issues such as memory leaks, security vulnerabilities, and performance bottlenecks.
memory leak detection and prevention tools, as well as static program analysis tools, can analyze source code to pinpoint problematic coding practices that could lead to memory leaks. These tools are designed to flag instances where memory allocation and deallocation are not handled correctly, enabling developers to fix these issues before the code is executed.
By applying the static program analysis, developers can gain insights into the codebase and identify potential memory management issues early in the development process, leading to more robust and reliable software with fewer memory-related issues tools, such as METAL, PREfix , Clouseau and CSSV.

II. Procedure for Paper Submission

The presenter [3] have developed a method using projections to spot memory leaks in complex control flows in C programming. This involves simplifying the original control flow graph by considering memory allocation and deallocation properties in C source code, which reduces the complexity of analysis.

The presenter [4], have introduced SMOKE, a phased approach to tackle memory leaks. Instead of applying a uniform precise analysis to all pathways initially, they use a scalable but imprecise analysis to identify a concise set of potential memory leak paths.

Furthermore the presenter have suggested a technique to access the effectiveness of static analysis tools in detecting different types of memory leaks. They start by categorizing memory leak vulnerabilities into 11 categories based on heap memory patterns and software architecture.

Additionally, the presenter have demonstrated a dynamic method for identifying and resolving memory leaks. They achieve this by instrumenting the program before execution and using dynamic symbolic execution to uncover memory breaches across all execution pathways.

Moreover, in [7], they have presented a unique hybrid automatic memory leak detection technique designed for soft real-time embedded system software. This methodology combines static and dynamic techniques to address unique constraints, using source code annotation and control flow graphs in the static phase, and simulating abstracted memory behavior using an abstract memory model (AMM) in the dynamic phase.

The existing literature on memory leak detection mainly focuses on different techniques for identifying memory leaks, with limited discussion on available memory leak detection tools. This paper aims to offer a brief overview of such tools and compare them based on various parameters.

III. MEMORY LEAK DETECTION TOOLS

Memory leaks can cause slowdowns, crashes, and security issues in an operating system. Memory leak detection tools monitor memory usage and allocation of applications and processes, and help identify the source and location of memory leaks. Here are some memory leak detection tools for different operating systems:

Windows

Valgrind, LeakTracer, LeakSanitizer, Memcheck, Memory Validator, Windows' Memory Diagnostics Tool

C/C++

CRT debug heap functions, C/C++ debugger

Linux

- GNU malloc, mtrace, Dmalloc, Electric Fence, Dbgmem, Memwatch, Mpatrol, Sar

Java

Eclipse Memory Analyzer (MAT), VisualVM, NetBeans Profiler, JProfiler, GC Viewer, Plumbr, nmon

Memory leak detection tools are essential for identifying and addressing memory management issues in software applications. Some popular memory leak detection tools include:

1. Valgrind: Valgrind is a widely used instrumentation framework for building dynamic analysis tools. It includes a tool called Memcheck, which can detect memory leaks, as well as other memory-related errors such as using uninitialized memory and accessing memory after it has been freed.

2. Purify: Purify is a memory debugging tool that can detect memory leaks, access errors, and other memory-related issues in C, C++, and Java programs. It provides detailed information about memory allocation and deallocation, helping developers pinpoint the source of memory leaks.

3. AddressSanitizer: AddressSanitizer is a runtime memory error detector designed to find out-of-bounds accesses and use-after-free bugs. It can also detect some types of memory leaks by keeping track of allocated memory blocks.

4. LeakSanitizer: LeakSanitizer is a memory leak detector that is part of the AddressSanitizer tool suite. It is designed to detect memory leaks in C and C++ programs by tracking dynamically allocated memory blocks and reporting those that were not deallocated before the program exits.

5. Electric Fence: Electric Fence is a memory debugging library that can help detect buffer overruns and underruns, as well as memory allocation and deallocation issues. It does this by placing "guard" pages before and after each dynamically allocated buffer, which can detect out-of-bounds memory accesses.

These tools provide developers with the means to identify and diagnose memory leaks in their code, ultimately leading to more stable and reliable software applications.

6. It is a heap analyzer, which tells the amount of heap memory used by the program. It tells the useful space and extra bytes allocated for alignment and book-keeping purposes. It can also show the stack memory used by the program (not the default behavior). The profiling data collected by this tool is written to a file.

Helgrind:

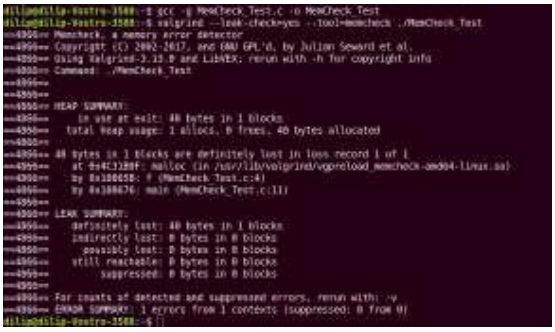
It is a tool to detect thread synchronization related errors. It basically detects 3 types of errors: misuse of POSIX pthreads API, deadlock and data race problems.

Figure 2 shows the code which suffers from memory leak issue. Inside the function f(), malloc() allocates the dynamic memory on heap and the reference to that memory is saved inside pointer x. But the programmer forgets to deallocate that memory after use, leading to the main memory space leak issue.

```
void f(void) {
  int* x = (int *)malloc(10 * sizeof(int));
  //use the dynamically allocated memory
}

int main(int argc, char* argv[]) {
  f();
  return 0;
}
```

When memcheck tool is run on code shown in Fig. 2, the memory leak issue observed is as shown in Fig. 3



```
void f(void) {
  int* x = (int *)malloc(10 * sizeof(int));
  //use the dynamically allocated memory

  free();
}

int main(int argc, char* argv[]) {
  f();
  return 0;
}
```

Fig 3: Memory Leak detected after running memcheck tool

There are 2 solutions to avoid memory leak in this code: (1) Use free() before the end of f(), which will deallocate the dynamically allocated memory by malloc() as shown in Fig. 4.

```
void f(void) {
  int* x = (int *)malloc(10 * sizeof(int));
  //use the dynamically allocated memory

  free();
}

int main(int argc, char* argv[]) {
  f();
  return 0;
}
```

Fig 4: Using free() inside called function to avoid memory leak

If the function f() is made to return the pointer x, then the caller function main can also free the allocated memory on heap as shown in Fig 5.

```
int f(void) {
  int* x = (int *)malloc(10 * sizeof(int));
  //use the dynamically allocated memory

  return x;
}

int main(int argc, char* argv[]) {
  int * p= f();
  free(p);
  return 0;
}
```

Fig 5: Using free() inside caller function to avoid memory leak

Fig 6 shows that there is no memory leak after using the free().

Fig 6: Memory leak issue is resolved after fixing the code

B. Electric fence

Electric fence [9] is a memory-debugging tool, which is implemented in the form of a library that our program needs to link to. It is capable of detecting overruns of memory allocated on a heap as well as memory accesses that have already been released.

After linking the code shown in Fig. 7 with an electric fence library, we get the output as shown in Fig. 8. It captures the segmentation fault using gdb.

```
void f(void) {
  int* x = (int *)malloc(10 * sizeof(int));
  //use the dynamically allocated memory
  x[11]=50;
}

int main(int argc, char* argv[]) {
  f();
  return 0;
}
```

Fig 7: Code in which invalid memory is being accessed. x[11]=50 will leadto segmentation fault

Fig 8: Segmentation fault is generated after linking code shown in Fig. 6with electric fence and running with gdb.

Electric fence tool actually adds extra memory (fence) before and after the allocated memory. If the program tries to access that memory (fence), segmentation fault is raised. It stops on the very first instruction where the segmentation fault is observed in code.

C. Mtrace

Mtrace [10] is a memory debugging tool included in the GNU C library. It helps to detect memory leaks. The source code needs to be modified to detect memory leak with mtrace. As shown in Fig 9, the code suffers from memoryleak issues as the allocated memory is not deallocated after use.

Fig 9: Code instrumented with mtrace() and muntrace().

The function mtrace installs handlers for malloc, realloc and free; the function muntrace disables these handlers. The output is written to a file mentioned by the environment variable MALLOC_TRACE. It can be clearly seen that the memory leak of 4 bytes after we check the trace file using mtrace as shown in Fig 10.

```
dilip@dilip-Vostro-3580:~$ gcc -g MtraceTest.c -o MtraceTest
dilip@dilip-Vostro-3580:~$ export MALLOC_TRACE=./mtrace.txt
dilip@dilip-Vostro-3580:~$ ./MtraceTest
dilip@dilip-Vostro-3580:~$ mtrace MtraceTest mtrace.txt

Memory not freed:
-----------------
           Address     Size     Caller
0x000055de698676a0      0x4  at 0x55de67d8b8f1
dilip@dilip-Vostro-3580:~$
```

Fig 10: Demo to show the memory leak issue in the code shown in Fig. 9

After freeing the memory pointed by p using free(p) as shown in Fig. 11, mtrace run on this code.

```
#include <stdio.h>
#include <stdlib.h>
#include <mcheck.h>

int main()
{
   int* p;

   //It is called before malloc(), calloc()
   mtrace();

   p = (int*) malloc(sizeof(int));
   *p=10;
   free(p);

   //Disable memory tracing before exit
   muntrace();

   return 0;
}
```

Fig 11: The memory pointed by p is freed by free(p).

Fig. 12 shows that no memory leak is detected after running mtrace on code shown in Fig. 11

```
dilip@dilip-Vostro-3580:~$ gcc -g MtraceTest.c -o MtraceTest
dilip@dilip-Vostro-3580:~$ export MALLOC_TRACE=./mtrace.txt
dilip@dilip-Vostro-3580:~$ ./MtraceTest
dilip@dilip-Vostro-3580:~$ mtrace MtraceTest mtrace.txt
No memory leaks.
dilip@dilip-Vostro-3580:~$
```

Fig 12: No memory leak is observed after running mtrace on the codeshown in Fig. 11

D. Deleaker

Deleaker[11] is a memory profiling tool, which supports languages like C++, C#, .Net and Delphi. This tool can either run as a standalone application or be integrated with many IDEs like Microsoft Visual Studio, Delphi, C++ Builder, and Qt Creator. We have integrated Deleaker with Visual Studio 2019 to perform our experiments. It finds leaks related to memory and handles. This tool can detect GDI (Graphics Device Interface) leaks which can hamper theperformance of applications in windows OS. Deleaker is supported on Windows only.

Fig 13 shows the code which has allocated memory on heap in line 6 and line 7. The code suffers from memory leak as the memory locations pointed by the pointers x and y are not deallocated.

```
#include<stdio.h>
#include<stdlib.h>
void f(void) {
  int* x = NULL;
  for(int i=0;i<5;i++) {
       x = (int*)malloc(10 * sizeof(int));
  }
  int* y = (int*)malloc(20 * sizeof(int));
  //use the dynamically allocated memory
}
int main(int argc, char* argv[]) {
  f();
  return 0;
}
```

Fig 13: Code having memory leak issues in line 6 and line 7

When the code is analyzed using Deleaker (used as aplugin inside Microsoft Visual Studio 2019), the memory leak problems were reported at line 6 and 7 as shown in Fig 13. The hit count shows the count of allocations made at the same place in application/code. As memory allocation has been done 5 times in for loop with the same pointer variable x, so the Fig 14 shows the hit count value as 5.

Fig 14: The inspection of code shows that there is memory leak observed after we inspect the code shown in Fig. 13 with visual studio using Deleaker Memory Leak detection tool.

```
#include<stdio.h>
#include<stdlib.h>
void f(void) {
  int* x = NULL;
  for(int i=0;i<5;i++) {
       x = (int*)malloc(10 * sizeof(int));
       free(x);
  }
  int* y = (int*)malloc(20 * sizeof(int));
  //use the dynamically allocated memory
    free(y);
}
int main(int argc, char* argv[]) {
  f();
  return 0;
}
```

If the memory pointed by pointer x and y is freed by free() as shown in Fig 15, then the memory leak problem is resolved as shown in Fig. 16. (No memory leak is observed)

Fig 15: Code having memory leak issues in line 6 and line 7

Fig 16: No memory leak is observed when Deleaker is runon Fig. 15

A. Comparison of Tools and Evaluation

A. *Comparasion*

Table 1 shows the comparison of various Memory leakage detection Tools based on different parameters.

Table 1: Comparison of Memory debugging tools based on various parameters

Tools	**OS supported**	**Languages Supported**	**Support for Multithreaded Application or Single Threaded Application**	**Need to modify Source code for error detection**	**Tells the exact Line Number bydefault**	**Thread Safe**	**Issues analyzed**
Valgrind	Linux, Solaris, Android	C, C++, Java, Perl, Python, assembly code, Fortran, Ada and manyothers.	Multithreaded Application	No	Yes	Yes	Memory leak, double free, uninitialised memory, read/write memory after malloc(), incorrect freeing of heap memory, mismatched use of malloc/new/new[] versus free/delete/delete[]
Electric fence	Linux	C, C++	Multithreaded Application	No	No (using GDB only)	No	overruns of memory allocated on a heap, issues with memory already freed
Mtrace	Linux	C, C++	Single Threaded Application	Yes	Yes	No	memory leaks caused by unbalanced calls to the *malloc()* and *free()* function
Deleaker	Windows	C++, C#, .Net, Delphi	Multithreaded Application	No	Yes	Yes	Memory leak, GDI leak, Handle leak, Leak caused by COM

Evaluation of Performance

The time taken by each tool is evaluated to report its analysis on the code shown in Fig 17.

```
#include <stdio.h>
#include <stdlib.h>
//#include <mcheck.h>
int main()
{
  int* p;
  for(int i=1;i<1000;i++)
  {
    for(int i=1;i<100;i++)
    {
      //mtrace();
      p = (int*) malloc(sizeof(int));
      //use the dynamically allocated memory
      free(p);
      //muntrace();
    }
  }
printf("Memory Issues are being Checked\n");
 return 0;
}
```

Fig 17: Code for Evaluation of time taken by different tools

The command time is used to evaluate the performance of Valgrind, Electric fence, Mtrace and Deleaker. For evaluating the performance of Mtrace, line 3, 11 and 15 should be uncommented in code shown in Fig 17.

IV. Units

Error-Correcting Code (ECC) memory is a type of computer memory that includes additional error detection and correction capabilities. It is commonly used in servers, workstations, and other critical systems where data integrity is of utmost importance.

ECC memory works by incorporating extra bits into each unit of memory, allowing it to detect and correct certain types of memory errors that might otherwise lead to system crashes or data corruption. When data is written to memory, these extra bits are used to store parity information, which can be used to identify and fix errors when the data is read back.

In brief, ECC memory provides a higher level of reliability compared to non-ECC memory, making it suitable for applications where data accuracy and system stability are crucial. It helps to ensure that the data stored in memory remains intact and free from errors, ultimately contributing to the overall robustness and dependability of the computing system.

V. Helpful Hints

Certainly! Here are some helpful hints for memory leak detection and prevention mechanisms:

1. Use tools like Valgrind, AddressSanitizer, and LeakSanitizer to detect memory leaks in your code during development and testing phases.

2. Implement proper memory allocation and deallocation practices in your code, such as using smart pointers in C++ or employing automatic garbage collection in languages like Java or C#.

3. Regularly perform code reviews and static code analysis to identify potential memory leaks early in the development process.

4. Consider using memory profiling tools to analyze memory usage patterns and identify potential areas for optimization.

5. Stay updated with the latest best practices and techniques for memory management in your programming language of choice.

By following these helpful hints, you can proactively address memory leaks and improve the overall stability and performance of your software. If you have any specific questions or need further assistance, feel free to ask!

VI. Some Common Mistakes

Certainly! Here are some common mistakes related to memory leak detection and prevention mechanisms:

1. **Failing to Release Allocated Memory**: One of the most common mistakes is forgetting to release memory that has been dynamically allocated using functions like `malloc` or `new`. This can lead to memory leaks over time.

2. **Lack of Error Checking**: Not checking for errors during memory allocation can result in memory leaks, especially when allocation fails and the program continues execution without properly handling the failure.

3. **Circular References**: In languages with automatic garbage collection, circular references between objects can prevent them from being properly deallocated, leading to memory leaks.

4. **Mismanagement of Resources**: Forgetting to release other resources, such as file handles or database connections, can also lead to resource leaks and indirectly contribute to memory leaks.

5. **Inadequate Testing**: Failing to thoroughly test the application for memory leaks, especially in long-running or resource-intensive scenarios, can result in undetected leaks.

By being aware of these common mistakes, developers can be more vigilant in their memory management practices and reduce the likelihood of memory leaks in their software. If you have any specific questions or need further assistance, feel free to ask!

VII. Conclusion

In conclusion, addressing memory leaks is crucial for maintaining the stability and performance of software systems. By utilizing effective memory leak detection tools and implementing best practices in memory management, developers can mitigate the risks associated with memory leaks.

It's important to prioritize proactive measures such as regular code reviews, testing with memory profiling tools, and staying informed about the latest advancements in memory management. Additionally, fostering a culture of awareness and accountability within development teams can contribute to early detection and resolution of memory leaks.

Continued research and development in this area are essential for refining existing detection mechanisms and exploring innovative approaches to memory leak prevention. By integrating these strategies into the software development lifecycle, organizations can minimize the impact of memory leaks and deliver more reliable and efficient software products.

If there are specific points you would like to add or discuss further in the conclusion, please feel free to let me know!

Acknowledgment

References

- Matthias Hauswirth and Trishul M. Chilimbi. 2004. Low-overhead memory leak detection using adaptive statistical profiling. SIGOPS Oper. Syst. Rev. 38, 5 (October 2004), 156-164. DOI=http://dx.doi.org/10.1145/1037949.1024412

- https://arxiv.org/pdf/2106.08938.pdf

- https://medium.com/@AlexanderObregon/java-memory-leaks-detection-and-prevention-25d1c09eaebe

- https://www.researchgate.net/publication/355759855_Memory_Leak_Detection_Tools_A_Comparative_Analysis/link/630518cfacd814437fcf429d/download?_tp=eyJjb250ZXh0Ijp7ImZpcnN0UGFnZSI6InB1YmxpY2F0aW9uIiwicGFnZSI6InB1YmxpY2F0aW9uIn19

- https://docs.openstack.org/self-healing-sig/latest/use-cases/memory-leak.html