From Allocation to Accumulation: Investigating Memory Leaks and Other Causes of Growing Memory Requirements on Linux

MASTER THESIS

Juliane Friedmann

Submitted on 2 March 2026

Friedrich-Alexander-Universität Erlangen-Nürnberg
Faculty of Engineering, Department Computer Science
Professorship for Open Source Software

Supervisor:
Prof. Dr. Dirk Riehle, M.B.A.
Jan Grembler, Dipl.-Inf., IQSIGHT Engineering GmbH

Friedrich-Alexander-Universität
Faculty of Engineering

Declaration of Originality

I confirm that the submitted thesis is original work and was written by me without further assistance. Appropriate credit has been given where reference has been made to the work of others. The thesis was not examined before, nor has it been published. The submitted electronic version of the thesis matches the printed version.

Erlangen, 2 March 2026

License

Erlangen, 2 March 2026

Abstract

Long-running embedded systems developed in C and C++ are prone to memory leaks and unbounded memory growth due to manual memory management. Traditional memory analysis and leak detection tools are often not suitable, as they require system restarts or recompilation, and impose excessive overhead on resource-constrained hardware. This thesis presents the design and implementation of *MemScope*, a lightweight, runtime-attachable tracing framework for embedded Linux systems. *MemScope* leverages Extended Berkeley Packet Filter (eBPF)-based user space probes to monitor heap allocation and deallocation events without modifying the target application. The framework can be attached to processes that are already running and monitor memory activity from the point of intervention. The framework reconstructs the memory state, enables time-resolved growth observability, and maps allocations to their respective call stacks for precise source-level attribution.

The system was evaluated on a resource-constrained embedded platform under controlled workloads. The results demonstrate bounded memory consumption, manageable runtime overhead, and stable operation during extended monitoring sessions. *MemScope* provides structured runtime allocation data that significantly reduces manual diagnostic effort.

Contents

1 Introduction **1**
- 1.1 Motivation . . . 1
- 1.2 Objective . . . 2
- 1.3 Scope of Work . . . 2
- 1.4 Outline . . . 3

2 Literature Review **5**
- 2.1 Classes of Memory Analysis Tools . . . 5
  - 2.1.1 Compile-Time Instrumentation . . . 5
  - 2.1.2 Dynamic Binary Instrumentation (DBI) . . . 6
  - 2.1.3 Library Interposition . . . 7
  - 2.1.4 Kernel-Level and System Tracing . . . 9
- 2.2 Orthogonal Design Dimensions and Comparative Analysis . . . 10

3 Requirements **13**
- 3.1 Observation Goals . . . 13
- 3.2 Functional Requirements . . . 13
- 3.3 Non-Functional Requirements . . . 14

4 Architecture **17**
- 4.1 Observation Model . . . 17
  - 4.1.1 Ambiguity of Process Memory Consumption . . . 17
  - 4.1.2 Memory Behaviour Alone Lacks Causality . . . 19
  - 4.1.3 Observation Constraints . . . 19
- 4.2 System Overview . . . 20
- 4.3 Data Model . . . 21
- 4.4 Processing Model . . . 22
  - 4.4.1 State Reconstruction . . . 23
  - 4.4.2 Incomplete Information Handling . . . 23
  - 4.4.3 Online Processing . . . 25
  - 4.4.4 Persistence Strategy . . . 26
- 4.5 Technical Implications for the Tracing Mechanism . . . 27

5 Design and Implementation **29**
5.1 Introduction to eBPF . . . 29
5.1.1 Execution Model . . . 30
5.1.2 Attachment Mechanisms . . . 31
5.1.3 Data Exchange . . . 33
5.2 eBPF Integration and Component Split . . . 35
5.2.1 Choice of eBPF Tooling Ecosystem . . . 35
5.2.2 Separation of Kernel and User Space State . . . 36
5.3 Tracing Strategy . . . 37
5.4 State Reconstruction and Correlation Strategy . . . 40
5.5 Incremental Context-Based Memory Analysis . . . 48
5.6 Data Transport and Backpressure Handling . . . 54
5.7 Persistence Strategy . . . 59
5.8 Degradation Handling and System Robustness . . . 60

6 Evaluation **65**
6.1 Evaluation Setup . . . 65
6.2 Evaluation of Functional Requirements . . . 65
6.3 Evaluation of Non-Functional Requirements . . . 68
6.4 Practical Use Cases . . . 74
6.5 Summary . . . 75

7 Conclusion **77**
7.1 Summary . . . 77
7.2 Limitations . . . 78
7.3 Future Work . . . 78

Appendices **79**
A Kernel Hook Implementation . . . 81
B User Space Ring Buffer Callback Implementation . . . 84
C User Space State Management . . . 85
D Offline Stack Trace Resolution . . . 88

References **91**

List of Figures

4.1 Illustration of typical heap allocation behaviours 18
4.2 Architectural overview 20
4.3 Conceptual data model 21
4.4 Allocation lifetime model 22
4.5 Conceptual illustration of context-based statistics. 26

5.1 High-level execution model of MemScope. 35
5.2 Kernel to user space event flow and correlation logic in *MemScope* for the processing of one single event. The blue arrows in the user space section indicate the optimal processing path, i.e., the scenario in which no temporal inconsistencies or incomplete information occur. 43
5.3 Data transport pipeline and backpressure decoupling: per-CPU batching in the kernel (512 events per batch), emission via the BPF ring buffer, persistence in user space, and expansion into single events for queue-based worker processing. 56

6.1 Memory growth over time for a stable and a leaking call site. . . . 67

List of Tables

2.1 Comparison of tracing approaches. 11

5.1 User space maps used in *MemScope*. 46
5.2 BPF kernel maps used in *MemScope*. 47
5.3 Controlled degradation and monitoring mechanisms in *MemScope*. 61

6.1 Runtime overhead measurement results. 69
6.2 Max RSS of *MemScope* over increasing monitoring durations, demonstrating bounded memory usage. 70

Listings

5.1 Syntax of a uprobe hook. 38
5.2 Uprobe attach in the user space. 39
5.3 Implementation of a stack trace map. 41
5.4 Using BPF helper functions to access BPF maps. 42
5.5 BPF helper function to retrieve stack trace and corresponding stack identifier. 48
5.6 Retrieving the virtual start address from the ELF file. 52
5.7 Example of an offline-resolved stack trace for one stack identifier. Frames from the target application resolve to file and line numbers, whereas vendor `libc` frames remain unresolved due to missing debug symbols. 52
5.8 User space implementation of the BPF ring buffer. 55
5.9 Kernel space batching and insertion to ring buffer. 57
A.1 Malloc return hook. Kernel implementation. 81
B.1 Ring buffer callback function in the user space. 84
C.1 Excerpt of user-space worker thread that is responsible for state management. 85
D.1 Python script for offline stack trace resolution. 88

Acronyms

AArch64 64-bit Arm

ASan AddressSanitizer

ASLR Address Space Layout Randomization

API Application Programming Interface

BCC BPF Compiler Collection

BPF Berkeley Packet Filter

CPU Central Processing Unit

CSV Comma-Separated Values

DBI Dynamic Binary Instrumentation

DoS Denial-of-Service

DWARF Debugging With Attributed Record Formats

eBPF Extended Berkeley Packet Filter

ELF Executable and Linkable Format

GCC GNU Compiler Collection

glibc GNU C Library

GWP Guarded Pool Allocator

JIT Just-In-Time

LKM Linux Kernel Module

Max RSS Maximum Resident Set Size

MSan MemorySanitizer

libc standard C library

LLVM Low Level Virtual Machine

PID Process Identifier

SSH Secure Shell Protocol

tcmalloc Thread-Caching Malloc

TGID Thread Group Identifier

TSan ThreadSanitizer

USDT User Statically Defined Tracing

1 Introduction

1.1 Motivation

Embedded software systems are frequently developed in C or C++. Since these languages lack automatic memory management, it must be handled manually. In large legacy codebases, memory management errors often remain undetected for extended periods. This can lead to memory leaks or unbounded memory growth, resulting in degraded performance, increased resource consumption, and system instability. Furthermore, such vulnerabilities can be exploited for Denial-of-Service (DoS) attacks by intentionally exhausting system resources.
Although migrating to memory-safe languages could address these issues, it is typically not a viable option given the scale and complexity of such codebases. Consequently, analysing runtime memory behaviour is essential to ensure long-term stability of production systems.

The traditional definition of a memory leak, that is, memory not released at program termination, assumes a finite execution lifetime. However, many embedded systems are designed for continuous operation and may never terminate under normal conditions. In such environments, program termination cannot be used as a reference point.
The relevant question for long-running systems is not whether memory is eventually released, but whether memory usage stabilises over time within the operational limits. Analysis tools must therefore operate with incomplete knowledge of the allocation history and classify behaviour as either bounded or unbounded growth rather than relying solely on the binary classification of memory leak versus no memory leak. For the purpose of this thesis, the following distinction is made:

- **Memory Leak:** Memory that remains allocated at program termination due to a missing corresponding deallocation.
- **Memory Growth:** A sustained increase in the total actively allocated memory during execution, regardless of whether individual allocations are eventually released. This may originate from leaks, delayed deallocations,

caching or fragmentation.

On embedded targets, even bounded memory growth can be critical due to strictly limited physical memory and Central Processing Unit (CPU) capacity. High workloads and real-time requirements dictate that any monitoring process must have only minimal influence on the target system. The tracing approach must operate within tight resource budgets, avoid inducing system instability, and function without requiring restarts, recompilation, or source code modification. As the target platform is based on standard Linux facilities, the solution must rely on native system mechanisms rather than specialised runtime environments or modified toolchains.

While various tools, ranging from source code instrumentation to library interposition (see Chapter 2) exist, most are unsuitable for resource-constrained embedded environments due to high overhead or the requirement for full-program observation.
The current approach to addressing this issue involves placing manual wrappers around memory-relevant code blocks and manually analysing the resulting log files. This process is time-intensive, requires significant maintenance effort, and remains a largely manual diagnostic challenge.

1.2 Objective

To enhance the stability of long-running embedded systems and reduce diagnostic effort, this thesis aims to design and implement a lightweight, runtime-attachable tracing framework, hereafter referred to as *MemScope*. The framework is intended to serve as an observability tool that extracts and processes heap data in embedded Linux environments under production constraints. By providing developers with structured data on runtime allocation patterns, *MemScope* facilitates the effective investigation of memory growth. A key requirement is that the framework must operate without any modifications to, or recompilation of, the target application.

1.3 Scope of Work

The scope of this work is limited to the runtime observation of dynamic heap memory behaviour in user space processes. The framework is designed for process-targeted tracing in embedded Linux environments and focuses on analysing allocation and deallocation behaviour under production constraints.

The framework does not provide automated diagnosis, formal memory correctness guarantees, static code verification, or debugger functionality. It does not attempt to prove the absence of memory leaks. The implementation demonstrates

feasibility of lightweight runtime-attached memory behaviour analysis rather than delivering a fully integrated production debugging suite.

1.4 Outline

Chapter 2 briefly reviews existing approaches for memory analysis on Linux systems and categories them by instrumentation level and operational characteristics.

Chapter 3 defines the functional and non-functional requirements that the tracing framework must satisfy in embedded production environments.

Chapter 4 presents a solution-independent architecture, describing the observation model, system components, data and processing model, as well as the resulting implications for the selected tracing mechanism.

Chapter 5 introduces eBPF as the selected tracing mechanism and details the resulting system design and implementation of the framework.

Chapter 6 evaluates the framework against the requirements defined in Chapter 3.

Chapter 7 concludes the thesis by summarising the key contributions, discussing limitations, and outlining directions for future work.

2 Literature Review

After defining the problem scope and system requirements, this chapter surveys existing approaches for analysing memory behaviour in software systems. A variety of techniques have been proposed both in research and practice, ranging from compile-time instrumentation to runtime tracing and statistical monitoring. No single technique dominates all use cases, as each approach involves trade-offs in terms of precision, overhead, intrusiveness, and deployment constraints.

Since this thesis focuses on investigating memory behaviour on resource-constrained embedded Linux systems, the review is intentionally scoped to techniques applicable to native user space programs, specifically dynamic approaches.

The objective of this chapter is to systemically map the solution space, classify existing methodological approaches, and position the proposed tracing framework within this broader landscape.

2.1 Classes of Memory Analysis Tools

Based on the surveyed literature and representative tools, memory analysis approaches can be grouped into the following methodological classes.

2.1.1 Compile-Time Instrumentation

Compile-time instrumentation modifies a binary by integrating additional logic into the execution path during the compilation process. The executable then contains built-in runtime verification mechanisms.

AddressSanitizer (ASan) is an example of compile-time instrumentation (Serebryany et al., 2012). It monitors memory accesses and maintains a shadow memory, containing the status of every byte in the actual memory. The status is used to detect invalid memory accesses, such as buffer overflows or use-after-free errors. Additionally, the compiler places safety gaps (redzones) around heap and stack objects. These gaps are marked with an invalid status in the shadow memory. For every memory access, the shadow memory is first inspected to see if the target

address is valid. The access of a memory address marked as invalid in the shadow memory results in a controlled crash of the program. Before terminating, ASan generates a detailed error report of the incident. ASan is widely adopted in modern software development and is integrated into major compiler toolchains such as Low Level Virtual Machine (LLVM) and GNU Compiler Collection (GCC) (Serebryany et al., 2012).
In contrast to ASan, MemorySanitizer (MSan) monitors if a memory address is initialised before it can influence the program behaviour. It also keeps a shadow memory but in contrast to ASan it maps every bit of the actual memory to a bit in the shadow memory to keep exact track of valid and invalid memory addresses (Stepanov & Serebryany, 2015).
Similarly, ThreadSanitizer (TSan) instruments memory accesses to detect data races in multithreaded programs (Serebryany & Iskhodzhanov, 2009).
ASan, MSan, and TSan are part of the LLVM-based sanitizers, which also include UndefinedBehaviorSanitizer (UBSan). They all share a common compiler-instrumentation architecture.

Guarded Pool Allocator (GWP)-ASan is a newer, sampling based version of ASan. It reduces runtime overhead by instrumenting only a subset of allocations, illustrating the trade-off between detection coverage and performance (Serebryany et al., 2024).

The advantage of compile-time instrumentation lies within its high precision and relatively low runtime overhead. The original ASan paper reports a slowdown of 73 %. However, compile-time instrumentation requires recompilation and therefore the restart of the application, while also increasing the size of the executable due to added instructions.

2.1.2 Dynamic Binary Instrumentation (DBI)

Dynamic Binary Instrumentation (DBI) is a method that inserts analysis code into the binary instruction code of a program during execution. It intercepts the execution stream, breaks it into basic blocks (instruction sequences), and rewrites them by inserting additional analysis code. These instrumented code blocks are then stored in a code cache.
The code cache is used to avoid extreme overhead due to continuously interpreting or recompiling instruction sequences. Whenever the program executes a block that has already been instrumented and placed in the code cache, it directly accesses the block from the code cache instead of the original binary.
To further increase performance the code blocks within the code cache are directly linked. This allows the control flow to proceed without returning to the central instrumentation dispatcher after a block in the code cache was executed.

The prominent DBI frameworks include Valgrind (Nethercote & Seward, 2007),

DynamoRIO (Bruening et al., 2003), and Intel Pin (Luk et al., 2005).

Valgrind is a framework for heavyweight DBI. Valgrind itself runs on the host CPU, while the execution of the program is intercepted, the machine code converted into an intermediate representation (IR) and executed within a synthetic CPU environment. Before IR are retranslated to host instructions, analysis logic is inserted. The instrumented code is stored in a code cache.
During execution, Valgrind mirrors the program's address space and stores metadata for each byte. This is referred to as shadow memory. Analysis tools like Memcheck (Seward & Nethercote, 2005) use this shadow memory and its metadata to detect memory leaks and similar issues, e.g., invalid memory accesses.
Instrumenting every instruction provides very high precision and granularity, but it typically introduces substantial runtime overhead. This is attributed to the computational cost of maintaining the synthetic environment and the shadow memory. The actual slowdown depends on the workload and tool configuration (Nethercote & Seward, 2007; Vasquez & Simmonds, 2021, pp. 593–594).

DynamoRIO is a lightweight DBI framework. Unlike Valgrind, it separates the applications original code into blocks and copies them into a code cache. Code then is exclusively executed from the code cache. While the code is copied into the code cache, analysis code can be inserted. Blocks that are executed in succession are aggregated into traces and chached.
As DynamoRIO does not use heavy IR, it reduces overhead while still enabling fine-grained instruction-level monitoring (Bruening et al., 2003).

The primary advantage of DBI is its high precision and flexibility. But, as instrumentation occurs during execution, it adds additional runtime overhead to the program, typically limiting DBI-based tools to offline or debugging analysis scenarios. However, DBI does not require the source code or any modification of the executable file on the disk.

2.1.3 Library Interposition

Library interposition replaces or intercepts function calls at the dynamic linking level. When intercepting a function call, an alternative implementation can be provided and executed instead. Consequently, a function in another shared object can be explicitly replaced. On Unix systems, this is typically realised through the dynamic linker, which resolves symbols at load time and can be instructed to search in custom libraries for symbols or functions, before doing so in the standard system libraries. Library interposition does not require any modification of the source code or binary ('The GNU C Library', 2026; Kerrisk, 2025b).

On Linux, the `LD_PRELOAD` environment variable enables library interposition. It instructs the dynamic loader to load a specific shared object and its symbol definitions before the standard libraries. This allows to replace symbols that

are also present in other libraries with custom code and enables transparent interception without recompilation of the target binary ('The GNU C Library', 2026; Kerrisk, 2025b).

`mtrace` is another library interposition mechanism. It is provided by the GNU C Library and instruments memory allocation functions to record allocation and deallocation events. The tracing can be enabled through environment variables and then logs every event to a file. As `mtrace` logs every event, its overhead is primarily determined by I/O. Therefore, performance will increase for programs with a high event rate. At the same time, compared to DBI, `mtrace` is a rather lightweight tool. It is important to note that the allocation log can only be analysed after program termination. Consequently, `mtrace` can only be used for offline leak detection. Moreover, it offers only limited contextual information into allocation patterns ('The GNU C Library', 2026; Kerrisk, 2025c; Vasquez & Simmonds, 2021, p. 592).

An alternative memory allocator is Google's `Thread-Caching Malloc (tcmalloc)`. It replaces the original memory allocator with an implementation that provides built-in profiling. In addition, `tcmalloc` optimises performance by using thread-local caches to minimise lock-contention. This makes it significantly more efficient that the standard GNU C Library (glibc) allocator in multi-threaded environments.
As `tcmalloc` records stack traces at specific intervals instead of intercepting every individual allocation, it is a sampling-based approach for continuous memory profiling with negligible overhead. Therefore, it can be used in high-load production systems.
Its deployment is highly flexible, as it can be linked during compilation or injected into existing, unmodifiable binaries using the `LD_PRELOAD` environment variable ('TCMalloc : Thread-Caching Malloc', 2024).

Library interposition-based tools provide a low integration barrier as no recompilation or full binary translation is required. Therefore, library interposition approaches introduce significantly less overhead than DBI-based tools. However, when using library interposition tools, the visibility is limited to intercepted functions and in general the abstraction level is much higher than for DBI. Library interposition relies on linking and can be bypassed if statically linked binaries are used or when allocation mechanisms are implemented outside the standard libraries. Furthermore, when standard library symbols are replaced, it is imperative to ensure proper implementation, with consideration for all other symbols and functions that may invoke this implementation. It is inevitable that discrepancies in implementation will result in heap corruption and program crashes.

2.1.4 Kernel-Level and System Tracing

Kernel-level tracing instruments the kernel itself for observation of system behaviour. Since the kernel manages process lifecycles and hardware resources, it is a central point of observation in the system. Relevant events, such as memory allocations or deallocations, can be observed by attaching instrumentation logic to predefined execution points within the operating system or the user space. Modern tracing mechanisms provide multiple points across these layers to attach customised analysis code. This makes kernel-level tracing a flexible approach without requiring modification of the application source code.

Originally introduced as a performance analysis framework for Linux, `perf` provides access to hardware performance monitoring units as well as kernel and user space trace events. As `perf` is typically used in performance profiling scenarios recording instruction pointers and stack traces at a defined frequency, its primary mechanism is sampling. However, it can also collect event-based traces (Vasquez & Simmonds, 2021, pp. 653–657; 'Perf: Linux Profiling with Performance Counters', 2024).

The Linux kernel provides an internal tracing framework, called `ftrace`. It is designed for function-level tracing and other instrumentation mechanisms. When enabled, it can record and timestamp kernel function calls (Vasquez & Simmonds, 2021, p. 660).

Tracepoints are predefined execution points, called hooks or probes, placed manually by kernel developers at key locations in the source code. As they are statically implemented, they offer more stability than dynamic probes, such as kprobes. When a tracepoint is triggered, structured event data is generated and made available to tracing frameworks such as `perf` or eBPF (Vasquez & Simmonds, 2021, p. 665).

These concepts build the foundation on which kernel tracing mechanisms, such as eBPF are built.
Extended Berkeley Packet Filter (eBPF) is a programmable tracing and instrumentation framework integrated into the Linux kernel. Users can attach small, verified programs to various hook points in both kernel and user space, including tracepoints, kprobes and uprobes. When triggered, these programs execute in kernel context and can collect event data specified by the developer. The data is then sent to user space programs for further processing. This approach allows flexible runtime tracing with low overhead. It does not require recompilation or source code modification of the target binary and eBPF programs can be attached to running processes without restart (Billimoria, 2024, p. 274).
A more detailed description of eBPF is provided in Chapter 5.

Recent research has explored eBPF-based approaches for dynamic memory leak

detection in Linux environments. A 2024 study proposes a method that combines eBPF with a Linux Kernel Module (LKM). In this approach, eBPF uprobes are attached to user space memory allocation functions to collect metadata.
To determine whether allocated memory remains actively used in long-running processes, the approach proposes the use of a LKM to inspect the accessed bit on page table entries at hardware level. This bit is periodically reset and checked. If the accessed bit remains false after reset, the page is classified as unused. The system aims to distinguish between temporarily retained memory and actual leaks without interrupting the execution of the target application.
However, relying on a dedicated LKM introduces additional deployment complexity and may limit applicability in production environments where kernel modifiactions are restricted (Xu & Chen, 2024).

Kernel-level tracing enables low-overhead runtime instrumentation without recompilation or binary rewriting. It is suitable for monitoring already deployed processes. However, observability is limited to available hook points and exposed kernel data. Applicability depends on kernel capabilities and configuration.

2.2 Orthogonal Design Dimensions and Comparative Analysis

The previous sections structured the solution space according to methodological classes of analysis approaches. However, these approaches exhibit characteristics that are independent of their underlying instrumentation mechanism but rather influence the applicability in practice. The following section therefore introduces orthogonal design dimensions that allow systematic comparison across classes.

The most fundamental trade-off among the examined approaches concerns precision versus runtime overhead. Techniques that provide instruction-level visibility and complete allocation tracking typically suffer from substantial execution overhead, whereas lightweight or sampling-based mechanisms suffer from partial observability.

The following design dimensions can be identified:

- Precision vs. Runtime Overhead
- Intrusiveness and Source Code Requirements
- Runtime Attachment Capability
- Observability Scope
- Intended Usage Context
- Embedded System Suitability

These dimensions provide a structural basis for comparison. Table 2.1 summarises the trade-offs discussed in this chapter.

Dimension	**Compile-Time Instrumentation (ASan)**	**DBI (Valgrind)**	**Library Interposition**	**Kernel-Level Tracing (eBPF)**
Precision	High	Very High	Medium	High
Source Required	Yes	No	No	No
Runtime Attach	No	No	No	Yes
Overhead	Moderate-High	Very High	Low-Moderate	Low
Embedded Suitability	Limited	Low	Moderate	High
Typical Usage Context	Testing/CI	Debugging	Debugging/-Monitoring	Monitoring/-Diagnostics

Table 2.1: Comparison of tracing approaches.

3 Requirements

This chapter outlines the requirements for the tracing framework. First, the observation goals are introduced. The requirements are then categorised as either functional or non-functional.

3.1 Observation Goals

The goal of this tracing framework is to make long-term memory behaviour observable instead of formally proving the existence of a memory leak. As discussed in Section 1.1, memory growth can have many different root causes. In order to distinguish persistent memory growth from legitimate temporary allocation patterns, insight into allocation lifetimes and recurring allocation patterns is required. Therefore, the system shall focus on displaying measurable characteristics that allow for interpreting the memory behaviour over time.

3.2 Functional Requirements

The functional requirements specify the capabilities that the tracing framework must provide.

Dynamic Observation Control

The system shall allow attaching to and detaching from a running process without requiring a restart, as applications in embedded systems typically run for extended or indefinite periods and memory growth may only become observable during operation. Consequently, instrumentation must be possible at the moment anomalous behaviour is suspected without disrupting the running system.

Allocation Event Acquisition

The system shall capture allocation and deallocation events and maintain the relationship between both operations. To reconstruct memory behaviour, alloc-

ation lifetimes must be determined. Without pairing allocation and deallocation events, long-term memory growth cannot be distinguished from temporary allocations.

Allocation Origin Identification

The system shall provide the calling context responsible for each allocation and deallocation event. To distinguish between intended allocation behaviour and memory growth, the origin of an allocation must be identifiable; therefore, the full call chain must be available rather than only the immediate caller in order to reconstruct the complete execution path that led to the allocation.

Runtime Memory State Inspection

The system shall provide the currently active allocations and their lifetimes at any point during the tracing. Only allocations that occurred after the tracing began and remain active can be reported. To correlate behaviour with system conditions, developers must be able to inspect the current memory state while the application is running.

Memory Growth Observability

The system shall provide time-resolved allocation data enabling the analysis of memory consumption trends. Identifying memory growth requires observing behaviour over time. It cannot be determined from a single snapshot of the memory state.

3.3 Non-Functional Requirements

Non-functional requirements are defined as the quality attributes and operational characteristics of a system.

Completeness of Observation

The system shall not lose any allocation or deallocation events after the tracing framework has been attached to the target process. An incomplete event stream would lead to incorrect representation of the runtime memory behaviour and may falsely indicate memory growth or hide actual memory growth.

Performance

The system shall impose only limited and acceptable runtime overhead on the target process. Tracing must not alter the execution behaviour beyond acceptable limits; otherwise, the observed memory behaviour would not reflect the real

system behaviour.
The system shall operate with bounded memory consumption, regardless of tracing duration. The tracing framework must remain stable during long-running observations and cannot itself lead to memory exhaustion. This requirement applies only to the runtime memory used by the tracing framework and does not include persistent storage of traced events.

Concurrency Safety

The system shall trace the memory behaviour of multi-threaded applications in a thread-safe manner. Concurrent allocations require synchronisation and the avoidance of race conditions to ensure consistent memory state reconstruction.
The system shall trace events originating from multiple CPU cores. Allocations and deallocations can occur simultaneously on different processors and therefore require correct ordering and correlation of events.

Robustness

The system shall maintain a consistent representation of the memory state even when tracing begins after the start of the program. The allocation history may be partial when attaching to a running process. Despite this incomplete history, the tracing framework must still provide a meaningful and internally consistent representation of the observed memory behaviour.

Operational Stability

The system shall operate reliably during extended observation periods without accumulating internal inconsistencies or degrading operational behaviour over time.

Operability

The system shall require minimal configuration and no modification of the target application. Diagnostic tools must remain usable during development, testing, and field operation without introducing additional engineering effort. In production environments, the target application often cannot be rebuilt or restarted.

Portability

The system shall operate across different devices and software variants within the same Linux-based product family without requiring specific adaptation. A tracing framework is intended for deployment in multiple hardware generations and software configurations. Integration per-device would significantly limit the practical usability in production environments.

The system shall avoid dependence on programming language specific instrumentation and instead operate on common runtime allocation interfaces. This allows analysing of applications implemented in different programming languages without requiring language-specific integration.

4 Architecture

This chapter presents the architecture of the proposed tracing framework, irrespective of the target platform. The focus of this chapter lies on the concept rather than its concrete realisation.
First, the observation model is defined, specifying which aspects of runtime memory behaviour must be observable in order to determine whether a system exhibits memory growth. Next, a high-level system overview introduces the components required to acquire, process, and provide this data. Section 4.3 then specifies the information handled by the system. Finally, the processing model explains how the collected data is processed to obtain a consistent and interpretable representation of memory behaviour.

4.1 Observation Model

In order to determine whether a system exhibits memory growth, it must first be clarified how memory growth can be identified in general. Programs often produce short-term memory peaks during execution, where large amounts of memory are temporarily allocated. Such peaks do not represent a memory leak and are not necessarily an indicator of memory growth. Only when the increased memory level persists over time does the allocated memory indicate memory growth. Therefore, memory behaviour must be evaluated over time as a sequence of state changes rather than at a single point in time. Memory growth is inherently a temporal phenomenon.

While the evaluation of memory growth over time is necessary, it is insufficient for a meaningful interpretation in itself. The subsequent sections provide a more detailed exposition on this matter.

4.1.1 Ambiguity of Process Memory Consumption

Figure 4.1 illustrates typical behaviours of dynamic heap allocations over time. The figure distinguishes between live memory (actively allocated memory) and reserved heap size (total heap space reserved by the allocator).

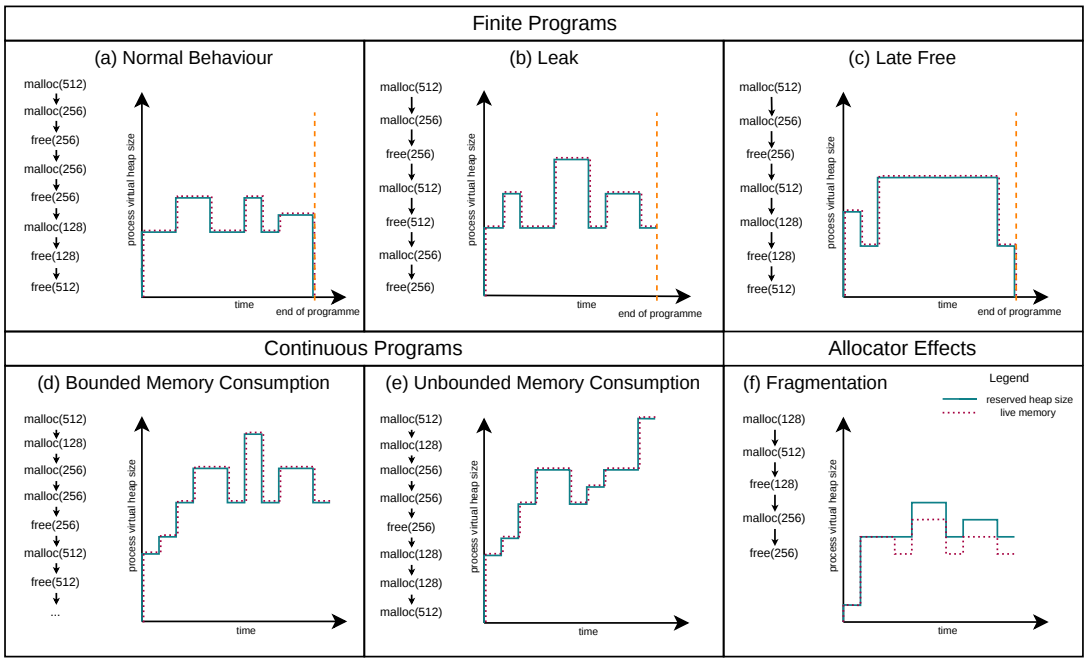

Figure 4.1: Illustration of typical heap allocation behaviours

- (a) Normal Behaviour - Finite Program
  Allocations and deallocations are balanced. Memory temporarily increases during execution but at program termination, all allocated memory is released. No persistent memory growth occurs.
- (b) Leak - Finite Program
  At least one allocation is never released. Live memory remains above zero at program termination and is permanently lost due to missing deallocation. Indicates a classical memory leak.
- (c) Late Free - Finite Program
  An allocation is held for an extended period of time so that memory appears elevated for a long time and may temporarily appear as a leak, but the memory is released before program termination.
- (d) Bounded Memory Consumption - Continuous Program
  Memory increases initially but converges to a stable level. Behaviour is intended and expected and is sustainable for continuous systems.
- (e) Unbounded Memory Consumption - Continuous Program
  Memory continuous to increases over time. No convergence towards a stable level. The behaviour leads to long-term resource exhaustion. This figure represents problematic memory growth in a continuous system.
- (f) - Fragmentation - Allocator Effects

Memory becomes fragmented over time. In theory, enough memory exists, but not in a contiguous memory region suitable for subsequent allocations. As a result, the reserved heap size increases despite balanced allocation and deallocation behaviour. Growth originates from allocator behaviour, not allocation imbalance.

Observing total memory consumption alone is insufficient because it does not allow a reliable interpretation of the system behaviour. Different system behaviours can produce a similar observed increase in heap memory usage. It follows that memory consumption alone is not a reliable indicator of unintended memory growth. Allocation lifetimes must be considered in the observation.

Furthermore, these phenomena do not occur in isolation. In real systems, fragmentation, late deallocations, and genuine memory growth may appear simultaneously, making the interpretation of aggregated memory metrics even more ambiguous.

4.1.2 Memory Behaviour Alone Lacks Causality

Even if allocation lifetimes are known, the behaviour cannot be interpreted without knowing which program location is responsible for the allocation. Allocations that are active over an extended period of time may result from intentional design decisions, such as caching mechanisms or data accumulation strategies.

Without identifying the originating allocation context or call site, it is not possible to distinguish between intended growth and faulty behaviour. In this context, causality is defined as the association of each allocation and its originating code location. This corresponds to the functional requirement of allocation origin identification defined in Section 3.2.

Only the combination of temporal and causal information enables a meaningful interpretation of memory behaviour. Since both pieces of information arise at the moment an allocation event occurs, individual allocation events must be observed and the memory state reconstructed from them.

4.1.3 Observation Constraints

The use of a reconstruction-based approach imposes strict constraints on the observation process. Since the memory state is derived exclusively from allocation and deallocation events, missing events would directly corrupt the reconstructed state. For example, if one deallocation event is not observed, the corresponding allocation would appear to remain active indefinitely and could be falsely interpreted as leak-like behaviour. Consequently, the observation cannot rely on statistical sampling, but must capture a continuous stream of events without loss.

This motivates the non-functional requirement for completeness of observation introduced in Chapter 3.3.

However, even with a continuous event stream, the observation window may not cover the full lifetime of all allocations. This limitation arises from the nature of runtime observation. When attaching the tracing framework to an already running process, some allocations may have been alive before the observation began. Similarly, detaching the tracing system before the observed process terminates inevitably results in missing the final deallocations.

As a result, the system never observes a complete allocation history. For this reason, the tracing framework cannot formally prove the existence or absence of a memory leak; it can only describe memory behaviour within the observed interval. This property is consistent with the dynamic observation control requirement defined in Chapter 3.2.

4.2 System Overview

This section presents a high-level overview of the tracing framework architecture. The system is decomposed into logical components that capture, process, store, and interpret memory events. Figure 4.2 provides a simplified schematic representation of these components. Subsequently, each component and its role within the overall architecture is explained.

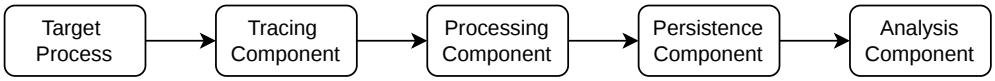

Figure 4.2: Architectural overview

Target Process
The application whose memory behaviour is to be analysed. It remains unmodified and is observed externally.

Tracing Component
Captures the memory related events from the target process. Transfers events to the processing component. The component operates in a transparent manner, without causing interference to the target process.

Processing Component
Receives the events from the tracing component and transforms them into structured representations suitable for storage and analysis.

Persistence Component
Stores both captured events and derived data representations for the purpose of long-term analysis and the investigation of long-term growth patterns. Ensures

that the data is available beyond the observation window. This facilitates repetition of the analysis, application of heuristics, and comparison of results, for example, after bugs in the target application have been fixed.

Analysis Component
Interprets the aggregated data and identifies growth patterns. Supports the developer in investigating memory behaviour. Operates independently of the runtime tracing.

The tracing and processing components constitute the monitoring infrastructure, while persistence and analysis components enable long-term interpretation of the collected data. To prevent system overload, runtime monitoring must be decoupled from computationally intensive tasks. This separation is achieved through the persistence component, which forms the boundary between data capture and interpretation.

4.3 Data Model

The data model describes the logical data entities used within the tracing framework. Figure 4.3 illustrates these entities and their relationships in a simplified conceptual representation. Subsequently, each data representation is briefly introduced, and its purpose within the overall model is explained. This section provides a comprehensive explanation of how the observed memory operations are represented and related to each other, independent of their technical implementation.

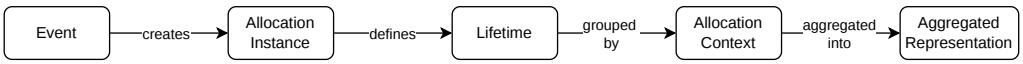

Figure 4.3: Conceptual data model

Event

An event represents a single observed memory operation. It corresponds either to an allocation event or a deallocation event. Each event is atomic and contains the following attributes: timestamp, size, memory address, process identifier, thread identifier, allocation context, and event type.

Allocation Instance

An allocation instance represents a concrete allocation during runtime. It is created by an allocation event and closed by the corresponding deallocation event, if such an event occurs within the observation window. An allocation instance does not exist independently but is derived from observed events. Each allocation

instance is uniquely identified by its allocation event and the associated memory address at that point in time. Multiple allocation instances may share the same allocation context, but they represent distinct runtime allocations.

Lifetime

Figure 4.4 illustrates the concept of a lifetime. The lifetime represents a time interval between an allocation and the corresponding deallocation event. It can only be assigned to an allocation instance. Open-ended lifetimes may occur due to incomplete observation or missing deallocation events. This is depicted in the right diagram.

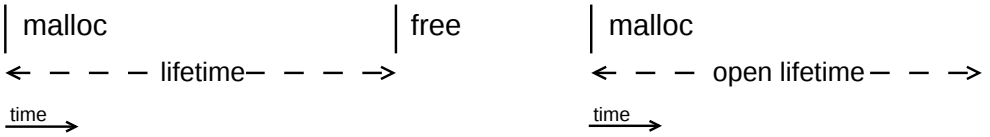

Figure 4.4: Allocation lifetime model

Allocation Context

The allocation context represents the program location responsible for the allocation event. Multiple allocation instances may share the same context. Deallocation events inherit the allocation context of their corresponding allocation event.

Aggregated Representation

The aggregated representation is derived from multiple allocation instances that belong to the same allocation context. It captures statistical properties over time, including the current active memory state, cumulative allocation metrics, lifetime characteristics, and historical extrema. The rationale for aggregating these metrics per allocation context is discussed in Section 4.4.3.

4.4 Processing Model

While the data model defines the structural representation of memory operations, the processing model describes how these data entities are transformed into an interpretable memory state. The following sections outline the logical processing steps required to reconstruct and interpret memory behaviour.

4.4.1 State Reconstruction

As established in Chapter 4.1, the reconstruction of the memory state from observed memory operations is essential for an accurate interpretation of memory behaviour. At this point, it must be emphasised that this state is not directly observed but derived exclusively from allocation and deallocation events, and therefore represents an event-sourced, inferred model rather than an intrinsic system property. The tracing framework maintains this reconstructed state as an internal representation of runtime memory behaviour.

More precisely, this internal state representation corresponds to a consistent snapshot of the process heap within the observation window. It is maintained by processing a continuous stream of allocation and deallocation events. Each allocation event creates a new allocation instance that is inserted into the active allocation set. This set contains all currently active allocation instances, including their allocation size, allocation timestamp, associated allocation context, and further attributes, as explained in Section 4.3. An allocation instance is considered active from the moment its allocation event is observed until the corresponding deallocation event is processed. At that point, the allocation instance is closed and removed from the active set. Considering this process, the memory state can be understood as an evolving function of the processed event stream.

The correctness of this reconstructed memory state depends entirely on the completeness and ordering of the observed event stream. Missing or incorrectly ordered events would directly corrupt the inferred memory model and may lead to an inaccurate representation of runtime memory behaviour.

4.4.2 Incomplete Information Handling

However, the assumption of a complete and perfectly ordered event stream represents an idealised model and does not fully reflect real-world conditions. In real observation environments, such conditions cannot be guaranteed.

As the tracing framework operates as an external observer, it only has access to the events that fall within the observation window. Every observation has a beginning and an end. When attaching the tracing framework to a continuously running application, allocations may already exist that were created before the tracing framework was attached. These allocations are unknown to the tracing system. Likewise, when detaching at an arbitrary point in time, some allocations may remain active because their corresponding deallocation events occur outside the observation window.

Consequently, from the perspective of the observer, a complete allocation history does not exist. The reconstructed memory state is therefore always relative to the observation window. The tracing framework cannot assume knowledge of events

that occurred before attachment or that will occur after detachment. Hence, the reconstructed state is inherently based on incomplete information. This incompleteness is a structural characteristic of runtime tracing and does not indicate a malfunction of the tracing mechanism. As a consequence, the system must tolerate partial information while still maintaining a meaningful representation of the memory behaviour.

Incomplete information can manifest in different ways. As already mentioned, deallocation events may be observed without a corresponding allocation event, or allocations may remain active without an observed deallocation event, either because no deallocation exists or because it occurs outside the observation window. In addition, temporary inconsistencies in event order may occur due to limitations that apply to real systems, such as finite processing speed, finite buffering capacity, and the absence of perfect synchronisation. Effects attributed to multithreading and multi-core execution further increase the likelihood of temporary inconsistencies in the event sequence. The tracing framework must tolerate both temporal inconsistencies and incomplete information.

To preserve consistency, the tracing framework follows a conservative interpretation strategy. This means that the reconstructed state only reflects what has been observed and does not speculate about unknown events. This conservative interpretation strategy results in concrete processing rules:

- An observed deallocation event without a matching allocation event is temporarily stored and only integrated into the reconstructed state if a corresponding allocation event is later observed.
- Allocation events that remain unmatched until the end of the observation window are retained as active instances without being automatically classified as leaks.

Where necessary, the system may apply only the minimal assumptions required to maintain internal consistency while clearly separating between observed and inferred state transitions. This applies when an allocation is observed for a memory address that is already considered active. The system then assumes that a deallocation event must have occurred, but the allocation event is processed first. The previous allocation instance is then closed with the inferred free, and a new one is opened. Such discrepancies can be attributed to temporal inconsistencies within the system.

These rules ensure the stability and internal consistency of the system, even in a scenario where the observed information may not be complete. The goal of the processing model is not to reconstruct a historically complete memory state but to maintain a consistent state representation within the boundaries of the observable information.

4.4.3 Online Processing

Section 4.4.1 outlined how the memory state is derived from allocation events. This section now explains how this reconstruction is performed and maintained over time.

The event-sourced reconstruction could in theory be performed entirely offline after the observation has finished. However, the reconstructed state is required to be available at any point during the observation window. For continuously running applications, the measurement process may execute for an extended period of time. Whenever a developer observes unexpected system behaviour, it should therefore be possible to retrieve the current reconstructed memory state without terminating the observation. This supports early detection of critical allocation contexts.

Consequently, instead of storing all events for post-mortem analysis, the tracing framework performs incremental reconstruction and aggregation while attached to the target process. Each observed allocation and deallocation event is directly integrated into the internal state representation, allowing the tracing framework to provide a consistent snapshot of the memory state at any time during the observation.

Context-Based Growth Analysis

During the online aggregation, the system also derives higher-level metrics from the evolving reconstructed memory state. This includes assigning each processed event to its allocation context so that statistics can be accumulated and critical code sections can be identified efficiently.

It is important to note that deallocation events are attributed to the allocation context of their corresponding allocation event. Memory growth does not originate from individual allocation or deallocation events in isolation, but from a persistent imbalance between allocations and corresponding deallocations within a specific allocation context. Each allocation context therefore represents a potential growth source.

By attributing both allocation and deallocation events to the originating allocation context, the system can assess whether allocations from a given context contribute to stable behaviour or long-term memory growth. This ensures that statistical evaluation reflects the behaviour of allocation sources rather than the code locations responsible for deallocation.

Figure 4.5 illustrates this aggregation process conceptually. This approach also reduces the need to repeatedly examine the entire event history in order to derive state information.

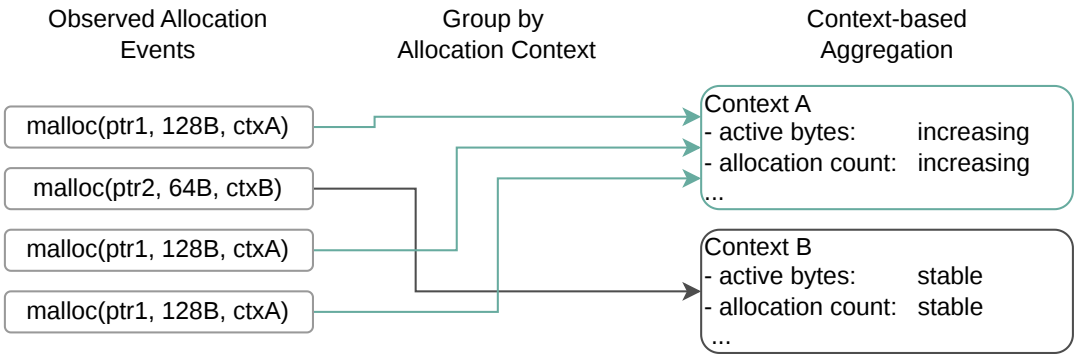

Figure 4.5: Conceptual illustration of context-based statistics.

Consistency under Incomplete Information

Beyond enabling context-based growth analysis, the online processing model contributes to the robustness of the system in the presence of incomplete or temporarily inconsistent event streams.

While Section 4.4.2 defined the rules for handling incomplete information, their practical applicability relies one the stateful nature of the online processing model. Because the reconstructed memory state is maintained incrementally during runtime, short-lived mismatches or deviations in the event order can be resolved as new events arrive. This would not be possible in a passive event recording model, where interpretation is deferred until after observation. In this way, overall consistency and a well-defined reconstructed memory state can be ensured at all times.

4.4.4 Persistence Strategy

In addition to the data retrieved during online aggregation, the tracing framework must ensure that relevant information remains available beyond the observation window. The data produced during the online aggregation is bound to the lifetime of the tracing session. For reproducibility, extended analysis, and comparison of different observation runs, persistent storage of selected data is required.

Therefore, the persistence strategy for this tracing framework consists of two complementary forms of stored information.

First, the complete raw event data is persisted. The raw event data consists of every allocation and deallocation event in the exact order in which they were processed. This includes all information stored with each event, as described in Section 4.3. This approach provides a permanent record of the basis for the reconstruction process. Consequently, storing the full raw event data ensures reproducibility of the observation and allows for later re-evaluation or extended

post-mortem analysis.

Second, the aggregated representations derived during the online processing are stored. These include structured summaries and statistics that evolve during the observation process, describing the memory behaviour of the target application. Storing the aggregated information enables rapid inspection of the memory behaviour, particularly growth characteristics, without having to reprocess the entire event history.

This separation of data serves two purposes. Runtime observation can be decoupled from subsequent, potentially computationally intensive analysis. In addition, the tracing framework preserves completeness of data beyond the observation window while enabling interpretability at the same time. The raw event data provides maximal detail, whereas the aggregated data delivers structured and condensed insight into the memory behaviour.

4.5 Technical Implications for the Tracing Mechanism

The architectural model described in this chapter imposes specific technical constraints on the underlying tracing mechanism. In order to realise the event-sourced reconstruction, online aggregation, and persistence strategy outlined above, the implementation must satisfy a concrete set of technical requirements.

Most importantly, the tracing mechanism must support dynamic attachment to and detachment from a running target and capture allocation-related events externally, without modifying the target application's source code, build pipeline, or binary. In production environments, the target software is often already deployed and cannot be modified, rebuilt, or restarted for diagnosis. Additionally, developers may wish to observe unexpected system behaviour at the moment it occurs. Restarting the application for diagnostic purposes would undermine the objective of the observation.

Furthermore, the mechanism must provide a complete stream of allocation and deallocation events. As established in Chapter 4.1, reconstruction of the memory state depends on a continuous and lossless event stream. Approaches that rely on statistical sampling are therefore insufficient.

The tracing mechanism must operate on target systems that are assumed to provide only extremely limited resources. This must be achieved without compromising accuracy or real-time requirements. The tracing mechanism must not alter the observed memory behaviour.

In addition, the tracing infrastructure must correctly capture and correlate events

originating from multi-threaded and multi-core execution environments. Events may occur simultaneously across different threads and processors. Despite parallel execution and potential reordering effects, consistent memory state reconstruction must remain possible.

Finally, the tracing approach must enable the retrieval of the allocation context, such as the program location responsible for the memory operation. Without this information, meaningful interpretation of memory growth patterns is not possible.

Among the mechanisms discussed in Chapter 2, only approaches that satisfy the above realisation requirements remain viable candidates.

5 Design and Implementation

A re-evaluation of the approaches outlined in Chapter 2 shows that most conventional techniques do not satisfy all derived realisation requirements simultaneously. While compile-time instrumentation tools require recompilation of the application, heavyweight dynamic binary instrumentation would require a restart of the program and apply excessive overhead to the target system. Library interposition, for example `LD_PRELOAD`-based, on the other hand, requires modification of the target application's execution environment, and statistical monitoring based tools cannot trace every single memory event as they rely on sampling.

Kernel-assisted tracing mechanisms provide the most suitable combination of completeness, the ability to attach during runtime, low overhead, and external observability. Among these, eBPF provides a suitable mechanism for meeting these requirements. Therefore, in this work, the tracing framework is implemented using eBPF technology.

This chapter presents the realisation of the previously defined architecture using eBPF as the underlying technology. It describes how the solution-independent architecture introduced in Chapter 4 is mapped onto the technical capabilities of eBPF and outlines the resulting design decisions.
After a brief introduction to eBPF integration, the chapter is structured according to functional building blocks of the framework. These include the tracing mechanism, state reconstruction and event correlation, context-based memory analysis, data transport and backpressure handling, persistence strategy and degradation handling.

5.1 Introduction to eBPF

The Extended Berkeley Packet Filter (eBPF) originated from the Berkeley Packet Filter (BPF), first introduced to Linux in 1997, based on the design by McCanne and Jacobson (1992). BPF was originally developed for packet filtering (Rice, 2023, p. 2; McCanne & Jacobson, 1992). eBPF evolved from this network-

restricted use case to a ”general-purpose virtual machine running inside the Linux kernel” (Vasquez & Simmonds, 2021, p. 671).

Today, eBPF provides a sandboxed execution environment, allowing programs to run inside the Linux kernel (Vasquez & Simmonds, 2021, p. 670). An eBPF program can be dynamically attached to various hook points in the kernel and the user space, allowing the tracing of specific events without restarting the target application(s) (Rice, 2023, p. 9). Furthermore, no modification or recompilation is required (Rice, 2023, p. 1). Due to these properties, eBPF has evolved into a versatile foundation for observability, tracing and general analysis in modern Linux systems.

This makes it particularly well suited for the observation model and processing strategy developed in Chapter 4.

This section introduce the relevant execution model, attachment mechanisms, and data exchange concepts that exist within eBPF.

5.1.1 Execution Model

An eBPF-based system always consists of two parts: the eBPF program executed in the kernel and a user space program responsible for loading the eBPF program into the kernel, configuring and retrieving data from it (Rice, 2023, p. 16; Xu & Chen, 2024).

The eBPF execution model is based on a lightweight virtual machine embedded in the Linux kernel. eBPF programs are usually written in C and compiled into eBPF bytecode using a compiler toolchain that supports the eBPF instruction set, most commonly the Clang/LLVM toolchain, although GCC has also provided an eBPF backend since version 10. Before being loaded into the kernel, the bytecode must pass a verification process to ensure the security and stability. The eBPF verifier statically analyses the program and ensures that all possible execution paths comply with strict safety constraints, such as valid memory access, bounded execution, and controlled kernel interaction. This process guarantees bounded execution and compliance with kernel constraints. Only verified programs are permitted to execute (Vasquez & Simmonds, 2021, p. 670; Rice, 2023, pp. 9, 38, 109–113; Linux Kernel Documentation Project, n.d.).

Following the successful verification process, the eBPF bytecode is compiled into native machine instructions by the Just-In-Time (JIT) compiler. This facilitates near-native execution speed, resulting in significant performance improvement (Rice, 2023, pp. 38, 48; Vasquez & Simmonds, 2021, p. 670).

Once loaded into the kernel, an eBPF program is dynamically attached to a specific hook point, such as a function entry or system call. The execution of the

program is then event-driven; whenever the hooked event occurs, the eBPF program is executed. Dynamic attachment means that neither the target application nor the system must be restarted and attachment can occur at runtime (Rice, 2023, p. 9).

Because execution takes place inside the kernel and is triggered directly by relevant events, eBPF enables complete tracing of relevant events without sampling, inducing only low overhead (Rice, 2023, p. 10).

This supports requirements defined in Chapter 3, such as Dynamic Observation Control, Completeness of Observation, Performance, Operability, and Portability.

5.1.2 Attachment Mechanisms

eBPF programs are not executed autonomously but must be attached to specific execution points in the kernel or user space. These execution points are referred to as hook points. Depending on the tracing objective, different attachment mechanisms can be used to observe kernel level or user space behaviour. This section briefly explains the most relevant hook types for *MemScope*.

Kprobe

Kprobes are hook points that can be attached almost anywhere in the kernel to collect debugging and performance information non-disruptively. While the probe type kprobe can be inserted on any instruction, the probe type kretprobe, also called return probe, executes the eBPF program when a specified function returns. Kprobes support dynamic attachment, so no recompilation of the kernel is required. When a kprobe is registered, the original instruction at the probe location is copied, and its first byte(s) are replaced with a breakpoint instruction. When the breakpoint is hit, a trap is triggered, the processor state is saved, and control is transferred to the kprobes framework. The associated pre-handler is executed, after which the copied instruction is single-stepped. If defined, a post-handler is then invoked, and execution resumes with the instruction following the probepoint. However, when attaching a kprobe to a symbol name or kernel function, it should be considered that lines of code may be modified from one kernel release to the next (Rice, 2023, pp. 128–130; 'Kernel Probes (Kprobes)', n.d.).

Tracepoint

A tracepoint is a statically defined instrumentation point in the Linux kernel code and can be used for tracing and performance accounting. Unlike kprobes, tracepoints cannot be attached to arbitrary locations but only to predefined locations. This improves robustness and compatibility across kernel versions. Compared to

kprobes, tracepoints provide more stable observation points. When using tracepoints, a user-specified code is executed each time the tracepoint is reached. The function is called in the execution context of the caller. After the function has reached its end, it returns to the caller. Tracepoints are not exclusive to eBPF ('Using the Linux Kernel Tracepoints', n.d.; Rice, 2023, pp. 131–133).

Uprobe

Uprobes are implemented using a breakpoint-based mechanism similar to kprobes, but operate on user space instruction addresses. They can dynamically attach to a user space application or library function. A uprobe is attached to the entry of a user space function and a uretprobe to the exit. In order to use uprobes, the target binary does not need to be modified or recompiled. But, the offset of the probepoint in the object needs to be calculated and provided to the uprobe event interface. Uprobes are attached to a running application externally without modifying its code or logic. The application behaves functionally equivalent to execution without uprobes (Rice, 2023, pp. 133–134; 'Uprobe-Tracer: Uprobe-based Event Tracing', n.d.).

USDT

User Statically Defined Tracing (USDT) are used to attach to predefined tracepoints within the application code or user space libraries. These predefined tracepoints need to be placed explicitly in the target application by the developer during development. Although USDTs offer stable observation points, prior instrumentation of the target binary is required. Equivalently as for uprobes, the offset of the probepoint needs to be provided to the event interface (Rice, 2023, pp. 133–134; 'USDT - eBPF Docs', 2026).

Perf Events

Perf is a performance analysis framework built around the Linux perf_event infrastructure. Perf events are generated by the kernel's performance monitoring subsystem and primarily represent hardware and software performance counter events, as well as selected kernel-defined tracepoints. eBPF programs can attach to perf events to enable event-driven execution. However, these events are typically generated based on counter overflows or sampling configurations. As a result, even though execution is triggerd by events, the underlying observation model is often sampling-based. In practice, perf events are therefore predominantly used for statistical profiling rather than for complete and deterministic event tracing (Vasquez & Simmonds, 2021, pp. 653–659; Kerrisk, 2024; 'Perf: Linux Profiling with Performance Counters', 2024).

eBPF provides numerous additional attachment mechanisms, including network-

ing, scheduling, and security hooks (Rice, 2023, pp. 125–141), that are not relevant for this tracing scenario.

The attachment mechanisms described in this section differ in terms of flexibility, stability guarantees, required prior instrumentation, and suitability for user space versus kernel space observation. The selection of the appropriate hook type suitable for this work, considering the architectural requirements defined in Chapter 4 is discussed in Section 5.3.

5.1.3 Data Exchange

Since eBPF programs run isolated within the kernel, mechanisms are required to exchange data with user space components. For this reason, eBPF provides interfaces to exchange data between the user space and the kernel program. Without those, no data could be exchanged, as direct memory access between kernel and user space is not permitted. The following section briefly explains the relevant data structures for this work.

Maps

A map is a kernel data structure that can be created by the eBPF framework and accessed and modified by the kernel and the user space in the same way. Maps are typically used to either

- store data or states within the kernel program for later access
- allow the user space to write configuration data to be retrieved by the kernel program
- provide metrics or results aggregated in the kernel to the user space for further processing or storing

There are various types of maps, all of which are based on key-value data structures. eBPF programs interact with maps through dedicated helper functions provided by the kernel. Maps are generally referred to as BPF_MAP_TYPE_* when writing code, reflecting that maps are kernel-level BPF objects. Therefore, in this work they will be referred to as BPF maps and not as eBPF maps ('BPF Maps', n.d.; Rice, 2023, pp. 20–24).

Streaming

For event-driven tracing scenarios, eBPF provides streaming mechanisms. In tracing scenarios, buffers are used to stream data from the kernel to the user space in a producer-consumer model. Any type of data can be inserted into the buffer. The data is then consumed by the user space program. Buffers enable

data transfer from the kernel to user space without the need for any synchronous context switching, making it very well-suited for continuous, low-overhead event delivery.

Buffers in eBPF generally have the following characteristics:

- variable-length records
- when full, reservation will fail; no blocking
- possibility to memory-map the data area for user space applications for efficient consumption and high performance
- user space applications can be notified about new incoming data via standard Linux event mechanisms such as epoll
- ability to do busy polling for new data to achieve lowest latency

When referring to a buffer in the context of eBPF, this typically means either the perf buffer or the BPF ring buffer, as these are the two primary mechanisms used for event streaming. The ring buffer was introduced specifically to improve event delivery for BPF programs, therefore its formal name is "BPF ring buffer", reflecting its integration into the core BPF infrastructure. The perf event buffer relies on the Linux perf subsystem and provides a per-CPU event buffer. The BPF ring buffer is a newer implementation and was introduced to overcome specific limitations of the perf buffer:

- more efficient memory utilisation by sharing the buffer across CPUs
- preserving the order of events that occur sequentially in time, even across multiple CPUs

Similarly to maps, eBPF programs interact with buffers through dedicated helper functions provided by the kernel (Rice, 2023, pp. 24–28; 'BPF Ring Buffer', n.d.).

Maps and streaming buffers combined form the foundation for incremental state reconstruction and online aggregation, as described in Chapter 4.

5.2 eBPF Integration and Component Split

As discussed in Section 5.1, an eBPF-based system consists of two components:

1. an eBPF program executed inside the kernel, and
2. a user space application responsible for loading, configuring, and managing the eBPF program.

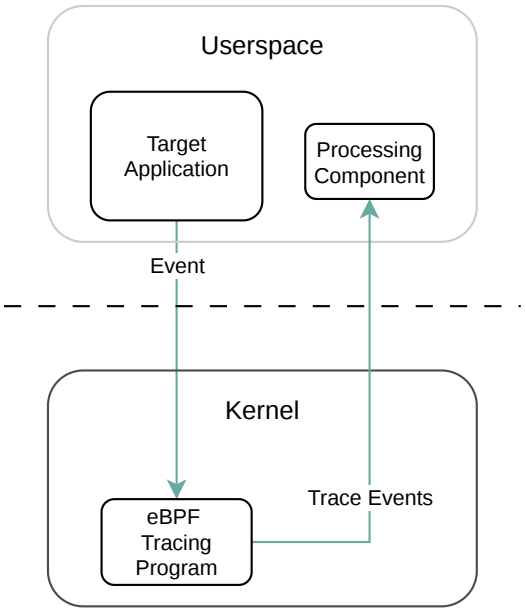

Figure 5.1: High-level execution model of MemScope.

Figure 5.1 illustrates the high-level integration of eBPF. Whenever the target application performs a memory operation, the associated hook triggers the execution of the eBPF program in the kernel. This program collects relevant metadata and exposes it to the user space processing component via BPF maps. Together, the eBPF tracing logic and the user space component constitute the tracing framework.

5.2.1 Choice of eBPF Tooling Ecosystem

When developing an eBPF-based system, one of the first decisions concerns the choice of the tooling ecosystem, which defines how eBPF programs are compiled, loaded, and integrated with user space components. eBPF offers numerous options. The most commonly used options include bpftrace, BPF Compiler Collection (BCC) and libbpf.

Given the constraints of the target environment, libbpf was selected as the implementation foundation. *MemScope* is intended to operate on a resource-restricted embedded Linux system without introducing significant runtime overhead or additional dependencies. High-level frameworks such as BCC rely on dynamic compilation and require the full LLVM toolchain to be present on the destination machine. This results in comparatively heavy user space runtimes, which are not suitable for minimal embedded deployments. Bpftrace, in contrast, is designed as an interactive high-level tracing tool based on a domain-specific language. It enables rapid development of eBPF-based tracing scripts but is intended as a stand-alone diagnostic tool rather than as a foundation for implementing a custom, stateful tracing framework (Rice, 2023, pp. 79–81, 185–188; Billimoria, 2024, p. 274).

Libbpf provides a lightweight C-based interface to the kernel's BPF subsystem, allowing direct control over program loading, map management, and buffer handling. Its minimal dependency footprint and native integration make it well suited for embedded environments. Among the available tooling ecosystems, libbpf was the only solution that satisfied the architectural, performance and deployment constraints simultaneously.

5.2.2 Separation of Kernel and User Space State

Although eBPF inherently implies a separation between kernel and user space, it remains a design decision how processing responsibilities are distributed between the two domains.

Although more complex processing could in principle be implemented inside the kernel, eBPF programs are subject to strict verifier constraints, bounded stack size, limited instruction complexity, and restricted memory resources. Exceeding these limits would either prevent the program from being loaded or increase the risk of resource exhaustion in kernel space. For this reason, *MemScope* deliberately restricts kernel-side processing to minimal preprocessing tasks.

Therefore, the kernel is only assigned to the following tasks:

- capturing relevant allocation and deallocation events,
- retrieving the allocation context by capturing the stack trace,
- maintaining small peripheral data structures required between entry and return hooks,
- aggregating simple global counters and statistics.

A stack trace represents the sequence of active function calls at a specific point in time. It is obtained by capturing the saved return addresses stored in the call stack, which allows reconstruction of the calling context.

All computationally intensive processing, including state reconstruction, allocation context statistics, and persistence, is performed and kept in the user space. In addition to kernel limitations, this is motivated by another consideration.

As described in Section 4.4.2, the reconstructed memory state may require corrections due to incomplete or temporarily inconsistent event streams. Performing full state management in the kernel would complicate such corrections and increase the risk of inconsistency.

Therefore, the kernel is limited to minimal data extraction and preprocessing, while the user space processing component maintains the authority over the reconstructed memory state and executes the higher-level analysis.

5.3 Tracing Strategy

This section explains in detail the tracing strategy adopted in this work and the eBPF mechanisms that are instrumented to realise it. As outlined in Chapter 4.1, the tracing framework must observe individual memory allocation and deallocation operations in order to reconstruct the memory state.

Event Retrieval

The target application is written in C and C++. In these programming languages, dynamic memory management is performed through a standardised Application Programming Interface (API). Within the scope of this work, allocations events triggered by `malloc` and deallocations events triggered by `free` are instrumented. These symbols are defined in the standard C library (libc).

Hooks are therefore placed at the symbol locations within the dynamically linked libc. Whenever the target application performs `malloc` or `free`, control is transferred to the corresponding implementation within the shared library. Attaching uprobes at these symbol entry and return points, guarantees that every instance of these functions can be observed. This ensures that all memory operations performed through the standard C heap interface are captured.

To associate the allocation size with the allocated memory address, it is necessary to capture both the function entry and return of `malloc`. The size is available at function entry, while the allocated address is only known after the function returns. In contrast, for `free`, the function return does not provide any additional information relevant to the tracing. Therefore, only the function entry is instrumented.

An additional advantage of hooking at the libc level is robustness against wrapper implementations. Even if a developer uses custom wrapper functions around `malloc` or `free`, the underlying allocation is still performed through the original

libc symbols. As a result, such instrumentations do not prevent the *MemScope* from observing the memory event.

However, this design has explicit limitations. Allocations cannot be observed if the target application is statically linked against libc, or if it employs a custom allocator that bypasses the standard heap interface. While tracing such cases would technically possible through additional instrumentation strategies, the present framework intentionally targets dynamically linked applications using the standard C/C++ heap allocation via libc. This design choice supports the portability and operability requirements defined in Chapter 3.

Hook Type

Section 5.1.2 introduced various attachment mechanisms available within eBPF and their characteristics. Before evaluating aspects such as stability or flexibility, it must first be determined, which mechanisms are applicable in this scenario.

Since the allocation and deallocation functions of interest are implemented in a user space shared library (libc), kprobes, kernel tracepoints, and perf events are not suitable, since they operate exclusively within the kernel. This leaves USDT and uprobes as viable candidates.

USDT requires prior code instrumentation of the target application. This contradicts the operability requirement defined in Chapter 3, which explicitly excludes source code modification or recompilation.

Uprobes, on the other hand, do not require prior modification of the target binary and allow dynamic attachment to user space functions. The can be placed at precise symbol offsets within shared libraries and therefore provide sufficient stability. Consequently, uprobes are used to attach to the standard C library symbols.

The following listing illustrates the basic structure of a uprobe hook. The `SEC` macro specifies the Executable and Linkable Format (ELF) section in which the hook is placed and determines the attachment type. In this case, the uprobe is associated with the entry point of the `malloc` function.

```
1 SEC("uprobe/malloc_entry")
2 int uprobe_malloc_entry(struct pt_regs *ctx) {
3 /* user-defined code */
4 }
```

Listing 5.1: Syntax of a uprobe hook.

The function follows the standard eBPF program signature and receives a pointer to the processor register state (`struct pt_regs`), which provides the arguments

of the probed function. The complete implementation of the `malloc` return hook is provided in Appendix A.

Hook Attach in User Space

Once the uprobe has been defined in the eBPF kernel program, the hook must be attached to the corresponding function. This attachment is performed in the user space using libbpf's `attach_uprobe_sym` function. It attaches the probe based on the symbol name (e.g. `malloc`) without requiring to manually calculate the address offset.

The function requires the eBPF program reference (`*prog`), the Process Identifier (PID) of the observed process (`pid`), a reference to the path to the shared object in which the symbol is defined (`*bin_path`), a reference to hooked symbol (`*sym`), and a flag (`retprobe`) indicating whether the probe should trigger on function entry (`false`) or return (`true`).

Libbpf then takes over the symbol offset resolution within the ELF object and configures the uprobe accordingly. The following listing shows all relevant code to perform the uprobe attach ('Libbpf API Error Handling', n.d.):

```
static struct bpf_link *attach_uprobe_sym(
                            struct bpf_program *prog,
                            pid_t pid,
                            const char *bin_path,
                            const char *sym, bool retprobe) {
    struct bpf_uprobe_opts opts = {
                        .sz = sizeof(opts),
                        .func_name = sym,
                        .retprobe = retprobe,
    };
    return bpf_program__attach_uprobe_opts(prog,
                                           pid,
                                           bin_path,
                                           0 /* func offset */,
                                           &opts);
}
struct bpf_link *l1 = attach_uprobe_sym(p_malloc_entry,
                                        target,
                                        libcpath,
                                        "malloc",
                                        false);
```

Listing 5.2: Uprobe attach in the user space.

The returned `bpf_link` object represents the attachment and must be explicitly detached during shutdown.

Program Attach in User Space

MemScope operates in a PID-targeted mode. Providing the PID of the target application when attaching the uprobes, ensures that only events originating from the selected process are captured.

eBPF also permits system-wide attachment when no PID is provided. However, limiting the scope of the observation reduces the overhead and therefore supports the performance requirement defined in Chapter 3.

5.4 State Reconstruction and Correlation Strategy

While the previous section defined how the events are captured, this section explains how the memory state is reconstructed from these events.

The reconstructed memory state is represented by all active allocation instances at a given point in time. These active allocation events are stored in a hash map maintained by the user space program. Each element of the map is keyed by its memory pointer, i.e., the address of the allocated memory, and contains all relevant information associated with the allocation event. Specifically, this includes the size (`size`) of the allocated memory, a unique identifier for the stack trace (`stackid`), the kernel time stamp at which the event occurred (`ts_ns`), the age of the allocation (`age`), and an internal unique identifier (`id`) .

The stack identifier, hereafter also referred to as stackid, is used to determine the exact allocation context of an event. As elaborated in Chapter 4.4.3, growth analysis must be performed by grouping allocation instances according to their allocation context. This allocation context is retrieved in the kernel program using a built-in BPF helper function. The function captures the complete stack trace of an event up to a user-defined depth and returns a unique identifier for that specific sequence of call sites. The depth parameter defines how many nested function calls are included in the stack trace. For *MemScope*, the depth is set to 20 frames to ensure that complete allocation call chains are captured.

The unique stack identifier is a hash computed over the captured stack trace and automatically generated by eBPF. The corresponding stack trace is stored in a dedicated BPF stack map keyed by this identifier. As with all BPF maps, the stack map can be accessed both by the kernel program, to store new stack traces, and by the user space program to resolve a stack identifier to its associated call sequence.

This code snippet illustrates how a stack trace map is defined and what the corresponding syntax looks like. The map is shown as a representative example

for the maps used within *MemScope*, as the overall structure and syntax are consistent; in most cases, only the map type and the key/value definitions differ.

```
struct {
    __uint(type, BPF_MAP_TYPE_STACK_TRACE);
    __uint(max_entries, MAX_STACKS);
    __uint(key_size, sizeof(s32));
    __uint(value_size, sizeof(u64) * MAX_STACK_DEPTH);
} stack_traces SEC(".maps");
```

Listing 5.3: Implementation of a stack trace map.

The constants `MAX_STACKS` and `MAX_STACK_DEPTH` are macro-defined constants that can be customised. The identifier `stack_traces` declares the map instance, while `SEC(".maps")` places the map definition into the dedicated ELF section used by libbpf to discover and initialise BPF maps during loading.

The second value stored with each allocation instance that requires further explanation is the age parameter. To understand the purpose of this parameter, it is first necessary to describe how the state of the memory is reconstructed in practice.

Whenever an event has been processed in the kernel, it is streamed to the user space. The user space program maintains a hash map of active allocations. Each allocation event that arrives is inserted into this map and keyed by its memory pointer, that is, the address of the allocated memory.

The memory pointer is used as the key for two reasons. First, when a deallocation event is observed, it provides the pointer to the memory address being released. This allows the user space program to efficiently locate the corresponding allocation instance in the map and close it. The second reason becomes relevant in the context of incomplete information handling and is therefore explained later in this section.

Each allocation instance remains in the active allocations map until a corresponding deallocation event is processed. Upon processing a deallocation event, the matching allocation instance is removed from the map. This effectively closes the allocation instance and removes the corresponding memory address from the set of currently active allocations of the observed process.

When analysing a program's memory behaviour, it is often informative to determine the lifetime of an allocation. This refers to how long a specific memory region has been held by the process at the moment the tracing session terminates. For this reason, each allocation event in the active allocations map contains an age parameter. This parameter stores the timestamp recorded when the allocation event was processed in user space.

At the termination of *MemScope*, there will typically still be entries in the active allocations map. For continuously running programs this does not necessarily indicate memory leaks, since the termination of the framework does not imply termination of the observed process. The process may legitimately still hold active allocations.

Nevertheless, the remaining active allocations provide valuable insight into memory behaviour. Upon termination, *MemScope* calculates the age of each remaining allocation by subtracting the stored allocation time stamp from the current timestamp. This allows the developer to reconstruct a temporal view of allocations that were still active at the end of observation and to asses whether any of them contribute to long-term memory growth.

Now that the incremental reconstruction of the memory state has been described conceptually, it is important to emphasise once more that, as discussed in Chapter 4.4.2, ideal conditions cannot be assumed in real-world tracing scenarios. In addition to concurrency effects, the event stream may exhibit short-term reordering due to buffering and batched delivery from kernel to user space. The concrete transport mechanism is discussed in Section 5.6.

Kernel-Side Event Processing

The following section describes how events propagate through the actual system implementation, where corner cases and temporary inconsistencies must be handled while maintaining a meaningful and consistent representation of the memory state at all times.

The propagation of events within the kernel is illustrated in the upper section of Figure 5.2.

Whenever the target process allocates memory, this triggers the `malloc_entry` hook in the kernel. This hook is executed when the `malloc` function is entered. At this point, the allocation size is available as a function argument. This information is no longer directly accessible in the `malloc_ret` hook. Therefore, the allocation size is temporarily stored in a BPF hash map keyed by `PID_TGID`.

This temporary map to is accessed via a BPF helper function. These functions are implemented in the Linux kernel and provided by the BPF subsystem to enable controlled interaction between BPF programs and kernel-managed objects. They are the only mechanism by which BPF programs can interact with kernel-resident map objects.

```
1 bpf_map_update_elem(&tmp_size_map, &pidtgid, &size, BPF_ANY);
2 u64 *sizep = bpf_map_lookup_elem(&tmp_size_map, &pidtgid);
```

Listing 5.4: Using BPF helper functions to access BPF maps.

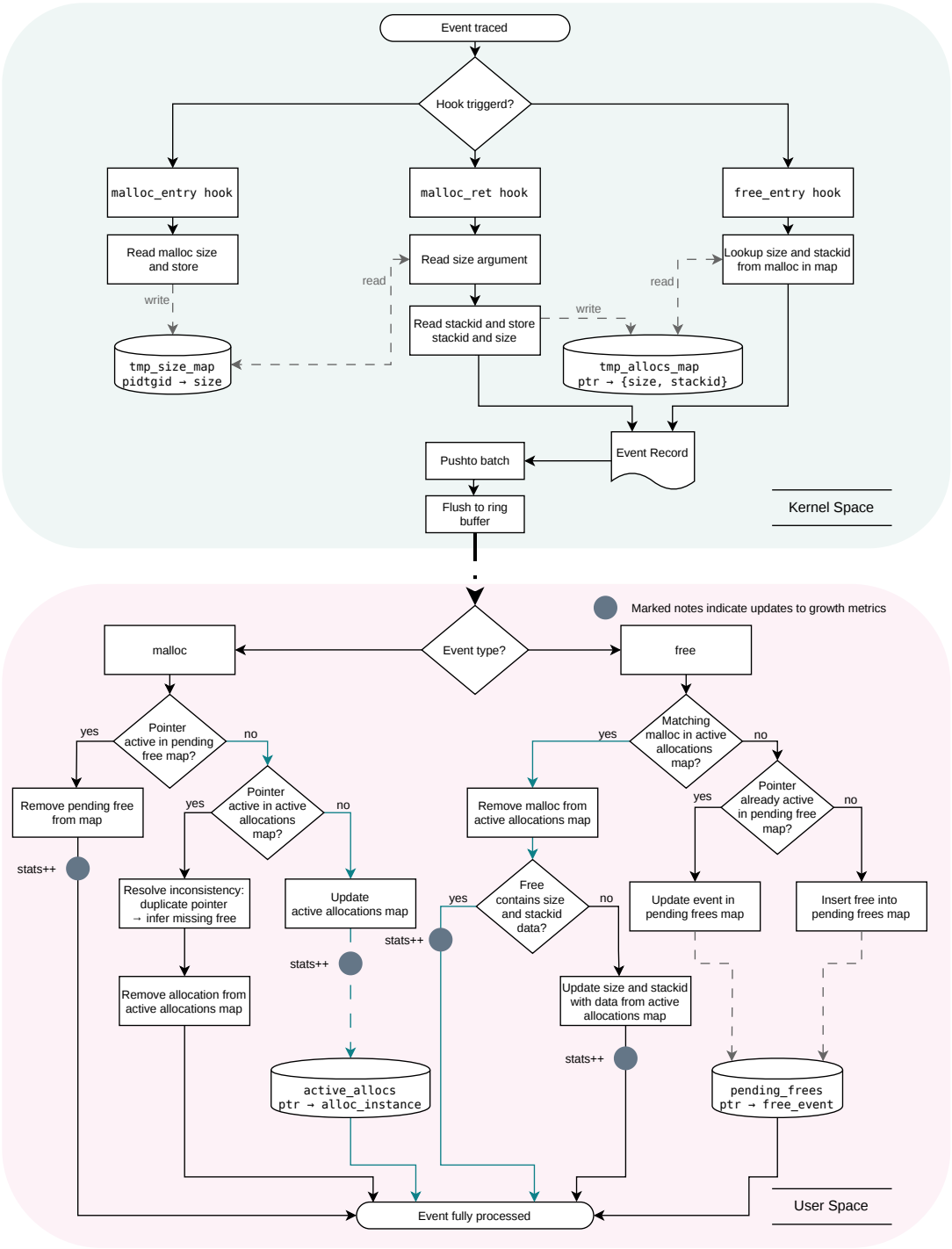

Figure 5.2: Kernel to user space event flow and correlation logic in *MemScope* for the processing of one single event. The blue arrows in the user space section indicate the optimal processing path, i.e., the scenario in which no temporal inconsistencies or incomplete information occur.

The function `bpf_map_update_elem()` inserts or updates a map entry for the given key, while `bpf_map_lookup_elem()` returns a pointer to the stored value associated with that key. These helper calls are representative of the general mechanism used to interact with BPF maps. In addition to update and lookup operations, helper functions such as `bpf_map_delte_elem()` are available to remove entries. Further map operations exist within the BPF API; however, as they are not required for the framework presented here, they are not discussed in detail.

The `PID_TGID` is a 64-bit kernel-provided identifier that encodes both the process ID (PID) and the Thread Group Identifier (TGID) into a single value. This enables distinction between threads within the same process. Since a single thread cannot execute two instances of the same function concurrently, the entry and return hooks can be safely correlated without additional synchronisation.
After storing the allocation size in the temporary map, the `malloc_entry` hook completes.

The `malloc_return` hook is triggered when the `malloc` function in libc returns. At this point, the return value of `malloc`, which corresponds to the allocated memory address, is available as a function argument. Using `PID_TGID`, the allocation size is retrieved from the temporary size map.
Subsequently, additional event data is collected, including the stack trace, the stack identifier, kernel timestamp, and `PID_TGID`. The event type is set to `malloc`. The stack identifier and allocation size are then stored in a second temporary hash map keyed by the memory pointer. This ensures that the information required to correlate a future `free` event remains available.
Once all event data has been collected, it is written to an event record (a struct data type). The record is then added to a per-CPU batch buffer and submitted to the ring buffer when the batch is full.

The use of separate entry and return hooks is required because the allocation size is only available at function entry, whereas the allocated memory pointer is only available at function return.

The `free_entry` hook is triggered whenever the observed process releases previously allocated memory. When the `free` function is called, the memory pointer to be released is passed as a function argument and is therefore available within the hook.
Similarly to the `malloc_ret` hook, all relevant event data is retrieved to construct the event record. This includes the timestamp, memory pointer, `PID_TGID`, stack identifier, and allocation size. The stack identifier and the allocation size are obtained from the temporary hash map created in the `malloc_ret` hook. It is accessed using a BPF helper function using the memory pointer as the key.
The event type is set to `free`, and the completed event record is submitted to the buffer.

User Space Event Processing

In the user space, event handling is strictly separated by event type, as can be observed in Figure 5.2.

When a `free` event is processed, *MemScope* first checks whether an active allocation with the same memory address exists in the `active_allocations` map. If such an entry is found, the corresponding allocation instance is removed from the map, thereby closing the allocation.

If no corresponding allocation entry is found in the `active_allocations` map, the system assumes, that either

1. the allocation occurred outside the observation window, or
2. due to temporary inconsistencies in event ordering, the deallocation event is processed before its corresponding allocation event.

In the latter case, discarding the `free` event would corrupt the reconstructed state. Therefore, the event must be preserved temporarily. This is realised through a hash map keyed by the memory pointer, hereafter referred to as `pending_frees` map.

If an entry for the same pointer already exists in the `pending_free` map, the older deallocation event is updated with the data of the newer event. If no such entry exists, the deallocation event is inserted into the `pending_frees` map.

When a `malloc` event arrives in user space, *MemScope* first checks whether a corresponding deallocation event exists in the `pending_frees` map. If such an entry is found, the allocation instance is immediately resolved: the deallocation event is removed from the `pending_frees` map and the allocation event is not inserted into the `active_allocations` map.

If no matching deallocation is found, the system then searches the `active_allocations` map for an entry with the same memory address. If such an entry exists, the previous allocation instance is removed and replaced by the new allocation event, thereby creating a new allocation instance.

This situation occurs because a memory address within a process can only be actively allocated once at a given time. If a new event references a memory address that is still considered active, the system assumes that the deallocation event for this previous allocation has not yet been processed. This condition is interpreted as an inferred `free`. Retrieving this information is the second reason, why active allocations are stored in a map keyed by their pointer, as described in the beginning of this chapter.

If neither an active allocation nor a pending deallocation entry exists for the memory pointer, a new allocation instance is created and inserted into the

`active_allocations` map.

This two-staged lookup strategy ensures that allocation and deallocation events are correlated correctly, even in the presence of temporary reordering or incomplete information. It represents the implementation of the allocation event acquisition requirement defined in Chapter 3. The corresponding implementation responsible for correlation `malloc` and `free` events is provided in Appendix C.

As described in this section, *MemScope* uses a set of BPF maps in the kernel and user space hash maps to maintain state, coordinate event propagation, and accumulate statistical metrics. Table 5.1 and 5.2 provide a structured overview of these data structures and their respective roles within the system architecture.

Name	**Type**	**Key**	**Value**	**Purpose**
`active_allocs`	uthash	`ptr`	`allocs_map`	Maintains currently active allocation instances for state reconstruction.
`pending_frees`	uthash	`ptr`	`allocs_map`	Temporarily stores unmatched free events.
`stack_stats`	uthash	`stackid`	`stackid_stats`	Aggregates growth metrics per allocation context.

Table 5.1: User space maps used in *MemScope*.

Name	Type	Key	Value	Purpose
`tmp_size_map`	HASH	`pid_tgid`	`size`	Stores allocation size between `malloc_entry` and `malloc_ret`.
`tmp_allocs_map`	HASH	`ptr`	`alloc_tmp`	Stores size and stackid between allocation and corresponding deallocation.
`global_stats`	ARRAY	0	`stats_global`	Stores global monitoring counters.
`cntrl`	ARRAY	0	`state_flag`	Controls tracing state (running, shutdown, stop).
`batch_map`	PERCPU_-ARRAY	0	`event_batch`	Per-CPU event batching.
`events_ringbuf`	RINGBUF	–	`event_batches`	Transfers batched allocation and deallocation events to user space.
`stack_traces`	STACK_-TRACE	`stackid`	`stack_traces`	Stores captured user space stack traces in kernel memory.
`known_stackids`	HASH	`stackid`	`known_flag`	Tracks already observed stack identifiers.
`stackids_-ringbuf`	RINGBUF	–	`stackid`	Streams newly detected stack identifiers to user space.

Table 5.2: BPF kernel maps used in *MemScope*.

5.5 Incremental Context-Based Memory Analysis

The reconstructed memory state forms the foundation for further analysis and already offers insight into overall system behaviour. However, the primary strength of *MemScope* is its detailed statistical evaluation of memory behaviour. These statistics enable developers to analyse dynamics in depth and to distinguish intended memory growth from unintended and potentially harmful behaviour.

The statistical model operates in parallel to state reconstruction. It does not replace it; rather, it is derived from the observed state transitions.

This section describes how these statistics are accumulated and what they represent.

Aggregation by Allocation Context

As elaborated in Chapter 4.4.3, identifying memory growth requires aggregation per allocation context. The framework therefore derives growth-related metrics incrementally during runtime. This enables visibility of allocation patterns and potential memory growth without requiring post-mortem analysis.

The allocation context is captured in the kernel by retrieving the current user space stack trace. This is realised using the BPF helper function `bpf_get_stackid`, which stores the stack trace in a dedicated BPF stack map and returns a unique identifier:

```
int stackid = bpf_get_stackid(ctx,
                              &stack_traces,
                              BPF_F_USER_STACK);
```

Listing 5.5: BPF helper function to retrieve stack trace and corresponding stack identifier.

The `ctx` parameter provides access to the hook-specific execution context supplied by the kernel, while `&stack_traces` is the name of the stack trace map used to store the captured user space stack frames.

Metrics Maintained per Allocation Context

For each stack identifier, the following statistics are maintained:

- **Initial characteristics:** initial allocation size
- **Active state:** current active bytes

- **Cumulative metrics:** total allocated bytes, total allocation count, total deallocation count
- **Extrema:** peak active bytes, minimum/maximum/average allocation size
- **Lifetime signals:** minimum/maximum/average allocation lifetime, peak age of the longest-living allocation
- **Historical data:** bounded history (up to N samples) of peak size and peak active bytes with corresponding timestamps
- **Robustness indicators:**
  - inferred `frees` counter (see Chapter 5.8),
  - never-freed-counter (allocations still active at termination),
  - allocations below max size/byte counter,
  - identifier of the oldest still active allocation

Update Semantics

Statistics are updated at defined points in the event workflow.

On each invocation of `malloc`, statistics are updated when:

- an allocation was directly resolved via a corresponding `free` from the `pending_frees` map (both events are accounted for), or
- a new allocation instance is inserted into `active_allocations` map.

These statistical updates triggered by `malloc` include:

- incrementing cumulative counters
- increasing active bytes
- updating peak values
- checking and updating minima
- creating historical entries when new peaks occur
- updating robustness counters as required

On each invocation of `free`, statistics are updated when:

- a corresponding allocation instance is resolved.

The statistical updates triggered by `free` include:

- reducing active bytes

- calculating allocation lifetime and incorporating into lifetime statistics
- updating lifetime peaks
- incrementing deallocation counter
- updating robustness counters (e.g., inferred `free`, below-peak classification)

The update logic is also reflected in Figure 5.2, where the points of statistical modifications are indicated.

Finalisation Phase

When tracing terminates, statistics are updated one final time. For each allocation still present in the `active_allocations` map:

- the current age is calculated and
- the `never_freed_counter` is incremented.

This ensures that long-living allocations are properly reflected in the aggregated metrics.

Stack Trace Resolution and Symbolisation

Stack trace resolution refers to the process of translating the raw instruction pointers collected during the tracing into corresponding source code locations. Symbolisation transforms the resolved information into a human-readable representation.
This section describes what information is needed and how this is done.

Various tools exist that can perform stack trace resolution and symbolisation. In this work, the GNU `addr2line` utility, specifically the `arrch64-linux-gnu-addr2line` variant, as the target application runs on a 64-bit Arm (AArch64) architecture. Using the architecture specific toolchain ensures that the correct binary format and instruction set are interpreted during the symbol resolution. The resolution does not need to be executed on the target device but can instead be performed on a host system, provided that the appropriate cross-toolchain utilities are available.

To resolve an address, `addr2line` requires two pieces of information: the executable or shared object in which the address resides and the address to be resolved. The binary must contain debugging information in order to resolve source file names and code lines numbers. This debugging information is typically stored in the Debugging With Attributed Record Formats (DWARF) debugging format within dedicated ELF sections such as `.debug_info` and `.debug_line`. When compiling a binary with debug symbols (e.g., using the `-g` compiler flag), the debug information is embedded in the ELF file.

Binaries that run on embedded targets often are compiled without debugging symbols; this is referred to as a stripped binary. This reduces storage requirements and enhances the security of an executable. As a result of stripped binaries on embedded devices, the runtime environment often does not contain the necessary debug information to resolve the stack trace directly on the device. For this reason, stack trace resolution and symbolisation are performed offline. The unstripped binary, or corresponding debug symbol files, can be used on a host system for post-processing of the recorded stack traces.

However, the instruction pointers captured cannot directly be passed to `addr2line`, for further processing.

Modern operating systems use Address Space Layout Randomization (ASLR), which is a security feature that causes memory regions such as the stack, heap, and dynamically loaded libraries to be mapped to different virtual addresses each time a program is started. This makes it more difficult for attackers to predict memory layout and exploit memory corruption vulnerabilities. Whether an executable supports relocation at load time depends on how it was compiled. Similarly to debug information, this behaviour is determined at compile time using flags such as `-fPIE` (for position-independent executables) or `-fno-pie` (position-dependent). Shared libraries, such as libc, are position-independent by design and are therefore always subject to relocation.

Resolving instruction pointers requires determining the runtime load address, defined as the base address at which an executable or shared object is mapped in the process's virtual address space. *MemScope* retrieves this information via the `proc/<pid>/maps` interface. This virtual file, provided by the Linux kernel, contains all memory mappings of a running process, including their start and end addresses, access permissions, and the path of the executable this memory range is mapped to. In *MemScope* this file is read and persisted once during the measurement to ensure offline stack trace resolution and symbolisation is possible.

However, simply subtracting the runtime mapping start from the captured instruction pointer does not yet provide the address in the format expected by `addr2line`. `addr2line` expects an address relative to the ELF object.

To compute the required ELF-relative address, the load bias (load_bias) must be determined. It represents the difference between the runtime mapping start address (runtime_mapping_start), retrieved from `proc/<pid>/maps`, and the virtual start address (elf_txt_vaddr) of the executable code segment defined in the ELF file (see 5.1).

$$\text{load\_bias} = \text{runtime\_mapping\_start} - \text{elf\_txt\_vaddr} \tag{5.1}$$

The virtual start address of the executable segment (i.e., `PT_LOAD` segment with execute permissions) is obtained directly from the ELF program headers. The

following listing is part of the offline Python script used to resolve the stack trace and shows the implementation of how the virtual start address (elf_txt_vaddr) is retrieved from the ELF file:

```
def get_elf_text_vaddr(path):
    with open(path, "rb") as f:
        elf = ELFFile(f)
        for seg in elf.iter_segments():
            if seg['p_type'] == 'PT_LOAD' and seg['p_flags'] & 1:
                return seg['p_vaddr']
```

Listing 5.6: Retrieving the virtual start address from the ELF file.

The complete Python script for the offline stack trace resolution is provided in Appendix D.

After performing these steps, the ELF-relative address for `addr2line` can be computed as:

$$\text{elf\_relative\_address} = \text{runtime\_ip} - \text{load\_bias} \tag{5.2}$$

This transformation removes the process-specific relocation offset introduced by ASLR and yields an address relative to the original ELF object. This address can now be passed to `addr2line`.

```
--- stackid 1994 ---
malloc_generator
    helper_b
    src/malloc_generator.c:57
malloc_generator
    callsite_b
    src/malloc_generator.c:69
malloc_generator
    main
    src/malloc_generator.c:120
libc.so.6
    __libc_init_first
    ??:?
libc.so.6
    __libc_start_main
    ??:?
malloc_generator
    _start
    ??:?
```

Listing 5.7: Example of an offline-resolved stack trace for one stack identifier. Frames from the target application resolve to file and line numbers, whereas vendor `libc` frames remain unresolved due to missing debug symbols.

Listing 5.7 shows an example of an offline-resolved user space stack trace derived from a captured stack identifier. Frames belonging to the target application can be resolved to source file and code line numbers. But frames originating from the vendor-provided `libc.so.6` remain unresolved (??:?) because the shared library was delivered as a stripped file.

The accuracy of the stack trace resolution depends on the availability of unstripped binaries and may be reduced for heavily optimised builds that use aggressive inlining or frame-pointer omission. The presented approach assumes that compatible binaries and symbol information are available for offline analysis (Billimoria, 2024, pp. 302–318, 319–329, 350–351; Kerrisk, 2025d; Kerrisk, 2026).

Memory-Bounded Online Aggregation

As the tracing system is designed to monitor continuously running processes, statistical accumulation must remain memory-safe. Therefore, the maintained metrics consist primarily of incrementally updated scalar values.

For each allocation context, *MemScope* maintains a bounded history of peak size and peak active byte time series (N samples). Once the buffer is full, additional upward transitions are not stored unboundedly. Instead, only the number of further upward steps is counted. This ensures that aggregation remains memory-bounded.

This design directly supports the performance requirement from Chapter 3.

Interpretation of the Metrics

The metrics can be analysed after tracing termination (see Section 5.7 for output format). Memory growth may be identified through manual inspection or via heuristics tailored to the observed program.

Potentially suspicious patterns include:

- average allocation size significantly exceeding the initial size (possible size drift),
- maximum lifetime approaching measurement duration,
- substantially more allocations than deallocations and
- repeated increase in peak size or peak active bytes.

These indicators may suggest memory growth or leaks. However, they do not automatically imply faulty behaviour. Long-lived allocations or sustained growth may be intentional. Interpretation therefore requires domain knowledge of the observed application. Providing generic, program-independent heuristics was not within the scope of this work. However, an offline analysis tool accompanies *MemScope*. It processes the generated output files and applies user-defined heuristics to support efficient exploration of large datasets.

Snapshot of the `active_allocations` Map

In addition to the aggregated metrics, a consistent snapshot of the `active_allocations` map can be derived at any time during observation. A description of the map can be found in Section 5.4. Providing access to the current active allocation state enables detailed runtime inspection of system behaviour. Details regarding consistency guarantees and persistence strategy are discussed in Section 5.7.

The contents of this section provide the implementation of the Allocation Origin Identification, Runtime Memory State Inspection and Memory Growth Observability requirements from Chapter 3.

5.6 Data Transport and Backpressure Handling

This section describes how events are transported from kernel to user space and how the system is designed to tolerate high event rates without violating the requirements of completeness, robustness, and performance defined in Chapter 3. The transport design directly influences both system robustness and metric consistency.

Section 5.1.3 introduces the two mechanisms provided by the BPF subsystem for transporting data from kernel to user space. In *MemScope*, event transport is realised using the BPF ring buffer. As discussed previously, the ring buffer provides a shared-memory design across CPUs with improved memory utilisation and simplified event semantics compared to the perf buffer. It introduces lower overhead and a programming model tailored specifically for eBPF workloads.

Ring Buffer Polling Behaviour

From user space, the ring buffer is accessed via a polling interface, specifically the `epoll_wait(2)` system call, with a configurable timeout (Kerrisk, 2025a):

- A timeout of 0 results in non-blocking polling. The call returns immediately if no data is available. Repeated invocation effectively leads to active polling

and increased CPU utilisation.

- A timeout of -1 blocks until data becomes available, minimising CPU consumption.
- A positive timeout blocks for the specified duration in milliseconds.

The polling interval is dimensioned with respect to the worst-case event rate, the per-batch aggregation size, and the overall ring buffer capacity. This provides bounded latency while maintaining sufficient buffering margins to tolerate short-term bursts without overflow.

For this framework, non-blocking polling mode (timeout = 0) was empirically evaluated and showed improved responsiveness under high event rates. However, this results in active polling behaviour and therefore increases CPU utilisation. Polling with a bounded timeout represents a trade-off between processing latency and resource efficiency.

Dual Ring Buffer Design

The tracing system employs two ring buffers:

- A primary ring buffer transporting batches of events.
- A secondary ring buffer streaming newly observed stack identifiers.

To avoid repeatedly emitting identical stack traces, a stack trace is transmitted only upon first observation of a new stack identifier in the kernel.

Stack traces are streamed continuously throughout the tracing process. This ensures that, if the developer requests a snapshot of active allocations during runtime, all stack identifiers observed up to that point are already available in user space. This allows immediate inspection of suspicious allocation behaviour, even during long-running observation sessions.

Whenever a new stack identifier is detected in the kernel, it is emitted to the user space via the secondary ring buffer. The user space callback then retrieves the corresponding stack trace from the kernel BPF stack map and appends it to the output file.

User Space Ring Buffer Management

In user space, a single ring buffer manager instance handles multiple ring buffer maps:

```
1 /* get handles for kernel ring buffers */
2 int events_ringbuf_fd =
3     bpf_object__find_map_fd_by_name(obj, "events_ringbuf");
4 int stackids_ringbuf_fd =
```

```
5       bpf_object__find_map_fd_by_name(obj, "stackids_ringbuf");
6
7   /* create ring buffer manager */
8   struct ring_buffer *ring_buf =
9       ring_buffer__new(events_ringbuf_fd,
10                       handle_event,
11                       handle_lost,
12                       NULL);
13
14  /* append second ring buffer map */
15  ring_buffer__add(ring_buf,
16                   stackids_ringbuf_fd,
17                   handle_stack,
18                   (void*)(long)stacktrace_fd)
19
20  /* poll all registered ring buffers */
21  ring_buffer__poll(ring_buf, timeout_ms);
```

Listing 5.8: User space implementation of the BPF ring buffer.

`ring_buffer__new` initialises the manager with the primary ring buffer. Whenever data becomes available, the associated callback `handle_event` is invoked automatically. The complete implementation of this function is provided in Appendix B. The `handle_lost` callback is triggered if the kernel reports lost records due to overflow. Error handling is discussed in Section 5.8.

Staged Transport Pipeline

Figure 5.3 illustrates the data transport pipeline and the data entities through which events propagate.

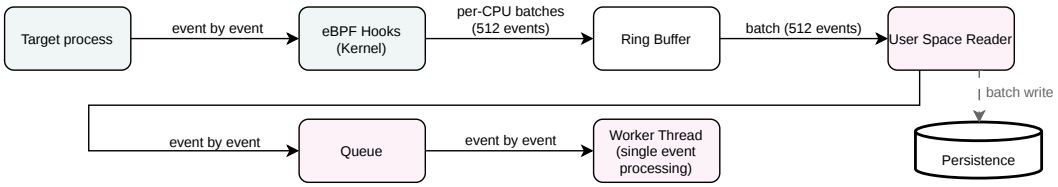

Figure 5.3: Data transport pipeline and backpressure decoupling: per-CPU batching in the kernel (512 events per batch), emission via the BPF ring buffer, persistence in user space, and expansion into single events for queue-based worker processing.

To sustain high event rates without blocking the traced application, *MemScope* employs a staged transport architecture that decouples kernel capture, persistence, and analysis.

Each memory operation triggers an eBPF hook in the kernel. Events are accumulated in batches of 512 events. Once full, the batch is submitted to the ring

buffer using the `bpf_ringbuf_output` helper; see Listing 5.9. After submission, the user space ring buffer thread polls for incoming batches. Upon arrival, the complete batch is written to the output file. Next, the thread iterates over the batch and inserts each event individually into the user space queue. The worker thread processes events from this queue in an event-by-event manner.

To avoid race conditions during concurrent execution across CPUs at kernel level, the per-CPU batching is implemented using a `BPF_MAP_TYPE_PERCPU_ARRAY` map. This map type provides one independent instance per CPU; therefore, each CPU maintains exactly one batch structure. Per-CPU buffering is required because eBPF programs may execute concurrently on multiple CPUs and allocation events can occur simultaneously across different execution contexts. While accessing or updating an eBPF maps is an atomic operation, reading, modifying and writing back a value is not synchronised. The eBPF execution environment does not permit arbitrary locking mechanisms, and only limited atomic operations are available. Thus, using a global shared buffer would require synchronisation that is not available in eBPF.

By employing a per-CPU map, each CPU maintains its own independent batch. This enables lock-free event emission, eliminates cross-core contention and prevents race conditions under high concurrency. Listing 5.9 illustrates the batch submission:

```
/* retrieve batch element */
struct event_batch *batch =
bpf_map_lookup_elem(&batch_map, &key);

/* retrieve next free spot in batch*/
element = &batch->events[batch->idx++];

/* initialise memory for element */
__builtin_memset(element, 0, sizeof(*element));

/* element is written */

/* if batch is full, submit to ring buffer */
if (batch->idx == BATCH_SIZE) {
    u32 bytes = batch->idx * sizeof(struct event);
    long ret = bpf_ringbuf_output(&events_ringbuf,
                                  batch->events,
                                  bytes,
                                  0);
    batch->idx = 0;
}
```

Listing 5.9: Kernel space batching and insertion to ring buffer.

Producer-Consumer Architecture

The system thus follows a producer-consumer architecture with two decoupling stages:

- Kernel →ring buffer
- Reader thread →worker thread

This separation ensures that I/O latency and metric computation do not interfere with ingestion throughput.

Ordering Trade-Off

Per-CPU batching improves throughput significantly. However, it may delay individual events and can cause short-lived deviations from the original execution order. The user space state machine is explicitly designed to tolerate such temporary reordering. Batching therefore represents a deliberate trade-off between performance and strict temporal ordering guarantees.

Map Dimensioning and Capacity Considerations

As already discussed in the section on ring buffer polling behaviour (5.6), when using BPF maps, it is necessary to consider that dimensioning directly influences the robustness and performance of the tracing system. Insufficient capacity may lead to event drops, failed insertions, or premature ring buffer overflow.

However, since BPF maps reside in kernel memory, their size must remain strictly bounded. If the eBPF program exceeds the resource bounds, map creation may fail. Consequently, maps sizing must remain conservative, even when considering worst case scenarios.

Therefore, all kernel maps (see Table 5.2) were dimensioned with respect to the expected upper bounds of the target system. Metrics that need to be considered are the amount of concurrent threads, the allocation frequency, and the stack diversity. The ring buffer size was chosen to tolerate short-term bursts in event production, while the hash map for temporary allocations and stack identifiers were dimensioned with respect to worst-case scenarios without unbounded growth.

Thread Architecture

The user space processing program consists of three threads:

1. **Ring buffer thread**
   Handles polling and invokes lightweight callback functions. This thread

contains minimal logic and avoids blocking operations to prevent buffer overflow.

2. **Worker thread**
   Processes events from the user space queue and performs updates of context-based statistics. The queue is implemented as a ring buffer protected by mutexes to ensure safe concurrent access.

3. **Control thread**
   Provides a minimal user interface enabling:

   - controlled termination of tracing and
   - runtime snapshot retrieval.

5.7 Persistence Strategy

As elaborated in Chapter 4.4.4, data persistence is required for multiple reasons. This section provides a short overview of persisted objects and their respective formats. It implements the persistence component of the system conceptually described in Chapter 4.2.

The raw event log is persisted immediately upon arrival in user space, before any further processing, as illustrated in Figure 5.3. Event batches are appended incrementally to a binary file. This reduces overhead and enables inspection of the raw log at any point during an ongoing measurement.

The second object persisted incrementally during tracing is the stack trace log. As explained in Section 5.5, whenever a new stack identifier is streamed to the user space, the corresponding stack trace is retrieved from the kernel BPF stack map and appended to a binary file. This approach ensures that all stack identifiers observed during runtime are available for later analysis.

In addition to raw events and stack traces, the contents of the `active_allocations` map can be exported at any time during the measurement. The purpose of this output is to provide developers with an interpretable overview of the current memory state. Therefore, the data is written in Comma-Separated Values (CSV) format. However, when exporting the contents of the `active_allocations` map during runtime, consistency must be ensured. The ring buffer thread and the worker thread continuously update the map. To avoid corruption of the data in the map or partial updates, the access to the data structure is synchronised. When the user requests a snapshot of the map during execution, the system creates a temporary copy using mutual exclusion. Once this copy is obtained, the system will export the snapshot. This guarantees that the exported snapshot represents a consistent view of the allocation state without damaging the data.

Similarly, the context-based statistics described in Section 5.5 are written to a CSV file after termination of the tracing process. Since these metrics are intended for a direct inspection and interpretation, a human-readable format is preferred.

At termination, *MemScope* generates two additional files. An information file that contains metadata about the measurement, such as measurement duration, total number of traced allocations, all degradation-handling relevant information (see Section 5.8), and an indication of whether the measurement is considered complete or incomplete. The second file is a snapshot of the `proc/<pid>/maps` file, read once during runtime. As explained in Section 5.3, this information is necessary for offline stack trace resolution.

For continuously growing data, such as the raw event log and the stack trace log, the binary format is chosen to minimise storage overhead and I/O costs. For objects that are bound in size, such as active allocations and the context-based statistics, CSV is used to enable immediate interpretability without additional tooling. Files that serve purely informational purposes and are inherently limited in size, such as the tracing metadata and the `proc/<pid>/maps` snapshot, are written in plain text format. This ensures accessibility even on the embedded target system directly.

5.8 Degradation Handling and System Robustness

The preceding sections described the nominal operation of *MemScope*. This section outlines how the system behaves under invalid input, resource constraints, and other non-ideal conditions.

Temporary inconsistencies and incomplete information are already addressed through the correlation strategy described in Section 5.4 (see Figure 5.2). The focus here is therefore on invalid event data, resource exhaustion, and controlled degradation behaviour.

In an event sourced system that reconstructs state from observed events, arbitrary event loss must be avoided, as it would directly compromise the reconstructed memory state. For this reason, events are discarded only in strictly defined situations. In most irregular cases, the system either enters a controlled degraded mode or records diagnostic information through counters. All counters are exposed to the user after termination of the tracing process in order to ensure transparency and support interpretation of the measurement results. The most important information about system degradation is summarised in the following table:

Condition	System Reaction
Kernel-Level Event Discard	
`malloc_ret: ptr == NULL`	Event discarded; `malloc_failed_counter++`
`free: ptr == NULL`	Event discarded; `free_null_counter++`
Ring buffer reservation failure	Event not emitted; Diagnostic message
Batch lookup failure	`dropped_counter++`
Missing or Corrupted Metadata	
Missing allocation size	`size = 0`; `size_unknown_counter++`
Stack trace retrieval failure	`stackid = -1`; `stackcapture_failed_counter++`
Temporary map lookup failure (`free`)	`size = 0; stackid = -1`; unmatched_frees_counter++
User Space Correlation Monitoring	
Pointer still active on `malloc`	Previous instance closed; `inferred_free++`
`Free` resolved from pending map	`out_of_order_free_resolved++`
`Free` without matching allocation	Insert into pending map; `out_of_order_free++`
`pending_frees` map collision	`pending_free_collision++`
Late metadata reconstruction	`late_resolved_free++`
Transport Integrity Monitoring	
Ring buffer overflow	`lost_batches++`
User space queue overflow	`queue_drops++`

Table 5.3: Controlled degradation and monitoring mechanisms in *MemScope*.

Kernel-Level Degradation

Only a small number of conditions result in an event being discarded in the kernel:

- If the pointer returned in `malloc_ret` is `NULL`, the allocation has failed and no memory was reserved. In this case, no event is generated and the `malloc_failed_counter` is incremented.
- If `free(NULL)` is invoked, the function has no effect. No event is generated and the `free_null_counter` is incremented.

A further degradation scenario concerns ring buffer submission. If space reservation in the BPF ring buffer fails, the event cannot be emitted and is therefore lost. Although this situation is not expected during normal operation, it is explicitly handled by incrementing the `dropped_counter`.

Failure to retrieve the per-CPU batch entry indicates an internal initialisation or configuration error rather than an observation artefact. In this case, the implementation emits a diagnostic message and aborts processing of the current event.

Handling of Missing or Corrupted Event Data

In addition to structural degradation, lookup operations may fail. These include

- failure to retrieve the stack trace via the BPF helper,
- failure to retrieve the allocation size from temporary maps and
- failed register or map lookups.

In these cases, default values are assigned and corresponding counters are incremented. Numerical fields such as size are set to 0, which does not corrupt aggregation statistics. If stack trace retrieval fails, the stack identifier is set to -1, and a dedicated counter is incremented. Events associated with stack identifier -1 are still aggregated, allowing the user to inspect such occurrences separately.

Monitoring Data Transfer Integrity

To ensure end-to-end consistency, events are counted at multiple stages of the pipeline.

In the kernel:

- every submitted allocation and deallocation event increments a submission counter and
- failed ring buffer submissions increment the `dropped_counter`.

In user space:

- all received events are counted,
- ring buffer overflow events increment `lost_batches`,
- worker thread processing increments processed-event counters and
- queue overflow increments `queue_drops`.

These counters allow verification that the number of processed events matches the number of emitted events. If inconsistencies are detected, the measurement is marked as incomplete.

User Space Consistency Monitoring

Additional counters monitor correlation behaviour:

- When a `malloc` replaces an already active allocation for the same pointer, the previous allocation is closed as an inferred free, and the `inferred_free` counter is incremented for the respective stack identifier.
- When a `malloc` is directly resolved by a `free` event from the `pending_free` map, the `out_of_order_free_resolved` counter is incremented.
- If a `free` has no matching allocation and is inserted into the `pending_frees` map, the `out_of_order_free` counter is incremented.
- If a collision occurs in `pending_frees`, the `pending_free_collision` counter is incremented.
- If allocation metadata must be reconstructed in user space due to missing kernel information, the `late_resolve_free` counter is incremented.

Startup Failures

Fatal initialisation errors, such as failure to attach a uprobe, result in immediate termination of the tracing process with a diagnostic error message.

In summary, *MemScope* integrates lock-free per-CPU kernel-level batching, ring buffer-based data transport, staged user space processing, and context-based aggregation to implement the architecture defined in Chapter 4.

6 Evaluation

6.1 Evaluation Setup

The evaluation was conducted on an embedded security camera platform based on an Ambarella CV22 system-on-chip running on a AArch64 architecture. *MemScope* was cross-compiled on a standard x86_64 Linux development machine and transferred to the target device via `SCP`. Due to the limited computational resources of the embedded security camera, no compilation was performed on the device itself. All execution on the device was limited to the pre-built tracing binary and test workloads. For configuration and data retrieval, the target was accessed remotely via Secure Shell Protocol (SSH). All stack trace post-processing and symbol resolution were performed offline on a host system using the corresponding AArch64 cross-toolchain utilities.

Unless stated otherwise, experiments were conducted using the default tracing configuration as described in Chapter 5. The evaluation focuses on the requirements defined in Chapter 3.

6.2 Evaluation of Functional Requirements

Dynamic Observation Control

The tracing framework is required to attach to and detach from a running target process at runtime, without requiring a restart of the traced application. This capability is provided by eBPF's dynamic instrumentation model, where uprobes can be installed and removed on-demand for a specific PID (Section 5.3.
eBPF was selected as tracing approach specifically to satisfy this requirement, whereas static instrumentation approaches were excluded (see Chapter 5).

During evaluation, *MemScope* was attached to a running process without restart and immediately started emitting allocation and deallocation events. The detachment of the *MemScope* stopped the tracing while the process continued.

Therefore, runtime controllability of observation is confirmed and **the requirement is fulfilled.**

Allocation Event Acquisition

The framework is required to capture allocation and deallocation events and maintain the relationship between both operations to reconstruct memory behaviour.
In this work, event acquisition is realised by attaching uprobes/uretprobes to the libc functions `malloc` and `free`, as described in Section 5.3. This ensures that all heap memory operations routed through libc are observed, regardless of whether the functions are directly called or via wrappers.

Function validation was performed using a test program that performs allocation and deallocation memory operations. Reconstruction of the memory state was consistent after termination.
Therefore, the acquisition of allocation events is confirmed and **the requirement is fulfilled.**

Allocation Origin Identification

The framework must provide the calling context responsible for each allocation and deallocation event.
MemScope captures the allocation context as a user space stack trace at the moment of each allocation event and stores the stack trace for offline resolution, as described in Section 5.5.

During evaluation with a test program, every allocation's calling context was provided and was successfully resolved to function names and code lines, for frames belonging to the target application (see example in Section 5.5). As described in that section, resolution may be limited if debug symbols are not available. However, it should be noted that this is not a weakness of the tracing framework itself, but rather is due to circumstances in embedded development environments.
Consequently, **the requirement is fulfilled.**

Runtime Memory State Inspection

The tracing framework must provide access to the currently active allocations in the target application and their lifetimes at any point during the tracing.
This requirement is realised by maintaining an `active_allocations` map in the user space, representing all currently active allocation instances at a given point in time (Section 5.4 and 5.5). This state can be exported as consistent snapshot (see Section 5.5) at anytime during the tracing.

During evaluation with a test program, snapshots could be produced while execution was continued, and the final snapshot at termination matched the expected state of the memory.
Therefore, inspection of the memory state during runtime is possible at any point in time and correct, and **the requirement is fulfilled.**

Memory Growth Observability

The tracing framework must provide time-resolved allocation data enabling the analysis of memory consumption trends.
To enable the observability of memory growth, *MemScope* maintains incremental, context-based metrics during runtime (Section 5.5), including active bytes, cumulative allocation volume, and a history of peaks with corresponding timestamps.

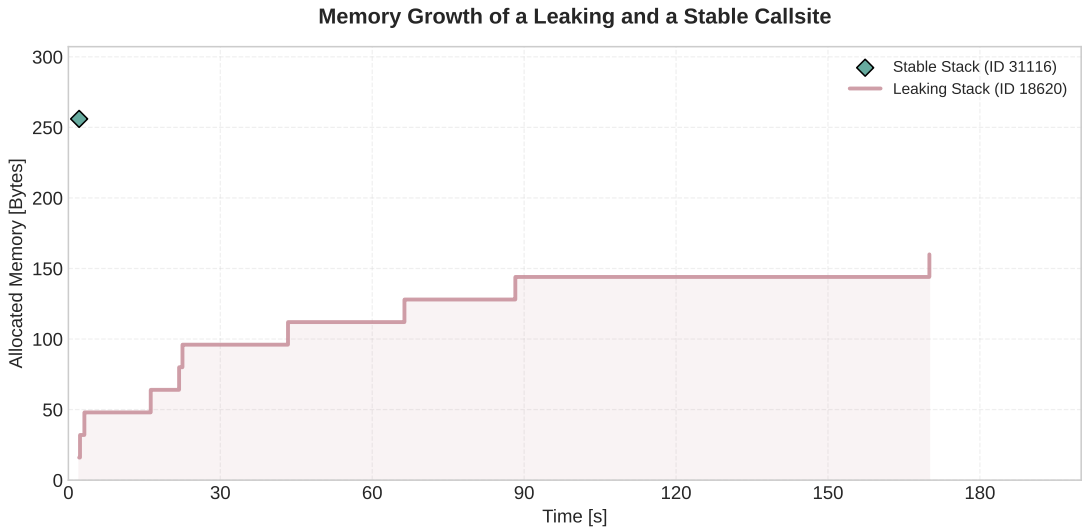

Figure 6.1: Memory growth over time for a stable and a leaking call site.

This requirement was validated using a test program that has two call sites.
One call site constantly allocates 256 Bytes of memory and directly releases it. Therefore, the peak metric is set to 256 Bytes at first observation and remains unchanged, as no subsequent allocation exceeds this value.
The second call site instead allocates 16 Bytes and will randomly produce leaks, leading to memory growth. Each time the amount of memory held by this call site exceeds the previous peak, the new peak will be stored with the timestamp at the time. The metrics accumulated by *MemScope* corresponded exactly to the expected values and are illustrated in Figure 6.1.
The requirement is fulfilled.

6.3 Evaluation of Non-Functional Requirements

Completeness of Observation

The tracing framework must observe and process every memory operation performed by the observed application, as losing events during the tracing would compromise the reconstructed memory state and significantly hinder the memory behaviour analysis.

To evaluate this requirement, a controlled workload was implemented that performs a predefined and known number of allocation and deallocation operations at a realistic frequency. The workload includes a short initial waiting phase to allow *MemScope* to attach before any memory operations are executed, ensuring that the full lifecycle of allocations is captured.
After the test program completed, the total number of allocation and deallocation events reported by *MemScope* was compared to the expected number of operations executed by the workload. The observed event counts matched with the expected values. Furthermore, no ring buffer overflows, lost batch reports, or user space queue drops were recorded during execution.

Under these controlled conditions, the framework demonstrated complete observation of all memory operations. Therefore, completeness was confirmed for the evaluated workload and **the requirement is fulfilled.**

Performance

1. The system must impose only limited and acceptable runtime overhead on the target process. High runtime overhead would distort execution behaviour and invalidate the collected tracing data. eBPF was selected as tracing mechanism partly because its execution model is designed to introduce low overhead by executing small, verified programs directly within the kernel (see Section 5.1.1).

   To evaluate runtime overhead introduced by *MemScope*, a controlled workload was executed on the target device both without and with *MemScope* attached. The test program repeatedly performs a fixed number of allocation and deallocation operations. For the different runs, the workload intensity (operations per second) is varied to compare how overhead evolves. The workload intensity was determined based on the baseline execution without tracing. Due to scheduling effects and dynamic CPU frequency scaling on the embedded platform, the effective event rate varied slightly between runs. For the measurements with attached framework the effective event rate decreases compared to the baseline runs. This is due to the overhead the framework induces on the system. Therefore, table 6.1 reports the measured effective event rate rather than the nominally configured rate.

Runtime was measured using the `/usr/bin/time` utility. The wall-clock time (`real`) was used as the primary performance metric. Additionally, user space CPU time (`user`) and system CPU time (`sys`) were recorded. Each configuration was executed three times, and the median runtime was used for evaluation. The relative runtime overhead was calculated by comparing the execution time of the program without (T_{baseline}) and with tracing (T_{traced}):

$$\text{Overhead} = \frac{T_{\text{traced}} - T_{\text{baseline}}}{T_{\text{baseline}}} \tag{6.1}$$

Tracing	**Events (total)**	**Observed rate (op/s)**	**Real (s)**	**User (s)**	**Sys (s)**	**Overhead (%)**
OFF	1 Mio.	2128	470.36	1.38	9.68	–
ON	1 Mio.	1927	519.03	2.55	33.59	10.35
OFF	1 Mio.	9709	103.31	0.83	8.44	–
ON	1 Mio.	6757	148.05	2	28.45	42.77
OFF	1 Mio.	19231	52.36	0.7	8.11	–
ON	1 Mio.	11236	89.31	1.91	27.06	70.57

Table 6.1: Runtime overhead measurement results.

To characterise scalability behaviour, three workload configurations were evaluated, corresponding to approximately 2000, 10000, and 20000 allocation events per second under baseline conditions.
As shown in Table 6.1, the measured runtime overhead increases with event rate. For the lowest workload, the relative overhead is approximately ∼10 %. At medium intensity, overhead rises to roughly ∼40 %, and at the highest tested rate, it reaches approximately ∼70 %. The highest workload exceeds typical expected workloads and serves as a stress-test scenario.
The results show that within the tested range, runtime overhead follows an approximately proportional trend with respect to event rate. No indications of instability or exponential scaling behaviour were observed.
Furthermore, the execution times show that the increase in runtime is primarily attributable to increased system time (kernel execution), while user space CPU time remains comparatively stable.
Specifically, the rise in system time reflects the overhead of frequent transitions between user and kernel space (context switches), as each allocation event generates a trap that forces the CPU to execute the BPF handler. This behaviour is consistent with the tracing architecture, as eBPF programs execute within the kernel context.

No event drops or transport integrity violations were detected at any workload.

The medium workload configuration most closely reflects the expected operating conditions of the target deployment scenario. Although a runtime overhead of approximately ∼40 % is non-negligible, *MemScope* is intended primarily for diagnostic and debugging sessions rather than continuous production monitoring. As no functional instability was introduced, the observed overhead is considered acceptable for the intended use case.
Therefore, **the requirement is fulfilled.**

2. The system must operate with bounded runtime memory consumption, even for extended monitoring sessions. This requirement concerns only in-memory state and buffering, and explicitly excludes persistent event logs, whose size grows with measurement time by design.
Bounded memory usage is ensured by design: kernel-side state is stored exclusively in eBPF maps with fixed capacities (see Section 5.6). In user space, the in-memory state consists primarily of maps of currently active allocations, pending `free`s, and per-context statistics. These structures grow only with the number of concurrently active allocations and distinct allocation contexts and do not increase unboundedly over time. Historical metrics are stored in fixed-size buffers per allocation context.

   To evaluate this requirement, the resident memory usage Maximum Resident Set Size (Max RSS) of the tracing process was monitored during runtime using the `/usr/bin/time` utility. The Max RSS value reflects the actual amount of physical memory occupied by the process during execution and therefore serves to verify that internal data structures, such as BPF maps and ring buffers maintain a fixed capacity without growing over time.
Contrary to the evaluation of the overhead, now the rate of operations per second was kept stable but the amount of events was increased. *MemScope* was executed for different durations and the Max RSS value was recorded.

Run	**Events (total)**	**Duration (s)**	**Event Rate (op/s)**	**Max RSS (KB)**
1	1 Mio.	146.05	6847	95664
2	6 Mio.	833.59	7198	95632
3	20 Mio.	2730.94	7323	95648

Table 6.2: Max RSS of *MemScope* over increasing monitoring durations, demonstrating bounded memory usage.

Although the table shows minor variations in the effective event rate, all measurements were executed with the exact same test program. This behaviour is attributed to scheduler granularity and warm-up effects such as cache initialisation and dynamic frequency scaling on the embedded platform.

The measurement did not show progressive growth in memory usage proportional to runtime duration. Instead, memory consumption stabilised, indicating that no unbounded runtime memory growth occurred.
Based on these observations, **the requirement is fulfilled.**

Concurrency Safety

1. The system must be capable of tracing multi-threaded applications in a thread-safe manner. Concurrency considerations were incorporated throughout the design, particularly in the separation of kernel and user space responsibilities (see Section 5.4) in the staged event transport architecture (Section 5.6).
   In the kernel, per-CPU batching ensures that events generated concurrently on different threads do not introduce race conditions. In the user space, event ingestion and processing are decoupled via a producer–consumer model (see 5.6), and shared data structures are protected appropriately to maintain state consistency. To validate this requirement, a multi-threaded test workload was executed, generating concurrent allocation and deallocation events across multiple threads. The reconstructed memory state and event statistics were consistent with the expected behaviour of the workload. No race conditions, state corruption, or inconsistent counters were observed. Therefore, **the requirement is fulfilled.**

2. The tracing framework must tolerate events originating from multiple CPU cores without deviating from its defined behaviours. This requirement was addressed by the use of per-CPU buffers, effectively eliminating cross-core contention during event aggregation (Section 5.6) and the design of the kernel and user space event processing (Section 5.4).
   Evaluation experiments were conducted on a multi-core embedded platform, as described in Section 6.1. During extended monitoring sessions, *MemScope* remained stable and no systematic corruption of event batches or cross-core race conditions were observed. Based on both the design and empirical validation, *MemScope* is considered robust against concurrent event emission from multiple CPU cores. **The requirement is fulfilled.**

Robustness

The tracing framework must maintain a consistent representation of the memory state even when tracing begins after the start of the program.
When attaching to a running process, the allocation history is inherently partial because allocations may have been created before the observation window. Consequently, *MemScope* does not attempt to reconstruct a historically complete memory state. Instead, it maintains an internally consistent state representation that is explicitly relative to the observation window using a conservative correlation strategy, as explained in Sections 4.4.2 and 5.4. In addition, exceptional cases such as unmatched `free`s, missing metadata, or stack capture failures are explicitly detected, counted, and isolated through degradation handling mechanisms described in Section 5.8, preventing silent state corruption.
Therefore, even when tracing begins after program start, the framework maintains a consistent memory state within the boundaries of observable information. **The requirement is fulfilled.**

Operational Stability

The system must operate reliably during extended observation periods while maintaining internal consistency and without degrading its operational behaviour.

Operational stability is ensured through a combination of architectural design decisions described in Chapter 5. Kernel data structures are strictly limited in size (Section 5.6), preventing uncontrolled memory growth. User space state management uses fixed-capacity buffers and bounded historical metrics (Section 5.5), ensuring that runtime memory consumption remains stable over time. Concurrency safety mechanisms (Section 5.3) prevent race conditions under multi-threaded and multi-core execution, while the degradation handling (Section 5.8) ensures that exceptional scenarios are explicitly detected and isolated rather than silently corrupting internal state. To validate this requirement, *MemScope* was executed during extended monitoring sessions on the target embedded platform. During prolonged runtime, no progressive increase in memory consumption, deadlocks, or instability attributable to the framework were observed. No indications of internal state corruption within *MemScope* were detected during extended monitoring.
Based on the system design and empirical observation, **the requirement is fulfilled.**

Operability

The system must require minimal configuration and no modification of the target application.

Operability was a primary design consideration when selecting the tracing mechanism (Section 5.1). eBPF uprobes were chosen as a hooking mechanism, because they allow dynamic attachment to user space functions without requiring any modification of the target application's source code or build process.
The deployment of *MemScope* requires only the tracing binary and the libbpf dependency to be present on the target device. No changes to the observed application, no recompilation, and no intrusive configuration steps are required. After integration into the software distribution, *MemScope* can be executed directly and attached to running processes without further system modification. **The requirement is fulfilled.**

Portability

1. The tracing framework must be operable across different devices and software variants within the same Linux-based product family.
   Portability is primarily ensured through the use of eBPF as the underlying tracing technology. eBPF is integrated into the Linux kernel and provides a standardised interface for dynamic program attachment that is independent of specific hardware platforms within the same architecture family (Section 5.1). As long as the target system runs a Linux kernel version that provides eBPF support and exposes the required tracing capabilities, the framework can be deployed without architectural modification.
   On systems where eBPF support is not enabled by default, the Linux kernel may need to be configured or rebuilt to provide the required functionality. This configuration step is platform-specific but does not require any modification of *MemScope* or the target application. Once eBPF support is available, portability across devices within the product family is achieved by deploying the tracing binary compiled for the target architecture and the corresponding libbpf dependency. No further adaptation to individual applications or software variants is required. Since the framework relies exclusively on standard Linux kernel mechanisms and does not depend on device-specific instrumentation, the portability **requirement is fulfilled.**

2. The system should not depend on programming language specific instrumentation and operate on common runtime allocation interfaces.
   MemScope relies solely on standard eBPF and libbpf interfaces (see Chapter 5.3). Memory operations are traced by attaching uprobes to the allocation symbols provided by the libc, such as `malloc` and `free`.
   These allocation interfaces are commonly used across multiple programming languages, including C, C++ and indirectly by higher-level languages such as Python. By tracing the libc allocation symbols instead of language-specific memory management functions, the framework remains independent of compiler instrumentation, language runtime modifications, or source

code adaptions.
If an application eventually performs memory allocation via the standard libc interfaces, it can be observed without additional integration effort. The **requirement is fulfilled.**

6.4 Practical Use Cases

Before the development of the presented tracing framework, identifying memory leaks or unintended memory growth required considerable manual effort and strict development discipline:
Developers had to implement dedicated wrapper functions for all memory allocation and deallocation operations in order to trace heap usage. Only allocations performed through these wrappers could be monitored. If a developer forgot to use the wrapper in a specific code location, the corresponding allocation would remain invisible to the tracing mechanism. As a result, potential memory leaks originating from such code remained undetected.
In addition, memory analysis was typically performed using tools such as `mtrace`. This required restarting the application with tracing enabled and let it run for an extended period of time, in order to capture sufficient runtime behaviour. The generated trace logs could reach sizes in the order of tens of gigabytes. These logs then had to be analysed manually. The data format was not easily readable, and extracting meaningful conclusions required substantial time and experience. Although this approach eventually led to identifying the source of the problem, the routine to achieve this was highly time consuming.

The introduction of *MemScope* significantly simplifies this process. Once eBPF and uprobes support is available in the kernel, the compiled tracing binary can be transferred to the target device and executed without modifying or restarting the observed application. Tracing can begin at any arbitrary point in time, even for applications that have already been running for an extended period.
At runtime, *MemScope* allows exporting the current set of active allocations, providing a consistent snapshot of the memory state. Such snapshots can be generated multiple times and compared to observe evolving behaviour. After the tracing has terminated, the collected data can be analysed in different ways. The CSV outputs enable manual inspection of aggregated statistics, while the offline Python-based analysis tool supports heuristic-driven investigation. Furthermore, the complete binary event log preserves every individual allocation and deallocation event. These events can be resolved to stack traces using the provided stack trace resolution script, allowing detailed inspection down to the source code.
Another improvement concerns applicability. Previously, memory monitoring was limited to a single application that had been explicitly instrumented with memory operation wrappers.

With the eBPF-based approach, it is possible to attach to any user space process on the device. This enables analysing memory behaviour not only in the originally instrumented application, but also in other services or auxiliary processes running on the system.

Overall, the new workflow reduces operational overhead, removes the need for source-level instrumentation, and enables flexible, runtime-based memory analysis across the entire system.

6.5 Summary

The evaluation demonstrates that the implemented tracing framework satisfies all defined functional and non-functional requirements. Core functionality, including time-resolved observability of memory growth, was validated and demonstrated. Performance measurements confirmed that *MemScope* operates reliably on a resource-constrained embedded platform. Although runtime overhead increases with event rate, no instability, event loss, or structural degradation was observed. Furthermore, the evaluation confirmed bounded memory consumption, rendering the framework suitable for longer observation periods.

The evaluation confirms that *MemScope* provides reliable and structured runtime allocation data suitable for memory behaviour analysis in embedded Linux environments.

7 Conclusion

This chapter summarises the contributions and key findings of the thesis. The limitations of the work are discussed along with the potential approaches for further development that arise from these limitations.

7.1 Summary

This thesis addressed the challenge of memory observability in long-running embedded systems, where manual memory management in C/C++ often leads to undetected memory growth and instability. The objective of designing and implementing a lightweight, runtime-attachable observability framework for embedded Linux systems was achieved.

The framework operates without requiring modification or recompilation of the target application and can be attached to running processes under realistic embedded constraints. The evaluation demonstrated system stability, bounded memory consumption, and manageable runtime overhead.

MemScope provides time-resolved growth observability, consistent state reconstruction, and a significant reduction in diagnostic effort required to identify problematic allocation contexts. This substantially enhances the practical workflow of memory analysis in embedded systems, when compared to manual labour-intensive correlation of allocation behaviour.

Its lightweight design and adaptability make it deployable on resource-constrained devices where traditional heavyweight analysis tools are infeasible. The framework therefore provides a practical and reliable diagnostic tool for systematic memory behaviour analysis in embedded Linux environments.

7.2 Limitations

Despite the demonstrated feasibility and stability of *MemScope*, several limitations remain. The present implementation focuses exclusively on dynamic heap allocations in user space and does not trace kernel space allocations.
Currently, the tracing is limited to the `malloc` and `free` symbols. Additional allocation mechanisms, such as `calloc`, `realloc`, or `mmap` and `break` are not yet covered.

Furthermore, the framework depends on the availability of eBPF and uprobe support within the target kernel and resolving the full stack trace requires access to appropriate debug symbols during offline analysis.

Although performance evaluation showed acceptable overhead, very high event rates may introduce increased runtime overhead and backpressure effects.
The current implementation represents a prototype-level system. Its intention is to demonstrate feasibility rather than a fully hardened production deployment.

7.3 Future Work

Future work may extend *MemScope* to support additional allocation APIs. Integrating kernel-level tracepoints could also broaden observability beyond user space heap allocations.

Further performance profiling may identify dominant overhead sources, particularly within user space event processing. This could indicate whether targeted optimisations of batching, queue management, or event transport mechanisms could improve scalability under high allocation rates.

From a system integration perspective, the introduction of remote data transport, automated data compression, or client-server architectures may simplify deployment in constrained embedded environments.

Finally, automated post-processing approaches may further reduce manual analysis effort and enhance the practical applicability of *MemScope*.

Appendices

This appendix provides representative excerpts of the implementation, covering kernel hooks, user-space event processing, and parts of the Python-based offline post-processing. For brevity, only representative components are shown. The complete implementation is provided separately.

A Kernel Hook Implementation

The following code excerpt presents the kernel implementation of the `malloc_ret` hook, shown as a representative example of the hook mechanisms used throughout *MemScope*. The `free` hook follows the same structural pattern. The `malloc_entry` hook only stores the size argument into a map.

```

/* ------------------ MALLOC RETURN ------------------ */
// read returned pointer (RC), lookup size from tmp_size_map,
// capture user stackid, emit event to batchmap / ringbuffer

SEC("uretprobe/malloc")
int uprobe_malloc_ret(struct pt_regs *ctx)
{
    u32 key = 0;
    u8 *stop_tracing = bpf_map_lookup_elem(&cntrl, &key);
    // set by user space, indicates stop of tracing
    if (stop_tracing && *stop_tracing == 1) {
        return 0;
    }
    u64 ts_ns = bpf_ktime_get_ns();
    s32 stackid;
    u64 size;
    struct event *malloc;
    struct stats_global *glst = bpf_map_lookup_elem(&global_stats,
    &key);
    u64 pidtgid = bpf_get_current_pid_tgid();
    u32 pid = pidtgid & 0xFFFFFFFF;
    u32 tid = pidtgid >> 32;

    u64 ptr = (u64)PT_REGS_RC(ctx); // malloc ret value
    if (!ptr){ // failed malloc, was never executed, no stats
        if (glst) {
            glst->malloc_failed_counter++;
        }
        return 0;
    }
    u64 *sizep = bpf_map_lookup_elem(&tmp_size_map, &pidtgid);
    if (!sizep){
        size = 0;
        if (glst != 0) {
```

```
35              glst->size_unknown_counter++;
36          }
37      } else {
38          size = *sizep;
39      }
40      // capture user stack & get hash to access it in map
41      int sid = bpf_get_stackid(ctx, &stack_traces, BPF_F_USER_STACK
    );
42      if (sid < 0) {
43          // stackid broken, still gather stats
44          stackid = STACKID_UNKNOWN;
45          if (glst) {
46              glst->stackcapture_failed_counter++;
47          }
48      } else {
49          stackid = (s32) sid;
50      }
51      // check if stackid is known or should be send to userspace
52      u8 *known = bpf_map_lookup_elem(&known_stackids, &stackid);
53      if (!known){
54          u8 one = 1;
55          bpf_map_update_elem(&known_stackids, &stackid, &one,
    BPF_ANY);
56          bpf_ringbuf_output(&stackids_ringbuf, &stackid, sizeof(
    stackid), 0);
57      }
58      // store in tmp_allocs_map for size lookup in free
59      struct alloc_tmp malloc_info = {
60          .malloc_stackid = stackid,
61          .size = size,
62      };
63      u64 ret = bpf_map_update_elem(&tmp_allocs_map, &ptr, &
    malloc_info, BPF_NOEXIST);
64      if (ret < 0) {
65          bpf_printk("malloc: couldn't update tmp allocs map. exit."
    );
66          if (glst) {
67              glst->double_ptr_counter += 1;
68          }
69      }
70      // update global stats
71      if (glst != 0) { // check required by kernel
72          glst->malloc_counter += 1;
73          glst->current_bytes = glst->current_bytes + size;
74          if (glst->max_bytes < glst->current_bytes){
75              glst->max_bytes = glst->current_bytes;
76          }
77      }
78      // reserve mem in batch struct array for event
79      struct event_batch *batch = bpf_map_lookup_elem(&batch_map, &
    key);
```

```
80      if (!batch){
81          bpf_printk("malloc: couldn't find current batch. exit.");
82          // cleanup before early return
83          bpf_map_delete_elem(&tmp_size_map, &pidtgid);
84          return 0;
85      }
86      if (batch->idx >= BATCH_SIZE){
87          batch->idx = 0;
88      }
89      // lookup next free address in batch for this event
90      malloc = &batch->events[batch->idx++];
91      // initialise with 0
92      __builtin_memset(malloc, 0, sizeof(*malloc));
93
94      /* fill event */
95      malloc->ts_ns = ts_ns;
96      malloc->pid = pid;
97      malloc->tid = tid;
98      malloc->ptr = ptr;
99      malloc->size = size;
100     malloc->stackid = stackid;
101     malloc->type = EVENT_TYPE_MALLOC;
102
103     // batch full, submit all events to ringbuf
104     if (batch->idx == BATCH_SIZE) {
105         u32 bytes = batch->idx * sizeof(struct event);
106         long ret = bpf_ringbuf_output(&events_ringbuf, batch->
    events, bytes, 0);
107         // failed to write batch to ringbuf
108         if (ret < 0) {
109             batch->dropped += 1;
110             bpf_printk("malloc_ret: Failed to write batch into
    events ringbuf.");
111             bpf_printk("malloc_ret: Batch dropped at: %llu", ts_ns
    );
112         }
113         batch->idx = 0;
114     }
115     /* CLEANUP */
116     bpf_map_delete_elem(&tmp_size_map, &pidtgid);
117     return 0;
118 }
```

Listing A.1: Malloc return hook. Kernel implementation.

B User Space Ring Buffer Callback Implementation

The following code excerpt presents the user-space ring buffer callback function. It handles both ring buffers. The event batches are also put into the user-space queue for the worker thread.

```
/***** callback function for events ringbuffer *****/
// keep short, avoid blocking
static int handle_event(void *ctx, void *data, size_t data_sz)
{
    (void)ctx; // ignore unused parameter warning
    if (data_sz < sizeof(struct event)) {
        return 0;
    }
    int num_events = (int)(data_sz/sizeof(struct event));
    valid_batches += (int)(num_events/512); // count per batch
    ssize_t ret = write(eventslog_fd, data, data_sz); // writing
    whole batch to logfile
    if (ret == -1){
        // only set flag and let worker print to avoid we don't
    block
        error_flag = 1;
        saved_errno = errno;
    }
    // write events into queue for worker thread
    struct event *events = data;
    struct event e;
    pthread_mutex_lock(&queue.lock);
    for(int i=0; i<num_events; i++) {
        e = events[i];
        push_to_queue(&e);
        callback_events++;
    }
    pthread_cond_signal(&queue.cond);
    pthread_mutex_unlock(&queue.lock);
    return 0;
}
```

Listing B.1: Ring buffer callback function in the user space.

C User Space State Management

The following excerpt presents the relevant section of the worker thread implementation, specifically the logic responsible for correlating `malloc` and `free` events, updating the internal state maps, and maintaining associated allocation statistics.

```
if (e.type == EVENT_TYPE_MALLOC) {
    worker_events++;
    uint64_t now = (get_time_ns() - start_ts_ns)/1000000ULL;
    // get struct; if not exisiting it is initialized with 0
    struct stackid_stats *s = get_stack_stats(e.stackid);
    // look for pending free wiht ptr in pending free map
    struct allocs_map *pfree = NULL;
    pthread_mutex_lock(&pfrees_lock);
    HASH_FIND(hh, pending_frees, &e.ptr, sizeof(uint64_t), pfree);
    if (pfree) {
        if (pfree->ts_ns > e.ts_ns){
            // if really transient order remove, otherwise leave
            HASH_DEL(pending_frees, pfree);
        }
    }
    pthread_mutex_unlock(&pfrees_lock);
    if (pfree) {
        if (pfree->ts_ns > e.ts_ns){
            // if in pending free, update stats for free & malloc
            update_stats(s, true, true, e.size, e.size, e.ts_ns,
   pfree->ts_ns, now);
            free(pfree);
            outoforder_free_resolved += 1;
        }
    }
    else {
        /*0. see if ptr already in active alloc map */
        struct allocs_map *existing = NULL;
        pthread_mutex_lock(&allocs_lock);
        HASH_FIND(hh, active_allocs, &e.ptr, sizeof(uint64_t),
   existing);
        if (existing) {
            HASH_DEL(active_allocs, existing);
            struct stackid_stats *s_old = get_stack_stats(existing
   ->stackid);
            if (s_old->current_bytes >= existing->size) {
                s_old->current_bytes -= existing->size;
            }
            else{
                s_old->current_bytes = 0;
            }
            // doesn't count as "normal free"
            // since it was assumed and not traced
```

```
41              s_old->inferred_frees += 1;
42              free(existing);
43          }
44          pthread_mutex_unlock(&allocs_lock);
45
46          /* 1. update active_allocs map */
47          struct allocs_map *m = malloc(sizeof(*m));
48          m->ptr = e.ptr;
49          m->size = e.size;
50          m->stackid = e.stackid;
51          m->ts_ns = e.ts_ns;
52          m->age = get_time_ns();
53          m->first_seen_event_id = worker_events; // event id
54          pthread_mutex_lock(&allocs_lock);
55          HASH_ADD(hh, active_allocs, ptr, sizeof(uint64_t), m);
56          pthread_mutex_unlock(&allocs_lock);
57          // update growth history
58          update_stats(s, true, false, e.size, 0, 0, 0, now);
59      }
60  }
61  else if ( e.type == EVENT_TYPE_FREE) {
62      worker_events++;
63      /* 1.search in active_allocs map */
64      struct allocs_map *alloc = NULL;
65      pthread_mutex_lock(&allocs_lock);
66      HASH_FIND(hh, active_allocs, &e.ptr, sizeof(uint64_t), alloc);
67      if (alloc) {
68          HASH_DEL(active_allocs, alloc);
69      }
70      pthread_mutex_unlock(&allocs_lock);
71      if (alloc) {
72          if (e.size == 0 && e.stackid == STACKID_UNKNOWN) {
73              e.size = alloc->size;
74              e.stackid = alloc->stackid;
75              late_resolved_frees += 1;
76          }
77          struct stackid_stats *s = get_stack_stats(alloc->stackid);
78          update_stats(s, false, true, 0, e.size, alloc->ts_ns, e.
    ts_ns, 0);
79          free(alloc);
80      }
81      else {
82          outoforder_free += 1;
83          // found no active allocation for this free
84          // check if there is already a pending free for this
    pointer
85          struct allocs_map *existing = NULL;
86          pthread_mutex_lock(&pfrees_lock);
87          HASH_FIND(hh, pending_frees, &e.ptr, sizeof(uint64_t),
    existing);
88          if (existing) {
```

```
89              pending_free_collision += 1;
90              if (e.ts_ns > existing->ts_ns){
91                  existing->size = e.size;
92                  existing->stackid = e.stackid;
93                  existing->ts_ns = e.ts_ns;
94              }
95              pthread_mutex_unlock(&pfrees_lock);
96              continue;
97          }
98          else{
99              pthread_mutex_unlock(&pfrees_lock);
100             // put in pending free map
101             struct allocs_map *f = malloc(sizeof(*f));
102             f->ptr = e.ptr;
103             f->size = e.size;
104             f-> stackid = e.stackid;
105             f->ts_ns = e.ts_ns;
106             pthread_mutex_lock(&pfrees_lock);
107             HASH_ADD(hh, pending_frees, ptr, sizeof(uint64_t), f);
108             pthread_mutex_unlock(&pfrees_lock);
109         }
110     }
111 }
```

Listing C.1: Excerpt of user-space worker thread that is responsible for state management.

D Offline Stack Trace Resolution

The following code excerpt presents the Python-based offline script used to resolve the stack identifiers into concrete instruction addresses by automatically processing the output file generated by *MemScope*, following the procedure outlined in Chapter 5.5.

```
#!/usr/bin/env python3

# pass arguemnts like:
#       ./resolvestacktrace.py \
#         --stacktrace stacktrace.bin \
#         --map maps.txt \
#         --sysroot /home/juliane/Development/thesis/code/eBPF/ \
#         --addr2line aarch64-linux-gnu-addr2line
#
#   sysroot is the location of the tracing data

import csv
import argparse
import struct
import subprocess
from pathlib import Path
from elftools.elf.elffile import ELFFile

MAX_STACK_DEPTH = 20    # adjust if changed in eBPF
# how stacktrace is build, 4 Byte stackid + MAX_STACK_DEPTH * 8 byte adresses
STACKREC_FMT = f"<i4x{MAX_STACK_DEPTH}Q"
STACKREC_SIZE = struct.calcsize(STACKREC_FMT)

##### parsing all files into a format we can work with #####
# extract data from /proc/pid/maps which was copied into maps.txt
def parse_maps (maps_file):
    #returns list of (start, end, path)
    mappings = []
    #open file and iterate lines
    with open(maps_file) as f:
        for line in f:
            parts = line.strip().split()
            # expect to have four columns, if more, skip line
            if len(parts) < 4 or parts[-1].startswith("["):
                continue
            addr = parts[0] # first elem is addresses
            path = parts[-1] #last elem is path
            #split first column in two and convert hex to integer
            start, end = [int(x, 16) for x in addr.split("-")]
            mappings.append((start, end, path))
    return mappings

```

```
# extract data from stacktraces.bin
def load_stacktraces(stacktrace_file):
    stacktraces = {}
    with open(stacktrace_file, "rb") as f:
        while True:
            data = f.read(STACKREC_SIZE)
            if not data:
                break
            if len(data) < STACKREC_SIZE:
                print(f"Warning incomplete record ({len(data)}
    bytes), skipping")
                break
            stackid, *ips = struct.unpack(STACKREC_FMT, data)
            # if less than 20 frames captured, remove empty
    entries
            ips = [ip for ip in ips if ip != 0]
            stacktraces[stackid] = ips
    return stacktraces

#helper to calculate address offsets for addr2line
# so script can handle PIE and non-PIE
# returns virtual start address of executed segment of path
def get_elf_text_vaddr(path):
    with open(path, "rb") as f:
        elf = ELFFile(f)
        for seg in elf.iter_segments():
            # looking for pt_load, the segment that is loaded into
     memory by the loader
            if seg['p_type'] == 'PT_LOAD' and seg['p_flags'] & 1:
                return seg['p_vaddr']
        raise RuntimeError(f"No executable LOAD segment found in {
    path}")

# resolve symbols
def addr2line (addr2line_bin, binary, addr):
    # resolves list of addresses -> list of strings
    if not addr:
        print("No addresses found for addr2line")
        return[]
    p = subprocess.run(
        [addr2line_bin, "-f", "-C", "-e", binary, f"0x{addr:x}"],
        capture_output=True,
        text=True,
    )
    return p.stdout.strip().splitlines()

##### main #####
def main():
    ap = argparse.ArgumentParser("Leak Analyzer")
    ap.add_argument("--stacktrace", required = True)
    ap.add_argument("--map", required = True)
```

```
    ap.add_argument("--addr2line", default = "aarch64-linux-gnu-
    addr2line")
    ap.add_argument("--sysroot", required = True)
    args = ap.parse_args()

    stacktraces = load_stacktraces(args.stacktrace)
    mappings = parse_maps(args.map)
    elf_vaddr_cache = {} # caching because this is expensive
    with open("stacktrace.txt", "w") as log:
        for stackid, ips in stacktraces.items():
            log.write(f"\n--- stackid {stackid} ---\n")
            for ip in ips:
                for start, end, path in mappings:
                    if start <= ip < end:
                        local_path = Path(args.sysroot) / Path(
    path).name
                        if local_path not in elf_vaddr_cache:
                            elf_vaddr_cache[local_path] =
    get_elf_text_vaddr(local_path)
                        # calculate offset
                        elf_vaddr = elf_vaddr_cache[local_path]
                        loadbias = start - elf_vaddr
                        elf_addr = ip - loadbias
                        log.write(f"{local_path}\n")
                        # now resolve symbols
                        for line in addr2line(args.addr2line,
    local_path, elf_addr):
                            log.write(f"    {line}\n")
                        break # mapping found

if __name__ == "__main__":
    main()
```

Listing D.1: Python script for offline stack trace resolution.

References

Billimoria, K. N. (2024, February 29). *Linux Kernel Programming: A comprehensive and practical guide to kernel internals, writing modules, and kernel synchronization*. Packt Publishing.

BPF maps. (n.d.). Linux kernel documentation. Retrieved February 22, 2026, from https://docs.kernel.org/bpf/maps.html

BPF ring buffer. (n.d.). Linux kernel documentation. Retrieved February 22, 2026, from https://docs.kernel.org/bpf/ringbuf.html

Bruening, D., Garnett, T., & Amarasinghe, S. (2003). An infrastructure for adaptive dynamic optimization. *International Symposium on Code Generation and Optimization, 2003. CGO 2003.*, 265–275. https://doi.org/10.1109/CGO.2003.1191551

The GNU C Library. (2026). sourceware.org. Retrieved March 1, 2026, from https://sourceware.org/glibc/manual/latest/html_mono/libc.html

Kernel Probes (Kprobes). (n.d.). Linux kernel documentation. Retrieved February 22, 2026, from https://www.kernel.org/doc/html/latest/trace/kprobes.html

Kerrisk, M. (2024). *Perf(1)*. man7.org. Retrieved February 22, 2026, from https://man7.org/linux/man-pages/man1/perf.1.html

Kerrisk, M. (2025a). *Epoll_wait(2)*. man7.org. Retrieved February 26, 2026, from https://man7.org/linux/man-pages/man2/epoll_wait.2.html

Kerrisk, M. (2025b). *Ld.so(8)*. man7.org. Retrieved March 1, 2026, from https://man7.org/linux/man-pages/man8/ld.so.8.html

Kerrisk, M. (2025c). *Mtrace(3)*. man7.org. Retrieved March 1, 2026, from https://man7.org/linux/man-pages/man3/mtrace.3.html

Kerrisk, M. (2025d). *Proc_pid_maps(5)*. man7.org Linux manual pages. Retrieved February 26, 2026, from https://man7.org/linux/man-pages/man5/proc_pid_maps.5.html

Kerrisk, M. (2026). *Addr2line(1)*. man7.org. Retrieved February 26, 2026, from https://man7.org/linux/man-pages/man1/addr2line.1.html

Libbpf API error handling. (n.d.). libbpf.readthedocs.io. Retrieved February 26, 2026, from https://libbpf.readthedocs.io/en/latest/api.html#libbpf-h

Linux Kernel Documentation Project. (n.d.). *eBPF verifier*. Linux kernel documentation. Retrieved February 21, 2026, from https://www.kernel.org/doc/html/latest/bpf/verifier.html

Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V. J., & Hazelwood, K. (2005). Pin: Building customized program analysis tools with dynamic instrumentation. *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 190–200. https://doi.org/10.1145/1065010.1065034

McCanne, S., & Jacobson, V. (1992). The BSD Packet Filter: A new architecture for user-level packet capture. *Proceedings of the USENIX Winter 1993 Conference*, 259–269.

Nethercote, N., & Seward, J. (2007). Valgrind: A framework for heavyweight dynamic binary instrumentation. *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 89–100. https://doi.org/10.1145/1250734.1250746

Perf: Linux profiling with performance counters. (2024). perf wiki. Retrieved February 22, 2026, from https://perfwiki.github.io/main/

Rice, L. (2023). *Learning eBPF: Programming the Linux kernel for enhanced observability, networking, and security*. O'Reilly Media.

Serebryany, K., Bruening, D., Potapenko, A., & Vyukov, D. (2012). AddressSanitizer: A fast address sanity checker. *Proceedings of the 2012 USENIX Annual Technical Conference*, 309–318. https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany

Serebryany, K., & Iskhodzhanov, T. (2009). ThreadSanitizer: Data race detection in practice. *Proceedings of the Workshop on Binary Instrumentation and Applications*, 62–71. https://doi.org/10.1145/1791194.1791203

Serebryany, K., Kennelly, C., Phillips, M., Denton, M., Elver, M., Potapenko, A., Morehouse, M., Tsyrklevich, V., Holler, C., Lettner, J., Kilzer, D., & Brandt, L. (2024). *GWP-ASan: Sampling-based detection of memory-safety bugs in production*. arXiv: 2311.09394 [cs]. https://doi.org/10.48550/arXiv.2311.09394

Seward, J., & Nethercote, N. (2005). Using Valgrind to detect undefined value errors with bit-precision. *Proceedings of the 2005 USENIX Annual Technical Conference*, 17–30.

Stepanov, E., & Serebryany, K. (2015). MemorySanitizer: Fast detector of uninitialized memory use in C++. *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 46–55. https://doi.org/10.1109/CGO.2015.7054186

TCMalloc : Thread-Caching Malloc. (2024). gperftools. Retrieved March 1, 2026, from https://www.phy.bnl.gov/~yuhw/larsoft837/src/gperftools/docs/tcmalloc.html

Uprobe-tracer: Uprobe-based Event Tracing. (n.d.). Linux kernel documentation. Retrieved February 22, 2026, from https://www.kernel.org/doc/html/latest/trace/uprobetracer.html

USDT - eBPF Docs. (2026). docs.ebpf.io. Retrieved February 22, 2026, from https://docs.ebpf.io/linux/concepts/usdt/

Using the Linux Kernel Tracepoints. (n.d.). Linux kernel documentation. Retrieved February 22, 2026, from https://www.kernel.org/doc/html/latest/trace/tracepoints.html

Vasquez, F., & Simmonds, C. (2021). *Mastering Embedded Linux Programming: Create fast and reliable embedded solutions with Linux 5.4 and the Yocto Project 3.1 (Dunfell)* (Third Edition). Packt Publishing Ltd.

Xu, D., & Chen, L. (2024). A dynamic memory leak detection method for {Linux} based on {eBPF} uprobe and LKM. *2024 4th International Conference on Computer Systems (ICCS)*, 192–197. https://doi.org/10.1109/ICCS62594.2024.10795839