**EMITTER International Journal of Engineering Technology**
ISSN: 2443-1168, Vol. 8, No. 2, December 2020, pp. 477~494
DOI : 10.24003/emitter.v8i2.550

# An Automated Approach of Detection of Memory Leaks for Remote Server Controllers

**Bhavana D[1], Veena M B[2], Santosh Kumar Sahu[3]**

[1]Student, Department of ECE, BMS College of Engineering, Bangalore, India
[2]Associate Professor, Department of ECE, BMS College of Engineering, Bangalore, India
[3]Dell R&D, Bengaluru, India
Email: [1]bhavana.d002@gmail.com, [2]veenamb.ece@bmsce.ac.in

*Received August 30, 2020; Revised November 8, 2020; Accepted December 8, 2020*

**Abstract**

Memory leaks are a major concern to the long running applications like servers which make the working set to grow with the program. This eventually leads to system crashing. This paper discusses a staged approach to detect leaks in firmware of remote server controller. Remote server controller monitors the server remotely with many processes running in the background. Any memory leak in the long running applications pose a threat to the performance of the system. The approach adopted here filters the processes running in the system with leaks based on time threshold in the first stage. These processes with leaks are passed to the next stage where precise memory leak detection is done using the open source dynamic instrumentation tool Valgrind. The system leverages an automated leak detection approach that invokes the leak detection process on encountering any severity in the system and generates a consolidated leak report. The proposed approach has less impact on the performance of the system and is faster compared to many available systems as there is no need to modify or re-compile the program. In addition, the automated approach offers an effective technique for detecting possible leakages in early software development phases.

**Keywords**: Memory leak, Remote Server Controller, Firmware, Valgrind tool, Memcheck

## 1. INTRODUCTION

Memory leak is a major cause of device instability and performance issues in software. Early design review of the program is a key factor in the design of electronic systems with growing design complexity. It helps in early design stage to overhaul or optimize the system. In the development stage of embedded systems, this holds bugs minimal. Particularly in embedded systems, as it causes the long running applications that will eventually run out of memory. A system running out of memory can lead to slowdown caused by frequent swapping in and out, and failure in process development due to no more available memory. In addition, memory leakages in certain large applications and services have had serious consequences. The identification of memory leaks is therefore very critical and necessary in order to ensure the software quality. This paper concentrates on memory leak detection of remote server controller firmware written in C language.

Memory leaks are allocated memory, which is no longer available to the program. This causes delay in execution of programs through increased paging, which may cause programs to run out of memory. Memory leaks are harder to track than unauthorized memory accesses. Memory leaks occur when a memory block has not been released, and therefore are omission errors rather than commission errors. These memory leaks are one of the causes of software aging [1].

Memory leaks significantly impact all computer software whether they are desktop applications, web services or service applications. The odd memory leak is often not of significant importance for many trivial applications or applications with very short lifetime of application and will go unnoticed. But for larger applications that use a lot of memory or have to run for a long time ( e.g. web servers) memory leaks a grave issue.

Memory leaks and access errors are simple to introduce into a code but difficult to remove. With the absence of memory access error detection facilities, it is dangerous for programmers to attempt to actively recover lost memory, because this may trigger freed-memory access errors with unpredictable consequences. In contrast, programmers can waste memory without feedback on memory leaks, by minimizing free calls. Memory leak reduces the system performance by decreasing the amount of available memory, increasing the amount of thrashing and eventually causing system failure or slow down. Memory leaks occurs because of the following reasons:

- unnecessary use of memory and improper allocation of memory.
- failure to release the allocated memory even after the termination of the program
- deletion of pointer to a memory block leading the block of memory no longer accessible
- programming languages like C and C++ without the support of garbage collection may cause memory related issues.
- when a function is returning a pointer to an allocated block of memory, but the calling routine ignores the returned value.
- occurs when a function containing a local variable which points to a block of memory, but failure to save the pointer in a global variable begore the function returns.

Remote sever controllers are basically baseboard management controllers (BMC) that are embedded in the device for remote monitoring. Memory leaks in the long running application for monitoring the servers remotely is a concern to the performance of the system. It does the job of deploying, tracking, handling, configuring, upgrading, troubleshooting, monitoring temperature, power and various other parameters and remedying servers from any location and without using agents. Any BMC has various daemons running in it for monitoring particular task. Each of these daemons may have leaks which on a long run are threat to the system as a whole.

Manual detection of leaks is time consuming as they are concealed without any imminent symptoms and hard to reproduce. Thus, an automated approach is required to detect leaks that are efficient and fast. This paper proposes an automated memory leak detection system through staged approach. The first stage detects the memory leaks in the daemons based on time threshold and the filtered daemons with leaks are passed onto the next stage for leak detection using open source tool Valgrind. Automated leak detection reduces the manual burden in industrial level testing. To improve the quality of the software, many programmers use program analysis tools, such as error checkers and profilers. One such class of tools is dynamic binary analysis (DBA) tools; they analyse programs at machine code level at run-time. DBA tools are often implemented with Dynamic Binary Instrumentation (DBI) which adds the analysis code to the original code. Valgrind is one such DBA tool. These DBA tools are heavyweight and slow down the analysis process. In order to reduce this impact, a staged approach is used here. In which the daemons with memory leaks in the first stage are shortlisted. These shortlisted candidates are passed to the second stage for leak detection using memcheck tool.

## 1.1 Motivating Example

In order to understand the behaviour of the system during memory leak, a leak is injected into a process running in the remote server controller and memory usage of that process is collected over time. It is depicted in the Figure 1. It is clear from the graph that the memory usage of the running process is increasing over the time and it does not free the memory blocks that it was allotted after its usage. It is evident that for long running applications it is a concern as it degrades the performance of the system gradually. There is a clear need for memory leak detection in long running applications and a system is designed here to tackle this problem.

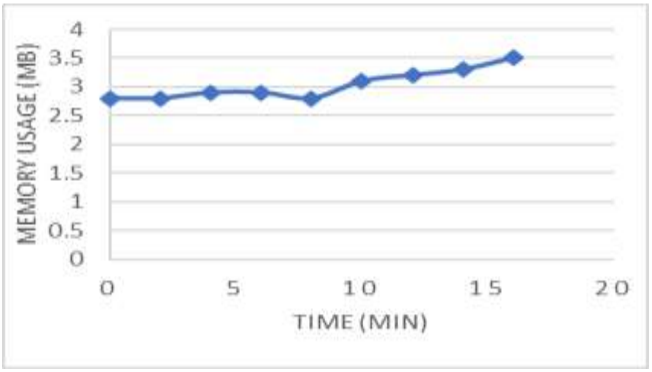


**Figure 1:** Memory Usage of a process with leak over time

## 2. RELATED WORKS

Various methods and tools are availale for memory leak detection. This can be classified into static and dynamic method of leak detection.

### 2.1 Static method of leak Detection

Static method of leak detetion happens without actual execution of the program. This method does not necessiate a test environment. Static approarch of leak detction mainly checks for semantics. It is a type of leak detection which involves static analysis of the source code. A few tools which belong to this category are Polyspace, CppChecker, and Flawfinder. Static techniques include analysis of reachability through a guarded value flow graph [2], dataflow analysis in a reverse manner or detection of constraint violation on ownership of the objects [3].The overhead of new test platform can be eliminated since there is no need to run the test programs in order to find the leaks. The overhead can be in terms of computational power usage, space consumption for storage. Static methods provide vast information regarding the defects as they traverse through every branch in the program.

### 2.2 Dynamic method of leak Detection

Dynamic memory allocation plays a vital role in C programming. Various hard to find bugs are caused mainly during dynamic programming. Freeing an allocated block of memory twice or overrunning the malloced block and failure in keeping track of addresses of allocated memory block are few of the common errors which irritates the programmer. These errors are quite difficult to debug because of the mysterious behavior that they manifest in them.

Dynamic detection method involves detection of memory leaks during run-time. Dynamic approaches [4] – [8] detect memory leaks through instrumentation and program execution. This method records of dynamic resource allocation and deallocation and then confirms if there are any memory leaks in the program running. Various tools available which fall under this category are Rational, Purify DDMEM and Memwatch. Programming languages such as Java have garbage collector in them for memory management. However, through this mechanism, not only the removal of memory leaks can be guaranteed but also this will lead to loss of system performance.

Despite the tremendous research progress in recent decades, the detection of memory leaks in industrial scale is still pretty much an unsolved problem. Most of the state of the art approaches face scalability issues [9] – [11]. Memory leak detection and optimization of the system plays a major role in the overall behavior of the system. Even though there has been a huge contribution in this field, industrial level scaling has not yet been achieved and various systems proposed require recompilation of the source code. In case of static memory leak detection, because of the path explosion problem and extensive use of the constraint solver, symbolic execution tools such as, CSA

[12] do not scale out to large programs. VFG-based model for memory leak detection is the cause for the failure of leak analysis scalability [13]. However, in case of dynamic approaches, the leak detection is achieved by instrumentation and program execution. Unlike the static approaches, dynamic leak detection approaches have relatively low false positive rate as they can access the program at runtime. Dynamic approaches indeed face the problem of overhead. In order to avoid high overhead, the number of candidates that are passed for leak detection through dynamic instrumentation can be limited by the staged approach as discussed in this paper.

## 3. ORIGINALITY

Memory leaks cause an application's working set to grow as the program runs. If the program runs long enough, memory leak increases and eventually leads to system crashing. In case of resource constrained embedded system, detecting memory leakage is a major concern as there is a need to operate with limited available memory. Here an industry scalable memory leak detection system with two staged approach is developed that address time constraints and performance hindrance in the existing approaches.

Memory leak detection in a Remote Server Controller for industrial level scaling is proposed in this paper through an automated approach. The leak detection system leverages two staged approach in order to minimize the impact of the memory leak detection process on the performance of the BMC or any long running embedded system for that matter. Time-threshold based leak detection is involved in the stage one of leak detection which has low impact on the target system. The candidates for leak detection are filtered through this stage and only those with possible leaks are passed to the second stage where a precise leak detection is done using open source tool Valgrind. This developed system is useful in software quality analysis which helps in overall product development life cycle.

## 4. SYSTEM DESIGN

Memory leaks in the Server Controller firmware is done through a staged approach. Any Baseboard Management Controller (BMC) may face critical scenarios in it like those that are related to hardware malfunction such as backplane or power supply issues. It has multiple daemons running on it for remote monitoring of the server. Leaks impact the performance of such systems. The system proposed uses time- theshold method and dynamic binary instrumentation tool for memory leak detection.

### 4.1 Dynamic binary Instrumentation

Dynamic binary analysis tools can be easily built using Dynamic Binary Instrumentation (DBI) framework [4]. DBA tools analyse the program at machine level code at run-time where the analysis code is attached to the main code at runtime. This is useful for the users because there is necessity of recompilation or relinking. It also offers the user-mode code a 100% instrumentation coverage, without demanding for source code.

Heavyweight DBA tools can be built by DBI framework like Valgrind. Valgrind tools are developed to the core of Valgrind as plug-ins, written in c. The basic view is: core Valgrind + plug-in tool = module Valgrind. Valgrind, like many other DBI systems, uses dynamic binary re-compilation. A Valgrind tool is invoked through a command by adding Valgrind – tool = (plus any Valgrind or tool options). The specified tool starts working, loading the software program into the same process one tiny block of code at a time, in a fashion guided just-in-time by execution. The core disassembles the program block into an intermediate representation (IR), instrumented by the device plug-in with analysis code, and then translated back into computer code by the core. The corresponding translation is preserved to be rerun in a code cache as required. Memcheck records all the calls into the functions of the dynamic memory allocator available in the malloc family. When launching the Memcheck device, there are many choices which can be transferred to the Valgrind command.

### 4.2 Remote Server Controller

Remote sever controllers are usually Arm-based System on chips (SoC) that are deployed in the server motherboard. The main function is remote access and monitoring of the server. This enables easy configuring, deployment of the sever without physical interference. They inform the administrator for any faults or malfunction in the system. Any remote sever controller is usually a BMC. It has various processes running in the background as daemons analysing each subsystem. The firmware is the software that actually tells the hardware what to do. The firmware is flashed on to the BMC. Any daemon with leak is a concern to the user as the available memory in the BMC is in terms MB. With such limited memory availability and constraints on the resources, leak detection becomes a must.

### 4.3 Overview

In case of production environment any changes in the build generated must go through various tests before release. This is a time consuming and a repetitive procedure. With thousands of lines of code for each daemon, finding the leaks related to memory becomes a tedious job for any programmer. The proposed system here concentrates on this issue. The leak detection process has been automated for ease of the programmer. The object file for the stage one of the leak detection and the Valgrind tool must be installed in the target system.

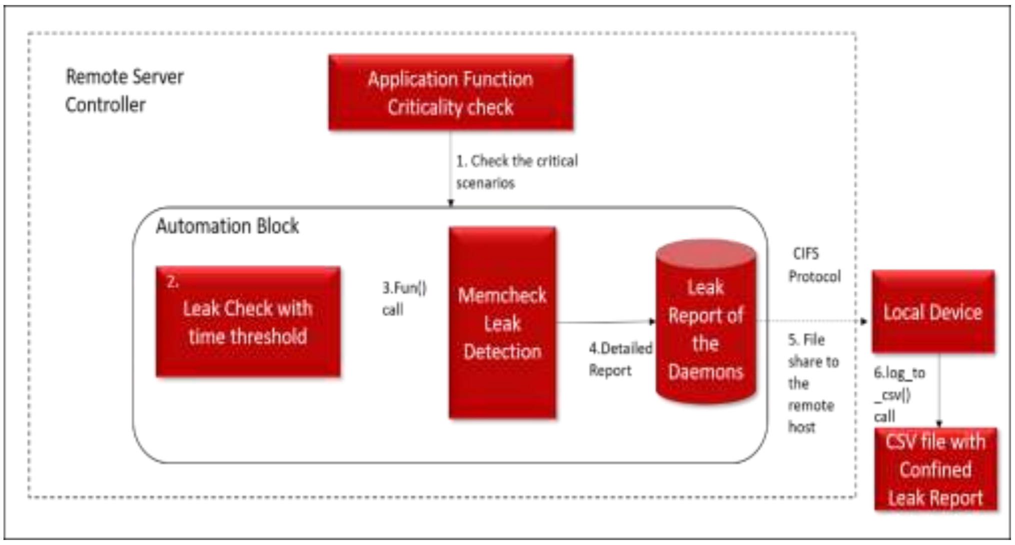

Remote Server Controller
Application Function Criticality check
1. Check the critical scenarios
Automation Block
2. Leak Check with time threshold
3.Fun() call
Memcheck Leak Detection
4.Detailed Report
Leak Report of the Daemons
CIFS Protocol
5. File share to the remote host
Local Device
6.log_to _csv() call
CSV file with Confined Leak Report

**Figure.2.** Block Diagram of Memory Leak Detection System for Remote Server Controller

The proposed system monitors the BMC for any criticality in the system using Remote Access controller administrator. If there is any criticality found, it triggers the automated memory leak detection block. The criticality in the system can be found by looking for the remote administrator log files. The time range between which the criticality in the system is prompted. Once the user inputs the required time range, search for criticality in the system is done. If there are any critical situations found, it immediately invokes the leak detection block. The automated memory leak detection system proposed in this paper is shown in Figure 2. The proposed design is implemented on a Linux machine with Remote Server Controller Firmware flashed in it with resource constraints.

The stage one of leak detection is based on the live time time of the memory alloction functions such as malloc and calloc. It is achieved by hooking memory functions through LD PRELOAD. The algorithm 1 shows the details of the process. Here if the alloc_count is equal to dealloc_count then program return 0 that is There is no requirement of altering or re-compiling the specified program, and the detection could be enabled or disabled during target run. To monitor references to allocated memory blocks one must first recognize where such blocks are positioned. Functions like malloc, calloc, realloc and their variants are considered for this purpose.

$number\ of\ bytes\ of\ memory\ leak = n_l$

$number\ of\ bytes\ allocated\ = n_a$

$number\ of\ bytes\ deallocated\ at\ = n_d$

$$\boxed{n_l = n_a - n_d} \tag{1}$$

$$expired_{count} = number\ of\ bytes\ of\ memory\ allocated\ at\ t_{sh}$$

$$free\_expired_{count} = number\ of\ bytes\ of\ memory\ deallocated\ at\ t_{sh}$$

$$number\ of\ blocks\ of\ memory\ leak = expired_{count} - free\_expired_{count} \quad (2)$$

$$number\ of\ bytes\ of\ memory\ leak = expired_{size} - free\_expired_{size} \quad (3)$$

The replacements call through to the initial functions in case of allocations and then record their return values. This recording takes place with the blocks having their start address hashed. While only the count of references to a block's start address is performed, this provides sufficient flexibility with respect to searching. The number of bytes still allocated and deallocated when the program exits at given time threshold ($t_{sh}$) is recorded. The basic way to detect memory leaks is through equation (1). This logic is applied in as shown in the equation (2) and equation (3) to find the blocks and bytes of memory leak respectively. If the $expired_{size}$ and $free\_expired_{size}$ is found to be equal, it is taken as there is no leak else any value greater than 0 is taken to be the bytes of memory that is leaking. If the memory function lives longer than the set threshold it takes the memory as leak. The time limit should be set as per the scenarios by the user. For instance, if users debug a keep-alive HTTP server and there are connections that last for over 5 minutes, users should set the threshold to 300 seconds to compensate it. hen any memory is supposed to be released in 2 second by the system, the threshold should be set to 3 seconds to obtain the report in time. The time threshold is set to 5 seconds, which is minimum value so that no leaks are missed. This can be changed as per the requirement of firmware. The report consists of full call stack at a dubious memory leak point and makes it easier to use compared to other similar libraries. It is easy to use as well as displays a complete call-stack at the spot of suspected memory leak. The flow diagram of this stage is shown in Figure 3. Following are the flags for the ease of the user:

--LEAK_EXPIRE=<threshold_time>
If a memory block is found to be not frees at the time threshold it is logged. This value is set according to the requirement as explained above.

--LEAK_AUTO_EXPIRE=<filename> [False]
If this flag is set to be true, the threshold time is automatically increased if the any block frees after the time expiry. This value is set to be false as default.

--LEAK_LOG_FILE=<filename>
Input the file location where the log files for leak details are to be future stored, this location is defined as tmp on the device by default. The flags corresponding syntax and input are shown in Table 1.

Table 1: Various flags under the stage one of memory leak detection

| Flags | Syntax | Input |
|---|---|---|
| `LEAK_EXPIRE` | `--LEAK_EXPIRE=<threshold_time>` | $t_{sh}$ |
| `LEAK_AUTO_EXPIRE` | `--LEAK_AUTO_EXPIRE=<true\|flase>` | `Default=false` |
| `LEAK_LOG_FILE` | `--LEAK_LOG_FILE=<filename>` | `Log file location` |

**Algorithm 1: Threshold based memory leak detection**

```
Input:
    Time-threshold ($t_{sh}$), leak report location
Output:
    Stage one leak report, filtered candidates with leak
1:  //initialise alloc_count and dealloc_count
2:  // record the bytes allocated and deallocated
3:  if (t< $t_{sh}$)
5:      alloc_count = record_alloc_count(item)
6:      dealloc_count = record_dealloc_count(item)
7:  end else
8:  else
9:      if alloc_count = dealloc_count
10:         //no leak
11:         discard(item)
12:     end if
13:     else
14:         //leak found
15:         save(item)
16:     end else
17: end else
18: report_leak()
```

The replacements call through to the initial functions in case of allocations and then record their return values. This recording takes place with the blocks having their start address hashed. While only the count of references to a block's start address is performed, this provides sufficient flexibility with respect to searching. The number of bytes still allocated and deallocated when the progam exits at given time threshold ($t_{sh}$) is recorded. As shown in the equation (1), if the value of $n_l$ is found to 0; the its taken as there is no leak else any value greater than 0 is taken to the byes of memory that is leaking. If the memory function lives longer than the set threshold it takes the memory as leak. The time limit should be set as per the scenarios by the user. It is easy to use as well as displays a complete call-stack at the spot of suspected memory leak. The flow diagram of this stage is shown in Figure 3.

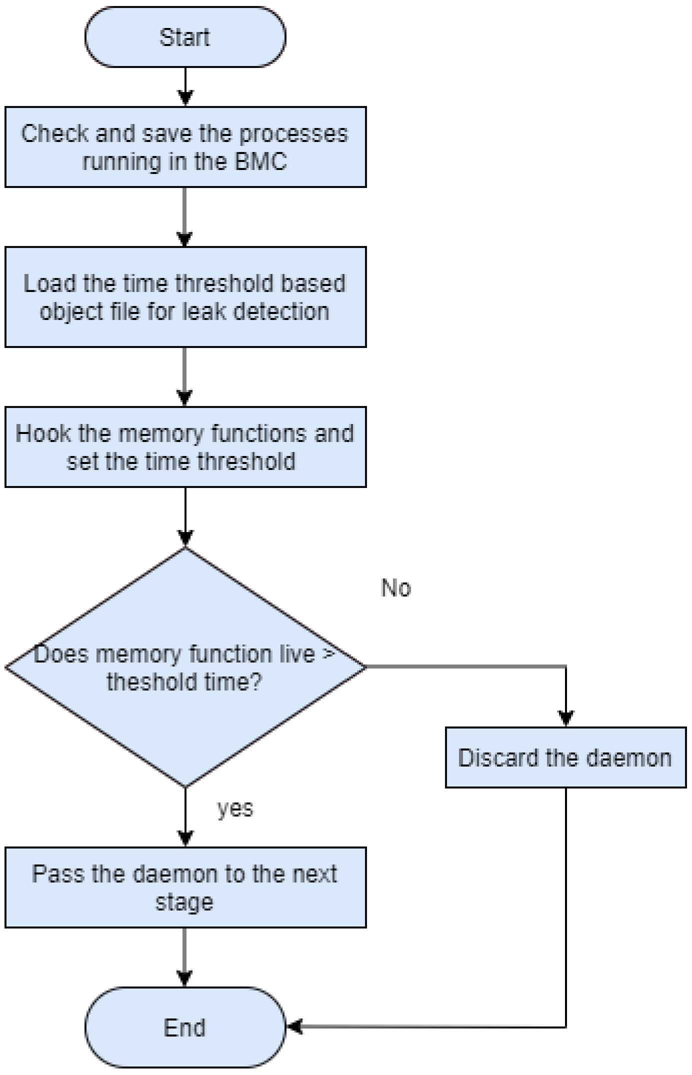

Start
Check and save the processes running in the BMC
Load the time threshold based object file for leak detection
Hook the memory functions and set the time threshold
Does memory function live > theshold time?
No
Discard the daemon
yes
Pass the daemon to the next stage
End

**Figure 3.** Flow Diagram of time-threshold based leak detection

During the second stage, the daemons found to have leaks in the stage one are passed onto this stage. Here the leak detection of the firmware is done using Memcheck tool. The Valgrind tool is deployed on the target machine. The filtered daemons from the stage one are checked for the memory leak and the leak summary is generated for each daemon. The memcheck tool adds its instrumentation code to the source code and returns back to the valgrind core. The valgrind core sends the source code to the selected tool, the tool adds the instrumentation code and sends back to the core. The valgrind core executes each process as a child process and logs the leak report. The flow diagram of automated approach is shown in Figure 4.

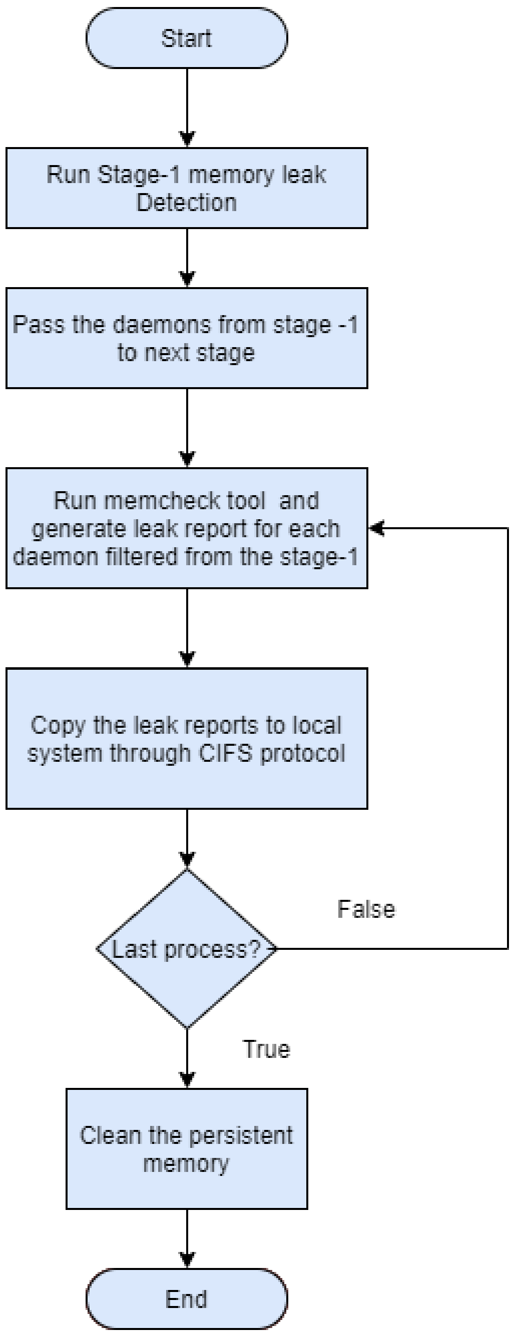

Start
Run Stage-1 memory leak Detection
Pass the daemons from stage -1 to next stage
Run memcheck tool and generate leak report for each daemon filtered from the stage-1
Copy the leak reports to local system through CIFS protocol
Last process?
False
True
Clean the persistent memory
End

**Figure 4**. Flow Diagram of the Automated Leak Detection Block

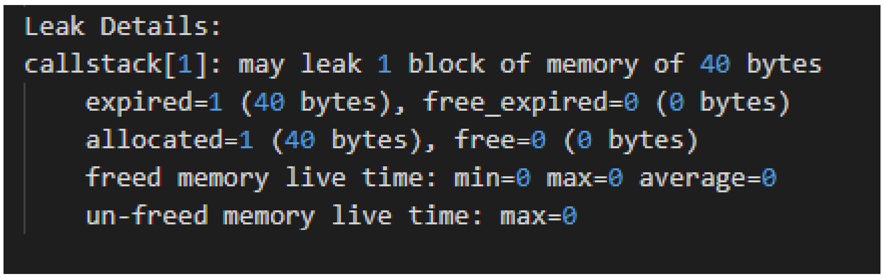
Since the target system has very little memory all the report generated are moved to the host system using the Common Internet File System (CIFS) protocol which is suitable for sharing files on a Local Area Network (LAN) as do not want to burden a resource constrained system. Here the log to csv file conversion is done in order to generate a confined report with only the required information in it there by saving the burden on the user to go through each daemon report line by line. All this process has been automated to ease the process of testing in industry level. Figure 8 depicts the flow of the leak detection process.

## 5. EXPERIMENT AND ANALYSIS

The proposed leak detection system was deployed in a Remote Server Controller environment for its performance evaluation. Since the target device to which the memory leak detection has to be done is BMC, the test object is BMC firmware. It is an Embedded Linux system kernel with limited memory in terms of few MB. Usually there are more than 50 daemons running in the system each with thousands of lines of code, this makes the memory leak analysis complex.

### 5.1 THRESHOLD BASED LEAK REPORT

The staged approach gave accurate result of memory leaks in the firmware. The output of the first stage with leak is shown in Figure 5 which considered a leak if the memory function lives longer than the given threshold time and this daemon was passed to the next stage for detail and precise analysis of the leaks.



```
Leak Details:
callstack[1]: may leak 1 block of memory of 40 bytes
    expired=1 (40 bytes), free_expired=0 (0 bytes)
    allocated=1 (40 bytes), free=0 (0 bytes)
    freed memory live time: min=0 max=0 average=0
    un-freed memory live time: max=0
```

**Figure 5.** Leak Details obtained from time threshold-based leak detection

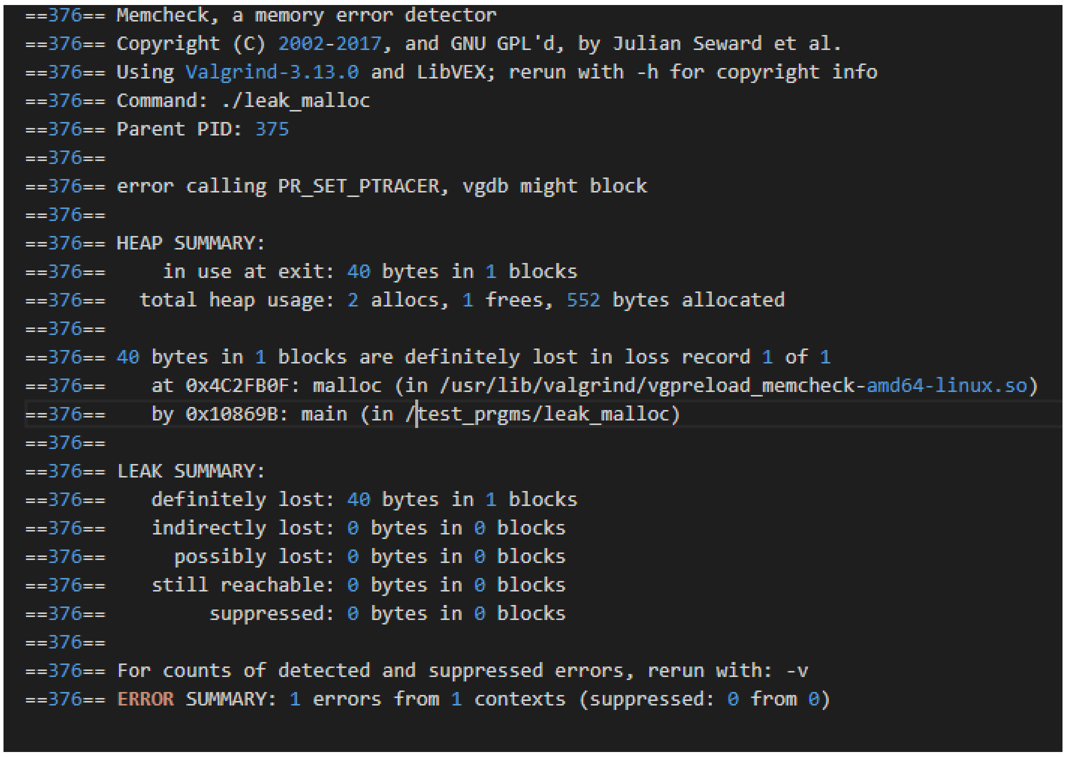
### 5.2 VALGRIND LEAK REPORT

Instead of checking the complete set of processes running in the system for memory leaks only the ones with suspicious leaks are passed to the second stage. Since Valgrind as comparatively higher overhead, the number of candidates passed to this stage is filtered in the first stage itself.

The leak summary obtained by Valgrind Memcheck tool is shown in Figure 6. The PID: 375 indicates the parent process ID, that is the valgrind memory leak detection process ID. This parent process invokes the child process that is *leak_malloc* which runs in the synthetic CPU of the valgrind core. Here the the memory related functions are monitored Here in the leak summary of Valgrind, each parameter is as follows:

"**Heap Summary**" shows you the number of bytes in use when the program ends, the number of allocations of memory (each time the new operator is used), the number of frees (every time the free/delete operator is used) and the overall number of bytes allocated.

"**Leak Summary**" shows the amount of memory leaked by the program. Anything lost means that the program can no longer reach a certain heap allocated memory.
"**Error Summary**" shows the total number of errors that occurred during the execution of the program.
"**definitely lost**" represents the program is leaking memory and they must be fixed
"**indirectly lost**" indicates that there is leak in the pointer-based structure of the program. (E.g. if "definitely lost" is the root node of a binary tree is, all the "indirectly lost" leaks will be the children.) By fixing the "definitely lost" leaks, the "indirectly lost" leaks will be fixed too.
"**possibly lost**" represents that there are memory leaks in the program as a result of unusual usage of the pointer.
"**still reachable**" shows memory leaks on account of failure to free the allocated memory. These types of leaks are quite common.



```
==376== Memcheck, a memory error detector
==376== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==376== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==376== Command: ./leak_malloc
==376== Parent PID: 375
==376==
==376== error calling PR_SET_PTRACER, vgdb might block
==376==
==376== HEAP SUMMARY:
==376==     in use at exit: 40 bytes in 1 blocks
==376==   total heap usage: 2 allocs, 1 frees, 552 bytes allocated
==376==
==376== 40 bytes in 1 blocks are definitely lost in loss record 1 of 1
==376==    at 0x4C2FB0F: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==376==    by 0x10869B: main (in /test_prgms/leak_malloc)
==376==
==376== LEAK SUMMARY:
==376==    definitely lost: 40 bytes in 1 blocks
==376==    indirectly lost: 0 bytes in 0 blocks
==376==      possibly lost: 0 bytes in 0 blocks
==376==    still reachable: 0 bytes in 0 blocks
==376==         suppressed: 0 bytes in 0 blocks
==376==
==376== For counts of detected and suppressed errors, rerun with: -v
==376== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

**Figure 6.** Leak Summary obtained by Valgrind with definite leaks

### 6.3 RESULT ANALYSIS

The number of candidates for memory leak detection through Valgrind was filtered in the first stage. There by the time consumed because of the addition of instrumentation code by the tool memcheck to the test code which slows down the time required for the source code execution was reduced. This also reduced the impact of heavyweight analysis tool on the host system. In order to analyze this, test was conducted with various background processes running in the Remote Controller. The time requirement for memory leak detection with and without automation, memory leak detection of all the running processes in the firmware was analysed. The manual leak detection involves obtaining leak reports for each process by passing valgrind command for normal approach. It was found through repeated manual execution of valgrind command that the time without automation was found to be 425 minutes in average. And for the staged approach, the daemons with suspected leaks in stage one are checked for leaks in the same way. It was found that the average time taken for obtaining leak report through Valgrind was found to be 5 minutes and would have to be killed forcefully else would take around 20 minutes for the generating the same report. So, in the automated approach, the code is designed to take care of this by killing the child process of valgrind which runs more than 5 minutes or go to the next process if it takes less than 5 minutes. This take limit can be varied depending on the behaviour of the firmware. Table 2 depicts the total number of candidates for memory test, candidates in stage two, normal approach which the direct use of stage two without filtration.

**Table 2.** Number of leak check candidates with and without staged approach

| Approach | No of candidates in stage 2 | Total test candidates | Time w/o automation(min) | Time with automation(min) |
|---|---|---|---|---|
| Staged Approach | 48 | 75 | 425 | 90 |
| Normal Approach | 75 | 75 | 480 | 150 |

These test programs monitor each service for the controller, they may or may not have leak in them. These candidates have thousands of lines of code which makes them complex for analysis. It was obtained from the tests conducted that the number of candidates for leak test that were passed to the second stage from the stage one of threshold-based leak detection is 48. Only 64% of the initial test candidates were passed to the final phase of leak detection. The figures given in the Table 3 clearly showed that the staged approach reduced the overall effect on the system because of heavyweight framework. This was achieved by filtering out the leak detection candidates in the first stage. In order to analyze the accuracy of the staged approach, various test programs with no leak and few with injected memory leaks of different variations were tested.

The accuracy of the stage one of memory leak was evaluated. In order to evaluate this following parameter were considered.

**TW** – Total number of test candidates

**TP** – Number of True Positives, there is memory leak and it is reported correctly.

**TN** – Number of True Negatives, there is no leak and it is reported correctly.

**FN** – Number of False negatives, there is leak but reported wrongly as no leak
**FP** - Number of False positives, there is no leak but reported wrongly as leak

**Table 3.** Results of time threshold-based leak detection

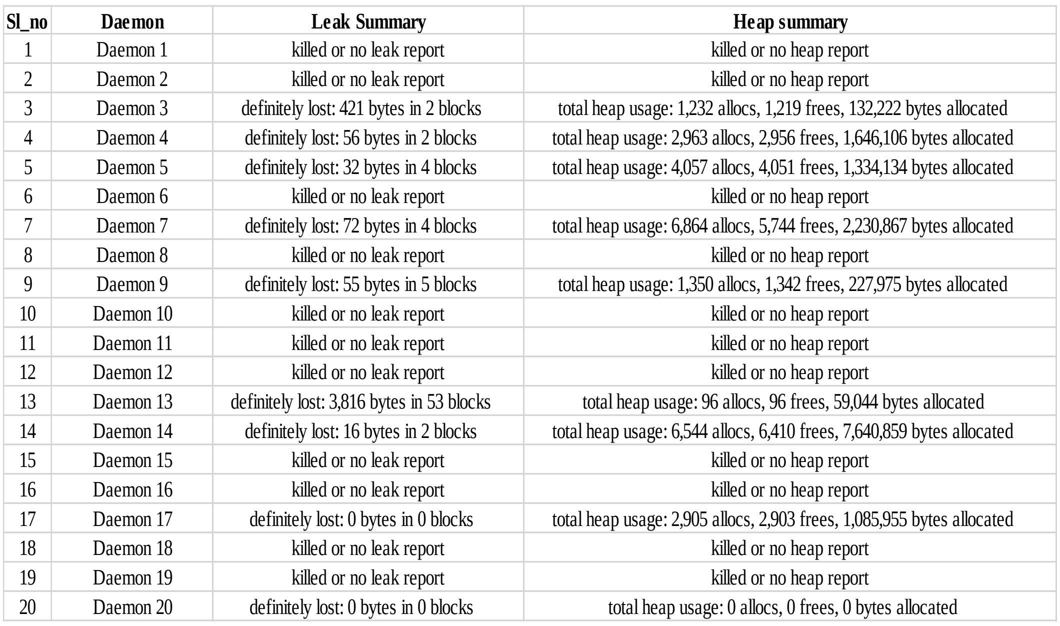
| TW | TP | TN | FN | FP |
|---|---|---|---|---|
| 59 | 28 | 20 | 4 | 7 |

The values obtained in test results are replaced in equation (4) in order to find the false positive rate.

$$FPR = \frac{FP}{FP + TN} \tag{4}$$

The total negatives (FP + TN) that is the number of candidates with no leak. This analysis showed that the false positive rate is 16.6% which is an acceptable value. This approach has lower false positive rate compared to the static leak detection approaches with better accuracy. And the complete details of the leak such as the position of the leak occurrence is obtained in the second stage. Valgrind does not affect the performance of the system as it runs the target program in its virtual CPU environment without any need for recompilation or altering the program.

The memory leak reports obtained were moved to the target system through CIFS protocol by prompting the user for IP address and the password of the system where the leak reports were to be moved. The leak reports range from 50 line to thousands of lines for each daemon, it is time consuming to go through each report. In order to ease the memory leak log files analysis, the log to csv file conversion is done. The target system is a baseboard management controller firmware which has various daemons for internal functionalities which have been named as daemon1-20. The analysis is done for the complete BMC firmware with all the running daemons with each source code with more than thousands of lines. The result obtained is shown in Figure 7. has a few of the daemons for which analysis was done.



| Sl_no | Daemon | Leak Summary | Heap summary |
|---|---|---|---|
| 1 | Daemon 1 | killed or no leak report | killed or no heap report |
| 2 | Daemon 2 | killed or no leak report | killed or no heap report |
| 3 | Daemon 3 | definitely lost: 421 bytes in 2 blocks | total heap usage: 1,232 allocs, 1,219 frees, 132,222 bytes allocated |
| 4 | Daemon 4 | definitely lost: 56 bytes in 2 blocks | total heap usage: 2,963 allocs, 2,956 frees, 1,646,106 bytes allocated |
| 5 | Daemon 5 | definitely lost: 32 bytes in 4 blocks | total heap usage: 4,057 allocs, 4,051 frees, 1,334,134 bytes allocated |
| 6 | Daemon 6 | killed or no leak report | killed or no heap report |
| 7 | Daemon 7 | definitely lost: 72 bytes in 4 blocks | total heap usage: 6,864 allocs, 5,744 frees, 2,230,867 bytes allocated |
| 8 | Daemon 8 | killed or no leak report | killed or no heap report |
| 9 | Daemon 9 | definitely lost: 55 bytes in 5 blocks | total heap usage: 1,350 allocs, 1,342 frees, 227,975 bytes allocated |
| 10 | Daemon 10 | killed or no leak report | killed or no heap report |
| 11 | Daemon 11 | killed or no leak report | killed or no heap report |
| 12 | Daemon 12 | killed or no leak report | killed or no heap report |
| 13 | Daemon 13 | definitely lost: 3,816 bytes in 53 blocks | total heap usage: 96 allocs, 96 frees, 59,044 bytes allocated |
| 14 | Daemon 14 | definitely lost: 16 bytes in 2 blocks | total heap usage: 6,544 allocs, 6,410 frees, 7,640,859 bytes allocated |
| 15 | Daemon 15 | killed or no leak report | killed or no heap report |
| 16 | Daemon 16 | killed or no leak report | killed or no heap report |
| 17 | Daemon 17 | definitely lost: 0 bytes in 0 blocks | total heap usage: 2,905 allocs, 2,903 frees, 1,085,955 bytes allocated |
| 18 | Daemon 18 | killed or no leak report | killed or no heap report |
| 19 | Daemon 19 | killed or no leak report | killed or no heap report |
| 20 | Daemon 20 | definitely lost: 0 bytes in 0 blocks | total heap usage: 0 allocs, 0 frees, 0 bytes allocated |

**Figure 7.** Confined report of memory leak

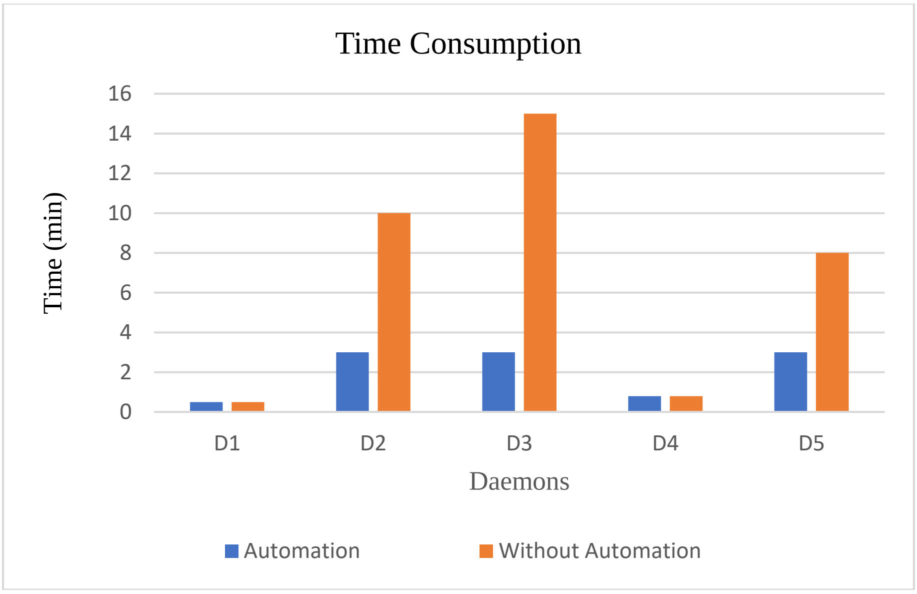


**Figure 8.** Time consumption for memory leak report generation with and without automation

The automated approach required less than 50% of the time required for manual analysis. The graph shown in Figure 8 depicts the time consumption for memory leak detection with and without automation. The parameters on the x-axis represent different daemons D1, D2, D3, D4 and the y-axis represents the time required to generate leak report for respective daemon. It also took care of termination of the child process initiated by the Valgrind which runs for indeterminate amount of time for few daemons. There were certain daemons which run without termination. Therefore, in order to avoid this problem, the leak detection is designed to kill the daemon after the given time range in stage-2. This proposed system of automated leak detection helps in uninterrupted software testing in the industry. However, the limitation of the work is that the instrumentation overhead is high in stage 2. There is scope for alternate approach in stage 2 of leak detection.

## 6. CONCLUSION

Detection of memory leaks is of significant importance for software quality analysis and testing. However it is a tedious task detect leaks with millions of lines of code at an industrial level. The impact of memory leaks on the free available memory of the controller is studied here. It is found that the long running processes with leaks led to decrease in the memory availabily in the controller. This paper presents an automated approach of two staged detection of memory leaks in the firmware for Remote Server Controllers. The first stage of leak detection is lightweight and filters the daemons with memory leak based on time threshold analysis. The second stage involves leak detection with the open source tool, valgrind memcheck. The number of candidates in the second stage of dynamic leak detection is reduced through time threshold-based leak analysis. It is evident from the test analysis done that the time requirement for leak detection through automation is less than 50% of the time required through manual analysis. False positive rate of the stage one found to be 16.6 % which is better compared to various available approaches and there is less impact on the overall performance of the system as Valgrind tool runs the leak detection analysis in its synthetic CPU without disturbing the services running in the remote controller.

The future work is to lower the instrumentation overhead of memcheck tool in the stage 2. Various other tools such as cachegrind, callgrind and massif are available under Valgrind, these tools can be utilized for memory profiling and optimization of leak detection process. The proposed memory leak detection system targets Baseboard Controller firmware, this can be extended to other real time embedded systems with memory constraints.

There is no recompilation or program alterations needed in both the stages of leak detection. This approach of leak detection is suitable for any embedded system with resource constraint and can be scalable for industrial applications. The false positive rate achieved is acceptable. The detection of leaks in the software at an early stage helps in improving the software quality there by leading to better throughput.

**Acknowledgements**
The authors would like to thank Dell Research and Development, and BMS College of Engineering Bangalore for providing the opportunity to carry out the work.

## REFERENCES

[1] F. Machida, N. Miyoshi, **Analysis of an optimal stopping problem for software rejuvenation in a deteriorating job processing system,** Reliability Engineering & System Safety, Vol. 168, pp. 128 – 135, 2017.

[2] S. Cherem, L. Princehouse, and R. Rugina, **Practical Memory Leak Detection Using Guarded Value-flow Analysis**, ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 480–491, 2007.

[3] D. L. Heine and M. S. Lam, **Static Detection of Leaks in Polymorphic Containers**, International Conference on Software Engineering (ICSE), pp. 252–261, 2006.

[4] Nicholas Nethercote, Julian Seward, **Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation**, Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation, USA, 2007, pp. 89-100, 2007.

[5] James Clause, Alessandro Orso, **Leakpoint: pinpointing the causes of memory leaks**, Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering, Vol. 1, pp. 515–524, 2010.

[6] Matthias Hauswirth, Trishul M. Chilimbi, **Low-overhead memory leak detection using adaptive statistical profiling**, ACM SIGOPS Operating Systems Review,Vol. 38, No. 5, 2004.

[7] Konstantin Serebryany and Derek Bruening, **AddressSanitizer: a fast address sanity checker**, Proceedings of the USENIX conference on Annual Technical Conference, pp.28, 2012.

[8] Changhee Jung, Sangho Lee, Easwaran Raman, Santosh S Pande, **Automated memory leak detection for production use**, Proceedings of the 36th International Conference on Software Engineering, pp. 825–836, 2014.

[9] R. Beneder, B. Glatz, M. Horauer and T. Rauscher, **Memory leak detection runtime-service for embedded Linux devices**, Proceedings of the 2014 IEEE Emerging Technology and Factory Automation (ETFA), Barcelona, pp. 1-6, 2014.

[10] Y. Sui, D. Ye and J. Xue, **Detecting Memory Leaks Statically with Full-Sparse Value-Flow Analysis**, in IEEE Transactions on Software Engineering, vol. 40, no. 2, pp. 107-122, Feb. 2014.

[11] Xiaohui Sun, Sihan Xu, Chenkai Guo, Jing Xu, et al. **A Projection-based Approach for Memory Leak Detection**, 42nd IEEE International Conference on Computer Software & Applications, IEEE, Vol. 2, pp. 430-435, 2018.

[12] The LLVM Foundation, **Clang static analyzer**, 2018.

[13] G. Fan, R. Wu, Q. Shi, X. Xiao, J. Zhou and C. Zhang, **SMOKE: Scalable Path-Sensitive Memory Leak Detection for Millions of Lines of Code**, 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), Montreal, QC, Canada, pp. 72-82, 2019.