Vasily Davydov

Detecting Memory Leaks in Long-Running Applications

Metropolia University of Applied Sciences

Bachelor of Engineering

Information Technology

Bachelor's Thesis

27 August 2023

Abstract

Author:	Vasily Davydov
Title:	Detecting Memory Leaks in Long-Running Applications
Number of Pages:	46 pages
Date:	27 August 2023
Degree:	Bachelor of Engineering
Degree Programme:	Information Technology
Professional Major:	Smart IoT Systems
Supervisors:	Keijo Länsikunnas, Senior Lecturer Guzman Alvarez, Master Software Developer Laszlo Pinter, Senior Software Developer

Efficient memory management is critical in preserving the reliability and stability of evolving software systems. Mismanaged memory may lead to leaks and subsequent system failure due to memory exhaustion. To address this problem, it is essential to have a tool for memory leak monitoring. While the industry has an array of different memory leak detection programs, most of them work only with finite applications and do not produce real-time data. Due to this, the challenges persist when debugging long-running applications, necessitating the development of a mechanism capable of real-time memory leak detection.

The main objective of this project was to develop a memory leak detection framework able to produce real-time output, while monitoring a long-running application. The library is written in partnership with the telecommunication service provider Ericsson to introduce a lightweight and efficient approach for memory debugging in long-running systems.

Applying an iterative development methodology helped to achieve significant improvements in the performance of the library during its implementation, by employing several data structures and algorithms to find the most suitable solution. A test run in Ericsson's development environment proved the library to be useful in several test scenarios. It also assisted with the ongoing maintenance issue related to an increase of memory in one of the applications.

In the end, the dynamic memory debugging framework proved its significance and potential by providing a possibility to report debugging information promptly, which is essential for systems with a long lifespan.

Keywords: real-time memory leak detection, long running application, dynamic linking

---

The originality of this thesis has been checked using Turnitin Originality Check service.

Contents

List of Abbreviations

OOM:	Out of memory
OS:	Operating System
LIFO:	Last In, First Out
CPU:	Central Processing Unit
VAS:	Virtual Address Space
MMU:	Memory Management Unit
API:	Application Programming Interface
LRA:	Long-Running Application
AVL:	Adelson-Velsky and Landis
PID:	Process ID
CI:	Continuous Integration

1 Introduction

In the modern world of technology, software plays a pivotal role as the foundation for evolving innovations. Ensuring stability and efficiency of software is key to driving progress forward. One approach to establishing the reliability of software is through memory management mechanisms; however, as complexity grows, so does the potential for undesirable behaviour of memory in the applications. Frequently, such exceptions may result in an out of memory (OOM) state, often caused by memory leaks. To address these issues, numerous techniques and tools are used in the industry. Nonetheless, depending on the background of corrupted application, these tools may produce an unwanted result.

A significant piece of complex software is encompassed with long-running applications that operate for continued periods, potentially running uninterrupted for months. Detecting origins of memory leaks in such applications can sometimes be challenging due to their nature. The majority of prominent memory tracking instruments typically provide output after the process has ended, whereas long-running systems expect real-time monitoring. The proposal to research and develop a solution for live memory leak detection was given to Ericsson, a global telecommunication company. As a result, Ericsson approved the initiative of this study, the central objective of which was to develop a lightweight and portable memory leak detection framework with the ability to produce real-time output. This was achieved by hooking onto allocator functions, storing data related to the allocated block and analysing it.

To fulfil the demands of Ericsson’s development environment, the core features of the framework were primarily designed with long-running applications in mind. Consequently, customisation of runtime parameters, such as timeout settings for leak detection and the starting point for detection, is essential before utilising the framework. These customisable parameters play a crucial role in enabling the framework's application across a broad spectrum of executable

programs, including C++ binaries. To ensure responsive efficiency, the framework was developed in the C Programming Language, utilising exclusively the GNU C libraries. In the testing phase, various tools will come into use, including the Robot Framework for conducting system tests and the Bazel build system for both building and executing unit tests.

2 Memory Management and Memory Leaks

2.1 Process Memory Layout

Every process is an instance of an application, which is currently being executed. Several users have the capability to execute the same program which upon success will result in many instances of the first. [1 p. 64.] Accordingly, each of these processes is wrapped with its own environment and has a unique ID. The execution of the program starts from the kernel loading the executable to prepare the start-up routine of the process. During the preparation, several memory segments are loaded into memory before accessing the main function, alternatively, the starting point of the program. Each memory segment serves its own purpose during the running process. [2 p. 301-313.] Figure 1 illustrates the essential arrangement of the segments.

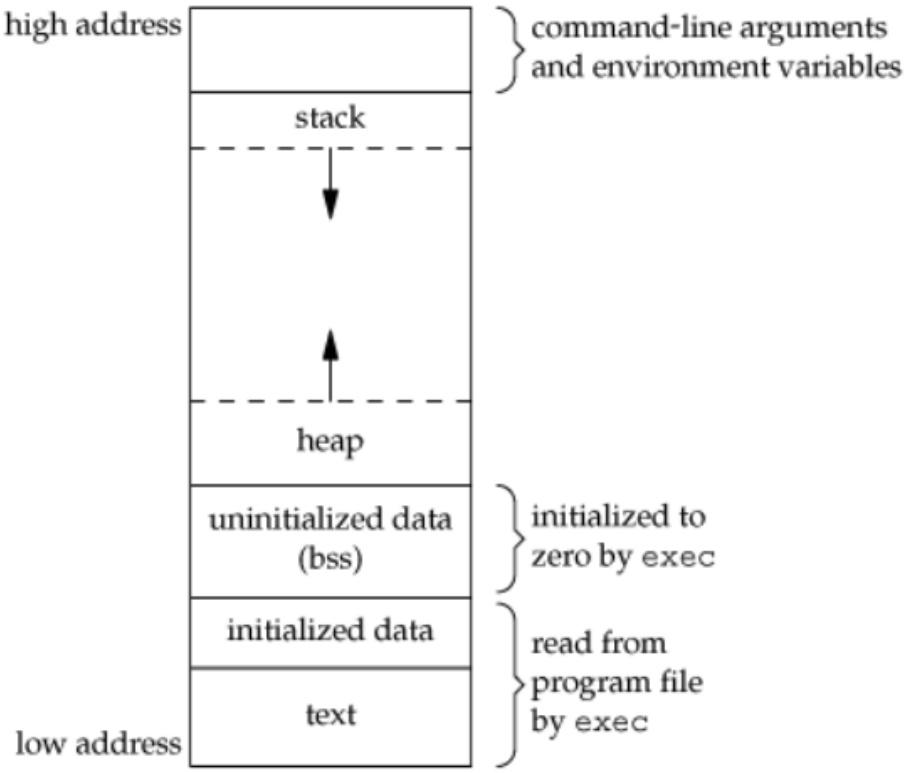

Figure 1. Typical Memory Arrangement of the Process [2 p. 314]

Starting from the highest address, the first memory segment is an Operating System (OS) kernel space accessed directly by system calls. This segment is not included in Figure 1, because it upholds strict memory isolation from user space to prevent unauthorised access and OS malfunction. [4, 6.] Kernel memory isolation is crucial in light of defects and vulnerabilities such as Meltdown, which was recently discovered as an OS security flaw. During the Meltdown attack, the whole kernel memory segment can be accessed from user space, prevailing isolation between them. Nevertheless, the Meltdown attack was defeated by the KAISER mechanism, mapping the kernel outside of user space. [14 p. 1-14.]

Below the kernel space, the top segment in Figure 1 is a memory segment reserved for process-related command-line arguments and environment variables. The current segment is accessible throughout the process from a scope of the main function by using the pointer variable **`extern char **environ`** for environment variables and the main function arguments **`int argc, char *argv[]`** for command-line arguments. [2 p. 311-313, 3]

Continuing to discuss memory layout, the next segment is stack. Essentially, the stack is a data structure that possesses the capability to add elements onto its top and remove them from the top, following the precise *Last In, First Out (LIFO)* terminology. [5 p. 4] The stack grows down towards the heap memory segment. Hence, in Figure 1, the top of the stack is located below its starting address. The primary purpose of the stack segment is to store runtime frames, including function calls and all associated data within their scope, such as function parameters and local variables. On each function call, a new stack frame is allocated regardless of a previous frame, enabling the use of recursive call technique. [2 p. 313, 4] There are two limiting factors for stack:

1. Stack pointer, which grows towards a lower address, meets the next memory segment, located below it. (Possible only on a system without page based virtual memory, which is described in more detail in Section 2.2.1) [1 p. 192, 4]

2. Stack pointer reaches the `RLIMIT_STACK` resource limit in bytes, which can be queried and modified using `getrlimit` and `setrlimit` system calls during runtime. [7]

Both situations will result in a termination of a process and its environment. [2 p. 303, 4, 7]

While the stack grows towards lower addresses, the Heap memory segment, located below the stack, expands from lower to higher addresses. Further details regarding the Heap segment and its role in storing a process's dynamically allocated memory addresses will be discussed in Section 2.2. [1 p. 220]

The next two segments both hold the information regarding data, hence called the data segment. From higher address, the first of the two sections is an uninitialized data segment, previously *bss* (Block Started by Symbol). [4] Before a program's execution, uninitialized data is mapped to this segment and is automatically assigned the value of arithmetic zero (or NULL) by the kernel. The initialised data segment serves a similar purpose, yet the data has a value initialised inside of the application. [2 p. 313]

The text memory segment represents the read-only space for central processing unit (CPU) instructions, typically loaded once for the efficient use of frequently utilised applications such as shells and compilers. [4]

To bring it all together, each process is enveloped in its own environment, consisting of public and private components to a process referred to as memory segments. Among these, two resizable segments are the Stack and Heap, both under the control of the process during runtime, yet separated from each other. The stack is expanded either statically or by the kernel itself, and thus intervening the process of its modifications with the functionality mentioned above is not preferred. [8] To get a better understanding of how the Heap

segment is resized, it is essential to grasp the fundamentals of Dynamic Memory Allocation.

2.2 Dynamic Memory Allocation

Most modern programs, despite primitive ones, require memory allocation during runtime [8 p. 1]. Memory allocation is accomplished by calling specialised allocator functions, such as `malloc` and `free` [2 p. 317] which will be explored in more substantial detail in Section 2.2.2. At its core, memory allocators supply memory regions from the heap at the request of the application to obtain them. When the application has no need for allocated regions, their ownership can be transferred back. [9, 10 p. 2] Previously, section 2.1 discussed the memory layout of the C application, which is divided into memory segments. The segmented model was introduced before the UNIX Operating Systems implemented the approach of organising memory virtually. Nevertheless, the modern approach of Virtual Memory does not conflict with segmented models; instead, it expands upon them. [8 p. 2]

2.2.1 Virtual Memory

A combination of both hardware and software is an approach virtual memory uses in its implementation. [6] By adding an additional layer of abstraction in memory management, the operating system is able to simulate vast storage space, which can be utilised by the process or processes. [11] To achieve the abstraction of physical addresses used to access the cells contained on memory chips, and to transform them into adjoining virtual address space (VAS), two main principles are used: address translation from virtual to physical addresses and VAS management. Address translation is put into practice by a hardware module called Memory Management Unit (MMU) located within the CPU, while the responsibility of managing the VAS falls on the OS. [1 p. 42, 6, 8 p. 2] The kernel, a core component of the OS, can arrange individual address space for each process, or create an address space shared with many processes at once. [6, 11 p. 3-4] This arrangement is done through mapping

physical addresses to virtual pages, which represent blocks of consecutive virtual memory addresses. Processes may use a list of virtual pages, which are connected to each other within a page-table. [6, 8 p. 2] Linux uses a three-level paging model, where each page-table's entry points to a page frame, directing the address to its corresponding location in physical memory. [1 p. 58] The model is presented on Figure 2.

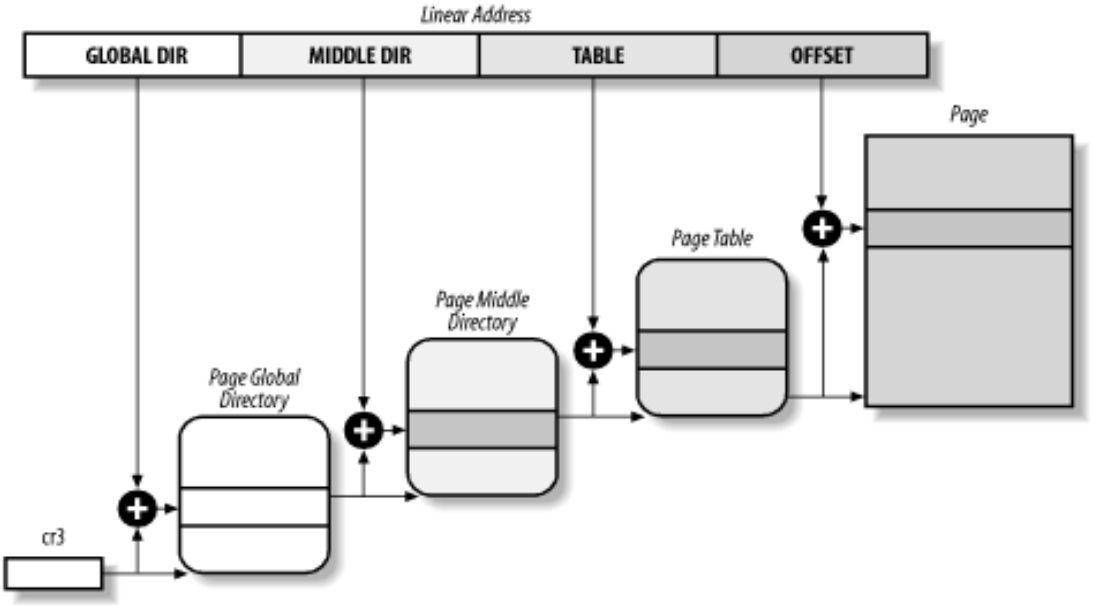

Figure 2. Linux Page Tables [1 p. 59]

When a program or the CPU generates a virtual address, it is divided into segments, which are stored in separate page-tables. Essentially, segmented virtual addresses are called linear addresses. [1 p. 50] Linear address ranges are represented as memory regions. Within the process, regions are structured as a linked list, collectively forming a VAS utilised by the process. One of the memory regions of the UNIX process is heap, which is used to supply dynamically allocated memory for the process, as mentioned in Section 2.1. [1 p. 189-220]

2.2.2 Heap Management

To manage dynamic memory directly, the kernel has a system call *brk(2)*, which is used to move the end of the data segment, alternatively, start of the heap data segment, resulting in memory allocation or deallocation for the process. [12] To move the end of the program's data segment, the kernel identifies available pages, possibly swapping out certain pages to free up memory. Hereafter, it inserts these pages into the process's page table. [1 p. 220-221, 8 p.2] A minority of executable programs use a *brk(2)* system call directly. This is primarily due to its inconvenient memory allocation behaviour, which allows the release of allocated blocks only in a LIFO order. Hence, the majority of executable programs opt for using *malloc(3)* library functions, which are implemented within the standard C library and utilise *brk(2)* internally. [1 p. 221, 9, 8 p. 2, 12]

To manage dynamic memory allocation within the program, the current standard C library ISO/IEC 9899:2018 provides the following *malloc(3)* family functions: [2 p. 317, 9, 13]

- void *malloc(size_t size);
- void *calloc(size_t nmemb, size_t size);
- void *realloc(void *ptr, size_t size);
- void free(void *ptr);

Essentially, each function, besides free(3), tries to allocate memory and returns a pointer to the allocated block, whereas free(3) accepts a block of memory to be released from allocation. Although there may be parametric differences for block allocation between these functions, the core principle is the same. [8, 9]

The implementation of *malloc(3)* monitors heap sectors internally allocated by *brk(2)*, acting as a server for a process, which requests memory from the system and hence gets a response. [1 p. 220-221] Keeping track of memory is achieved by dividing the heap segment into small pieces, which are

interconnected, forming a doubly-linked list of blocks, flagged as either ‘used’ or ‘free’. Depending on the circumstances, blocks may be divided into smaller pieces or merged together forming one big entity. The same technique is used for the *free(3)* function to release the memory by marking the block as ‘free’. The only two exceptions are when the pointer provided to *free(3)* is NULL, which indicates releasing the block is not possible and when the provided address has already been freed, which may lead to undefined behaviour. [8 p. 2, 9, 15 p. 26] When malloc(3) is called with a parameter of size, it iterates through the list to find a suitable memory block to be returned to a main program upon success or returns NULL in case of failure. [16 p. 149]

There are two other variations of *malloc(3),* which wrap the primary functionality of malloc with their unique features: *calloc(3)* and *realloc(3)*. To make a call to *calloc(3)*, one needs to provide the number of elements and the size of one element, hence resulting in the size of elements multiplied by the size of one element. The elements of the array allocated by *calloc(3)* are overwritten with a value of zero in memory, before returning the block to the caller program. Another alternative for memory allocation is *realloc(3)*, which is capable of resizing the current memory block bi-directionally, as well as allocating a new memory block. The function accepts a pointer to an old memory block and a new size of the block as its calling parameters. With this in mind, there are three scenarios for its behaviour depending on the parameters provided:

1. Passing the *NULL* as a pointer to an old memory block results in a subsequent call of *malloc(3)* with the size parameter provided.

2. The parameter size with the value of zero will result in releasing the pointer provided, if it is not NULL. It is noteworthy that *malloc(3)* and *calloc(3)* differ in their behaviour from *realloc(3)*. When a size with a value of zero is supplied, both return a unique pointer, which can be passed afterwards to the *free(3)* function to release it safely.

3. If both, the pointer to an old block and new size have non-zero values, the *realloc(3)* will perform resizing of the old memory block. In the case when a memory block needs expansion, but there is no space to grow in front, a new memory block is being searched to fit the contents of reallocation. As soon as the new block is found, old data is copied to it and the original block is released automatically. Hence, the new address is returned with old values remaining. [2 p. 318, 8 p. 2, 9]

Void pointers returned by functions within the *malloc(3)* family are aligned to multiples of 8 bytes, which ensures compatibility with different data types. Furthermore, to avoid writing out of boundaries of allocated blocks and hence corrupting the heap list, allocator functions usually give marginally more memory space than needed. [2 p. 317-318] It is evident that certain potential error cases are already handled by dynamic memory allocation functions, such as memory block overflow and the allocation of memory with a size of zero. However, dynamic memory allocation is a low-level Application Programming Interface (API) that interacts with the VAS of a process, carrying the potential for both power and risks.

2.2.3 Memory Leaks

One of the potential risks of using dynamic memory allocation is memory leaking. A memory leak is a block of dynamically allocated memory, which is eventually never released or can not be released. [17 p. 1] An example of the first case, when a block is never released can be seen in Listing 1.

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

int main () {
   while(1) {
       char* p = malloc (100);
       p = "hello, world\n";
       printf("%s", p);
   }
```

```
    return 0;
}
```

Listing 1. Allocated block is never released.

The example shows that the program enters in an infinite loop, allocating memory for a string on each iteration, which is never freed. Each call to *malloc(3)* returns a new block from the heap, because the block allocated on the previous iteration is still marked as used, although in reality, the program never uses it again. The second case illustrates a situation, in which the allocated block cannot be released even if it is explicitly passed to *free(3).* An example is shown on Listing 2.

```
#include <stdlib.h>

int main () {
    char* p = malloc (100);
    p = NULL;
    free (p);
    return 0;
}
```

Listing 2. Allocated block cannot be released.

Memory is initially allocated in line 3, and then it is set to NULL, resulting in the loss of the memory block. This is because assigning NULL to a pointer changes its initial contents, hence providing the pointer without a value to *free(3)* is inconsequential, as discussed in Section 2.2.2.

As has been mentioned earlier in Section 2.1, before starting the process, the kernel loads the executable file to prepare its environment, hence allocating resources for its VAS, by invoking the *execve()* system call. When a process terminates, the kernel cleans up all memory owned by the process, including dynamically allocated memory in process address space. [1 p. 99-220] Memory leaks in small applications can be dangerous, as they can lead to the wastage of a machine's resources. Nevertheless, as mentioned above, they are handled

by the kernel upon the termination of the process, causing a temporary increase in memory usage only for a brief period of time. Alternatively, resource leakage in long-running processes may lead to severe consequences and threaten the runtime of the application or the system itself. Even a minor leak in a long-running application will eventually lead to running into an OOM, subsequently crashing the program. While most modern desktop or server machines have enough resources to ‘stay alive’ during the memory leaking, embedded devices are memory-wise way more limited and hence the effects of leaking may occur much sooner. [17 p. 1, 19 p. 1-3, 20]

The phenomenon of the decreased performance or functionality of an application can be referred to as *software aging*. Occurrence of such phenomena can be noticed in long-running applications, which are most vulnerable to memory leakage. Although *software aging* is essentially a result of an array of errors in long-running software, most of the faults are related to memory leaking. They are classified as volatile effects of the process, since its re-initialization removes them until further notice. Non-persistent faults are hard to monitor, since they are present during runtime and affect the runtime environment; hence the system must be monitored while it is running. [19, 21]

2.3 Techniques for Detecting Memory Leaks

There is a wide range of tools available for efficient memory leak detection, each with its own purpose and use-case. These tools can be categorised into two classes: static and dynamic techniques. [19 p. 4-5]

2.3.1 Static Techniques

Utilising static methods does not involve the execution of a program. Rather its code is analysed in a fixed manner. One common approach includes leak detection via Guarded Value Flows, where the flow of an allocated value is followed by a deallocation of it. The method operates on simple rules of consequent value reachability during the flow of the code. [22 p. 2-3] Another

method includes the block ownership model, particularly the detection of violation on ownership of the allocated objects or blocks. In a straightforward manner, the ownership model expects only one owning reference to an allocated object, until it is reclaimed. The individual or entity holding this reference carries the liability for either releasing the object or transferring this obligation to another pointer. [23 p. 18-19]

Static techniques have no overhead on performance of the executable, hence, do not require a separate test environment for evaluation of memory related issues. There are several tools used to analyse code using static techniques, such as *Polyspace, PREfix, FlawFinderm, LCLint* and *CppCheck*. These tools can identify the exact location of the leak and determine other weak points of the code base statically. [19 p. 4, 23 p. 16, 24, 25, 26]

2.3.2 Dynamic Techniques

Dynamic analysis involves the real-time assessment of code as it runs, providing insights during or after the execution. This is typically achieved through customised execution of the code that includes instrumentation to monitor dynamic memory allocation during runtime. Applying instrumentation results in a runtime overhead, hence decreasing the speed of the execution. Because of its distinctive characteristics, dynamic analysis is used precisely when there is no harm in creating an overhead, the memory related problem is clear or it is crucial to follow the flow of the code in real-time. [19 p. 4-5, 23 p. 16]

The most common approach of detecting memory leaks during runtime involves preloading a debugging shared library before the executable program, replacing dynamic memory allocation functions with ones supplied by the library. Replaced functions instrument code during runtime to store allocated data and analyse it effectively, hence creating a performance overhead. Preloading is applied by many well-known dynamic memory debugging tools, such as *Valgrind, mtrace, AdressSanitizer* and *dmalloc*. [18, 20, 27 p. 3, 28, 29]

Although the preloading approach in the mentioned tools is the same, the logic behind the symbols differ tremendously. For instance, *Valgrind* uses a form of dynamic binary instrumentation to analyse and track the execution of a program. Injected instructions create a safe environment for the process, which is fully controlled by *Valgrind*, excluding system calls. Furthermore, the tool is not dependent on GNU C libraries to avoid any collisions with standard library symbols and hence is fully autonomous in tracing process faults. The isolated environment makes it possible to register each call to memory allocation. [27 p. 3-6] On the other hand, *mtrace* simply hooks onto the *malloc(3)* family function symbols and injects additional instructions for tracking purposes. Monitored information is later used to generate trace reports that identify memory-related issues in a program. [18] As can be observed, while there is a vast difference in the logical implementation of the preloading mechanism, the technique of instrumenting calls to dynamic memory allocation functions stays the same across all dynamic memory leak detection tools.

2.4 Challenges of Memory Monitoring in Long-Running Applications

Long-running applications vary in parameters. There is no straightforward definition to determine the minimum period of time for an application to be considered long-running. Therefore, this thesis will not define it on the global scope. However, within the scope of this thesis, Long-Running Application (LRA) is defined as a process, which in normal cases is terminated only by an external event or intrusion, which has a right to end it. This definition will mimic the corresponding model of a service, similar to one of Ericsson's products.

Large applications often involve complex patterns and equations, which require significant computational power. To manage these resources efficiently, modern programs employ multithreading, a concept that enables applications to divide tasks into separate threads within a single process. [2 p. 546-547] Multithreading requires initialization, which by itself relies on dynamic memory allocation. There are many allocations made at the startup phase to allocate

minimal resources for a complex application to effectively work. After the startup phase, the execution phase begins, invoking the main program within separate threads. Considering that threads run asynchronously within one application, memory allocation inside of them is also happening independently. Alike, threads with different parameters can lead to varying runtime semantics, making it harder to standardise the flow of the allocated object.

With all that in mind, debugging LRA must include either a real-time analysis in conjunction with live output data or a way to end the running process after a certain period of time, specific to an application to produce sufficient amount of resources to be monitored. Additionally, depending on allocation frequency, the overhead added from the dynamic technique, described in section 2.3.2, can affect the performance and sometimes the functionality of the process. Thus, with more frequent systems, the latency of the live output, which acts as the runtime overhead, must be controllable by the user.

Moreover, keeping records of the initial state appends to the execution overhead, creating redundancy in monitoring. Objects allocated during the setup phase serve as the foundation for the execution phase, until they are no longer needed after its completion. Since the model describes a system that stops only due to an external event, the application does not take control of startup objects during the execution phase, and therefore, they do not require strict monitoring. Controlling the exact point of the beginning of monitoring is crucial in debugging the memory management in LRA.

Besides latency of the live output and starting point for leak detection, the allocated blocks have a lifetime interval, which can be defined as the estimation of the time between its allocation and deallocation. A pointer to an allocated object may be or is used for an extended period of time before it is safely deallocated. The monitoring system must detect when the pointer exceeds its lifetime interval and hence expires.

The parameters described above are suggested requirements for maximising memory management monitoring performance and accuracy in an LRA. None of the dynamic memory allocation tools mentioned earlier in Section 2.3.2 have all three parameters in one, to adjust the monitoring of a long-running system. According to documents, *Valgrind* and *AddressSanitizer,* as powerful dynamic monitoring tools capable of providing real-time output, do not offer customisation options for setting the lifetime interval of memory allocations or specifying the starting point for memory leak detection. [27, 29] The development of a custom framework is necessary to create a system that can effectively utilise all three parameters for efficient memory allocation monitoring in LRA.

3 Framework Design, Architecture and Implementation

Monitoring memory allocation and tracing possible leaks in real-time are the key objectives of the developed framework, called *leaktracer*. In addition, portability plays an important role, hence the customisation of the framework runtime involves various parameters to control the flow of the monitoring. From the user perspective, there are two files to interact with: *build.sh* and *liblt.so*. By executing the *build.sh* script, the build folder is created, containing the shared library object file *libtlt.so* compiled from the source code.

The framework is intended for use with various executables without requiring recompilation of either the executables or the framework itself. To achieve this, the framework is compiled as a *shared library object*, constructed from individual modules, and consolidated into a single source file. [2 p. 316] The core idea lies in the replacement of dynamic memory allocation functions with predefined copies, applying additional logic to track their usage. This approach is utilised in numerous tools designed for dynamic memory leak detection, as previously discussed in section 2.3. Therefore, this framework shares a similar dynamic characteristic.

Each module is designed to act as an independent component, resulting in modular micro-libraries within one framework. This solution ensures a possibility of testing each module on its own and hence minimising the scope of the error-handling during the development and runtime processes. While the modules can act independently, their functionality is combined in the main source file, creating the entity of a shared library.

The shared library has an entrypoint mechanism that leverages the compiler's ability to set attributes on the library initialisation function, ensuring its execution prior to the *main()* function. The initialisation routine is responsible for allocating initial resources needed to monitor the memory management and initiating links between dynamic allocation symbols and the framework. [30] Likewise, the end

of the program has a mechanism to release resources allocated at the beginning and report any leftover allocations made by the main program.

3.1 Utilisation of Dynamic Linker

The preloading of custom symbols is accomplished by utilising the *dynamic linker* program. *Dynamic linker* is responsible for preparing the environment of the program, by finding needed shared objects for it and loading them to execute the binary. Every Linux program requires shared objects, unless the executable is explicitly marked as static. Linking is happening through inspection of string tokens appearing within the binary. Nevertheless, there is a possibility to expand string tokens within the binary dynamically. One of the possible scenarios of string token expansion is preloading of custom symbols. To achieve this, the linker has an environmental variable called LD_PRELOAD. Essentially, it allows preloading a shared object within the binary and overriding the initial functionality with custom logic. [31]

Using the linker variable to preload symbols is a common approach to link the dynamic library with the executable code. This exact method is harnessed by many powerful debugging tools, such as *Valgrind* and *mtrace*. [18, 27 p. 3] No recompilation or relinking of the ordinary executable is required for preloading the symbols, shaping the portable and lightweight method to inject the framework's code into the supervised program.

3.2 Modules and Components

As mentioned above, the *leaktracer* framework is created with a modular approach. Every module can be considered as a separate entity, which is revised independently from others. However, together, components form a cohesive implementation of the memory leak detection mechanism.

There are three main modules in the framework: *internal allocations, process maps* and *allocation tree.* Two of these components are used through the

lifetime of the detection (*allocation tree* and *process maps*) and one during the initial startup phase (*internal allocations*). In addition to three main modules, there is one small library for running unit tests, described in more detail in Section 4.4.1.

3.2.1 Internal allocations

By utilising the dynamic linker to replace the *malloc(3)* family functions, described in section 2.2.2, their functionality is overridden by custom implementation and hence there has to be a way to preserve the original features with the ability to apply additional logic. To achieve this, *leaktracer* collects the addresses of the original *malloc(3)* functions and stores them in corresponding function pointer variables. The collection of addresses is done via the dynamic linking library's function *dlsym(3)* during initialisation, demonstrated on Listing 3.

```
system_malloc = dlsym(RTLD_NEXT, "malloc");
system_calloc = dlsym(RTLD_NEXT, "calloc");
system_realloc = dlsym(RTLD_NEXT, "realloc");
system_free = dlsym(RTLD_NEXT, "free");
```

Listing 3. Obtaining addresses of *malloc(3)* family functions

As can be noticed in Listing 3, the invocation of *dlsym(3)* uses the parameter RTLD_NEXT, instructing the function to locate the next occurrence of the symbol specified as the second argument. This 'next' corresponds to the one following the current, therefore enabling the discovery of the original function, because the current symbol is the one, within the library. [32]

Before the address of the original *malloc(3)* is obtained, the invocation of *dlsym(3)* requires memory allocation internally. Because the current symbol for *malloc(3)* is reserved by the redefinition in the *leaktracer*, there is a need for a temporary memory allocator, which does not collide with the original

implementation. As a consequence, the *internal allocations* library is implemented to supply temporary memory allocation to *dlsym(3).*

The implementation of the allocator is based on a solution, similar to the *brk(2)* system call usage, described in Section 2.2.2 in more detail. Allocation is primitively done by allocating a big chunk of memory and moving the current location on it during each allocation. A substantial memory block for storing the addresses of temporary allocations is located on the stack memory segment and hence is not modifiable during the runtime. Originally, the internal allocator was needed to substitute the replacement of *malloc(3)* during the invocation of *dlsym(3)* and will not be needed later. The implementation of the temporary allocator is presented in Listing 4.

```
static byte lt_xalloc_buffer[LT_XALLOC_BUFFER_SIZE_B] = {0};
static byte *lt_xalloc_buffer_holder = lt_xalloc_buffer;
static inline size_t round_to_eight_bytes(size_t size) {
 return ((size + 7) & ~7);
}
void *lt_xalloc(size_t size) {
      const size_t r_size = round_to_eight_bytes(size);
      assert((lt_xalloc_buffer_holder - lt_xalloc_buffer + r_size)
      <  LT_XALLOC_BUFFER_SIZE_B);
      void *ptr = lt_xalloc_buffer_holder;
      lt_xalloc_buffer_holder += r_size;
      return ptr;
}
```

Listing 4. Temporary buffer allocator.

The `lt_xalloc` function represents the model of a primitive allocator, which uses the `lt_xalloc_buffer` variable in conjunction with `lt_xalloc_buffer_holder` to supply the most recent address. It is worth noting that, before a new address is supplied, the provided size is aligned to 8 bytes, mirroring the implementation of *malloc(3)*, described in section 2.2.2. The alignment ensures that the address is capable of holding diverse types of data structures assigned to it. Rounding the size is achieved through addition of 7 and setting the last three bits to 0. Adding 7 to the original size shifts the

number to the next highest multiple of 8. Setting the last three bits to 0, by applying the bitwise AND with the inverted number seven, ensures rounding up to the closest multiple of 8. Thereafter, the buffer's ability to provide a chunk of a requested size is asserted in advance of assigning the pointer variable to the next address in the buffer.

As mentioned earlier, the internal allocator is provided to replace the original *malloc(3)* momentarily. In the light of this, the buffer has a static size, defined as a macro LT_XALLOC_BUFFER_SIZE_B, expanded to a size of 256 KB. The given size is defined through testing the internal allocation of *dlsym(3)* on *x86* systems with various hardware configurations supplying the additional space for exceptional circumstances. Furthermore, the potential OOM issue is reported through assertion. In the unlikely event that the system requires additional space for its internal allocator, it can be expanded by modifying the macro mentioned above.

There are several other functions in the library designed to mimic the *realloc(3)* and *calloc(3).* Internally, they request the pointer from lt_xalloc and perform additional operations on it. Hence, they are not described in more detail. Additionally, there is no *free(3)* implementation in the *interna*l *allocations* library, because there is no convenient way of releasing stack memory as mentioned in section 2.1, and implementing the complex mechanism of 'allocated' and 'free' flags is unnecessary for a particular solution.

3.2.2 Process Maps

To provide additional information regarding the source of the possible leak in code, the current stack frame can be saved to provide the call tree of the particular function. Section 2.1 stipulates that the current stack frame is stored in the stack memory segment. To retrieve the current frame from the stack, *leaktracer* employs an open source library *libbacktrace* included in many Linux distributions by default, which subsequently uses stack unwinding mechanism provided by *libgcc*. [33] The API of the *libbacktrace* specifies a function that accepts a callback to retrieve the current stack frame address, which correlates

to an entry in the specific range of the memory region in the process VAS, as noted in section 2.2.1.

To get the values of the corresponding ranges, the mapping of the VAS must be collected within the library and saved for further usage. The *process maps* library is designed specifically for that purpose. Listing 5 illustrates a data structure created for storing the information regarding a specific memory range in the VAS.

```
struct PROC_MAP_LIB {
 const char *lib_name;
 size_t lib_start_addr;
 size_t lib_end_addr;
};
```

Listing 5. A structure for storing the information about the library in VAS.

The `PROC_MAP_LIB` structure stores the memory range of the specific mapping, by remembering the beginning and end addresses. The value, later supplied to identify the location of the frame, is stored as `lib_name`. Given that one process may involve numerous mapped libraries, the *leaktracer's* storage capacity is statically pre-defined as 1024, thus allowing for comprehensive coverage of competitively large applications.

The micro-library has two public functions:

- bool collect_process_maps(const char *proc_id_maps_filename,
  struct PROC_MAP_LIB *process_libs);
- const char *get_lib_name_by_pc(uintptr_t pc,
  struct PROC_MAP_LIB *process_libs);

The first function performs the collection of mappings in the VAS of a process, stored in the linux system by the path of */proc/{process_id}/maps,* where the *process_id* is a unique number given to each process by the kernel, as

discussed in section 2.1. The example of the memory mapping of a process is presented in Listing 6.

```
...
7f6964289000-7f69642a4000 r-xp 00003000 fe:00 11811946 /usr/lib/liblz4.so.1.9.4
7f69642a4000-7f69642a6000 r--p 0001e000 fe:00 11811946 /usr/lib/liblz4.so.1.9.4
7f69642a6000-7f69642a7000 r--p 00020000 fe:00 11811946 /usr/lib/liblz4.so.1.9.4
7f69642a7000-7f69642a8000 rw-p 00021000 fe:00 11811946 /usr/lib/liblz4.so.1.9.4
7f69642a8000-7f69642b7000 r--p 00000000 fe:00 11812489 /usr/lib/libgcrypt.so.20.4.2
7f69642b7000-7f69643a7000 r-xp 0000f000 fe:00 11812489 /usr/lib/libgcrypt.so.20.4.2
...
```

Listing 6. Example of a section in a *proc/process_id/maps* file.

Meanwhile there are several components forming a mapping, the most valuable for the purpose of collecting the library ranges are the start and the end addresses, library permissions and the name of the library, or its path. The first two components of each line, displayed on Listing 6, represent the memory range of a shared object, indicating the beginning and ending addresses. The next block, separated from address range by space, marks the permissions of the library and lastly, the component at the end of each line holds the value of the shared library's path. The `collect_process_maps` function traverses through every line in the mapping and accepts it based on the four rules:

1. Permissions include the executable flag 'x'.
2. The path name does not start with a character '/'.
3. The path name involves the ".so" string identifying the library as a shared object.
4. The amount of scanned elements on each line is 8, which corresponds to the amount of components on each line.

With the rules defined above, the function effectively collects all shared objects in the mapped VAS of a process. To address stored values and give the corresponding location inside the *libbacktrace's* callback, the second function of the *process maps* library is called during the collection of stack frames.

Given the name `get_lib_name_by_pc`, the function returns a library name based on the provided program counter, alternatively, the current stack frame address. A simple iteration is performed to find the library path, based on the

evaluation of the statement, determining if the program counter falls within the scope of address range in one of the elements of a `PROC_MAP_LIB` array. If the function fails to evaluate the program counter, the value of the returned library path becomes NULL.

It should be pointed out that the collection of process maps is performed once during the initialization phase of the leaktracer, unlike the second function, which is invoked on every new allocation made by the supervised process.

3.2.3 Allocation Tree

Due to continuous runtime, the amount of dynamic memory allocation in LRA can be sufficiently large. In the worst scenario, when the allocated memory is not being released, the book of records can grow rapidly in a small period of time. Each allocation adds an overhead in a case of a traversal of the data structure, where one is stored. To manage allocations efficiently, *leaktracer* utilises the Adelson-Velsky and Landis (AVL) binary tree, alternatively self-balancing binary search tree. The idea of a self-balanced binary tree lies in balancing itself on each modifiable operation, resulting in a $O(log\, N)$ time complexity for *N* nodes. Alternatively, the unbalanced variant leads to $O(N)$ in a worst-case scenario, which corresponds to the complexity of a basic array. [34] Usual implementation implies having two children nodes on each node: left and right, where the left child holds the key value less than or equal to the parent node, and the right child holds a key value greater than the parent node.

Leaktracer has its own AVL binary tree implementation, with the ability to perform the following public operations on it:

- push(struct BT_NODE *root, void *data);
- delete(struct BT_NODE *root, void *address);
- cleanup_tree(struct BT_NODE **root);
- search_data_in_tree(struct BT_NODE *root, void *address);
- for_each(struct BT_NODE *root, int (*callback)(void *));

The first two operations `push` and `delete` are responsible for keeping the tree in balance, as both are involved in the modification of the tree. The key of the tree node lies in an address of an allocation. Since all addresses returned by dynamic memory allocators, while being used, are unique, it is a perfect value for a node key. Hence, it can be noticed that both `search_data_in_tree` and `delete` accept an address, which serves the purpose of an identifier of a block, holding all of the information regarding the made allocation. The `push` operation accepts a *void* pointer, which indicates that any data type can be transferred to it. However, the data type/structure needs to have the key and since *allocation_tree* is implemented particularly for *leaktracer's* internal functionality, the data utilises the format presented in Listing 7.

```
struct SYSTEM_ALLOC_DATA {
 void *address;
 size_t size;
 bool allocation_is_expired;
 time_t ts_created;
 struct SYSTEM_ALLOC_BT_DATA backtrace_data;
};
```

Listing 7. Leaktracer's allocation data structure.

As can be noticed, the data structure has several values holding the allocation related information, some of which are going to be examined in more detail later in Section 3.3. Nevertheless, the `address` value plays a crucial role in the correct operation of the *allocation_tree* microlibrary. Referring back to Section 2.2, the pointer to the block of memory, alternatively virtual address, returned back from an allocator function is one of the many possible values located in the VAS of a process. The value of a virtual address is essentially just a number, constructed from segments of separate page-tables, resulting in a numerical value, similar to one of the address range values presented previously on Listing 6. Taking all that into account, the address value returned from the memory allocator function has suitable features for being a key to identify the position of the data node in the binary tree.

While the both `push` and `delete` functions may modify the tree, which requires additional balancing and internal memory allocation, the `cleanup_tree` operation's purpose lies in releasing all the internally allocated memory of the tree. Essentially, the latter traverses through each node and invokes a `free` call on each element. The cleanup is done once during the execution of the main program in case of a scheduled or manual termination.

`Search_data_in_tree` is an operation performed in a case of an urge to delete a specific node from the tree. As a precautionary measure, the function is created to identify whether the `delete` operation is needed or not, to avoid possible undefined behaviour of free invocations, mentioned in Section 2.2.2. In the case of a successful match of an address provided to the function and the corresponding key of a node, the search function returns the Boolean value True. In an alternative case, where there is no match, it returns the Boolean value False.

The last function of the *allocation_tree* is essential for performing an action on each node in the tree. The `for_each` operation, besides the root node, requires a callback function, serving the purpose of an action with the possible return value. The return value of the callback function determines whether to continue the iteration further, or returns back to the root. In case of a negative or a zero value, the traversal is terminated. Additionally, the callback accepts a pointer to the data of the node, making it possible to execute modifications on it.

Leaktracer uses the API of the *allocation_tree* continuously during runtime, while capturing, managing and monitoring the allocation data. Previously discussed functions form a base for the memory leak detection mechanism, employed by the *leaktracer*, yet the correct utilisation of them is crucial during all phases of a running process.

3.3 Managing and Monitoring Allocation Data

As soon as the executable linked with the leaktracer encounters one of the allocator functions and the replaced version of it is invoked from the linked library, a set of actions is performed to manipulate the allocated block. Depending on the instruction invoked, all four *malloc(3)* family functions have a similar structure in their respective wrapper provided by a memory detection library. While there may be significant differences when examining the code in parts, on a larger scale, each wrapper function's code can be divided into four sections.

1. Pre-processing
2. Invocation of the real allocation symbol
3. Managing current allocation data
4. Monitoring allocation blocks.

To provide a better picture of each section, the replacement of *realloc(3)* is going to be taken as an example, since it has the most versatile manipulation of data compared to alternative *malloc(3)* family functions.

The first section performs conditional checks before processing any further. The whole part of the pre-processing section is revealed in Listing 8, which showcases different scenarios of the both leaktracer's internal functionality and realloc's course of action, based on parameters provided to the function from the monitored program.

```
void *realloc(void *ptr, size_t size) {
 if (system_realloc == NULL) {
   return lt_realloc(ptr, size);
 }
 if (ptr == NULL) {
   return malloc(size);
 }
 if (size == 0) {
   free(ptr);
```

```
    return NULL;
  }
...
```

Listing 8. Pre-processing section of the wrapper of *realloc(3)*.

The first check makes a decision either to use the internal allocation, which is possible only during the initialisation phase of the library as outlined in Section 3.2.1 , or to continue with further checks. The decision is made by verifying the value of the function pointer of the replacement function. In case when the function pointer is not yet initialised by the dynamic linker, the library is urged to use internal allocation and gracefully return back with a value obtained from `lt_realloc`. The next two checks follow the functionality of *realloc(3)* in corner-case situations. As suggested in Section 2.2.2, in case the value of a supplied pointer equals to NULL, the subsequent invocation of *malloc(3)* should be performed. Additionally, if the provided pointer has a non-zero value, but the provided size equals zero, the function acts as a *free(3)*, releasing the provided pointer. In both cases, the code does not proceed beyond the pre-processing phase if one of the conditions is satisfied.

Shortly after, the invocation of a real allocation symbol takes place. Because the first condition of a pre-processing phase determines the symbol is successfully linked, parameters from the *realloc(3)* wrapper are being forwarded to it, as visible in Listing 9.

```
...
void *alloc_address = system_realloc(ptr, size);
 if (use_ltlib(alloc_address) == false) {
    return alloc_address;
 }
...
```

Listing 9. Invocation of the real allocation symbol section of the wrapper of *realloc(3)*.

Leaktracer performs additional validation based on the value returned by the `system_realloc` and a few other variables. The `use_ltlib` function determines if the code should go into the next phase of managing allocation data, or to return back from the point of a simple allocation. The decision is based on the several runtime flags, determining the current readiness state of the library. If the library is ready to proceed further, the function checks whether the value passed to it is NULL or not. In case of a NULL value, no allocation is performed. Hence, there is no data to manage, and the empty value is returned back to the main program. If the depicted condition is not satisfied, the code proceeds to the next phase, shown in Listing 10.

```
...
set_system_alloc_guard_state(true);
 pthread_mutex_lock(&lt_mutex);
 if (alloc_address == ptr) {
   create_allocation_block(&allocation_tree, ptr, size);
 } else {
   pthread_mutex_lock(&alloc_tree_mutex);
   if (search_data_in_tree(allocation_tree, ptr)) {
     allocation_tree = delete(allocation_tree, ptr);
   }
   pthread_mutex_unlock(&alloc_tree_mutex);
   create_allocation_block(&allocation_tree, alloc_address, size);
 }
...
```

Listing 10. Managing current allocation data section of the wrapper of *realloc(3)*.

In case of *realloc(3)*, there are several ways the resizing of the block can happen. The managing phase illustrates the behaviour described in Section 2.2.2. By evaluating the return value of the real *realloc(3)* from the previous section, the allocation block in the tree is either modified, or firstly deleted and created with new values, following the real implementation. The `create_allocation_block` gathers all data within an instance of a SYSTEM_ALLOC_DATA structure, presented on Listing 7, and utilises a `push` operation from *allocation_tree* microlibrary to append it to the root of the tree. It is worth mentioning that when the *realloc(3)* returns a new address, the old is freed internally. As mentioned in Section 3.2.3, the `delete` operation in

conjunction with `search_data_in_tree` function removes a node containing the old data, but does not release the allocation itself, which is the responsibility of the real *realloc(3)*. It is observable that besides data manipulation, there is other functionality present during the managing phase.

Due to one of the challenges introduced above in Section 2.4, a multithreaded environment can employ simultaneous access to data, which may lead to conflicts, when several threads are trying to write into one place in memory. To synchronise the access to data, the locking mechanism called *mutex* was invented. [2 p. 564-566] The functions `pthread_mutex_lock` and `pthread_mutex_unlock` secure the vulnerable block of code, to ensure the access is safely synchronised. Another mechanism displayed in Listing 10 is *system_alloc_guard_state*, which is one of the flags checked by the `use_ltlib` function, presented above. Before starting to manage the allocation data, *system_alloc_guard_state* is set to a Boolean value True. This action results in all subsequent memory allocation calls inaccessible to the management section, hence preventing the leaktracer from tracking memory allocations done during the management phase. This mechanism effectively prevents the library from entering an infinite loop of consecutive memory allocation function invocations.

Lastly comes the monitoring allocation blocks section, being responsible for marking blocks as expired and reporting the possible leaks. While the section is comparatively small in size, it makes several subsequent calls to other helper functions resulting in monitoring of all blocks stored in the tree. The section itself is presented in Listing 11.

```
...
 catch_expired_blocks();

 pthread_mutex_unlock(&lt_mutex);
 set_system_alloc_guard_state(false);
 return alloc_address;
}
```

Listing 11. Monitoring allocation blocks section of the wrapper of *realloc(3)*.

The last section includes a function for monitoring all blocks stored by *leaktracer*, called `catch_expired_blocks`, as well as unlocking the *mutex* locked in the phase and setting the *system_alloc_guard_state* to Boolean value of False, indicating the end of the wrapper function. Returning back to monitoring of allocations, the `catch_expired_blocks` function works as a wrapper for additional *mutex* locking to protect the access to tree, and the invocation of `for_each` function accessing it. Essentially, the `for_each` operation is utilised for the purpose of applying an action on each node through a pre-made callback, operating with the data stored as SYSTEM_ALLOC_DATA structure on nodes. The callback is featured in Listing 12.

```
static int expire_nodes(struct SYSTEM_ALLOC_DATA *data) {
 time_t current_time = time(NULL);
 bool allocation_just_expired =
            (current_time-data->ts_created)    >=
            runtime_parameters.allocation_expiration_s;
 if (allocation_just_expired && !data->allocation_is_expired) {
   data->allocation_is_expired = true;
   if (fill_expired_buffer(data) == false) {
     flush_expired_buffer();
     fill_expired_buffer(data)
     return -1;
   }
 }
 return 0;
}
```

Listing 12. Callback applied to each node during monitoring of allocation blocks.

Once the callback function `expire_nodes` gets invoked, the current timestamp is saved for further comparison with the timestamp of an allocation collected from the `create_allocation_block` wrapper and put to the tree in a node containing SYSTEM_ALLOC_DATA. The condition of expiration is also dependent on the *allocation_is_expired* flag, set initially in each block to boolean value False. In practice, such a mechanism ensures blocks expired once are not expiring again, which reduces duplication in the program's output and removes redundant overhead.

Now that the monitoring mechanism has been described, it is noteworthy to observe how expired allocations are transferred to a user in real time.

3.4 Real Time Output

As outlined in section 2.3, most of the memory leak detectors output the report of the program's memory management on one's termination, creating a full summary of leaked memory. While the *leaktracer* is capable of behaving in the same manner, there is also a possibility to produce output in real time with the library. The previous section already indicated the technique of capturing the expiration of a memory block using the timestamp difference, yet there is another part, which is executed during the flagging of *allocation_is_expired* in the `expire_nodes` function, presented in Listing 12.

Leaktracer utilises a buffering technique to output gathered information in chunks of a specific size. In use, the buffer is filled with an expired allocation block every time one is found, while the buffer's capacity is reduced by one, respectively. As soon as the capacity of a buffer reaches zero and filling is not possible, all of the buffered data is flushed to a file, reviving its capacity to an initial level. After the flushing the captured block is pushed to an empty buffer. To control the size of the expired buffer, as well as a few other parameters, *leaktracer* provides a possibility to configure runtime settings through adaptive parameters.

3.5 Adaptive Parameters

Before preloading the library, the initializer function goes through a set of predefined environment variables and collects their values for adjusting the runtime behaviour of the framework. The variables are essential to provide flexibility in debugging different scale LRA, which may be challenging for a set of reasons mentioned earlier in Section 2.4. LRA can include a possible memory initialization at the beginning, which does not require tracking.

Additionally, depending on the frequency of allocation in the debuggable LRA, there must be a way to change the timeout for the allocation's expiration in conjunction with the frequency of real-time output provided by the library. *Leaktrcer* has the following adaptive parameters available for the user:

- LEAKTRACER_LOG_PATH - a file path for the log file, where *leaktracer* will output all the information regarding the monitored process.
- LEAKTRACER_CHECK_AFTER_S - a timeout in seconds from the beginning of the monitored application, after which the tracking of memory allocation begins.
- LEAKTRACER_ALLOC_EXPIRE_S - a value specifying the timeout in seconds for each allocation, until it becomes expired. If the allocation is freed before the expiration, it is not considered as a leak.
- LEAKTRACER_EXPIRED_BUFFER - a size of the buffer, holding expired allocations, which determines the amount of expirations rendered to a log file at once.

Each of the parameters have boundaries and rules. Both timeouts have a rule for the value to be in range from 0 to UINT32_MAX macro, defined in the `stdint.h` header. To provide a log path correctly, one must set the variable to the full path of the file, starting from the system's root directory. Finally, the buffer size must have a value set between 0 and 1024. In addition to rules, there are macros defined with default values for each parameter presented in the following Listing 13.

```
#define LT_PARAMS_DEFAULT_OUTPUT_FILE_PATH_VALUE "/tmp/leaktracer"
#define LT_PARAMS_DEFAULT_CHECK_AFTER_VALUE 0
#define LT_PARAMS_DEFAULT_ALLOCATION_EXPIRATION_VALUE 60
#define LT_PARAMS_DEFAULT_EXPIRED_BUFFER_SIZE 16
```

Listing 13. Default values for *leaktracer's* runtime parameters.

The log of the process is being stored in the */tmp* directory by a filename *leaktracer* followed by a postfix containing the process id (PID) of the monitored

process. The memory allocation monitoring is initiated at the start of the supervised process with the default value set to zero, and each allocation is set to expire after 60 seconds. The buffer of the expired allocations is automatically flushed with a default value every 16th expiration or when the program is terminated before the buffer is filled.

4 Development and Testing

4.1 Development of the Framework

The first commit in the *leaktracer's* repository is dated to 4th June 2022, initiating the first iteration of the memory allocation debugging project. Before the alpha release, three iterations took place, each employing distinct implementations and techniques for instrumenting, storing, and analysing allocated memory.

At the beginning, there was no presence of utilising a dynamic linker to inject the instrumentation of the *malloc(3)* family functions. To enable monitoring of each allocation, a set of macros was defined to substitute the initial calls with *leaktracer's* functions. This approach utilises a preprocessor statically instrumenting custom functionality to memory allocators. Compared to current functionality, the preprocessor instrumentation is inferior, because it requires precise placement of substitute definitions on top of all other headers. Moreover, if something is linked with the LD_PRELOAD mechanism, it may be impossible to track linked code with preprocessor instrumentation.

During the first iteration up until the last, memory blocks were stored in a linked-list data structure, which enabled fast inserting of a new block, yet finding and deleting took a substantial amount of time. In an LRA with frequent memory management, each allocation appended an overhead, which could not be managed as quickly as it grew. The difference in time can be noticed in Figure 3.

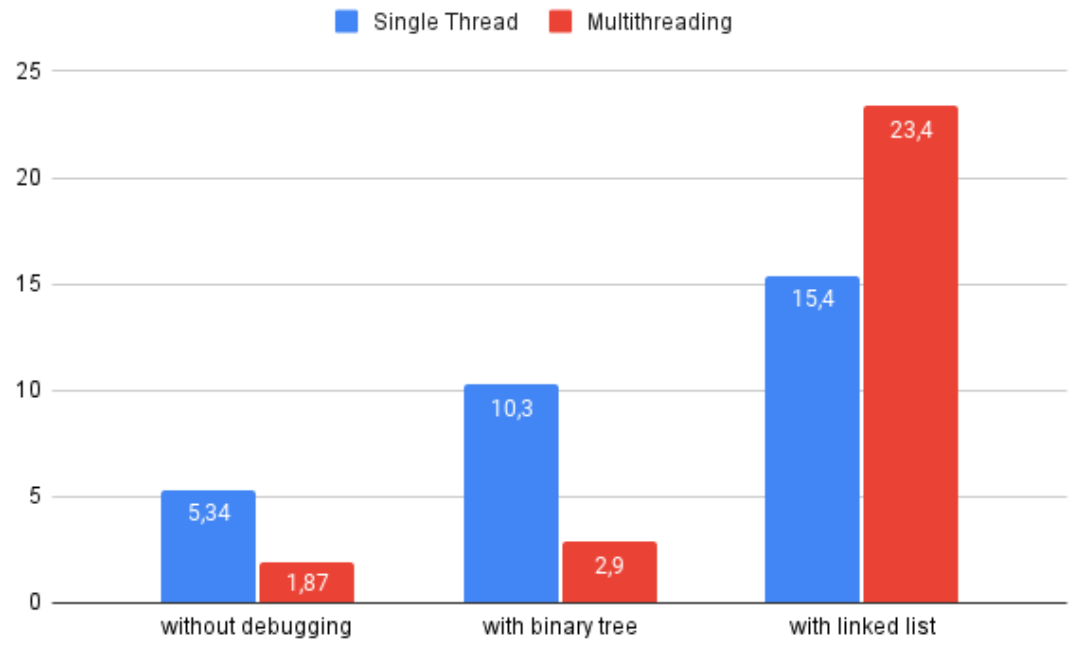

Figure 3. Comparison of leaktracer's performance time with a neural network application.

The graph above shows the time taken in seconds for a process to execute with different versions of *leaktracer's* allocation block management and without *leaktracer*. The application being supervised is a neural network made for analysing trading data, written in C++. [35] The neural network application is able to run in both multithreaded and single threaded modes. It is noteworthy that the application is resource heavy. Thus, depending on hardware, the consumed time may vary. As can be noticed, the performance increase with the binary tree implementation is tremendous compared to the performance of the older version with a linked list. The time difference between the binary tree and the linked list on a single thread is 5.1 seconds, while with multithreading, it is 20.5 seconds. Hence, the library's overhead is comparatively lower when utilising a binary tree. Specifically, the linked list shows an overhead of 10.06 seconds on a single thread and 21.52 seconds with multithreading, whereas the binary tree demonstrates an overhead of 4.96 seconds on a single thread and 1.02 seconds with multiple threads. The result is achieved by applying different techniques during the development phase and improving them on each iteration of a project.

Overall, the iterative approach to the development of the project helped to identify different solutions for each problem occurring during the implementation phase. The current solution consisting of modules and components is the most stable compared to previous versions. Thanks to its modular structure, the implementation of new features occurs seamlessly, involving the following steps: module initialization, unit testing, expansion of core functionalities and testing the whole system.

4.2 Testing

During the whole development phase of the *leaktracer*, tests played a major role in the implementation of the software. The library is currently undergoing testing at two levels: the unit level and the system level. With action rules defined in GitHub, each git `push` operation triggers an automatic launch of the tests for the latest commit. Both unit and system tests are deployed in containerized environments, enabling the execution on any platform with Docker installed, completely isolating the testing environment from the main userspace. It is worth mentioning that tests are configured in such a way that they cannot be executed outside of containers.

4.2.1 Unit Tests

Referring back to Section 3.2, it was mentioned that one of the *leaktracer's* modules implements a simple unit testing library. The library was written to unload the codebase from redundant dependencies. Essentially, there are only a few methods the library provides for successful unit test execution:

- void assert_true(int condition, const char* message)
- void assert_false(int condition, const char* message)
- int run_all_tests(TestCase * tests, size_t amount)

The basic assertion of a condition is done in both ways, when the condition needs to be true and otherwise. Additionally, when the assertion fails, a message provided to the function is printed alongside the assertion result. The run_all_tests function accepts an array of test cases, consisting of TestCase structures. Each instance of the structure stores a callback function for the test and a string with the name of the test case. When all tests are successfully passed, the function returns value 0, or value 1 in an alternative situation. The returned value can be passed to the main function, which will determine the final outcome of the unit test suite.

The compilation and execution of test cases are managed by a Bazel build tool, which utilises a caching technique and the capability to tag executables as tests, hence ensuring effortless management of unit test builds. [36] An example of a successful run of all test suites bootstrapped with Bazel is illustrated in Listing 14.

```
INFO: From Testing //tests:lt_alloc_unit_test:
==================== Test output for //tests:lt_alloc_unit_test:
Running test: Test normal behavior of lt_xalloc
[PASS] lt_xalloc shall not return NULL

Running test: Test normal behavior of lt_xralloc
[PASS] lt_xralloc shall not return NULL
[PASS] Reallocated memory matches old memory

...

Running test: test_search_data_in_tree
[PASS] Data should be found in the tree

Running test: test_cleanup_tree
[PASS] Root should be NULL after cleanup

Total tests: 35, Passed: 35, Failed: 0
All tests passed!
INFO: Elapsed time: 9.111s, Critical Path: 0.46s
```

Listing 14. An output of *leaktracer's* unit test run.

In total, 35 unit test cases are executed automatically in GitHub Actions, covering all public functionality of each module, described in Section 3.2. It was decided that even minor functionality shall be tested to avoid situations, where the reason for unexpected behaviour of the library may be caused by one of the modules. However, to determine if the library works correctly in general, there should be a way to check its functionality on a higher level, with system tests.

4.2.2 System Tests

There are various scenarios for Long-Running Applications, including multithreading environments, frequent allocations, long running time and initial memory setup. To cover possible runtime scenarios, a set of system tests was implemented to run with the *leaktracer*. All of the test files are C applications, which are executed with predefined values for *leaktracer's* runtime parameters. After the process is finished, the report file for each executable is read by a corresponding test case implementation, which is done in the Robot Framework. [37] Mainly, three conditions must be satisfied for the test case to pass: the report file exists, the actual length of the file is the same as the expected length, and if the test should generate expired blocks, they are present in the report. Considering these factors, a set of mock scenarios are employed to effectively check the system behaviour of the library through system tests:

- **mock_init_25KB_and_exit** - Allocating a total of 25 kilobytes in blocks of 5 bytes and exiting the application without releasing them. The purpose of the test is to check for the ability to detect all leaked blocks.
- **mock_init_25KB_and_free** - Allocating a total of 25 kilobytes in blocks of 5 bytes and releasing them before exiting. After the test, the report should not have any leaks present.
- **mock_init_memory_and_sleep_for_30s** - Initializing the application with a small allocation at the beginning and again allocating memory at the end. This scenario ensures the possibility to skip the monitoring of the initial phase of the application, concentrating on the allocations made after the timeout.
- **mock_multithread_init_memory_and_exit** - Creating several threads and allocating memory in each thread, without releasing at the end. The test aims to confirm the correct behaviour of the *leaktracer* in a multithreading environment.

- **mock_test_all_allocators** - The test utilises all *malloc(3)* allocator functions to guarantee their correct behaviour with the library's instrumentation.
- **run_for_1_hour_allocate_each_15min** - This is the LRA emulation, which runs for one hour, allocating memory each quarter of an hour and subsequently releasing it.

Altogether, system tests cover the major part of the *leaktracer's* functionality by testing the known corner cases of allocator functions and adaptive parameters. While the system tests broadly demonstrate the library's core functionality with applications mimicking real LRA systems, there are several limitations to library's usage that are worth noting.

4.2.3 Testing with the real LRA

The early phase testing of the library's integration into Ericsson's LRA was launched to identify potential conjunction issues between the library and the main application. Furthermore, there was an opportunity to assist in identifying a potential memory leak in an internal maintenance issue within the company, concerning a rise in memory usage in one of the products. The deployment of the application with preloaded memory allocator symbols was achieved by modifying certain build parameters in the LRA compilation system and execution scripts. With all modifications, *leaktracer* was successfully set in motion to run with a real product on side. After one night of execution under disturbance, with the expiration timeout set to 5 minutes, the dynamic debugging library reported a potential location of the leak. The discovery assisted in pointing out the exact location of the potential leak and hence assisted in the ongoing uplift of an OpenSSL functionality, which required replacement of the deprecated API, causing the leak in the system.

4.3 Addressing Implementation Limitations

Due to the instrumentation mechanism used in the *leaktracer*, the current suitable platform for its operation is GNU Linux. The replacement of the memory allocators happens through dynamic linker library functions, as mentioned in Section 3.2.1. Extending the information written above, certain *dlsym(3)* arguments require a definition of a _GNU_SOURCE macro to be obtained, hence not being a part of platform-independent C code. [32] Moreover, the LD_PRELOAD mechanism does not work the same way on other platforms, necessitating the implementation of platform-specific linking for dynamic symbol injection.

Another limitation of the library is its overhead in the matter of managing monitored allocations. Currently, with its own implementation of an AVL binary tree, the performance of the *leaktracer* has increased significantly from the last development iteration employing a linked list for the same purposes, which is observed in Figure 3. While the AVL binary tree has a time complexity of $O(log\ N)$ for most of its operations, when the monitored application becomes vast and significantly complex, the overhead is still noticeable. The increased overhead may result in a change of behaviour in a monitored program, which is a problem for applications that are time sensitive.

While the library is multithread-safe, the potential creation of another process by LRA running alongside the original application introduces an area of concern. The detached process monitoring is not implemented and hence anything related to its handling can be considered as an undefined behaviour.

4.4 Future Plans

Integration to Ericsson’s development environment is the main focus on the current stack of future plans. As of present, the library was executed within one of the products successfully and generated informative results. Because of the security precautions within the Ericsson’s LRA there is no easy way of

integrating the library into the deployment action, hence the launch was blended manually by changing certain build parameters in the application. Inserting the memory monitoring tool as one of the runtime parameters within the product requires proper planning and execution, which is not covered in the scope of this project. The dynamic memory check can act as a manual tool for the developer that can be launched with parameters adjusted on the fly or the automated Continuous Integration (CI) job, executed based on the premade rules. The example of the latter is demonstrated in the discussion of system tests in Section 4.2.2. Due to there being a choice, as *leaktracer* is now prepared and successfully deployed manually within the Cloud Native LRA, the question of its integration remains open, and this matter will be discussed in further detail internally within the company.

Alongside the integration process, there is a need to address the limitations highlighted in the previous section, leaving room for further improvement. Enhancing the dynamic detector will involve prioritising and subsequently resolving the current issues. One of the plans remains to maintain the project as much as possible during and after integration. The biggest concern is the overhead that comes along with the binary tree time complexity. One possible solution would be to introduce a hashmap as a managing data structure, which would replace the current *allocation_tree* module. The average time complexity of the hashmap's operations is $O(1)$, which is significantly superior to an average time complexity of an AVL binary tree $O(log\ N)$. [38] Moreover, the current implementation employs a recursive technique, which expands the stack during the application's runtime, as discussed in Section 2.1. Hence, the runtime memory usage could potentially be decreased by replacing the allocation management module with a hashmap.

To provide more information about the performance of the library, after all technical issues are resolved and possible optimizations are in place, the comparison between other dynamic memory debuggers needs to be done. This may show further enhancement strategies for the leaktracer and be put in place

as a variant of memory detection application with additional features specialised for real time debugging.

5 Conclusion

Essentially, the main goal of this project was to develop a real time memory leak debugging framework, with the ability to adapt to the runtime environment of the monitored application. At the time of writing this thesis, the final version of the library is capable of monitoring the memory management of the LRA, generating real-time output based on the runtime parameters set before launch. All challenges related to memory monitoring in LRA, described in Section 2.4, are also covered by the framework's adaptive parameters.

The common pattern of having an initial phase in the application, which allocates memory for its whole lifetime, produces a significant amount of redundant blocks for the monitoring process. One of the adaptive parameters within the *leaktracer* is capable of adjusting the timeout for the start of the monitoring phase, to skip the initialization supervision. Additionally, the real time analysis produces an extra overhead to the main application’s runtime. Hence, depending on frequency of allocations, there is a need to control the amount of the added overhead. Another adaptive parameter of the library is capable of customising the portion of the live output, which correlates to the excess of the overhead. Furthermore, the expiration of allocations may be modified in the same pattern, allowing the library to determine the lifespan of each allocation. In addition to real-time output, the library reports all remaining blocks upon process termination, enabling it to work similarly to its variants, yet encompassing additional features needed for LRA debugging.

The first iterations of the development involved memory block management through linked list data structure, which produced a significant amount of runtime overhead. In the latest development phase, this structure was replaced with a self-balancing binary tree, leading to a nearly fifty percent reduction in overhead in a single-threaded environment and an approximately eight times decrease in a multithreaded environment, as illustrated in Figure 3. However, as discovered in Section 4.3, the overhead is increasing in line with the expanding amount of monitored allocations, indicating the need for further revision of the monitoring data structure.

The future goals involve maintaining the library’s features and upgrading its functionality. As has been mentioned in Section 4.4, one potential candidate for the new management data structure is a hashmap. Its implementation in the framework may increase its performance even further in addition to reliability of the stored data. Alongside the plans of maintaining the project, there is a need to consider possible integration possibilities within Ericsson's development environment. The library is rendering the live output into a separate file, which can be used to analyse the data after the execution or on the fly. This approach provides several options for its integration. The debugging can be placed as a part of a CI pipeline with tests analysing the output or used as a tool for developers to execute monitoring on any binary program.

In total, the developed library, with its features to monitor memory management in real time and to modify the runtime semantics of the monitoring, provides a unique approach to detecting memory leaks in long running environments.

References

1 Daniel P. Bovet,. Marco Cesati. Understanding the Linux Kernel. 2000. O'Reilly.

2 W. Richard Stevens., Stephen A. Rago. Advanced Programming in the UNIX® Environment, Second Edition. 2005. Addison Wesley Professional.

3 Linux Documentation Project. environ(7) - Linux Manual Page. [Online]. Accessed on 17 September 2023. Available from: https://man7.org/linux/man-pages/man7/environ.7.html

4 Gabrielle Tolomei. In-Memory Layout of a Program (Process). [Online]. 2015. Accessed on 17 September 2023. Available from: https://gabrieletolomei.wordpress.com/miscellanea/operating-systems/in-memory-layout/

5 Vlad-Andrei Cursaru. Improving the Dynamic Elimination-Combining Stack Implementation. 2022. Amsterdam: Vrije Universiteit Amsterdam

6 Gabrielle Tolomei. Virtual Memory, Paging, and Swapping. [Online]. 2014. Accessed on 17 September 2023. Available from: https://gabrieletolomei.wordpress.com/miscellanea/operating-systems/virtual-memory-paging-and-swapping/

7 Linux Documentation Project. getrlimit(2) - Linux Manual Page. [Online]. Accessed on 17 September 2023. Available from: https://man7.org/linux/man-pages/man2/getrlimit.2.html

8 Kamp, Poul-Henning. Malloc (3) revisited. 1998. USENIX Annual Technical Conference (USENIX ATC 98).

9 Linux Documentation Project. malloc(3) - Linux Manual Page. [Online]. Accessed on 18 September 2023. Available from: https://man7.org/linux/man-pages/man3/malloc.3.html

10 Moreta, S., & Telea, A. Visualizing Dynamic Memory Allocations. 2007. *4th IEEE International Workshop on Visualizing Software for Understanding and Analysis* (pp. 31-38). IEEE.

11 Denning, P. J. Virtual Memory. 1996. *ACM Computing Surveys (CSUR)*, *28*(1), 213-216.

12 Linux Documentation Project. brk(2) - Linux Manual Page. [Online]. Accessed 21 September 2023. Available from: https://man7.org/linux/man-pages/man2/brk.2.html

13 ISO-9899.info. ISO/IEC 9899 - Information about the C Programming language standards. [Online]. Accessed 21 September 2023. Available from: www.iso-9899.info

14 Lipp, M., Schwarz, M., Gruss, D., Prescher, T., Haas, W., Mangard, S., ... & Hamburg, M. Meltdown. 2018. *arXiv preprint arXiv:1801.01207*.

15 Parlante, N. Linked list basics. 2001. *Stanford CS Education Library*, *1*, 25.

16 Kernighan, B. W., & Ritchie, D. M. The C Programming Language. 2002. Prentice−Hall.

17 Andrzejak, A., Eichler, F., & Ghanavati, M. Detection of memory leaks in C/C++ code via machine learning. 2017. In *2017 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)* (pp. 252-258). IEEE.

18 The GNU C Library. Allocation Debugging. 2008. [Online] Accessed on 3 October 2023. Available from: https://www.gnu.org/software/libc/manual/html_node/Allocation-Debugging.html

19 Bhavana, D., Veena, M. B., & Sahu, S. K. An Automated Approach of Detection of Memory Leaks for Remote Server Controllers. 2020. *EMITTER International Journal of Engineering Technology*, *8*(2), 477-494.

20 Erickson, C. Memory Leak Detection In Embedded Systems. 2002. *Linux Journal*, *2002*(101), 9.

21 Grottke, M., Matias, R., & Trivedi, K. S. The Fundamentals Of Software Aging. 2008. In *2008 IEEE International conference on software reliability engineering workshops (ISSRE Wksp)* (pp. 1-6). Ieee.

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

23 Heine, D. L. *Static Memory Leak Detection*. 2005. Stanford University.

24 Bush, W. R., Pincus, J. D., & Sielaff, D. J. A Static Analyzer For Finding Dynamic Programming Errors. 2000. *Software: Practice and Experience*, *30*(7), 775-802.

25 Evans, D., Guttag, J., Horning, J., & Tan, Y. M. LCLint: A Tool For Using Specifications To Check Code. 1994. *ACM SIGSOFT Software Engineering Notes*, *19*(5), 87-96.

26 Cppcheck. [Online]. Accessed on 3 October 2023. Available from: http://cppcheck.net/

27 Nethercote, N., & Seward, J. Valgrind: A program supervision framework. 2003. *Electronic notes in theoretical computer science*, *89*(2), 44-66.

28 Watson G. Debug malloc home page. [Online] Accessed on 4 October 2023. Available from: https://dmalloc.com/

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

30 The GNU Compiler Collection. Function Attributes.[Online] Accessed on 9 October 2023. Available from: https://gcc.gnu.org/onlinedocs/gcc-4.7.0/gcc/Function-Attributes.html

31 Linux Documentation Project. ld.so(8) - Linux Manual Page. [Online]. Accessed 9 October 2023. Available from: https://www.man7.org/linux/man-pages/man8/ld.so.8.html

32 Linux Documentation Project. dlsym(3) - Linux Manual Page. [Online]. Accessed 9 October 2023. Available from: https://man7.org/linux/man-pages/man3/dlsym.3.html

33 Ian Lance Taylor. libbacktrace. [Online]. Accessed 10 October 2023. Available from: https://github.com/ianlancetaylor/libbacktrace

34 Erik Alexander. AVL Trees. [Online] Accessed 17 October 2023. Available from: https://pages.cs.wisc.edu/~ealexand/cs367/NOTES/AVL-Trees/index.html

35 Mikhail Stefantsev. Trader. [Online] Accessed 29 October 2023. Available from: https://github.com/MStefan99/trader

36 Bazel. Bazel: Fast, Correct, Reproducible Builds. [Online] Accessed on 30 October 2023. Available from: https://bazel.build/

37 Robot Framework Foundation. [Online] Accessed on 30 October 2023. Available from: https://robotframework.org/

38 Mehlhorn, K., & Sanders, P. "4 Hash Tables and Associative Arrays." Algorithms and Data Structures: The Basic Toolbox. 2008. Springer, pp.81-98.