Linköping University | Department of Computer and Information Science
Master's thesis, 30 ECTS | Datateknik
2025 | LIU-IDA/LITH-EX-A--2025/001--SE

Memory Allocation Tracing for C++ Applications in Live Systems

Spårning av minnesallokeringar för C++-applikationer i aktiva system

Martin Högstedt
Fabian Johansson

Supervisor : Dominik Drexler
Examiner : Jendrik Seipp

External supervisor : Matus Maruna

LINKÖPING UNIVERSITY

Linköpings universitet
SE–581 83 Linköping
+46 13 28 10 00 , www.liu.se

Upphovsrätt

Copyright

Abstract

The abstract resides in file `Abstract.tex`. Here you should write a short summary of your work.

Acknowledgments

Baljan och Byttan

Contents

Abstract iii

Acknowledgments iv

Contents v

List of Figures vii

List of Tables viii

1 Introduction 1
- 1.1 Motivation . . . 1
- 1.2 Aim . . . 2
- 1.3 Research questions . . . 2
- 1.4 Approach . . . 2

2 Background 3
- 2.1 Types of Memory Leaks . . . 3
- 2.2 History and Impacts of Memory Issues . . . 3
- 2.3 Stack Frames . . . 4
- 2.4 User and Kernel Space in Linux . . . 5
- 2.5 Dynamic Libraries in GNU/Linux . . . 5
- 2.6 objdump . . . 6
- 2.7 Perf . . . 6
- 2.8 Lightweight Dynamic Tracing of User Apps . . . 7
- 2.9 eBPF . . . 9
- 2.10 LLVM . . . 12

3 Related Work 13
- 3.1 Memwatch . . . 13
- 3.2 Dynamic Tainting . . . 14
- 3.3 Dynamic Binary Analysis and Binary Instrumentation . . . 14
- 3.4 Dynamic Instrumentation . . . 14
- 3.5 SWAT . . . 16
- 3.6 Sniper . . . 16
- 3.7 Large scale memory detection . . . 17
- 3.8 PROMT . . . 17
- 3.9 AddressSanitizer . . . 17
- 3.10 MESH: A Memory-Efficient Safe Heap for C/C++ . . . 17
- 3.11 Snapshot . . . 18
- 3.12 AddressWatcher . . . 18

4 Project Plan 20

4.1 Literature Study . . . 20
4.2 Development Phase . . . 21
4.3 Time Plan . . . 23

5 Results **25**
5.1 Literature study results: Why existing tools don't solve this problem . . . 25
5.2 Test Setup . . . 27
5.3 eBPF Probe Overhead . . . 27
5.4 Hooking Malloc via the Procedure Linkage Table (PLT) . . . 28
5.5 Mem-hook implementation . . . 29
5.6 How to get function name from backtraces . . . 35

6 Evaluation **36**
6.1 Evaluation method . . . 36
6.2 Results . . . 36
6.3 Summary of results . . . 36
6.4 Dumping all tests here . . . 36

Bibliography **38**

A Appendix **43**
A.1 Benchmark Results . . . 43
A.2 Test Applications . . . 47

List of Figures

2.1 Overview of the eBPF ecosystem. 10

5.1 Disassembly of the PLT entry for `malloc` using OBJDUMP. 29
5.2 A memory space output example for a running process. 29
5.3 A minimal C++ library example for hooking malloc 30
5.4 A minimal example that showcases loading a dynamic library with GDB. 30
5.5 A function address retrieval example with GDB 31
5.6 An example showing the `malloc` PLT entry modified with GDB 31
5.7 The code for the malloc hook implementation. 33

6.1 Allocation time for various backtrace methods and depths. 37

List of Tables

2.1 Performance measurements of ptrace based tools and ad hoc kernel modules. . . . 8

3.1 A summary of the important properties of different related techniques and tools. . 13

4.1 Search Terms Used Across Databases 21

5.1 Measured overhead of eBPF probes on malloc for different allocation sizes. 28
5.2 Measured overhead of eBPF probes in *Mimir*. 28
5.3 Measured overhead hooking malloc with PLT (in nanoseconds). 31

6.1 Measured allocation time of MEMHOOK with different backtrace methods and depths. 37
6.2 Tests done on realistic.cpp with different profilers 37

1 Introduction

Efficient memory management has been a critical issue in computer science for a long time and has direct influence over the performance, reliability, and security of the application. Ensuring correct memory management is especially important for languages like C++ where developers have direct control over the memory allocation strategy. Although this control can increase flexibility and performance, it also introduces significant risks. Mistakes in handling memory, such as forgetting to free memory or accessing deallocated regions, can lead to memory leaks and undefined behaviour. These issues do not only lead to poorer performance, but can also cause exploits and crashes.

1.1 Motivation

Several tools and frameworks already exist to trace dynamic memory allocations in C++ applications [1, 2, 3, 4, 5]. These tools and frameworks have a few issues that prevent them from running in real-time on resource-limited systems, such as Ericssons radio access network. They are either too resource-intensive, cannot be executed during run-time, or lack support for the target hardware. For example, tools such as Valgrind can introduce significant performance overhead, making them less ideal for real-time applications [1]. A resource-limited system is a computing environment with strict hardware or software limitations. In the context of this thesis, it is a system constrained by computing power and memory.

Writing safe C/C++ code is generally challenging. Incorrect memory usage is often a source of degrading system performance and efficiency [6]. Persistent programs such as servers are also significantly more impacted since short lived programs frequently free all resources. Long living application are more vulnerable to subtle memory leaks, that takes days or weeks to manifest as they are difficult to spot during the development and testing phase [7]. It can also introduce different kinds of memory-related bugs, e.g. *user-after-free* or *double-free*, that can become an exploitable security issue [8]. Memory profiling tools with low performance overhead are important for systems that require high availability and efficiency [9]. In environments with constrained resources, excessive overhead can reduce system capacity, degrade performance, and increase hardware requirements. This is particularly important for critical real-time applications, such as telecommunication systems. In the case of radio access networks, which form part of critical infrastructure, taking the system offline for

profiling is not feasible. Instead, profiling has to run seamlessly while the system is operating, without impacting its performance and throughput.

1.2 Aim

The purpose of this thesis is to design a dynamic memory profiling tool. This tool should be usable with already running code, a requirement that constraints implementation possibilities. The main goal of this tool is a low CPU overhead. Achieving 20% additional load on the CPU was the initial target. The tool should not make any guarantee of finding *all* instances of leaked memory. It should detect if there *is* a memory leak and provide statistics about allocations and deallocations to help the developer find the leak.

1.3 Research questions

1. What are the options for developing a low performance memory tool for continuously running processes?
2. What is the overhead of these tools?
3. Which problems can such tools detect?
4. In what areas are such tools useful?
   Last question seems vague, i vote we stick with first 3

1.4 Approach

The research consists of a literature study followed by the development of a prototype. The study includes comparisons between different established profiling tools. Performance and overhead requirements are evaluated according to the performance goals. It is then investigated how these tools can be modified and applied to meet the specific goals.

The latter part consists of prototype development. It is designed to help address the research questions. The main goal of creating the prototype is to see if the desired software is a feasible project. It will be used to collect data about its performance impact and explore what features are possible.

This paper will evaluate the performance impact of running a profiling tool in a live system with limited resources. It will also provide insight into how such a profiling tool can be designed. The performance will be evaluated using benchmarks supplied by Ericsson.

The developed custom profiler works by overwriting the addresses of the functions that perform allocations and deallocations (`malloc`, `free`, `new`, `new[]`, `new no throw`, `delete`, `delete[]`) with hooks that collect meta data about the allocations and deallocations. The addresses are overwritten in the PLT table. This means that the software to profile requires no changes and already running processes can be profiled without the need to restart or stop them.

1.4.1 Delimitation

This thesis will focus on Ericsson's specific use case and the prototype will be written to work with the architecture that Ericsson uses: the intel family of processors. Developing a modular profiling tool that can run on multiple architectures such as ARM and the AMD family could require significant more work. Despite this, this thesis aims to provide insight into the techniques and methods used such that future work can be done and the tool can be expanded upon.

2 Background

In the following section, this paper will present background work, outlining foundational studies and relevant prior research.

2.1 Types of Memory Leaks

Since the concept of a memory leak can be interpreted in different ways, the following section clarifies how it is defined and understood in the context of this thesis.

There are two types of memory leaks: *unreachable* and *dead* memory [3]. *Unreachable* memory is memory that the program can no longer access. This class of memory leaks can be addressed by methods such as garbage collection [10], statistical analysis [11, 12, 13, 14, 15], and model checking [4]. *Dead* memory is memory the program has access to but will never use again. Deciding whether a chunk of memory will be accessed in the future is, in the general case, an undecidable problem. This paper will only deal with *unreachable* memory and will refer to it as memory leaks or leaked memory.

2.2 History and Impacts of Memory Issues

The following provides a brief history and overview of the impact of memory leaks, highlighting why they are important to address.

Van der Veen et al. [16] have done a thorough evaluation of the history and impacts of memory issues, which are among the oldest challenges in the field of computer science. A fortitude of different approaches have been proposed, such as safe languages [17] [18], bounds checkers [19] [16] [20], and other types of countermeasures [21] [22] [23] to prevent memory exploits. However, the authors mean that these efforts have not been enough. Despite these extensive attempts, memory errors keep undermining system security. Many of the exploits affecting systems today are direct consequences of unsafe memory [24] [25]. The opinions in the field appear to differ quite drastically. Some suggest that it is a solved problem while others propose to stop funding research altogether since the problem is not solvable. Although this may be true, the author's research shows that memory issues are still an immense risk to modern systems [26] [27]. They reason that it is an important problem to solve for both the industry and society as a whole. To reflect this, different industry actors have announced cash prizes to researchers who improve the safety of memory handling [28].

2.3 Stack Frames

Collecting backtraces is a key feature of the developed profiler. This section provides the necessary background to understand its relevance.

Procedures are reusable blocks of code that implement some logic with a set of arguments and an optional return value. They can be invoked from different parts of the program and can take many forms, such as functions, methods, subroutines, handlers etc. Procedures must support three key mechanisms: passing control, passing data, and allocating and deallocating memory. [29]

2.3.1 The Run-Tim Stack

A powerful feature of the procedure-calling mechanism of C is its ability to use the stack data structure to pass data to new procedures. Assume there is a procedure `P` making a call to the procedure `Q`. While `Q` is executing, `Q` is responsible for memory allocation for its local variables and any procedure calls. This means that `P` can freely be suspended from executing. Thus, each procedures storage can be managed using a single stack, where arguments and a potential return value is passed through the registers. [29]

x86-64 supports passing up to six integral (pointer and integer) arguments through registers. If an argument cannot be passed in registers, such as when there are seven or more integral arguments, or when a local variable's address is needed (e.g using the &-operator), it must passed in the procedure's *stack frame*. Assume procedure `P` passes seven integrals as arguments to `Q`, the seventh integral value will be placed in the stack frame of `Q`. [29]

The stack frame contains all information about the procedure. This includes arguments, local variables, the return address, and the return value (unless it is passed by registers). To allocate memory for a new stack frame, the stack pointer is increased. To instead free memory from the previous stack, the pointer is decremented. A procedure can allocate more memory by incrementing the stack pointer and can similarly free memory by decrementing it. [29]

2.3.2 Stack Frames Back Traces

Malloc and *free* are two examples of procedures, one with a return value (malloc) and one without. By overriding *malloc* and *free*, one can use the stack frames and their return addresses to walk back to previous frames, gathering information about the allocation such as allocation size and what procedure allocated it.

The header file `execinfo.h` [30] from glibc contains three functions that manipulate and gather backtraces of the current thread. The first function `backtrace` has two arguments, `buffer` and `size`. It obtains a backtrace of the current thread as a list of pointers and places them into `buffer`. It places at most `size` number of pointers into buffer and returns the actual number of pointers places into `buffer`. The pointers in `buffer` represent return addresses that are extracted by inspecting the stack, with one return address corresponding to each stack frame. Note that some compile optimizations such as inlining a function will make `backtrace` interpret the stack content incorrectly.

The second function `backtrace_symbols` translates the information gained from `backtrace` into an array of strings. Each string represents a printable element from the backtrace. The last function `backtrace_symbols_fd` also translates the information from `backtrace`; it writes each string to a file descriptor, one line per string.

There are two specific compiler flags that deal with stack frame pointers, `-fno-omit-frame-pointer` and `-mno-omit-leaf-frame-pointer`. They specify that the compiler should *keep* the stack pointer from functions that don't need it [31]. When optimizing compilers may omit the stack pointer to free a register that can be used for other values. According to the fedora project [32] the improved effectiveness of profiling and debugging tools is worth the performance cost. In their wiki they summarized a few bench-

marks when they compared packages built with frame pointers against the same packages built without them.

- Compiling the kernel with GCC is 2.4% slower.
- Running Blender to render a specific frame is 2% slower.
- openssl/botan/zstd was not significantly affected.
- CPython benchmarks ran anywhere from 1-10% slower depending on the benchmark.
- Redis benchmarks was not significantly impacted.

The wiki post links to a specific comment that discusses the Python benchmark performance [33]. The reason for the big increase was due to a large chunk of the CPU time was spent in _PyEval_EvalFrameDefault. When the user looked at the implementation in the C code they found that it was a humongous function with a gigantic switch statement that implements Python instruction handling logic. The function also has a lot of local states in different branches.

The user claims that this explains the slowdown, since _PyEval_EvalFrameDefault already took up a large CPU chunk and now needs to spend extra time to load values from the stack that was previously saved in the extra register. The user also claims that it is not representative of how software in the real world looks nor is it a good idea to write code this way.

2.4 User and Kernel Space in Linux

This section provides the necessary background to understand why the cost of certain tools makes them unsuitable for our target.

In operating systems, the terms *kernel* and *user* are commonly used to describe different privilege levels and execution contexts. At a high level, the kernel refers to the core part of the operating system that runs with high privileges, while user space typically refers to applications running with restricted access and lower privileges. However, these terms can carry more specific meanings depending on the context.

User mode and kernel mode often describe the CPU's execution modes. Code running in kernel mode has full control over the system, including hardware and memory, while code in user mode operates with limited permissions. For example, enabling or disabling local CPU interrupts can only be done in kernel mode. If a user-mode program tries to perform such an operation, it triggers an exception, and the kernel steps in to handle it.

User space and kernel space are often used when talking about memory protection or virtual memory. In simplified terms, kernel space is a protected region of memory reserved for the kernel, while user space is the memory assigned to a specific user process. User applications cannot directly access kernel space for security and stability reasons, but code running in kernel mode can access both spaces. [34]

Switching from user to kernel space requires a context switch, which is expensive [35].

2.5 Dynamic Libraries in GNU/Linux

The developed profiler is partly implemented as a library. This section provides the necessary context to understand its design choices.

Libraries are designed to encapsulate related functionality into a single unit, making it easier to share code with other developers. This approach led to modular programming, where programs are built from reusable modules. In Linux, there are two types of libraries, dynamic and static. Static libraries are linked to a program at compile time, embedding their

functionality directly into the executable. In contrast, dynamic libraries are loaded when the application starts, with binding occurring at runtime. [36]

2.5.1 Dynamically link

In GNU/Linux there are two ways to deal with dynamic libraries. The first option is to *dynamically link* the program with the shared libraries. When an application starts a Executable and Linking Format (ELF) image is invoked. The kernel loads this image into user space virtual memory and looks at the `.interp` section which indicates which dynamic linker to be used. The kernel then bootstraps the linker which initializes itself. The dynamic linker (ELF interpreter) loads the shared objects and performs the relocation. Once the linker is done, control is transferred back to the program.

The relocation uses a Global Offset Table (GOT) and a Procedure Linkage Table (PLT). These provides the addresses of functions and data which is used during the relocation process. The code that uses these tables does not change, the contents in these tables change. This relocation can also happen when a function is needed, this is called *dynamic loading*.

Once the relocation is complete, the linker allows shared objects to execute initialization code. This code is defined in the `.init` section of an ELF image and allows libraries to initialize data and prepare for use. When a library is unloaded it may call a termination function defined in the `.fini` section of the ELF image. [36]

2.5.2 Dynamically load

Instead of Linux loading and linking libraries it is possible to let the program do it. This process is called *dynamic loading*. The application specifies a shared library to load and can use it as an executable after. The Dynamic Loading (DL) API provides the necessary functions. These four are: `dlopen`, `dlsym`, `dlerror`, and `dlclose`.

2.5.3 LD_PRELOAD

`ld_preload` [37] is an environment variable on Unix-like system such as Linux that allows the user to specify ELF shared objects to be loaded before all other objects (such as the standard C library). This feature is useful for overriding functions in other shared objects. Several objects can be specified and they are loaded in the left-to-right order specified in the argument. The objects are added to the file `/etc/ld.so.preload` that specifies what objects to load *before* the program.

This means that `ld_preload` can only be used for overriding functions in shared library *before* starting the process, and not while the process is running.

2.6 objdump

The profiler utilizes `objdump`. This section explains the functionality of `objdump`.

`objdump` is a command-line tool that displays information about one or more object files. It is part of GNU Binutils and provides features such as disassembling binaries [38].

`objdump` can be used to get function offsets from a binary as well as inspect different section such as the `.plt` section.

2.7 Perf

Perf [39] is a Linux profiler that provides a common interface to the CPU performance counters as well as tracepoints, kprobes, and uprobes (described further under 2.8.2).

- **Hardware event:** The hardware events uses the CPU:s Performance Monitoring Unit (PMU) and can provide the user with information such as number of retried instructions, number of clock cycles, cache misses and so on.
- **Hardware cache event:** The hardware cache event also uses the CPU:s PMU and can provide the user with more detailed cache related information: stores, loads, prefetch and store misses as well as load misses.
- **Tracepoints:** Tracepoints are instrumentation points strategically placed in code to monitor events like system calls, TCP/IP activities, and file system operations. They have minimal overhead when inactive and can be enabled using the perf command to gather data such as timestamps and stack traces. Additionally, perf can dynamically create tracepoints using the kprobes and uprobes frameworks, enabling dynamic tracing in the kernel and userspace.

If the user monitors more events than there are counters, Perf will use time multiplexing. Perf will then scale the result to estimate the final counting as if it did not multiplex with the following formula: $final\_count = raw\_count \times time\_enabled / time\_running$

Users can specify whether they want to sample events in user space, kernel space, or both. The user can specify if the sampling should occur per-thread, per-process, per-cpu or system wide. Per-thread is the default setting.

Perf has a long list of command that allows the user to collect the desired data. One of these commands `perf annotate` allows the user to sample the instructions in a given function. Compiling the program with flags `-ggdb` allows perf to display source code next to the assembly code

2.8 Lightweight Dynamic Tracing of User Apps

There exists several techniques for lightweight dynamic tracing of user applications. Some of them are explained below.

2.8.1 Gdb and Ptrace

`gdb` is an application debugger that runs on Linux and other UNIX systems. It is an interactive debugger but can also be used with a set of commands requiring no human interaction during the debugging process. Gdb uses ptrace requests and has *significant* performance overhead. There are linux commands that uses `ptrace` such as `strace` and `ltrace`. `strace` exposes ptrace's `ptrace_syscall`, which allows ptrace to continue executing until the next entry or exit from a system call. `strace` has *significantly less* performance overhead but it is still its number one-complaint. `ltrace` is another command that uses `ptrace`. It traces calls to dynamically linked library functions. `ltrace` also suffers from performance overhead inherent in ptrace-based tools. One of the reasons for this is the overhead of accessing the tracee's memory. The overhead is on the order 10x to 100x, compared with equivalent in-kernel access. [40]

2.8.2 Kernel-Based Tracing for Memory Profiling

Kprobes

Kernel probes [41] allow the user to dynamically insert "probes" into the kernel for purposes such as debugging, tracing, fault injection etc. These probes essentially "probes" into the execution of the kernel at runtime to observe its behaviour without modifying the code. The user can specify handlers that runs a user specified routine before/after the probed instruction. The user can also specify a fault handler that runs if a fault happens at the probed instruction

or in one of the pre_/post_handlers. The handlers all run in kernel mode allowing the users to instrument in kernel mode.

Utrace

To increase separation and layering of code between the architecture-specific and architecture agnostic parts, `utrace` [40] was implemented. It was designed as an abstraction layer used for writing debugging and tracing applications. The already existing solution, ptrace, had some limitations that utrace aimed to mitigate. These limitations include: Not being a POSIX system call, overall overhead, and difficulty to work with.

Utrace provided three main functions: It reported events of interest, it provided thread control, and it provided access to the threads state. Utrace did this by placing tracepoints in the kernel code that execute callbacks when reached. These callbacks executed when a process in user-space was executing. Since they were placed in kernel code they execute in kernel mode and thus, utrace clients executed in kernel mode. [40]

UTRACE is deprecated and therefore not an option [42].

Uprobes

User-space probes [40] works similar to kprobes with the difference that they insert probes into user-space applications. Uprobes use the same API as kprobes, since its infrastructure lives in the kernel. This allows a single instance of instrumentation software to collect all information that could be useful when tracing user-space applications, since it can trace user-space application in kernel space. Users write a kernel module specifying what virtual address and process to probe, as well as the handler to run when the probepoint is activated. One potential draw back is the inability to probe a specific thread in a multithreaded software. This is due to how uprobes are designed.

SystemTap

SystemTap [40] can be seen as a wrapper for kprobes, allowing users to use the kernel and other instrumentation tools through a scripting language. SystemTap passes this data to user-space where its post-processing infrastructure presents and formats the data.

Tool	Event Counted	Overhead (usec) per Event
ltrace -c	function calls	22
gdb -batch	function calls	265
strace -c	system calls	25
kprobes	function calls	0.25
uprobes	function calls	3.4
utrace	system calls	0.16

Table 2.1: Performance measurements of ptrace based tools and ad hoc kernel modules.

2.8.3 Advantages and Disadvantages of Kernel-Based Tracing

Kernel-based tracing is fast, as shown by Keniston et. al [40] in figure 2.1. They measured ptrace based tools (bottom) against ad hoc kernel modules (top). Kernel-based tracing also provides a way for users to trace issues through user-space all the way into specific kernel modules. Kernel code also runs at the highest privilege and can access all process space by the user applications. Thus all relevant information for the user application, including kernel data structures, third party library etc, can be accessed and collected.

Since the kernel runs at a higher privilege level assumptions that are valid for a user-space program might not be valid for a kernel-space programs. Writing kernel level code is also difficult and its unreasonable to expect everyone to be able to write their own kernel module. The information the kernel can access from a user-space program might also not include all information needed to decode it (debug information and symbol table are two such examples). Unless this information is provided the kernel must rely on post-processing in the user-space.

SystemTap [40] uses the running kernels debug information to mitigate some of these drawbacks. It can determine source file, line number and location of variables.

Kernel-Based Tracing for memory profiling

Kernel-based tracing provides powerful tools for analyzing and debugging both kernel and user-space activities. Tools such as kprobes, uprobes, and SystemTap allow users to insert probes into the kernel or user-space applications, enabling detailed instrumentation for tracing memory-related events.

Despite these advantages, kernel-based tracing has limitations. Writing kernel-level code is *difficult*, and debugging tools like uprobes may face challenges in multithreaded environments due to its inablility to probe specific threads. Additionally, tracing in kernel mode involves higher privileges, which may introduce additional risks and complexity.

2.9 eBPF

Extended Berkely Packet Filter (eBPF) is a tool that allows efficient sandboxed execution of programs within the Linux kernel. It was originally designed for packet filtering in networking applications but has evolved into a framework for modifying the kernel without requiring kernel modifications or recompilation. eBPF programs run in a restricted virtual machine within the kernel and can be hooked onto different events, such as system calls, tracepoints, and hardware counters.

One of the advantages of eBPF is its ability to execute custom logic with minimal overhead while maintaining safety and stability. The Linux kernel verifies the program before execution and enforces various constraints such as bounded loops and memory safety. This is intended to prevent crashes and security exploits in kernel space. Furthermore, eBPF can interact with user-space applications through *maps*, which allows for different types of data collection. [43]

2.9.1 Technical Overview of eBPF

The following technical explanation is based on the work of Gbadamosi et al. [43]. To understand how eBPF can be used for memory profiling, it is important to examine its underlying architecture. Unlike traditional kernel modules or instrumentation tools, eBPF provides a dynamic, safe, high-performance mechanism for extending kernel functionality in real-time. By executing within a virtual machine, eBPF allows user-defined programs to be executed securely within the kernel without requiring recompilation or system restarts.

eBPF Execution Model

The eBPF execution model consists of several key components:

- **eBPF Bytecode:** eBPF programs are written in a high-level programming language, usually C, and compiled into eBPF bytecode. The virtual machine executes the bytecode as an *eBPF program*. Each program is made up of *subprograms*, which are self contained units of bytecode that can be compared to *functions*.

- **eBPF Maps:** eBPF maps are abstract data structures, usually an array or hash map, that allow eBPF programs to store and exchange data between kernel and user space.
- **User-space Loader:** eBPF bytecode is loaded into the kernel with the help of *user-space loaders*, such as BCC [44], bpftrace [45], or libbpf [46]. The bytecode is loaded through the *BPF_PROG_LOAD* system call and attached to relevant hook points. The user-space loader also manages the corresponding *maps* for the program.
- **Verifier:** Before execution, the eBPF verifier ensures that the program is safe by checking for out-of-bounds memory access, infinite loops, and other potential security risks.
- **JIT Compilation:** After verification, the bytecode is *just-in-time* (JIT) compiled into native machine instructions to optimize performance. If compilation is not supported, the bytecode is dynamically interpreted instead.
- **eBPF Hooks:** eBPF programs are attached to predefined hooks and are executed when they are triggered in the kernel. Hooks can be attached to different places in the kernel, such as system calls, functions, sockets, tracepoints, and more. Custom hook points can also be created to extend the functionality, called kernel probes (kprobes) or user probes (uprobes).

Types of eBPF Programs

eBPF supports multiple program types, each optimized for different use cases:

- **Tracing Programs:** Attach to system calls, kernel functions, or user-space applications for profiling and debugging.
- **Networking Programs:** Used in packet filtering, load balancing, and security monitoring.
- **Security Programs:** Restrict program behaviour by enforcing custom security policies.

The eBPF Lifecycle

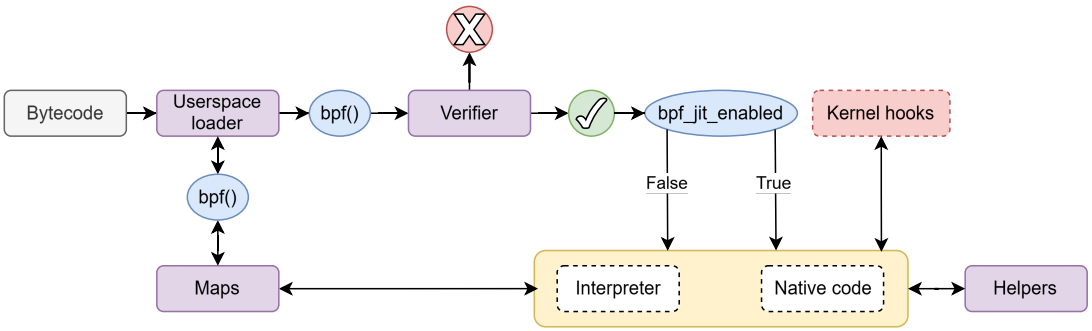


Figure 2.1: Overview of the eBPF ecosystem.

Once an eBPF program is loaded, the kernel takes over its management and exposes the program to user-space through a *file descriptor*. When the last file descriptor referencing an eBPF object is closed, the kernel automatically cleans up its resources. To extend an eBPF object's lifetime beyond the process that created it, the kernel provides a mechanism called *bpffs* (BPF file system). Pinning an eBPF object to bpffs ensures that it remains accessable even after the process terminates, allowing other processes to keep using the object.

eBPF links provide further flexibility in managing lifecycles and program attachments. Instead of directly attaching an eBPF program to a kernel hook, an eBPF link ties the program

to a file descriptor. This simplifies resource management and ensures that an eBPF probe remains active even after the process that created it terminates. Only the link owner can detach or modify the link, ensuring system integrity and preventing unintended modifications.

2.9.2 eBPF for Memory Profiling

Even though eBPF is mostly used for networking and security monitoring, it has memory profiling capabilities. By attaching eBPF probes to memory allocation functions such as *malloc* and *free*, memory usage can be tracked in real time. In contrast to traditional memory profiling tools, which often require recompilation or introduce significant run-time overhead, eBPF provides a more lightweight approach without the need for recompilation. [47]

For programs running in production, such as Ericsson's radio access network, eBPF offers a great alternative. Overriding *malloc* and *free* introduces substantial performance overhead due to *function interposition*; a technique that allows for interception and modification of calls to existing functions within a program [48]. In contrast, eBPF can trace memory allocations in kernel space with substantially lower overhead [47]. Theoretically, it could allow continuous memory monitoring without impacting system performance beyond an acceptable level.

2.9.3 Performance Impacts of eBPF

Given the strict performance requirements of radio access networks, any memory profiling tool must introduce minimal CPU and memory overhead. eBPF could be well-suited for this task, given its ability to execute in kernel space with *just-in-time* (JIT) compilation [43]. For instance, Lian et al. [49] showed that eBPF can achieve 98.5% reduction in performance overhead for *working set estimation*. Working set estimation tries to approximate the amount of memory a process need during a specific period of time. Even though they are not tracking memory leaks, it shows that general memory profiling could be greatly improved with this technique.

2.9.4 eBPF Tools for Memory Analysis

Memleak

Memleak is a utility within the *BPF Compiler Collection* (BCC) that uses eBPF technology to monitor and identify memory leaks in real time on Linux systems. By attaching to running processes, *memleak* tracks memory allocations and deallocations, providing insights into potential leaks without code modifications or restarts. It can attach to both user-space and kernel functions, offering flexibility in monitoring different system components [44]. Users can also specify different criteria, such as allocation range sizes to focus on. The tool captures stack traces associated with memory allocation to find the source of potential memory leaks [50].

While memleak is powerful, it can introduce substantial overhead, especially when monitoring processes with high allocation rates. In extreme cases, performance overhead of up to 100 times has been measured. Several key factors contribute to this degradation. Firstly, it hooks onto every *malloc* and *free* call, which causes frequent user-kernel transitions and increases CPU overhead. It also captures full stack traces for every allocation, which is expensive in high-frequency scenarios. Furthermore, the data collected by memleak is frequently sent to user space, causing performance overhead due to excessive syscalls and context switches. Finally, every allocation is stored in an *eBPF hash table*, and every *free* requires a lookup, leading to further performance bottlenecks. [50]

bpftime

Zheng et. al [51] introduced bpftime, a novel user-space eBPF runtime. They implemented bpftime in user-space, and achieved 10x speed enhancements compared to the kernel coun-

terparts which requires additional context switches. Bpftime supports user-space probing and syscall hook capabilities and is compatible with exisiting eBPF toolchains. The authors claim that due to the context-switches made when using UPROBES, there is a *significant* overhead making it unsuitable for real-time monitoring in latency-sensitive applications.

However, since the project is under active development the current state of bpftime does not compile. The code base defines a function `int ioctl (int filedes, int command, ...)`, this function also exists in one of the included header files and thus a duplicate function declaration error is generated when compiled.

The project does have tags for verions 0.0.3, 0.0.2, 0.0.1 neither which compiles on any system the authors have access to.

bpftrace

Bpftrace is a high-level tracing tool that simplifies the development of eBPF programs for system monitoring. Development is done through a custom scripting language which minimizes boilerplate code. Unlike other eBPF tools that require low-level C or LLVM knowledge, bpftrace abstracts away many of these complexities, making system tracing accessible to developers and administrators. It uses a high-level syntax, inspired by *awk* [52] and *DTrace* [53], and supports both kernel and user-space tracing. It has great flexibility and allows fast prototyping of one-liner scripts without requiring a precompiled tool. However, bpftrace is not ideal for continuous monitoring during production, since its overhead can be higher than optimized eBPF programs written in C with libbpf. [54]

2.10 LLVM

One of the related works uses LLVM. This section aims to provide enough information to understand why it can be used to develop a profiler.

The LLVM project [55] is a collection of sub-projects consisting of modular and reusable compiler and toolchain technologies. LLVM is also an acronym for *low level virtual machine*, however the LLVM project is not an acronym, LLVM is the full name of the project.

One of the primary sub-projects is the LLVM Core project. The library is built around a well specified code representation called LLVM IR and provides source- and target-independent optimizer and code generation support for modern CPUs. It is easy to use the LLVM Core libraries to invent or port a language with LLVM as the optimizer and code generator. [56]

3 Related Work

In the following section, this paper will provide an overview of related work, discussing prior research and studies relevant to the topic. Below a table is provided to show a short overview of the related tools and techniques. The use case this paper aims to solve is a low overhead with no need to restart or recompile. A question mark means that a answer can not exist (e.g. managable overhead for Smoke, which is not a dynamic profiler and thus does not add overhead to the running process) or that we do not know. In order for an existing tool or technique to solve this issue the answers must be: Yes, No, No.

Table 3.1: A summary of the important properties of different related techniques and tools.

Technique/Tool	Managable overhead	Rebuild needed	Restart needed
Memwatch	No	Yes	No
Dynamic Tainting	No	?	?
DBA/DBI	No	?	Yes
Dynamic Instrumentation	No	?	?
SWAT	No	?	?
Sniper	Yes	?	?
Smoke	?	-	-
Promt	?	Yes	?
AddressSanitizer	No	?	Yes
Mesh	No	No	?
Snapshot	Yes	?	Yes
AddressWatcher	No	Yes	Yes

3.1 Memwatch

Memwatch is an internally developed tool at Ericsson that overrides malloc/free and new/delete to store a context of meta data at every allocation. It does this by overriding their implementations and thus requires to be compiled with the program. Memwatch records a backtrace in order to analyze where the allocation appear. The usage of Memwatch in live system is limited by the overhead of tracing every allocation.

3.2 Dynamic Tainting

Dynamic tainting [57] is a technique used to track information and control flow in a program. This is done by setting one or more marks on data (such as variables, function inputs, etc) and letting these marks propagate with the data as the program executes. Dynamic tainting has been successfully used in the security domain for preventing several different attacks [58, 59, 60, 19]. Clause and Orso [2] used this technique to develop their prototype called LEAKPOINT. Their evaluation of LEAKPOINT shows that this technique can detect memory leaks to the same ability as existing tools, reports zero false positives, and is also effective at helping developers solve the memory issue. The overhead of LEAKPOINT is 100-300 times.

Dynamic Tainting provides several advantages. It detects leaks as good as other techniques, helps developers solve the issue, and reports zero false positives. However the overhead of 100-300 times in LEAKPOINT is too big of a cost for a tool to be run in production.

3.3 Dynamic Binary Analysis and Binary Instrumentation

Memcheck [61] is a *dynamic binary analysis* (DBA) tool. It is implemented with Valgrind [1], a *dynamic binary instrumentation* (DBI) tool. Memcheck analyses the program during run-time and provides the user with four main features. It detects accesses to unaddressable memory, it detects memory leaks, it checks for overlapping memory in function call to functions such as `memcpy()` and `strcpy()`, and lastly, it detects undefined value errors.

Memcheck belongs to the group of DBA that uses *shadow values* [1]. They shadow every register and memory value in software. These shadows store values that state something about the value. On average Memcheck reduces a program's performance 20-30 times [61].

Other DBA's that are implemented using Valgrind include TAINTCHECK [59]. It tracks tainted bytes and usage of those bytes.

The slowdown of Memcheck is partly due to the cost of running Memchecks checking, but also partly due to the overhead of Valgrind. TAINTCHECK have a similar cost due to the same reasons. Newsome and Song [59] mentions in their comparison of their tests that running the programs without Valgrind is fastest. Using Nulgrind [1] (Valgrind with no analysis code) slows down the programs between 2 and 13 times, and using TAINCHECK slows down the program even more.

Running any form of profiler with Valgrind is not feasable due to the inherit overhead of Valgrind itself. Newsome and Song showed that Nullgrinds minimal performance overhead in their tests was a factor of 2, which is already too high.

On top of this VALGRINDS manual suggests to recompile the program without optimization if the user intends to use the default tool MEMCHECK. MEMCHECK can on rare occasions wrongly report uninitialized value error or missing uninitialized value errors. Fixing this issue would make the tool even slower and thus the best solution is to run the program without optimizations [62].

3.4 Dynamic Instrumentation

Dynamic instrumentation is a technique used to collect data from programs by modifying the program itself. Program instrumentation works by inserting instrumentation code at specific points of interest within a program. During execution, this code runs alongside the original program allowing the collection of data [63].

3.4.1 Frida

Frida describes itself as a dynamic instrumentation toolkit for developers, reverse-engineers, and security researchers. It's core is written in C and allows the user to inject QuickJS, a small

and embeddable Javascript engine into a target process. The user can then utilize hooking functions to intercept function calls in the attached process. Frida has a C API that provides the same functionality as its JavaScript counterpart. Frida is made up of several modules where frida-gum allows the user to augment and replace functions using C. This paper does not consider frida an option due to the lack of documentation on how to use frida-gum and other frida modules. [64]

3.4.2 DIOTA

Maebe et al. describe DIOTA [63], a method for instrumenting binaries. DIOTA works by keeping the original program intact and generate code on the fly elsewhere. The generated code will still use the original program for data accesses and the generated code (clone) will be used to instrument the code.

DIOTA disassembles code once instrumentation has started until a branch is encountered. If it can follow the destination (immediate target addresses) it does, otherwise the code is evaluated when it is executed. Once a criteria to stop instrumenting is met a trampoline is inserted in the generated code. This trampoline pushes the address of the next instruction onto the stack and then returns control to DIOTA. DIOTA restores the stack pointer and jumps to the generated code. If DIOTA is reactivated through a trampoline it first checks if it has already instrumented the target address. If not it generates a new block of code.

Since DIOTA works by generating new code and not altering the existing code no relocation has to be done. When a subroutine call is encountered the return address is pushed onto the stack. This is necessary since some functionality such as backtraces relies on this information. The return is handled with a variable destination. One difference from how branches are handled is that it is just replaced in the generated code to a jump to DIOTA. This is done since the target address is already at the top of the stack.

DIOTA provides several execution modes, most of which can be combined. The relevant for this paper are mentioned below.

Memory instrumentation

All instructions that access memory are preceded by at least one call to a user defined routine. These routines have access to the type of memory access (load, store, modify) and its address and size of the memory location.

Interception of function calls

DIOTA allows interception of routines in dynamically linked libraries. This does however require the library and/or program to contain symbol information.

Performance

The authors claim that DIOTA achieves a slowdown of 20-400% on average. This slowdown increases when memory instrumentation is activated, since every instruction that accesses memory is accompanied by at least one callback.

Although the authors does not give any concrete numbers of the slowdown when using the relevant modes of execution the base slowdown of 20-400% is in *best case* on the verge of being too high. For this to be feasible the added performance cost of *anything* related to gathering information about the memory usage of the program would have to be essentially zero, which is not realistic.

3.5 SWAT

Chilimbi and Hauswirth [9] based their memory leak detection model on the simple idea that *heap objects not accessed for a 'long' time are memory leaks*. This ensures that their software detects all leaks during a given execution run. They named their memory leak detection tool SWAT. The authors describe SWAT as an implementation of "an adaptive profiling scheme that addresses this by sampling execution of code segments at a rate inversely proportional to their execution frequency".

SWAT has a low performance overhead (<10%), low memory overhead (<10%) and low rate of false positives (<10%). SWAT can also provide information about where the heap objects were leaked. Since SWAT operates by sampling access sites and allocation size through the CPUs instructions, it can provide the user with this data.

To sample code segments Chilimbi and Hauswirth duplicate the code of each procedure and only instrument one of these. For each procedure they only sample when the produces own sample counter reaches zero. They also state that for infrequent program events such as dynamic heap allocations they always instrument.

For Ericssons use case dynamic heap allocations are most likely not an infrequent program event and always instrumenting dynamic memory allocations would yield to big of an overhead, since this is what their previous implementation MEMWATCH did. It is also unclear whether this technique requires restarts and stops or recompilations of the target software, something that is not acceptable for this use case.

3.6 Sniper

Jung et al. presents SNIPER [3], an automated memory leak detection tool for C/C++ programs. SNIPER builds on the idea that *objects not accessed for a long time are memory leaks*. It uses a combination of performance monitoring units (PMU) and traces of malloc and free to reconstruct the accessing patterns of memory object after program execution. SNIPER achieves a low overhead of less than 3% on average while robust against false positives.

SNIPER works on the assumption that *the staleness* of a leaking object is much higher than a normal object. Since objects allocated in different places in the code is expected to have different life times it must account for this. Thus SNIPER applies *local* and *global* anomaly detection. Firstly it treats each allocation site as its own and looks for anomalies within each site. Since it could be the case that *all* objects from a single site are leaks it applies global anomaly detection on sites where it found zero leaks. Global detection compares objects staleness to the staleness of all objects in the application. The idea is that if a single site only contains leaks these objects are very likely to have a staleness much higher than objects from other sites. This heuristic could end up reporting a lot of false positives in areas where an allocation is made but rarely, if ever accesses without it being a leak, such as GUI objects. Thus SNIPER focuses on objects that impact the global memory consumption. For small objects SNIPER only applies local detection. The parameter that determines what a *small* object is, is configurable. [3]

The authors do not explicitly state whether it is possible to hook into a running processes or not. However, it appears to be feasible, given that plt hooking is a technique that works. However it is not obvious if performance counters is easy to integrate. They do mention that the report is generated when the program terminates, but it is unclear whether this is a strict requirement or simply an unfortunate choice of wording.

Additionally, their focus is on memory leaks that impact the entire system, and they seem to be comfortable with not detecting smaller memory leaks. This is not true for the intended use case of the profiler this papers aims to develop.

3.7 Large scale memory detection

Fan et. al. present SMOKE [65], a static memory leak detector aimed to provide precise and scalable analysis for industrial scale project. SMOKE achieves this through a two stage process. The first stage uses an imprecise but scalable analysis. It utilizes a new type of program representation called use-flow graphs. The use-flow graph contains control-flow information and encodes object definitions and object uses of problem-relevant heap objects. It also contains control flow order of uses of the same heap objects. This makes it possible to check for memory leaks by checking properties of a finite state machine. In stage two the authors apply constraint solvers. This processes is needed since some heap memory is managed during conditions (branches) that needs to be taken into account when analysing for leaks. The authors first filter infeasible paths and then applies a SMT solver such as Z3 [66] for the remaining complex path conditions.

SMOKE was able to detect 30 previous unknown memory leaks in 29 different mature project where one of the leaks was assigned a CVE ID. It is also significantly faster with a speed up ranging from 5.2X to 22.8X.

Even though the focus of SMOKE is to find memory leaks for large industrial scale projects it is a statistical analysis method and does not solve the issue this paper aims to solve.

3.8 PROMT

Xu Ziyang et. al. introduces PROMT, a framework for developing fast memory profilers. It consists of a frontend and a backend that communicates through an event queue. The separation has three main benefits. The developers can focus on the core profiling logic, it reduces the interference of the profiled program, and it allows for multiple backends to run in parallel. PROMT has also standardized profile events. Thus profile developers only need to specify the profiling events, and implement the core profiling logic. Its instrumentation is built upon the LLVM compiler infrastructure. Most of its profiling events are implemented at the LLVM IR level by adding callback functions with the desired information. The authors ported two state-of-the-art profiler with PROMT, perserving all features and achived a 5.3 and 7.1 times speedup. [67]

Since PROMT is built upon the LLVM compiler infrastructure it needs to be compiled with the program, and linked with PROMT's runtime. Thus PROMT is not a solution for the issue this paper aims to solve.

3.9 AddressSanitizer

Ensuring memory safety in C++ has been a critical focus for researchers for a long time. ADDRESSSANITIZER (ASan) [68] is one of the most popular tools for this purpose. It provides memory safety through a concept called *red zones* and delayed memory reuse. A *red zone* is a protected memory region that helps detect memory corruption. The system places the zone adjacent to the allocated memory to detect buffer overflows or invalid memory accesses.

ASan is also able to detect memory leaks though its integrated leak detector LeakSanitizer (LSan) [69]. During execution LSan adds very little overhead. At the end of execution there it adds an extra leak detection phase. It is possible to run LSan without ASan for performance critical sections. However since it adds an extra leak detection phase at *end of execution* it does not suite our use case.

3.10 MESH: A Memory-Efficient Safe Heap for C/C++

An alternative to ASAN is SoftBound/CETS [70, 71], which offers a more robust solution. It ensures that memory is only accessed during its valid lifetime and ensures its memory

bounds. However, this tool also introduces significant memory overhead, making it infeasible for constrained systems [71].

Vintila, Zieris, and Horsch [72] propose a solution that provides memory safety with minimal memory overhead. It stores object bounds and validity in a compact *MESH-table*. By storing tags in redundant bits in 64-bit pointers, each pointer links to its associated metadata. They are then validated during runtime to ensure safe memory usage. One of its key strengths is its low memory overhead, which can be as small as 2 MB. Unlike ASan and SoftBound/CETS, it avoids a large *red zone*, making it ideal for resource-limited systems. It is also fully compatible with unmodified C++ code, enabling seamless integration with existing systems.

The limitation of this approach is memory leaks cannot be identified on their own. Only incorrect usage of heap-allocated memory is detected, such as *use-after-free* or *double-free* bugs. This severely limits the applicability of this method, since it does not identify memory leaks, it only protects against incorrect memory usage. Furthermore, even though the performance overhead was shown to be around 5% for some real-world applications, it increased to 170% during CPU benchmarking. This performance impact is unacceptable for highly critical systems, such as *radio access networks*.

3.11 Snapshot

Azhari et. al. proposes a tool [7] to detect memory leaks based on the growth of memory blocks. Their tool uses `ld-preload` to override `malloc`, `calloc`, `realloc`, and `free` before the profiled software starts. These functions track and update the statistics of memory consumption and store that information in a shared memory block available from a shell CLI, called a snapshot. The tool creates one heap walker at the end of execution. The heap walker uses information such as the size of allocations, age of the blocks, call-stack id and process/thread id, all of which is dumped into a file at the end of execution. This information is then analysed in the leak detection stage.

When analysing for leaks, backtraces are used to group allocations together and the allocations are then visualized to help aid developers in finding the leak. The visualization consolidates different call-stacks modelling them as call chains that merge equivalent entries into a single point. This creates a network flow graph with the amount of flow corresponding to the amount of allocated memory at a particular call site.

The tool has a CPU overhead of 5-12%, a memory footprint of less than 10% and can be turned on and off per application.

One thing the tool can't do is hook onto already running processes. It is a consequence of using `ld_preload` to override the third-party libraries. The authors also state that the tool only needs one snapshot collected at the *end* of execution. This could be an unfortunate wording, but if it isn't it is a limitation in the technique. Since our profiler must be able to hook onto running processes that does not have an *end of execution* this tool will not solve our use case.

3.12 AddressWatcher

Murali et. al. presents their framework ADDRESSWATCHER [73]. Static analysis methods face a scalability problem when the target software has a lot of paths to evaluate while dynamic approaches can evaluate a single path with high precision. ADDRESSWATCHER builds up a data base over the heap allocated objects execution path over several test cases which allows it to both find when objects leak as well as suggest where to free the leaking objects.

To be able to track accesses (reads/writes) to heap allocated objects ADDRESSWATCHER implements a lightweight profiler that operates during compile time. The profiler inserts instrumentation before each access and uses LSan [69] to check for memory leaks.

The data base of object accesses are built up over different test cases. Assume there is a leaking object A that in test case t_1 is accessed (in order) o_3, o_4, o_5. Assume the same object is accessed in test case t_2 (in order) o_1, o_2, o_6. ADDRESSWATCHER would now suggest that A should be freed in o_5 and o_6.

One of the drawbacks of using ADDRESSWATCHER is the need for consistent memory layout. Thus ASLR (Address Space Layout Randomisation) is disabled.

When the authors tested their framework on binutils, openssh, tmux, openssl and git they successfully found 50 memory leaks and in 23 of the 50 cases ADDRESSWATCHER could point to a free location to fix the issue. The performance overhead of the framework is 142-178%.

Since ADDRESSWATCHER needs to recompile the program and has an overhead that is at best seven times higher than the target it is simply not a solution to the problem this paper aims to solve.

4 Project Plan

This chapter proposes a methodology for developing efficient memory tracing in live systems. First, a literature study will identify different tools and techniques for memory tracing. The primary focus will be their suitability for real-time systems, performance impact, and overall overhead. Based on these results, a custom memory tracing tool will be designed for the specific limitations. Finally, the prototype will be tested for performance, overhead, and scalability against already established tools.

4.1 Literature Study

To find suitable tools for memory tracing, a structured approach inspired by the guidelines for systematic literature reviews in *Experimentation in Software Engineering* [74] will be followed.

4.1.1 Planning the Study

Firstly, the scope and primary goal of the study is to identify existing tools that provide efficient real-time memory tracing with minimal overhead. Similar tools that do not support C++ or cannot run in live systems will be ignored. With regard to this, there are a few questions that will be in focus while evaluating the literature:

1. What tools/techniques currently support live memory tracing in C++ applications?
2. What are the performance impact and limitations of these tools/techniques?
3. How do these tools address overhead in resource-limited environments? (Maybe rewrite this question? its purpose is to ask what techniques they use to trace memory and achieve their overhead) (How can these tools/techniques be uses to trace memory and manage overhead in resource-constrained environments?)

4.1.2 Conducting the Study

To find suitable literature, a comprehensive search will be performed across different databases. The primary focus will be tools/techniques that support C/C++ and can be run in a live environment. The relevant data from the literature are then investigated, such as their capabilities and performance overhead.

Table 4.1: Search Terms Used Across Databases

Database	Search Terms	Comments/Notes
ACM Digital Library	"real-time memory tracing" OR "low-overhead profiling C++"	Used filters: C++ tools only
IEEE Xplore	"live memory analysis" AND "performance optimization"	Limited to papers after 2018
Google Scholar	"memory profiling C++" AND "real-time monitoring"	Broad search for comprehensive results
SpringerLink	"real-time memory tracking" OR "efficient memory usage"	Focused on low-overhead studies

4.1.3 Analyzing the Results

The results of the literature study will build the foundation for prototype development. Therefore, it is important that the selected tools/techniques are carefully evaluated for their compatibility with the requirements. They will be evaluated according to different metrics, such as memory use, performance impact, and capability of running on live systems. For the existing tools, it is important that they are flexible, well-documented, and possible to modify. This ensures that they can be expanded during the development phase to create a prototype. Conversely, the existing techniques only need to be evaluated according to the first metrics. They primarily serve as conceptual frameworks or algorithms that will be adapted or implemented as needed during the development phase.

4.2 Development Phase

The following section will describe the development phase in more detail.

4.2.1 Prototype Design

Designing the prototype will need consideration about both *performance constraints* and *real-time deployment*. Since 5G radio towers operate with very limited overhead, the tool must be lightweight, efficient, and able to integrate with existing C/C++ programs. The design will focus on minimal CPU overhead and deployment on running processes while being able to identify memory leaks. The prototype design will be structured as follows:

1. **Minimal performance overhead**
    - The tool should not exceed approximately 20% of CPU overhead to not interfere with the system's performance (20% of the total CPU time for the application).
    - Memory overhead should be minimal but it is not a requirement.
2. **Deployment on live processes**
    - The tool must be able to attach to running processes. Alternatively, it should use techniques that do not require restarting or recompiling the program.
    - If necessary, small changes and recompilation of code is possible. However, it should only be needed once as an installation step.
3. **Support for C/C++ programs and x86-architecture**
    - The tool should be designed specifically for C/C++ software.
    - The prototype will be developed primarily for *x86-architecture*. If compatibility with other architectures is possible, it will be considered a bonus addition.

4.2.2 Technology and Tools

Developing an efficient memory-tracking tool will require different tools and techniques. A key aspect during development is access to **low-level memory operations**, since it ensures minimal performance overhead. There are different components, needed at different levels, that fit the criteria. Some of them are as follows:

1. **Programming languages**
    - **C and C++:** The prototype will be implemented in C/C++ to ensure access to low-memory operations and compatibility with the target application.
    - **Python:** A simple speed up of the profiler would be to offload parts of the processing to another process. This part could be implemented in Python to make prototyping easier.
    - **Assembly (if needed):** Assembly might be used to allow intel family specific instructions (such as *ptwrite* [75]) or to optimize performance critical operations of the prototype.
2. **Techniques**
    - **ptrace:** Can be used to dynamically monitor memory operations of a process. [76]
    - **kprobes:** Can be used to monitor kernel functions such as kmalloc and kfree. [40]
    - **uprobes:** Can be used to monitor user-space applications such as libc malloc and free. [40]
    - **eBPF:** Provides a safe sandbox environment where applications can execute within the kernel. [40]
    - **PLT:** Can be used to overwrite the addresses of allocators and deallocators to intercept those calls.
    - **Backtraces:** Can be used to obtain and manipulate backtraces of the current thread [30].
3. **Benchmarking**
    - **perf:** A Linux performance monitoring tool that can be used to analyze CPU overhead. [39]
    - **Valgrind:** Can be used to evaluate memory allocation patterns and efficiency of the prototype. [77]
    - **Projects:** Projects at Linköping University and/or other open source projects such as StockFish [78] will be used. The overhead will be measured by running the application "as is" with the tool. The tools ability to detect unreachable memory will be evaluated once unreachable memory has been introduced into the application.
    - **Memwatch:** The tool will be evaluated against memwatch in terms of overhead and its ability to detect unreachable memory (should there be enough time).
    - **Custom benchmarking tools:** If necessary, custom test programs will be implemented to test the prototype in real-world scenarios.
4. **Build System**
    - **CMake:** To manage the build system across different platforms. [79]
    - **GCC/Clang:** To compile the prototype.

4.2.3 Implementation

The implementation of the prototype will follow three different steps to ensure efficiency, minimal overhead, and compatibility with live C/C++ systems.

Initial System Integration

The first step will involve integrating a tracking mechanism into existing C/C++ applications without requiring a process restart. There are several techniques, listed in subsection 4.2.2, that will be considered. It is important that they do not introduce any substantial performance overhead.

Data Collection

Once the tracking mechanism is in place, the focus will shift to data collection. The system should monitor *memory allocations*, *deallocations*, *and access patterns* in real-time and capture the relevant events. The collected data will be analyzed to find potential memory leaks.

Optimization for Live Systems

The final step involves evaluating the performance and optimizing the system for real-time deployment.

4.2.4 Testing and Validation

The tool will be evaluated using open source projects, projects at Linköping University, and live applications on Ericssons production servers (if the project progresses far enough). It is important that the testing software is neither too complex nor outdated. It must be possible to rewrite parts of the software to introduce *memory leaks* without it being cumbersome.

Once memory leaks have been introduced, the overhead will be measured by comparing the CPU usage of running the applications with and without the tool.

Valgrind and Memcheck will be used to evaluate what leaks it can find.

4.3 Time Plan

The following section will describe the time plan of this paper.

4.3.1 Main Tasks

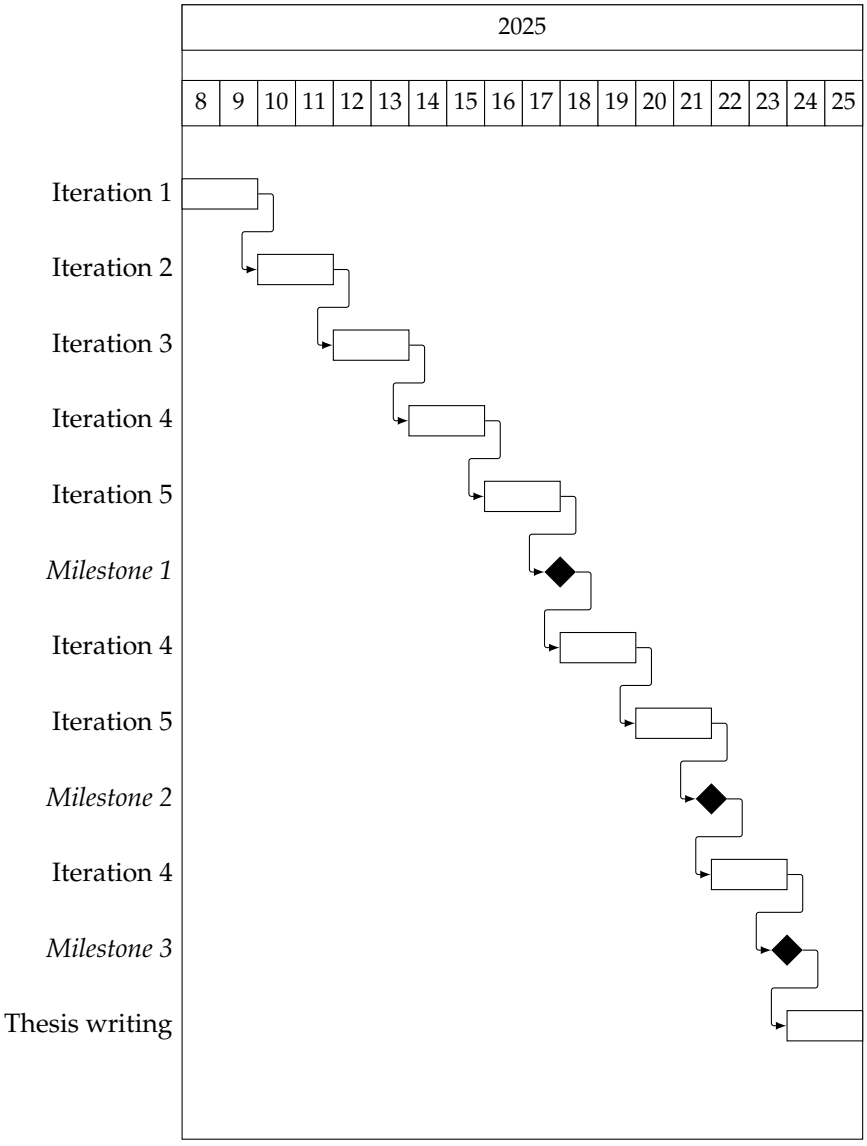
1. Setup phase
2. Literature study
3. Literature evaluation
4. Prototype design
5. Prototype implementation
6. Implementation testing
7. Implementation evaluation
8. Writing the final report

To avoid a waterfall time plan the project will follow an agile time line. The project will run on iterators consisting of two weeks. Each iteration will perform at least one of the following: research new techniques or methods (including writing about them), get a minimal implementation to work, further develop last iterations implementation, and test existing implementation.

4.3.2 Milestones

To ensure that the project does not fall behind, three milestones are set.

- **Finished implementation:** After five iterations the implementation is expected to be finished. This also includes writing about the different techniques and methods that has been considered and/or tried during these five weeks.
- **Testing finished:** After two additional iterations the implementation is expected to pass all tests.
- **Evaluation finished:** After one additional iteration the implementation is expected to have been evaluated, mearing its performance overhead as well as its ability to find memory leaks.



5 Results

This chapter describe the results obtained during the thesis project. Firstly a short description of the test suit is presented, secondly the the result from the literature study, thirdly the result of the initial prototype, and lastly, the results from testing the prototype is presented.

Så typ först testar vi det vi hittar i literaturstudien, sen går vi igenom hur vi implementerade mem-hook, sen visar vi testerna på mem-hook

5.1 Literature study results: Why existing tools don't solve this problem

Memory leaks as a problem is nothing new and this paper has introduced a number of techniques and programs that aim to solve and/or help developers deal with this issue. None of these techniques and programs solve the specific use case this paper aims to solve: detect memory leaks without the need to restart, stop, or recompile the target software.

Previous implementation: MEMWATCH

TODO: Check Memwatch accuraly with Anders or someone knowledgeable
Ericssons previous implementation MEMWATCH had to be compiled with the target software to enable profiling. Once compiled with the software MEMWATCH could be turned on and off, and had zero performance impact when not engaged. It was limited by its performance cost.

Valgrind and its tools

TODO: Add a reference to a testcase to verify
VALGRIND is simply too heavy to run in production. It should also be recompiled without optimizations if intended to be used with its default tool MEMWATCH.

Gdb, Ptrace, Strace, and Ltrace

TODO: Add a reference to a testcase to verify
GDB, PTRACE, STRACE, and LTRACE all suffer from the same performance overhead inherit in PTRACE-based tool.

Utrace

UTRACE is deprecated.

Kernel probes/SystemTap

KPROBES allows the user to probe functions in the Linux kernel for purposes such as debugging, tracing, or fault injecting. Tracking memory allocation made with `malloc` and released back to the operative system with `free` is not possible since `malloc` and `free` lies in user-space. Since SYSTEMTAP can be seen as a wrapper for KPROBES it also can not track allocations made with `malloc`.

User-space probes/eBPF

UPROBES have to perform context-switches when probing user-space programs since they are implemented in the kernel. These context-switches add overhead to the point where UPROBES are no longer a suitable solution. Figure 5.3 shows the overhead of hooking `malloc` with UPROBES in EBPF and the overhead adds roughly 2000 nanoseconds per call, which is simply too much to be usable in production code. It is possible to write a UPROBE-hook without EBPF, however that requires writing a kernel module which among other things specifies the virtual address and process to probe. Writing kernel modules is beyond the scope of this paper.

Perf

Since PERF implements tracepoints through UPROBES and KPROBES it suffers from the same context-switch overhead as UPROBES.

Swat

TODO: I think swat needs to recompile or inject code into the process which seems either too hard (or not possible due to Write XOR Execute). It checks accesses to memory at a rate inversely proportionately to their execution frequency and marks data not accessed as "leaks".
SWAT implemented the technique of sampling instructions inversely proportionately to their execution frequency to check for memory leaks. The authors defined a memory leak as *memory not accessed for a long period of time*.

Mesh

The approached used in the implementation of MESH cannot identify memory leaks on its own. Since it solves a different problem than the problem this paper aims to solve the technique of storing additional information in the redundant bits of 64-bit pointers is not of interest for this paper.

ASan

ADDRESSSANITIZSER suffer from the same issue as VALGRIND, GDB, and PTRACE-based tools. Its performance overhead is too big to be used in production.

Object Code Insertion

Object code insertion is a technique that inserts instructions into the object code, it is implemented in Purify by Hastings and Joyce [5]. Purify inserts the object code before linking making this technique unusable for this problem, since the techniques used must be usable with an already running process.

TODO: Program att testa och mäta overhead med alt skriva varför dom inte är relevanta för vårt användningsområde
Först lista alla tekniker/program som pappret kollat på, skriv om vilka program/tekniker som inte stödjer att attachas på en körandes process. Sen skriv om vilka program/tekniker som inte löser vårat problem (typ MESH som inte kan hitta minnesläckor alls), sen gör mätningar på samma program/domän i CPDDL som vi kommer testa vår implementation på?

- Memwatch? - Done
- Perf - Done
- gdb, Ptrace, Strace, Ltrace - Done
- Kprobes/SystemTap - Done
- Utrace - Done
- Uprobes - Done
- eBPF - Done
- Valgrind w. memcheck/taintcheck - Done
- Swat
- Object Code Insertion / Purify
- Mesh - Done

5.2 Test Setup

The tests were conducted on a system running Arch Linux 6.13.3 with an Intel Core i5-8350U (4 cores) @ 3.60 GHz processor. The benchmarking process was designed to minimize external interference:

- CPU frequency scaling was disabled to ensure consistent measurements. By default, modern CPUs adjust their frequency dynamically to save power, which can introduce variability in benchmarking [80].
- Tests were executed on an isolated core using *taskset* to reduce scheduler-induced variability.
- Each test was repeated 10 times, and the median execution time was recorded to minimize the impact of outliers.

5.3 eBPF Probe Overhead

To get an understanding of the implicit overhead of probing user-space functions with eBPF, a series of tests was designed to measure this impact. Specifically, the overhead introduced by tracing calls to malloc at both entry and exit points was measured.

5.3.1 Test Variants

- **Baseline:** Running *malloc* without any probes attached to measure the natural execution time.
- **Malloc Enter Probe:** Attaching an eBPF program to the entry of *malloc*.
- **Malloc Exit Probe:** Attaching an eBPF program to the exit of *malloc*.
- **Both Enter and Exit Probes:** Attaching probes at both function entry and exit.

5.3.2 Test Cases

To ensure accurate test data, different allocation sizes were tested, covering small, medium, and large memory allocations. Each allocation was run one million times. This test was then repeated ten times, and the median result was taken.

- **Small allocations:** 16 B, 32 B, 64 B, 128 B.
- **Medium allocations:** 256 B, 512 B, 1 KB.
- **Large allocations:** 4 KB, 64 KB.

Allocation Size	Baseline (ns)	Enter Only (ns)	Exit Only (ns)	Both (ns)
16 B	30	2043	2805	2884
32 B	35	2048	2812	2888
64 B	46	2061	2823	2902
128 B	70	2087	2847	2928
256 B	119	2135	2898	2977
512 B	214	2234	2997	3074
1 KB	404	2430	3196	3277
4 KB	1545	2424	4377	4463
64 KB	1846	3593	4781	4785

Table 5.1: Measured overhead of eBPF probes on malloc for different allocation sizes.

5.3.3 Real-World Application Test

In addition to the controlled malloc tests, a real-world application was also benchmarked to understand the practical overhead introduced by eBPF probes in a more complex scenario. The application used in this test was *Mimir* [81], a C++ tool for learning-based planning. The tests were performed with the same configuration as the previous tests, and the same probing scenarios were used. Each test was run 10 times, and the median execution time was recorded. The benchmark results are summarized in the table below:

Application	No Probes (ms)	Enter Only (ms)	Exit Only (ms)	Both (ms)
Mimir	3	19	25	26

Table 5.2: Measured overhead of eBPF probes in *Mimir*.

5.4 Hooking Malloc via the Procedure Linkage Table (PLT)

Since the overhead of user-space probing with EBPF appears to be too significant for production environments, a different approach is needed. One possible alternative is intercepting `malloc` through the PROCEDURE LINKAGE TABLE (PLT). This would add minimal overhead compared to EBPF and would not require any recompilation of the application.

5.4.1 Overview of the PLT

The PLT is a crucial mechanism in dynamically linked ELF binaries. When a dynamically linked function such as `malloc` is called for the first time, the PLT resolves the symbol via the GLOBAL OFFSET TABLE (GOT) and updates the entry for future calls [36]. By overwriting the corresponding PLT entry with the hook's function address, the call can be intercepted.

5.5 Mem-hook implementation

The section below will explain in detail how we implemented our profiler. The general structure of the profiler is a C++ "frontend" with a python "backend".

When the profiler is hooked onto a running process the python backend compiles the C++ code, inserts the hooks, and reads and processes the collected data. The C++ frontend collects the data and writes it into a shared memory the python backend reads. Below the following three parts will be explained in detail: plt hooking, collecting the meta data, and processing the data.

5.5.1 Hooking Implementation

The details of our hook implementation is described below.

PLT Offset Extraction

Firstly, the memory address of the function's PLT entry must be found. The ELF binary is disassembled using OBJDUMP to find this offset. For example, as shown in Figure 5.1, malloc's entry is located at `0x2f92` relative to the instruction pointer (which always points to the next instruction). Since the `jmp` instruction is 6 bytes long [82], `malloc's` PLT entry is located at `0x2f92` + `0x10a0` + `0x6` = `0x4038`, relative to the process's memory space.

```
> objdump -d <binary>

00000000000010a0 <malloc@plt>:
    10a0:       ff 25 92 2f 00 00       jmp    *0x2f92(%rip)        # 4038
        ↪ <malloc@GLIBC_2.2.5>
    10a6:       68 07 00 00 00          push   $0x7
    10ab:       e9 70 ff ff ff          jmp    1020 <_init+0x20>
```

Figure 5.1: Disassembly of the PLT entry for `malloc` using OBJDUMP.

Process Base Address Resolution

To find the absolute address of the PLT entry, the starting address of the process's memory space is obtained through `/proc/<pid>/maps`. For example, in Figure 5.2, the process starts at address `0x6422b1a3c000`. This puts `malloc's` PLT entry at `0x6422b1a3c000` + `0x4038` = `0x6422b1a40038`.

```
> cat /proc/<pid>/maps

6422b1a3c000-6422b1a3d000 r--p 00000000 103:02 3805174
6422b1a3d000-6422b1a3e000 r-xp 00001000 103:02 3805174
6422b1a3e000-6422b1a3f000 r--p 00002000 103:02 3805174
6422b1a3f000-6422b1a40000 r--p 00002000 103:02 3805174
6422b1a40000-6422b1a41000 rw-p 00003000 103:02 3805174
```

Figure 5.2: A memory space output example for a running process.

Library Injection

To add the custom hook function to an already running process, the hook is compiled as a shared dynamic library. Figure 5.3 showcases a minimal example of a `malloc` hook. The library is loaded into the processes through GDB as shown in Figure 5.4.

```
#include <iostream>
#include <dlfcn.h>

// Define a function pointer for the original functions
void* (*malloc_real)(size_t) = nullptr;

// The hook function for malloc
extern "C" void* malloc_hook(size_t size) {
    std::cout << "malloc hooked: " << size << " bytes" << std::endl;
    return malloc_real(size);  // Call the original malloc
}

void set_original_malloc() {
    malloc_real = (void* (*)(size_t)) dlsym(RTLD_NEXT, "malloc");
    if (!malloc_real) {
        std::cerr << "Failed to find original malloc: " << dlerror() << std
            ↪ ::endl;
        exit(1);
    }
}

// Entry point for the shared library
__attribute__((constructor)) void initialize() {
    set_original_malloc();
}
```

Figure 5.3: A minimal C++ library example for hooking malloc

```
> sudo gdb -p <pid>

(gdb) call (void*) dlopen("path/to/hook.so", 1)
$1 = (void *) 0x5b1ff70b8870
```

Figure 5.4: A minimal example that showcases loading a dynamic library with GDB.

Function Address Retrieval

Once the dynamic library has been loaded, the address of both the original `malloc` and the hook must be retrieved. By leveraging GDB once again, the addresses can be found as shown in Figure 5.5. In this case, they are located at `0x711b769bc190` and `0x711b76f7c189` respectively. The address of the original `malloc` is needed to restore the process once the hooking is complete.

PLT Entry Modification

The last step is to replace the PLT entry of the original `malloc` with the hook. For example, Figure 5.6 shows how `malloc's` PLT entry at `0x6422b1a40038` is replaced by the hook located at `0x711b76f7c189`.

```
> sudo gdb -p <pid>

(gdb) p malloc
$2 = {<text variable, no debug info>} 0x711b769bc190 <malloc>
(gdb) p malloc_hook
$3 = {<text variable, no debug info>} 0x711b76f7c189 <malloc_hook>
```

Figure 5.5: A function address retrieval example with GDB

```
> sudo gdb -p <pid>

(gdb) set *(void **) 0x6422b1a40038 = 0x711b76f7c189
```

Figure 5.6: An example showing the `malloc` PLT entry modified with GDB

Allocation Size	Baseline (ns)	Malloc hook (ns)
16 B	30	32
32 B	35	37
64 B	46	49
128 B	70	74
256 B	119	123
512 B	214	221
1 KB	404	417
4 KB	1545	1568
64 KB	1846	1872

Table 5.3: Measured overhead hooking malloc with PLT (in nanoseconds).

5.5.2 Collecting meta data

Once the relevant functions (`malloc`, `free`, `new`, `delete`, `new[]`, `delete[]`, `new no throw`) have been replaced by hooks they can collect the relevant data for each allocation/deallocation, hereby after referred to as a *trace*. For each trace the following information was collected: pointer to the allocation, timestamp when the trace was made, its size, what kind of operation that invoked the trace (`malloc`, `free`, `new`, `delete`, `new[]`, `delete[]`, `new no throw`), the number of backtraces, and lastly the actual backtraces.

Once the desired information has been collected in the hook it writes this to a shared memory that it shares with the python backend thorugh a call to `memcpy`. The shared buffer is explaind in more detail under 5.5.3.

Dealing with unknown allocations

Since the profiler supports hooking onto already running processes it sometimes receives a deallocation of an unknown allocation (e.g. the allocation happened before the profiler hooked onto the process). In this case it simply reports the size as unknown. If the user views the current allocations in the graph, it reports its size as zero instead. (? Is this true ?)

Collecting timestamp

The profiler supports three different ways of collecting the current time. The first method uses C++ chrono high resolution clock to get the time in nanoseconds. The second method uses `__rdtscp` to get the current value of the processors time-stamp counter [83]. The last method reads the time when the trace is read by the python backend. The last option is not as

exact but implemented as a possibility for the absolute minimum overhead when profiling. The profiler defaults to the C++ chrono option since the processors time-stamp counter will fluctuate with scaling frequency and the backend time will not be as exact as C++ chrono.

Backtrace

When collecting the backtrace the initial implementation used the `backtrace` function from the header `execinfo.h`. However, due to performance reasons the current implementation uses a custom backtrace function. As shown in table 6.1 the custom backtrace implementation reduces the cost of each allocation time by about 1500 nano seconds. Since different allocation sizes differers in the base time this effect is most prominent in the smaller allocation sizes.

Meta programming

In order to change how the profiler measures times and what backtrace function to use the profiler utilizes meta programming. Since the Python backend compiles the C++ hook before inserting it the backend can freely perform string replacements to insert valid C++ into the hook.

Meta programming is used for a few areas in the hook: to insert filters for different allocation sizes and ranges, to changing what buffer size the shared memory has, to set the time measuring method, and what backtrace function to use. In the C++ code there are "tags" on the form `<<<NAME>>>` which the backend can perform string replacement on. Looking at figure 5.7 one can see how this works. The tag `<<<ALLOC_FILTER_RANGE>>>` will either be replaced by a series of if-statements checking if the size of the trace is within the range of interest, or it will be replaced by the empty string removing it from the function. The same thing will happen for the tag `<<<ALLOC_FILTER>>>`.

The timestamp tag will never be replaced by an empty string. This is a design decision made to simplify the interface of the traces. It is simpler to collect the data from each trace in the backend if they all have the same structure. This means that the trace will always provide a value for the time. If the user specified that the time should be collected in the backend the tag will be replace by a line of code initializing the variable `timestamp` to zero and an if-statement in the backend will replace this value with a call to the Python function `time.time()`.

The tags `<<<USE_BACKTRACE_FAST>>>` and `<<<USE_BACKTRACE_GLIBC>>>` will be replaced by either the empty string or a function call, depending on what method the user specified.

5.5.3 The shared memory

To achieve as low of an overhead as possible the profiler should perform as little instructions as possible. One way to reduce the work done is to use the shared memory as a ring buffer. That way the hook would never have to erase anything and could instead continuously write information to the buffer whenever it collected a new trace. The hook would never have to clear a previous entry and could instead just write over it.

This requries that the frontend and backend keeps track of where they currently are writing and reading from. Thus the shared memory contains a header with a head describing where the frontend writes, a tail describing where the backend reads, and an overflow bool the frontend sets to true *if and only if* there is a *potential* of lost information (note that it is possible for the Python backend to read this information, and then for C++ to write over this information before Python increments the tail, thus setting the overflow to true even though no information was lost).

To calculate what address the frontend should write to and what address the backend should read from the following calculation is performed: start address of shared memory + size of head + (head or tail * size of a trace)

```
1  // The hook function for malloc
2  extern "C" void* malloc_hook(uint32_t size) {
3      void* const ptr{malloc_real(size)}; // Call the original malloc
4
5      <<<ALLOC_FILTER_RANGE>>>
6      <<<ALLOC_FILTER>>>
7      <<<TIMESTAMP>>>
8
9
10     std::array<void*, 20> backtrace_buffer{};
11
12     <<<USE_BACKTRACE_FAST>>>
13     <<<USE_BACKTRACE_GLIBC>>>
14
15     Trace trace{ptr, timestamp, size, backtrace_size, MALLOC,
           ↪ backtrace_buffer};
16     buffer.write(trace);
17     return ptr;
18 }
```

Figure 5.7: The code for the malloc hook implementation.

To get the next index the following calculation is performed:
(address of head or tail + 1) % (shared memory size / size of a trace), where % is the modular operator and / is integer division.

The calculation (shared memory size / size of a trace) gives the amount of traces that fits in the shared memory. Some space could end up never being used, but it is at most the size of one trace.

5.5.4 Python backend

The backend continuously reads for new traces. It never removes or writes anything to the shared memory (except for the tail).

When there are new traces to be read, the backend reads the traces and depending on if it is an allocation or a deallocation it does one of two options.

Allocations

For every type of allocation (`malloc`, `new`, `new[]`, `new no throw`) it stores each address as the key in a dictionary. The value of the pair is the sum of allocations made in that address. If a profiled program makes two allocations with the backtraces containing addresses 1,2,3 and 3,4,5 of 100 bytes each, the resulting dictionary would contain the keys 1,2,3,4,5 with all values being 100 except 3, which holds that value 200.

Deallocations

For every type of deallocation (`free`, `delete`, `delete[]`) it performs the same action but for deallocations. If the profiler was running when the allocation was made it also removes the allocation from the allocation dictionary. Continuing with the example from the paragraph above assume the program deallocates 100 bytes with a backtrace of 6, 7, 8, 9. The pointer to the allocation matches the allocation collected previous with backtrace 1,2,3, so the backend performs two actions. First it adds the addresses 6,7,8,9 as keys to the deallocation dictionary (unless they already exists) and increments their value with the size of the deallocation, 100 in this example. Second it removes the value 100 from the allocation dictionary on all keys matching *that* allocations backtrace, which in this case would be 1,2,3.

Stored information

The backend keeps track of all allocations and all deallocations made. This allows the backend to keep track of what backtraces have outstanding allocations, as well as what backtraces that perform allocations/deallocations with the traces information such as sizes, time, etc.

When starting the profiler the user can enter an optional argument to specify a file path. If this argument is provided the profiler will write all available information to the file when turned off. Otherwise it simply outputs the current status to the terminal in regular intervals.

5.6 How to get function name from backtraces

Since backtraces gives addresses that have been mangled by the Address space layout randomization (ASLR) we need to calculate its true address. This is done by subtracting the highest address that is less than or equal to the address in the backtrace. The address groups can be read from /proc/<pid>/maps. Once the real address has been calculated the function can be gathered by inspecting the binary of the program/library from the group used in /proc/<pid>/maps. It will most likely map to a call function so once the real address in the correct binary has been gathered we can "walk" up the address space in the binary until we reach a function. The only things that need to be done will the hook is running is reading /proc/<pid>/maps. All other calculations can be made offline.

6 Evaluation

Assuming that some kind of prototype/artifact/system/component has been developed/adapted/changed/implemented this should typically be evaluated. This chapter then becomes a "mini"-version of the IMRaD model. The structure below is just one example of how one can structure this kind of chapter. It must be adapted to the kind of evaluation that is being done.

6.1 Evaluation method

Describe the method used in the evaluation. Be detailed and specific. Use tables to summarise parameters used if you are running simulations. Describe what experiments that are conducted, and their purpose. Explain the metrics used, and motivate their choice.

6.2 Results

Show the results of your evaluation. Explain to the reader what the results show.

6.3 Summary of results

It is often useful to provide some kind of summary of the results, especially if there are many graphs and tables.

6.4 Dumping all tests here

Allocation Size	Baseline (ns)	Custom Backtrace (ns)			Glibc Backtrace (ns)		
		5	10	20	5	10	20
16 B	24	130	145	171	714	1139	1935
32 B	25	133	148	174	722	1144	1936
64 B	35	146	158	184	731	1161	2019
128 B	55	165	179	207	754	1187	1985
256 B	93	204	220	248	800	1224	2030
512 B	170	237	250	277	829	1311	2111
1 KB	318	395	417	421	1011	1480	2291
4 KB	1190	1314	1289	1328	2050	2497	3299
64 KB	1461	1565	1555	1578	2278	2781	3538

Table 6.1: Measured allocation time of MEMHOOK with different backtrace methods and depths.

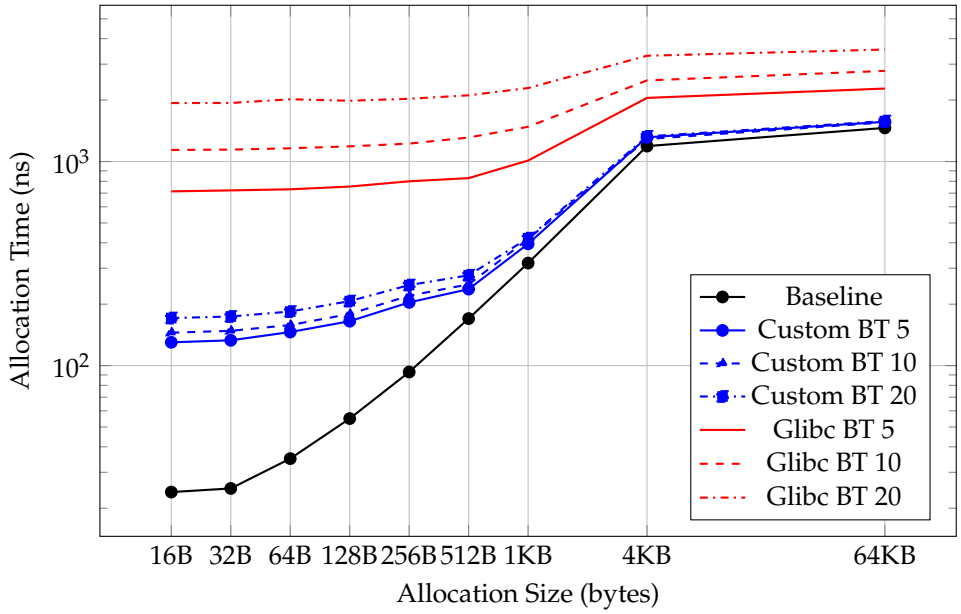
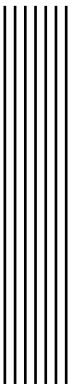


Figure 6.1: Allocation time for various backtrace methods and depths.

Application	Execution Time (ms)
Baseline	14
custom profiler	283
Memleak [50]	603

Table 6.2: Tests done on realistic.cpp with different profilers



Bibliography

[1] Nicholas Nethercote and Julian Seward. "Valgrind: a framework for heavyweight dynamic binary instrumentation". In: *ACM Sigplan notices* 42.6 (2007), pp. 89–100.

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

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

[4] Junfeng Yang, Paul Twohey, Dawson Engler, and Madanlal Musuvathi. "Using model checking to find serious file system errors". In: *ACM Transactions on Computer Systems (TOCS)* 24.4 (2006), pp. 393–423.

[5] Reed Hastings. "Purify: Fast detection of memory leaks and access errors". In: *Proceedings of the USENIX Winter'92 Conference*. 1992, pp. 125–136.

[6] Mark Mansi and Michael M Swift. "Characterizing Physical Memory Fragmentation". In: *arXiv preprint arXiv:2401.03523* (2024).

[7] Vahid Azhari, Simar Bhamra, Naser Ezzati-Jivan, and François Tetreault. "Efficient heap monitoring tool for memory leak detection and root-cause analysis". In: *2021 IEEE International Conference on Big Data (Big Data)*. IEEE. 2021, pp. 3020–3030.

[8] Santosh Nagarakatte, Milo MK Martin, and Steve Zdancewic. "Watchdog: Hardware for safe and secure manual memory management and full memory safety". In: *ACM SIGARCH Computer Architecture News* 40.3 (2012), pp. 189–200.

[9] Matthias Hauswirth and Trishul M Chilimbi. "Low-overhead memory leak detection using adaptive statistical profiling". In: *Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*. 2004, pp. 156–164.

[10] Hans-Juergen Boehm. "Space efficient conservative garbage collection". In: *ACM SIGPLAN Notices* 28.6 (1993), pp. 197–206.

[11] David L Heine and Monica S Lam. "A practical flow-sensitive and context-sensitive C and C++ memory leak detector". In: *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*. 2003, pp. 168–181.

[12] Yichen Xie and Alex Aiken. "Context-and path-sensitive memory leak detection". In: *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*. 2005, pp. 115–125.

[13] Sigmund Cherem, Lonnie Princehouse, and Radu Rugina. "Practical memory leak detection using guarded value-flow analysis". In: *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2007, pp. 480–491.

[14] Yungbum Jung and Kwangkeun Yi. "Practical memory leak detector based on parameterized procedural summaries". In: *Proceedings of the 7th international symposium on Memory management*. 2008, pp. 131–140.

[15] Dacong Yan, Guoqing Xu, Shengqian Yang, and Atanas Rountev. "Leakchecker: Practical static memory leak detection for managed languages". In: *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*. 2014, pp. 87–97.

[16] Victor Van der Veen, Nitish Dutt-Sharma, Lorenzo Cavallaro, and Herbert Bos. "Memory errors: The past, the present, and the future". In: *Research in Attacks, Intrusions, and Defenses: 15th International Symposium, RAID 2012, Amsterdam, The Netherlands, September 12-14, 2012. Proceedings 15*. Springer. 2012, pp. 86–106.

[17] Trevor Jim, J Gregory Morrisett, Dan Grossman, Michael W Hicks, James Cheney, and Yanling Wang. "Cyclone: a safe dialect of C." In: *USENIX Annual Technical Conference, General Track*. 2002, pp. 275–288.

[18] George C Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. "CCured: Type-safe retrofitting of legacy software". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 27.3 (2005), pp. 477–526.

[19] Periklis Akritidis, Manuel Costa, Miguel Castro, and Steven Hand. "Baggy Bounds Checking: An Efficient and Backwards-Compatible Defense against Out-of-Bounds Errors." In: *USENIX Security Symposium*. Vol. 10. 2009, p. 96.

[20] Olatunji Ruwase and Monica S Lam. "A Practical Dynamic Buffer Overflow Detector." In: *NDSS*. Vol. 2004. 2004, pp. 159–169.

[21] Tzi-cker Chiueh and Fu-Hau Hsu. "RAD: A compile-time solution to buffer overflow attacks". In: *Proceedings 21st International Conference on Distributed Computing Systems*. IEEE. 2001, pp. 409–417.

[22] Michalis Polychronakis, Kostas G Anagnostakis, and Evangelos P Markatos. "Comprehensive shellcode detection using runtime heuristics". In: *Proceedings of the 26th Annual Computer Security Applications Conference*. 2010, pp. 287–296.

[23] Gaurav S Kc, Angelos D Keromytis, and Vassilis Prevelakis. "Countering code-injection attacks with instruction-set randomization". In: *Proceedings of the 10th ACM conference on Computer and communications security*. 2003, pp. 272–280.

[24] SANS. *CWE/SANS TOP 25 Most Dangerous Software Errors*. Accessed: 2024-12-02. URL: https://www.sans.org/top25-software-errors/.

[25] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. "Return-oriented programming: Systems, languages, and applications". In: *ACM Transactions on Information and System Security (TISSEC)* 15.1 (2012), pp. 1–34.

[26] Stephen Fewer. *Pwn2Own 2011: IE8 on Windows 7 hijacked with 3 vulnerabilities*. Accessed: 2024-12-02. May 2011. URL: https://www.zdnet.com/article/pwn2own-2011-ie8-on-windows-7-hijacked-with-3-vulnerabilities/.

[27] Ryan Naraine. *Safari/MacBook first to fall at Pwn2Own*. Accessed: 2024-12-02. Mar. 2011. URL: https://www.zdnet.com/article/safarimacbook-first-to-fall-at-pwn2own-2011/.

[28] Microsoft BlueHat. *Microsoft BlueHat Prize Contest*. 2011.

[29] Randal E Bryant and David Richard O'Hallaron. *Computer systems: a programmer's perspective*. Prentice Hall, 2011.

[30] Free Software Foundation. *Backtraces*. `https://www.gnu.org/software/libc/manual/html_node/Backtraces.html`. Accessed: 2025-03-10. 2025.

[31] LLVM Project. *Clang Command Line Reference*. Accessed: 2025-04-08. 2025. URL: `https://clang.llvm.org/docs/ClangCommandLineReference.html`.

[32] Fedora Project. *Changes/fno-omit-frame-pointer - Fedora Project Wiki*. Accessed: 2025-04-08. 2025. URL: `https://fedoraproject.org/wiki/Changes/fno-omit-frame-pointer`.

[33] Jakub Jelinek. *Comment on Change proposal: Add -fno-omit-frame-pointer to default compilation flags*. Pagure Issue #2817, Comment #826636. Accessed: 2025-04-10. 2023. URL: `https://pagure.io/fesco/issue/2817#comment-826636`.

[34] Linux Kernel Labs. *Introduction to the Linux Kernel*. Accessed: 2025-04-07. 2024. URL: `https://linux-kernel-labs.github.io/refs/heads/master/lectures/intro.html`.

[35] Kai Luke. *Interaction Between the User and Kernel Space in Linux*. 2017.

[36] M Tim Jones. "Anatomy of Linux dynamic libraries". In: *IBM developer-Works* (2008).

[37] Linux man-pages project. *ld.so(8) – dynamic linker/loader*. Accessed: 2025-04-24. 2025. URL: `https://man7.org/linux/man-pages/man8/ld.so.8.html`.

[38] Free Software Foundation. *GNU Binutils*. Accessed: 2025-03-07. 2025. URL: `https://man7.org/linux/man-pages/man1/objdump.1.html`.

[39] perf Wiki. *perf: Linux profiling with performance counters*. Accessed: 2025-02-02. 2024. URL: `https://perfwiki.github.io/main/`.

[40] Jim Keniston, Ananth Mavinakayanahalli, Prasanna Panchamukhi, and Vara Prasad. "Ptrace, utrace, uprobes: Lightweight, dynamic tracing of user apps". In: *Proceedings of the 2007 Linux symposium*. Vol. 1. 2007, pp. 215–224.

[41] Ananth Mavinakayanahalli, Prasanna Panchamukhi, Jim Keniston, Anil Keshavamurthy, and Masami Hiramatsu. "Probing the guts of kprobes". In: *Linux Symposium*. Vol. 6. 2006, p. 5.

[42] SystemTap Developers. *Utrace - SystemTap Wiki*. `https://sourceware.org/systemtap/wiki/utrace`. Accessed: 2025-03-13. 2014.

[43] Bolaji Gbadamosi, Luigi Leonardi, Tobias Pulls, Toke Høiland-Jørgensen, Simone Ferlin-Reiter, Simo Sorce, and Anna Brunström. "The eBPF Runtime in the Linux Kernel". In: *arXiv preprint arXiv:2410.00026* (2024).

[44] IOVisor Project. *BPF Compiler Collection (BCC)*. Accessed: 2025-02-10. 2025. URL: `https://github.com/iovisor/bcc`.

[45] bpftrace Developers. *bpftrace: High-Level Tracing Language for eBPF*. Accessed: 2025-02-10. 2025. URL: `https://github.com/bpftrace/bpftrace`.

[46] Linux Kernel Documentation. *libbpf Overview*. `https://docs.kernel.org/bpf/libbpf/libbpf_overview.html`. Accessed: 2025-02-10. 2025.

[47] Milo Craun, Khizar Hussain, Uddhav Gautam, Zhengjie Ji, Tanuj Rao, and Dan Williams. "Eliminating eBPF Tracing Overhead on Untraced Processes". In: *Proceedings of the ACM SIGCOMM 2024 Workshop on eBPF and Kernel Extensions*. 2024, pp. 16–22.

[48] Juan Lopez, Leonardo Babun, Hidayet Aksu, and A Selcuk Uluagac. "A survey on function and system call hooking approaches". In: *Journal of Hardware and Systems Security* 1 (2017), pp. 114–136.

[49] Zhilu Lian, Yangzi Li, Zhixiang Chen, Shiwen Shan, Baoxin Han, and Yuxin Su. "eBPF-based working set size estimation in memory management". In: *2022 International Conference on Service Science (ICSS)*. IEEE. 2022, pp. 188–195.

[50] Mankier. *bcc-memleak - BPF-based memory leak detection tool*. Accessed: 2025-02-10. 2025. URL: https://www.mankier.com/8/bcc-memleak.

[51] Yusheng Zheng, Tong Yu, Yiwei Yang, Yanpeng Hu, XiaoZheng Lai, and Andrew Quinn. *bpftime: userspace eBPF Runtime for Uprobe, Syscall and Kernel-User Interactions*. 2023. arXiv: 2311.07923 [cs.OS].

[52] The GNU Project. *GNU Awk User's Guide*. Accessed: 2025-02-17. 2025. URL: https://www.gnu.org/software/gawk/manual/gawk.html.

[53] DTrace Project. *DTrace: Dynamic Tracing Facility*. Accessed: 2025-02-17. 2025. URL: https://dtrace.org/.

[54] Brendan Gregg. *bpftrace for Linux 4.9*. Accessed: 2025-02-17. 2019. URL: https://www.brendangregg.com/blog/2019-08-19/bpftrace.html.

[55] Chris Lattner and Vikram Adve. "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation". In: *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*. Palo Alto, California, USA: IEEE Computer Society, 2004, pp. 75–88. URL: https://dl.acm.org/doi/10.5555/977395.977673.

[56] LLVM Project. *LLVM Compiler Infrastructure*. https://llvm.org/. Accessed: 2025-05-02. 2025.

[57] James Clause, Wanchun Li, and Alessandro Orso. "Dytan: a generic dynamic taint analysis framework". In: *Proceedings of the 2007 international symposium on Software testing and analysis*. 2007, pp. 196–206.

[58] Jingfei Kong, Cliff C Zou, and Huiyang Zhou. "Improving software security via runtime instruction-level taint checking". In: *Proceedings of the 1st workshop on Architectural and system support for improving software dependability*. 2006, pp. 18–24.

[59] James Newsome and Dawn Xiaodong Song. "Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software." In: *NDSS*. Vol. 5. Citeseer. 2005, pp. 3–4.

[60] Feng Qin, Cheng Wang, Zhenmin Li, Ho-seop Kim, Yuanyuan Zhou, and Youfeng Wu. "Lift: A low-overhead practical information flow tracking system for detecting security attacks". In: *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*. IEEE. 2006, pp. 135–148.

[61] Julian Seward and Nicholas Nethercote. "Using Valgrind to Detect Undefined Value Errors with Bit-Precision." In: *USENIX Annual Technical Conference, General Track*. 2005, pp. 17–30.

[62] Valgrind Developers. *Valgrind User Manual*. Accessed: 2025-03-11. 2025. URL: https://valgrind.org/docs/manual/manual-core.html.

[63] Jonas Maebe, Michiel Ronsse, and Koen De Bosschere. "DIOTA: Dynamic instrumentation, optimization and transformation of applications". In: *Compendium of Workshops and Tutorials held in conjunction with PACT'02*. Citeseer. 2002.

[64] Frida Project. *Frida: Dynamic instrumentation toolkit for developers, reverse-engineers, and security researchers*. Accessed: 2025-03-07. 2025. URL: https://frida.re/.

[65] Gang Fan, Rongxin Wu, Qingkai Shi, Xiao Xiao, Jinguo Zhou, and Charles Zhang. "Smoke: scalable path-sensitive memory leak detection for millions of lines of code". In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE. 2019, pp. 72–82.

[66] Leonardo De Moura and Nikolaj Bjørner. "Z3: An efficient SMT solver". In: *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2008, pp. 337–340.

[67] Ziyang Xu, Yebin Chon, Yian Su, Zujun Tan, Sotiris Apostolakis, Simone Campanoni, and David I August. "PROMPT: A Fast and Extensible Memory Profiling Framework". In: *Proceedings of the ACM on Programming Languages* 8.OOPSLA1 (2024), pp. 449–473.

[68] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. "{AddressSanitizer}: A fast address sanity checker". In: *2012 USENIX annual technical conference (USENIX ATC 12)*. 2012, pp. 309–318.

[69] Google. *AddressSanitizer*. `https://github.com/google/sanitizers/wiki/addresssanitizer`. Accessed: 2025-03-13. 2019.

[70] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. "CETS: compiler enforced temporal safety for C". In: *Proceedings of the 2010 international symposium on Memory management*. 2010, pp. 31–40.

[71] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. "SoftBound: Highly compatible and complete spatial memory safety for C". In: *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2009, pp. 245–258.

[72] Emanuel Q Vintila, Philipp Zieris, and Julian Horsch. "MESH: A Memory-Efficient Safe Heap for C/C++". In: *Proceedings of the 16th International Conference on Availability, Reliability and Security*. 2021, pp. 1–10.

[73] Aniruddhan Murali, Mahmoud Alfadel, Meiyappan Nagappan, Meng Xu, and Chengnian Sun. "AddressWatcher: Sanitizer based Localization of Memory Leak Fixes". In: *IEEE Transactions on Software Engineering* (2024).

[74] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in Software Engineering*. 2nd. Berlin, Germany: Springer, 2024. ISBN: 978-3-662-69305-6. DOI: `10.1007/978-3-662-69306-3`.

[75] felixcloutier.com. *PTWRITE — Write Data to a Processor Trace Packet*. Accessed: 2025-02-10. NA. URL: `https://www.felixcloutier.com/x86/ptwrite`.

[76] man7.org. *ptrace(2) - Linux manual page*. Accessed: 2025-02-02. 2024. URL: `https://man7.org/linux/man-pages/man2/ptrace.2.html`.

[77] Valgrind Developers. *Valgrind: Instrumentation Framework for Building Dynamic Analysis Tools*. Accessed: 2025-02-02. 2024. URL: `https://valgrind.org/`.

[78] The Stockfish developers (see AUTHORS file). *Stockfish*. URL: `https://github.com/official-stockfish/Stockfish`.

[79] Kitware, Inc. *CMake: Cross-Platform Build System*. Accessed: 2025-02-02. 2024. URL: `https://cmake.org/`.

[80] Zygmunt Mazur and Piotr Boryczko. "Impact of Processor Frequency Scaling on Performance and Energy Consumption for WZ Factorization on Multicore Architecture". In: *Proceedings of the 2023 Annals of Computer Science and Information Systems*. Accessed: 2025-03-03. 2023. URL: `https://annals-csis.org/proceedings/2023/drp/pdf/6213.pdf`.

[81] Simon Ståhlberg. *Mimir*. `https://github.com/simon-stahlberg/mimir`. Accessed: 2025-03-03. 2025.

[82] Felix Cloutier. *JMP — Jump*. `https://www.felixcloutier.com/x86/jmp`. Accessed: 2025-03-11.

[83] Felix Cloutier. *RDTSCP — Read Time-Stamp Counter and Processor ID*. Accessed: 2025-04-28. 2023. URL: `https://www.felixcloutier.com/x86/rdtscp`.

A Appendix

A.1 Benchmark Results

A.1.1 Allocation Latency Benchmark Baseline

```
> taskset -c 5 ./test_alloc_time
```

Run	Allocation Time (ns) per Size (B)
1	16: 23.5209, 32: 25.6207, 64: 33.8892, 128: 55.4473, 256: 92.3778, 512: 167.473, 1024: 318.324, 4096: 1182.26, 65536: 1457.41
2	16: 22.6832, 32: 26.3558, 64: 36.0379, 128: 55.9557, 256: 97.229, 512: 173.736, 1024: 327.965, 4096: 1197.87, 65536: 1476.35
3	16: 23.4974, 32: 25.2574, 64: 37.04, 128: 54.9354, 256: 92.9587, 512: 166.984, 1024: 314.644, 4096: 1176.32, 65536: 1453.02
4	16: 24.2638, 32: 27.3865, 64: 36.6828, 128: 54.9326, 256: 101.066, 512: 172.681, 1024: 323.125, 4096: 1191.5, 65536: 1474.77
5	16: 24.5626, 32: 25.2756, 64: 34.3628, 128: 51.9848, 256: 97.1034, 512: 173.698, 1024: 319.023, 4096: 1189.26, 65536: 1461.73
6	16: 24.2598, 32: 26.4324, 64: 34.2273, 128: 54.242, 256: 93.7054, 512: 170.027, 1024: 318.459, 4096: 1187.61, 65536: 1460.09
7	16: 22.004, 32: 25.7274, 64: 38.1623, 128: 55.5057, 256: 92.9101, 512: 168.205, 1024: 315.805, 4096: 1190.03, 65536: 1456.21
8	16: 21.5524, 32: 25.3247, 64: 34.188, 128: 52.5676, 256: 91.8284, 512: 169.642, 1024: 319.653, 4096: 1195.53, 65536: 1444.62
9	16: 22.328, 32: 25.0784, 64: 34.5611, 128: 52.8354, 256: 90.7516, 512: 167.491, 1024: 311.607, 4096: 1164.04, 65536: 1460.94
10	16: 23.5646, 32: 24.8237, 64: 34.841, 128: 58.4268, 256: 95.5488, 512: 170.606, 1024: 317.933, 4096: 1218.46, 65536: 1470.3

A.1.2 Allocation Latency Benchmark of Memhook (custom backtrace, depth 5)

```
> taskset -c 5 ./test_alloc_time
```

```
> sudo python mem-hook.py -se 1000000 -hf malloc free -bm fast -mb 5 -p <
    ↪ PID>
```

Run	Allocation Time (ns) per Size (B)
1	16: 168.876, 32: 170.843, 64: 180.677, 128: 201.33, 256: 241.286, 512: 238.51, 1024: 419.869, 4096: 1315.81, 65536: 1559.11,
2	16: 132.002, 32: 133.329, 64: 143.668, 128: 164.018, 256: 204.152, 512: 234.344, 1024: 391.552, 4096: 1311.92, 65536: 1558.23,
3	16: 131.572, 32: 133.622, 64: 143.936, 128: 168.42, 256: 208.277, 512: 239.508, 1024: 395.231, 4096: 1306.11, 65536: 1600.72,
4	16: 129.708, 32: 131.998, 64: 143.389, 128: 164.048, 256: 202.645, 512: 236.597, 1024: 392.823, 4096: 1328.34, 65536: 1571.83,
5	16: 129.751, 32: 132.466, 64: 142.461, 128: 163.163, 256: 201.217, 512: 237.639, 1024: 391.988, 4096: 1317.18, 65536: 1572.73,
6	16: 129.243, 32: 131.523, 64: 147.035, 128: 163.14, 256: 204.013, 512: 234.698, 1024: 398.047, 4096: 1311.81, 65536: 1561.21,
7	16: 129.816, 32: 133.011, 64: 151.164, 128: 165.752, 256: 205.051, 512: 236.416, 1024: 395.444, 4096: 1312.49, 65536: 1569.54,
8	16: 131.224, 32: 136.356, 64: 148.086, 128: 164.746, 256: 203.897, 512: 235.243, 1024: 390.057, 4096: 1309.96, 65536: 1559.55,
9	16: 129.577, 32: 136.11, 64: 145.708, 128: 172.799, 256: 214.248, 512: 236.417, 1024: 397, 4096: 1336.61, 65536: 1557.37,
10	16: 129.277, 32: 132.761, 64: 163.364, 128: 169.567, 256: 202.498, 512: 246.305, 1024: 395.946, 4096: 1359.2, 65536: 1568.97,

A.1.3 Allocation Latency Benchmark of Memhook (custom backtrace, depth 10)

```
> taskset -c 5 ./test_alloc_time
> sudo python mem-hook.py -se 1000000 -hf malloc free -bm fast -mb 10 -p <
    ↪ PID>
```

Run	Allocation Time (ns) per Size (B)
1	16: 153.203, 32: 149.247, 64: 158.167, 128: 180.149, 256: 219.887, 512: 250.098, 1024: 406.309, 4096: 1284.16, 65536: 1578.32,
2	16: 144.853, 32: 147.267, 64: 180.69, 128: 179.258, 256: 219.74, 512: 250.87, 1024: 416.025, 4096: 1287.87, 65536: 1558.03,
3	16: 145.524, 32: 149.462, 64: 159.58, 128: 185.812, 256: 229.987, 512: 251.184, 1024: 415.058, 4096: 1288.9, 65536: 1561.05,
4	16: 144.226, 32: 146.8, 64: 157.672, 128: 180.022, 256: 222.586, 512: 251.271, 1024: 418.933, 4096: 1288.55, 65536: 1539.45,
5	16: 144.832, 32: 147.807, 64: 158.149, 128: 179.58, 256: 220.211, 512: 250.319, 1024: 413.562, 4096: 1299.73, 65536: 1552.65,
6	16: 144.77, 32: 146.518, 64: 157.655, 128: 178.88, 256: 219.656, 512: 250.113, 1024: 434.696, 4096: 1302.95, 65536: 1551.92,
7	16: 144.484, 32: 146.603, 64: 157.836, 128: 177.711, 256: 217.169, 512: 248.827, 1024: 394.415, 4096: 1341.04, 65536: 1596.92,
8	16: 144.857, 32: 148.444, 64: 159.237, 128: 178.484, 256: 219.478, 512: 252.745, 1024: 422.514, 4096: 1278.7, 65536: 1523.4,
9	16: 146.162, 32: 147.89, 64: 157.407, 128: 181.541, 256: 219.45, 512: 249.891, 1024: 439.439, 4096: 1284.55, 65536: 1546.97,
10	16: 145.068, 32: 147.347, 64: 158.115, 128: 178.616, 256: 218.642, 512: 250.247, 1024: 420.349, 4096: 1299.47, 65536: 1563.38,

A.1.4 Allocation Latency Benchmark of Memhook (custom backtrace, depth 20)

```
> taskset -c 5 ./test_alloc_time
> sudo python mem-hook.py -se 1000000 -hf malloc free -bm fast -mb 20 -p <
    ↪ PID>
```

Run	Allocation Time (ns) per Size (B)
1	16: 171.427, 32: 174.184, 64: 184.761, 128: 204.465, 256: 245.905, 512: 274.892, 1024: 418.181, 4096: 1310.28, 65536: 1575.63,
2	16: 170.367, 32: 173.904, 64: 184.261, 128: 205.383, 256: 244.286, 512: 277.406, 1024: 420.279, 4096: 1328.4, 65536: 1577.32,
3	16: 172.611, 32: 173.858, 64: 185.357, 128: 208.318, 256: 251.798, 512: 287.295, 1024: 437.094, 4096: 1339.85, 65536: 1595.58,
4	16: 170.947, 32: 174.87, 64: 187.511, 128: 207.007, 256: 250.072, 512: 278.562, 1024: 426.162, 4096: 1327.06, 65536: 1600.6,
5	16: 171.897, 32: 179.729, 64: 186.541, 128: 209.707, 256: 247.642, 512: 277.02, 1024: 423.437, 4096: 1334.76, 65536: 1578.58,
6	16: 172.93, 32: 173.142, 64: 184.582, 128: 207.826, 256: 248.746, 512: 276.116, 1024: 421.389, 4096: 1318.62, 65536: 1565.77,
7	16: 171.412, 32: 172.708, 64: 184.142, 128: 205.086, 256: 248.802, 512: 278.104, 1024: 420.527, 4096: 1338.89, 65536: 1577.67,
8	16: 171.445, 32: 172.982, 64: 183.519, 128: 206.425, 256: 249.8, 512: 274.977, 1024: 416.926, 4096: 1324.34, 65536: 1571.6,
9	16: 170.719, 32: 173.985, 64: 184.248, 128: 205.812, 256: 242.803, 512: 275.246, 1024: 424.166, 4096: 1329.29, 65536: 1582.28,
10	16: 170.298, 32: 174.107, 64: 183.765, 128: 216.952, 256: 246.401, 512: 277.139, 1024: 416.156, 4096: 1314.97, 65536: 1601.42,

A.1.5 Allocation Latency Benchmark of Memhook (Glibc backtrace, depth 5)

```
> taskset -c 5 ./test_alloc_time
> sudo python mem-hook.py -se 1000000 -hf malloc free -bm glibc -mb 5 -p <
    ↪ PID>
```

Run	Allocation Time (ns) per Size (B)
1	16: 709.894, 32: 715.933, 64: 724.39, 128: 751.609, 256: 792.219, 512: 825.865, 1024: 1007.84, 4096: 2059.54, 65536: 2262.7,
2	16: 711.232, 32: 737.632, 64: 738.071, 128: 746.395, 256: 795.617, 512: 825.219, 1024: 1010.48, 4096: 2016.05, 65536: 2282.37,
3	16: 714.107, 32: 718.874, 64: 727.838, 128: 754.531, 256: 794.642, 512: 826.185, 1024: 1008.88, 4096: 2019.92, 65536: 2279.52,
4	16: 713.065, 32: 722.397, 64: 857.723, 128: 754.233, 256: 814.706, 512: 832.296, 1024: 1006.69, 4096: 2020.16, 65536: 2270.35,
5	16: 725.601, 32: 730.08, 64: 734.75, 128: 772.98, 256: 798.354, 512: 832.074, 1024: 1124.03, 4096: 2016.28, 65536: 2272.91,
6	16: 726.557, 32: 737.17, 64: 738.299, 128: 763.317, 256: 802.643, 512: 855.918, 1024: 1012.69, 4096: 2070.78, 65536: 2343.08,
7	16: 709.562, 32: 716.216, 64: 731.399, 128: 746.207, 256: 790.557, 512: 825.735, 1024: 1010.25, 4096: 2070, 65536: 2318.23,
8	16: 713.2, 32: 718.809, 64: 724.625, 128: 774.618, 256: 810.625, 512: 826.357, 1024: 1010.67, 4096: 2052.47, 65536: 2276.45,
9	16: 714.385, 32: 720.608, 64: 729.019, 128: 752.168, 256: 837.66, 512: 831.118, 1024: 1018.01, 4096: 2139.16, 65536: 2249.09,
10	16: 716.055, 32: 738.36, 64: 729.614, 128: 775.943, 256: 820.64, 512: 835.642, 1024: 1012.64, 4096: 2049.79, 65536: 2297.92,

A.1.6 Allocation Latency Benchmark of Memhook (Glibc backtrace, depth 10)

```
> taskset -c 5 ./test_alloc_time
> sudo python mem-hook.py -se 1000000 -hf malloc free -bm glibc -mb 10 -p <
    ↪ PID>
```

Run	Allocation Time (ns) per Size (B)
1	16: 1146.63, 32: 1147.04, 64: 1162.96, 128: 1183.24, 256: 1233.3, 512: 1259.17, 1024: 1436.78, 4096: 2441.15, 65536: 2724.9,
2	16: 1137.37, 32: 1144.2, 64: 1146.53, 128: 1171.33, 256: 1218.41, 512: 1299.39, 1024: 1476.79, 4096: 2476.26, 65536: 2735.98,
3	16: 1136.91, 32: 1143.14, 64: 1158.67, 128: 1196.3, 256: 1262.97, 512: 1349.21, 1024: 1472.63, 4096: 2499.97, 65536: 2775.65,
4	16: 1136.98, 32: 1137.62, 64: 1156.04, 128: 1174.77, 256: 1216.47, 512: 1375.53, 1024: 1484.51, 4096: 2488.93, 65536: 2794.72,
5	16: 1141.27, 32: 1143.82, 64: 1205.88, 128: 1190.32, 256: 1220.1, 512: 1308.65, 1024: 1479.77, 4096: 2539.12, 65536: 2786.18,
6	16: 1156.82, 32: 1149.53, 64: 1185.07, 128: 1250.63, 256: 1238.16, 512: 1374.59, 1024: 1488.79, 4096: 2504.36, 65536: 2797.56,
7	16: 1137.2, 32: 1138.08, 64: 1185.11, 128: 1202.18, 256: 1219.19, 512: 1301.78, 1024: 1472.45, 4096: 2536.7, 65536: 2783.92,
8	16: 1146.43, 32: 1145.92, 64: 1157.19, 128: 1385.87, 256: 1227.28, 512: 1311.49, 1024: 1483.88, 4096: 2497.44, 65536: 2781.94,
9	16: 1140.84, 32: 1136.68, 64: 1166.86, 128: 1174.93, 256: 1220.64, 512: 1311.32, 1024: 1490.28, 4096: 2491.4, 65536: 2766.15,
10	16: 1135.21, 32: 1147.02, 64: 1151.73, 128: 1178.68, 256: 1264.32, 512: 1319.99, 1024: 1480.97, 4096: 2488, 65536: 2779.6,

A.1.7 Allocation Latency Benchmark of Memhook (Glibc backtrace, depth 20)

```
> taskset -c 5 ./test_alloc_time
> sudo python mem-hook.py -se 1000000 -hf malloc free -bm glibc -mb 20 -p <
    ↪ PID>
```

Run	Allocation Time (ns) per Size (B)
1	16: 1932.83, 32: 1936.37, 64: 1948.28, 128: 1975.61, 256: 2018.51, 512: 2102.2, 1024: 2291.82, 4096: 3273.19, 65536: 3537.86,
2	16: 1930.62, 32: 1935.19, 64: 2082.53, 128: 1990.66, 256: 2042.8, 512: 2112.96, 1024: 2297.86, 4096: 3538.13, 65536: 3556.91,
3	16: 1933.18, 32: 1942.53, 64: 1987.47, 128: 1972.84, 256: 2028.47, 512: 2230.43, 1024: 2285.94, 4096: 3281.59, 65536: 3537.67,
4	16: 1935.05, 32: 1936.94, 64: 2092.46, 128: 2017.4, 256: 2029.64, 512: 2131.01, 1024: 2293.19, 4096: 3321.68, 65536: 3566.22,
5	16: 1947.22, 32: 1936.35, 64: 1951.92, 128: 2143.25, 256: 2038.91, 512: 2108.93, 1024: 2340.19, 4096: 3355.29, 65536: 3523.47,
6	16: 1938.55, 32: 1931.86, 64: 1985.26, 128: 1979.9, 256: 2102.43, 512: 2106.37, 1024: 2277.53, 4096: 3309.23, 65536: 3535.72,
7	16: 1951.19, 32: 1942.84, 64: 1968.01, 128: 1992.78, 256: 2023.55, 512: 2103.62, 1024: 2289.54, 4096: 3275.21, 65536: 3555.63,
8	16: 1928.19, 32: 1931.98, 64: 2320.84, 128: 1969.93, 256: 2017, 512: 2176.61, 1024: 2469.94, 4096: 3307.57, 65536: 3524.77,
9	16: 1940.65, 32: 1944.11, 64: 2050.78, 128: 2006.82, 256: 2040.14, 512: 2118.28, 1024: 2277.32, 4096: 3277.69, 65536: 3574.93,
10	16: 1927, 32: 1934.79, 64: 2113.35, 128: 1974.77, 256: 2019.96, 512: 2097.4, 1024: 2269.61, 4096: 3290.52, 65536: 3536.92,

A.2 Test Applications

A.2.1 Allocation Latency Benchmark

```
#include <algorithm>
#include <chrono>
#include <iomanip>
#include <iostream>
#include <malloc.h>
#include <thread>
#include <unistd.h>
#include <vector>

size_t static const NUM_ALLOCS{100000};

// 16B, 32B, 64B, 128B, 256B, 512B, 1KB, 4KB, 64KB, 1MB
size_t alloc_sizes[] = {16, 32, 64, 128, 256, 512, 1024, 4096, 65536};

/* Run several malloc and return the time it took */
double run_test(size_t alloc_size, size_t num_allocs) {
    void* allocations[num_allocs];
    auto const start{std::chrono::high_resolution_clock::now()};

    for (size_t i = 0; i < num_allocs; i++) {
        allocations[i] = malloc(alloc_size);
    }

```

```
24      auto const end{std::chrono::high_resolution_clock::now()};
25      auto const duration{
26          std::chrono::duration_cast<std::chrono::nanoseconds>(end - start)};
27
28      // Free the allocated memory
29      for (void* ptr : allocations) {
30          free(ptr);
31      }
32
33      std::this_thread::sleep_for(std::chrono::seconds(1));
34
35      return duration.count();
36  }
37
38  void print_results(std::vector<double> const& results) {
39      int const COL_WIDTH{15};
40
41      std::cout << std::left << std::setw(COL_WIDTH) << "Size (B)" << std::
            ↪ left
42                << std::setw(COL_WIDTH) << "Time (ns)" << std::endl;
43
44      for (size_t i = 0; i < results.size(); i++) {
45          size_t const size{alloc_sizes[i]};
46          double const time{results[i]};
47
48          std::cout << std::left << std::setw(COL_WIDTH) << size << std::left
49                    << std::setw(COL_WIDTH) << time << std::endl;
50      }
51
52      std::cout << "{";
53      for (size_t i = 0; i < results.size(); i++) {
54          size_t const size{alloc_sizes[i]};
55          double const time{results[i]};
56          std::cout << size << ": " << time << ", ";
57      }
58      std::cout << "}" << std::endl;
59  }
60
61  int main() {
62      std::cout << "Press Enter to start the tests..." << std::endl;
63      std::cin.get();
64
65      // Disable fastbin caching (makes allocations behave more predictably)
66      mallopt(M_MXFAST, 0);
67      // Force memory to be returned to the OS immediately
68      mallopt(M_TRIM_THRESHOLD, 0);
69
70      // Run allocations before the tests to not get initialisation overhead
71      run_test(256, 10000);
72
73      std::vector<double> results{};
74      std::transform(std::begin(alloc_sizes), std::end(alloc_sizes),
75                     std::back_inserter(results), [](size_t size) {
76                         return run_test(size, NUM_ALLOCS) / NUM_ALLOCS;
77                     });
78      print_results(results);
79      std::cin.get();
```

```
80 };
```

Listing A.1: A simple C++ tool that measures the average time per allocation for varying allocation sizes using malloc.